**ROMÂNIA**
**MINISTERUL APĂRĂRII NAŢIONALE**
**ACADEMIA TEHNICĂ MILITARĂ „FERDINAND I”**

**FACULTATEA DE SISTEME INFORMATICE ȘI SECURITATE CIBERNETICĂ**

**Specializarea: Calculatoare și sisteme informatice pentru apărare și securitate națională**


# PLATFORMĂ EDUCATIVĂ PENTRU TESTARE CU ÎNTREBĂRI GENERATE PROCEDURAL

ABSOLVENT:
**Std. Sg. Maj. Porfirie-Denissa PILIPĂUȚANU**

CONDUCĂTOR ŞTIINŢIFIC:
**Ș.L. dr. ing. Cristian CHILIPIREA**

Conține _____ file
Inventariat sub nr. _____
Poziția din indicator: _____
Termen de păstrare: _____

**BUCUREŞTI**
**2022**

## Abstract

With an increasing number of students enrolled in programming classes, it is a well-known fact that the teachers regularly face the challenge of producing new knowledge assessment content due to the publication of existing materials and open access to them. Thus, a need to create a large set of exercises in a short time arises.

Therefore, this paper aims to present an automated evaluation system that ensures the variety of content through the procedural generation of questions and their answers. As a consequence of the fact that the project was designed in order to innovate the evaluation method within the Data Structures and Algorithms course, it is focused on the generation of programming exercises.

There are a variety of types of exercises that can be used to assess a student's programming skills. The paper addressed the exercises that assess the knowledge and application of algorithms, as well as the semantic exercises that consist in understanding the flow of a program.

# Rezumat

Formarea unui număr mare de specialiști devine o provocare pentru resursa umană limitată din mediul universitar. Profesorii se confruntă în mod regulat cu sarcina de a crea itemi pentru testare din cauza publicării materialelor existente și accesului deschis la acestea. Astfel, apare nevoia de a crea într-un timp scurt un set amplu de exerciții.

Prin urmare, această lucrare își propune să prezinte un sistem de evaluare automatizat ce asigură varietatea de conținut prin intermediul generării procedurale a întrebărilor și a răspunsurilor aferente acestora. Ca o consecință a faptului că proiectul a fost conceput cu scopul de a inova modalitatea de evaluare din cadrul cursului de Structuri de Date și Algoritmi, acesta este focusat asupra generării exercițiilor de programare.

Există o multitudine de tipuri de exerciții care pot fi folosite pentru a evalua cunoștințele de programare ale unui student. În cadrul lucrării au fost abordate exercițiile ce evaluează cunoașterea și aplicarea unor algoritmi, precum și exercițiile semantice ce constau în înțelegerea fluxului unui program.

# Cuprins

# Listă figuri

# Listă tabele

# Listă de acronime

| | |
|---|---|
| AIG | Automatic Item Generation |
| API | Application Programming Interface |
| AST | Abstract Syntax Tree |
| AQG | Automatic Question Generation |
| IEEE | Institute of Electrical and Electronics Engineers |
| DBMS | Database Management System |
| HTTP | Hypertext Transfer Protocol |
| SDT | Syntax Directed Translation |

# 1. Introducere

În procesul educațional, testarea acumulării de cunoștințe prin întrebări este una dintre metodele fundamentale de evaluare. Acestea oferă beneficii precum stimularea memoriei pentru regăsirea și sedimentarea informației, focusarea pe conceptele importante din materialul studiat și în anumite circumstanțe obținerea de feedback atunci când există o nelămurire.

Ca o consecință a dezvoltării rapide și a nevoii de personal tot mai mari în industria tehnologiei informației, formarea unui număr mare de specialiști devine o provocare pentru resursa umană limitată din mediul universitar.

Profesorii se confruntă în mod regulat cu sarcina de a crea itemi pentru testarea cunoștințelor din cauza publicării materialelor existente și accesului deschis la acestea, iar compunerea manuală a întrebărilor poate deveni un proces complex ce necesită experiență, resurse și timp. Astfel, apare nevoia de a crea într-un timp scurt un set amplu de exerciții.

Printre sarcinile comune pe care profesorii trebuie să le îndeplinească se numără și corectarea testelor sau oferirea de feedback asupra rezolvărilor propuse de către studenți.

Caracterul repetitiv al sarcinilor expuse anterior subliniază importanța rezolvării problemei și impactul puternic pe care l-ar avea o soluție automatizată, precum reducerea resurselor necesare pentru producerea unui test cât și maximizarea timpului care poate fi dedicat de către profesori pentru sarcini mult mai importante.

În momentul de față soluțiile cele mai cunoscute și utilizate de evaluare automată au reușit să degreveze cadrele didactice de partea de corectare, folosind întrebările și răspunsurile furnizate de către acestea, fiind pretabile pentru orice domeniu. Pentru a preveni plagiatul, este necesară versatilitatea și complexitatea itemilor din cadrul testelor, iar din acest punct de vedere soluțiile menționate anterior necesită o bancă mare de itemi. Există și soluții particulare care au reușit să degreveze cadrele didactice de partea de creare de itemi și generare a răspunsurilor, precum și să asigure o versatilitate a itemilor, utilizând șabloane parametrizate ale întrebărilor și algoritmi specifici care să genereze răspunsul. Acestea nu mai sunt, în schimb, pretabile oricărui domeniu, iar utilizarea unor șabloane, deși previne plagierea, nu asigură întotdeauna înțelegerea materiei, rezolvarea acestora putând fi învățată mecanic.

Se poate observa că generarea procedurală are un rol important în cadrul automatizării sarcinii de creare a itemilor. Aceasta este una din ramurile creării de conținut multimedia, cunoscută pentru producerea, manipularea și modificarea datelor prin mijloace automate și se bazează pe fuziunea dintre conținutul furnizat de om și aleatorismul utilizat în cadrul algoritmilor[1].

O constrângere în domeniul generării automate de întrebări este reprezentată de faptul că nu poate fi construită o soluție universal valabilă. Odată stabilite constrângerile particulare ale unui generator destinat unui domeniu, șabloanele parametrizate pot asigura o oarecare versatilitate a itemilor, însă nu asigură înțelegerea acestora de către student. Prin urmare, problema dificilă care trebuie rezolvată constă în furnizarea unui grad și mai mare de versatilitate.

Lucrarea de față propune o soluție la această problemă, țintind domeniul programării. Proiectul a fost conceput cu scopul de a dezvolta un sistem de testare automatizat și de a inova modalitatea de evaluare din cadrul cursului de Structuri de Date și Algoritmi. Predarea programării se confruntă cu probleme specifice, cum ar fi înțelegerea conceptelor de programare, dar și a algoritmilor pentru rezolvarea sarcinilor de programare. În cadrul cursurilor introductive, studenții încearcă să evite înțelegerea conceptelor urmând niște scurtături, cum ar fi învățarea codului prin memorare sau copierea programelor de la colegi.

