CS-320

Computer Language Processing

Exercise Session 2

October 8, 2018

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?
$\Rightarrow$ 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?
$\Rightarrow$ It transforms a stream of symbols into a stream of tokens.

How can we automatically generate lexers?
$\Rightarrow$ We will use regular languages and automata.

Lexer

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

$$\langle \mathit{Token}_1 := e_1,\ \mathit{Token}_2 := e_2,\ \ldots \rangle$$

where earlier token classes have higher priority than later ones.

E.g.

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

Ambiguity in tokenization

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

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.

$$\texttt{i0 <= size} \quad \overset{tokenize}{\Rightarrow} \quad \underset{\texttt{i0}}{ID} \;\; \underset{\texttt{<=}}{LE} \;\; \underset{\texttt{size}}{ID}$$

Ambiguity in tokenization

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

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.

$$\texttt{i0 <= size} \overset{tokenize}{\Rightarrow} \underset{\texttt{i0}}{ID}\ \underset{\texttt{<=}}{LE}\ \underset{\texttt{size}}{ID}$$

- How do we avoid ambiguities?

$$\texttt{i0 <= size} \overset{???}{\Rightarrow} \underset{\texttt{i0}}{ID}\ \underset{\texttt{<}}{LT}\ \underset{\texttt{=}}{EQ}\ \underset{\texttt{s}}{ID}\ \underset{\texttt{iz}}{ID}\ \underset{\texttt{e}}{ID}$$

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:

$$\underbrace{\texttt{c}}_{T_4}\ \underbrace{\texttt{a c}}_{T_2}\ \underbrace{\texttt{c}}_{T_4}\ \underbrace{\texttt{a}}_{T_1}\ \underbrace{\texttt{b a c a c}}_{T_2}\ \underbrace{\texttt{c b a}}_{T_3}\ \underbrace{\texttt{b}}_{T_2}\ \underbrace{\texttt{c}}_{T_4}$$

$$\texttt{c c c a a b a b a c c b a b c c b a b a c}$$

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:

$$\underbrace{\texttt{c}}_{T_4}\ \underbrace{\texttt{a c}}_{T_2}\ \underbrace{\texttt{c}}_{T_4}\ \underbrace{\texttt{a}}_{T_1}\ \underbrace{\texttt{b a c a c}}_{T_2}\ \underbrace{\texttt{c b a}}_{T_3}\ \underbrace{\texttt{b}}_{T_2}\ \underbrace{\texttt{c}}_{T_4}$$

$$\underbrace{\texttt{c c c}}_{T_4}\ \underbrace{\texttt{a a b a b}}_{T_1}\ \underbrace{\texttt{a c}}_{T_2}\ \underbrace{\texttt{c b a}}_{T_3}\ \underbrace{\texttt{b}}_{T_2}\ \underbrace{\texttt{c c}}_{T_4}\ \underbrace{\texttt{b}}_{T_2}\ \underbrace{\texttt{a}}_{T_1}\ \underbrace{\texttt{b a c}}_{T_2}$$

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:

$$\underbrace{\texttt{c}}_{T_4}\ \underbrace{\texttt{a c}}_{T_2}\ \underbrace{\texttt{c}}_{T_4}\ \underbrace{\texttt{a}}_{T_1}\ \underbrace{\texttt{b a c a c}}_{T_2}\ \underbrace{\texttt{c b a}}_{T_3}\ \underbrace{\texttt{b}}_{T_2}\ \underbrace{\texttt{c}}_{T_4}$$

$$\underbrace{\texttt{c c c}}_{T_4}\ \underbrace{\texttt{a a b a b}}_{T_1}\ \underbrace{\texttt{a c}}_{T_2}\ \underbrace{\texttt{c b a}}_{T_3}\ \underbrace{\texttt{b}}_{T_2}\ \underbrace{\texttt{c c}}_{T_4}\ \underbrace{\texttt{b}}_{T_2}\ \underbrace{\texttt{a}}_{T_1}\ \underbrace{\texttt{b a c}}_{T_2}$$

▷ 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:

$$\textbf{Token def.s} \overset{\textit{translate}}{\Rightarrow} \textbf{NFA} \overset{\textit{determinize}}{\Rightarrow} \textbf{DFA} \overset{\textit{minimize}}{\Rightarrow} \textbf{DFA}$$

