UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

IMP Parser

realizat de:

Andrei Agafiței

Sesiunea: *Februarie, 2020*

Coordonator științific

Prof. Dr. Lucanu Dorel

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

IMP Parser

Andrei Agafiței

Sesiunea: *Februarie, 2020*

Coordonator științific

Prof. Dr. Lucanu Dorel

Avizat,

Îndrumător Lucrare de Licenţă

Titlul, Numele şi prenumele

_____

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conţinutului lucrării de licenţă

Subsemntatul(a)

……

domiciliul în ……..

născut(ă) la data de ……., identificat prin CNP ……., absolvent(a) al(a) Universităţii „Alexandru Ioan Cuza" din Iaşi, Facultatea de …….. specializarea ……., promoţia ……., declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul: _____

_____

_____elaborată sub îndrumarea dl. / d-na _____, pe care urmează să o susţină în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

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

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării fasificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie

răspundere că lucrarea de faţă 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 „*IMP Parser*”, 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 *Andrei Agafiței*

_____

(semnătura în original)

DECLARAŢIE PRIVIND ORIGINALITATE ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licenţă cu titlul "*IMP Parser*" este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
- codul sursă, imagini etc. preluate din proiecte *open-source* sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
- rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iaşi, *data*

Absolvent *Andrei Agafiței*

_____

(semnătura în original)

1. Cuprins

2. Introducere

Pe parcursul celor trei ani de facultate, și ulterior lucrând în domeniul informaticii, am avut șansa să am contact cu diferite limbaje de programare (C/C++, Java, Python, Scala). Un limbaj de programare este un set bine definit de expresii, reguli și tehnici valide de formulare a instrucțiunilor pentru un computer. Așadar, din cele spuse mai sus reiese fapul că fiecare limbaj de programare are un set de reguli sintactice și semantice.

Această temă, parsarea unui limbaj IMP, a reieșit din dorința de a înțelege mai bine modul în care un program, de dimensiuni și complexitați diferite, este adus din forma sa inițială (codul scris într-un editor de text, codul sursă) în forma sa finală (output-ul, rezultatul afișat în urma executării codului sursă).

Parsarea sau analiza sintactică reprezintă parcurgerea și analizarea unui text fie in limbaj natural sau formal, cu identificarea elementelor ireductibile care îi corespund, în raport cu o gramatică formală. Un parser sau analizor sintactic este o componentă a unui interpretor, în cadrul cărora identifică structura textului de intrare și o aduce într-o formă potrivită pentru prelucrări ulterioare, căutând erori de sintaxă în acest text.

Astfel de proiecte sunt disponibile pe internet. Diferența dintre proiectele existente și proiectul meu îl reprezintă limbajul de programare folosit pentru implementarea lucrării (și anume Scala), și modul de parsare și evaluare al inputului. La majoritatea proiectelor existente, limbajul predominant folosit este Haskell (limbaj functional). În schimb, proiectul meu este scris in Scala (limbaj funcțional, dar în același timp și limbaj orientat obiect), limbaj asemănător cu Java, iar modalitatea de parsare și evaluare este ușor de înțeles dar și ușor de modificare în scopul testării și învățarii.

Metodologia folosită este relativ simplă. Codul sursă pentrul limbajul IMP este preluat dintru-un fișier text. Acesta este supus unor reguli de parsare , iar dacă inputul respectă regula, aceasta va întoarce o structură de date care va fi folosită ulterior la crearea unui AST (Abstract Syntax Tree) care devine o componentă importantă în relizarea proiectului. Utilizatorului i se oferă o interfață grafică prietenoasă, ușor de folosit prin care poate introduce cod sursă atât scris de utilizator cât și încărcat dintr-un fișier dorit de acesta.

3. Contribuții

În relizarea proiectului cele mai importante contribuții au venit din partea absolventului. Contribuțiile absolventului în realizarea proiectului sunt:

- înțelegerea principiilor de funcționare ale bibliotecii FastParse;
- documentarea și înțelegerea realizării unui parser pentru limbajul IMP;
- stabilirea unui plan de lucru în realizarea lucrării;
    - testarea în scopul învățării blibliotecii FastParse;
    - crearea interfeței cu utilizatorul;
    - crearea codului proprou zis;
    - crearea legăturilor dintr-e interfața grafică și codul propriu zis;
- căutarea și alegerea unei soluții optime;
- implementarea soluției;
    - crearea modulelor de parsare al inputului;
    - crearea regulilor de parsare al inputului;
    - crearea AST-ului;
    - evaluarea AST-ului;
- testarea soluției;

Pentru realizarea lucrării, absolventul a folosit limbajul de programare Scala, biblioteca FastParse, o bibliotecă OpenSource, JavaFX pentru realizarea interfeței cu utilizatorul.

4. Implementare și testare

4.1. Limbajul IMP

Limbajul IMP are expresii aritmetice care includ domeniul numerelor întregi, expresii boolene, instrucțiuni de atribuire, instrucțiuni condiționale (If – else), instrucțiuni repetitive (While). Toate variabilele utilizate într-un program IMP sunt declarate la începutul programului, pot păstra doar valori întregi (pentru simplitate nu există valorile boolene în IMP) și sunt instanțiate cu valoarea implicită 0.

