

CS-320

Computer Language Processing

Exercise Session 2

October 2, 2019

Overview

Today we will have a deeper look at lexers.

What does a lexer (*aka.* tokenizer) do?

How can we automatically generate lexers?

Overview

Today we will have a deeper look at lexers.

What does a lexer (*aka.* tokenizer) do?

⇒ It transforms a stream of symbols into a stream of tokens.

How can we automatically generate lexers?

Overview

Today we will have a deeper look at lexers.

What does a lexer (*aka.* tokenizer) do?

⇒ It transforms a stream of symbols into a stream of tokens.

How can we automatically generate lexers?

⇒ We will use regular languages and automata.

Lexer

First, let us define a lexer as an ordered set of token names and regular expressions

$$\langle \textit{Token}_1 := e_1, \textit{Token}_2 := e_2, \dots \rangle$$

where earlier token classes have higher priority than later ones.

E.g.

$$\langle \textit{ID} := \text{letter} (\text{letter} \mid \text{digit})^*, \textit{LE} := \leq, \textit{LT} := <, \textit{EQ} := = \rangle$$

Ambiguity in tokenization

Tokenization differs from matching using a single regular expression (say $(e_1 \mid e_2 \mid \dots)^*$).

Rather, the result of tokenizing an input stream of symbols is a stream of tokens. Each token maps to a subsequence of the input stream, and none of the tokens' subsequences overlap.

E.g.

`i0 <= size` $\xRightarrow{\text{tokenize}}$ `ID LE ID`
`i0 <= size`

Ambiguity in tokenization

Tokenization differs from matching using a single regular expression (say $(e_1 \mid e_2 \mid \dots)^*$).

Rather, the result of tokenizing an input stream of symbols is a stream of tokens. Each token maps to a subsequence of the input stream, and none of the tokens' subsequences overlap.

E.g.

$$i0 \leq size \xrightarrow{\text{tokenize}} \begin{array}{ccc} ID & LE & ID \\ i0 & \leq & size \end{array}$$

▷ How do we avoid ambiguities?

$$i0 \leq size \xrightarrow{???} \begin{array}{cccccc} ID & LT & EQ & ID & ID & ID \\ i0 & < & = & s & iz & e \end{array}$$

Tokenization rules

Given an input string w the lexer will match tokens on a prefix u of $w = uv$, output the matching token and repeat the process on the remaining string v .

To disambiguate between different possible tokenizations we employ two additional rules:

- ▶ *Longest match*: If we find matching tokens for prefixes of varying lengths, we pick the longer prefix.
- ▶ *Token priority*: If multiple tokens match a prefix of the same length, we pick the token that has higher priority.

A simple tokenization example

Exercise 1

▷ Given the lexer

$$\langle T_1 := a(ab)^*, T_2 := b^*(ac)^*, T_3 := cba, T_4 := c^+ \rangle$$

tokenize the following input strings:

c a c c a b a c a c c b a b c

A simple tokenization example

Exercise 1

▷ Given the lexer

$$\langle T_1 := a(ab)^*, T_2 := b^*(ac)^*, T_3 := cba, T_4 := c^+ \rangle$$

tokenize the following input strings:

c a c c a b a c a c c b a b c

c c c a a b a b a c c b a b c c b a b a c

A simple tokenization example

Exercise 1

▷ Given the lexer

$$\langle T_1 := a(ab)^*, T_2 := b^*(ac)^*, T_3 := cba, T_4 := c^+ \rangle$$

tokenize the following input strings:

