Automata, Grammars and Languages

Discourse 01

Introduction

Fundamental Questions

Theory of Computation seeks to answer fundamental questions about computing

- What is computation?
  - Ancient activity back as far as Babylonians, Egyptians
  - Not precisely settled until circa 1936
- What can be computed?
  - Different ways of computing (C, Lisp, ...) result in the same "effectively computable" functions from input to output?
- What cannot be computed?
  - Not \( \sqrt{2} \) but can get arbitrarily close
  - Are there precisely defined tasks ("problems") that cannot be carried out? Yes/No decisions that cannot be computed?
- What can be computed efficiently? (Computational Complexity)
  - Are there inherently difficult although computable problems?

Basic Concepts: Automata, Grammars & Languages

- Language: a set of strings over some finite alphabet \( \Sigma \)
  - Ex: \( L = \{ TAA, TGA, TAG, \ldots \} \) DNA codons
  - \( \Sigma = \{ A, G, C, T \} \)
- Automaton (Machine): abstract (=simplified) model of a computing device. Used to "recognize" strings of a language \( L \)
  - Ex: Finite Automaton (Finite State Machine)
- Grammar: finite set of string rewriting rules. Used to specify (derive) strings of a language
  - Ex: \( S \rightarrow SS \) Context-Free Grammar
  - Ex: \( S \rightarrow x \) (CFG)
Languages

\[ L_1 = \{aa, ab, ba, bb\} \quad \Sigma = \{a, b\} \]
\[ L_2 = \{e, a, aa, aaaa, \ldots\} \quad \Sigma = \{a\} \]
\[ L_3 = \{e : e \text{ is a well-formed arithmetic expression in C}\} \quad \Sigma = \{0-9, \pm, \times, \div, (, ), \} \]
\[ L_4 = \{p : p \text{ is a well-formed C program}\} \quad \Sigma = \{\text{ASCII}\} \]
\[ L_5 = \{p : p \text{ is a w.-f. C program that halts for all inputs}\} \]
\[ L_6 = \{(x, y) : x \text{ is a decimal integer and } y \text{ is its binary representation}\} \]

Types of Machines

• Logic circuit
  • memoryless; values combined using gates
  ![Logic Circuit Diagram]
  
  Circuit size = 5
  Circuit depth = 3

Types of Machines (cont.)

• Finite-state automaton (FSA)
  • bounded number of memory states
  • step: input, current state determines next state & output
  ![Finite-State Automaton Diagram]
  
  Mod 3 counter state/output (Moore) machine
  \[ \delta(q_j, a) = (q_j, 2) \]
  
  models programs with a finite number of bounded registers
  reduce to 0 registers
Types of Machines (cont.)

- Pushdown Automaton (PDA)
  - finite control and a single unbounded stack
  - $L = \{a^n b^n : n \geq 1\}
  - $\delta(q_0, a, \epsilon) = (q_1, A)$

  models finite program + one unbounded stack of bounded registers

Types of Machines (cont.)

- Random access machine (RAM)
  - finite program and an unbounded, addressable random access memory of "registers"
  - models general programs
    - unbounded # of bounded registers
    - Simple 1-addr instructions

  Example:
  - $R_0 \leftarrow R_0 + R_1$
  - $L_0$ : $\text{INC } R_0$
  - $\text{DEC } R_0$
  - $\text{JMP } L_0$
  - $L_0$ : $\text{CONTINUE}$

Types of Machines (cont.)

