Regular expressions

- You might be familiar with these.
- Example: "^int .*(.*\()\;" is a (flex format) regular expression that appears to match C function prototypes that return ints.
- In our treatment, a regular expression is a program that generates a language of matching strings when you "run it".
- We will use a very compact definition that simplifies things later.
Regular expressions

**Definition.** Let $\Sigma$ be an alphabet not containing any of the special characters in this list: $\varepsilon \emptyset () \cup \cdot \ast$

We define the syntax of the (programming) language $\text{REX}(\Sigma)$, abbreviated as $\text{REX}$, inductively:

- **Base cases**
  1. For all $a \in \Sigma$, $a \in \text{REX}$. In other words, each single character from $\Sigma$ is a regular expression all by itself.
  2. $\varepsilon \in \text{REX}$. In other words, the literal symbol $\varepsilon$ is a regular expression. In this context it is *not* the empty string but rather the single-character *name* for the empty string.
  3. $\emptyset \in \text{REX}$. Similarly, the literal symbol $\emptyset$ is a regular expression.

**Notes:**
- $\text{REX}$ is not defined in our textbook, but is helpful in continuing to build our diagram of languages.
- In our textbook, $a$ represents language $\{a\}$, $\varepsilon$ represents language $\{\varepsilon\}$. 

3
Regular expressions

Definition continued

Induction cases

4. For all $r_1, r_2 \in \text{REX}$, 
   \[ (r_1 \cup r_2) \in \text{REX} \text{ also} \]

5. For all $r_1, r_2 \in \text{REX}$, 
   \[ (r_1 \cdot r_2) \in \text{REX} \text{ also} \]

Note: Later we remove dot, which is denoted by empty circle in textbook (later also removed).
Regular expressions

Definition continued

Induction cases continued

6. For all $r \in \text{REX}$,
   $(r^*) \in \text{REX}$ also

Examples over $\Sigma =$ \{0, 1\}

- $\varepsilon$ and 0 and 1 and $\emptyset$
- $(((1 \cdot 0) \cdot (\varepsilon \cup \emptyset))^*)$
- $\varepsilon \varepsilon$ is not a regular expression

Remember, in the context of regular expressions, $\varepsilon$ and $\emptyset$ are ordinary characters

Note: Textbook also defines $R^+ = R \cdot R^*$, where $R$ is a regular expression.
Semantics of regular expressions

Definition. We define the meaning of the language $\text{REX}(\Sigma)$ *inductively* using the $L()$ operator so that $L(r)$ denotes the language generated by $r$ as follows:

- **Base cases**
  1. For all $a \in \Sigma$, $L(a) = \{ a \}$. A single-character regular expression generates the corresponding single-character string.
  2. $L(\varepsilon) = \{ \varepsilon \}$. The symbol for the empty string actually generates the empty string.
  3. $L(\emptyset) = \emptyset$. The symbol for the empty language actually generates the empty language.
Regular expressions

- **Definition continued**
  - **Induction cases**
    4. For all $r_1, r_2 \in \text{REX}$,
       \[ L\left( r_1 \cup r_2 \right) = L(r_1) \cup L(r_2) \]
    5. For all $r_1, r_2 \in \text{REX}$,
       \[ L\left( r_1 \cdot r_2 \right) = L(r_1) \cdot L(r_2) \]
    6. For all $r \in \text{REX}$,
       \[ L\left( r^* \right) = (L(r))^* \]
  - **No other string is in \text{REX}(\Sigma)**

- **Example**
  - \[ L\left( \left( \left((1 \cdot 0) \cdot (\varepsilon \cup \emptyset)\right)^* \right) \right) \text{ includes} \]
    \[ \varepsilon, 10, 1010, 101010, 10101010, \ldots \]
Orientation

- We used highly flexible mathematical notation and state-transition diagrams to specify DFAs and NFAs
- Now we have a precise programming language REX that generates languages
- REX is designed to close the simplest languages under $\cup$, $\ast$, $\cdot$. 
Abbreviations

- Instead of parentheses, we use precedence to indicate grouping when possible.
  - * (highest)
  - (lowest)
  - (lowest)

- Instead of ·, we just write elements next to each other
  - Example: (((1·0)·(ε∪∅))*) can be written as (10(ε∪∅))*

- If \( r \in \text{REX}(\Sigma) \), instead of writing \( rr^* \), we write \( r^+ \)
Abbreviations

- Instead of writing a union of all characters from $\Sigma$ together to mean "any character", we just write $\Sigma$.
  - In a flex/grep regular expression this would be called ".".