Prin urmare, se profilează obiectivul principal pe care generatoarele de exerciții de programare trebuie să îl îndeplinească: furnizarea varietății necesare de exerciții pentru studenți astfel încât să le testeze abilitățile practice de-a lungul cursului și să rezolve problema plagiatului. Pentru a ajuta la rezolvarea problemei de scalabilitate a cursurilor, generatoarele de exerciții ar trebui să îndeplinească și alte obiective adiționale. O varietate de exerciții nu asigură faptul că acestea reflectă conceptele și tehnicile de programare urmărite în evaluarea cunoștințelor. Prin urmare, trebuie asigurată și îndeplinirea acestui obiectiv. De asemenea, testarea corectitudinii soluției studentului și furnizarea de feedback în mod automat reprezintă alte modalități prin care profesorul poate fi degrevat de o sarcină repetitivă și niște componente necesare în cadrul unui sistem de evaluare.

Există o multitudine de tipuri de exerciții care pot fi folosite pentru a evalua cunoștințele de programare ale unui student. În cadrul lucrării au fost abordate exercițiile ce evaluează cunoașterea și aplicarea unor algoritmi, precum și exercițiile semantice ce constau în înțelegerea fluxului unui program.

[1] https://en.wikipedia.org/wiki/Procedural_generation, accesat la 15 mai 2022

Pentru primul tip de exerciții soluția propusă constă în furnizarea tiparului întrebării, a unui fișier de configurare cu ajutorul căruia se determină termenii ce vor fi înlocuiți în cadrul acesteia cu datele generate și un fișier producător care încapsulează algoritmul vizat de către întrebare, care compilat și rulat va genera datele întrebării și rezultatul aferent.

Al doilea tip de exerciții a necesitat o soluție mai complexă deoarece o parte din întrebare constă dintr-o secvență de cod care trebuie să reflecte conceptele urmărite în evaluarea cunoștințelor. Pentru a produce acel cod a fost indispensabilă o abordare originală asupra unor concepte de design a compilatoarelor precum gramaticile sau tabela de simboluri. Spre deosebire de tipul precedent de exerciții acesta utilizează și un fișier cu gramatica folosită pentru generarea secvenței de cod. O altă diferență constă în faptul că fișierul producător este inițial incomplet și necesită inserarea secvenței de cod generate pentru a putea fi compilat și rulat.

Întrebarea și fișierele sunt introduse în cadrul platformei prin intermediul unei interfețe administrator. Ulterior, profesorul poate crea un test care să cuprindă întrebarea și să poată fi susținut de către student. O imagine mai detaliată asupra întregului sistem este furnizată în capitolele următoare.

Primul capitol prezintă succint importanţa temei, utilitatea unui astfel de sistem, precum și obiectivele lucrării ce se doresc a fi îndeplinite prin elaborarea proiectului. De asemenea, este furnizată o imagine de ansamblu asupra conținutului acestei lucrări și a modului în care aceasta este structurată alături de o prezentare sumară a rezultatelor obţinute.

Capitolul al doilea își propune să ofere prin intermediul literaturii relevante recente o perspectivă generală asupra contextului actual în care se află domeniul generării procedurale de întrebări și o analiză detaliată a soluțiilor similare și a implementărilor existente, cu un focus pe exercițiile de programare. Adițional, acesta va expune API-uri și componente software relevante.

Cel de-al treilea capitol prezintă sumar etapele parcurse de-a lungul întregului proces de compilare pentru a furniza o perspectivă de ansamblu asupra conceptelor și contextului în care au fost utilizate și a modului de funcționare al acestora. Analiza sintactică, generarea de cod intermediar și tabela de simboluri constituie pilonii procesării de text și generării de cod pentru întrebările axate pe înțelegerea fluxului programelor din prezentul proiect.

Al patrulea capitol cuprinde detaliile privind implementarea practică precum definirea cerinţelor complete pentru sistemul propus, prezentarea

cazurilor de utilizare, definirea arhitecturii și proiectarea sistemului, tehnologiile sau API-urile folosite pentru dezvoltarea acestuia, descrierea cazurilor de testare și elaborarea raportului de testare, precum și problemele întâmpinate pe parcursul întregului proces.

Capitolul cinci este dedicat prezentării rezultatelor experimentale, în urma măsurătorilor de performanță, iar ultimul capitol face o sinteză asupra principalelor idei din lucrare, furnizând direcţii și idei pentru continuarea acesteia.

# 2. Literatură relevantă recentă

Pentru a reduce costurile asociate procesului manual de generare al întrebărilor și pentru a rezolva problema unui flux constant de conținut, au fost dezvoltate tehnici de generare procedurală ale acestora.

Acest capitol își propune să furnizeze o imagine de ansamblu asupra contextului actual în care se află domeniul generării procedurale de întrebări, o analiză detaliată a soluțiilor similare și a implementărilor existente, precum și dezavantajele fiecărei propuneri, punând accent pe cele care vizează exercițiile de programare.

În ciuda faptului că volumul cercetărilor științifice în domeniul generarării procedurale de întrebări a cunoscut o creștere în ultimii ani, munca privind generarea automată de exerciții pentru programare a fost relativ limitată în comparație cu celelalte subdomenii. Au apărut mai multe propuneri pentru dezvoltarea de software educațional specializat în a ajuta studenții să înțeleagă conceptele de bază de programare și să dezvolte abilități de rezolvare a problemelor. Soluțiile propuse în prezent se bazează în mare măsură pe tipare parametrizate și nu oferă gradul dorit de versatilitate și complexitate, augmentând astfel utilitatea prezentei lucrări.

## 2.1. Metode generative pentru întrebările cu răspuns liber

Ghader Kurdi, Jared Leo, Bijan Parsia, Uli Sattler și Salam Al-Emari [1] au analizat peste 90 de lucrări științifice legate de generarea întrebărilor în domeniul educațional, furnizând o imagine de ansamblu asupra comunității AQG, direcțiile curente și progresele realizate. Acesta prezintă etapele procesului generator și metodele generative pentru diferite tipuri de întrebări printre care se numără și întrebările cu răspuns liber vizate în cadrul proiectului, precum și modalitățile prin care se poate face evaluarea calității itemilor.

În domeniul evaluării automate există mai multe categorii de întrebări care pot fi utilizate în cadrul testelor. Exercițiile care pot fi folosite cu precădere în domeniul programării pot fi împărțite în trei categorii de bază: întrebări cu răspunsuri multiple, întrebări cu răspuns liber și sarcini de programare. Simplitatea întrebărilor cu alegere multiplă le-a făcut foarte populare în sistemele de management al învățării, cum ar fi Moodle. Cu toate acestea, în predarea programării, întrebările cu răspunsuri multiple pot fi utile doar pentru

adoptarea unor elemente teoretice de bază, nu și pentru dobândirea de abilități practice în rezolvarea sarcinilor de programare. De aceea, proiectul curent este focusat exclusiv pe întrebările cu răspuns liber.

