#### UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

**FACULTATEA DE INFORMATICĂ**



LUCRARE DE LICENȚĂ

Big Step Semantics Assistant

#### propusă de

### Nicolae Leuciuc

Sesiunea: *iulie, 2018*

#### Coordonator științific

### [Prof.dr. Dorel](https://profs.info.uaic.ro/~orar/participanti/orar_lucanu.html) Lucanu

#### UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

**FACULTATEA DE INFORMATICĂ**

Big Step Semantics Assistant

***Nicolae Leuciuc***

Sesiunea**:** *iulie, 2018*

#### Coordonator științific

### [Prof.dr. Dorel](https://profs.info.uaic.ro/~orar/participanti/orar_lucanu.html) Lucanu

Avizat,

Îndrumător Lucrare de Licență

Titlul, Numele și prenumele

Data Semnătura

#### DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemntatul(a) ………………………………………………………………………………………

domiciliul în …………………………………………………………………………………………………..

născut(ă) la data de ………………..…., identificat prin CNP ,

absolvent(a) al(a) Universității „Alexandru Ioan Cuza” din Iași, Facultatea de

………………………. specializarea …………………………………………………………, promoția

…………………………., declar pe propria răspundere, cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licență cu titlul:

\_

\_

elaborată sub îndrumarea dl. / d-na

, pe care urmează să o susțină în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului său într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice in vederea facilitării fasificării de către cumpărător a calității de autor al unei lucrări de licență, de diploma sau de disertație și în acest sens, declar pe proprie

răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am

întreprins-o.

Dată azi, ………………………… Semnătură student …………………………

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Titlul complet al lucrării*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, *data*

Absolvent *Prenume Nume*

(semnătura în original)

ACORD PRIVIND PROPRIETATEA DREPTULUI DE AUTOR

Facultatea de Informatică este de acord ca drepturile de autor asupra programelor-calculator, în format executabil și sursă, să aparțină autorului prezentei lucrări, *Prenume Nume.*

Încheierea acestui acord este necesară din următoarele motive:

*[Se explică de ce este necesar un acord, se descriu originile resurselor utilizate în realizarea*

*produsului-program (personal, tehnologii, fonduri) și aportul adus de fiecare resursă.]*

Iași, *data*

Decan *Prenume Nume* Absolvent *Prenume Nume*

(semnătura în original) (semnătura în original)

**CUPRINS**

**Introducere 8**

**Contribuții 10**

**Capitolul 1. Descrierea problemei 11**

*1.1* *AST 11*

*1.2* *Regula 12*

*1.3* *Fuctionalităti 15*

*1.4* *Concluzii 15*

**Capitolul 2. Descrierea soluției 17**

*2.1 Parser pentru limbajul de programare 17*

*2.2 Parser pentru formule 17*

*2.3 Implementarea regulilor 18*

*2.4 Vizualizare 22*

*2.5 Concluzii 22*

**Concluzii și posibilități de continuare 23**

**Bibliografie 24**

**Anexe 25**

**Introducere**

„Big Step Semantics Assistant” este un instrument software ce asistă la aplicarea regulilor ce definesc Big Step Operational Semantics ale unui limbaj de programare.

Cunoscută și sub numele de semantică naturală, semantică relațională sau semantică de evaluare, big step structual operational semantics [1], pe scurt big-step SOS, se poate vedea ca definiții de funcții sau de relații. Este foarte ușor și natural de folosit astfel că este recomandat să se folosească de fiecare dată când există posibilitatea.

Curiozitatea spre acest domeniu dar mai ales dorința de aprofundare m-au împins spre crearea unei aplicații ce ajută la o mai bună înțelegere atât a regulilor semanticii cât și a aplicabilității lor. Am creat astfel un sistem capabil de a primi ca input o formulă, cu starea finală necompletată, ce corespunde tiparului stabilit în gramatică și de a oferi ca output formula completă, calculând pas cu pas stările de la fiecare subformula. Modul de afișare este unul cât mai sugestiv: fiecare formulă rezultată din aplicarea regulilor reprezintă un nod dintr-un arbore ce are ca rădăcină formula inițială.

