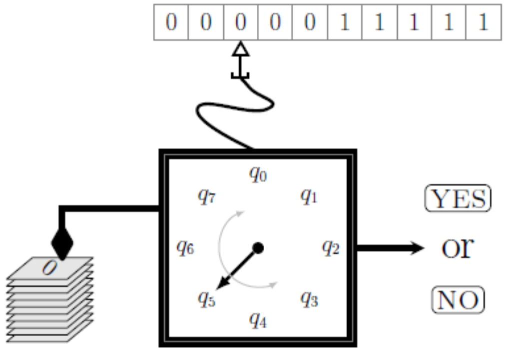
Context Free Grammars

Reading

- Malik Magdon-Ismail. Discrete Mathematics and Computing.
  - Chapter 25

Adding Memory

- DFAs have no scratch paper. It's hard to compute entirely in your head.
- **Stack Memory.** Think of a file-clerk with a stack of papers.
- The clerk's capabilities:
  - see the top sheet;
  - remove the top sheet (*pop*)
  - *push* something new onto the top of the stack.
  - no access to inner sheets without removing top.



- DFA with a stack is a *pushdown automaton (PDA)*
- How does the stack help to solve $\mathcal{L}_{0^n1^n} = \{0^{\bullet n}1^{\bullet n} | n \geq 0\}$?
  1. When you read in each 0, write it to the stack.
  2. For each 1, pop the stack. At the end if the stack is empty, accept.
- The memory allows the automaton to "remember" n.

Overview

- Solving a problem by listing out the language.
- Rules for Context Free Grammars (CFG).
- Examples of Context Free Grammars.
  - English.
  - Programming.
- Proving a CFG solves a problem.
- Parse Trees.
- Pushdown Automata and non context free languages.

Recursively Defined Language: Listing a Language

$$\mathcal{L}_{0^n1^n} = \{0^{\bullet n}1^{\bullet n} | n \geq 0\}$$

- How do we define this language recursively?

$\varepsilon \in \mathcal{L}_{0^n1^n}$ [**basis**]

$x \in \mathcal{L}_{0^n1^n} \rightarrow 0x1 \in \mathcal{L}_{0^n1^n}$ [**constructor rule**]

Nothing else is in $\mathcal{L}_{0^n1^n}$ [**minimality**]

- To test if $0010 \in \mathcal{L}_{0^n1^n}$: generate strings in order of length and test each for a match:

$$\varepsilon \rightarrow 01 \rightarrow 0011 \rightarrow 000111$$

NO

- A Context Free Grammar is like a recursive definition
  1. $S \rightarrow \varepsilon$
  2. $S \rightarrow 0S1$

Rules for Context Free Grammars (CFGs)

- Production rules of CFGs
  1. $S \rightarrow \varepsilon$
  2. $S \rightarrow 0S1$
- Each production rule has the form

$$\begin{array}{ccc} \texttt{variable} & \rightarrow & \texttt{expression} \\ \uparrow & & \uparrow \\ P, Q, R, S, T, \ldots & & \text{string of variables and terminals} \end{array}$$

$$\begin{array}{ccccccccc} S & \overset{2}{\Rightarrow} & 0S1 & \overset{2}{\Rightarrow} & 00S11 & \overset{2}{\Rightarrow} & 000S111 & \overset{2}{\Rightarrow} & 0000S1111 \\ {\scriptstyle 1}\Downarrow & & {\scriptstyle 1}\Downarrow & & {\scriptstyle 1}\Downarrow & & {\scriptstyle 1}\Downarrow & & {\scriptstyle 1}\Downarrow \\ \varepsilon & & 01 & & 0011 & & 000111 & & 00001111 \end{array}$$

Rules for Context Free Grammars (CFGs), cont'd

- Production rules of CFGs
  1. $S \rightarrow \varepsilon$
  2. $S \rightarrow 0S1$
- Each production rule has the form

`variable` → `expression`

$P, Q, R, S, T, \ldots$ — string of variables and terminals

1. Write down the start variable (form the first production rule, typically *S*).
2. Replace *one* variable in the current string with the expression from a production rule that *starts* with that variable on the left.
3. Repeat step 2 until no variables remain in the string.

