

Tema LFA 2018-2019

Interpreter pentru limbajul IMP

Responsabili: Alexandru Bogdan Andrei, Vlad Nedelcu

Data publicare cerință:	29.10.2018
Data publicare checker și teste:	TBA
Deadline:	14.12.2018

1. Specificații temă

1.1. Obiectiv

Obiectivul temei constă în aplicarea cunoștințelor teoretice dobândite în cursul de “Limbaje Formale și Automate” prin realizarea unui analizor lexical pentru limbajul IMP.

1.2. Cerință

Să se implementeze un parser pentru IMP, simplul limbaj imperativ prezentat în cadrul cursului, echipat cu facilități minimale precum *if*, *while*, atribuiri, expresii aritmetice și boolene. Parser-ul va fi realizat folosind generatorul de analizoare lexicale JFlex, respectiv generatorul de parsere ANTLR (bonus). Se cere apoi realizarea interpretării pentru programele IMP parsate folosind JFlex.

2. Limbajul IMP

2.1. Limbajul de descriere

Limbajul este descris printr-o gramatică BNF și folosește următoarea convenție de culori:

- **albastru** - neterminali
- **verde** - operatori ai limbajului BNF și paranteze ajutătoare
- **rosu** - terminali (elemente care fac parte efectiv din limbajul descris)

Pentru a simplifica sintaxa, se folosesc operatorii *****, **+** și **?** cu semnificația din expresiile regulate.

2.2. Primitive

Primitivele pot fi variabile (String), valori algebrice (numere în baza 10) sau valori boolene (True sau False).

```

<Digit> ::= (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)*
<Number> ::= <Digit>(0 | <Digit>)* | 0
<String> ::= ( a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y |
z )*
<Var> ::= <String>
<AVal> ::= <Number>
<BVal> ::= True | False

```

2.3. Expresii aritmetice și boolene

Expresiile aritmetice se pot referi la o variabilă, o valoare algebrică sau suma/împărțirea dintre alte două expresii aritmetice. Expresiile boolene se pot referi la o valoare booleană, la valoarea de adevăr a unei comparații între două expresii algebrice (“>”), sau la o asociere cu “și” logic între mai multe alte expresii boolene.

Se va considera că operatorul algebric “/” are o prioritate mai mare față de “+”, iar operatorul boolean “!” are prioritate mai mare față de “&&”.

În cazul în care rezultatul împărțirii a două expresii aritmetice nu este un număr întreg, se va păstra doar partea întreagă.

```

<AExpr> ::= <Var> | <AVal> | <AExpr> + <AExpr> | <AExpr> / <AExpr> | ( <AExpr> )
<BExpr> ::= <BVal> | <BExpr> && <BExpr> | <AExpr> > <AExpr> | ! <BExpr> |
( <BExpr> )

```

2.4. Corpul unui program

Corpul programului este reprezentat printr-un statement. Acesta poate fi gândit ca un ansamblu de mai multe statement-uri separate prin “;” și reprezentând o atribuire, o structură de tip “if”, sau una de tip “while”. Statement-urile pot fi încadrate între acolade, formând un Block. Block-urile vor intra obligatoriu în componența structurilor “if” și “while”. În cazul în care un “if” nu va avea nimic pe ramura de “else”, atunci vom utiliza un Block gol.

```

<Block> ::= {} | { <Stmt> }
<Stmt> ::= <Var> = <AExpr> ; |
    <Block> |
    if ( <BExpr> ) <Block> else <Block> |
    while ( <BExpr> ) <Block> |
    <Stmt> ; <Stmt>

```

2.5. Lista de variabile

La începutul fiecărui program există o listă de variabile separate prin “,”.

```

<VarList> ::= <Var> | <Var> , <VarList>