Alcătuirea întrebării (setul întrebare-răspuns corect) este principala sarcină a întregului proces de generare. În funcție de tipul acesteia și de formatul răspunsului sunt implicate diverse abordări. În cadrul clasificării metodelor de generare, metoda folosită în mod uzual pentru producerea întrebărilor cu răspuns liber este cea bazată pe șabloane ce constă în folosirea unei structuri de bază, ale cărei goluri sunt customizate cu valori care întrunesc caracteristicile sintactice sau semantice precizate. Șabloanele necesită intervenția umană prin construirea manuală a acestora, ceea ce face ca din perspectiva costului, această metodă să fie considerată scumpă. Un alt dezavantaj al folosirii acestei metode este dat de structură limitată a întrebărilor generate din punct de vedere al diversității lingvistice.

Din perspectiva costului nu a putut fi identificată o alternativă mai bună pentru generarea întrebărilor cu răspuns liber, din cauză că și restul metodelor prezentate au în acest moment un cost ridicat (ex. necesitatea unei cantități mari de date etichetate corespunzător). Metodele alternative nu au însă dezavantajul structurii rigide a întrebării. Deși acestea produc întrebări mult mai diversificate din punctul de vedere al topicii propoziției și al limbajului, nu sunt potrivite exercițiilor ce urmăresc să reflecte anumite concepte și tehnici de programare. Se justifică, astfel, folosirea uzuală a metodei șabloanelor în acest domeniu.

Ca practică generală, în urma generării, se realizează evaluarea calității întrebării. Criteriile de evaluare se orientează asupra calității lingvistice și educaționale. Din punct de vedere lingvistic se urmărește corectitudinea gramaticală, fluența, ambiguitatea semantică și lipsa erorilor. Din punct de vedere educațional se urmărește utilitatea întrebării, relevanța în cadrul domeniului și obiectivul didactic atins de subiectul evaluat prin răspunsul la acea întrebare. De asemenea, mai sunt și alte metrici relevante pentru calitate precum dificultatea sau nivelul cognitiv țintă. Determinarea dificultății întrebării se poate face prin determinarea numărului de persoane evaluate care au răspuns corect din totalul celor examinați, compararea dificultății prezise cu performanțele obținute de studenți sau prin utilizarea unor mecanisme de rezolvare automată.

Ținând cont de observațiile făcute de către autori pe baza analizei compilației de lucrări științifice, implementarea acestui proiect folosește la bază ca metodă de generare șabloanele. Prin utilizarea acestei metode, întrebările

întrunesc criteriile de calitate menționate anterior deoarece șablonul trece prin filtrul unei persoane cu experiența necesară a-l întocmi.

## 2.2. Soluții ce utilizează șabloane parametrizate

Generatorul de exerciții de programare implementat în Python de Danijel Radošević, Tihomir Orehovački și Zlatko Stapić [2], se bazează pe o listă de specificații, un fișier de configurare și tipare de cod în limbajul C++.

Lista de specificații conține elemente de tipul atribut:valoare, fiind constituită inițial din numele fișierului de output. Aceasta este ulterior modificată prin adăugarea mai multor elemente alese aleator din liste predefinite ce conțin nume de fișiere șablon sau diverse denumiri pentru identificatorii variabilelor. Lista finală de specificații conține atribute precum fișierul de output, numele studentului, ID-ul studentului, tiparul principal și alți parametrii specifici exercițiului.

Autorii lucrării menționează că au definit mai multe tipuri de tipare: un tipar cu codul principal al programului și cu exercițiile sub formă de comentarii și tipare cu dimensiuni mai mici ce conțin secvențe de cod. Exemplul furnizat de aceștia arată o posibilă structură pentru tiparul principal.

În cadrul procesului generativ, porțiunile delimitate de caracterul „#” din tipare sunt înlocuite prin intermediul valorilor asociate din fișierul de configurare, care referențiază fie un atribut din lista de specificații fie numele unui fișier cu tiparul de dimensiune mai mică. Înlocuirea se face, astfel, fie prin obținerea directă a valorii atributului din lista de specificații fie prin generarea tiparului menționat.

O problemă a acestei soluții sunt cerințele relativ mari pentru profesori, din cauză că fiecare exercițiu necesită un set particular de tipare. Deși acestea ar putea fi moștenite parțial de la exercițiile anterioare, iar configurația generatorului ar putea fi, de asemenea, moștenită parțial, întregul proces necesită un control granular asupra mulțimii de fișiere, ceea ce nu este fezabil pentru o diversitate de tipuri de exerciții. În implementarea proiectului, această problemă a multitudinii de tipare ce trebuie întocmite, a fost rezolvată prin generarea unei diversități de secvențe de cod prin definirea unei gramatici.

O altă problemă este lipsa furnizării unei soluții automate de notare a exercițiilor și metoda de furnizare a acestora prin comentarii. Acest aspect necesită timp suplimentar din partea profesorilor și ar putea fi îmbunătățit. Prin

urmare, lucrarea de față a definit în procesul generator, un mecanism de producere a răspunsurilor, pentru ca ulterior soluția studentului să poată fi corectată în mod automat.

Akiyoshi Wakatani și Toshiyuki Maeda [3] au pornit de la premiza că la baza învățării unui limbaj de programare se află atât dobândirea cunoștințelor de gramatică a limbajului cât și capabilitatea de a înțelege fluxul unui program. Prin urmare, aceștia au pregătit două tipuri de exerciții: exerciții de sintaxă prin care cursanții să corecteze erorile programelor furnizate, și exerciții de semantică prin care cursanții să estimeze rezultatul programelor, fără a le executa. Acestea pot fi accesate prin intermediul unei aplicații web în limbajul PHP, care le generează automat folosind șabloane în limbajul C. Exercițiile generate sunt evaluate automat.

Pentru primul tip de exerciții, interfața grafică permite vizualizarea unui program cu erori și a descrierilor erorilor furnizate de compilator. Prin intermediul acesteia, cursanții corectează programul utilizând cunoștințele despre gramatica limbajului și mesajele de eroare până când toate erorile sunt complet eliminate.

Exercițiile sunt generate prin adăugarea de bug-uri unor programe șablon și prin înlocuirea numelor de variabile. Au fost definite trei metode de adăugare a erorilor la programul șablon. În primul rând, numele variabilelor sunt înlocuite cu valori alese aleator dintr-o listă predefinită de nume. Astfel, pot apărea în program variabile nedeclarate. A doua metodă se bazează pe scrierea greșită a unor șiruri de caractere definite în prealabil (de exemplu, stdio.h poate fi schimbat în stdi.h). A treia metodă înlocuiește un șir de caractere cu un alt cuvânt desemnat, pentru a genera o eroare întâlnită des (de exemplu, înlocuirea „;” cu „:”).

După ce un cursant corectează un program care are erori, scriptul PHP salvează programul corectat ca fișier cu un nume unic, îl compilează și trimite înapoi mesajul de la compilator către cursant. Șirul de caractere „OK” este trimis înapoi, dacă compilatorul nu produce niciun mesaj.

Autorii menționează că programele șablon sunt pregătite în conformitate cu progresul învățării și sunt necesare între 10 și 20 programe șablon pentru a produce o varietate de programe.

Al doilea tip de exerciții, cele de estimare a output-ului unui program, au scopul de a îmbunătăți capacitatea de înțelegere a programelor din punct de vedere semantic. Interfața grafică a aplicației web permite vizualizarea unui program care conține diverse instrucțiuni. Cursantul citește programul și

estimează rezultatul funcției prin înțelegerea semanticii acestuia. După scrierea valorii estimate în zona de răspuns și apăsarea butonului de verificare, este afișată comparația răspunsului corect cu estimarea.