- Instead of writing $L(r)$ when $r$ is a regular expression, we consider $r$ alone to simultaneously mean both the expression $r$ and the language it generates, relying on context to disambiguate.
Abbreviations

- Caution: regular expressions are *strings* (programs). They are equal *only when* they contain exactly the same sequence of characters.
- `(((1·0)·(ε∪∅))*)` can be *abbreviated* `10(ε∪∅)*`
- however `(((1·0)·(ε∪∅))*)` ≠ `10(ε∪∅)*` as strings
- but `(((1·0)·(ε∪∅))*)` = `(10(ε∪∅))*` when they are considered to be the generated languages
- more accurately then,
  \[ L( (((1·0)·(ε∪∅))*) ) = L( (10(ε∪∅))* ) \]
  = \[ L( (10)* ) \]
Examples

- Find a regular expression for \( \{ w \in \{0,1\}^* \mid w \neq 10 \} \)

- Find a regular expression for \( \{ x \in \{0,1\}^* \mid \text{the 6}^{\text{th}} \text{ digit counting from the rightmost character of } x \text{ is } 1 \} \)

- Find a regular expression for \( L_3 = \{ x \in \{0,1\}^* \mid \text{the binary number } x \text{ is a multiple of } 3 \} \)

  (foreshadowing: can be done by starting with DFA and then ripping states)

+ Selected examples from textbook Example 1.53 (p. 65)
Facts

- REX(\(\Sigma\)) is itself a language over an alphabet \(\Gamma\) that is
  \[\Gamma = \Sigma \cup \{ ( ) , ( , , , , , , * , \varepsilon , \emptyset)\}\]
- For every \(\Sigma\), \(|\text{REX}(\Sigma)| = \infty\)
  \(\emptyset,(\emptyset^*),((\emptyset^*)^*),\ldots\)
  even without knowing \(\Sigma\) there are infinitely many elements in \(\text{REX}(\Sigma)\)
- **Question**: Can we find a DFA or NFA \(M\) with \(L(M) = \text{REX}(\Sigma)\)?
The DFA for $L_3$

```
Regular expression:
(0 ∪ 1 ___(0 1* 0)*___ 1 ) *
```

(Recall precedence of operators.)
Regular expression for $L_3$

- $(0 \cup 1 (0 1^* 0)^* 1)^*$

- $L_3$ is closed under concatenation, because of the overall form $(\ )^*$

- Now suppose $x \in L_3$. Is $x^R \in L_3$?

  - Yes: see this is by reversing the regular expression and observing that the same regular expression results

  - So $L_3$ is also closed under reversal
Equivalence with Finite Automata

**Theorem 1.54** A language is regular if and only if some regular expression describes it.

**Proof: 2 directions**

**Lemma 1.55:** If a language is described by a regular expression, then it is regular.  
(Proof idea: Convert to an NFA.)

**Lemma 1.60:** If a language is regular, then it is described by a regular expression.  
(Proof idea: Convert from DFA to GNFA to regular expression.)
Regular expressions generate regular languages

**Lemma 1.55**  For every regular expression $r$, $L(r)$ is a regular language.

**Proof** by induction on regular expressions.

- We used induction to create all of the regular expressions and then to define their languages, so we can use induction to visit each one and prove a property about it.

*Recall that regular expressions were defined inductively.*
\( L(\text{REX}) \subseteq \text{REG} \)

**Base cases:**

1. For every \( a \in \Sigma \), \( L(a) = \{ a \} \) is obviously regular:
   
   ![Diagram](image)

2. \( L(\varepsilon) = \{ \varepsilon \} \in \text{REG} \) also

3. \( L(\emptyset) = \emptyset \in \text{REG} \)
L(REX) ⊆ REG

**Induction cases:**

4. Suppose the induction hypothesis holds for \( r_1 \) and \( r_2 \). Namely, \( L(r_1) \in \text{REG} \) and \( L(r_2) \in \text{REG} \). We want to show that \( L( (r_1 \cup r_2) ) \in \text{REG} \) also. But look: by definition,

\[
L( (r_1 \cup r_2) ) = L(r_1) \cup L(r_2)
\]

Since both of these languages are regular, we can apply Theorem 1.45 (closure of \( \text{REG} \) under \( \cup \)) to conclude that their union is regular.
\( L(\text{REX}) \subseteq \text{REG} \)

**Induction cases:**

5. Now suppose \( L(r_1) \in \text{REG} \) and \( L(r_2) \in \text{REG} \).

By definition,

\[ L((r_1 \cdot r_2)) = L(r_1) \cdot L(r_2) \]

By Theorem 1.47 (closure of REG under \( \cdot \)), this concatenation is regular too.

6. Finally, suppose \( L(r) \in \text{REG} \).

Then by definition,

\[ L((r^*)) = (L(r))^* \]

By Theorem 1.49 (closure of REG under \( ^* \)), this language is also regular. \( \text{QED} \)
On to $\text{REG} \subseteq L(\text{REX})$

- Now we'll show that each regular language (one accepted by an automaton) also can be described by a regular expression
  - Hence $\text{REG} = L(\text{REX})$
  - In other words, regular expressions are equivalent in power to finite automata
- This equivalence is called **Kleene's Theorem** (Theorem 1.54 in book)
Converting DFAs to REX

- Lemma 1.60 in textbook
- This approach uses yet another form of finite automaton called a GNFA (generalized NFA)
- The technique is easier to understand by working an example than by studying the proof
Syntax of GNFA

