**CS164 Programming Languages and Compilers** **Spring 2024**

# Discussion Worksheet 1: Week of 1/22

## 1 Lexical Analysis

Lexical analysis is the first step in the process of compiling a program. Lexical analysis turns a raw string into something interpretable by the parser:

- Input: Raw string
- Output: List of (token, lexeme) pairs, which are sent to the parser.

A token is a *syntactic category* capturing the meaning of a substring of the input. The *lexeme* is the substring in question.

For example, given the raw string "`31 + a`", the lexer would output:

(`NUM`, `31`), (`OPDELIM`, `+`), (`ID`, `a`)

**Exercise 1** For each of the following strings, give the sequence of (token, lexeme) pairs outputted by the lexer, or write "ERROR" if the string cannot be lexed. Use the 'maximal munch' rule to select the longest possible substring for each token. Use the following python-inspired syntactic categories:

- `NUM`: numbers, consisting of sequences of one or more digits
- `STR`: string literals, consisting of letters and digits enclosed in "`"`"
- `OPDELIM`: operators and delimiters: `+`, `-`, `=`, `(`, `)`, and `:` .
- `ID`: identifiers such as `x`, which must start with a letter, then can contain letters, underscores, and digits
- `KEYWORD`: keywords such as `if`, `else`, `lambda`, etc.

**1.1** `x(4)`

**1.2** `if (x + 3)`

**1.3** `x @ 4`

**1.4**
```
x = lambda x:  x + "a"
```

**1.5** `foo x`

**1.6** `"abc`

**Notes.** Note there is no sense of association; tokenization does not tell us what variables the lambda takes, or what its body is. All lexing does is organize the raw text into more useful atomic units. Usually, lexical analysis will also discard any tokens that are irrelevant to the syntax or meaning of the program; e.g. whitespace or comments. In Python, newlines and indent tokens are relevant to the syntax, so they will be kept around.

## 2 Regex

Recall the definition of a languuage. Let $\Sigma$ be a set of characters (the *alphabet*). A language $\mathcal{L}$ is a set of strings of characters drawn from $\Sigma$. Specifying a explicitly as a set language explicit is tedious, or impossible if the set is infinite. While informal descriptions of a language (e.g. "string literals, consist of letters and digits enclosed in `""`") can help humans understand languages, we need a more formal way of specifying languages to build a lexer. We will explore multiple ways to formalize languages in this class.

Let's start with *regular expressions* (regexes), which can describe *regular languages*. Given a regular expression $A$, let $\mathcal{L}(A)$ be the language described by the regular expression. If $A$ is an atomic character, e.g. '$a$', then $\mathcal{L}(\text{`}a\text{'}) = \{\text{`}a\text{'}\}$. We create more complex regexes with three primitive operators:

- concatenation: $\mathcal{L}(AB) = \{ab | a \in \mathcal{L}(A) \wedge b \in \mathcal{L}(B)\}$. For example, $\mathcal{L}(\text{`}a\text{'}\text{`}b\text{'}) = \{\text{`}ab\text{'}\}$.
- union: $\mathcal{L}(A|B) = \{s | s \in \mathcal{L}(A) \vee s \in \mathcal{L}(B)\}$. For example, $\mathcal{L}(\text{`}a\text{'}|\text{`}b\text{'}) = \{\text{`}a\text{'}, \text{`}b\text{'}\}$.
- iteration (or Kleene closure): $\mathcal{L}(A^*) = \{\text{`'}\} \cup \mathcal{L}(A) \cup \mathcal{L}(AA) \cup \mathcal{L}(AAA) \cup \ldots$. For example, $\mathcal{L}(\text{`}a\text{'}^*) = \{\text{`'}, \text{`}a\text{'}, \text{`}aa\text{'}, \text{`}aaa\text{'}, \ldots\}$

All other regex operators (like character lists, classes, options such as "$a\{3,5\}$") can be expressed by composing these primitive operators. We will omit the `'` below for simplicity.

**Exercise 2** The following are all examples of standard regex abbreviations. How would you write them using the three core operators?

**2.1** `[0-9]` (all numbers from 0 to 9)

**2.2** `[0-3a-d]` (all numbers from 0 to 3, and all lowercase letters from a to d)

**2.3** `ab+` ('a' followed by one or more 'b's)

**2.4** `abc?` ('ab' followed optionally by 'c')

**2.5** `a{3,5}` (3 to 5 'a's)

**Exercise 3** Turn each of the following natural-language descriptions into regexes.

**3.1** `NUM`: sequences of one or more digits

**3.2** `STR`: sequences of letters and digits enclosed in "`"`"

**3.3** `ID`: sequence of one or more letters, underscores, and digits, starting with a letter

# 3 Finite-State Automata

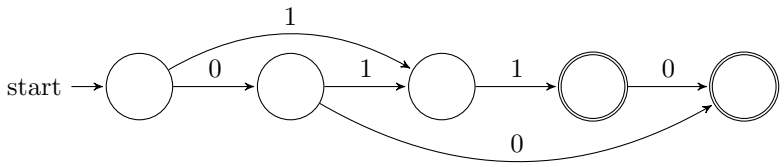
Finite-State Automata (FSAs) are computational models of an abstract machine that, at any time, is in exactly one of a finite number of states.

FSAs consist of a set of states $Q$ and labeled *transitions* between the states. If the automata has a state $q$ with a transition to $q'$ labeled $a$, then when the automata is in state $q$, if it sees the input character $a$, it can transition to $q'$. Within the set of states $Q$, there is a start state $q_0$, which is the state in which the machine starts before it has seen any input. There is also a set of final states $F \subseteq Q$: if after seeing an input, the machine ends up in some $q \in F$, we say the FSA *accepts* the input.

Deterministic FSAs (DFAs) necessitate deterministic state transitions. This means that, for any state and an input symbol, there is exactly one next state (which may be an implicit non-accepting fail-state, for which all transitions loop back to itself).

**Exercise 4** DFA practice.

**4.1** Describe the language that the following DFA accepts. (Recall that final states are written as two concentric circles.)




**4.2** Draw a DFA that accepts the language of binary strings consisting of one 1 in between one or more 0s (i.e. `0+10+`).

Unlike DFAs, which only have one transition per input per state, Nondeterministic Finite Automata (NFAs) can have multiple transitions for one input in a given state, as well as $\varepsilon$ moves. This makes them nondeterministic in the sense that, for a given input, a DFA can only end up in one state, while an NFA can end up in one of multiple states.

**Exercise 5** NFA practice.

**5.1** Describe the language that the following NFA recognizes.