CS-320

Computer Language Processing

Exercise Session 2

October 4, 2017

Overview

Today we will have a deeper look at tokenization.

What does a tokenizer do?

How can we automatically generate tokenizers?

Overview

Today we will have a deeper look at tokenization.

What does a tokenizer do?
$\Rightarrow$ It transforms a stream of symbols into a stream of tokens.

How can we automatically generate tokenizers?

Overview

Today we will have a deeper look at tokenization.

What does a tokenizer do?
$\Rightarrow$ It transforms a stream of symbols into a stream of tokens.

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

Tokenizer

First, let us define a tokenizer 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.

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

Recap: Ambiguity in tokenization

Recall that 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.

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

Recap: Ambiguity in tokenization

Recall that 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.

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

▷ How do we avoid ambiguities?

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

Tokenization rules

Given an input string w the tokenizer 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 tokenizer

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

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 tokenizer

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

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 tokenizer

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

tokenize the following input strings:

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

`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 tokenizer

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

tokenize the following input strings:

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

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

A simple tokenization example

Exercise 1

- ▷ Given the tokenizer

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

tokenize the following input strings:

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

$$\underbrace{\mathtt{c\ c\ c}}\ \underbrace{\mathtt{a\ a\ b\ a\ b}}\ \underbrace{\mathtt{a\ c}}\ \underbrace{\mathtt{c\ b\ a}}\ \underbrace{\mathtt{b}}\ \underbrace{\mathtt{c\ c}}\ \underbrace{\mathtt{b}}\ \underbrace{\mathtt{a}}\ \underbrace{\mathtt{b\ a\ c}}$$

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

Constructing a tokenizer

To automatically construct a tokenizer 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 tokenizer (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

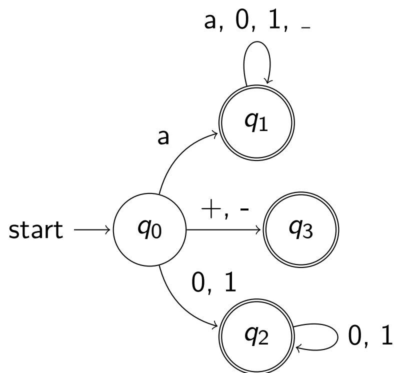
$$\langle ID := a\,(a \mid 0 \mid 1 \mid _)^*,\ INT := (0 \mid 1)\,(0 \mid 1)^*,\ OP := + \mid -\rangle$$

we have

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

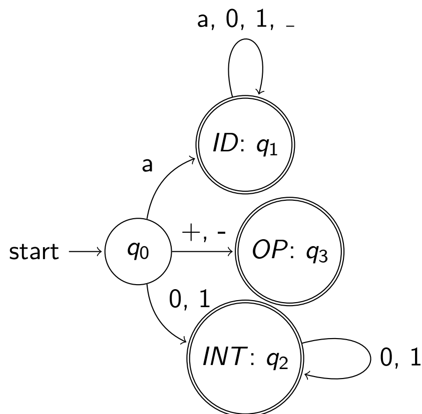
Constructing a tokenizer (3)

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



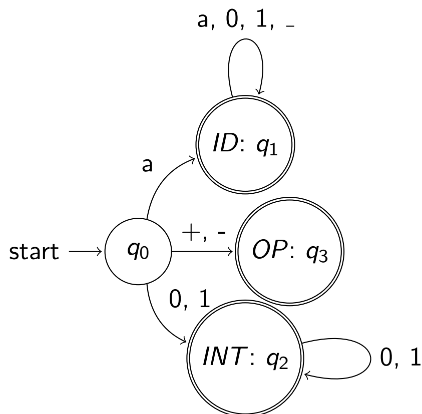
Constructing a tokenizer (3)

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



Constructing a tokenizer (3)

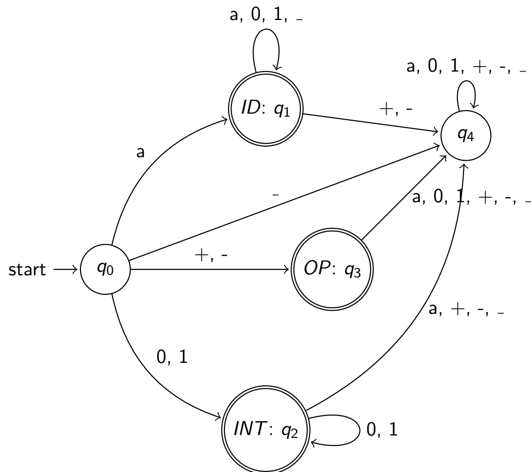
Convert the regular expression to an automaton and specify the token class being recognized by each accepting state:



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

Constructing a tokenizer (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` $= \perp$.

A tokenizer for XML

Exercise 3 (Quiz 2015)

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