CPSC 121 - Models of Computation
Module 11. Models of Computation

Prof. Karina Mochetti

2020.W1
What is new!

http://www.cs.ubc.ca/~mochetti/CPSC121.html
Goals

- Trace execution of an instruction through a working computer in a logic simulator.
- Feel confident that, given sufficient time, you could understand how the circuit executes machine-language instructions.
- Explain the difference between a DFA and a non-deterministic finite-state automaton (NFA).
- Translate a regular expression into a NFA that accepts exactly the strings matched by the regular expression.
- Give an example of a problem that we can prove we can not solve, and explain at a high level how the proof works.
Disjoint Sets

Two sets are called disjoint if, and only if, they have no elements in common.

A and B are disjoint ⇔ \( A \cap B = \emptyset \)

Partition of a Set

A finite or infinite collection of nonempty sets \( \{A_1, A_2, A_3, \ldots \} \) is a partition of a set A if, and only if:
1. A is the union of all the \( A_i \);
2. The sets \( A_1, A_2, A_3, \ldots \) are mutually disjoint.
Power Set

Given a set $A$, the power set of $A$, denoted $\mathcal{P}(A)$, is the set of all subsets of $A$.

Example: $\mathcal{P}(\{x, y\}) = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$

Tuple

Let $n$ be a positive integer and let $x_1, x_2, \cdots, x_n$ be elements. The ordered $n$-tuple, $(x_1, x_2, \cdots, x_n)$, consists of $x_1, x_2, \cdots, x_n$ together in the given order.

Cartesian Product

The Cartesian product of $A_1, A_2, \cdots, A_n$ denoted $A_1 \times A_2 \times \cdots \times A_n$, is the set of all ordered $n$-tuples $(a_1, a_2, \cdots, a_n)$ where $a_1 \in A_1, a_2 \in A_2, \cdots, a_n \in A_n$. 
Function

A function $f$ from a set $X$ to a set $Y$, denoted $f : X \rightarrow Y$, is a relation from $X$, the domain, to $Y$, the co-domain, that satisfies two properties:

1. every element in $X$ is related to some element in $Y$, and
2. no element in $X$ is related to more than one element in $Y$.

1. Every element of $X$ has an arrow coming out of it.
2. No element of $X$ has two arrows coming out of it that point to two different elements of $Y$.
Function Equality

If $F : X \rightarrow Y$ and $G : X \rightarrow Y$ are functions, then $F = G$ if, and only if, $F(x) = G(x)$ for all $x \in X$.

Logarithm

Let $b$ be a positive real number with $b \neq 1$. For each $x \in \mathbb{R}^+$, the logarithm with base $b$ of $x$ is the exponent to which $b$ must be raised to obtain $x$.

$$\log_b x = y \iff b^y = x$$

Boolean Function

An ($n$-place) Boolean function $f$ is a function whose domain is the set of all ordered $n$-tuples of 0's and 1's and whose co-domain is the set $\{0, 1\}$.
Very very well done overall!!! \o/

Just be careful:

- A term $f(z)$ cannot be a subset, a term is an element and subset is an operation between two sets!
A working Computer: Von-Neumann Architecture

- A **processing unit** that contains an arithmetic logic unit and processor registers
- A **control unit** that contains an instruction register and program counter
- **Memory** that stores data and instructions
- **External mass storage**
- **Input and output** mechanisms

![Diagram of a computer architecture](image)
Von-Neumann Architecture

**Memory**

- Contains both instructions and data.
- Divided into a number of memory locations, like a list or array.
- Each memory location contains a fixed number of bits (most commonly 8 bits, or 1 byte).
- Values that use more than 8 bits are stored in multiple consecutive memory locations.

<table>
<thead>
<tr>
<th>K</th>
<th>m</th>
<th>18,061,982</th>
</tr>
</thead>
<tbody>
<tr>
<td>01001011</td>
<td>01101101</td>
<td>...</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>420</td>
</tr>
<tr>
<td></td>
<td></td>
<td>421</td>
</tr>
<tr>
<td></td>
<td></td>
<td>422</td>
</tr>
<tr>
<td></td>
<td></td>
<td>423</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Arithmetic Logic Unit (ALU)

- Performs arithmetic and logical operations.
- It can detect overflow and other status.
- Opcode usually refers to the operation (+, -, *, /, and, or, ...)