- "Replace variable with expression, no matter where (independent of context)"
- Shorthand: 1: $S \rightarrow \varepsilon | 0S1$

Language of Equality, CFG_{bal}

- How do we write the CFG for equality?

$$\text{CFG}_{\text{bal}} \qquad 1: S \to \varepsilon | 0S1S | 1S0S$$

- A derivation of 0110 in CFG_{bal} (each step is called an inference).

$$S \overset{1}{\Rightarrow} 0S1S \overset{1}{\Rightarrow} 0S11S0S \overset{1}{\Rightarrow} 0\varepsilon 11S0S \overset{*}{\Rightarrow} 0110$$

- Notation

$$S \overset{*}{\Rightarrow} 0110$$

($\overset{*}{\Rightarrow}$ means "A derivation starting from S yields 0110")

- Distinguish S from a *mathematical* variable (e.g. x)

$$0S1S \qquad \text{versus} \qquad 0x1x$$

- Two S's are replaced independently. Two x's must be the same, e.g. $x = 11$

A CFG for English

Rensselaer

1. $S \rightarrow$ <phrase><verb>
2. <phrase> → <article><noun>
3. <article>→A␣|The␣
4. <noun>→cat␣|dog␣
5. <verb>→walks. | runs. | walks. ␣S | runs.␣S

$S \overset{1}{\Rightarrow}$ <phrase><verb>

$\overset{5}{\Rightarrow}$ <phrase>walks

$\overset{2}{\Rightarrow}$ <article><noun>walks

$\overset{3}{\Rightarrow}$ A␣ <noun>walks

$\overset{4}{\Rightarrow}$ A␣ cat␣ walks

A CFG for Programming

1. $S \rightarrow$ <stmt>;S|<stmt>;
2. <stmt> → <assign>|<declare>
3. <declare>→int␣<variable>
4. <assign>→<variable>=<integer>
5. <integer>→ <integer><digit>|<digit>
6. <digit>→0|1|2|3|4|5|6|7|8|9
7. <variable>→x|x<variable>

Constructing a CFG to Solve a Problem

- Going back to our language of balanced 0s and 1s

$$\mathcal{L}_{bal} = \{strings\ with\ an\ equal\ number\ of\ 1s\ and\ 0s\}$$

$$001011010110 = 0 \bullet \underbrace{\mathbf{0101}}_{\text{in } \mathcal{L}_{bal}} \bullet 1 \bullet \underbrace{\mathbf{010110}}_{\text{in } \mathcal{L}_{bal}}$$

$$= 0S1S$$

- Every large string in $\mathcal{L}_{bal}$ can be obtained (recursively) from smaller ones.

$$S \rightarrow \varepsilon | 0S1S | 1S0S$$

- We must prove that:
  - Every string generated by this CFG is in $\mathcal{L}_{bal}$
  - Every string in $\mathcal{L}_{bal}$ can be derived by this grammar

Proving a CFG Solves a Problem

Rensselaer

- Going back to our language of balanced 0s and 1s
$$\mathcal{L}_{bal} = \{strings\ with\ an\ equal\ number\ of\ 1s\ and\ 0s\}$$
- Every large string in $\mathcal{L}_{bal}$ can be obtained (recursively) from smaller ones.
$$S \to \varepsilon | 0S1S | 1S0S$$
- Proof. [*Every derivation in CFG$_{bal}$ generates a string in* $\mathcal{L}_{bal}$]
  - Use strong induction on the length of the derivation
    - i.e., number of production rules invoked
  - Base Case. length-1 derivation gives ε.
  - Induction step. The derivation starts in one of two ways:
    1. $S \to 0\underset{x}{\underset{*\Downarrow}{S}}1\underset{y}{\underset{*\Downarrow}{S}}$
    2. $S \to 1\underset{x}{\underset{*\Downarrow}{S}}0\underset{y}{\underset{*\Downarrow}{S}}$
  - The derivations of x and y are shorter.
  - By the induction hypothesis, $x, y \in \mathcal{L}_{bal}$, so the final strings are in $\mathcal{L}_{bal}$