Autorii menționează că programele date pot include apeluri recursive de funcții și calcule complicate, ceea ce face mai dificil de estimat rezultatul. Pentru a crea programe corecte, este indispensabilă înțelegerea comportamentului programelor care includ instrucțiuni condiționale, bucle și apeluri de funcții. Prin urmare, aceștia susțin că aceste exerciții sunt utile pentru a îmbunătăți capacitatea cursanților de a înțelege programe. Prin schimbarea adecvată a numelor variabilelor și a valorilor constantelor și menținând lungimea programului suficient de compactă pentru estimare, cursanții pot efectua exerciții în mod repetat pentru a spori eficacitatea educației. Parametrii constanți și modelele de calcul trebuie să fie date corect, deoarece parametrii greșiți pot provoca excepții precum împărțirea la zero sau erori de acces la memorie.

Justificarea obiectivelor didactice atinse de către exercițiile menționate, le fac pretabile pentru a fi produse de către generatorul implementat în cadrul proiectului. Ținând cont de faptul că primul tip vizează noțiuni de bază introductive și nu reprezintă o provocare pentru studenții acomodați cu un limbaj de programare, a fost ales doar al doilea tip de exerciții. Pentru acesta, soluția curentă furnizează o gamă largă de secvențe de cod unice fără a mai fi necesară definirea a 10-20 de tipare.

## 2.3. Soluții ce utilizează noduri de instrucțiuni

Deși întrebările parametrizate pot fi utile pentru a preveni plagiatul, ele necesită destul de mult timp din partea cadrelor didactice pentru crearea tiparelor. Un potențial răspuns la această problemă este furnizat de Thomas James Tiam-Lee și Kaoru Sumi [4] care prezintă o abordare pentru generarea procedurală a exercițiilor de programare fără a utiliza șabloane parametrizate. Pentru a reprezenta exercițiile într-o manieră structurată, fiecare exercițiu este tratat ca o succesiune de instrucțiuni pe care elevul trebuie să le efectueze. Fiecare instrucțiune este reprezentată de un singur nod, care poate fi asociat cu o operație, o condiție sau returnarea unei valori. Prin urmare, exercițiul este o colecție de noduri, care va fi parcursă succesiv.

Un nod de operație, se referă la o singură operație aritmetică și este urmat întotdeauna de un singur nod. Un nod de condiție, se referă la o singură expresie

condițională și este întotdeauna urmat de două noduri, unul pentru când condiția este adevărată și unul pentru când condiția este falsă. Un nod de returnare, se referă la un punct final al funcției și nu este urmat de un alt nod. Operații de nivel superior pot fi adăugate prin abstractizarea lor ca blocuri de operații sau de condiții. Blocurile constau într-o secvență internă de operații elementare.

Pornind de la o valoare definită a complexității codului dorit, se generează o structură aleatorie. Complexitatea unei structuri este o metrică definită ca numărul de noduri asociate cu o operație sau cu o condiție. Această definiție se bazează pe ideea că un exercițiu devine mai complex pe măsură ce crește numărul de instrucțiuni necesare pentru a-l finaliza.

Primul nod este inițializat fie ca nod de operație fie ca nod de condiție. Apoi, într-unul din locurile disponibile din cadrul structurii este atașat aleatoriu, în mod asemănător, fie un nod de operare, fie un nod de condiție. Această operațiune este efectuată până când structura conține un număr de noduri egal cu complexitatea. În locurile libere rămase, sunt atașate nodurile de returnare.

După definirea structurii exercițiului, parametrii fiecărui nod sunt asignați utilizând un algoritm ce a luat în calcul considerațiile detaliate în cadrul lucrării. Odată construită secvența de instrucțiuni, fiecare nod este parcurs și în funcție de tipul acestuia este generat atât un text în limbaj natural pentru crearea exercițiului, cât și codul Java asociat pentru a evalua soluția elevului. Pentru crearea cerinței, există mai multe texte predefinite pentru fiecare tip de nod, iar din acestea este ales unul, în mod aleator. Ulterior, porțiunile marcate goale sunt înlocuite cu variabile și parametrii din structura nodului.

În acest studiu au fost folosite doar operații simple de calcul pentru producerea exercițiilor de programare precum operațiile aritmetice (de exemplu, adunarea, înmulțirea), operațiile condiționale (de exemplu, mai mare, mai mic) și câteva blocuri. De asemenea, tipurile de date sunt limitate doar la numere întregi. Din acest motiv, exercițiile generate, în ciuda faptului că nu sunt derivate dintr-un șablon parametrizat, arată în continuare similare între ele.

```c
int theFunction(int I1) {
  int X1;
  X1 = I1 + 2;
  if (X1 > 10) {return I1;}
  else {return 10;}
}


int theFunction(int I1) {
  int X1;
  if (I1 >= 2) {return I1;}
  else {
    X1 = I2 * 3;
    return X1;
  }
}


int theFunction(int I1) {
  int X1, X2, N1, N2;
  X1 = I1 + 5;
  N1 = X1 + 8;
  X2 = N1 / 2;
  N2 = X2 % 2;
  if (N2 == 0) {return 1;}
  else {return 2;}
}
```

*Figură 2-1 - Secvențe de cod generate de James Tiam-Lee și Kaoru Sumi*

```c
void generate_output() {
    nod *p = vf;
    while (p->prec) {
        p->n += 2;
        if (p->n % 3 == 1)
            p->n *= 2;
        p = p->prec;
    }
    nod *u = new nod;
    u->n = 6;
    u->prec = vf;
    vf = u;
}
```

```c
void generate_output() {
    nod *q = vf;
    while (q->prec) {
        q->n += 2;
        if (q->n != 5)
            q->n *= 2;
        q = q->prec;
    }
}
```

*Figură 2-2 - Secvențe de cod generate în cadrul acestui proiect*

Prin urmare, soluția propusă în lucrarea de față și-a propus și a reușit să obțină secvențe de cod mai complexe care conțin și bucle și care nu se limitează doar la un anumit tip de date. De asemenea, ideea că un exercițiu devine mai complex pe măsură ce crește numărul de instrucțiuni necesare pentru a-l finaliza a contribuit la definirea metricii asociate complexității folosite în cadrul proiectului.

## 2.4. Soluții ce utilizează gramatici (biblioteca Tracery.js)

Generarea procedurală este utilizată într-un spectru larg de domenii, însă cel mai popular dintre acestea este cel al jocurilor, unde texturile, efectele sonore și povestea sunt produse cu ajutorul gramaticilor.

O gramatică este constituită dintr-un set de simboluri terminale (simbolurile elementare ale limbajului definit de gramatică), un set de neterminale (un neterminal reprezintă un șir de terminale), un set de reguli de producție (fiecare regulă de producție constă dintr-un neterminal, o săgeată și o secvență de terminale și/sau neterminale) și un simbol de start desemnat din rândul neterminalelor. Aceasta definește sintaxa unui limbaj.