Pentru a defini o sintaxă pentru limbajul IMP, folosim forma *Backus-Naur* (BNF), notație pentru gramaticile fără context iar apoi se va folosi o notație algebrică mixfix alternativă și complet echivalentă. Acesta din urmă este în general mai potrivit pentru evoluțiile semantice ale unui limbaj.

Sorts: Int, Bool , Var, AExp, BExp, Stmt, Pgm;
Subsorts: Int, Var < AExp ; Bool < BExp;
Operations:
_ (arithmetic_operator) _ : AExp x AExp → AExp
_(comparison_operator)_ : AExp x AExp → BExp
_(binary_operator)_ : BExp x BExp → BExp
not (_) : BExp → BExp
_ := _ : Var x AExp → Stmt
if _ then _ else _ endif; : BExp x Stmt x Stmt → Stmt
while _ do _ endwhile; : BExp x Stmt → Stmt
arithmetic_operator : + , - , * , / , % ;
comparison_operator: <, > , =, !=
binary_operator: and, or

Schema 1

Sintaxa limbajului IMP ca o semnătură algebrică

Int ::= the domain of (unbounded) integer numbers
Bool ::= the domain of booleans
VarId ::= standard identifiers
AExp ::= Int
| VarId
| AExp (arithmetic_operator) AExp
BExp ::= Bool
| AExp (comparison_operator) AExp
| not BExp
| BExp (binary_operator) BExp
Stmt ::= VarId := AExp
| if BExp then Stmt else Stmt
| while BExp do Stmt
Pgm ::= vars List{VarId } ; Stmt

Schema 2

Sintaxa limbajului IMP, folosind algebrica BNF

4.2. Generarea AST-ului

Ce este un AST (Abstract Syntax Tree)?

Un arbore de sintaxă abstractă (AST), sau doar un arbore de sintaxă , este o reprezentare a arborelui structurii abstracte sintactice a codului sursă scris într-un limbaj de programare . Fiecare nod al arborelui semnifică o construcție care apare în codul sursă. Sintaxa este abstractă în sensul că nu reprezintă fiecare detaliu care apare în sintaxa reală, ci doar detaliile structurale sau legate de conținut. Un AST are mai multe proprietăți care ajută la etapele suplimentare ale procesului de compilare:

- Un AST poate fi editat și îmbunătățit cu informații precum proprietăți și adnotări 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.
- Comparativ cu codul sursă , un AST nu include punctuație și delimitare neesențiale (paranteze, virgule, paranteze etc.).
- Un AST conține, de obicei, informații suplimentare despre program, datorită etapelor consecutive de analiză de către compilator.

AST-ul este o componentă esențială în realizarea lucrării, deoarece reprezintă structura de bază a proiectului. El este folosit atât la verificarea codului sursă, dar cel mai important rol este acela de a evalua codul sursă. În cadrul proiectului meu, AST-ul este format dintr-o structură de dată specifică limbajului Scala, și anume “object”. Acesta conține o serie de “sealed trait”-uri , formate la rândul lor dintr-o serie de “case object”-uri. Fiecare sealed trait și case object sunt împărțite pe categorii cum ar fi: operatori, instructiuni si expresii. Proprietatea pe care se bazeaza relațiile între clase este *moștenirea*.

Exemplificând întru-un exemplu, vom lua “sealed trait operator”. Acest “sealed trait” se referă la operatorii aritmetici (adunare, scădere, înmulțire, împărțire, mod). Fiecare din acești operatori reprezintă un “case object”, cum se poate vedea în dreapta, care moștenesc “sealed trait”-ul *operator*. Această moștenire îmi asigură faptul că pot crea expresii aritmetice înlănțuite, iar la final, regula de parsare să îmi întoarcă o strcutură de date de tip *operator*.

```
sealed trait operator
case object operator{
  case object Add extends operator
  case object Sub extends operator
  case object Mul extends operator
  case object Div extends operator
  case object Mod extends operator
}
```

Figura 1. Sealed trait operator

Cele mai importante 2 obiecte sunt reprezentate de "sealed trait"-urile *stmt* (care conține instrucțiunile specific limbajului: If-Else, While, Assign, Declaration, Print) și *expr* (care conține operațiile aritmetice și operațiile boolene).

