CS 241 — Lecture 12

Bartosz Antczak *Instructor: Kevin Lanctot* *February 9, 2017*

12.1 Scanning

This applied to assignment 6, question 1.
The scanning for assembly language was fairly straightforward, but scanning high-level languages are more complicated. A scanner takes in input code (e.g., "`i += 1`") and outputs a sequence of tokens.

12.1.1 Scanning Approach

We'll implement this scanner by reading in the file that contains the high-level language (which, recall, is a sequence of characters) and reading the file one character at a time.
The basic idea is to keep reading characters until you reach the error state:

1. check next state based on character c_i
2. If you reach an error (i.e., there doesn't exist a transition c_i for the current state), look back at the previous state:
   (a) if it was not a final state, report error
   (b) if it was whitespace, ignore
   (c) if it was an accepting state, output the corresponding token
   (d) then go back to start state (i.e., begin looking for the next token)

The pseducode for this approach is:

```
i = 0                                   // start at first char
state = q0                              // start state of DFA
loop:
  next_state = ERROR                    // assume worst case
  if ( i < k ):                         // if not at end of input
     next_state = δ(state, ci)          //  1: go to next state
  if (next_state == ERROR):
     if (state is not an accepting_state):
        report error and exit           //  2a: report error
     if (state is not WHITE_SPACE):     //  2b: ignore white space
        output appropriate token        //  2c: output token
     state = q0                         //  2d: return to start state
     if (i == k):                       //  exit if no more input
        exit
  else:                                 //  process next char
     state = next_state
     i = i + 1
```

Figure 12.1: Courtesy of Prof. Lanctot's slides.

12.2 Context-free Grammar

We can now scan words and tokenize them in out programming language. What we have to do now if recognize the *sentences* and their *meaning*, this is called *parsing*. We will now identify the syntax and semantics of the language. Right now, we'll focus on **syntax**.

12.2.1 Current Challenge - Syntax

Regular expressions, DFAs and NFAs aren't enough. We need something more powerful. For example, in any high-level language, we require balanced parentheses and balances braces:

$$(()(())) \qquad \{\{\{\}\}\}$$

The previous parentheses and braces are **balanced**. Unless if the difference between the left and right brace/parenthesis is fixed, it's impossible to create a DFA. Can you imagine if in C++ you can only create 3 nested `if` statements (because if there were more, the DFA wouldn't be able to read them)? That's ridiculous.

When reading a line, we'll check if each token aligns properly with the token following it (e.g., if we have an integer, a semicolon is a possible accepting state that follows it). To determine if the tokens are aligned properly, we'll refer to a set of rules. For our WLP4 language, a set of rules is listed on WLP4 Language Specification on the CS 241 website, under "context-free syntax". The rules are structured such that if the current state leading to another state is included in the list, the it's valid.