- Turing Machine (TM)
  - finite control & tape of bounded cells unbounded in # to R
  - Input left adjusted on tape at start with blank cell terminating
  - current state, cell scanned determine next state & overprint symbol
  - control writes over symbol in cell and moves head 1 cell L or R
  - models simple "sequential" memory; no addressability
  - fixed amount of information (b bits) per cell

  $\delta(q, X) = (p, Y, R)$
Theory of Computation
Study of languages and functions that can be described by computation that is finite in space and time
• Grammar Theory
  ▪ Context-free grammars
  ▪ Right-linear grammars
  ▪ Unrestricted grammars
  ▪ Capabilities and limitations
  ▪ Application: programming language specification
• Automata Theory
  ▪ FA
  ▪ PDA
  ▪ Turing Machines
  ▪ Capabilities and limitations
  ▪ Characterizing “what is computable?”
  ▪ Application: parsing algorithms

Theory of Computation (cont’d)
• Computational Complexity Theory
  ▪ Inherent difficulty of “problems”
  ▪ Time/space resources needed for computation
  ▪ “Intractable” problems
  ▪ Ranking of problems by difficulty (hierarchies)
  ▪ Application: algorithm design, algorithm improvement, analysis

FSA Ex: Specifying/Recognizing C Identifiers
• Deterministic FA
  ▪ \( \Lambda = \{ a, ..., z, A, ..., Z, \_ \} \) \( \Delta = \{ 0, ..., 9 \} \)
  ▪ State diagram (labeled digraph)

• Regular Expression
  \( ( + a + ... + A + ...) \cdot ( + a + ... + A + ... + 0 + ... 9 )^* \)

• Right-Linear Grammar
  \begin{align*}
  S & \rightarrow aT | ... | zT \\
  T & \rightarrow aT | ... | zT \\
  & | AT | ... | ZT \\
  & | _T \\
  & | 0T | ... | 9T | T
  \end{align*}
Floating Constants

- A floating constant consists of an integer part, a decimal point, a fraction part, an e or E, an optionally signed integer exponent (and an optional type suffix …). The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing; …

- (The type is determined by the suffix; F or f makes it a float, L or l makes it a long double; otherwise it is double.)

---

Floating (cont’d)

- Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

---

Calculator Language

- Syntactic Classes
  - Numerals 3 4 0
  - Digits 0 1 9
  - Expressions 3*9 40-3*3
  - Commands 3*9= 40-3*3=

- Context-Free Grammar
  - C → E=
  - E → N
  - E → E+N
  - E → E-N
  - E → E*E
  - D → 0...

  terminals $\Sigma = \{e, +, -, *, 0, \ldots, 9\}$
  - rules $G$
  - variables $V = \{N, D, E, C\}$
  - start variable $= C$
  - grammar $G = (V, \Sigma, R, C)$

Note: no division & no decimal point
Calculator Language (cont’d)
- Syntax Trees—exhibit “phrase structure”
- Numerals N
- Expressions E
- Commands C

\[
\begin{align*}
N & \rightarrow 3 \mid 9 \\
E & \rightarrow N \mid E + N \\
C & \rightarrow * \\
\end{align*}
\]

Is this the parse you expected?

TM Ex: An “Algorithmically Unsolvable” Problem
- Q: Is there an algorithm for deciding if a given program \( \mathcal{P} \) halts on a given input \( x \)?
- A: No. There is no program that works correctly for all \( \mathcal{P}, x \)
- For the proof, we will need a simple programming language‡: \( \text{Nat}_{\mathcal{C}} \)—a simplified \( \text{C} \)
  - One data type: \( \text{nat} = \{0, 1, 2, \ldots\} \). All variables of type \( \text{nat} \)
  - All programs have one \( \text{nat} \) input and one \( \text{nat} \) output

‡We will later on use Turing Machines to model a “simple programming language”. \( \text{Nat}_{\mathcal{C}} \) is simpler to describe.