Am ales să structurez explicarea aplicației create în două capitole în felul următor:

Capitolul 1 include o descriere amplă a problemei abordate prin aplicația dezvoltată. Astfel, acesta cuprinde explicații clare cu privire la noțiunile folosite pe parcursul realizării “Big Step Semantics Assistant” precum: noțiunea de Abstract Syntax Tree (AST), noțiunea de formulă și de regulă (pentru care trebuie mai întâi să fie înțeles conceptul de semantică operațională, de asemnea prezentat în acest capitol). În același timp, este descrisă în amănunt modalitatea de a aplica regulile prezentate pe anumite formule, precum și funcționalitatea aplicației folosind aceste noțiuni și concepte.

Capitolul 2 conține o descriere a soluției în detaliu. Sunt astfel explicate pas cu pas fiecare dintre următoarele funcționalități explicând pe rând procesul realizării fiecăreia:

- Parser pentru limbajul de programare;

- Parser pentru formule;

- Generare arbore de deducție;

- Vizualizare;

La sfârșitul fiecărui capitol există de asemenea și o concluzie în care sunt prezentate informațiile sintetizate descrise în capitolul respectiv.

**Contributii**

Procesul de dezvoltare a acestei aplicații poate fi descris(structurat) în următorii păși:

1. Am preluat parserul pentru limbajul de programare de la documentația JavaCC [2] din exemplul interpreter pentru a putea compila și înțelege codul scris în interiorul formulei.

2. Pe baza parserului de mai sus am dezvoltat gramatica limbajului de programare pentru a include și formulele din Big Step SOS.

3. Am generat un arbore de deducție care să aplice regulile semanticii astfel ca în final să rezulte starea finală corectă.

4. Tratarea cazurilor în care execuția programului ajunge într-o buclă infinită prin plafonarea numărului maxim de noduri ale arborelui

5. După ce arborele a fost construit în memorie, el este convertit într-un limbaj care poate fi intepretat de Graphviz. Acest software de vizualizare a grafurilor open source afișează arborele așa cum a fost reținut în memorie, într-un mod sugestiv pentru utlilizator.

Toate aceste tehnologii foloste precum și pașii prin care am trecut pentru a dezvolta această aplicație sunt descrişi mai pe larg în cadrul capitolului 2 (Descrierea soluției).

**Capitolul 1. Descrierea problemei**

* 1. *Arbore Sintactic Abstract (AST)*

Un arbore abstract de sintaxă (AST) [3]  reprezintă o reprezentare [arborescentă](https://en.wikipedia.org/wiki/Directed_tree) a structurii [sintactice abstracte](https://en.wikipedia.org/wiki/Abstract_syntax) a [codului sursă](https://en.wikipedia.org/wiki/Source_code) scrise într-un [limbaj de programare](https://en.wikipedia.org/wiki/Programming_language). Fiecare nod al copacului denumește un construct care apare în codul sursă. Sintaxa este "abstractă" în sensul că nu reprezintă fiecare detaliu care apare în sintaxa reală. De exemplu, gruparea [parantezelor](https://en.wikipedia.org/wiki/Bracket#Parentheses) este implicită în structura arborescentă și o construcție sintactică ca o condiție if - atunci expresia poate fi marcată printr-un singur nod cu trei ramificații.

Aceasta distinge arborii de sintaxă abstracti de arborii de sintaxă concreți, [paraziți](https://en.wikipedia.org/wiki/Parse_tree) desemnați în mod tradițional , construiți în mod obișnuit de către un [parser](https://en.wikipedia.org/wiki/Parser) în timpul traducerii codului sursă și al procesului de [compilare](https://en.wikipedia.org/wiki/Compiler) . Odată construit, se adaugă informații suplimentare AST prin intermediul procesării ulterioare, de exemplu, [analiza contextuală](https://en.wikipedia.org/wiki/Semantic_analysis_(compilers)) .

Arborii de sintaxă abstractă sunt, de asemenea, utilizați în [analiza](https://en.wikipedia.org/wiki/Program_analysis)[programelor](https://en.wikipedia.org/wiki/Program_transformation) și în sistemele de [transformare a programelor](https://en.wikipedia.org/wiki/Program_transformation) .

Arborii de sintaxă abstractă sunt [structuri de date](https://en.wikipedia.org/wiki/Data_structures) utilizate pe scară largă în [compilatoare](https://en.wikipedia.org/wiki/Compilers) pentru a reprezenta structura codului programului. Un AST este de obicei rezultatul fazei de [analiză a sintaxei](https://en.wikipedia.org/wiki/Syntax_analysis) unui compilator. Acesta deseori reprezintă o reprezentare intermediară a programului prin mai multe etape pe care compilatorul le cere și are un impact puternic asupra rezultatului final al compilatorului.