Proving a CFG Solves a Problem

- Going back to our language of balanced 0s and 1s
$$\mathcal{L}_{bal} = \{strings\ with\ an\ equal\ number\ of\ 1s\ and\ 0s\}$$
- Every large string in $\mathcal{L}_{bal}$ can be obtained (recursively) from smaller ones.
$$S \rightarrow \varepsilon | 0S1S | 1S0S$$
- Proof. [*Every derivation in* CFG_{bal} *generates a string in* $\mathcal{L}_{bal}$]
- Proof. [*Every string in* $\mathcal{L}_{bal}$ can be derived within CFG_{bal}]
  - Use strong induction on the length of the derivation
    - i.e., number of production rules invoked
  - Base Case. length-1 derivation gives ε.
  - Induction step. Any string $w \in \mathcal{L}_{bal}$ has one of two forms:
    1. $w = 0w_1 1w_2$
    2. $w = 1w_1 0w_2$
  - where w_1, w_2 have same number of 1s and 0s and have smaller length
  - Why must w_1 and w_2 exist?
    - String w must have a balanced substring (0 and 1 may be endpoints of w)
  - By the induction hypothesis, $S \overset{*}{\Rightarrow} w_1$ and $S \overset{*}{\Rightarrow} w_2$, so $S \overset{*}{\Rightarrow} w$

Practice

- Exercise 25.5

Union

- Consider these two languages and their CFGs:

$$\mathcal{L}_1 = \{0^{\bullet n} 1^{\bullet n} | n \geq 0\}$$
$$A \rightarrow \varepsilon | 0A1$$

$$\mathcal{L}_2 = \{1^{\bullet n} 0^{\bullet n} | n \geq 0\}$$
$$B \rightarrow \varepsilon | 1B0$$

- What is $\mathcal{L}_1 \cup \mathcal{L}_2$?
  - All strings of equal 0s and 1s, where either all 0s come first or all 1s come first
- What is the CFG?

$$S \rightarrow A|B$$
$$A \rightarrow \varepsilon | 0A1$$
$$B \rightarrow \varepsilon | 1B0$$

Concatenation

- Consider these two languages and their CFGs:

$$\mathcal{L}_1 = \{0^{\bullet n} 1^{\bullet n} | n \geq 0\}$$
$$A \rightarrow \varepsilon | 0A1$$

$$\mathcal{L}_2 = \{1^{\bullet n} 0^{\bullet n} | n \geq 0\}$$
$$B \rightarrow \varepsilon | 1B0$$

- What is $\mathcal{L}_1 \bullet \mathcal{L}_2$?
  - All strings of equal 0s and 1s, where the first part comes from $\mathcal{L}_1$ and the second part comes from $\mathcal{L}_2$ ☺
- What is the CFG?

$$S \rightarrow AB$$
$$A \rightarrow \varepsilon | 0A1$$
$$B \rightarrow \varepsilon | 1B0$$

Kleene-star

- Consider these two languages and their CFGs:

$$\mathcal{L}_1 = \{0^{\bullet n} 1^{\bullet n} | n \geq 0\}$$
$$A \to \varepsilon | 0A1$$

$$\mathcal{L}_2 = \{1^{\bullet n} 0^{\bullet n} | n \geq 0\}$$
$$B \to \varepsilon | 1B0$$

- What is $\mathcal{L}_1^*$?
  - All concatenations of strings in $\mathcal{L}_1$
- What is the CFG?

$$S \to \varepsilon | SA$$
$$A \to \varepsilon | 0A1$$

- **Example 25.2.** CFGs can implement DFAs, and so are strictly more powerful.

