DD1362
Programming Paradigms

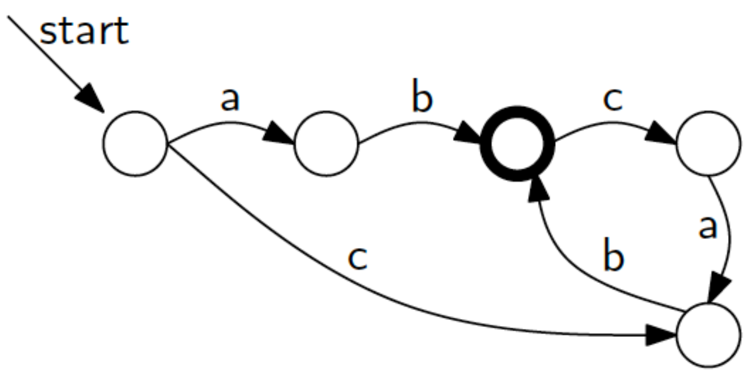
Formal Languages and Syntactic Analysis
Lecture 2

Philipp Haller

April 12th, 2021

Review of Lecture 1

- Formal languages
  - "Language = subset of Σ^*"
- Regular expressions
  - Example: letter (letter | digit)*
- Finite automata
  - Example:



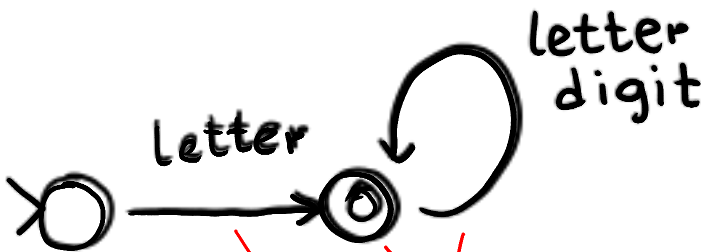
Σ^* = set of all words over alphabet Σ

Today's Lecture

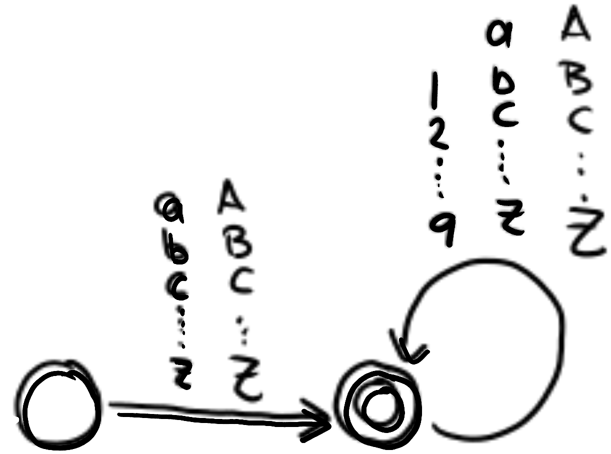
- Finite automata formally
- Regular languages
- Context-free grammars

Finite Automata Formally

Finite Automata Formally



i.e.



$$A = (\Sigma, Q, q_0, \delta, F)$$

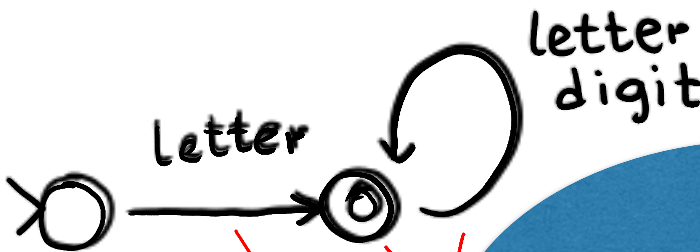
$$\delta \subseteq Q \times \Sigma \times Q$$

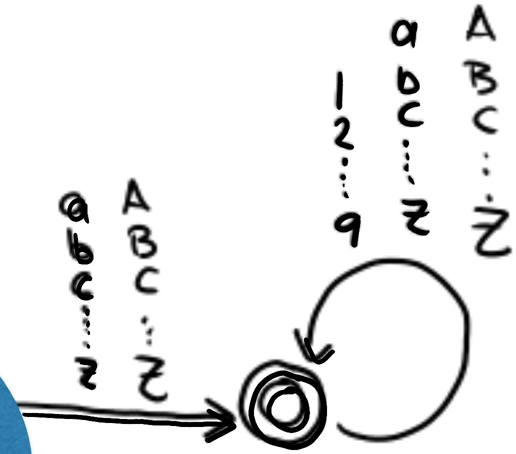
$$(q_1, a, q_2) \in \delta$$

$$q_1 \xrightarrow{a} q_2$$

- Σ - alphabet
- Q - states (nodes in the graph)
- q_0 - initial state (with '>' sign in drawing)
- δ - transitions (labeled edges in the graph)
- F - final states (double circles)