Un AST are mai multe proprietăți care ajută la următorii pași ai procesului de compilare:

* Un AST poate fi editat și îmbunătățit cu informații cum ar fi proprietățile și adnotările pentru fiecare element pe care îl conține. O astfel de editare și adnotare este imposibilă cu codul sursă al unui program, deoarece ar presupune schimbarea acestuia.
* În comparație cu [codul sursă](https://en.wikipedia.org/wiki/Source_code) , un AST nu include semne de punctuație și delimitări neesențială (paranteze, punct și virgulă, paranteze etc.).
* Un AST conține, de obicei, informații suplimentare despre program, datorită etapelor consecutive de analiză efectuate de compilator. De exemplu, poate stoca poziția fiecărui element din codul sursă, permițând compilatorului să imprime mesaje de eroare utile.

AST este utilizată intens în timpul [analizei semantice](https://en.wikipedia.org/wiki/Semantic_analysis_(compilers)) , unde compilatorul verifică utilizarea corectă a elementelor programului și a limbii. De asemenea, compilatorul generează [tabele de simboluri](https://en.wikipedia.org/wiki/Symbol_table) bazate pe AST în timpul analizei semantice. Traversarea completă a copacului permite verificarea corectitudinii programului.

După verificarea corectitudinii, AST servește drept bază pentru generarea de coduri. Este adesea folosit pentru a genera o reprezentare intermediară (IR), uneori numită o [limbă intermediară](https://en.wikipedia.org/wiki/Intermediate_language) , pentru generarea de coduri.

* 1. *Regula*

Semantica operatională este o categorie de semantică formală a limbajului de programare în care anumite proprietăti dorite ale unui program, cum ar fi corectitudinea, siguranța sau securitatea, sunt verificate prin construirea dovezilor din declarațiile logice despre execuția și procedurile sale, mai degrabă decât prin atașarea semnificațiilor matematice la termenii săi ( semantica denotațională ).

Semanticile operaționale sunt clasificate în două categorii: semantica operatională structurală (sau small-step semantics) descrie în mod formal modul în care etapele individuale ale unui calcul au loc într-un sistem bazat pe calculator și semantica naturală de opoziție (sau big-step semantics) descrie modul în care se obțin rezultatele generale ale execuțiilor.

Alte abordări pentru asigurarea unei semantici formale a limbajelor de programare includ semantica axiomatică și semantica denotatională .

Semantica operațională pentru un limbaj de programare descrie modul în care un program valid este interpretat ca secvențe de etape de calcul. Aceste secvențe sunt apoi semnificația programului. În contextul [programelor funcționale](https://en.wikipedia.org/wiki/Functional_program) , pasul final dintr-o secvență de terminare returnează valoarea programului.

Având în vedere că semantica Big Step se bazează pe reguli, există un set de reguli bine stabilit [4]. În continuare voi lua un exemplu de formulă și voi aplică regulile pe ea.

O stare este o înșiruire de Map-uri, un Map fiind legarea unei variabile cu o valoare. De exemplu: x|->7; Se citește x este mapat la 7, adică x are valoarea 7 în starea <x|->7;>. O stare poate fi privită că o funcție: dacă privim starea <x|->7;> că σ, atunci

σ(x) = 7.

Starea σ din care plecăm va fi σ(x) = 8 și σ(y) = 0, cu x, y ∈ Id.

Alegem formula : (x / (y / x + 2), σ) ⇓ (?), la care vom calcula configurația din dreapta, care este reprezentată printr-o stare. Prima regulă pe care o putem aplica este împărțirea(BigStep-Div), astfel rezultând următoarea schemă :

(x*, σ*) ⇓ (8) (y / x + 2*, σ*) ⇓ (?)

(x / (y / x + 2)*, σ*) ⇓ (?)

În continuare se observă că (x, σ) ⇓ (8) este deja aplicată regula BigStep-LookUp și că pe formula (y / x + 2, σ) ⇓ (?) se poate aplica regula adunării(BigStep-Add) și rezultă:

. (y / x*, σ*) ⇓ (?) (2*, σ*) ⇓ (2)

(x*, σ*) ⇓ (8) (y / x + 2*, σ*) ⇓ (?)

(x / (y / x + 2)*, σ*) ⇓ (?)