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

E.g., for the token classes

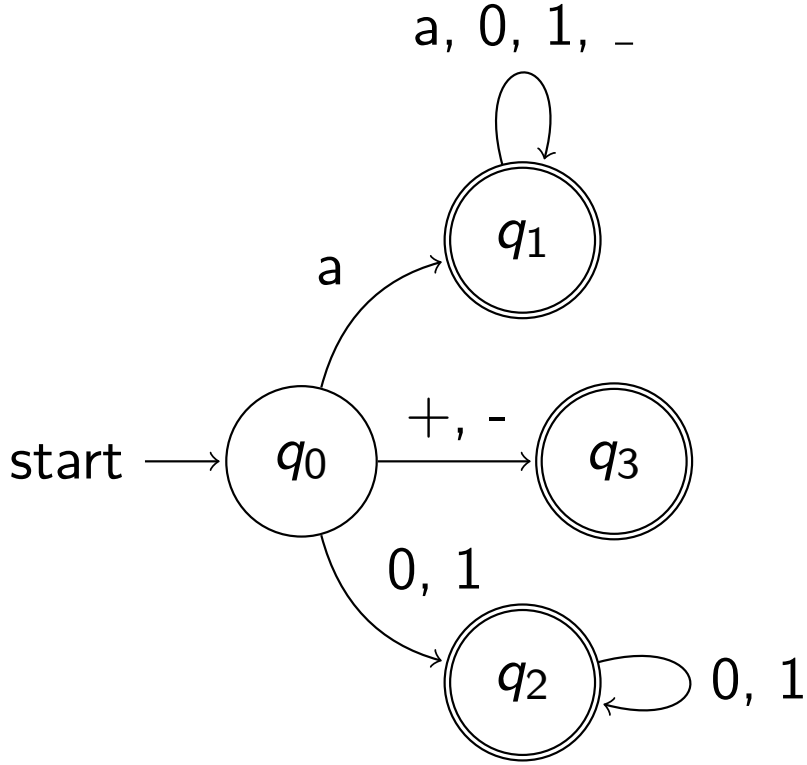
$$\langle ID := a\,(a \mid 0 \mid 1 \mid \_)^*,\ INT := (0 \mid 1)\,(0 \mid 1)^*,\ OP := + \mid - \rangle$$

we have

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

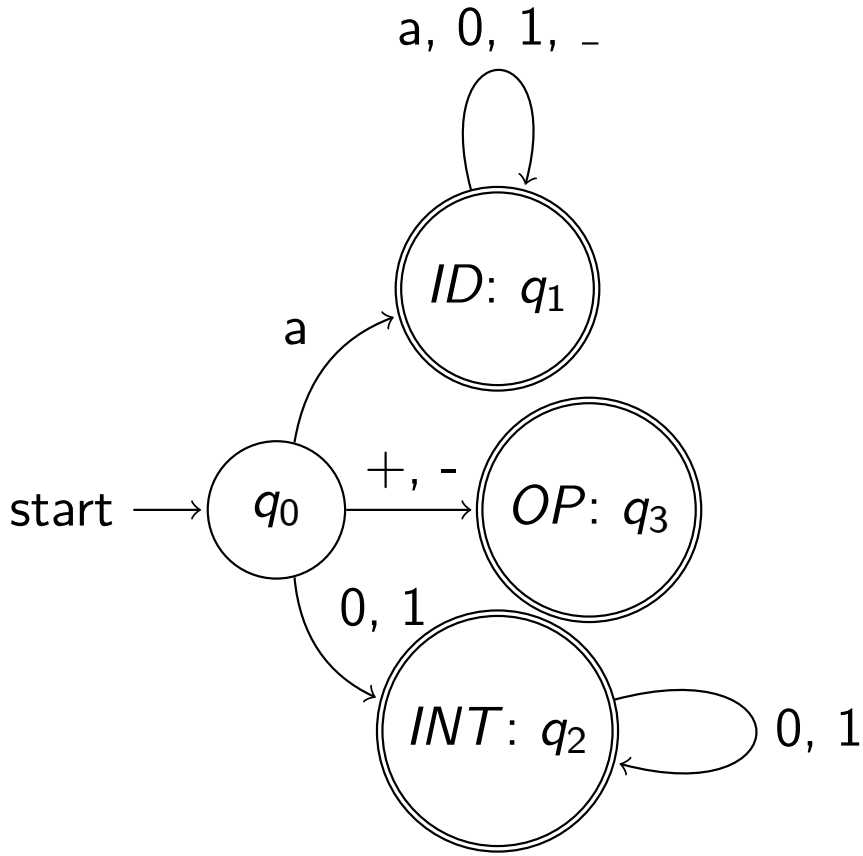
Constructing a lexer (3)