Kinds of Finite State Automata





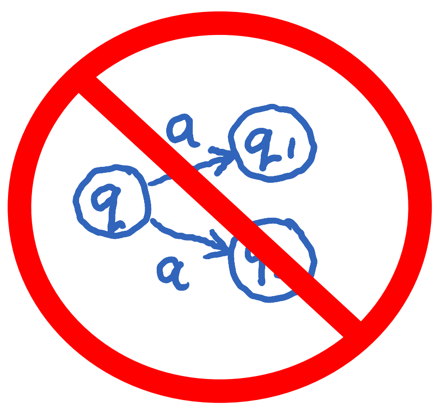
DFA =
Deterministic Finite Automaton

$A = (\Sigma, Q, q_0, \delta,$

$\times \Sigma \times Q$

$(q_1, a, q_2) \in \delta$

- Deterministic: δ is a function



$$\frac{(q, a, q_1) \in \delta \qquad (q, a, q_2) \in \delta}{q_1 = q_2}$$

- Otherwise: non-deterministic

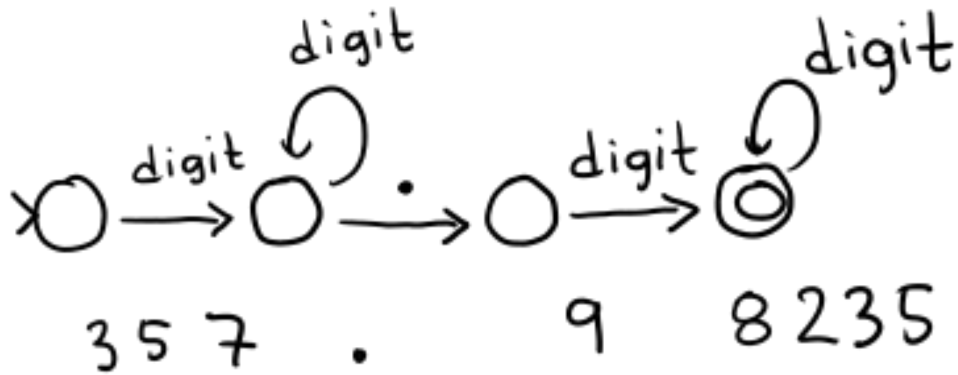
Regular Expressions and Automata

Theorem:

If L is a set of words, then it is a value of a regular expression *if and only if* it is the set of words accepted by some finite automaton.

Using DFAs to Recognize Languages

DFA for recognizing valid floating-point numbers?



Corresponding regular expression?

digit digit* . digit digit*

Regular Languages

Regular Languages

> Regular expressions and finite automata have ***the same expressive power***

- They “describe the same class of languages”
- Languages that can be described using either a regular expression or a DFA are called ***regular languages***
- Examples of regular languages:
    - Names of labs in DD1362
    - All binary strings (“”, “0”, “1”, “00”, “01”, …)
    - Valid identifiers in a programming language

Properties of Regular Languages

Suppose A and B are regular languages over Σ. Then, the following properties hold:

- $\overline{A} = \Sigma^* - A$ (complement of A) is regular
  Proof idea: swap accepting and non-accepting states in DFA for A
- $A \cup B$ is regular
  Proof idea: describe using $R_A \mid R_B$
- $A \cap B$ is regular
  Proof idea: use $A \cap B = \overline{(\overline{A} \cup \overline{B})}$
- $A - B$ is regular
  Proof idea: use $A - B = A \cap \overline{B}$

Limitations of Regular Expressions

- Are there languages that ***cannot be described using a regular expression*** (or a DFA)?
- If such languages exist, how can we prove that (no matter how clever we are) we could never find a regular expression describing that language?
- Example language: $L = \{ a^n b^n \mid n >= 0 \}$
  Intuition: ***(not a proof!)***
    - After reading k ‘a’s and then only ‘b’s, the DFA needs to ***remember*** that only k ‘b’s may be read
    - A DFA can only remember a ***fixed number of states***, however, in language L the number n is ***arbitrarily large***