- A generalized NFA is a 5-tuple $(Q, \Sigma, \delta, q_s, q_a)$ such that
  1. $Q$ is a finite set of states
  2. $\Sigma$ is an alphabet
  3. $\delta: (Q - \{q_a\}) \times (Q - \{q_s\}) \rightarrow \text{REX}(\Sigma)$ is the transition function
  4. $q_s \in Q$ is the start state
  5. $q_a \in Q$ is the (one) accepting state
GNFA syntax summary

- Arcs are labeled with regular expressions
  - Meaning is that "input matching the label moves from old state to new state" -- just like NFA, but not just a single character at a time
- Start state has no incoming transitions, accept has no outgoing
- Every pair of states (except start & accept) has two arcs between them
  - Every state has a self-loop (except start & accept)
Construction strategy

- Will convert a DFA into a GNFA then iteratively shrink the GNFA until we end up with a diagram like this:

\[ q_s \rightarrow \text{giant regular expression} \rightarrow q_a \]

meaning that exactly that input that matches the giant regular expression is in the language.
Converting DFA to GNFA

Adding new start state $q_s$ is straightforward.

Then make each DFA accepting state have an $\varepsilon$ transition to the single accepting state $q_a$.

Note: $\emptyset$ transitions are not drawn here for sake of clarity, but can be important later on.
Interpreting arcs

\[ \delta: (Q-\{q_a\}) \times (Q-\{q_s\}) \rightarrow \text{REX}(\Sigma) \]

In this diagram, for example,

- \( \delta(0,1) = 1 \)
- \( \delta(2,0) = \emptyset \)
- \( \delta(2,q_a) = \emptyset \)
- \( \delta(1,1) = \emptyset \)
- \( \delta(2,2) = 1 \)
- \( \delta(0,q_a) = \varepsilon \)
Eliminating a GNFA state

We arbitrarily choose an interior state (not $q_s$ or $q_a$) to **rip** out of the machine.

**Question**: how is the ability of state $i$ to get to state $j$ affected when we remove rip?

Only the **solid** and **labeled** states and transitions are relevant to that question.
We produce a new GNFA that omits rip

- Its i-to-j label will compensate for the missing state
- We will do this for every \((i,j) \in (Q-\{q_a\}) \times (Q-\{q_s\})\)
- So we have to rewrite every label in order to eliminate this one state
- New label for i-to-j is \(R_4 \cup (R_1 \cdot (R_2)^* \cdot R_3)\)
Don't overlook

- The case \((i, i) \in (Q-\{q_a\}) \times (Q-\{q_s\})\)
- New label for i-to-i is still \(R_4 \cup (R_1 \cdot (R_2)^* \cdot R_3)\)

- Example proceeds on whiteboard, but first we’ll do textbook p. 75 (Figure 1.67) for a simpler one.
g/re/p

- What does grep do?
  
  \((\text{int} \mid \text{float})_\text{rec}.^*\text{emp}\) becomes 
  
  \((\Sigma^*)(\text{int} \cup \text{float})_\text{rec}(\Sigma^*)\text{emp}(\Sigma^*)\)

- What does it mean?

- How does it work?
  
  - Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$
    
    state reduction
  
  - Then run DFA against each line of input, 
    
    printing out the lines that it accepts
# State machines

- Very common programming technique

```java
while (true) {
    switch (state) {
    case NEW_CONNECTION:
        process_login();
        state=RECEIVE_CMD;
        break;
    case RECEIVE_CMD:
        if (process cmd() == CMD QUIT)
            state=SHUTDOWN;
        break;
    case SHUTDOWN:
        ...
        break;
    ...
    ...
}
```

This chapter so far

§1.1: Introduction to languages & DFAs
§1.2: NFAs and DFAs recognize the same class of languages
§1.3: REX generates the same class of languages

✓ Three different programming "languages" specified in different levels of formality that solve the same types of computational problems
✓ Four, if you count GNFAs
Strategies

- If you're investigating a property of regular languages, then as soon as you know $L \in \text{REG}$, you know there are DFAs, NFAs, Regexes that describe it. Use whatever representation is convenient.

- But sometimes you're investigating the properties of the programs themselves: changing states, adding a * to a regex, etc. Then the knowledge that other representations exist might be relevant and might not.
All finite languages are regular

**Theorem** (not in book) \( \text{FIN} \subseteq \text{REG} \)

**Proof**  Suppose \( L \in \text{FIN} \).

Then either \( L = \emptyset \), or \( L = \{ s_1, s_2, \ldots, s_n \} \)

where \( n \in \mathbb{N} \) and each \( s_i \in \Sigma^* \).

A regular expression describing \( L \) is, therefore, either \( \emptyset \) or

\[
s_1 \cup s_2 \cup \cdots \cup s_n \quad \text{QED}
\]

**Note that** this proof does not work for \( n=\infty \)
Each point is a language in this Venn diagram.

REG = L(DFA) = L(NFA) = L(REX) = L(GNFA) ≠ FIN

"the class of languages generated by DFAs"

is there a language out here?

ALL