$\underbrace{c}_{T_4} \underbrace{ac}_{T_2} \underbrace{c}_{T_4} \underbrace{a}_{T_1} \underbrace{bacac}_{T_2} \underbrace{cba}_{T_3} \underbrace{b}_{T_2} \underbrace{c}_{T_4}$

c c c a a b a b a c c b a b c c b a b a c

A simple tokenization example

Exercise 1

▷ Given the lexer

$$\langle T_1 := a(ab)^*, T_2 := b^*(ac)^*, T_3 := cba, T_4 := c^+ \rangle$$

tokenize the following input strings:

$$\underbrace{c}_{T_4} \underbrace{ac}_{T_2} \underbrace{c}_{T_4} \underbrace{a}_{T_1} \underbrace{bacac}_{T_2} \underbrace{cba}_{T_3} \underbrace{b}_{T_2} \underbrace{c}_{T_4}$$

$$\underbrace{ccc}_{T_4} \underbrace{aabaab}_{T_1} \underbrace{ac}_{T_2} \underbrace{cba}_{T_3} \underbrace{b}_{T_2} \underbrace{cc}_{T_4} \underbrace{b}_{T_2} \underbrace{a}_{T_1} \underbrace{bac}_{T_2}$$

A simple tokenization example

Exercise 1

▷ Given the lexer

$$\langle T_1 := a(ab)^*, T_2 := b^*(ac)^*, T_3 := cba, T_4 := c^+ \rangle$$

tokenize the following input strings:

$$\underbrace{c}_{T_4} \underbrace{ac}_{T_2} \underbrace{c}_{T_4} \underbrace{a}_{T_1} \underbrace{bacac}_{T_2} \underbrace{cba}_{T_3} \underbrace{b}_{T_2} \underbrace{c}_{T_4}$$

$$\underbrace{ccc}_{T_4} \underbrace{aabaab}_{T_1} \underbrace{ac}_{T_2} \underbrace{cba}_{T_3} \underbrace{b}_{T_2} \underbrace{cc}_{T_4} \underbrace{b}_{T_2} \underbrace{a}_{T_1} \underbrace{bac}_{T_2}$$

▷ Are there alternative tokenizations if we disregard the longest match rule?

Constructing a lexer

To automatically construct a lexer from token class definitions we go through a series of transformations:

Token def.s $\xRightarrow{\text{translate}}$ **NFA** $\xRightarrow{\text{determinize}}$ **DFA** $\xRightarrow{\text{minimize}}$ **DFA**

The resulting DFA is then repeatedly used to produce tokens for an input string.

(The minimization step is optional.)

Constructing a lexer (2)

(Token def.s \Rightarrow NFA)

Let e_1, \dots, e_n be the regular expressions for each token class and consider the regular expression $(e_1 \mid \dots \mid e_n)$.

E.g., for the token classes

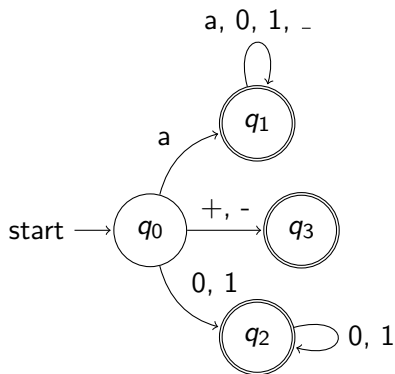
$\langle ID := a(a \mid 0 \mid 1 \mid -)^*, INT := (0 \mid 1)(0 \mid 1)^*, OP := + \mid - \rangle$

we have

$a(a \mid 0 \mid 1 \mid -)^* \mid (0 \mid 1)(0 \mid 1)^* \mid (+ \mid -)$.

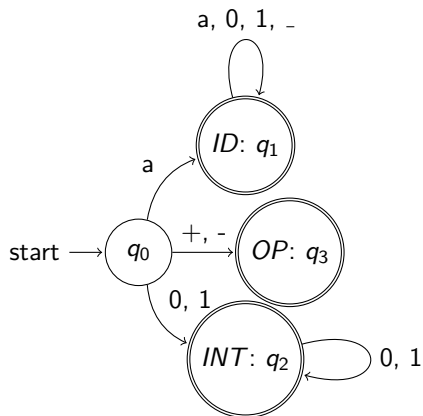
Constructing a lexer (3)

Convert the regular expression to an automaton and specify the token class being recognized by each accepting state:



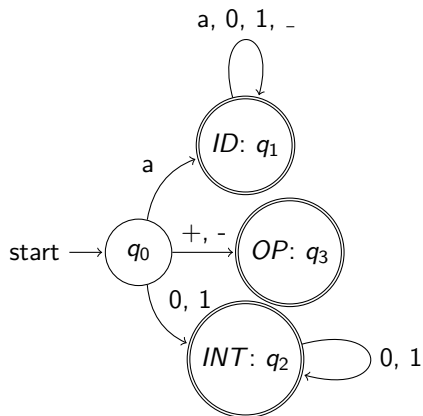
Constructing a lexer (3)

Convert the regular expression to an automaton and specify the token class being recognized by each accepting state:



Constructing a lexer (3)

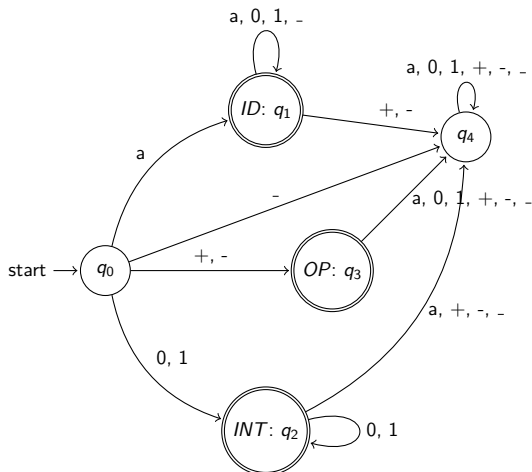
Convert the regular expression to an automaton and specify the token class being recognized by each accepting state:



In case of ambiguities we pick the token class of higher priority.

Constructing a lexer (4)

Finally, we determinize and minimize the automaton:



Lexing

We can then produce tokens for an input string as follows:

1. Initialize variables `lastToken` and `lastTokenPos` to the initial value \perp , resp. -1 , and set the automaton's state to q_0 .
2. Consume the next input character and make the corresponding transition in the automaton.
 - ▶ If we have arrived in an accepting state, update `lastToken` with the corresponding token and set `lastTokenPos` to the current position in the input string.
 - ▶ If we have arrived in a state which cannot lead to acceptance (a trap state, effectively):
 - ▶ If `lastToken` = \perp , report an error.
 - ▶ Otherwise, output `lastToken`.
 - ▶ Reset `lastToken`, and restart the automaton with the input string starting at `lastTokenPos + 1` (Continue with step 2).
3. If there is no more input to consume, output the `lastToken`, or report an error, if `lastToken` = \perp .

Alternate way to build the lexer

Instead of building a single regular expression and converting it to a NFA and then a DFA, we can convert each regular expression to a DFA independently and then build the *parallel composition* of the DFAs as our lexer.

The parallel composition of multiple DFAs $\langle \Sigma, Q_i, s_i, \delta_i, F_i \rangle$ on the same alphabet Σ is the DFA $\langle \Sigma, Q, s, \delta, F \rangle$ where:

- ▶ Q is the cartesian product of all Q_i ,
- ▶ $s \in Q$ is the tuple of all initial states s_i ,
- ▶ $\delta : (Q \times \Sigma) \rightarrow Q$ applies the different δ_i to the respective points, and
- ▶ $F \subseteq Q$ is the set of accepting states. A tuple $q \in Q$ is in F iff there exists a q_i such that $q_i \in F_i$.

Exercise 2

Build a DFA that accepts binary numbers that are multiples of 2 or 3. There should be different (accepting) states for multiples of 2 only, multiples of 3 only, and multiples of 2 and 3.

Hint: Build 2 DFAs, and then use parallel composition.

Exercise 3

Let A be the singleton alphabet containing only the symbol 1. Let L be the language of words over A whose size is a prime number.

$$L = \{ w \in A^* \mid |w| \text{ is prime} \}$$

Prove that L is regular by building a regular expression for L , or prove that L is not regular using the pumping lemma. The pumping lemma states that, for any regular language L , there exists a strictly positive constant number p , such that every word w in L whose length is at least p can be written as $w = xyz$ where:

- ▶ $|y| > 0$
- ▶ $|xy| \leq p$
- ▶ For any number i , we have that xy^iz is in L .