object ***AST***

- case class **Id**(name: String)
- sealed trait ***operator***
  - case object ***And*** extends ***boolop***
  - case object ***Or*** extends ***boolop***
- sealed trait ***cmpop***
  - case object ***Add*** extends ***operator***
  - case object ***Sub*** extends ***operator***
  - case object ***Mul*** extends ***operator***
  - case object ***Div*** extends ***operator***
  - case object ***Mod*** extends ***operator***
- sealed trait ***expr***
  - case class ***BoolOp***(operator: boolop, values: Seq[expr]) extends ***expr***
  - case class ***Compare***(left: expr, ops: cmpop, right: expr) extends ***expr***
  - case class ***Num***(n: Any) extends ***expr***
  - case class ***Str***(s : String) extends ***expr***
  - case class ***Bool***(n: Any) extends ***expr***
  - case class ***ADD***(left: expr, right: expr) extends ***expr***
  - case class ***SUB***(left: expr, right: expr) extends ***expr***
  - case class ***MUL***(left: expr, right: expr) extends ***expr***
  - case class ***DIV***(left: expr, right: expr) extends ***expr***
  - case class ***MOD***(left: expr, right: expr) extends ***expr***
  - case class ***AND***(left: Seq[expr], right: Seq[expr]) extends ***expr***
  - case class ***OR***(left: Seq[expr], right: Seq[expr]) extends ***expr***
  - case class ***NOT***(target: expr) extends ***expr***
- sealed trait ***stmt***
  - case class ***Assign***(target: Id, value: expr) extends ***stmt***
  - case class ***While***(condition: expr, body: Seq[stmt]) extends ***stmt***
  - case class ***If***(condition: expr, body: Seq[stmt], orelse: Option[Seq[stmt]]) extends ***stmt***
  - case class ***Declaration***(targets: Seq[Id]) extends ***stmt***
  - case class ***Print***(target: expr) extends ***stmt***
- sealed trait ***unaryop***
  - case object ***Not*** extends ***unaryop***

Schema 3

Schema completă al AST-ului

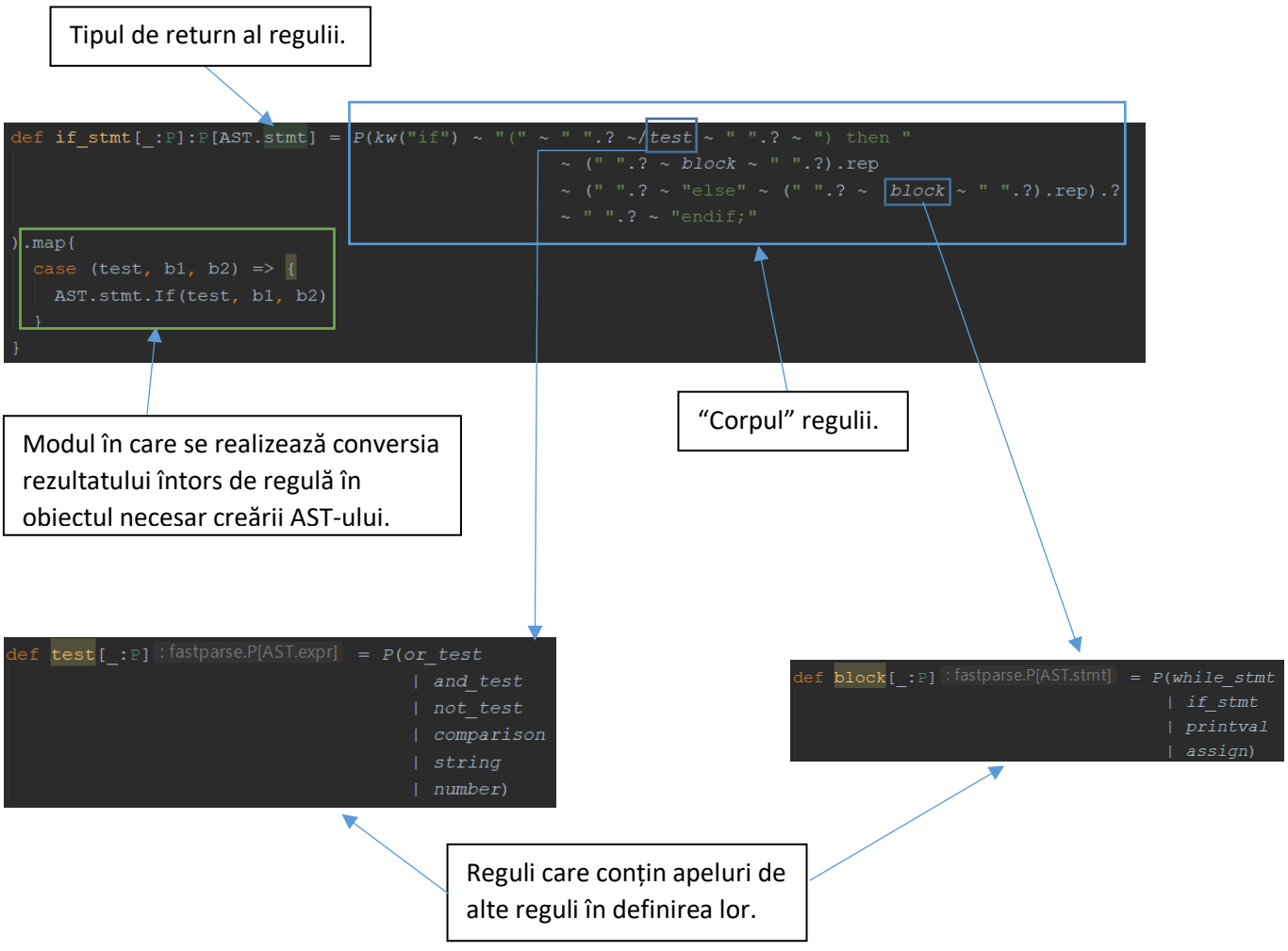
4.3. Regulile de parsare

Parserul a fost creat cu ajutorul bibliotecii *FastParse*, librărie specifică limbajului Scala. Această librărie este foarte ușor de înțeles, oferă posibilitatea creării de reguli pentru orice tip de limbaj, dar nu numai.