Automaton that Claims to Recognize $L = \{ a^n b^n \mid n >= 0 \}$

- Assume there is a DFA recognizing L
- Let the DFA have K states, $|Q| = K$
- Feed it a, aa, aaa, …. Let q_i be state after reading a^i
- Consider the following sequence:
  q_0 , q_1 , q_2 , … , q_K
- This sequence has length K+1 → a state must repeat: there is an index i such that
  $q_i = q_{i+p}$ for some $p > 0$
- Then the automaton should accept a^{i+p} b^{i+p}
- But then it must also accept a^i b^{i+p}
- because it is in the same state after reading a^i as after a^{i+p}
- Thus, it does not accept the given language — contradiction
- Therefore, no such DFA exists. QED

Context-free Grammars

Grammars

- To describe languages that cannot be described using regular expressions (or DFAs) ***more powerful formalisms*** are needed
- Grammars, more precisely, ***context-free grammars*** are such a formalism
- Let us revisit language L which cannot be expressed using a DFA (or regular expression): $L = \{ a^n b^n \mid n >= 0 \}$
- How could we describe L mathematically?

Grammar for Language L

- We can define $L = \{ a^n b^n \mid n >= 0 \}$ mathematically using induction:
- ***Base case i = 0:***
  The empty word $\varepsilon = a^0 b^0$ is a word in L
- ***Inductive case i+1 > 0:***
  Let word $w = a^i b^i$ for some i >= 0 (w is in L)
  Then, awb is a word in L
- Grammars enable describing such languages using recursion

Grammar for Language L

- Grammar for language L:
  $W \rightarrow \varepsilon \mid aWb$
- The above grammar can be understood as follows:
  “A valid word W is either the empty word ε or an ‘a’ followed by a valid word followed by a ‘b’.”

Elements of a Grammar

Grammar G for language L:
$W \rightarrow \varepsilon \mid aWb$

- ***Terminal symbols*** are elements of the underlying alphabet
  - Grammar G: terminal symbols a, b in alphabet $\Sigma = \{a, b\}$
- ***Non-terminal symbols*** stand for parts of a word
  - Grammar G: non-terminal symbol W
- ***Productions*** are rules for how non-terminal symbols are composed of terminal symbols and other non-terminal symbols
  - Two productions in grammar G (abbreviated using ‘|’):
    $W \rightarrow \varepsilon$
    $W \rightarrow aWb$

Derivations

$S \rightarrow B \mid AA$
$A \rightarrow cA \mid dB$
$B \rightarrow aSa \mid \varepsilon$

Start symbol S

This process is called ***derivation***

Q: ***How to find strings that are in the language?***

A: ***Apply the production rules*** to generate valid strings starting from the grammar's start symbol

Each step: replace exactly one non-terminal

- For example, two derivations:
  $S \rightarrow B \rightarrow aSa \rightarrow aBa \rightarrow aa$
  $S \rightarrow B \rightarrow aSa \rightarrow aAAa \rightarrow acAAa \rightarrow acdBAa \rightarrow acdAa \rightarrow acddBa \rightarrow acdda$
- Normally, however, we are interested in the converse:
  Given a program, does it correspond to the grammar of [Java/Scala/Go/...]?

Notation for Grammars

Two common notations for grammars:

Mathematical notation:

$S \rightarrow B \mid AA$

$A \rightarrow cA \mid dB$

$B \rightarrow aSa \mid \varepsilon$

Backus-Naur-Form (BNF):

```
<S> ::= <B> | <A><A>
<A> ::= "c" <A>
      | "d" <B>
<B> ::= "a" <S> "a"
      | ""
```

Different syntax for terminal and non-terminal symbols!

