3. Formal Languages, Grammars and Automata
Contents

• Languages, their Syntax and Semantics
• Formal Languages and Formal Grammars
• Regular Languages and Expressions
• Finite Automata: DFA, NFA, PDA
• Conversion Regular Expressions into NFA & DFA
• Context-Free Languages and Grammars
• BNF Notation
• Derivations
• Check Your Understanding
Languages: Some Definitions

- **Language** is a formal symbolic system governed by grammatical rules of combination to communicate meaning.

- Chomsky’s **generative theory of grammar** (structuralist view):
  - **Language** as a particular set of sentences that can be generated from a particular set of rules.
  - **Grammar** is the set of structural rules that govern the composition of clauses, words, sentences.
  - **Generative grammar** is finite set of syntactical rules that can be applied to *generate* all those and only those sentences that are grammatical (well-formed) in a given language.
Syntax and Semantics

• **Syntax** (structure) and **semantics** (meaning) of language are inextricably linked: structure is instrumental in recognition (understanding) of its meaning.

• Chomsky’s sentence that is grammatically correct but semantically nonsensical.

  ![ Syntax Example Diagram]

  Colorless green ideas sleep furiously.

• Another approach is to create a syntactically-correct sentence using nonsense words: **Lewis Carroll's Jabberwocky:**
  
  "'Twas brillig, and the slithy toves…"
Example of Languages

• Small English
  – Syntax of the language:
    \texttt{sentence} \rightarrow \texttt{subject} \texttt{predicate}
    \texttt{subject} \rightarrow \texttt{cats} | \texttt{dogs}
    \texttt{predicate} \rightarrow \texttt{sleep} | \texttt{eat}
  – Language consists of possible sentences:
    • cats sleep
    • cats eat
    • dogs sleep
    • dogs eat

• More Formal Language
  – Syntax:
    \text{S} \rightarrow \text{AB}
    \text{A} \rightarrow x | y
    \text{B} \rightarrow z | w
  – Language consists of possible sentences:
    • xz
    • xw
    • yz
    • yw

Terminal and Non-terminal symbols, Productions, Meta-symbols.
Operators on Languages

- Formal specification of syntax requires a set of rules (operators)
- **Concatenation:** \( L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \} \)
- **Union:** \( L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \} \)
- **Exponentiation:** \( L^0 = \{ \varepsilon \} \quad L^1 = L \quad L^2 = LL \)
- **Kleene Closure** (Iteration)
  \[
  L^* = \bigcup_{i=0}^{\infty} L^i \quad \text{where } L^i = LLLLLL... (i \text{ times})
  
  L^* = \bigcup_{i=0}^{\infty} L^i \quad \text{where } L^i = LLLLLL... (i \text{ times})
  
- **Positive Closure**
  \[
  L^+ = \bigcup_{i=1}^{\infty} L^i
  
  L^+ = \bigcup_{i=1}^{\infty} L^i
  
  \]
Formal Languages and Grammars (1)

- Formal Language is a set of strings of symbols.
- Formal Grammar is a set of formation rules for strings in formal languages.
  - Grammar can be thought as both *Language Generator* and *Language Recogniser*.
- Let Grammar $G$ be specified by the tuple:
  - $G = G(T, N, P, S)$
    - A finite set $T$ of *terminal symbols* (vocabulary)
    - A finite set $N$ of *non-terminal symbols* (grammatical categories, none appears in strings formed from $G$.
    - A set $P$ of *productions* (syntactical rules) of the form:
      $(T \cup N)^*N(T \cup N)^* \rightarrow (T \cup N)^*$
    - A symbol $S \in N$ called the *start symbol*. 

---

2012-13

NCCA SDAGE Level I: Languages and Compilers

Dr Valery Adzhiev
Formal Languages and Grammars (2)

- To define language of $G$ denoted as
  - $L(G) \overset{\text{def}}{=} \{ \xi \mid S \Rightarrow^*_G \xi \text{ and } \xi \in (T \cup N)^* \}$
  - $L(G)$ consists of all finite length sentences (including empty sequence $\epsilon$ of length zero).

  - Given grammar $G$, binary relation $\Rightarrow^*_G$ (“$G$ derives in zero or more steps) on strings in $(T \cup N)^*$ as reflective transitive closure of $\Rightarrow_G$
  - Sentential form (sequence) is a member of $(T \cup N)^*$ that can be derived in a finite number of steps from the start symbol $S$. Sentential form that contains no non-terminal symbols $s$ called a **sentence**.
  - Sentence $\eta$ can be *directly derived* from sentence $\xi$ if and only if there exist sequences $\alpha, \beta, \xi', \eta'$ such as
    - $\xi = \alpha \xi' \beta$
    - $\eta = \alpha \eta' \beta$
    - $P$ contains production $\xi' \rightarrow \eta'$
  - Thus $S \Rightarrow^*_G \eta$
  - $S \Rightarrow^*_G \xi z$
  - $\Rightarrow^*_G \xi z$
  - $S \Rightarrow^*_G \xi z$
  - Since $xz \in T^*$, then $xz \in L$. 