O regulă preia un string (input) și întoarce o structură de date numai dacă parsarea inputului a avut success, sau un mesaj de eroare în caz contrar. Realizarea AST-ului, prezentat mai sus), a avut loc numai prin intermediul obiectelor întoarse de regula de parsare. O regulă de parsare este o înșiruire de caractere cu o anumită logică, precedate de simboluti de legătură. Conținutul unei reguli poate varia de la o regulă la alta:

- regula poate conține mai multe reguli în corpul acesteia;
- regula poate conține mai multe apeluri de alte reguli în definirea sa.

Un exemplu de regulă este următoarea:



Ce semnifică corpul regulii?

Regula prezentată mai sus este regula pentru parsarea If-urilor. Pentru limbajul IMP, sintaxa If-ului este următoarea:

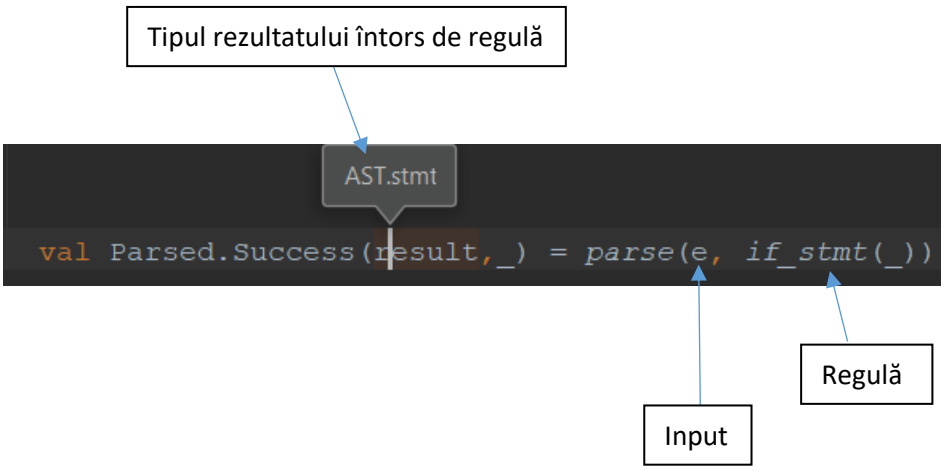
if(condiție_booleană) then bloc_de instrucțiuni_1 else bloc_de instrucțiuni_2 endif;

Condiția booleană trebuie să respecte pattern-ul: oricâte spații urmate de un rezultat boolean urmat de oricâte spații. Blocurile de instrucțiuni sunt asemănătoare: oricâte spații urmate de alte instrucțiuni, urmate de oricâte spații, iar acesteă structură repetânduse de minim o dată. În final, If-ul trebuie să respecte pattern-ul: grupul de cuvine "if", urmat de paranteză rotundă deschisă, urmată condiția booleană (descrisă mai sus), urmată de paranteză rotundă închisă, urmată de un singur spațiu, urmat de grupul de cuvinte "then", urmat de blocul de instrucțuni 1, urmat de grupul " else " , urmat de blocul de instrucțuni 2, urmat de grupul "endif; ".

Dacă inputul peste care se aplică regula respectă pattern-ul regulii, aceasta va returna obiectul specific regulii.

Când se aplică regula?

Regula se aplică peste string atunci când acesta este parsat. La parsarea inputului are loc verificarea sintaxei limbajului pentru cazul respectiv, dar și salvarea rezultatului.



