CISC 5200: Computer Language Theory

Chapter 2
Context-Free Languages

Arthur G. Werschulz

Fordham University
Department of Computer and Information Sciences

Spring, 2020

Overview

- In Chapter 1, we introduced two equivalent methods for describing a language: finite automata and regular expressions.
- In this chapter, we do something analogous.
  - We introduce context-free grammars (CFGs)
  - We introduce pushdown automata (PDAs)
  - PDAs recognize CFGs
  - We have another Pumping Lemma

Why Context-Free Grammars?

- First used to study human languages
  You may have even seen something like them before.
- They are definitely used for many typical computer languages (C, C++, . . .).
  - They define the language.
  - A *parser* uses the grammar to parse the input.
    Unix provides:
    - `lex`, `flex`: lexical analyzer generator
    - `yacc`, `bison`: parser generator
  - Of course, you can also parse English.

Section 2.1: Context-Free Grammars

A context-free grammar

- Here is G_1, an example of a CFG:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

- A grammar has *substitution rules* or *productions*:
  - Each rule has a variable, an arrow, and a combination of variables and *terminal symbols*.
  - We capitalize variables, but not terminal symbols.
  - The *start symbol* is a special variable: usually on left-hand side of topmost rule.
- Here: variables are A and B, terminals are 0, 1, #.

Using the grammar

Use the grammar to generate a language by replacing variables using the rules in the grammar.

- Start with the start variable.
- Give me some strings that G_1 generates?
  - One answer: 000#111.
  - Sequence of steps: the *derivation*.
  - For this example, the derivation is

$$A \to 0A1 \to 00A11 \to 000A111 \to 000\#111.$$

- Can also represent with a parse tree.

The language of grammar G_1

- $L(G_1)$ is the *language* of all strings generated by G_1.
- $L(G_1) = \{0^n\#1^n : n \geq 0\}$.
- This should look familiar. Can we generate this with an FA?

An example: simplified English grammar G_2

⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⟨NOUN-PHRASE⟩ → ⟨CMPLX-NOUN⟩ | ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩
⟨VERB-PHRASE⟩ → ⟨CMPLX-VERB⟩ | ⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩
⟨PREP-PHRASE⟩ → ⟨PREP⟩⟨CMPLX-NOUN⟩
⟨CMPLX-NOUN⟩ → ⟨ARTICLE⟩⟨NOUN⟩
⟨CMPLX-VERB⟩ → ⟨VERB⟩ | ⟨VERB⟩⟨NOUN-PHRASE⟩
⟨ARTICLE⟩ → `a` | `the`
⟨NOUN⟩ → `boy` | `girl` | `flower`
⟨VERB⟩ → `touches` | `likes` | `sees`
⟨PREP⟩ → `with`

Derivation for "`a boy sees`"?

Formal definition of a CFG

A *context-free grammar* (CFG) is a 4-tuple (V, Σ, R, S), where

- V is a finite set of *variables*,
- Σ is a finite set, disjoint from V, of *terminals*,
- R is a finite set of *rules*, with each rule $v \to s$ consisting of a variable $v \in V$ and a string $s \in (V \cup \Sigma)^*$ of variables and terminals, and
- $S \in V$ is the *start variable*.

Example

Grammar $G_3 = (\{S\}, \{\mathrm{a}, \mathrm{b}\}, R, S)$, where the set R consists of only one rule, namely,

$$S \to \mathrm{a}S\mathrm{b} \,|\, SS \,|\, \varepsilon.$$

What does this generate?

- abab, aaabbb, aababb, ...
- If you view a as (and b as), then you get all strings of properly nested parentheses.
  - Note that ()() is permissible.
  - Key property? You have as many a's to the left of any given point in the string as you do b's.

Another example

- Grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$, where $V = \{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ and $\Sigma = \{\mathrm{a}, +, \times, (,)\}$.
- Productions:

$$\begin{aligned} \langle \text{EXPR} \rangle &\to \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \,|\, \langle \text{TERM} \rangle \\ \langle \text{TERM} \rangle &\to \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \,|\, \langle \text{FACTOR} \rangle \\ \langle \text{FACTOR} \rangle &\to (\langle \text{EXPR} \rangle) \,|\, \mathrm{a} \end{aligned}$$

- Let's do parse trees for $\mathrm{a} + \mathrm{a} \times \mathrm{a}$ and $(\mathrm{a} + \mathrm{a}) \times \mathrm{a}$

