Today: Modeling Computations part 1

- How do we model/describe computation?
- What are the limits of these ways of modeling?

Method 1: Functions

- The lambda calculus focuses on functions without side effects — Alonzo Church, 1930s
- \( \text{fun } f(x) = x^2 + 2x \)
  - Is equivalent to \( f: x \mapsto x^2 + 2x \)
  - Is equivalent to \( f = \lambda x. x^2 + 2x \)
- \( f(a+3) = (a+3)^2 + (a+3) \times 12 \) done as
  - \( (\lambda x. x^2 + 2x)(a+3) \)
  - Use body of \( \lambda \) function and replace \( x \) with \( a+3 \)
- \( (\lambda x. x^2 + 2x)(a+3) = (a+3)^2 + 2(a+3) \)
**The λ calculus is a universal model of computation**

- We can use the λ calculus to write a program for any algorithm.
- Won't look like a program written in e.g., C or Java.
- Doesn't include software engineering concepts.
- The λ calculus is the base for functional languages like Haskell

---

**Method 2: Formal Languages**

- **Formal languages** are sets of strings.
- We use **grammar rules** to define them.
- Different families of grammar rules lead to different **families of languages**
- E.g., **Regular Languages** (expressible by basic regular expressions) — easy to define and use
- **Context-Free Languages** are what we use to define programming languages
  - Much richer languages than regular languages
  - But also harder to define and much harder to use.
  - (We write parsers.)
Method 3: Abstract Machines ("Automata")

- We don't build a physical machine, we define what the states and transitions and memory a machine requires.
- Then we can code one up
- Simpler machines are easier to describe and use, less powerful than more complicated machines.
- **Finite State Automata (FSA)** have a limited number of internal states, fixed pattern of state changes, and no external memory.
- **Push-Down Automata (PDA)** add a stack to FSAs
- **Turing Machines (TMs)** replace the stack with a "tape" (unbounded array of symbols).

TMs are a Universal Model of Computation

- We can write TM programs equivalent to any program in any programming language we know.
- (But it's like writing in the worst assembler language ever, blechh.)
- This includes the lambda calculus.
- But you can also implement TMs in the $\lambda$ calculus.
Church-Turing Thesis

- A claim about the nature of "effective computations"
- Computations that can be done "mechanically"

- The claim is that any effective computation on the natural numbers can be done using Turing Machines or the λ Calculus.
- Not a provable claim — more an extension of observations about reality.

A Look at Formal Languages

- In CS, we generally use languages by defining syntax rules.
- Natural languages have much more complicated syntax rules (plus a lot of semantic knowledge to say when things aren't reasonable).
- <subject> crawls into <object>
  - "Cat crawls into box" vs "Box crawls into cat"
The format of allowed syntax rules makes a big difference in how complicated the language can be

- Can we express some concept with it?
- Can we identify text as adhering to the language?
  - I.e., can we parse?

**Grammar-oriented descriptions of languages**

- A grammar $G = (V, T, S, P)$ consists of
  - $V$: The vocabulary, a set of symbols
  - $T \subseteq V$: The *terminal symbols* (basic units of the lang)
  - $S \in V$: The *start symbol*
  - $P$: A set of *production rules*

---

**Grammar Example**

- $V = \{ (, ), E \}$ — set of 3 symbols
- $T = \{ (, ) \}$ — the two basic symbols
- $S = E$ — productions start with $E$
- $P = \{ E \rightarrow EE, E \rightarrow (E), E \rightarrow \epsilon \}$  \( \epsilon = \) empty string
  - A rule $lhs \rightarrow rhs$ rewrite rule says that the $lhs$ can be replaced by the $rhs$
  - Exist other formats, e.g., $< E > ::= < E > < E >$
    - (Backus-Naur form)
      
      $$
      E \Rightarrow (E) \Rightarrow ( )
      
      E \Rightarrow (*) \text{ you can generate } ()
      $$

---

© J. Sasaki 2021
Examples of Productions

- \( E \rightarrow \varepsilon \) (i.e., "")
- \( E \rightarrow (E) \rightarrow ( ) \)
- \( E \rightarrow EE \rightarrow (E)E \rightarrow (( ))E \rightarrow ((( )))E \rightarrow ((( )))() \)

\( L(G) \) = the language generated by a grammar

= The set of all terminal symbol strings
that can be obtained by productions starting
with the start symbol

\[ L(G) = \{ w \in T^* \mid S \Rightarrow^* w \} \]

- Informally (i.e., in English), \( L(G) \) for our example grammar is .... ?