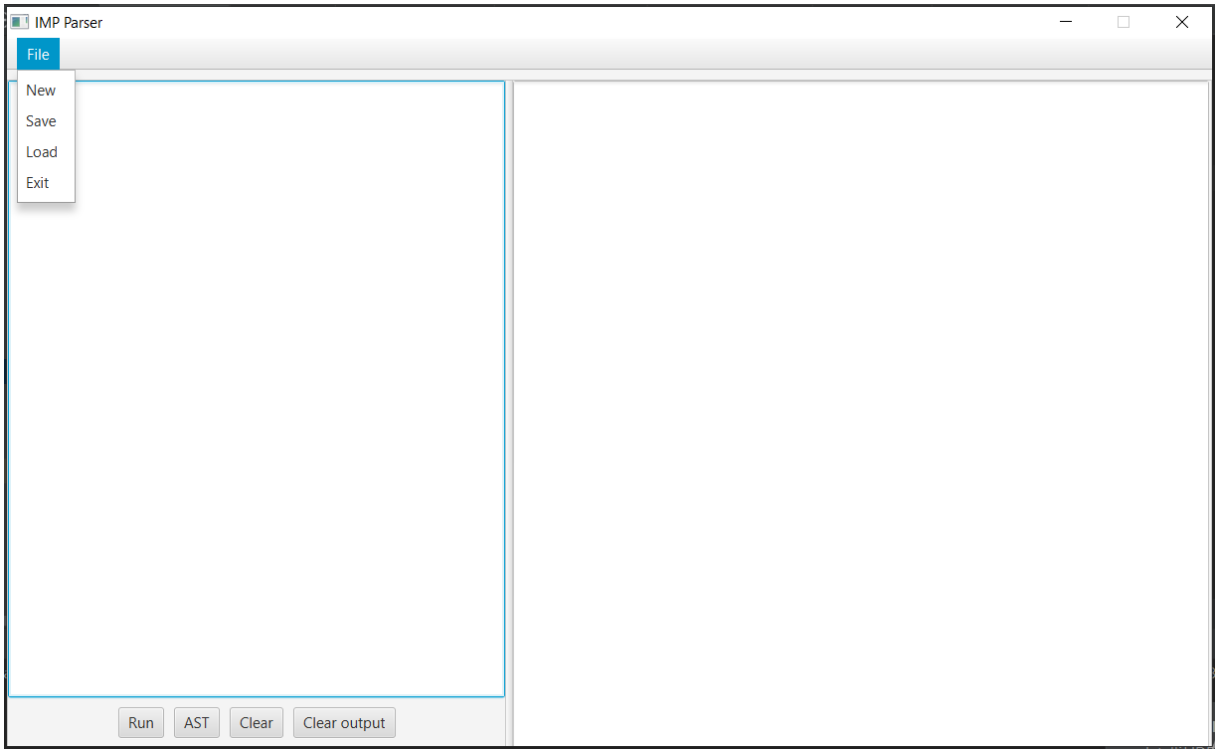
4.4. Fluxul de lucru

Ce se întâmplă atunci când este pornită aplicația?

La apăsarea butonului de *Run*, aplicația va deschide fereastra de interacțiune cu utilizatorul. Această fereastră permite utilizatorului să scrie propriul cod sau să încarce un cod deja existent. După realizarea primei acțiunui (încărcarea codului), urmează salvarea acestuia. Salvarea acestuia are un rol important, deoarece , în caz contrat, în cazul încărcarii unui fișier și modificării acestuia într-un pas viitor, când butonul de *Run* va fi apăsat, se va evalua versiunea anterioară salvării. Pentru a preveni acest incident, la apăsarea butonului de *Run* va avea loc salvarea automată a codului. După salvarea acestuia, sunt posibile două acțiuni:

- rularea codului; AST-ul va fi evaluat iar rezultatul va fi printat în chenarul din partea dreapta a interfației cu utilizatorul;
- afișarea AST-ului;

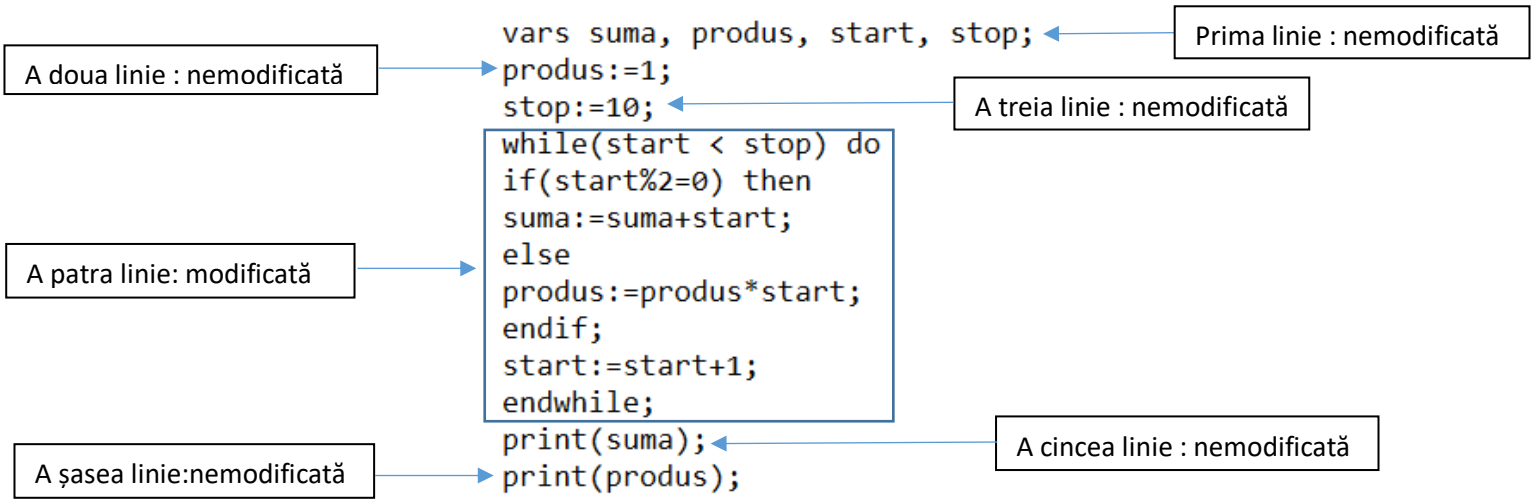
Am vorbit mai sus despre interfața cu utilizatorul. Această interfață este împărțită în *InputPane* și *OutputPane*. Prima fereastră este rezervată preluării codului sau scrierii acestuia, conținând 4 butoane de acțiune. A doua fereastră este rezervată afișării output-ului dar și pentru afișării AST-ului.