Grammars for Programming Languages

- Context-free grammars are powerful enough to describe the ***syntax of general-purpose programming languages*** (Java, C#, Scala, …)
- Syntax specifications of programming languages ***often make use of context-free grammars***
  - Specs may introduce specific notations (example: Java Language Specification)
- Such grammars are the ***primary basis for the implementation of parsers*** for the specified languages

Example: Grammar for Expressions in the Scala language (Extract)

```
Expr        ::=  (Bindings | [‘implicit’] id | ‘_’) ‘=>’ Expr
              |  Expr1
Expr1       ::=  ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] ‘else’ Expr]
              |  ‘while’ ‘(’ Expr ‘)’ {nl} Expr
              |  ‘try’ Expr [‘catch’ Expr] [‘finally’ Expr]
              |  ‘do’ Expr [semi] ‘while’ ‘(’ Expr ‘)’
              |  ‘for’ (‘(’ Enumers ‘)’ | ‘{’ Enumers ‘}’) {nl} [‘yield’] Expr
              |  ‘throw’ Expr
              |  ‘return’ [Expr]
              |  [SimpleExpr ‘.’] id ‘=’ Expr
              |  SimpleExpr1 ArgumentExprs ‘=’ Expr
              |  PostfixExpr
              |  PostfixExpr Ascription
              |  PostfixExpr ‘match’ ‘{’ CaseClauses ‘}’
PostfixExpr ::=  InfixExpr [id [nl]]
InfixExpr   ::=  PrefixExpr
              |  InfixExpr id [nl] InfixExpr
PrefixExpr  ::=  [‘-’ | ‘+’ | ‘~’ | ‘!’] SimpleExpr
```

Extended Backus-Naur Form (EBNF)

Example: Grammar for Lists in Haskell

Let us create a grammar that describes the syntax for lists in Haskell, for example:

`[1, 2, 3]`   `1:[2,3]`   `[]`   `1:2:3:[]`

Inductive definition:

- ***Base case 1:***
  The empty list [] is a Haskell list
- ***Base case 2:***
  If L is a comma-separated sequence of list elements, then [L] is a Haskell list
- ***Inductive case:***
  If H is a list element and T is a list, then H:T is a list

Example: Grammar for Lists in Haskell

Inductive definition:

- ***Base case 1:***
  The empty list [] is a Haskell list
- ***Base case 2:***
  If L is a comma-separated sequence of list elements, then [L] is a Haskell list
- ***Inductive case:***
  If H is a list element and T is a list, then H:T is a list

The three cases expressed in BNF:

```
<List> ::= "[]"
         | "[" <ListElems> "]"
         | <ListElem> ":" <List>
```

Example: Grammar for Lists in Haskell

```
<List> ::= "[]"
         | "[" <ListElems> "]"
         | <ListElem> ":" <List>
```

Non-terminal `<List>` defined in terms of two additional non-terminals `<ListElems>` and `<ListElem>`

Inductive definition of `<ListElems>`:

- ***Base case:*** if E is a list element, then E is also a sequence of list elements
- ***Inductive case:*** if E is a list element and L is a sequence of list elements, then E, L is a sequence of list elements

Example: Grammar for Lists in Haskell

```
<List> ::= "[]"
         | "[" <ListElems> "]"
         | <ListElem> ":" <List>

<ListElems> ::= <ListElem>
              | <ListElem> "," <ListElems>
```

Inductive definition of `<ListElems>`:

- ***Base case:*** if E is a list element, then E is also a sequence of list elements
- ***Inductive case:*** if E is a list element and L is a sequence of list elements, then E, L is a sequence of list elements

Example: Grammar for Lists in Haskell

```
<List> ::= "[]"
         | "[" <ListElems> "]"
         | <ListElem> ":" <List>

<ListElems> ::= <ListElem>
              | <ListElem> "," <ListElems>
```

How to define `<ListElem>`?

- In real Haskell, list elements can be arbitrary Haskell expressions
- To keep our example simple enough: limit lists to list elements 0 and 1

Example: Grammar for Lists in Haskell

```
<List> ::= "[]"
         | "[" <ListElems> "]"
         | <ListElem> ":" <List>
<ListElems> ::= <ListElem>
              | <ListElem> "," <ListElems>
<ListElem> ::= "0" | "1"
```

How to define `<ListElem>`?

- In real Haskell, list elements can be arbitrary Haskell expressions
- To keep our example simple enough: limit lists to list elements 0 and 1

Example: Grammar for Lists in Haskell

```
<List> ::= "[]"
         | "[" <ListElems> "]"
         | <ListElem> ":" <List>
<ListElems> ::= <ListElem>
              | <ListElem> "," <ListElems>
<ListElem> ::= "0" | "1"
```

Complete grammar for Haskell lists where each list element is either 0 or 1