---

Noam Chomsky Language Classification (1956)

- Four types of languages, defined by what kinds of production rules they can use
- Type 0: Unrestricted
  - Type 1: Context-Sensitive
  - Type 2: Context-Free
  - Type 3: Regular

- Type 3: Regular languages
  - Rule formats \( A \rightarrow aB \) and \( A \rightarrow B \) where \( A \) and \( B \) are nonterminal symbols and \( a \) is a terminal symbol\(^1\)
  - \( (A, B \in V - T, a \in T) \)

\(^1\) \( A \rightarrow aB \) for right-regular grammars, \( A \rightarrow B \) for left-regular grammars
Type 2: Context-Free Languages

- Rule format: \( A \rightarrow \beta \)
- \( A \) is a nonterminal symbol (\( A \in V - T \))
- \( \beta \) is a string of terminal &/or nonterminals (\( \beta \in V^* \))
- You can replace \( A \) anytime it appears

Type 1: Context-Sensitive Languages

- Rule format: \( \beta \) \( A \gamma \rightarrow \delta \) (\( A \in V - T \), \( \beta \), \( \gamma \), \( \delta \in V^* \)) and \( \text{len}(lhs) \leq \text{len}(rhs) \).
- You can replace \( A \) but only within the context \( \beta...\gamma \).

Type 0: Unrestricted Languages

- No restrictions on grammar: \( \beta \rightarrow \gamma \)

Examples of Grammars

What language family?

\( G_1 \): \( V = \{ ( ), [ ] \}, \ T = \{ ( ), \}, \ S = S, \ P = \{ S \rightarrow \epsilon | ( S ) | S S \} \) — means \( \{ S \rightarrow \epsilon, S \rightarrow ( S ), S \rightarrow S S \} \)

\( G_2 \): \( V = \{ 0, 1, A, B \}, \ T = \{ 0, 1 \}, S = A, \ P = \{ A \rightarrow 0 A | 1 B | \epsilon, B \rightarrow 0 B | 1 A \} \)

\( G_3 \): \( V = \{ 0, 1, 2, A, B, C \}, \ T = \{ 0, 1, 2 \}, S = A, \ P = \{ A \rightarrow 0 B A 2 | 0 1 2, B 0 \rightarrow 0 B, B 1 \rightarrow 1 1 \} \)
G₃ is a tricky language

- \( A \Rightarrow 0 \ 1 \ 2 \)
- \( A \Rightarrow 0 \ B \ \bar{A} \ 2 \Rightarrow 0 \ B \ \bar{O} \ 1 \ 2 \ 2 \Rightarrow 0 \ 0 \ B \ 1 \ 2 \ 2 \Rightarrow 0 \ 0 \ 1 \ 1 \ 2 \ 2 \)
- \( A \Rightarrow 0 \ B \ \bar{A} \ 2 \Rightarrow 0 \ B \ 0 \ B \ A \ 2 \ 2 \Rightarrow 0 \ B \ 0 \ B \ 0 \ 1 \ 2 \ 2 \ 2 \)
  \( \Rightarrow 0 \ B \ 0 \ 0 \ B \ 1 \ 2 \ 2 \ 2 \Rightarrow 0 \ 0 \ B \ 0 \ B \ 1 \ 2 \ 2 \ 2 \Rightarrow 0 \ 0 \ 0 \ B \ B \ 1 \ 2 \ 2 \ 2 \)
  \( \Rightarrow 0 \ 0 \ 0 \ B \ 1 \ 1 \ 2 \ 2 \ 2 \Rightarrow 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 2 \ 2 \ 2 \)

- General case: \( L(G_3) = \{ 0^n1^n2^n \mid n \geq 1 \} \)

\[ L = \left\{ 0^n1^n2^n \mid n \geq 1 \right\} \]

\[ S \to OS1 \]
\[ S \to \epsilon \]

Next time — Abstract Automata

- Grammars concentrate on \textit{generating} strings in a language.
- Machine-oriented techniques concentrate on designing abstract devices (automata) to \textit{recognize} or \textit{accept} strings in a language.
- A lot of theoretical CS looks at automata
  - Can we (or how hard is it to) recognize strings using this class of automata?
  - Do these automata recognize the same language?