Automata, Grammars and Languages

Discourse 01

Introduction

Fundamental Questions

Theory of Computation seeks to answer fundamental questions about computing

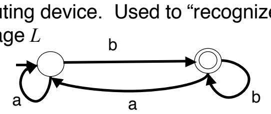
- What is computation?
  - Ancient activity back as far as Babylonians, Egyptians
  - Not precisely settled until circa 1936
- What can be computed?
  - Different ways of computing (C, Lisp, …) result in the same "effectively computable" functions from input to output?
- What cannot be computed?
  - 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 Σ
  - **Ex:** $L = \{TAA, TGA, TAG, \dots\}$ 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:** $S \to +SS$

    $S \to x$

    Context-Free Grammar (CFG)

Languages

$L_1 = \{aa, ab, ba, bb\} \quad \Sigma = \{a, b\}$

$L_2 = \{\varepsilon, a, aa, aaa, aaaa, \ldots\} \quad \Sigma = \{a\}$

$L_3 = \{e : e$ is a well-formed arithmetic expression in C$\}$

$\Sigma=\{$0-9,a-z,A-Z,+,-,*,/,(,),.,&,!,$\cdots\}$

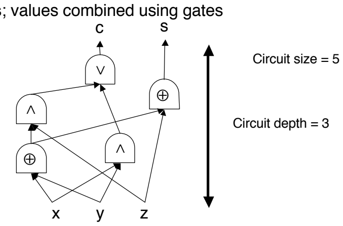
$L_4 = \{p : p$ is a well-formed C program$\}$ $\Sigma = \{\text{ASCII}\}$

$L_5 = \{p : p$ is a w.-f. C program that halts for all inputs$\}$

$L_6 = \{(x, y) : x$ is a decimal integer and y is its binary representation$\}$

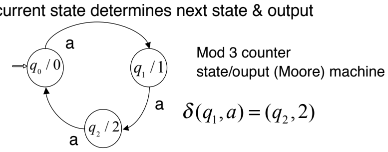
Types of Machines

- Logic circuit
  - memoryless; values combined using gates

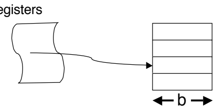


Types of Machines (cont.)

- Finite-state automaton (FSA)
  - bounded number of memory states
  - step: input, current state determines next state & output



- models programs with a *finite* number of *bounded* registers
  - reducible to 0 registers

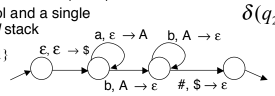


Types of Machines (cont.)

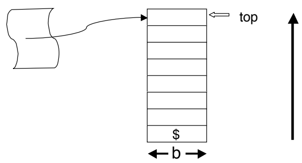
- Pushdown Automaton (PDA)
  - finite control and a single *unbounded* stack

$\delta(q_2, a, \varepsilon) = (q_2, A)$

$L = \{a^n b^n \# : n \geq 1\}$



models finite program + one *unbounded* stack of *bounded* registers

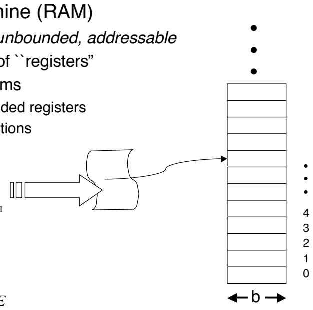


Types of Machines (cont.)

- 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

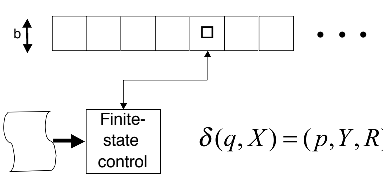
Example: $R_0 \leftarrow R_0 + R_1$

$L_0 : JMPZ\ R_1\ L_1$
$INC\ R_0$
$DEC\ R_1$
$JMP\ L_0$
$L_1 : CONTINUE$



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

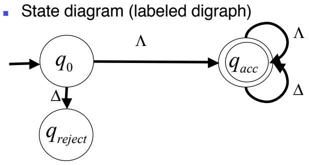
- Grammar Theory
  - Context-free grammars
  - Right-linear grammars
  - Unrestricted grammars
  - Capabilities and limitations
  - *Application: programming language specification*
- Automata Theory
  - FA
  - PDA
  - Turing Machines
  - Capabilities and limitations
  - Characterizing "what is computable?"
  - *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 Λ={a,…,z,A…,Z, _ } Δ={0,…,9}
  - State diagram (labeled digraph)



  - Regular Expression

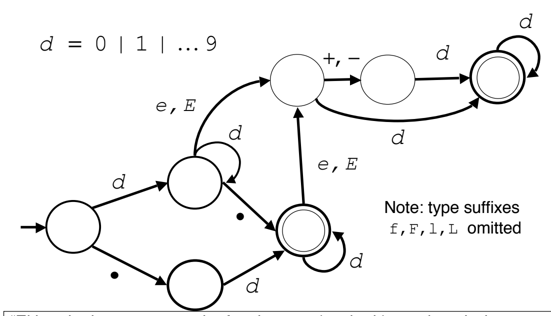
    (_+ a + … + A + …) · (_+ a + … + A + … + 0 + … 9) *
  - Right-Linear Grammar

```
S → aT | … | zT      T → aT | … | zT
  | AT | … | ZT        | AT | … | ZT
  | _T                 | 0T | … | 9T | _T
```

FSA Ex: C 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. ..."
  - --B. W. Kernighan and D.M. Ritchie, ***The C Programming Language***, Prentice-Hall, 1978
  - (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.)