```

2.6. Structura programului

Un program va conține mai întâi lista de variabile ce pot fi întâlnite în program, având neapărat tipul întreg. Apariția altor variabile în corpul programului decât cele declarate în listă va trebui semnalat printr-o eroare corespunzătoare (mai multe despre erorile ce vor trebui afișate în secțiunea 3). Urmează apoi corpul efectiv al programului.

```
<Prog> ::= int <VarList> ; <Stmt>
```

3. Raportarea de erori

În timpul rulării programului pot apărea erori, a căror apariție va trebui semnalată, iar rularea programului va fi sistată imediat. Aceste erori pot fi:

- *UnassignedVar* – în corpul programului este folosită o variabilă nedeclarată în lista inițială de variabile
- *DivideByZero* – se încearcă împărțirea la 0 a unei expresii aritmetice

Nu este neapărat ca erorile să fie detectate la run time, dar este obligatoriu ca parsarea să fie dusă până la capăt, iar AST-ul să fie construit complet, după modelul de la secțiunea 4.

4. Specificatii program

4.1. Descriere comportament input/output

Programul va citi dintr-un fișier numit “input” programul în limbajul IMP ce va trebui interpretat.

Ca ieșire:

- se va afișa într-un fișier cu numele “arbore” arborele sintactic (AST-ul) în formatul specificat în secțiunea 4.2;
- se va afișa într-un fișier cu numele “output”, pe câte o linie distinctă, valoarea fiecărei variabile din lista de variabile de la începutul programului (e.g. “a=21”); dacă una din variabile nu a fost atribuită pe durata rulării programului se va afișa “null” ca valoare asociată (e.g. “a=null”);

Pentru bonus (parsarea cu ANTLR) se va cere doar construcția arborelui sintactic în modul descris în secțiunea 4.2 și afișarea sa într-un fișier cu numele “arbore-b”.

ATENȚIE! Dacă programul este întrerupt în urma unei excepții fișierul “output” va conține doar numele acelei excepții care a întrerupt rularea programului, urmat de un spațiu și linia din fișierul sursă la care a fost depistată (e.g. “UnassignedVar 8”)!

4.2. Descriere arbore sintactic

În timpul parsării se va construi un arbore sintactic pentru un program IMP. Fiecare regulă descrisă anterior va avea un corespondent în lista următoare de noduri:

- **MainNode** = **<Prog>**
 - va reprezenta nodul rădăcină al AST-ului
 - poate avea oricâți copii
 - va fi printat "**<MainNode>**"
- **IntNode** = **<AVal>**
 - va reprezenta o valoare numerică
 - nu va avea copii, fiind un nod terminal
 - va fi printat "**<IntNode>** **<Number>**"
- **BoolNode** = **<BVal>**
 - va reprezenta o valoare booleană
 - nu va avea copii, fiind un nod terminal
 - va fi printat "**<BoolNode>** True" sau "**<BoolNode>** False"
- **VarNode** = **<Var>**
 - va reprezenta o variabilă
 - nu va avea copii, fiind un nod terminal
 - va fi printat "**<VarNode>** **<String>**"
- **PlusNode** = **<AExpr> + <AExpr>**
 - va reprezenta operația aritmetică de adunare
 - va avea 2 copii
 - va fi printat "**<PlusNode>** +"
- **DivNode** = **<AExpr> / <AExpr>**
 - va reprezenta operația aritmetică de împărțire
 - va avea 2 copii
 - va fi printat "**<DivNode>** /"
- **BracketNode** = **(<AExpr>) | (<BExpr>)**
 - va reprezenta parantezele utilizate în cadrul expresiilor aritmetice sau booleene
 - va avea 1 copil
 - va fi printat "**<BracketNode>** ()"
- **AndNode** = **<BExpr> && <BExpr>**
 - va reprezenta operația booleană ȘI
 - va avea 2 copii
 - va fi printat "**<AndNode>** &&"
- **GreaterNode** = **<AExpr> > <AExpr>**
 - va reprezenta operatorul de comparare "mai mare"
 - va avea 2 copii

