LUCRAREA 5

Analiza lexicală folosind JLex

Cuprinsul lucrării:

- Generatori de analizori lexicali. JLex[1].
- Interfaţa JLex.
- Metodologia de lucru.
- Generarea de analizori lexicali folosind JLex.

Unele exemple şi figuri din acest capitol au fost preluate şi adaptate din [5] şi [11].

5.1 Generatori de analizori lexicali. JLex

Un analizor lexical (*lexer* în limba engleză) împarte un şir de caractere de intrare în atomi lexicali. Procesul de elaborare a unui astfel de analizor poate fi extrem de dificil dacă este realizat manual, astfel încât au fost concepute instrumente software dedicate pentru acest lucru, care poartă numele de generatoare de analizori lexicali.

Cel mai cunoscut astfel de analizor este Lex [6]. Acesta este un instrument software ce rulează sub Linux şi este destinat a lucra în limbajul de programare C. Primeşte ca intrare un fişier care conţine specificaţiile analizorului lexical şi crează un fişier sursă C asociat cu acel analizor.

[1] `https://www.cs.princeton.edu/~appel/modern/java/JLex/`

JLex este un program care se bazează pe Lex şi care generează codul Java corespunzător unui lexer.

Metodologia de lucru este prezentată în Figura 5-1. Se proiectează specificaţiile lexer-ului conform cerinţelor într-un fişier cu extensia `.jlex`. Acest fişier este procesat de generatorul JLex (clasa `JLex.Main`), care pe baza specificaţiilor va genera codul sursă (Java) al lexer-ului propriu zis. Acest cod est mai apoi compilat folosind `javac` pentru a obţine fişierul de tip `.class` al lexerului, care este ulterior instanţiat şi folosit în programul propriu zis. În final, lexer-ul astfel generat va produce, pe baza şirului de caractere primit la intrare, un flux de atomi lexicali recunoscuţi.

5.2 Configurarea mediului de lucru

(a) Verificaţi că Java este instalat (comenzile `java` şi `javac` funcţionează într-un terminal/comand prompt)

(b) Creaţi în spaţiul de lucru un director `lab4` şi în acesta încă un director `lab4/JLex`

(c) Descărcaţi clasa `Main.java` de la adresa: `https://www.cs.princeton.edu/~appel/modern/java/JLex/current/Main.java`https://www.cs.princeton.edu/ a l/modern/java/JLex/current/Main.java în directorul lab4/JLex

(d) Compilaţi Main.java ca o clasă Java obişnuită:
`javac Main.java`
În urma acestei comenzi ar trebui să fie generate mai multe fişiere de tip clase Java, printre care şi Main.class

(e) Urmăriţi în continuare laboratorul. Fişierele sursă le veţi scrie în directorul `lab4/`

5.3 Interfaţa JLex

Un fişier sursă JLex are de regulă extensia `.jlex`, iar codul este structurat în trei părţi despărţite prin caracterele %%. Un exemplu simplist de fişier sursă JLex este prezentat în Codul sursă 5.1 .