Formula (x, σ) ⇓ (8) ajunge în stare finală, iar (y / x + 2, σ) ⇓ (?) derivează 2 subformule, una din care se observă că are regula BigStep-Int deja aplicată, iar pe cealaltă se poate aplica BigStep-Div:

. .

(y*, σ*) ⇓ (0) (x*, σ*) ⇓ (8) .

. (y / x*, σ*) ⇓ (?) (2*, σ*) ⇓ (2)

(x*, σ*) ⇓ (8) (y / x + 2*, σ*) ⇓ (?)

(x / (y / x + 2)*, σ*) ⇓ (?)

Observăm că (y, σ) ⇓ (0) și (x, σ) ⇓ (8) sunt variante finale și că starea din formula (y / x, σ) ⇓ (?) se poate completa ușor făcând calculul dintre stările din subformule:

. .

(y*, σ*) ⇓ (0) (x*, σ*) ⇓ (8) .

. (y / x*, σ*) ⇓ (0) (2*, σ*) ⇓ (2)

(x*, σ*) ⇓ (8) (y / x + 2*, σ*) ⇓ (?)

(x / (y / x + 2)*, σ*) ⇓ (?)

Același lucru se poate spune și despre (y / x + 2*, σ*) ⇓ (?):

. .

(y*, σ*) ⇓ (0) (x*, σ*) ⇓ (8) .

. (y / x*, σ*) ⇓ (0) (2*, σ*) ⇓ (2)

(x*, σ*) ⇓ (8) (y / x + 2*, σ*) ⇓ (2)

(x / (y / x + 2)*, σ*) ⇓ (?)

Și rezultatul final:

. .

(y*, σ*) ⇓ (0) (x*, σ*) ⇓ (8) .

. (y / x*, σ*) ⇓ (0) (2*, σ*) ⇓ (2)

(x*, σ*) ⇓ (8) (y / x + 2*, σ*) ⇓ (2)

(x / (y / x + 2)*, σ*) ⇓ (4)

* 1. *Fucționalități*

Crearea unui instrument software care ar putea aplica regulile descrise mai sus și rezolvarea unui astfel de sistem ar avea de nevoie de următoarele funcționalități:

- Parserul pentru limbajul de programare pentru a putea citi și interpreta expresiile, statement-urile și programul din configurații;

- Parserul pentru formule pentru a putea citi și interpreta formulele pentru a putea aplica regulile necesare, prezente în anexe;

- Implementarea regulilor și generarea arborelui de deducție;

- Vizualizarea concretă a arborelui creat anterior.

* 1. *Concluzii*

Ne putem folosi de Abstract Syntax Tree pentru parcugerea mult mai ușoară a formulelor și a codului scris în ele.

După cum am observat la descrierea regulilor, am putea spune că rezultatul final poate fi văzut ca un arbore care are puncte ca frunze și instanțe a regulilor ca noduri.

Un astfel de arbore complet(toate frunzele sunt puncte și toate nodurile sunt instanțe corecte ale regulilor) se numește Proof Tree.

Structura finală fiind un arbore ne permite să derivăm fapte, secvențe din programe direct prin semantică. Putem numi rădăcină faptul(formula) care a fost „dovedita”.

**Capitolul 2. Descrierea soluției**

1. *Parser pentru limbajul de programare*

Parserul pentru limbajul de programare a fost preluat de la documentația JavaCC [2] din exemplul Interpreter cu scopul de a compila și înțelege codul scris în interiorul formulei.

1. *Parser pentru formule*

Pe baza parserului pentru limbajul de programare am dezvoltat gramatica pentru a include și formulele din Big Step SOS.

Am extins gramatica menționată mai sus pentru a putea interpreta și formule din semantică. După cum știm din anexă, o formulă este alcătuită din 2 configurații, și configurațiile sunt de mai multe feluri:

- Literal

- Statement State

- Expression State

- State

- Statement

În interiorul configurației am creat un ConfigContent, adică un container pentru configurație. Astfel, acel container am adăugat cele 5 modalități de a scrie o configurație.

- Un Statement State conține o serie de statement-uri (assignment-uri, if-uri, while-uri etc.) și o stare

- Un Expression State conține o expresie (o adunare, o verificare de egalitate, un && etc.) și o stare

- Un State constă în o serie de Map-uri, despărțite printr-un „ ; ”

- Un Map constă în un PrimaryExpression și un Literal despărțite prin simbolul „ |->”

- Un PrimaryExpression constă în un Id, un Literal sau o expresie.

