ECOLE POLYTECHNIQUE DE LOUVAIN

LINGI2132 - LANGUAGES AND TRANSLATORS

Assignement 2 - Report

Professor :

Pierre SCHAUS

Students :

Benoît BAUFAYS	22200900
Julien COLMONTS	41630800

Program :

SINF21MS

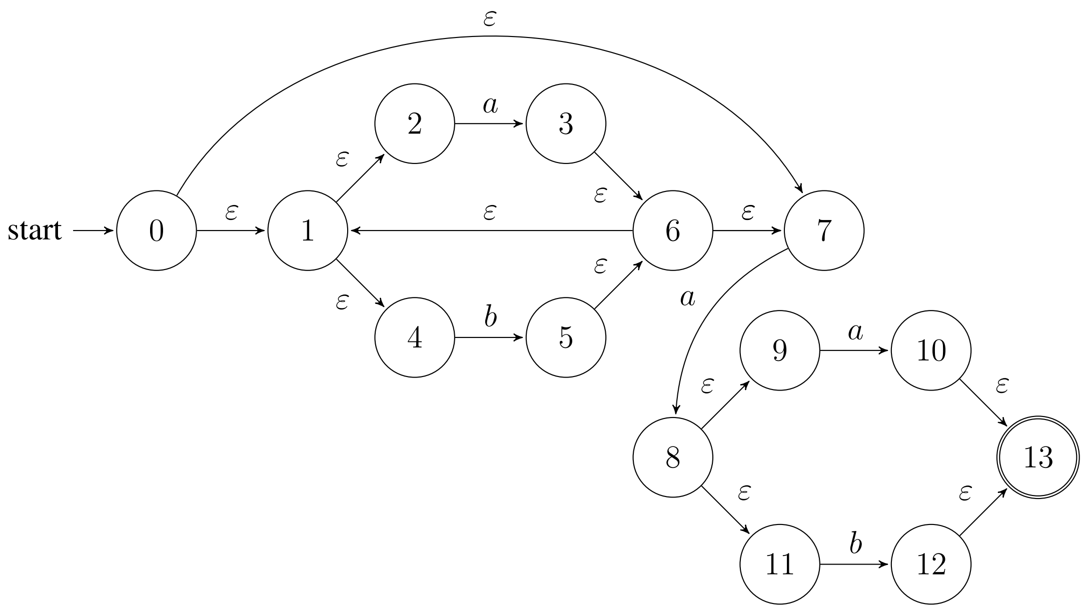
Academic Year 2013-2014

1 Lexical Analysis

1.1 Give 5 different strings belonging to the language described by this regExp.

1. aab
2. bbaa
3. ababab
4. aaaaaaa
5. bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbab

1.2 Construct the NFA from this regExp (Thompson Construction



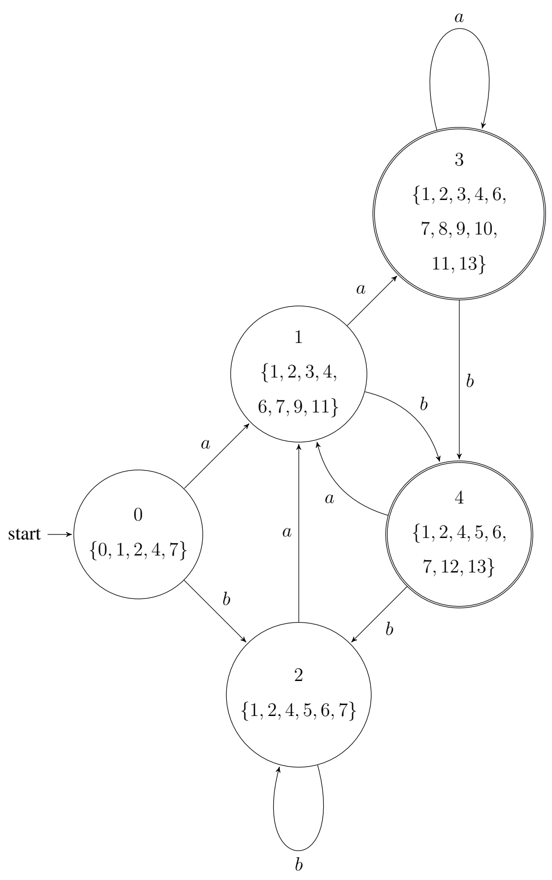
1.3 Transform the NFA into a DFA (justify important steps, ε-closures).