Designing CFGs

- Like designing FA, some creativity is required.
- CFGs perhaps harder than FAs: programming grammars vs. programming specific tasks.
- Here are some guidelines:
  - If the CFL is the union of simpler CFLs, design grammars for the simpler ones and then combine. For example, $S \to G_1 \,|\, G_2 \,|\, G_3$.
  - If the language is regular, then can design a CFG that mimics a DFA:
    - Make a variable R_i for every state q_i.
    - If $\delta(q_i, \mathrm{a}) = q_j$, then add rule $R_i \to \mathrm{a}R_j$.
    - Add $R_i \to \varepsilon$ if q_i is an accepting state.
    - Make R_0 the start variable, where q_0 is the start state of the DFA.
  - Assuming that this really works, what did we just show? *The regular languages are a strict subset of the context-free languages.*

Designing CFGs (continued)

A final guideline:

- Certain CFLs contain strings that are linked, in the sense that a machine for recognizing this language would need to remember an unbounded amount of information about one substring to "verify" the other substring.
  - This is sometimes trivial with a CFG.
  - Example: The language $0^n 1^n$. Grammar is:

$$S \to 0S1 \mid \varepsilon.$$

- Recursive structures can be handled by recursive rules.

Ambiguity

- Sometimes a grammar can generate the same string in multiple ways.
- If a grammar generates *even a single string* in multiple ways, the grammar is *ambiguous*.
- Example:

$$\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid \mathtt{a}.$$

  - This generates the string $a + a \times a$ ambiguously.
  - Try it: generate two parse trees.
  - Using your extensive knowledge of arithmetic, insert parentheses to show what each parse tree *really* expresses.

An English example from grammar G_2

- Grammar G_2 ambiguously generates "`the girl touches the boy with the flower`".
- Given your extensive knowledge of English, what are the two meanings of this phrase?

Definition of ambiguity

- A grammar generates a string ambiguously if there are two different parse trees for said string.
- Two derivations may differ in the order that the rules are applied, but if they generate the same parse tree, it is not really ambiguous.
- Definitions:
  - A derivation is a *leftmost derivation* if at every step, the leftmost remaining string is replaced.
  - A string w is derived *ambiguously* in a CFG if it has two or more different leftmost derivations.

Chomsky Normal Form

- It is often convenient to convert a CFG into a simplified form.
- A CFG is in *Chomsky normal form* if every rule is of the form
  - $A \to BC$ or
  - $A \to a$,

  where a is any terminal and A, B, and C are variables, except that neither B nor C can be the start variable. The start variable can also go to ε, i.e., we permit $S \to \varepsilon$.
- Any CFL can be generated by a CFG in Chomsky normal form.

Converting CFG to Chomsky Normal Form

- Add rule $S_0 \to S$, where S was original start variable.
- Remove ε-rules whose LHS is not the start variable: Remove $A \to \varepsilon$, and for each occurrence of such an A on RHS, add a new rule with that A deleted.
  Example: If $A \to \varepsilon$ and $R \to uAvAw$ are in the grammar, replace the latter by

$$\begin{aligned} R &\to uvAw \\ R &\to uAvw. \\ R &\to uvw \end{aligned}$$

- Handle all unit rules.
  Example: If we had $A \to B$, then whenever a rule $B \to u$ exists, we add $A \to u$.

Converting CFG to Chomsky Normal Form (cont'd)

- Replace rules $A \to u_1 u_2 \ldots u_k$ with

$$\begin{aligned} A &\to u_1 A_1 \\ A_1 &\to u_2 A_2 \\ A_2 &\to u_3 A_3 \\ &\vdots \\ A_{k-2} &\to u_{k-1} u_k \end{aligned} \quad .$$

- You *will* have a homework question like this.

Example: Convert CFG to Chomsky Normal Form)

Grammar G_1 from above:

$$\begin{aligned} A &\to 0A1 \mid B \\ B &\to \# \end{aligned}$$

- No ε-rules.
- Remove unit rule $A \to B$, getting

$$\begin{aligned} A &\to 0A1 \mid \# \\ B &\to \# \qquad \text{(this is now unneeded)} \end{aligned}$$

- Replace $A \to 0A1$ with

$$\begin{aligned} A &\to U_1 U_2 \\ U_1 &\to 0 \\ U_2 &\to AU_3 \\ U_3 &\to 1 \end{aligned}$$

Example: Convert CFG to Chomsky Normal Form (cont'd)

So the Chomsky Normal Form of

$$A \to 0A1 \mid B$$
$$B \to \#$$

is

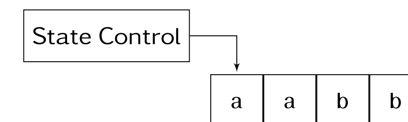
$$A \to U_1 U_2 \mid \#$$
$$U_1 \to 0$$
$$U_2 \to AU_3$$
$$U_3 \to 1$$

Section 2.2: Pushdown Automata

Pushdown Automata (PDA)

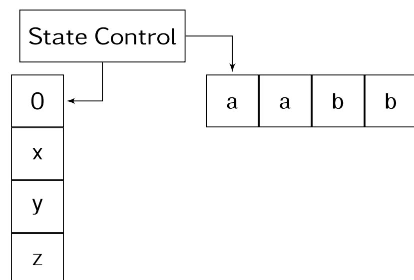
- Similar to FA, but have an extra component, called a *stack*. Stack provides extra (unbounded) memory that is separate from the control.
- Allows PDA to recognize non-regular languages.
- Equivalent in power/expressiveness to a CFG.
- Some languages easily described *generators*; others by *recognizers*.