Această metodă de generare nu este comun folosită în domeniul educațional. Pornind de la faptul că întrebările sunt de fapt un text, iar gramaticile pot produce o gamă diversificată de texte utilizând un set de reguli de producție, este foarte probabil ca această metodă să se preteze domeniului. De asemenea, pentru o particularizare a întrebărilor de natură generală la întrebări ce vizează exerciții de programare, această abordare e o opțiune la fel de viabilă. Astfel, utilizând această euristică, lucrarea de față și-a propus să obțină întrebări cu un grad mare de versatilitate prin intermediul gramaticilor.

Tracery [5] este un instrument generativ de text conceput cu scopul de a fi folosit de către autori începători și experți sau de către persoane care nu se pot autoidentifica drept „programatori” pentru crearea unor combinații surprinzătoare de text în mod algoritmic. În practică, această bibliotecă a fost utilizată pentru crearea unei game largi de povești, poezii, dialoguri și chiar imagini în format SVG sau pagini HTML cu gramatici profund imbricate.

Utilizarea șabloanelor și gramaticilor pentru a crea text structurat, dar variabil în același timp reprezintă o tehnică comună în generarea de povești. Generatorul primește la intrare o gramatică formală scrisă ca obiect JSON, ce conține simbolurile și regulile de expandare asociate acestora. Aceasta va trece printr-un sistem modular de părți interfuncționale: un parsator și un motor de expandare. Prin înlocuirea treptată a simbolurilor cu una din regulile aferente acestora aleasă în mod aleator, se asigură variabilitatea în timp și se ajunge la forma finală a textului. Sintaxa limbajului Tracery este simplă și lizibilă, permițând utilizatorului să definească o gramatică în limbaj natural. Pentru a semnala simbolurile care vor fi extinse recursiv sunt folosite hashtag-uri.

Această tehnică a fost îmbunătățită cu ajutorul unui mecanism care generează text într-un mod mai riguros prin utilizarea unor funcții care pot fi aplicate după ce un simbol este expandat. Aceste funcții sunt numite modificatori. Astfel, au fost rezolvate problemele generative comune precum pluralizarea, conjugarea sau scrierea cu literă mare.

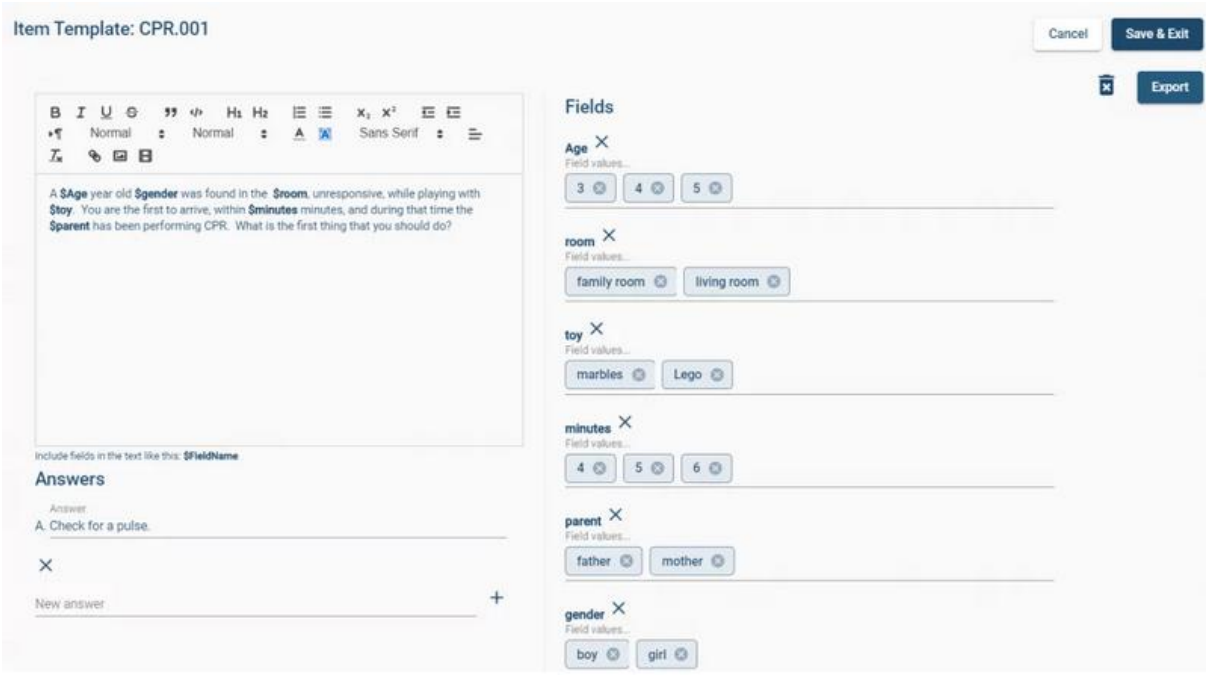
Cu această structură de date simplistă, autorii pot produce un text structurat și interesant din prisma faptului că poate fi lipsit pe alocuri de logică, însă în cazul exercițiilor de programare, acest lucru nu este de dorit. Deși biblioteca prezentată se pretează generării de text, nu permite un control mai fin al procesului generator pentru a putea produce un cod fără erori. Prin urmare, proiectul a implementat un generator bazat pe conceptele menționate anterior, adăugând un set de filtre care să verifice dacă o instrucțiune are sens în raport cu instrucțiunile generate anterior.

## 2.5. Soluții comerciale similare (ASC, Brio)

Există mai multe instrumente folosite în procesul de evaluare, variind de la instrumente folosite doar de anumite universități până la instrumente care au fost dezvoltate pentru uz comercial.

Platforma de testare ASC [6] utilizează generarea automată de itemi (AIG), încercând să automatizeze o parte din efortul implicat în crearea articolelor, din cauză că acesta este unul dintre aspectele cele mai consumatoare de timp ale dezvoltării unui test.

Tehnologia curentă de generare automată a unei game largi de itemi se bazează pe șabloane. Utilizatorul marchează prin intermediul simbolului $ porțiunile din text care sunt inserabile și definește pentru fiecare porțiune niște valori ce vor fi folosite în momentul înlocuirii. Algoritmul va produce toate permutările posibile ale șablonului. Itemii rezultați nu sunt mutați automat în banca de itemi, ci sunt revizuiți de persoane autorizate, din motive precum crearea de scenarii puțin probabile sau irelevante. Cu toate acestea, eficiența generală în procesul de creare al întrebărilor s-a îmbunătățit considerabil, iar itemii generați sunt de o calitate mai bună decât itemii scriși în mod tradițional, deoarece procesul i-a făcut pe autori să se gândească mai profund la ceea ce evaluează și cum. Platforma nu recomandă utilizarea mai multor instanțe ale aceluiași tip de item generat procedural în cadrul unui formular de testare.




*Figură 2-3 - Interfața de creare a unor itemi ce vor fi generați automat în cadrul platformei ASC*

După cum se poate observa în Figura 2-3, un dezavantaj al platformei este faptul că răspunsurile trebuie furnizate de către autorul întrebării, problemă care a fost adresată în cadrul proiectului curent. De asemenea, utilizarea unor șabloane parametrizate care au o structură rigidă, nu are avantajele utilizării unei gramatici ce are atât o structură rigidă prin regulile ei, dar și variabilitate prin modul aleator de alegere al regulilor.