Steps from NFA to DFA :

- $s_0 = \epsilon - closure(\{0\}) = \{0, 1, 2, 4, 7\}$
- $m(s_0, a) = s_1$, where $s_1 = \epsilon - closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 9, 11\}$

- $m(s_0, b) = s_2$, where $s_2 = \epsilon - closure(\{5\}) = \{1, 2, 4, 5, 6, 7\}$
- $m(s_1, a) = s_3$, where $s_3 = \epsilon - closure(\{3, 8, 10\}) = \{1, 2, 3, 4, 6, 7, 8, 9, 11, 13\}$
- $m(s_1, b) = s_4$, where $s_4 = \epsilon - closure(\{5, 12\}) = \{1, 2, 4, 5, 6, 7, 12, 13\}$
- $m(s_2, a) = s_1$
- $m(s_2, b) = s_2$
- $m(s_3, a) = s_3$
- $m(s_3, b) = s_4$
- $m(s_4, a) = s_1$
- $m(s_4, b) = s_2$

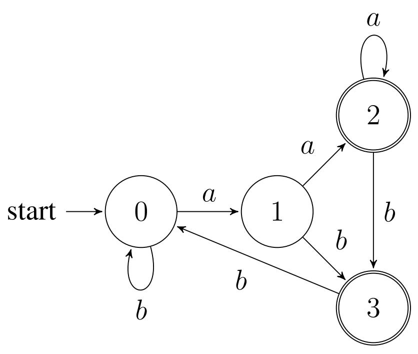
Since s_3 and s_4 contain state 13 which was final in the NFA, they are both final state too.



1.4 Minimize the DFA (Hopcroft Algorithm).

The DFA could be divided in four partitions :

- Two partitions for final states
- A partition which contains previous states $\{0, 2\}$ because :
  - $m(0, a) = 1$
  - $m(2, a) = 1$
  - $m(0, b) = 2$
  - $m(2, b) = 2$
- A partition which contains previous state $\{1\}$:
  - $m(1, a) = 3$
  - $m(1, b) = 4$



2 Parsing

2.1 LL(1) Grammar

This grammar is not LL(1) because it has a rule with left recursion: $B := Bv$ and $B := w$.

2.2 Modify the grammar

We replaced the left recursion with a right recursion to avoid the problem indicated in the first exercise.

$$B := wB'$$
$$B' := vB'$$
$$B' := \epsilon$$

Thus, we removed from the grammar the rules $B := Bv$ and $B := w$ and we introduced rules above.

2.3 Sets for this grammar

$first(u) = \{u\}$

$first(v) = \{v\}$

$first(w) = \{w\}$

$first(x) = \{x\}$

$first(y) = \{y\}$

$first(z) = \{z\}$

$\epsilon \in first(E), first(F), first(B')$

Rule 1:$S :== uBDz$

$first(uBDz) = \{u\}$

$\rightarrow u \in first(S)$

Rule 2: $B ::= wB'$

$first(wB') = \{w\}$

$\rightarrow w \in first(B)$

Rule 3: $B' ::= wB'$

$first(vB') = \{v\}$

$\rightarrow v \in first(B')$

Rule 5: $D ::= EF$

$first(EF) = first(E) \backslash \{\epsilon\} \cup first(F)$ (car $E \in first(E)$)= $\{x, y, E\}$