Schematic of an FA



- The *state control* represents the states and transition function.
- *Tape* contains the input string.
- *Arrow* represents the input head and points to the next symbol to be read.

Schematic of a PDA



- The PDA adds a *stack*:
  - LIFO (Last In First Out) data structure, with *push* and *pop* operations.
  - Stack is *unbounded*.
- We can push (or pop) symbols onto (or from) the stack.

PDA and the language $0^n 1^n$

- Can a PDA recognize $0^n 1^n$?
- Yes, because stack is unbounded.
- Let's describe PDA that recognizes $0^n 1^n$:
  - **while** next input symbol is 0, push it onto the stack.
  - **while** next input symbol is 1 and stack is nonempty, pop the stack.
  - **if** no more input symbols and stack is nonempty **then** accept **else** reject.
- We reject if either of the following holds:
  - Stack is empty and input string is nonempty.
  - Stack is nonempty and input string is empty.

Formal definition of a PDA: warmup

The formal definition of a PDA is similar to that of an FA, but we now have a stack.

- Stack alphabet may be different than input alphabet; denote stack alphabet by Γ.
- Transition function is key part of definition:
  - Domain of transition function is now $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$. (Recall that $X_\varepsilon = X \cup \{\varepsilon\}$ for any set X.)
  - Current state, next input symbol, and top stack symbol determine next move.

Formal definition of PDA

A *pushdown automaton* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where:

- Q is a finite set of *states*,
- Σ is a finite *input alphabet*,
- Γ is a finite *stack alphabet*,
- $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathscr{P}(Q \times \Gamma_\varepsilon)$ is a *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accepting states*.

Note that PDA is nondeterministic!
However, we will try to use deterministic PDA wherever possible.

How does a PDA compute?

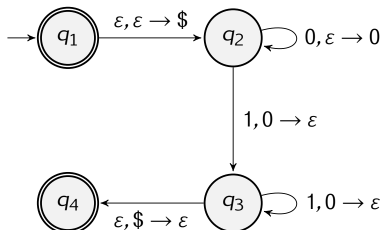
- The following three conditions must be satisfied for a string to be accepted by a PDA M:
  - M must start in the start state with an empty stack.
  - M must move according to the transition function.
  - At the end of the input, M must be in an accepting state.
- To simplify checking for an empty stack, a \$ is initially pushed onto the stack.
  If you ever see a \$ at the top of the stack, you know it is empty.
  Programmer's note: If your language provides a stack implementation, use it!

Notation

- We write $a, b \to c$ on the state diagram to mean
  if the current input symbol is a and stack top is b
  pop the stack and push c onto the stack.
- Any of a, b, or c can be ε.
  - If $a = \varepsilon$, then make stack change without reading an input symbol.
  - If $b = \varepsilon$, then no need to pop a symbol (just push c).
  - If $c = \varepsilon$, then no new symbol is written (just pop b).

Example 1: A PDA for $0^n 1^n$

- Let's formally describe PDA M_1 that accepts $\{0^n 1^n : n \geq 0\}$.
- $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, F)$, where
  - $Q = \{q_1, q_2, q_3, q_4\}$,
  - $\Sigma = \{0, 1\}$,
  - $\Gamma = \{0, \$\}$,
  - $F = \{q_1, q_4\}$, and
  - $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathscr{P}(Q \times \Gamma_\varepsilon)$ is described by the following state diagram:

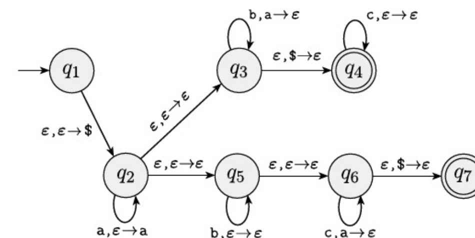


Example 2: A PDA for $a^i b^j c^k$, where $i = j$ or $i = k$

- Let's come up with a PDA M_2 that recognizes

$$\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i = j \text{ or } i = k)\}.$$

  - Come up with an informal description, as we did initially for $0^n 1^n$.
  - Can you do this without using non-determinism? No!
  - With non-determinism?
    Easy: Similar to $0^n 1^n$, except that we guess whether to match a's with b's or a's with c's.