În partea de sus a ferestrei de interacțiune cu utilizatorul se pot observa un meniu (File) care conține 4 acțiuni : *New* (apăsarea acestui buton va determina curățarea spațiului rezervat scierii de cod. După curățarea spațiului, utilizatorul poate introduce codul dorit), *Save* (apăsarea acestui buton determină salvarea codului scris, dar și a unui fișier importat.), *Load* (apăsarea acestui buton determină deshiderea unui *explorer* într-o locație stabilită. Următorul pas este alegerea fișierului dorit și importarea coduluiîn secțiunea de cod) și *Exit* (apăsarea acestui buton determină închiderea interfeței).

În partea de jos a ferestrei de interacțiunde cu utilizatorul sunt prezente alte 4 butoane car prelucrează AST-ul dar și cele 2 mini-ferestre. Butonul de *Run* apelează o funcție care are ca scop prelucrarea AST-ului și evaluarea acestuia pentru obținerea outputlui. Butonul de *AST* are ca scop afișarea AST-ului specific codului sursă introdus de utilizator. Butonul de *Clear* curăță fereastra specifică introducerii inputului, iar butonl de *Clear Output* curăță fereastra specifică afisării rezultatului.

După ce utilizatorul încarcă codul sursă dorit, acesta este salvat întru-un fișier. Fișierul este citit de program linie cu linie și face o parsare a inputului. Deoarece rezultatul după parsare este folosit pentru crearea AST-ului, este nevoie de o parsare atentă al inputului. Acesta este verificat astfel încât să respecte regulile de parsare al AST-ului. Astfel, liniile care sunt instrucțiunilor simple (asignarea, print-ul) vor fi puse într-o listă exact așa cum sunt găsite în fișier. În cazul If-ului și al While-ului metoda diferă: se salvează linia unde s-a găsit prima data o instrucțiune If sau While apoi se merge mai departe în fișier până se găsește ultima apariție al sfârșitului de instrucțiune (*endif;* sau *endwhile;*). În cazul apariției unei alte instrucțiuni, aceasta nu va fi luată în calcul, și va fi inclusă în corpul primei instrucțiuni găsite. Când sfârșitul de instrucțiune este găsit, poziția lui va fi salvată. Ultimul pas este crearea unui string care pornește de la poziția de început al instrucțiunii și se termină la poziția de sfârșit, tot ce este între aceste poziții reprezintă un input, și va fi adăugat în lista care va fi trimisă mai departe creării AST-ului.



Schema 4

Schema de prelucrare al inputului

În schema prezentată mai sus se explică sumar modul de creare al listei din care va rezulta AST-ul programului. Această listă va fi utilizată în pasul următor, acela de creare al AST-ului: se va parcurge lista rezultată în pasul anterior și se vor trata cazurile

- caz 1: când linia începe cu stringul "*vars*" se va aplica regula pentru declarare;
- caz 2: când linia începe cu stringul "*if*" se va aplica regula de parsare pentru *IF;*
- caz 3: când linia începe cu stringul "*while*" se va aplica regula de parsare pentru *WHILE;*
- caz 4: când linia începe cu stringul "print" se va aplica regula de parsare pentru *PRINT;*
- caz 5: se va aplica regula de parsare pentru asignare (este singurul caz rămas);

După fiecare aplicare de regulă asupra stringurilor din listă va rezulta o bucată de AST care va fi adăugată într-o listă, care în final va reprezenta lista AST-ului programului. În schema 5 prezentată mai jos sunt și prezente și verificările pentru declarare:

- declarările se fac la începutul codului pe prima linie;
- să nu existe mai mult de un rând cu declarări;

```
input.foreach(e=>{
  if(input.count(_.trim.take(4).equals("vars")) > 1) throw new Exception("Multiple declarations!! Just one line with declarations!")
  if(input.tail.count(_.trim.take(4).equals("vars")) > 0) throw new Exception("Declarations must be on the first line on your code!!")

  if(e.trim.take(2).equals("if")) {
    val Parsed.Success(result,_) = parse(e, if_stmt(_))
    ast = ast :+ result
  }
  else if(e.trim.take(5).equals("while")) {
    val Parsed.Success(result,_) = parse(e, while_stmt(_))
    ast = ast :+ result
  }
  else if(e.trim.take(4).equals("vars")){
    val Parsed.Success(result,_) = parse(e, declaration(_))
    ast = ast :+ result
  }
  else if(e.trim.take(5).equals("print")){
    val Parsed.Success(result,_) = parse(e, printval(_))
    ast = ast :+ result
  }
  else
    {
      val Parsed.Success(result,_) = parse(e, assign(_))
      ast = ast :+ result
    }
})
```

Schema 5

Algoritmul de creare al AST-ului

După ce AST-ul a fost creat cu succes, acesta este folosit atât pentru verificarea codului inițial cât și pentru evaluarea și afișarea rezultatului așteptat. Pe baza AST-ului se realizaeză evaluarea codului.