Cele două configurații din formulă sunt despărțite prin „ => ”, iar Statement State/Expression State din containerul din interiorul configurațiilor este despărțit prin „ # ” de State.

1. *Implementarea aplicării regulilor*

Regulile [4] sunt descrise într-o manieră destul de ordonată în documentația despre Big Step Semantics [1] și această ordine, dacă urmată cum trebuie, este foarte folositoare la implementare. Descrierea începe cu regula de la integer care spune că un întreg, în orice stare ar fi, el va fi acel întreg. Continuă cu LookUp, care spune că variabila x, în starea σ este σ(x), dacă σ(x)!= ⊥.

Deocamdată, aceste reguli sunt frunze(noduri la care nu se mai poate aplica nicio altă regulă). Acest lucru se va schimba totuși odată cu regula Add, care face adunarea dintre doi întregi. Regula creează doi fii: în primul se adaugă prima parte a adunării iar în al doilea, a două parte. Similar cu adunarea este și împărțirea, înmulțirea etc.

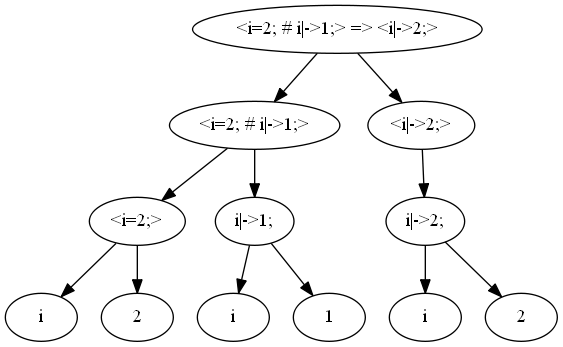
Urmează regulile NotTrue și NotFalse, care creează un sigur fiu și acel fiu constă în mare parte în aplicarea regulii de LookUp, care verifică dacă variabila este true sau false și returnează ca stare finală inversul variabilei.

Regula And creează un fiu, face LookUp și verifică valoarea booleană a expresiei.În cazul în care aceasta este falsă, se oprește și returnează false ca stare finală. Dacă valoarea întoarsă la LookUp este true, se mai face un fiu și se mai execută odată LookUp și orice valoare întoarce, aceasta va fi rezultatul final.

Regula Block menționează creearea unui singur nod care constă în conținutul block-ului și o stare σ’ care este rezultatul final al execuției codului. Dacă block-ul este gol nu se mai creeza nici un fiu. Aici eu consider că ordinea este mai slab structurată deoarece începând cu Block-ul, se face tranziția pe Statement State, până acum fiind un Expression State. În momentul în care am început implementarea la Statement State am observat că este mult mai ușor să creez Assignment-ul și după Sequence și apoi instrucțiuni precum Block, While, If.

Am întâmpinat probleme în momentul în care am ajuns la regula Sequence. Această regulă spune că dacă sunt 2 sau mai multe statement-uri, prima se duce în fiul stâng, alături de starea inițială și va rezulta o stare σ1, iar restul statement-urilor se duc în al doilea fiu, și se vor executa în starea σ1, rezultant o stare finală σ2. Problema a fost că fiul din dreapta se baza pe starea calculată în fiul din stânga. Același lucru pot spune că a fost și la regula And(unde dacă fiul din stânga avea starea True, fiul din dreaptă se creea, altfel nu se creea), doar că la un nivel mai mic. Acum este la un nivel mai înalt și pentru asta am observat că am nevoie de o funcție computeRHS care să calculeze configurația din partea dreaptă a formulei. Această funcție calcula recursiv de fiecare data când întâlnea configurația „ <?> „. În aceste cazuri ea era apelată în interiorul funcției applyRule imediat după ce era creeat fiul din stânga și calcula configurația dreaptă a formulei, mai departe funcția applyRule prelua configurația calculată și o transfera ca stare în formula a doua.

Aceste reguli au fost implementate apelând AST-ul creeat pentru fiecare formulă ca în următorul exemplu:



Avem formula inițială care se împarte în două configurații: una este de tipul Statement State și cealaltă este de tipul State. Configurația de tipul Statement State se împarte în un Assignment și un State. State-urile sunt la rândul lor o serie de Map-uri.