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



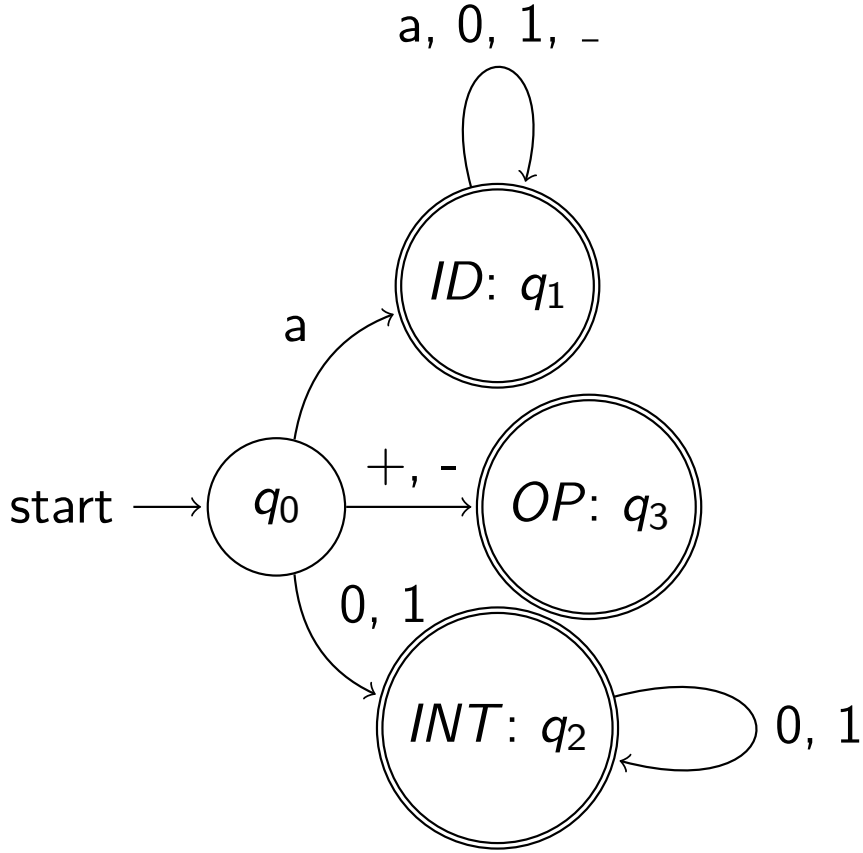
Constructing a lexer (3)

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



Constructing a lexer (3)