Parse Trees

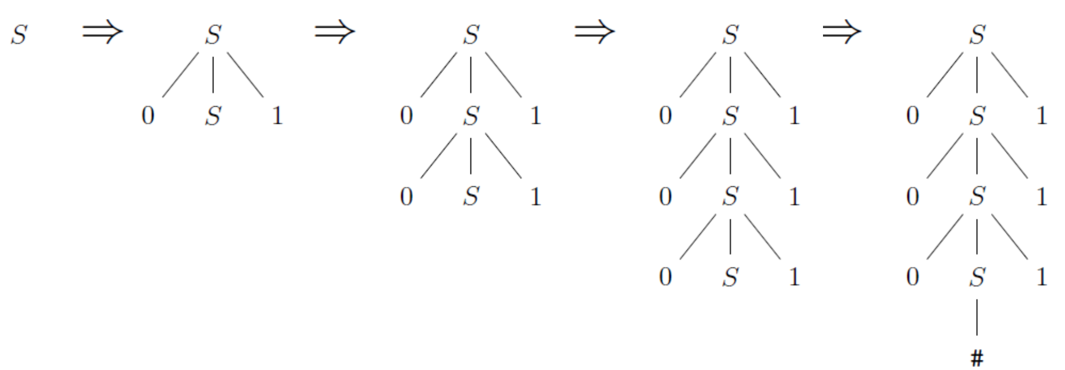
- Consider the CFG

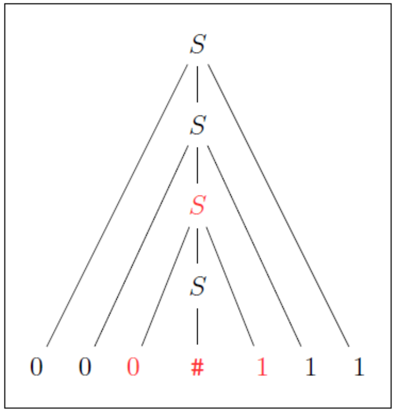
$$S \rightarrow \#|0S1$$

- What is the derivation of $000\#111$?

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000\#111$$

- The *parse tree* gives us more information than a derivation





- Clearly shows how a substring was derived from its parent variable.

CFG for Arithmetic

- Here is the CFG for arithmetic

$$S \rightarrow S + S | S \times S | (S) | 2$$

  - (the terminals are +, ×, (,) and 2)

- What are two derivations of $2 + 2 \times 2$

$$S \Rightarrow S + S \Rightarrow S + S \times S \stackrel{*}{\Rightarrow} 2 + 2 \times 2$$
$$S \Rightarrow S \times S \Rightarrow S + S \times S \stackrel{*}{\Rightarrow} 2 + 2 \times 2$$

- Parse tree ↔ How you interpret the string.
- Different parse trees ↔ different meanings.
- **BAD!** We want unambiguous meaning
  - programs, html-code, math, English, . . .

CFG for Arithmetic

- Here is the CFG for arithmetic
$$S \to S + S | S \times S | (S) | 2$$
  - (the terminals are $+$, $\times$, (,) and 2)
- What are two derivations of $2 + 2 \times 2$
$$S \Rightarrow S + S \Rightarrow S + S \times S \overset{*}{\Rightarrow} 2 + 2 \times 2$$
$$S \Rightarrow S \times S \Rightarrow S + S \times S \overset{*}{\Rightarrow} 2 + 2 \times 2$$
- How do we write an unambiguous version of the arithmetic CFG?
  - Use parenthesis and order of operations!
$$S \to P | S + P$$
$$P \to T | P \times T$$
$$T \to 2 | (S)$$

Pushdown Automata: DFAs with Stack Memory

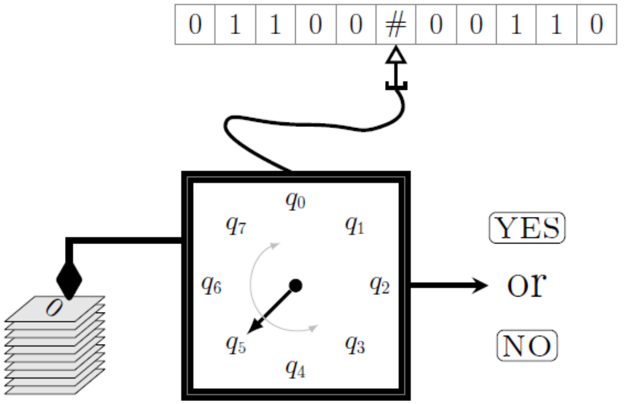
- Consider the palindrome-like language:

$$\mathcal{L} = \{ w\#w^R \mid w \in \{0,1\}^* \}$$

- What is the CFG?

$$S \rightarrow \# \mid 0S0 \mid 1S1$$

- Why can't a DFA decide whether words are in this language?
  - It needs to remember the first string w
- We can come with a DFA for deciding this language if we give it memory!
- DFA with stack memory (push, pop, read)
- Push the first half of the string (before #).
- For each bit in the second half, pop the stack and compare.