Codul sursă 5.1: Exemplu de fişier sursă JLex.

```
%%
%integer
```

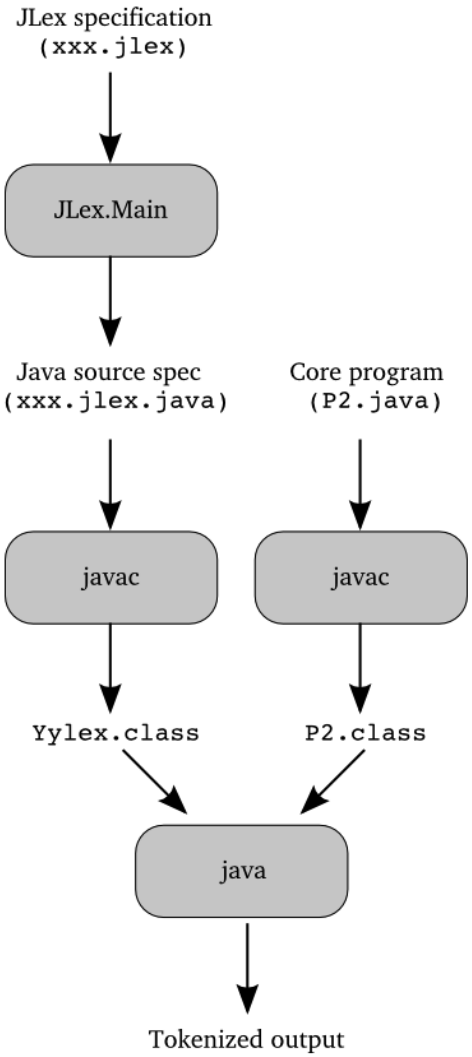

Figura 5-1: Metodologia de lucru JLex.

```
numbers = 0|[1-9][0-9]*
whitespace = [\ \t\n\r]+
%%
{whitespace} { System.out.println("S-a gasit un spatiu:");
    }
{digits} { return new Token(INT, Integer.parseInt(yytext())
    ; }
. { System.out.println("ERROR!"); }
```

Prima secţiune conţine cod Java care este copiat direct, fără a fi executat în prealabil, în clasa Java Yylex care este numele lexer-ului. În cazul exemplului de mai sus, această secţiune este goală. A doua secţiune cuprinde diferite definţii şi directive JLex. Directivele sunt precedate întotdeauna de caracterul %. În exemplul dat, secţiunea conţine o directivă şi trei definiţii.

Directiva `%integer` specifică faptul că atomii care sunt returnaţi în urma analizei lexicale vor fi de tip întreg(`int`). Specificarea acestei directive mai conduce la definirea de către JLex a constantei `YYEOF = -1` în clasa Yylex, constantă ce va fi returnată în mod automat de Yylex la întâlnirea sfârşitului de fişier (end of file).

Cele două definiţii (`numbers` şi `whitespace`) definesc valori numerice şi respectiv spaţii folosind expresii regulate. Astfel, `numbers = 0|[1-9][0-9]*` defineşte un număr ca fiind fie cifra 0, fie orice succesiune de oricâte cifre, dar măcar una, din care prima este obligatoriu diferită de 0. A doua definiţie, `whitespace = [\ \t\n\r]+`, include caracterele de tip spaţiu, tab şi sfârşit de linie.

A treia secţiune din fişierul .lex specifică expresiile regulate pentru care se caută potriviri şi acţiunile specifice în fiecare caz. În exemplul dat, expresiile regulate vizează spaţiile (`whitespace`) şi numerele (`numbers`). În cazul identificării de spaţii, se afişează un mesaj corespunzător care anunţă acest lucru dar nu se execută alte acţiuni. La identificarea de valori numerice, nu se afişează nimic, în schimb se returnează un atom (Token) cu valoarea numerică identificată pe baza expresiei regulate. Se observă caracterul punct (.) aflat pe ultima linie din fişierul .lex. Acest caracter cuprinde orice caracter cu excepţia unei noi linii din fişier şi este utilizat ultimul pentru a semnala printr-un mesaj de eroare faptul că s-a găsit un caracter ce nu a putut fi potrivit cu expresiile regulate specificate mai sus.

Trebuie reţinut faptul că în ultima secţiune din fişierul .lex contează ordinea în care sunt specificate expresiile regulate, această ordine marcând prioritatea cu care sunt potrivite caracterele din fişier. Dacă un şir de caractere se potriveşte cu mai multe expresii regulate, JLex alege expresia aflată „mai sus“ în listă, deci cu prioritate mai mare.

5.4 Generarea de analizori lexicali folosind JLex

5.4.1 Generarea unui scanner de spaţii

Pentru început se va implementa un lexer care va scana spaţiile dintr-un fişier text (Codul sursă 5.2).

Codul sursă 5.2: Codul sursă al unui scanner de spaţii.

```
class IntLit{
}
%%
WHITESPACE = [\n\t\040]
%type IntLit
%function next_token
%%
{WHITESPACE}+ { System.out.println("Am gasit un spatiu!");
   }
```

În a doua secţiune a fişierului .jlex sunt incluse o definiţie şi două directive. Definiţia `WHITESPACE = [\n\t\040]` cuprinde toate caracterele de tip linie nouă, tab, sau spaţiu (definit prin caracterul `\040`). Directiva următoare (`%IntLit`) specifică faptul că lexer-ul va returna atomi lexicali de tip `IntLit`. Acest tip de atom trebuie definit în prima secţiune a fişierului jlex (lucru realizat prin declaraţia `class IntLit {}`).