Platforma de testare Brio [7] furnizează elevilor din ciclul primar (I-IV) teste de antrenament bazate pe generarea programatică, dinamică și randomizată de itemi. Această platformă deține un număr foarte mare de itemi calibrați prin testarea pe eșantioane reprezentative de elevi români, garantând acoperirea minuțioasă a întregii programe școlare anuale, în cadrul fiecărui test.


*Figură 2-4 - Itemii generați automat pentru un test în cadrul platformei Brio*

În ciuda faptului că platforma Brio generează automat atât itemii cât și răspunsurile, tiparele întrebărilor sunt destul de simpliste. Deși nu este precizată explicit, metoda utilizată pentru generare pare a fi bazată pe șabloane. În acest caz, o gramatică ar permite obținerea unor soluții mai versatile din punctul de vedere al cerinței. De asemenea, în loc să fie construite o mulțime de tipare, o gramatică permite generarea mai multor tipare prin simpla definire a ei.

# 3. Noțiuni generale de funcționare a unui compilator

Modelele, teoria și algoritmii asociați unui compilator, explicate în detaliu de Alfred Aho, Monica Lam, Ravi Sethi, și Jeffrey Ullman [8], pot fi aplicate unei game largi de probleme în design-ul și dezvoltarea de software. Prin urmare, prezentarea sumară a etapelor parcurse de-a lungul întregului proces de compilare poate furniza o perspectivă de ansamblu asupra conceptelor și contextului în care au fost utilizate și a modului de funcționare al acestora. Analiza sintactică, generarea de cod intermediar și tabela de simboluri constituie pilonii procesării de text și generării de cod pentru întrebările axate pe înțelegerea fluxului programelor din prezentul proiect.

## 3.1. Etapele procesului de compilare

**Faza de analiză** a unui compilator constă în descompunerea codului sursă în părțile constitutive ale limbajului de programare, impunerea unei structuri gramaticale și crearea unei reprezentări intermediare asociate codului. O structură gramaticală implică prezența unor reguli de sintaxă specifice limbajului. **Sintaxa** unui limbaj de programare descrie forma adecvată a programelor sale, în timp ce **semantica** limbajului definește sensul acestora, mai precis, ce face fiecare program atunci când este executat. Pentru specificarea sintaxei se folosește o notație utilizată pe scară largă, numită gramatică independentă de context[2].

Sarcina principală a **analizatorului lexical** este să citească caracterele codului sursă, să le grupeze în lexeme (șiruri de caractere) și să producă o secvență de tokeni formați din tipul lexemului și opțional, un pointer către intrarea din tabela de simboluri. Atunci când analizatorul lexical descoperă un lexem care constituie un identificator (nume de constante, variabile sau funcții), introduce acel lexem în tabelă. Tokenii sunt trimiși către parsator pentru analiza de sintaxă[3].

**Parsatorul** obține un șir de tokeni de la analizatorul lexical și verifică faptul că șirul de tokeni poate fi generat de gramatica limbajului. Parsatorul va raporta orice erori de sintaxă într-un mod inteligibil. Pentru programele bine

[2] Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, Boston, USA, 2006, p. 40
[3] Ibidem, p. 109

formate, analizatorul construiește un arbore de parsare și îl transmite restului compilatorului pentru procesare ulterioară.

Fazele ulterioare ale compilatorului trebuie să analizeze rezultatul parsatorului, pentru a asigura conformitatea programului cu regulile care nu sunt verificate de parsator[4]. O gramatică nu distinge între identificatorii care sunt șiruri de caractere diferite. În schimb, toți identificatorii sunt reprezentați printr-un simbol precum id în gramatică. Într-un compilator pentru un astfel de limbaj, faza de **analiză semantică** verifică dacă identificatorii sunt declarați înainte de a fi utilizați[5].

Dacă partea de analiză detectează pe parcursul **verificării statice** (verificare făcută de compilator) incorectitudinea codului sursă din punct de vedere sintactic sau semantic (nu sunt urmate regulile sintactice și semantice ale limbajului în care a fost scris), atunci trebuie să furnizeze mesaje informative, astfel încât utilizatorul să poată lua măsuri corective.

**Verificarea sintactică** nu verifică doar dacă programul respectă gramatica limbajului. De exemplu, constrângerile precum declararea unui identificator cel mult o dată într-un domeniu de valabilitate, sau încadrarea obligatorie a instrucțiunii break într-o buclă sunt de natură sintactică, deși nu sunt impuse de o gramatică utilizată pentru parsare.

**Verificările de tip** ale unui limbaj asigură că operatorii sau funcțiile sunt aplicate numărului și tipului corect de operanzi. Dacă este necesară conversia între tipuri (de exemplu, atunci când un număr de tip întreg este adăugat la un număr de tip float), atunci verificatorul de tip poate insera un operator în arborele de sintaxă pentru a reprezenta acea conversie[6]. În detectarea erorilor, au un rol esențial atât gramatica limbajului cât și tabela de simboluri[7].

Pe parcursul etapei de analiză sunt colectate și stocate informații despre programul sursă într-o structură de date numită **tabelă de simboluri**, care este transmisă împreună cu reprezentarea intermediară către etapa de sinteză[8].

**Faza de sinteză** translatează programul intermediar în programul țintă, utilizând informațiile din tabela de simboluri.

Partea de analiză este adesea numită partea de frontend a compilatorului, iar partea de sinteză este numită partea de backend.

[4] Ibidem, p. 209
[5] Ibidem, p. 216
[6] Ibidem, p. 97
[7] Ibidem, p. 92
[8] Ibidem, p. 4

## 3.2. Analiza sintactică

Implicit, prin design, fiecare limbaj de programare are reguli precise care descriu structura sintactică a programelor corecte (de exemplu, în C, un program este format din funcții, o funcție din declarații și instrucțiuni, o instrucțiune din expresii și așa mai departe)[9]. Structură ierarhică a elementelor constitutive ale unui limbaj de programare este descrisă de o gramatică. De exemplu, o instrucțiune if-else poate avea forma din Figura 3-1.

```
if (expresie)
    instrucțiune
else
    instrucțiune
```

*Figură 3-1 - Structura unei instrucțiuni if-else*

Această instrucțiune este reprezentată de concatenarea cuvântului cheie if, o paranteză deschisă, o expresie, o paranteză închisă, o instrucțiune, cuvântul cheie else și o altă instrucțiune. Folosind variabila expr pentru a desemna o expresie și variabila stmt pentru a desemna o instrucțiune, această regulă de poate fi exprimată sub forma din Figura 3-2.

```
stmt -> if ( expr ) stmt else stmt
```

*Figură 3-2 - Regula de producție pentru o instrucțiune if-else*

O astfel de regulă se numește o regulă de producție. Într-o regulă de producție, elementele lexicale precum cuvântul cheie if și parantezele sunt numite terminale. Variabilele precum expr și stmt sunt numite neterminale[10].