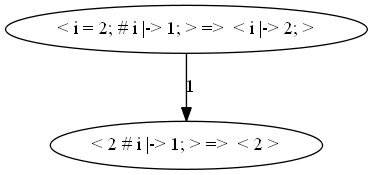
De exemplu dacă voiam să accesez i-ul din Assignment-ul i=2 trebuie să formulez în modul următor:

formula.getChild(0).getChild(0).getChild(0);

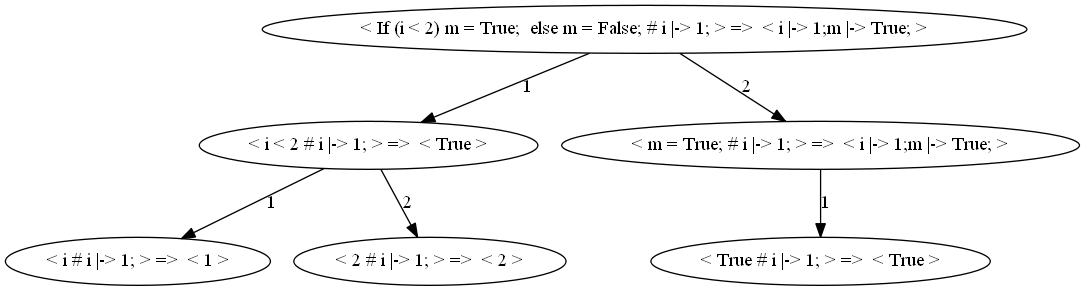
Pentru Map-ul i->1:

formula.getChild(0).getChild(1);

Pentru exemplul de mai sus, un arbore generat după aplicarea regulilor ar arată în felul următor:



Având în vedere că este un simplu Assingment se aplică regula și rezultă un singur nod. Lucrurile în schimb se complică destul de rapid în momentul în care introducem un While, statement care creează cel puțin trei noduri(cazul în care condiția este falsă) pentru fiecare iterație. De exemplu un While în care se execută o singură iterație creează treisprezece noduri.



În exemplul cu If se observă cum fiul stâng este condiția și că este verificată și că în fiul din dreaptă se execută codul din do și nu din else. La final se adaugă Map-ul „ m|->True” la configurația din dreaptă a formulei.

1. *Vizualizare*

# Graphviz este un software de vizualizare a grafurilor open source. Vizualizarea grafurilor este o metodă de a reprezenta în mod structurat informația ca diagrame de grafuri abstracte. Se folosește în aplicații de networking, bioinformatică, ingineria software, baze de date, web design și în interfețe vizuale pentru alte domenii tehnice.

# Graphviz primește descrieri de grafuri într-un limbaj simplu text și face diagrame în formate folositoare. Are multe opțiuni folositoare precum culori, fonturi, hyperlink-uri, forme personalizate după nevoile utilizatorului.

# Arborele generat prin aplicarea regulilor este după convertit în limbajul menționat mai sus, care se numește DOT. Acest limbaj este înțeles de Graphviz.

# Convertirea are loc cu ajutorul funcției prettyPrint, care scrie în interiorul unui fișier în modul următor:

# - Un nodeID

# - Și un label care conține formula din interiorul nodului

# - Apoi afișează legătura dintre nodul curent cu cel al copiilor

# - Și se apelează recursiv

# 

1. *Concluzii*

# Folosind informațiile din capitolul 1, aplicația Big Step Semantics Assistant a fost implementată astfel încât să ofere utilizatorului o înțelegere mai bună a semanticii și pentru ajutarea în calcularea unei configurații.

# La nivelul aplicației s-a încercat obținerea unei reprezentări cât mai sugestive a aplicării regulilor, structura de date finală find un arbore de arbori, este afișată într-un mod ușor de înțeles pentru oricine.

# **Concluzii și posibilități de continuare**

# Lucrarea „Big Step Semantics Assistant” surprinde implementarea unei gramatici care sa inteleaga si sa interpreteze un limbaj de programare si o formula, aplicand regulile Big Step Operational Semantics pe formulele primite ca input, astfel incat sa calculeze configuratia rezultata dupa executia codului.

# Am preluat parserul pentru limbajul de programare de la documentatia JavaCC de la exemplul Interpreter si am extins gramatica pentru a interpreta formulele, totodata am implementat in Java regulile din semantica si o modalitate de calculare a configuratiei finale. Am folosit ca structura de date un arbore care contine in fiecare nod un arbore abstract de sintaxa(formula). Acest arbore a fost construit pe baza regulilor si apoi convertit in limbajul DOT, pentru a fi inteles de Graphviz si afisat ca o imagine.