Evaluarea are loc prin parcuregrea AST-ului și verificarea tipului clasei. Am creat doua funcții recursive pentru evaluarea expresiilor și al instrucțiunilor. Cea mai importantă funcție este evaluare expresiilor deoarece acestea stau la baza tuturor instrucțiunilor (la realizarea asignărilor pentru expresii matematice, la verificarea condițiilor la *IF* și *WHILE*) și se bazează pe recunoașterea tipurilor claselor. Recunoașterea tipurilor claselor se face cu un "*match*" peste componentele AST-ului. De exemplu: avem ADD(Str(Suma), Str(Start)). Match-ul va prinde clasa ADD unde se vor evalua cei doi membri (stâng: Str(Suma) și drept: Str(Start)). Cei doi membri vor fi supuși evaluării iar match-ul care va prinde tipul clasei este *Str*, care va returna valoarea singurului său membru (Suma și respectiv Start). După returnarea celor 2 valori, se va calcula suma celor 2 membri iar valoarea este returnată.

```
def arithmetic_eval(arith: AST.expr)(implicit map: Map[String, Int]): Int = {
  var result = 0
  arith match {
    case nr: AST.expr.Num => result = nr.n.asInstanceOf[Int]
    case str: AST.expr.Str if str.s.equals("true") => result = 1
    case str: AST.expr.Str if str.s.equals("false") => result = 0
    case str: AST.expr.Str => result = if (map.contains(str.s)) map(str.s) else throw new Exception("Variable " + str.s + " isn't declared.")
    case add: AST.expr.ADD => result = arithmetic_eval(add.left) + arithmetic_eval(add.right)
    case sub: AST.expr.SUB => result = arithmetic_eval(sub.left) - arithmetic_eval(sub.right)
    case mul: AST.expr.MUL => result = arithmetic_eval(mul.left) * arithmetic_eval(mul.right)
    case div: AST.expr.DIV => result = arithmetic_eval(div.left) / arithmetic_eval(div.right)
    case mod: AST.expr.MOD => result = arithmetic_eval(mod.left) % arithmetic_eval(mod.right)
  }
  result
}
```

Figura 2. Funcția de evaluare a expresiilor

Valorile variabilelor și modificările acestora sunt ținute într-o structură de date numită *Map*. În final după evaluarea AST-ului folosind funcția de evaluare a instrucțiunilor această mapă va conține valorile finale ale variabilelor. Folosind funcția *print* putem afișa valoarea variabilelor în orice moment.

Figura 3. Schema transformării inputului. Input -> AST -> output

4.5. Tratarea erorilor

Parsarea unui limbaj de programare este o muncă dificilă indiferent de limbajul de programare deoarece trebuie luate în considerare multe cazuri pentru care codul inserat de utilizator. În cazul lucrării mele de licență pot apărea erori cum ar fi:

- utilizatorul poate începe o instrucțiune *IF*, dar să uite să o închidă (folosind *endif;*). Același caz poate apărea și în cazul instrucțiunei *WHILE*;

Modul de rezolvare al erorii este tratat atunci când se parsează input-ul. Se numără aparițiile structurilor *if(expresie)* , *while(expresie)*, *endif;* și *endwhile;* iar dacă la finalul parcurgerii codului sursă, aceste valori nu sunt egale cu 0, atunci avem eroare.

```
if( i == seq.size -1) {
  if(if_c < 0) throw new Exception("You have "+ -if_c +"\"if" + "\"if\" statement unfinished (\"endif;\" missing)!!")
  if(while_c < 0) throw new Exception("You have "+ -while_c +"\"if" + "\"if\" statement unfinished (\"endwhile;\" missing)!!")
  if(if_c > 0) throw new Exception("You have "+ if_c + "\"if\" statement start and unfinished!!")
  if(while_c > 0) throw new Exception("You have "+ while_c + "\"if\" statement start and unfinished!!")
}
```

- utilizatorul poate încheia o instrucțiune fără să fie începută;

Modul re rezolvare este tratat mai sus.

- utilizatorul declară variabile pe mai multe linii de cod;

Modul de rezolvare este prin verificarea rezultatului după parsarea inputului. Este permis doar un singur rând cu declarații, iar acesta este primul rând din program. Alte rânduri care conțin declarații sunt interzise.

```
if(input.count(_.trim.take(4).equals("vars")) > 1) throw new Exception("Multiple declarations!! Just one line with declarations!")
if(input.tail.count(_.trim.take(4).equals("vars")) > 0) throw new Exception("Declarations must be on the first line on your code!!")
```

- folosirea variabilelor care nu au fost declarate;

Modul de rezolvare este prin parcurgerea AST-ului și verificarea mapei în care se află variabilele dacă variabila curentă aparține mapei de valori;

```
case str: AST.expr.Str => {
  result = if (map.contains(str.s)) map(str.s) else throw new Exception("Variable " + str.s + " isn't declared.")
}
```

- folosirea greșită a semnului de asignare (:=);

4.6. Despre Scala

Scala este un limbaj de programare multi-paradigmă ce îmbină concepte din programarea orientată pe obiecte și programarea funcțională.