O **gramatică independentă de context** are patru componente:

1. Un set de simboluri terminale, reprezentate de simbolurile elementare ale limbajului definit de gramatică;

2. Un set de neterminale, unde un neterminal reprezintă un șir de terminale;

3. Un set de reguli de producție, unde fiecare regulă de producție constă dintr-un neterminal (partea stângă a regulii de producție), o săgeată și o secvență de terminale și/sau neterminale (partea dreaptă a regulii de producție);

[9] Ibidem, p. 191
[10] Ibidem, p. 42

4. Un simbol de start desemnat din rândul neterminalelor.

O specificare a unei gramatici se face prin enumerarea regulilor de producție, începând cu regula de producție a simbolului de start. Cifrele, semnele (de exemplu, <, >, =) și cuvintele cheie (de exemplu, while) sunt terminale. Regulile de producție asociate aceluiași neterminal pot fi grupate prin simbolul |, cunoscut în mod convențional drept „sau”[11].

```
program  ->  block
   block  ->  { decls stmts }
   decls  ->  decls decl | decl | ε
    decl  ->  type id ;
   stmts  ->  stmts stmt | stmt
    stmt  ->  flowstmt | statement ;
flowstmt  ->  if (expr) stmt
           | if (expr) stmt else stmt
           | while (expr) block
           | do block while (expr);
```

*Figură 3-3 - Gramatica unui limbaj de programare primitiv ce generează un set limitat de instrucțiuni*

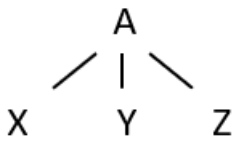
**Derivarea** unor șiruri de caractere pe baza unei gramatici se face începând cu simbolul de start și înlocuind în mod repetat un neterminal din regula de producție a simbolului cu corpul unei reguli de producție asociate acelui neterminal (un neterminal poate avea mai multe reguli de producție asociate). Șirurile terminale care pot fi derivate din simbolul de start formează limbajul definit de gramatică[12].

**Parsarea (analiza sintactică)** este definită de aflarea modului în care un șir de terminale este derivat din simbolul de start al gramaticii, iar în cazul în care șirul de terminale nu poate fi derivat din simbolul de start, de raportarea erorilor de sintaxă.

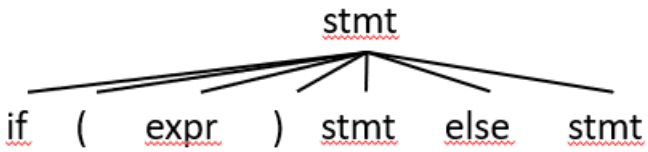
Un **arbore de parsare (arbore de sintaxă abstract)** este o imagine a unei derivări, aratând cum simbolul de start al unei gramatici derivă un șir în limbajul definit de gramatică. Fiecare nod dintr-un arbore de sintaxă reprezintă un element constitutiv al limbajului[13].

[11] Ibidem, p. 43
[12] Ibidem, p. 44
[13] Ibidem, p. 353

Dacă neterminalul A are asociată o regulă de producție A -> XYZ, arborele de parsare are un nod interior etichetat cu A cu trei copii etichetați X, Y și Z, de la stânga la dreapta ca în Figura 3-4.




*Figură 3-4 - Arborele de sintaxă al neterminalului A*




*Figură 3-5 - Arborele de sintaxă al unei instrucțiuni if-else*

Având în vedere o gramatică independentă de context, un arbore de parsare este un arbore cu următoarele proprietăți:

1. Rădăcina este simbolul de start;

2. Fiecare frunză este un terminal;

3. Fiecare nod interior este un neterminal[14].

Construcția de sus în jos a unui arbore de parsare ca cel din Figura 3-5, se face pornind de la rădăcină, și efectuând în mod repetat următorii doi pași:

1. Pentru nodul N se va selecta una dintre regulile de producție asociate și vor fi construiți copiii pentru simbolurile din corpul regulii de producție;

2. Se va găsi următorul nod la care urmează să fie construit un subarbore (de obicei, cel mai din stânga neterminal neexpandat al arborelui)[15].

[14] Ibidem, p. 45
[15] Ibidem, p. 61

## 3.3. Generarea de cod intermediar

Ideea de a asocia diverse proprietăți cu părțile constitutive ale unui limbaj de programare (de exemplu, valori și tipuri cu expresii) poate fi exprimată în cadrul gramaticilor. Neterminalele și terminalele pot avea asociate **atribute sintetizate**, iar regulile de producție pot avea asociate **reguli semantice** care descriu modul în care atributele sunt calculate la nivelul nodurilor arborelui de parsare unde acea regulă de producție este folosită.

Reprezentarea intermediară a codului se realizează cu ajutorul arborilor de sintaxă abstractă ce modelează structura sintactică ierarhică a codului sursă[16]. Odată ce arborele de sintaxă este construit, codul intermediar poate fi generat prin evaluarea atributelor și executarea fragmentelor de cod asociate nodurilor din arbore[17].

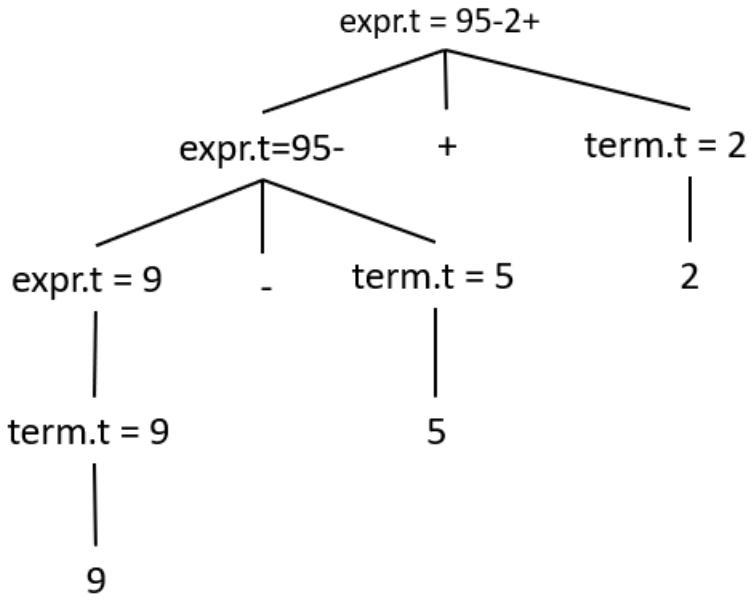
Se spune că un atribut este sintetizat dacă valoarea lui la un nod al arborelui analizat N este determinată de valorile atributelor copiilor lui N și N însuși. Atributele sintetizate au proprietatea că pot fi evaluate în timpul unei singure traversări de jos în sus a unui arbore de parsare[18].

Fragmentele de cod aferente regulilor semantice sunt executate atunci când regula de producție este utilizată în timpul analizei de sintaxă și realizează **translatarea direcționată de sintaxă**[19]. Procesul de evaluare a valorii atributului, se numește **definiție direcționată de sintaxă**.

Pentru un șir de intrare dat, este construit arborele de parsare, iar apoi sunt aplicate regulile semantice pentru a evalua atributele fiecărui nod din arbore.