# Pentru viitor exista mai multe directii in care se poate dezvolta aplicatia:

# Parcurgerea pas cu pas a fiecarei aplicari de reguli

# Construirea nivel cu nivel a arborelui final si mentionarea regulei care se aplica

# Acceptarea ca input o formula deja completa si verificarea corectitudinii

# Creearea unui mediu mai user-friendly pentru scrierea formulelor

**Bibliografie**

|  |  |
| --- | --- |
| [1] | Big-Step Structural Operational Semantics, Illinois, 2015. |
| [2] | „JavaCC,” [Interactiv]. Available: https://javacc.org/doc. |
| [3] | „Wikipedia,” [Interactiv]. Available: https://en.wikipedia.org/wiki/AST. |
| [4] | „Anexa 4”. |

**Anexe**

**Big Step Operational Semantics**

Cunoscuta si sub numele de semantica naturala, semantica relationala sau semantica de evaluare, big step structual operational semantics, pe scurt big-step SOS, se poate vedea ca definitii de functii sau de relatii. Este foarte usor si natural de folosit incat este recomandat sa se foloseasca de fiecare data cand este posibilitatea. Din pacate exista anumite limitari care fac aplicarea semanticii imposibila sau inconvenabila:

- definirea caracteristicilor lingvistice de control intensiv

- caracteristici non-deterministe

Semantica unui limbaj de programare este data de un sistem de reguli. Secventele sunt relatii asupra configuratiilor de forma C=>R, unde R este configuratia obtinuta dupa evaluarea completa a lui C. Un exemplu de regula ar putea fi:

*C*1 ⇓ *R*1 *C*2 ⇓ *R*2 *. . . Cn* ⇓ *Rn*

*C*0 ⇓ *R*0

unde C0, C1, ... Cn sunt configuratii ce retin fragmente de limbaj de programare impreuna cu toate componentele necesare pentru semantica, R0, R1,...Rn sunt configuratiile rezultate sau ireductibile, adica configuratiile care mai pot fi impartite in alte configuratii.

Semantica descrie cum se face evaluarea finala a rezultatului, bazandu-se pe combinarea rezultatelor obtinute de catre configuratiile copil. De exemplu pentru adunare se procedeaza in modul urmator:

(*a*1*, σ*) ⇓ (*i*1) (*a*2*, σ*) ⇓ (*i*2)

(*a*1 + *a*2*, σ*) ⇓ (*i*1 +*Int i*2)

Intelesul unei relatii precum (*a*1*, σ*) ⇓ (*i*1) este ca expresia aritmetica a1 este evaluata in starea *σ* la intregul i1. Este intalnit destul de des ca valorile singure precum intregii sau valorile boolene sa nu se inscrie intr-o configuratie, ci sa se scrie asa (*a*1*, σ*) ⇓ *i*1. Nu exista o regula clara asupra folosirii parantezelor rotunde sau a semnului ⇓, ci se pot folosi paranteze drepte, acolade si semnul =>.

Big Step Operational Semantics (BSOS) se bazeaza pe formule si pe aplicarea anumitor reguli pe ele. O formula consta in doua configuratii care se impart si ele la randul lor in mai multe moduri:

* Literal
* Statement State
* Expression State
* State
* Statement

Un Literal:

* Integer
* Boolean

Statement State:

* Assignment, While, If, orice fel de statement care se gaseste intr-un limbaj de programare(efectiv o bucata de cod)
* O stare in care se execută codul de mai sus

Expression State:

* O expresie ce urmeaza a fi evaluata(==,<=,<, etc.)
* O stare in care se face evaluarea

State:

* O stare

Statement:

* Un Statement State cu o stare vida

Dupa cum se poate observa din clasificarea de mai sus, unele configuratii nu au nevoie de o stare si unele nu au nevoie de cod. De exemplu (*i*1) este o configuratie ce retine doar un singur intreg, care se poate obtine ca rezultatul evaluarii unei expresii aritmetice, pe cand (*σ*) este o configuratie ce contine doar o stare care poate fi obtinuta dupa evaluarea unui Statement.