![Diagram of ALU]

- Operand A
- Operand B
- Opcode
- Status
- Result Y
Control Unit

- Decides which instructions to execute.
- Executes these instructions sequentially. Not quite true, but this is how it appears to the user.
- It contains the Program Counter (PC), a special register that has the memory address of the next instruction.

```
PC

irmovl 0x3,%eax
irmovl 0x35,%ebx
subl %eax,%ebx
halt
```
Our working Computer

- Implements the design presented in the textbook by Bryant and O’Hallaron (used for CPSC 213/313).
- A small subset of the IA32 (Intel 32-bit) architecture.

It has:

- 12 types of instructions.
- 1 program counter register (PC).
- 8 general-purpose 32-bit registers, used for values that we are currently working with.

Registers are faster to access them memory and are used for values that we are currently working with.
Example 1

irmovl 0x1A, %ecx

- This instruction stores a constant in a register.
- In this case, the value 1A (in hexadecimal, 26 in decimal) is stored in the register %ecx.
Example 2

\texttt{subl \%eax, \%ebx}

- The \texttt{subl} instruction subtracts its arguments.
- The names \texttt{\%eax} and \texttt{\%ebx} refer to two registers.
- This instruction takes the value contained in \texttt{\%eax}, subtracts it from the value contained in \texttt{\%ebx}, and stores the result back in \texttt{\%ebx}.

\[ \texttt{ebx} = \texttt{ebx} - \texttt{eax} \]
Example 3
jge $1000

- The **conditional jump** instruction.
- It checks to see if the result of the last arithmetic or logic operation was zero or positive (**G**reater than or **E**qual to 0).
- If so, the next instruction is the instruction stored in memory address 1000 (hexadecimal).
- If not, the next instruction is the instruction that follows this jge instruction.
irmovl 0x3,%eax
irmovl 0x35, %ebx
subl %eax, %ebx
halt
A working Computer

How does the computer know which instruction does what?

- Each instruction is a sequence of 8 to 48 bits.
- Some of the bits tell it which instruction it is.
- Other bits tell it what operands to use.
- These bits are used as select inputs for several multiplexers.
Example 1

\texttt{subl} \ %\texttt{eax}, \ %\texttt{ebx}

This instruction is represented in hexadecimal by

\texttt{6103}

- Arithmetic or logic operation (the use of 6 to represent them instead of 0 or F or any other value is completely arbitrary).
- Subtraction
- \texttt{eax}
- \texttt{ebx}
Example 2

irmovl 0xfacade, %ecx

This instruction is represented in hexadecimal by

\[ \text{30F100FACADE} \]

- Move constant into register.
- Ignored
- no register here
- ecx
- Constant value in hexadecimal, 0xFACADE
This CPU divides the execution into 6 stages:

- **Fetch**: read instruction and decide on new PC value
- **Decode**: read values from registers
- **Execute**: use the ALU to perform computations
- **Memory**: read data from or write data to memory
- **Write-back**: store value(s) into register(s).
- **PC update**: store the new PC value.

Not all stages do something for every instruction.
**Example 1**

```
irmovl 0xfacade, %ecx
```

1. **Fetch:** current instruction ← 30F100FACADE  
   next PC value ← current PC value + 6

2. **Decode:** nothing needs to be done

3. **Execute:** \( valE \leftarrow valC \)

4. **Memory:** nothing needs to be done

5. **Write-back:** \( \%ecx \leftarrow valE \)

6. **PC update:** PC ← next PC value
Executing an Instruction

Example 2

\texttt{subl \%eax, \%ebx}

1. **Fetch:** current instruction \(\leftarrow 6103\)
   
   next PC value \(\leftarrow\) current PC value + 2

2. **Decode:** \(\text{valA} \leftarrow \%\text{eax}\)
   
   \(\text{valB} \leftarrow \%\text{ebx}\)

3. **Execute:** \(\text{valE} \leftarrow \text{valB} - \text{valA}\)

4. **Memory:** nothing needs to be done

5. **Write-back:** \(\%\text{ebx} \leftarrow \text{valE}\)

6. **PC update:** PC \(\leftarrow\) next PC value
The greater irony was that RollerCoaster Tycoon otherwise stood as a monument to what a single person can accomplish in programming. Written almost entirely in assembly code (like Sawyer's previous Transport Tycoon), RollerCoaster Tycoon and its sequel squeezed and re-squeezed the processors of the time to simulate rides, economies, and up to thousands of visitors and their states of mind. Churning through so many numbers in real-time without hitching demanded a lean, uncompromising approach and not the slower, more user-friendly C family of languages. And in ultra-lean assembly, where letters stand in for ones and zeroes, one speaks directly to the processor.