Language $L$:

- $S \rightarrow AB$
- $A \rightarrow x|y$
- $B \rightarrow z|w$
Chomsky Hierarchy

- **Hierarchy of grammars** was described by Noam Chomsky. Classes are characterised in two ways by:
  - Type of rules to *generate* set of strings
  - Type of formal machine capable of *recognising* the language

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Languages</th>
<th>Automation</th>
<th>Productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-0</td>
<td>Recursively enumerable</td>
<td>Turing Machine</td>
<td>$\alpha \rightarrow \beta$ (no restrictions)</td>
</tr>
<tr>
<td>Type-1</td>
<td>Context-sensitive</td>
<td>Linear-bounded non-deterministic Turing machine</td>
<td>$\alpha A \beta \rightarrow \alpha \xi \beta$</td>
</tr>
<tr>
<td>Type-2</td>
<td>Context-free</td>
<td>Non-deterministic PDA</td>
<td>$A \rightarrow \xi$</td>
</tr>
<tr>
<td>Type-3</td>
<td>Regular</td>
<td>Finite state automaton</td>
<td>$A \rightarrow a$ or $A \rightarrow aB$</td>
</tr>
</tbody>
</table>
Regular Languages

- **Regular Language** is denoted by *Regular Expressions* (RE)
- They are *recognised* by Scanner
- REs are used to specify **tokens** – the shortest string of characters with individual meaning.
- To specify tokens we use *notation* of REs. r as RE is one of:
  - A character ‘a’: denotes set \{a\}
  - Empty string denoted ‘ε’: denotes set \{ε\}
  - Two REs concatenated ‘(r\(^1\)) (r\(^2\))’: denote \(L(r\(^1\)) L(r\(^2\))\)
  - Two REs separated by | (i.e. ‘or’ - alternation) ‘(r\(^1\)) | (r\(^2\))’: \(L(r\(^1\)) \cup L(r\(^2\))\)
  - RE followed by Kleene star (concatenation of zero or more strings)
    \((r)^*\) denotes \(L(r)^*\)
- Parentheses to group operations to avoid ambiguity
- Precedence:
  - Closure: highest; concatenation: middle; alternation: lowest
    - \(ab^*|c\ means (a(b)^*)|(c)\)
Regular Expressions: Examples (1)

- \(0|1\) \(\Rightarrow \{0,1\}\)
- \((0|1)(0|1)\) \(\Rightarrow \{00,01,10,11\}\)
- \(a^*\) the set of strings of zero or more a's
- \(0^*\) \(\Rightarrow \{\varepsilon,0,00,000,0000,\ldots\}\)
- \((0|1)^*\) \(\Rightarrow\) all strings with 0 and 1, including the empty string
- \(aa^* = a^+\) strings of one or more a's
- \((a|b)^*\) all strings of a's and b's
- \(a|ba^* = a|(ba^*)\) either the string "a" or the set of strings starting with 'b' followed by zero or more
Regular Expressions: Examples (2)

- Syntax of numerical constants (typical for simple calculator)

  - Symbols on the left provide names for REs, one of these (‘number’) serves as token name.
  - While definitions allowed to built on one another, nothing is defined in terms of itself, even indirectly.
  - To generate valid number, expand out sub-definitions and then scan resulting expressing from left to right, choosing among alternatives at each ‘|’, and choosing number of repetitions at each Kleene star. Within each repetition may make different choices at ‘|’, generating different substrings.
Exercise in RE

• Write regular expressions for **Strings in C**
  – These are
    • delimited by double quotes ("),
    • may not contain newline characters `nl`.
    • may contain double-quote (") or backslash (\) characters if and only if those characters are “escaped” by a preceding backslash.
    • Hint: You may find it helpful to introduce shorthand notation to represent any character that is *not* a member of a small specified set.
  – Answer: ??!
Deterministic Finite Automata (DFA)

- DFA is formal machine that accepts (recognises) Regular Language.
- Formally, DFA $M = (Q, \Sigma, q_1, F, \delta)$:
  - A finite set $Q$ of states
  - A finite alphabet $\Sigma$ of input symbols
  - A distinguished initial state $q_1 \in Q$
  - A set of distinguished final states $F \subseteq Q$
  - A transition function $\delta : Q \times \Sigma \rightarrow Q$ that chooses new state for $M$ based on current state and current input symbol
- When final symbol consumed, $M$ accepts it ("yes!") or rejects it ("no!")