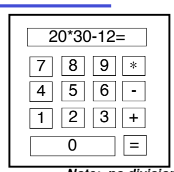
FSA Ex: C Floats (cont'd)



"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"

CFG Ex: A Calculator Language

- Syntactic Classes
  - Numerals `3 40`
  - Digits `0 1 9`
  - Expressions `3*9 40-3*3`
  - Commands `3*9= 40-3*3=`
- Context-Free Grammar



Note: no division & no decimal point

```
C →E=
E →N
  →E+N
  →E-N
  →E*N
N →ND
N →D
D →0...
  →9
```

rules R

terminals Σ = {=,+,–,*,0,...,9}

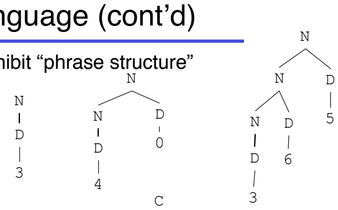
variables V = {N,D,E,C}

start variable = C

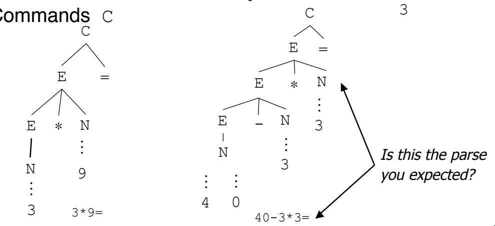
grammar $G = (V, \Sigma, R, C)$

Calculator Language (cont'd)

- Syntax Trees—exhibit "phrase structure"
- Numerals N

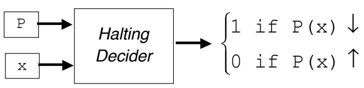


- Expressions E
- Commands C



TM Ex: An "Algorithmically Unsolvable" Problem

- Q: Is there an algorithm for deciding if a given program `P` halts on a given input `x`?



- A: No. There is *no program* that works correctly for *all* `P,x`
- For the proof, we will need a simple programming language‡: *NatC*—a simplified `C`
  - One data type: `nat` = {0,1,2, …}. All variables of type `nat`
  - All programs have one `nat` input and one `nat` output

‡We will later on use **Turing Machines** to model a "simple programming language". *NatC* is simpler to describe.

Unsolvable Problem (cont'd)

- Observations:
  - A standard C compiler can be modified to accept only *NatC* programs as "legal"
  - Every *NatC* program `P` computes a function from natural numbers to natural numbers. f_P : *nat* → *nat*
  - Note: f_P may not be defined for some inputs, i.e., it is a *partial function*

```
nat P(nat x)
{
    if (x=3)
        return(6);
    else {
     while(x=x) do x=x+1;
    return x;
         }
```

Ex: `P` does not *halt* for some inputs

Unsolvable Problem (cont'd)

- Enumeration
  - A systematic list of all *NatC* programs $P_0, P_1, P_2, \ldots$
  - For program P_i `i` is called the program's *index*
  - 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!!!
  - 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 `{nat x; 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 *NatC* program
    - Given `P` can compute index `n` such that $P = P_n$ with *NatC*

Unsolvable Problem (cont'd)

- Unsolvability Result: Does P_n halt on input `n` ? Question *cannot* be settled by an algorithm.
- ***Theorem***: Define the function h: `nat` → `nat` by
  - h `(x)` = *if* P_x *halts on input* `x` *then* `1` *else* `0`

  Then h is *not computable* by any *NatC* program.

  Proof: Proof by contradiction. *Suppose* (contrary to what is to be proved) that h is computable by a program called `halt`. `halt` has input variable `x`, and output variable `y`.

  By assumption (i.e., that it exists) it has the following behavior:

  f_{halt} `(x)` = *if* P_x *halts on input* `x` *then* `1` *else* `0`

Unsolvable Problem (cont'd)

- Modify `halt` to a *NatC function* `nat halt(nat x)`
- Construct the following *NatC* program:

```
nat diagonal(nat n)
{ nat y;
  if halt(n)=0
      y:=1;
  else {
         y:=1;
         while (y!=0) do
y:=y+1;}
  return y;
}
```

- Consequences
  - If **`halt`** is a legal program, so is "**`diagonal`**"
  - Therefore, **`diagonal`** has some index `e` in the enumeration:
    - P_e = `diagonal`

Unsolvable Problem (cont'd)

- How does `diagonal` behave on *its own index* `e` ?
- $f_{\texttt{diagonal}}(\texttt{e}) = 1 \Leftrightarrow f_{\texttt{halt}}(\texttt{e}) = 0 \Leftrightarrow \texttt{P}_{\texttt{e}}$ does not halt on `e` $\Leftrightarrow$ `diagonal` does not halt on `e`
- $f_{\texttt{diagonal}}(\texttt{e}) = \textit{undefined} \Leftrightarrow f_{\texttt{halt}}(\texttt{e}) = 1 \Leftrightarrow \texttt{P}_{\texttt{e}}$ halts on `e` $\Leftrightarrow$ `diagonal` halts on `e`
- ∴ `diagonal` halts on `e` ⇔ `diagonal` does not halt on `e`
- Contradiction!!!
- ∴ program `diagonal` cannot exist Q.E.D.
- The "Halting Problem" is unsolvable
  - Undecidable, recursively undecidable, algorithmically undecidable, unsolvable—all synonyms