Unsolvable Problem (cont’d)
- Observations:
  - A standard \( \text{C} \) compiler can be modified to accept only \( \text{Nat}_{\mathcal{C}} \) programs as “legal”
  - Every \( \text{Nat}_{\mathcal{C}} \) program \( \mathcal{P} \) computes a function from natural numbers to natural numbers: \( \mathcal{f}_\mathcal{P} : \text{nat} \rightarrow \text{nat} \)
  - Note: \( \mathcal{f}_\mathcal{P} \) may not be defined for some inputs, i.e., it is a partial function

\[
\text{nat } \mathcal{P}(\text{nat } x) =
\begin{cases}
1 & \text{if } x=3 \\
0 & \text{if } x \neq 3
\end{cases}
\]

Ex: \( \mathcal{P} \) does not halt for some inputs
Unsolvable Problem (cont’d)

• Enumeration
  - A systematic list of all \texttt{NatC} programs $P_0, P_1, P_2, \ldots$
  - For program $P_i$ its index is called the program’s index
  - \texttt{program-index}: write out program as bit sequence in ASCII; interpret the bit sequence as a binary integer—its index
  - A program is just a string of characters!!!
  - \texttt{index-program}: given $i \geq 0$, convert to binary. Divide into 8-bit blocks. If such division is impossible (e.g., 3 bits) or if some block is not an ASCII code, or if the string is not a legal program, $P_i$ will be the default “junk” program \texttt{nat n; read(x); while(x=x) do x=x+1;write(x);} which is undefined (“diverges”) for every legal input.
  - Conclusions about enumeration $P_0, P_1, P_2, \ldots$
    - Given $n$ can compute $P_n$ with \texttt{NatC} program
    - Given $r$ can compute index $n$ such that $P = P_r$ with \texttt{NatC}.

Unsolvable Problem (cont’d)

• Unsolvability Result: Does $P_n$ halt on input $n$? Question cannot be settled by an algorithm.
• \textbf{Theorem:} Define the function $h: \texttt{nat} \rightarrow \texttt{nat}$ by
  - $h(n) = \begin{cases} 1 & \text{if } P_n \text{ halts on input } n \text{ then } 1 \text{ else } 0 \\ 0 & \text{otherwise} \end{cases}$
  - Then $h$ is not computable by any \texttt{NatC} program.
  - \textbf{Proof:} Proof by contradiction. Suppose (contrary to what is to be proved) that $h$ is computable by a program called $\texttt{halt}$. $\texttt{halt}$ has input variable $x$, and output variable $y$.
    By assumption (i.e., that it exists) it has the following behavior:
    
    $f_{\text{halt}}(x) = \begin{cases} 1 & \text{if } P_n \text{ halts on input } x \text{ then } 1 \text{ else } 0 \\ 0 & \text{otherwise} \end{cases}$

Unsolvable Problem (cont’d)

• Modify $\texttt{halt}$ to a \texttt{Nat} function $\texttt{nat\ halt(nat\ x)}$
  - Construct the following \texttt{Nat} program:
    - $\texttt{nat diagonal(nat n)}$
      - $\texttt{nat y; if halt(n)=0}$
        - $y:=1$;
      - else {
        - $y:=1$;
        - while ($y!=0$) do
          - $y:=y+1$;
        - return $y$;
      }$
  - Consequences $n$
    - If $\texttt{halt}$ is a legal program, so is “diagonal”
    - Therefore, diagonal has some index $n$ in the enumeration:
      * $P_n = \text{diagonal}$
Unsolvable Problem (cont’d)

- How does diagonal behave on its own index \( e \)?
- \( f_{\text{diagonal}}(e) = 1 \) \iff \( f_{\text{halt}}(e) = 0 \) \iff \( P_e \) does not halt on \( e \) \iff \( \text{diagonal} \) does not halt on \( e \)
- \( f_{\text{diagonal}}(e) = \text{undefined} \) \iff \( f_{\text{halt}}(e) = 1 \) \iff \( P_e \) halts on \( e \) \iff \( \text{diagonal} \) halts on \( e \)
- \( \therefore \) diagonal halts on \( e \) \iff diagonal does not halt on \( e \)
- Contradiction!!!
- \( \therefore \) program diagonal cannot exist \( \text{Q.E.D.} \)
- The “Halting Problem” is unsolvable
  - Undecidable, recursively undecidable, algorithmically undecidable, unsolvable—all synonyms