This DFA accept strings of 'a's and 'b's that begin and end with same symbol
DFA as Implementation of REs

• One can extend transition function $\delta$ to *take strings*, rather than symbols: $M$ accepts string $x$.

• *Language* accepted by $M$: $L(M) = \{x| \delta (q_1, x) \in F\}$

• It can be proven that REs and DFA are equivalent:
  – One can construct DFA that accepts language defined by given RE, and vice versa.
  – DFA *implements* RE

• DFA can be easily and efficiently implemented in compiler

• But there is no obvious one-step algorithm to convert a set of REs into equivalent DFA.

• Another formal machine can help – **Non-deterministic Finite Automata (NFA)** that is also equivalent to DFA.
Non-Deterministic Finite Automata (NFA)

- In NFA, transition function $\delta$ is multivalued:
  \[ \delta: Q \times (\Sigma \{ \epsilon \}) \rightarrow P(Q) , \ P \text{ is Power set of } Q. \]

- The automaton can move to any of a set of possible states from a given state on a given input.
  - Same input may produce multiple paths
  - Transitions to different states given input
  - In addition, it may move from one state to another “spontaneously”; such transitions are said to take input empty string $\epsilon$.
  - $\epsilon$-transitions provides a convenient way of modelling the systems whose current states are not precisely known.
Example: NFA that accepts string containing either 101 or 11 as a sub-string
- Read symbol, clone a machine for each matching transition
- If any ε transitions, clone a machine for each ε transition
- If a symbol is read and there is no way to exit from a state, then machine says “no!”
- At end of input if any machine accepts than “yes!”
Construction of NFA Equivalent to RE

- **Basic case:**
  - Trivial RE of a single character c is equivalent to to two-state NFA (=DFA)
  - Similarly, RE ε is equivalent to 2-state NFS with arc labeled by ε.

- **Three sub-constructions**
- Each step preserves 3 invariants:
  - No transitions into initial state
  - There is single final state
  - No transitions out of final state
  - They allow smaller machines to be joined into larger ones without ambiguity where to create connections and without unexpected paths:
Construction of NFA from Given RE (Example)

- **RE to be realised by NFA:**
  - decimal → digit* ( . digit | digit . ) digit*
    - String of decimal digits containing single decimal point:
    - With only one digit point can be at beginning or end: ( .d | d. )
    - Arbitrary numbers of digits can then be added at beginning or end

- **Construction:**
  - Top row: NFA for ., d and Kleene closure for d*
  - 2 row: concatenation for .d, d.
  - 3 row: alternation for ( .d | d. )
  - 4 row: concatenation to complete NFA
Construction of DFA Equivalent to NFA

- Will use “set of subsets” construction
  - The state of DFA will represent the set of states that NFA might have reached on the same input (including $\varepsilon$).
  - DFA initial state for set of NFA states: ‘1’ + those with $\varepsilon$ transition from 1; etc.
  - DFA states D-G are marked as final as they contain state 14 of NFA. E.g., given input $d$, there exist path from 1 to 14 of NFA.
  - Here DFA is smaller that NFA but in theory DFA number of states may be exponential in number of states in NFA.
Minimization of DFA

• Once $d$ and $\cdot$ processed, the only valid transitions are on $d$, so single final state should be possible.
• This can be formalised to apply to any DFA via inductive construction:
  – Initially place DFA states into 2 equivalence classes: final and nonfinal.
  – Then in each step split set of states to eliminate a transition ambiguity.
• First state ABC has ambiguous transitions of both $d$ and $\cdot$.
• After splitting, new AB has a self-loop and new C moves to State DEFG.
• AB has ambiguity on $\cdot$, so split it and get final four-state minimal DFA.

Smaller DFA that produced from NFA might exist.
Context-Free Languages and Grammars

• Programming languages have **recursive structure**:
  – Structure of Arithmetic expression:
    • \( expr \rightarrow id \mid number \mid - expr \mid ( expr ) \mid expr \ op \ expr \)
    • \( op \rightarrow + \mid - \mid * \mid / \)

• **Context-Free Grammar (CFG)**:
  – \( A \rightarrow \xi, A \in N \text{ and } \xi \in (N \cup T)^* \)
  – Left side of each production consists of single non-terminal symbol and can be replaced by \( \xi \) regardless of context in which \( A \) occurs

• **CF Language** is precisely defined by CFG.