$\rightarrow x, y, E \subset first(D)$

Rule 6: $E ::= y$

$y \in first(E)$

Rule 8: $F ::= x$

$x \in first(F)$

$first(S) = \{u\}$

$first(B) = \{w\}$

$first(B') = \{v, \epsilon\}$

$first(D) = \{x, y, \epsilon\}$

$first(E) = \{y, \epsilon\}$

$first(F) = \{x, \epsilon\}$

Follow

$\#E\, follow(S)$

Rule 1: $S ::= uBDz$

$\rightarrow \{x, y, z\} \subset Bfollow(B)$

$and\, z \in follow(D)$

Rule 2: $B ::= wB'$

$\rightarrow follow(B) = \{x, y, z\} \subset follow(B')$

Rule 5: $D ::= EF$

$\rightarrow x \in follow(E)$

$follow(D) = \{z\} \subset follow(E)$

$and\, follow(D) = \{z\} \subset follow(F)$

$follow(S) = \{\#\}$

$follow(B) = \{x, y, z\}$

$follow(B') = \{x, y, z\}$

$follow(D) = \{z\}$

$follow(E) = \{x, z\}$

$follow(F) = \{z\}$

2.4 Parsing Table

Rule 1: $S ::= uBDz$

$\rightarrow table[S, u] = 1$

Rule 2: $B ::= wB'$

$\rightarrow table[B, w] = 2$

Rule 3: $B' ::= vB'$

$\rightarrow table[B', v] = 3$

Rule 4: $B' ::= \epsilon$

$\rightarrow table[B', x] = 4$

$\rightarrow table[B', y] = 4$

$\rightarrow table[B', z] = 4$

Rule 5: $D ::= EF$

$\rightarrow table[D, x] = 5$

$\rightarrow table[D, y] = 5$

$\rightarrow table[D, z] = 5$

Rule 6: $E ::= y$

$\rightarrow table[E, y] = 6$

Rule 7: $E ::= \epsilon$

$\rightarrow table[E, x] = 7$

$\rightarrow talbe[E, z] = 7$

Rule 8: $F ::= x$

$\rightarrow table[F, x] = 8$

Rule 9: $F ::= \epsilon$

$\rightarrow table[F, z] = 9$

The final parsing table is :

	u	v	w	x	y	z
S	1					
B			2			
B'		3		4	4	4
D				5	5	5
E				7	6	7
F				8		9

3 Programming Part

3.1 Recursive Descent

We created a parsing method for each non-terminal symbol. In each of these methods, we checked if the current position was respecting the given rule and then recursively parsed next positions. After the `end` token, we checked if we reached the end of the input string. Finally, we wrote some additional tests in another JUnit Java class.

3.2 Programming directly in Java bytecode

Following the objective of writing the `generateClass`, we first had to understand how a compiler build a stack. To check if our implementation using the `CLEmitter` class was correct, we used the `javap` command on a class file compiled with `javac`. Even if the real java compiler adds more informations (Line Numbers informations, a Stack Map Table,etc.), we could observe if our way of fill the stack was correct. Finally, after launched the ant build to compile our work, we tried directly if we could execute the generated class with some random integers parameters.

3.3 J

Lexical Analysis

`Handwritten compiler`

For this part, we modified the behaviour of the Scanner class when it was scanning the next token. When a slash character was read, a star character or another slash could follow. If it was going in a multi-line comment, the scanner had to wait for a star followed by a slash string and ignore everything else. To prevent errors, we had to check on all ignored characters if it wasn't parsing a end of file character.

`Generated Compiler`

We modified the j–.jj file and added the following rules :

```
MORE: { "/*": IN_MULTI_LINE_COMMENT }
<IN_MULTI_LINE_COMMENT>
SPECIAL_TOKEN: {
<MULTI_LINE_COMMENT: "*/" > : DEFAULT }
<IN_MULTI_LINE_COMMENT>
MORE: { < ~[] > }
```

These rules allow to skip all the content between comment symbols.

Parsing

1. `Conditional Expression`

We added two new tokens which are the question mark and the colons. We modified the parser to make it able to understand the meaning of these tokens. The line in java that corresponds to these tokens must be divided in three parts : first, a conditional expression followed by a question mark that will introduce two expression split around the colons. We had to create a new kind of expression which is `JConditionalExpression`. It corresponds to the AST node of this conditional expression. To support multiple type returns, we don't check if the same type is returned in all cases. As a choice of implementation, we force the final type of representation to be the one in the THEN part of the conditional expression.

In the j–.jj file, we added the corresponding tokens and we modified the assignment expression to make it able to use the new conditional expression method written underneath.

2. `For loop`

There are two parts for the for loop implementation: first the basic one, containing an iteration, and an enhanced one, iterating on an array. We modified the statement method in the parser class to observe the for loop in the parsed code. We created the JForInitExpression class to manage both kind of loops initialisation which is an abstract class. For each kind of loop, we have a specific which extends the standard initialisation. When it's needed to declare a variable, like in the basic for loop, we use the JForInitVarDeclaration class. Otherwise, we use the JForInitStatement class.

The AST node representation of the basic for loop is implemented via the JBasicForStatement class. It is used to analyse and generate the node. The representation of the enhanced for loop is declared in the JEnhancedForStatement class. The behaviour of this class methods uses to build the representation of the enhanced for loop, making it corresponding to a basic for loop. This could be interpreted as syntaxic sugar to help programmers to work more efficiently.