Scala rulează pe platforma Java compilatorul generând bytecode compatibil cu programele Java existente iar rezultatul să fie rulat pe o mașină virtuală Java . Scala asigură interoperabilitatea limbajului cu Java, astfel încât bibliotecile scrise în oricare limbaj pot fi trimise direct în codul Scala sau Java. La fel ca Java, Scala este orientat pe obiect , și folosește o sintaxă curly-brace care amintește de limbajul de programare C . Spre deosebire de Java, Scala are multe caracteristici ale limbajelor de programare funcționale precum *Scheme* , *Standard ML* și *Haskell* , inclusiv *currying* ,inferenta tipului , imuabilitate , evaluare leneșă și potrivire a modelului . De asemenea, are un sistem avansat de tip care acceptă tipuri de date algebrice , covarianță și contravarianță , tipuri de ordin superior (dar nu tipuri de rang superior) și tipuri anonime . Alte caracteristici ale Scala care nu sunt prezente în Java includ supraîncărcarea operatorilor , parametrii opționali, parametrii numiți și șirurile brute. Proiectat pentru a fi concis, multe dintre deciziile de proiectare ale Scala au avut ca scop abordarea criticilor asupra Java . Numele Scala vine de la "scalable language" (în română "limbaj scalabil"), semnificând faptul că este proiectat să crească o dată cu necesitățile utilizatorilor.

4.7. Despre sbt

SBT este un instrument open-source construi pentru proiecte Scala si Java. Principalele sale caracteristici sunt:

- Suport nativ pentru compilarea codului Scala și integrarea cu multe cadre de testare Scala
- Compilare, testare și implementare continuă
- Testare incrementală și compilare
- Creați descrieri scrise în Scala folosind un DSL
- Gestionarea dependenței folosind Ivy (care acceptă depozitele în format Maven)
- Suport pentru proiecte mixte Java / Scala

SBT este instrumentul de construire *de facto* din comunitatea Scala, folosit de cadrul de operare Lift și Play Framework .

Piața comercială a societății Scala, Lightbend Inc. , a numit sbt „*probabil cel mai bun instrument pentru construirea proiectelor Scala*”, spunând că cele două caracteristici mai importante ale acesteia sunt compilarea incrementală și un shell interactiv. Atunci când este introdus modul de compilare continuă, compilatorul Scala este instanțat o singură dată, ceea ce elimină costurile de pornire ulterioare, iar modificările fișierului sursă sunt urmărite astfel încât doar dependențele afectate sunt recompilate. Consola interactivă permite modificarea setărilor de construire în zbor și intrarea în Scala REPL împreună cu toate fișierele de clasă ale proiectului. SBT deja a fost alimentat în biblioteca standard Scala înainte, când API-ul procesului său a fost adoptat în Scala 2.9.

4.8. Despre JavaFX

JavaFX este o platformă software pentru crearea și livrarea de aplicații desktop , precum și aplicații desktop RCA (Rich Client Application) care pot rula pe o mare varietate de dispozitive. JavaFX este destinat să înlocuiască Swing ca biblioteca standard GUI pentru Java SE , dar ambele vor fi incluse pentru viitorul prevăzut. JavaFX are suport pentru computere desktop și browsere web pe Microsoft Windows , Linux și macOS.

În realizarea lucrării de licență am folosit JavaFX pentru construirea meniului aplicației datorită ușurinței de lucru dar și multitudinilor de controale built-in (butoane, etichete, text) care pot fi folosite pentru o mai ușoară dezvoltare a scenei.

4.9. Despre FastParse

FastParse este o librărie specific limbajului Scala pentru parsarea stringurilor si biților în structură de date. Aceasta îți oferă libertatea de creare a unor parsere într-un mod eficient, ușor de înțeles la o viteză mare, cu sistem de raportare a erorilor foarte efficient. Datorită modului ușor de creare a regulilor, a vitezei mari de execuție, a debugerului puternic și a sistemului de raportare de erori, FastParse poate parsa stringuri simple până la limbaje complexe. În comparație cu alte instrumente de generate parsere(ANTLR, Lex/Yacc) , FastParse nu necesită nicio etapă specială de construie sau generare de cod: regulile sunt obiecte definite direct in cod si metodele de apel.

5. Concluzii

În concluzie, consider că proiectul *IMP Parser* este o experiență din care am avut de învățat multe informații utile în viitor. Pornind de la o idee, documentându-mă, testând și exersând anumite noțiuni (înțelegerea librăriei FastParse, crearea unui algoritm folosind această librărie, implementarea sintaxei pentru limbajul IMP) și până la realizarea întregului parser am avut ocazia să deprind anumite informații, să creez anumite legături , să îmi dezvolt o altfel de gândire care mă va ajuta în domeniul profesional și nu numai.

Consider că înțelegerea proiectului, modul de gândire, structura lui este ușor de înțeles, ușor de modificat astfel încât oricine poate realiza un nou parser, sau poate aduce modificări asupra acestuia, luând în considerare alte variante ale aceleiași teme (parsarea unui limbaj IMP).

6. Bibliografie

http://fsl.cs.illinois.edu/images/0/0d/CS522-Spring-2011-PL-book-imp.pdf

http://www.kframework.org/images/6/60/CS422-Spring-2010-02-BigStep.pdf

https://en.wikipedia.org/wiki/Scala_(programming_language)

https://github.com/lihaoyi/fastparse

https://en.wikipedia.org/wiki/JavaFX