**Regulile Semanticii Big Step**

Rolul unui sistem este de a deriva secvente sau fapte. Faptele care deriva din acest sistem vor avea forma urmatoare:

* (*a, σ*) ⇓ (*i*)
* (*b, σ*) ⇓ (*t*)
* (*s, σ*) ⇓ (*σ’*)
* (p) ⇓ (*σ*)

unde a este expresie aritmetica, b este expresie booleana, s este statement, p este cod, i este variabila intreaga, t este variabila boolean, si *σ, σ’* sunt stari.

Intelesul triplului de forma (*a, σ*) ⇓ (*i*) este ca expresia aritmetica a se evalueaza/executa/transforma in intregul i in starea *σ*; intelesul (*b, σ*) ⇓ (*t*) este similar, numai ca cu valori boolene. Motivul pentru care este suficient sa derivam fapte asa de simple este pentru ca evaluarea nu are nici un efect advers.

In cazul limnajului IMP simplu, relatia de tranzitie va fi determinista, adica i1=i2 daca (*a, σ*) ⇓ (*i1*) si (*a, σ*) ⇓ (*i2*) pot fi deduse. In contextul limbajelor non-deterministe, triplul (*a, σ*) ⇓ (*i*) este posibil ca *a* sa fie evaluatala *i* in starea *σ*, dar si la alti intregi(similar pentru expresii,statement-uri si programe boolene).

Urmatoarele reguli fac parte din limbajul IMP:

|  |  |
| --- | --- |
| (*i, σ*) ⇓ (*i*) | (BigStep-Int) |
| (*x, σ*) ⇓ (*σ*(*x*)) if *σ*(*x*) != ⊥ | (BigStep-Lookup) |
| (*a*1*, σ*) ⇓ (*i*1) (*a*2*, σ*) ⇓ (*i*2)  (*a*1 + *a*2*, σ*) ⇓ (*i*1 +*Int i*2) | (BigStep-Add) |
| (*a*1*, σ*) ⇓ (*i*1) (*a*2*, σ*) ⇓ (*i*2)  (*a*1 / *a*2*, σ*) ⇓ (*i*1 */Int i*2) | (BigStep-Div) |
| (*t, σ*) ⇓ (*t*) | (BigStep-Bool) |
| (*a*1*, σ*) ⇓ (*i*1) (*a*2*, σ*) ⇓ (*i*2) (*a*1 <= *a*2*, σ*) ⇓ (*i*1 ≤*Int i*2) | (BigStep-Leq) |
| (*b, σ*) ⇓ (true) (! *b, σ*) ⇓ (false) | (BigStep- Not-True) |
| (*b, σ*) ⇓ (false) (! *b, σ*) ⇓ (true) | (BigStep- Not- False) |
| (*b*1*, σ*) ⇓ (false)  (*b*1 && *b*2*, σ*) ⇓ (false) | (BigStep-And-False) |
| (*b*1*, σ*) ⇓ (true) (*b*2*, σ*) ⇓ (*t*) (*b*1 && *b*2*, σ*) ⇓ (*t*) | (BigStep-And-True) |
| ({}*, σ*) ⇓ (*σ*) | (BigStep-Empty-Block) |
| (*s, σ*) ⇓ (*σ*j) ({ *s* }*, σ*) ⇓ (*σ*j) | (BigStepBlock) |
| (*a, σ*) ⇓ (*i*)  *(x* = *a;, σ*) ⇓ *(σ[i/x*]) | (BigStep-Asgn) |
| (*s*1*, σ*) ⇓ (*σ*1) (*s*2*, σ*1) ⇓ (*σ*2) (*s*1 *s*2*, σ*) ⇓ (*σ*2) | (BigStep-Seq) |
| (*b, σ*) ⇓ (true) (*s*1*, σ*) ⇓ (*σ*1)  (if (b) *s*1 else *s*2*, σ*) ⇓ (*σ*1) | (BigStep-If-True) |
| (*b, σ*) ⇓ (false) (*s*2*, σ*) ⇓ (*σ*2) (if (b) s1 else *s*2*, σ*) ⇓ (*σ*2) | (BigStep-If-False) |
| (*b, σ*) ⇓ (false) (while (*b*) *s, σ*) ⇓ (*σ*) | (BigStep-While-False) |
| (*b, σ*)⇓(true)(s while(b)s, *σ)*⇓( *σ’*)  while (*b*) *s, σ*) ⇓ (*σ’* ) | (BigStep-While-True) |
| (*s, xl* ›→ 0) ⇓ (*σ*)  (int *xl*; *s*) ⇓ (*σ*) | (BigStepPgm) |