Prin directiva `%function next_token()` este specificat numele metodei care va fi generată şi folosită pentru a returna atomi lexicali.

A treia secţiune specifică, folosind expresii regulate şi definiţiile anterioare, ce şiruri de caractere vor fi găsite de lexer. Astfel, expresia `{WHITESPACE}+` va găsi toate şirurile de cel puţin un spaţiu, iar pentru fiecare şir găsit se va executa acţiunea definită în continuare `{ System.out.println("Am gasit spatii!"); }`.

JLex procesează specificaţiile din fişierul .jlex şi le transformă în codul Java care implementează propriu-zis scannerul definit în mod abstract prin aceste specificaţii.

Pentru aceasta, la momentul compilării, compilatorului de Java trebuie să i se specifice locaţia pachetului Jlex `Main.class`. Spre exemplu, dacă directorul de lucru în care se găsesc fişierele .lex este `lab_jlex`, iar sursele JLex se găsec în `../lab_jlex/JLex`, pentru a compila sursa comanda este următoarea:

```
java JLex.Main whitespace.jlex
```

Dacă nu au fost erori, rezultatul ar trebui să fie similar cu cel de mai jos:

```
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 9 states.
```

```
Working on character classes.::
NFA has 3 distinct character classes.
Creating DFA transition table.
Working on DFA states....
Minimizing DFA transition table.
3 states after removal of redundant states.
Outputting lexical analyzer code.
```

Se pot observa din mesajele returnate de compilator următoarele:

1. Procesarea secvenţială a celor trei secţiuni ale codului sursă .jlex descrise anterior în laborator: cod utilizator, declaraţii JLex şi reguli lexicale.

2. Metodologia de lucru este următoarea: pe baza specificaţiilor introduse, JLex construieşte un AFN, care este mai apoi convertit la un AFD şi în final automatul rezultat este minimizat.

Puteţi vizualiza codul Java produs de JLex pe baza specificaţiilor, deschizând într-un editor de text fişierul `whitespace.jlex.java`. Studiaţi fişierul:

1. Observaţi diferenţa între specificaţiile minimale din fişierul .jlex şi dimensiunea şi complexitatea codului rezultat

2. Fişierul defineşte două clase: `IntLit` (tipul de atom definit de către noi) şi `Yylex` („motorul“ propriu-zis al lexer-ului)

Acest fişier Java generat trebuie compilat:
`javac whitespace.jlex.java`
Compilarea va produce două fişiere de tip clasă Java, corespunzătoare claselor identificate mai sus: `IntLit.class` şi `Yylex.class`. Aceste clase nu pot fi utilizate pur şi simplu. Este nevoie de scrierea unui program Java în care acestea sunt instanţiate şi folosite corespunzător. Acest lucru este realizat în fişierul `Driver.java`, conform Codului sursă 5.3.

Codul sursă 5.3: Driver.java

```
import java.io.*;
public class Driver{
    public static void main(String[] args) throws Exception
        {
         File f = new File("in.txt");
         FileInputStream fis = new FileInputStream (f);

         Yylex scanner = new Yylex(fis);
```

```
        do{
            scanner.next_token();
        } while(true);
    }
}
```

Programul deschide fişierul `in.txt` ca obiect de tip `InputStream` deoarece acesta este tipul care trebuie furnizat lexerului. În continuare, sunt citiţi atomi lexicali din fişier într-o buclă infinită. Citirea următorului atom se realizează prin apelul funcţiei `scanner.next_token()`.

Pasul următor îl reprezintă compilarea fişierului sursă Driver.java:

```
javac Driver.java
```

Compilarea a generat fişierul `Driver.class` care poate fi lansat în execuţie. Înainte de a face acest lucru, trebuie creat fişierul `in.txt`. Creaţi acest fişier în care introduceţi spaţii de toate tipurile (linie nouă, tab, spaţiu). Apoi se poate executa clasa `Driver` prin comanda:

```
java Driver
```

Programul ar trebui să afişeze un mesaj de tipul:

```
Am gasit un spatiu!
```

Exerciţii:

1. De ce programul rămâne blocat în execuţie?
2. De ce este identificat un singur atom lexical şi în consecinţă afişat un singur mesaj deşi fişierul conţine mai multe spaţii?
3. Modificaţi fişierul `in.txt` introducând şi alte caractere în afară de spaţii. Rulaţi din nou `Driver`. Observaţi rezultatul.
4. Modificaţi fişierul `whitespace.jlex` astfel încât, în condiţiile de la punctul 3, programul să nu îşi mai termine execuţia printr-o eroare ci să afişeze un mesaj corespunzător.