Read more on https://www.pcgamesn.com/rollercoaster-tycoon/code-chris-sawyer
Questions?

Ask CPSC 121

http://www.cs.ubc.ca/~mochetti/askCPSC121.html
DFAs and Regular Expressions

DFA definition

A DFA is a 5-tuple \((\Sigma, S, s_0, F, \delta)\) where

- \(\Sigma\) (sigma) is a finite set of characters (input alphabet)
- \(S\) is a finite set of states.
- \(s_0 \in S\) is the initial state.
- \(F \subseteq S\) is the set of accepting states.
- \(\delta : S \times \Sigma \rightarrow S\) (delta) is the transition function.

\[\begin{align*}
\Sigma &: \{a, b, c\} \\
S &: \{1, 2, 3\} \\
s_0 \in S &: 1 \\
F \subseteq S &: \{1\} \\
\delta &: S \times \Sigma \rightarrow S \\
\delta(2, a) &= 3, \text{ for example}
\end{align*}\]
In lab 7, you wrote regular expressions matching the patterns we gave you.

Regular expressions are very useful when you need to read and validate input.

Many modern programming languages provide functions that allow you to manipulate regular expressions: Dr. Racket, Java, Python, Perl, etc.

**Theorem**

Every set of strings matched by a regular expression can be recognized by a DFA.
How do we build the DFA given a regular expression?

1. First we build a NFA (non-deterministic finite-state automaton) for the regular expression.
2. Then we convert the NFA into a DFA.

**DFA**

DFA refers to **Deterministic finite automaton**, a finite-state automaton in which for an input symbol, there is a single resultant state.

**NFA**

NFA refers to **Nondeterministic Finite Automaton**, in which there is more than one possible transition from one state on the same input symbol.
NFA is like a DFA but

- There can be multiple arrows with the same label leaving from a state.
- There can be arrows labelled $\varepsilon$ (empty string) that we can take without reading the next input character.
- We can sometimes choose which state to go to.
- A NFA accepts a string if at least one sequence of choices leads to an accepting state.
What regular expression corresponds to the strings that this NFA accepts?

- a. $\varepsilon | ab$
- b. $\varepsilon | abaa$
- c. $\varepsilon | ab | abaa$
- d. $\varepsilon | ab | (aba)^+a$
- e. None of the above.
Theorem

We can transform every regular expression into a NFA with
- Exactly one accepting state
- No arcs pointing to the initial state
- No arcs leaving the accepting state

Fact!
Every regular expression can be rewritten to use only the following:
- The empty string $\varepsilon$
- Individual characters
- The operators $|$, $\ast$, and string concatenation.
Rewrite the following expressions to use only the options listed on the operators |, *, and string concatenation:

- $a?$:
- $a+$:
- $a\{3, 5\}$:
- $\backslash d$:
Rewrite the following expressions to use only the options listed on the operators |, *, and string concatenation:

- $a?$: $\varepsilon | a$
- $a+$:
- $a\{3,5\}$:
- $\backslash d$: 
DFAs and Regular Expressions

Rewrite the following expressions to use only the options listed on the operators |, *, and string concatenation:

- $a?$: $\varepsilon | a$
- $a+$: $aa^*$
- $a\{3, 5\}$:
- $\backslash d$: 
Rewrite the following expressions to use only the options listed on the operators $\mathsf{|}$, $\mathsf{*}$, and string concatenation:

- $a?$: $\varepsilon|a$
- $a+$: $aa^*$
- $a\{3,5\}$: $aaa|aaaa|aaaaa$
- $\mathsf{\backslash d}$:
Rewrite the following expressions to use only the options listed on the operators |, *, and string concatenation:

- $a?: \varepsilon|a$
- $a+: aa^*$
- $a\{3, 5\}: aaa|aaaa|aaaaaa$
- $d: 0|1|2|3|4|5|6|7|8|9$
Theorem

We can transform every regular expression into a NFA with
- Exactly one accepting state
- No arcs pointing to the initial state
- No arcs leaving the accepting state

Proof: by induction on the characters of the regular expression.

Assume that the regular expression only uses the operators |, *, and string concatenation.
Regular Expressions into a NFA

**Base:**

- The NFA for the expression that matches the empty string \((n=0)\): 
  ![Diagram for empty string NFA]

- The NFA for the expression that matches no string \((n=0)\):
  ![Diagram for no string NFA]

- The NFA for the expression that matches a single character \(a\) \((n=1)\):
  ![Diagram for single character NFA]
**Induction Hypothesis**: Consider a regular expression with \( n \) characters. Suppose the theorem holds for every regular with \( k < n \) characters.