• CF Languages are proper superset of Regular Languages:
  – They are defined using concatenation, alternation and recursion (which subsumes Kleene closure)

• Notation for CFG is sometimes called **Backus-Naur Form (BNF)**
  – initially devised for definition of Algol-60.

• There is several **EBNF** (Extended BNF) notations
BNF and EBNF Notations

- **BNF specification** is a set of productions (derivation or rewriting rules), with non-terminal on LHS, sequences on RHS and ‘→’ or ‘::=‘ in the middle.
- LHS if first listed production is Start symbol.
- BNF can describe:
  - Nested structures and Lists of similar constructs (using recursion, if needed): id → id; id_list → id_list , id
  - Operator precedence and associativity.
- Production can have more than one RHS separated by ‘|’.
- In EBNF:
  - Kleene star * and meta-level parenthesis of REs: id_list → id ( , id )*.
  - {...} used to indicate zero or more instances of symbols.
  - [...] used to indicate zero or one instances of symbols (i.e. those symbols are optional): id → lt { [ let | dig ] } ; let → a – z; dig → 0 – 9.
Push-Down Automata (PDA)

- **PDA** implements (i.e., recognises) CF Language.
- **PDA** employs **stack**
  - uses the top of the stack to decide which transition to take
  - manipulates the stack as part of performing a transition
  - three parameters (input symbol, current state, and the symbol at the top of the stack completely determine the transition path that is chosen)

A PDA is formally defined as a 7-tuple:

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F) \]

where

- \( Q \) is a finite set of states
- \( \Sigma \) is a finite set which is called the **input alphabet**
- \( \Gamma \) is a finite set which is called the **stack alphabet**
- \( \delta \) is a finite subset of \( Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^* \)
- \( q_0 \in Q \) is the **start state**
- \( Z \in \Gamma \) is the **initial stack symbol**
- \( F \subseteq Q \) is the set of **accepting states**

http://en.wikipedia.org/wiki/Push-down_automaton
CFG Derivation (Example)

- **GFG: Arithmetic expression**
  
  \[
  \text{expr} \rightarrow \text{id} \mid \text{number} \mid \text{- expr} \mid ( \text{expr} ) \mid \text{expr op expr} \\
  \text{op} \rightarrow + \mid - \mid * \mid /
  \]

- **Derivation as a sequence of replacement operations:**
  - Replacement strategy: right-most derivation: at each step replace right-most non-terminal with RHS of some production

- **Parse Tree:**
  - Not unique: at 2 level we could’ve chosen to turn operator into a * instead a +, etc.
  - So this CFG is ambiguous that can result in problems

Slope: \( x \times x + \text{intercept} \)

\[
\begin{align*}
\text{expr} & \Rightarrow \text{expr op expr} \\
& \Rightarrow \text{expr op id} \\
& \Rightarrow \text{expr op expr} + \text{id} \\
& \Rightarrow \text{expr op id} + \text{id} \\
& \Rightarrow \text{id} \times \text{id} + \text{id} \\
& (\text{slope})(x)(\text{intercept})
\end{align*}
\]
### CFG: Example

- **Expression grammar with precedence and associativity**
  - This grammar is unambiguous
  - It captures precedence in way *factor, term* and *expr* build on one another, with different operation at each level
  - It captures associativity in second halves of lines 1 and 2, which builds sub *exprs* and sub *terms* to the left of operator, not to the right

```plaintext
1. expr  →  term | expr add_op term
2. term  →  factor | term mult_op factor
3. factor →  id | number | - factor | ( expr )
4. add_op →  + | -
5. mult_op →  * | /
```
• Parse tree for expression grammar (with precedence) for $3 + 4 * 5$
  – Building precedence into grammar makes it clear that multiplication groups more tightly than addition, even without parenthesis
• Parse tree for expression grammar (with left associativity) for $10 - 4 - 3$
  – Subtraction groups more tightly to the left, so the result would be 3
Check Your Understanding (1)

- What is difference between natural languages and formal languages?
- Define language syntax and semantics. How do they relate?
- What are the main notions concerned with formal grammar?
- Specify the formal procedure how language is derived given the grammar.
- What is the difference between grammars of different type?
- What are three basic operations used to build complex regular expressions from simpler ones?
- Explain the difference between DFA, NFA, PDA.
Check Your Understanding (2)

• How to built NFA from a regular expression?
• How to built DFA from NFA and then optimise it?
• What are context-free grammars for? What additional operation (beyond the three of REs) is provided and for what?
• Why there was a need to devise Extended BNF?
• What are context-free grammars for? What additional operation (beyond the three of REs) is provided and for what?
• When discussing CF languages, what is a derivation?
• Why are associativity and precedence important in Parse Tree?