- DFAs with stack memory closely related to CFGs.

Non Context Free Expressions

- What languages do you think are hard to solve with a simple stack?
  - Repetition (a stack only counts in reverse)

$$\{w\#w\}$$

  - Multiple-equality (similar to repetition)

$$\{0^{\bullet n}1^{\bullet n}0^{\bullet n}\}$$

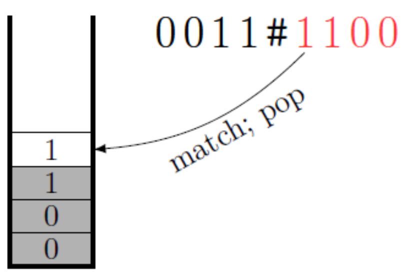
  - Squaring

$$\left\{0^{\bullet n^2}\right\}, \left\{0^{\bullet n}1^{\bullet n^2}\right\}$$

  - Exponentiation

$$\left\{0^{\bullet 2^n}\right\}, \left\{0^{\bullet n}1^{\bullet 2^n}\right\}$$

- Let's see why $w\#w^R$ is context-free:



0011 is pushed. DFA matches 1100 by popping

Non Context Free Expressions, cont'd

- What languages do you think are hard to solve with a simple stack?
  - Repetition (a stack only counts in reverse)

$$\{w\#w\}$$

  - Multiple-equality (similar to repetition)

$$\{0^{\bullet n}1^{\bullet n}0^{\bullet n}\}$$

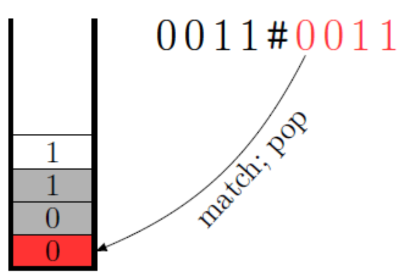
  - Squaring

$$\{0^{\bullet n^2}\}, \{0^{\bullet n}1^{\bullet n^2}\}$$

  - Exponentiation

$$\{0^{\bullet 2^n}\}, \{0^{\bullet n}1^{\bullet 2^n}\}$$

- Let's see why $w\#w$ is not context-free:



0011 is pushed. DFA needs bottom-access to match

Non Context Free Expressions, cont'd

- What languages do you think are hard to solve with a simple stack?
  - Repetition (a stack only counts in reverse)

$$\{w\#w\}$$

  - Multiple-equality (similar to repetition)

$$\{0^{\bullet n}1^{\bullet n}0^{\bullet n}\}$$

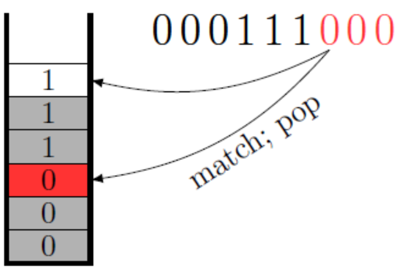
  - Squaring

$$\left\{0^{\bullet n^2}\right\}, \left\{0^{\bullet n}1^{\bullet n^2}\right\}$$

  - Exponentiation

$$\left\{0^{\bullet 2^n}\right\}, \left\{0^{\bullet n}1^{\bullet 2^n}\right\}$$

- Let's see why $0^{\bullet n}1^{\bullet n}0^{\bullet n}$ is not context-free:



000111 is pushed. DFA needs random access to match

Non Context Free Expressions, cont'd

Rensselaer

- The file clerk who only has access to the top of the *stack* of papers has fundamentally less power than the file clerk who has a *filing cabinet* with access to all papers.

- **We need a new model,** one with Random Access Memory (RAM).
  – Random doesn't mean probabilistic. Means the machine can access any part of the memory