- va fi printat "<GreaterNode> >"
- **NotNode** = ! <BExpr>
 - va reprezenta operatorul de negare
 - va avea 1 copil
 - va fi printat "<NotNode> !"
- **AssignmentNode** = <Var> = <AExpr> ;
 - va reprezenta operația de atribuire
 - va avea 2 copii
 - va fi printat "<AssignmentNode> ="
- **BlockNode** = {} | { <Stmt> }
 - poate cuprinde un statement în el sau să rămână gol
 - va avea 1 copil sau niciun copil (în cazul unui Block gol)
 - va fi printat "<BlockNode> {}"
- **IfNode** = if (<BExpr>) <Block> else <Block>
 - va fi compus dintr-o Condiție și 2 Block-uri. În funcție de valoarea de adevăr a condiției, se va merge pe ramura *Then* sau *Else*
 - va avea 3 copii (Condiția, Body si Else)
 - va fi printat "<IfNode> if"
- **WhileNode** = while (<BExpr>) <Block>
 - va fi compus dintr-o Condiție și 1 Block. Block-ul va fi executat până când condiția devine falsă
 - va avea 2 copii (Condiția, Body)
 - va fi printat "<WhileNode> while"

În fișierul *arbore.out* se va printa AST-ul folosind caracterul TAB ("\t") pentru a evidenția nivelul de indentare. Fiecare nod va fi printat pe nivelul său de indentare, iar copiii săi vor avea un TAB în plus față de acesta. MainNode va avea nivelul de indentare **0**.

ATENȚIE! Prima linie (în care vom declara variabilele folosite în program) nu va intra în componența AST-ului și nici nu va trebui printată.

4.3. Exemplu

Pentru a fi mai clar, vom explica printarea arborelui pe un exemplu scurt. Pentru următorul program vom avea AST-ul:

```
a=0;
if (!(a > 3)) {
    a = 1;
} else {}
```

```

<ProgramRoot>
  <AssignmentNode> =
    <VarNode> a
    <IntNode> 0
  <IfNode> if
    <BracketNode> (
      <NotNode> !
      <BracketNode> (
        <GreaterNode> >
        <VarNode> a
        <IntNode> 3
      <BracketNode> {
        <AssignmentNode> =
          <VarNode> a
          <IntNode> 1
        <BracketNode> {

```

Pentru alte exemple, consultați fișierele de referință.

5. Punctaj

Checker-ul oferă un punctaj între 0 și 150. 50 de puncte din 150 sunt bonus. Vor exista atât teste publice, cât și private (care nu au însă o complexitate mai mare decât cele publice, având ca unic scop evitarea primirii punctajului pentru hardcodări). Punctajul pentru fiecare categorie va fi:

- 60p = realizarea parser-ului în JFlex
 - 10p = atribuirii
 - 15p = expresii aritmetice
 - 25p = expresii boolene, construcții de tipul “if”
 - 10p = construcții de tipul “while”
- 25p = interpretarea programelor folosind JFlex
- 10p = raportarea erorilor
- 5p = fișier README cu detalierea implementării
- 50p = realizarea parser-ului folosind ANTLR

6. Trimitere temă

Arhiva trebuie să conțină un director cu numele JFlex și unul cu numele ANTLR (dacă există implementare pentru bonus), fiecare dintre acestea conținând:

1. surse, a căror organizare nu vă este impusă
2. un fișier Makefile care să aibă target de build, clean și run

3. un fisier README in care să descrieți implementarea temei
Arhiva trebuie să fie **zip**, nu tar.gz, rar, 7z, ace sau alt format ezoteric. Aceasta va fi trimisă pe vmchecker, în momentul în care se finalizează configurarea platformei.

ATENȚIE! Deadline-ul va fi atât soft, cât și hard, neacceptându-se arhivele trimise ulterior.

7. Bibliografie

JFlex - <http://jflex.de/download.html>

ANTLR - <https://github.com/antlr/antlr4/blob/master/doc/java-target.md>

JFlex マニュアル - http://jflex.de/jflex_manual_j.html