5.4.2 Generarea unui scanner care însumează valorile numerice

În cele ce urmează vom modifica fişierul `whitespace.jlex` pentru a recunoaşte un flux de valori numerice întregi, conform Codului sursă 5.4. Vom redenumi noul fişier ca `sum.jlex`.

Codul sursă 5.4: sum.jlex

```
class IntLit{
```

```
    private int value;
    public IntLit(String textIn) {
        value = Integer.parseInt(textIn);
    }
    public String toString() {
        return "Value: " + value;
    }
    public int valueGet(){
        return value;
    }
}
%%
WHITESPACE = [\n\t\040]
DIGIT = [0-9]
not_star = [^*];
%type IntLit
%function next_token

%%
{DIGIT}+ { System.out.println("Am gasit o valoare numerica"
   );
     return new IntLit(yytext()); }
{WHITESPACE}+ { System.out.println("Am gasit un spatiu!");
   }
{not_star} { System.out.println("not star"); }
. { System.out.println("ERROR!"); }
```

Faţă de codul din `whitespace.jlex`, observăm o serie de modificări. În a doua secţiune a fost introdusă o nouă definiţie care cuprinde cifrele `DIGIT = [0-9]`. Regula lexicală pe baza acestei definiţii, ce se regăseşte în secţiunea a treia a fişierului, specifică acţiunea ce se execută în momentul întâlnirii unui şir de unul sau mai multe cifre (`{DIGIT}+`). În acest caz, lexer-ul va afişa un mesaj corespunzător şi va returna valoarea găsită (prin intermediul funcţiei `yytext()`, o metodă din clasa `Yylex`). De asemenea, a mai fost introdusă o definiţie `not_star`, care pe baza expresiei regulate `[^*]` va găsi toate caracterele diferite de caracterul *. Pentru această definiţie, a fost adăugată şi o regulă lexicală care afişează un mesaj corespunzător la găsirea unui caracter care se potriveşte cu expresia regulată.

Pentru a putea returna valoarea numerelor găsite, trebuie modificat corespunzător şi constructorul clasei atomului IntLit. Astfel, au fost implementate:

- data membră `value` (de tip `private int`)
- constructorul `public IntLit(String textIn)` care iniţializează data `value` cu valoarea primită ca parametru

- metoda `public String toString()` care permite afişarea datei membre `value` la afişarea unui atom

- o metodă `public int valueGet()` care permite citirea datei membre private `value`

În conformitate cu modificările aduse fişierului sursă `sum.jlex`, trebuie modificat şi programul care instanţiază şi utilizează lexer-ul generat pe baza specificaţiilor. Astfel, se va scrie un nou fişier `Driver2.java` conform Codului sursă 5.5.

Codul sursă 5.5: Driver2.java

```
import java.io.*;

publi class Driver2{
    public static void main(String[] args) throws Exception
       {
        File f = new File("in2.txt");
        FileInputStream fis = new FileInputStream (f);

        Yylex scanner = new Yylex(fis);
        int sum = 0;
        do{
            IntLit token = scanner.next_token();
            System.out.println(token);
            if (token == null) {
                break;
            }
            sum = sum + token.valueGet();
        } while(true);

        System.out.println("Suma este " + sum);
    }
}
```

Modificările aduse fişierului `Driver2.java` faţă de versiunea corespunzătoare parserului de spaţii implementează următoarele funcţionalităţi:

- fiecare atom lexical recunoscut de analizor este scris într-un obiect de tip `IntLit` (`IntLit token = scanner.next_token()`)

- valoarea atomului este afişată (`System.out.println(token)`)

- se verifică dacă atomul lexical are valoarea `null`, caz în care se opreşte execuţia programului (cu instrucţiunea `break` se iese din bucla `while`)

- valorile atomilor lexicali sunt adunaţi la variabila `sum`, care este afişată pe ecran înainte de terminarea execuţiei programului

Exerciţii:

1. Executaţi fişierul `sum.jlex` folosind JLex şi compilaţi fişierul java rezultat.

2. Scrieţi un fişier `in2.txt` care să conţină spaţii, valori numerice, caractere oarecare precum şi caractere *.

3. Compilaţi şi executaţi `Driver2.java`. Observaţi ce afişează programul în funcţie de fişierul `in2.txt` introdus.

4. Ce se întâmplă dacă nu includem în `Driver2.java` porţiunea de cod cu ajutorul căreia se iese din buclă la întâlnirea unui atom nul?

