Tema LFA 2018-2019
Interpretor pentru limbajul IMP

Responsabili: Alexandru Bogdan Andrei, Vlad Nedelcu

Dată publicare cerinţă:	**29.10.2018**
Dată publicare checker şi teste:	**22.11.2018**
Deadline:	**04.01.2019**

1. Specificaţii temă

1.1. Obiectiv

Obiectivul temei constă în aplicarea cunoştinţelor teoretice dobândite în cursul de "Limbaje Formale şi Automate" prin realizarea unui analizor lexical pentru limbajul IMP.

1.2. Cerinţă

Să se implementeze un parser pentru IMP, simplul limbaj imperativ prezentat în cadrul cursului, echipat cu facilităţi minimale precum *if*, *while*, atribuiri, expresii aritmetice şi 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 şi foloseşte următoarea convenţie de culori:

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

Pentru a simplifica sintaxa, se folosesc operatorii ∗, + şi ? cu semnificaţia 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 şi boolene

Expresiile aritmetice se pot referi la o variabilă, o valoare algebrică sau suma/împărţirea dintre alte două expresii aritmetice. Expresiile boolene se pot referi la o valoare booleană, la valoarea de adevăr a unei comparaţii între două expresii algebrice (“>”), sau la o asociere cu “şi” logic între mai multe alte expresii boolene.
Se va considera că operatorul algebric “/” are o prioritate mai mare faţă de “+”, iar operatorul boolean “!” are prioritate mai mare faţă de “&&”. Se va considera că operatorii “+”, “/” si “&&” asociază la stânga.
În cazul în care rezultatul împărţirii 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 “;” şi 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 componenţa structurilor "if" şi "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 conţine mai întâi lista de variabile ce pot fi întâlnite în program, având neapărat tipul întreg. Apariţia altor variabile în corpul programului decât cele declarate în listă va trebui semnalat printr-o eroare corespunzătoare (mai multe despre erorile ce vor trebui afişate în secţiunea 3). Urmează apoi corpul efectiv al programului.

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

3. Raportarea de erori

În timpul rulării programului pot apărea erori, a căror apariţie 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 iniţială de variabile sau se incearcă folosirea valorii unei variabile neasignate
- *DivideByZero* – se încearcă împărţirea 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 secţiunea 4.

4. Specificatii program

4.1. Descriere comportament input/output

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

Ca ieşire:

- se va afişa într-un fişier cu numele “arbore” arborele sintactic (AST-ul) în formatul specificat în secţiunea 4.2;
- se va afişa într-un fişier 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 afişa “null” ca valoare asociată (e.g. “a=null”);

Pentru bonus (parsarea cu ANTLR) se va cere doar construcţia arborelui sintactic în modul descris în secţiunea 4.2 şi afişarea sa într-un fişier cu numele “arbore-b”.

ATENŢIE! Dacă programul este întrerupt în urma unei excepţii fişierul “output” va conţine doar numele acelei excepţii care a întrerupt rularea programului, urmat de un spaţiu şi linia din fişierul 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
  - va avea 1 copil
  - 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 operaţia aritmetică de adunare
  - va avea 2 copii
  - va fi printat "<*PlusNode*> +"
- DivNode = <AExpr> / <AExpr>
  - va reprezenta operaţia aritmetică de împărţire
  - va avea 2 copii
  - va fi printat "<*DivNode*> /"
- BracketNode = (<AExpr>) | (<BExpr>)
  - va reprezenta parantezele utilizate în cadrul expresiilor
  - va avea 1 copil
  - va fi printat "<*BracketNode*> *()*"
- AndNode = <BExpr> && <BExpr>
  - va reprezenta operaţia booleană ŞI
  - 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 operaţia 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 Condiţie şi 2 Block-uri. În funcţie de valoarea de adevăr a condiţiei, se va merge pe ramura *Then* sau *Else*
    - va avea 3 copii (Condiţia, Body si Else)
    - va fi printat "*<IfNode> if*"
- WhileNode = while (<BExpr>) <Block>
    - va fi compus dintr-o Condiţie şi 1 Block. Block-ul va fi executat până când condiţia devine falsă
    - va avea 2 copii (Conditia, Body)
    - va fi printat "*<WhileNode> while*"
- SequenceNode = <Stmt> <Stmt>
    - va fi compus din 2 Statement-uri.
    - va avea 2 copii
    - va fi printat "*<SequenceNode>*"

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

ATENŢIE! Prima linie (în care vom declara variabilele folosite în program) nu va intra în componenţa AST-ului şi nici nu va trebui afişată.

4.3. Exemplu

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

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

```
<MainNode>
        <AssignmentNode> =
                <VarNode> a
                <IntNode> 0
        <IfNode> if
                <BracketNode> ()
                        <NotNode> !
                                <BracketNode> ()
                                        <GreaterNode> >
                                                <VarNode> a
                                                <IntNode> 3
                <BlockNode> {}
                        <AssignmentNode> =
                                <VarNode> a
                                <IntNode> 1
                <BlockNode> {}
```

Pentru alte exemple, consultați fișierele de referință.

5. Punctaj

Checker-ul oferă un punctaj între 0 si 150. 50 de puncte din 150 sunt bonus. Vor exista atât teste publice, cât şi 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 in JFlex
  - 5p = atribuiri
  - 15p = expresii aritmetice
  - 20p = expresii boolene, construcţii de tipul “if”
  - 10p = construcţii de tipul “while”
  - 10p = teste generale (private)
- 25p = interpretarea programelor folosind JFlex
- 10p = raportarea erorilor
- 5p = fişier README cu detalierea implementării
- 50p = realizarea parser-ului folosind ANTLR

6. Trimitere temă

~~Arhiva trebuie să conţină un director cu numele JFlex şi unul cu numele ANTLR (dacă există implementare pentru bonus), fiecare dintre acestea conţinând:~~

2 arhive (JFlex şi ANTLR), fiecare dintre acestea conţinând un Makefile şi un folder JFlex / ANTLR (în funcţie de arhivă):

1. surse, a căror organizare nu vă este impusa
2. un fişier Makefile care să aibă următoarele target-uri:
   a. build: compilează tema şi generează fişierele .class
   b. run: rulează clasa Main şi generează cele două fişiere de ieşire
   c. clean: şterge fişierele de ieşire şi fişierele obiect

   Makefile-ul va trebui pus în rădăcina arhivei!!!
3. un fişier README în care să descrieţi implementarea temei
4. pentru arhiva de ANTLR, veti adauga si fisierul antrlr.jar pe care l-ati folosit local (verificati exemplul actualizat de pe cs.curs)

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.

ATENŢIE! Deadline-ul va fi atât soft, cât şi hard, neacceptându-se arhivele trimise ulterior. Deadline-ul de 04.01.2019 este hard.

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