**Induction Step**:  
- Given an expression \( E \), we consider three cases: it has the operator |, **string concatenation** or *. If it doesn’t, then refer to the base cases.

- The NFA for the expression \( E_1|E_2 \) where \( E_1, E_2 \) are regular expressions with less than \( n \) characters:
Regular Expressions into a NFA

- The NFA for the expression $E_1E_2$ where $E_1$, $E_2$ are regular expressions with less than $n$ characters:

- The NFA for the expression $E^*$ where $E$ is a regular expression with less than $n$ characters:
Regular Expressions into a NFA

\[(a|b)^*c\]
Regular Expressions into a NFA

\[ a \quad b \quad c \]

\[ \text{Diagram showing transitions for } a, b, \text{ and } c. \]
Regular Expressions into a NFA

\[(a|b) \quad c\]
Regular Expressions into a NFA

\[(a|b)^* c\]
Regular Expressions into a NFA

\[(a|b)^*c\]
Regular Expressions into a NFA

(a|b)*c

Diagram: A non-deterministic finite automaton (NFA) for the regular expression (a|b)*c.
How do we transform a NFA into a DFA?

- A DFA that reads a string with $n$ characters ends up in exactly one state.
- A NFA that reads a string with $n$ characters may end up in many different stages.

**Can we figure out which states?**
Which state(s) will the following NFA end up in after reading the string ab?

a. S1 only
b. S6 only
c. S3 or S7
d. S4 or S6
e. None of the above.
So we build the DFA as follows:

- The DFA has one state for every subset of the states of the NFA (so $2^n$ states in total).
- If the DFA is in state $\{S_1, S_2, \ldots, S_k\}$, and it sees a character $x$, then the new state is the state that contains every NFA state that we can get to from one of $S_1, S_2, \ldots, S_k$ upon reading $x$.
- A state of the DFA is accepting if it contains the accepting state of the NFA.
We have discussed several models of computation in the course:

- Combinational circuits
- Sequential circuits (the working computer).
- DFAs

Things computer scientists (we) like to know about their (our) computational models is:

- What can they do?
- **What can they not do?**
What a Sequential Circuits cannot do

### Halting Problem

Given a program \( P \) and an input \( I \), will \( P \) halt if we run it on input \( I \)?

### Theorem

It is not possible to write a program that solves the halting problem.

**Proof 1:** lazy proof by googling it!

- [https://www.youtube.com/watch?v=92WHN-pAFCs](https://www.youtube.com/watch?v=92WHN-pAFCs)
Theorem

It is not possible to write a program that solves the halting problem.

Proof 2: by contradiction.

- Suppose this program exists.
- Let us call it will-halt.
- We use this function or method to write the following function or method:

  define (paradox input)
  
  (if (will-halt input input)
   (paradox input) ; go into an infinite recursion true))

- What happens when we call this program with itself as input?
- If it halts, then will-halt returns true, and so it won’t halt.
- If it doesn’t halt, then will-halt returns false, and so it will halt.
- So we always we end up with a contradiction.
- Therefore this program does not exist. ■
Questions?

Ask CPSC 121

http://www.cs.ubc.ca/~mochetti/askCPSC121.html
The End!

That's all Folks!
Memories are commonly divided into locations with 8 bits (1 byte) each. Complete the memory positions below assuming that all values are sequentially stored and characters have 1 byte, integers have 4 bytes and doubles have 8 bytes.

- _____: character 'H'
- _____: character 'I'
- _____: integer 123
- _____: character '?'
- _____: integer 0
- 0x628A: double -18.0
- _____: integer 42
- _____: double 0.3333
- _____: integer -999999
- _____: double 1806.1982
Memories are commonly divided into locations with 8 bits (1 byte) each. Complete the memory positions below assuming that all values are sequentially stored and characters have 1 byte, integers have 4 bytes and double have 8 bytes.

- 0x627F: character 'H'
- 0x6280: character 'I'
- 0x6281: integer 123
- 0x6285: character '?'
- 0x6286: integer 0
- 0x628A: double -18.0
- 0x6292: integer 42
- 0x6296: double 0.3333
- 0x629E: integer -999999
- 0x62A2: double 1806.1982