5. Ce se întâmplă dacă se inversează ultimele două linii din fişierul `sum.jlex`. Argumentaţi răspunsul.

5.4.3 Specificarea de stări şi tranziţii în fişierul JLex

În fişierul sursă `.jlex` se pot specifica stări precum şi tranziţiile dintre ele, ceea ce reflectă faptul că lexer-ul rulează practic un AFD. Vom modifica fişierul `sum.jlex` de mai sus în felul următor:

1. Se adaugă o nouă directivă `%state negative`, care va defini o nouă stare cu denumirea ,,negative“

2. Se adaugă o nouă regulă lexicală (antepenultima, înainte de regulile {not_star} şi .) care va conduce în starea negative definită la întâlnirea şirului de caractere ,,NEG“. Tranziţia într-o altă stare se realizează cu metoda `yybegin()` a clasei `Yylex`, ce primeşte ca parametru numele stării definite ca directivă `NEG { yybegin(negative); }`

3. Regulile lexicale se vor modifica astfel încât, odată intrat în starea `negative`, toate valorile numerice întâlnite vor fi considerate a fi negative, iar la întâlnirea unui nou şir de caractere ,,NEG“ se iese din starea `negative` şi se intră în starea implicită `YYINITIAL`

În consecinţă, secţiunea a treia a noului fişier .jlex va arăta conform Codului sursă 5.6.

Codul sursă 5.6: Secţiunea a treia din noul fişier sum.jlex.

```
<negative>{DIGIT}+ { System.out.println("Am gasit o valoare
    numerica negativa!");
                     yybegin(YYINITIAL);
                    return new IntLit("-" + yytext()); }
{DIGIT}+ { System.out.println("Am gasit o valoare numerica
   pozitiva!");
            return new IntLit(yytext()); }
{WHITESPACE}+ { System.out.println("Am gasit un spatiu!");
   }
<negative>"NEG" { yybegin(YYINITIAL); }
"NEG" { yybegin(negative); }
{not_star} { System.out.println("not star"); }
. { System.out.println("ERROR!"); }
```

Modificaţi fişierul `in2.txt` pentru a arăta ca în exemplul de mai jos:

```
NEGNEG
1
NEG
NEG3
42 NEG 20
a b c *
```

Exerciţii:

1. Executaţi fişierul `sum.jlex` folosind JLex şi compilaţi fişierul java rezultat.
2. Executaţi `Driver2.java`. Observaţi ce afişează în funcţie de fişierul `in2.txt` introdus. Modificaţi fişierul `in2.txt` şi analizaţi ce efecte au loc asupra mesajelor afişate de lexer.

5.5 Temă de laborator

Presupunem că, într-un cod Java, există o funcţie denumită `myFunction` a cărei execuţie consumă foarte mult timp, deci într-un program există limitări a numărului de apeluri datorită cerinţelor de performanţă.

Proiectaţi specificaţiile JLex care generează un scanner ce numără apelurile acestei funcţii dintr-un cod Java, urmând cerinţele de mai jos:

- Scannerul trebuie să afişeze un mesaj de fiecare dată când întâlneşte un apel la funcţia `myFunction` însoţit de numărul liniei din cod (folosind yyline).

- După terminarea de parcurs a codului, se va afişa un mesaj cu numărul total de apeluri ale funcţiei `myFunction`.
- Codul sursă de intrare este scris în Java.
- În cod pot exista şi alte variabile denumite `myFunction` sau alte funcţii care au în denumirea lor `myFunction`.
- Cu toate acestea, definirea (declararea, antetul) funcţiei `myFunction` nu este prezentă în codul de intrare, doar apeluri la această funcţie
- Nu există constrângeri cu privire la tipul returnat de `myFunction`, deci scannerul implementat va fi unul generic, acceptând orice tip returnat de această funcţie (pentru a simplifica implementarea)