De exemplu, Figura 3-6 prezintă un arbore de parsare adnotat pentru 9-5+2. Valoarea 95-2+ a atributului de la rădăcină este notația postfixă pentru 9-5+2.

[16] Ibidem, p. 41
[17] Ibidem, p. 99
[18] Ibidem, p. 54
[19] Ibidem, p. 53




*Figură 3-6 - Valorile atributelor nodurilor arborelui de sintaxă unei expresii*

Fiecare neterminal (expr, term) are un atribut t sub forma unui șir de caractere care reprezintă notația postfixă pentru expresia generată de acel neterminal. Simbolul || din regula semantică este operatorul utilizat în mod convențional pentru concatenarea șirurilor.

*Tabel 3-1 - Definiția direcționată de sintaxă a formei postfixe a unei expresii*

| REGULA DE PRODUCȚIE | REGULA SEMANTICĂ |
|---|---|
| expr -> expr1 + term | expr.t = expr1.t \|\| term.t \|\| ’+’ |
| expr -> expr1 - term | expr.t = expr1.t \|\| term.t \|\| ’-’ |
| expr -> term | expr.t = term.t |
| term -> 0 | term.t = ’0’ |
| term -> 1 | term.t = ’1’ |
| ... | ... |
| term -> 9 | term.t = ’9’ |

Forma postfixă a unei cifre este cifra însăși. De exemplu, regula semantică asociată cu regula de producție term -> 9 face ca term->t să fie 9 ori de câte ori această regulă de producție este utilizată la nivelul unui nod din arborele de parsare. Celelalte cifre sunt translatate similar. Când regula de producție expr -> $expr_1$ + term este aplicată, valoarea term->t preia valoarea expr->t[20].

Definiția direcționată de sintaxă din exemplul precedent are următoarea proprietate importantă: șirul care reprezintă translația neterminalului de la

[20] Ibidem, p. 55

începutul fiecărei reguli de producție este concatenarea translațiilor neterminalelor din corpul regulii de producție, în aceeași ordine ca și în regulă. O definiție direcționată de sintaxă cu această proprietate este denumită simplă.

Translatarea expr->t este concatenarea translatărilor lui $expr_1$ și $term_1$, urmată de simbolul +. expr1 și term apar în aceeași ordine atât în corpul regulii de producție, cât și în regula semantică. Nu există simboluri suplimentare înainte sau între translatări. În acest exemplu, singurul simbol suplimentar apare la sfârșit[21].

Parcurgerile de arbori sunt folosite pentru specificarea execuției fragmentelor de cod în cadrul translatării și evaluarea atributelor. O traversare a unui arbore începe de la rădăcină și vizitează fiecare nod al arborelui într-o anumită ordine. Procedura vizitează(N) din Figura 3-7 reprezintă o traversare în adâncime care vizitează copiii unui nod de la stânga la dreapta. Pe parcursul traversării fiecărui nod, este evaluat la final atributul (după ce cu siguranță au fost evaluate atributele copiilor).

```
vizitează(nod N) {
    pentru fiecare copil C al nodului, de la stânga la dreapta
        vizitează(C);
    evaluează regulile semantice de la nivelul nodului N;
}
```

*Figură 3-7 - Pseudocodul unei traversări în adâncime a unui arbore de sintaxă și al evaluării atributelor nodurilor*

O definiție direcționată de sintaxă nu impune nicio ordine specifică pentru evaluarea atributelor dintr-un arbore de parsare. Orice ordine de evaluare care calculează un atribut după toate celelalte atribute de care depinde este acceptată. Atributele sintetizate pot fi evaluate în timpul oricărei traversări de jos în sus, adică traversarea care evaluează atributele unui nod după ce au fost evaluate atributele copiilor săi[22].

Un element constitutiv poate avea cel mult un atribut asociat. În cazul identificatorilor (numelor de constante, variabile sau funcții) valoarea atributului este un pointer către intrarea din tabela de simboluri aferentă acestuia care permite accesul la mai multe informații[23].

[21] Ibidem, p. 56
[22] Ibidem, p. 57
[23] Ibidem, p. 112

## 3.4. Tabela de simboluri

Tabelele de simboluri sunt structuri de date care sunt utilizate de compilatoare pentru a păstra informații despre identificatori (nume de constante, variabile și funcții). Informațiile sunt colectate în mod incremental de fazele de analiză ale unui compilator și utilizate de fazele de sinteză pentru a genera codul țintă. Intrările din tabela de simboluri conțin informații despre identificatori, cum ar fi șirul de caractere asociat(lexemul), tipul lor, adresa lor din memorie și orice alte informații relevante. În cazul procedurilor, sunt stocate informații precum numărul și tipurile argumentelor, metoda de transmitere a fiecărui argument (de exemplu, prin valoare sau prin referință) și tipul returnat[24].

Tabelele de simboluri trebuie de obicei să accepte mai multe declarații ale aceluiași identificator în cadrul unui program (din domenii de valabilitate diferite)[25].

În consecință, rolul unei tabele de simboluri este de a transmite informații de la declarațiile variabilelor către utilizarea acestora. O acțiune semantică pune informații despre identificator în tabela de simboluri, atunci când este analizată declarația acestuia. Ulterior, o acțiune semantică asociată cu o regulă de producție obține informații despre identificator din tabela de simboluri[26].

[24] Ibidem, p. 4
[25] Ibidem, p. 85
[26] Ibidem, p. 89

# 4. Detalii privind implementarea practică

# 5. Rezultate experimentale

# 6. Concluzii

# Bibliografie

[1] J. L. B. P. U. S. S. A.-E. Ghader Kurdi, „A Systematic Review of Automatic Question Generation for Educational Purposes,” *International Journal of Artificial Intelligence in Education,* vol. 30, pp. 121-204, 2020.

[2] T. O. Z. S. Danijel Radošević, „Automatic On-line Generation of Student's Exercises in Teaching Programming,” în *Central European Conference on Information and Intelligent Systems*, Varaždin, 2010.

[3] T. M. Akiyoshi Wakatani, „Automatic Generation of Programming Exercises for Learning Programming Languages,” în *IEEE International Conference on Computer and Information Science,* Las Vegas, 2015.

[4] K. S. Thomas James Tiam-Lee, „Procedural Generation of Programming Exercises with Guides Based on the Student’s Emotion,” în *IEEE International Conference on Systems, Man and Cybernetics*, Miyazaki, 2018.

[5] B. K. M. M. Kate Compton, „Tracery: An Author-Focused Generative Text Tool,” în *International Conference on Interactive Digital Storytelling*, Copenhagen, 2015.

[6] „What is automated item generation?,” [Interactiv]. Available: https://assess.com/what-is-automated-item-generation/. [Accesat 8 Iunie 2022].

[7] „Teste școlare,” [Interactiv]. Available: https://brio.ro/teste-scolare. [Accesat 8 Iunie 2022].

[8] M. L. R. S. J. U. Alfred Aho, Compilers: Principles, Techniques, and Tools, Boston: Addison Wesley, 2006.

PAGINA GOALĂ