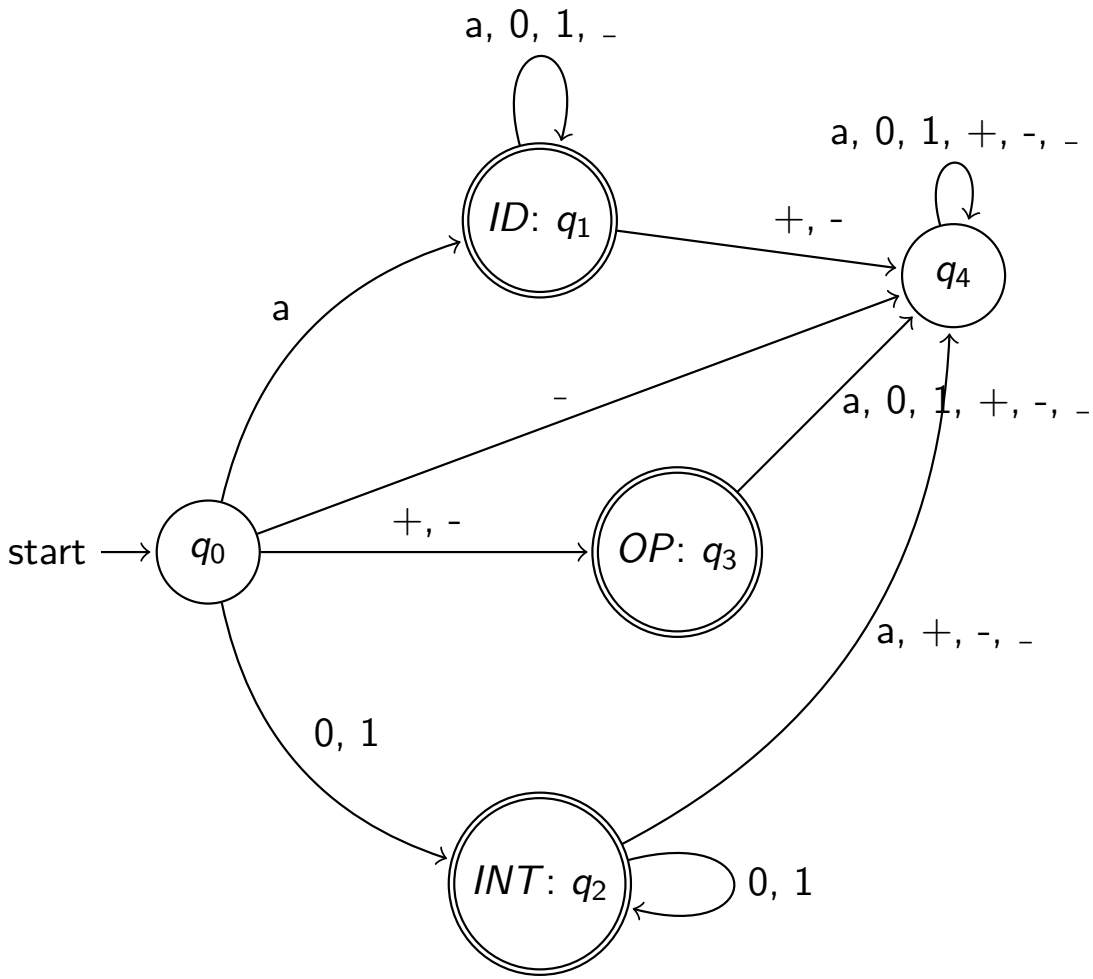
Convert the regular expression to an automaton and specify the token class being recognized by each accepting state:



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 $\bot$, 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` $= \bot$, 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` $= \bot$.

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,
- $\delta : (Q \times \Sigma) \to Q$ 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.

A lexer for XML

Exercise 3 (Quiz 2015)

Your goal is to construct a lexer (i.e., a DFA) that tokenizes an XML input stream into the tokens listed below. Note that *WS* denotes a whitespace character.

Token name	**Regular expression**
OP	$<$
CL	$>$
OPSL	$< /$
CLSL	$/ >$
EQ	$=$
NAME	$\mathit{letter}(\mathit{letter} \mid \mathit{digit})^*$
NONNAME	$(\mathit{digit} \mid \mathit{special})(\mathit{letter} \mid \mathit{digit} \mid \mathit{special})^*$
STRING	$''(\mathit{letter} \mid \mathit{digit} \mid \mathit{special})^*''$
COMMENT	$<! -- (\mathit{letter} \mid \mathit{digit} \mid \mathit{special})^* -- >$
SKIP	*WS*

A lexer for XML (2)

Exercise 3 (Quiz 2015)

▷ Construct the labelled DFA described in the lectures for the tokens defined above. Note that every final state should be labelled by the token class(es) it accepts.

A lexer for XML (2)

Exercise 3 (Quiz 2015)

▷ Construct the labelled DFA described in the lectures for the tokens defined above. Note that every final state should be labelled by the token class(es) it accepts.

Consider the following XML string.

```
<jsonmessage>
    <!--CommunicationOfJSonObjects-->
    <from  ip="">EPFLserver</from>
    <message>{"field":1}</message>
</jsonmessage>
```

▷ Show the list of tokens that should be generated by the lexer for the above XML string. You **need not** show SKIP tokens, which correspond to whitespaces.

Supporting tokens in FSMs

Once we try to put things together as outlined before we note that the usual notion of finite-state automatons does not support tokens.

What we really want is a notion of outputs rather than accepting states.

▷ How can we adapt the formal definitions of finite-state automatons to support such outputs?
▷ Identify where and how the transformations we have seen (regular expressions to NFAs, determinization and minimization) need to be adapted.