Example 3: PDA for $ww^{\mathcal{R}}$

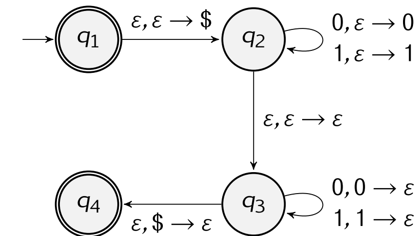
Let's come up with a PDA for the language

$$\left\{ww^{\mathcal{R}} : w \in \{0,1\}^*\right\}.$$

(Since $w^{\mathcal{R}}$ is the reversal of w, this is the language of palindromes over $\{0, 1\}$.)
Can you informally describe the PDA?

- As a deterministic PDA? No!
- As a nondeterministic PDA?
  Push symbols that are read onto the stack. At some point, nondeterministically guess that you are in the middle of the string, and then pop off stack values as they match the input.
  - If no match, then reject.
  - If you use up all the stack symbols and remaining input symbols, then accept.

Diagram of PDA for $ww^{\mathcal{R}}$



Equivalence with CFGs

- **Theorem:** A language is context-free iff some PDA recognizes it.
- **Lemma:** If a language L is context-free, then some PDA recognizes it.
  (We won't do other direction.)
- **Proof sketch:** Let $L = L(G)$ for some CFG G. We show how to convert it into a PDA P.
  - Thus P accepts a string s iff $s \in L(G)$.
  - Each main step of PDA involves application of one rule in G.
  - Stack contains intermediate strings generated by G.
  - Since G has a choice of rules to apply, P will be nondeterministic.
  - One issue: since P can only access the top of the stack, any terminal symbols pushed onto the stack must be immediately checked against the input string.
    - If the terminal symbol matches the next input character, then advance input string.
    - If the terminal symbol does not match the next input character, then terminate that path.

Informal description of P

- Place marker symbol \$ and the start symbol onto the stack.
- Repeat forever:
  - If the top of the stack is a nonterminal A, nondeterministically select one of the rules for A and substitute A by the string on the RHS of the rule.
  - If the top of the stack is a terminal symbol a, read the next symbol from the input and compare it to a.
    - If they match, repeat.
    - If they don't match, reject this branch.
  - If the top of the stack is the symbol \$, enter the accept state.
    Doing so accepts the input if it has all been read.

Example

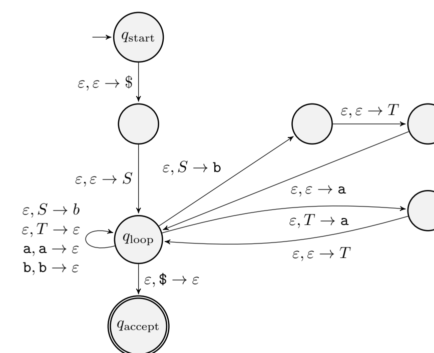
- Look at Example 2.25 on page 120 for an example of constructing a PDA P_1 from the CFG G with rules

$$S \to aTb \mid b$$
$$T \to Ta \mid \varepsilon$$

- Note that the top path that branches to the right will replace S by aTb.
  It first pushes b, then T, then a.
- Note that the path below that replaces T with Ta.
  It replaces T with a, then pops T on top of that.

Example (cont'd)

$$S \to aTb \mid b$$
$$T \to Ta \mid \varepsilon$$



Example (cont'd)

Our task: Show how this PDA accepts the string aab, which has the derivation

$$S \to aTb \to aTab \to aab.$$

In the following, the left is the top of the stack.

- We start with $S\$$.
- We take the top branch to the right, and we get the following as we go through each state:

$$S\$ \to b\$ \to Tb\$ \to aTb\$$$

- We read a and use rule $a, a \to \varepsilon$ to pop stack, getting $Tb\$$.
- We next take second branch going to right:

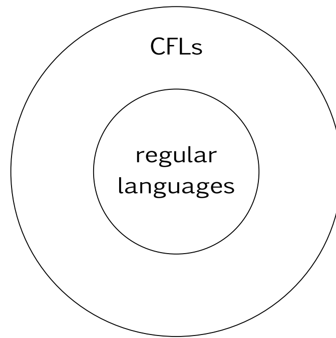
$$Tb\$ \to ab\$ \to Tab\$$$

- We next use rule $\varepsilon, T \to \varepsilon$ to pop T, getting ab\$.
- We next pop a, then pop b, at which point we have \$.
- Everything read, so accept.

Relationship between regular languages and CFLs

- We know that CFGs define CFLs.
- We now know that a PDA recognizes the same class of languages, and hence recognizes CFLs.
- We know that every FA is a stackless PDA (i.e., a PDA that doesn't use its stack).
- Thus PDAs recognize regular languages.
- Thus every regular language is a CFL.
- But since we know that no FA can recognize the CFL $0^n 1^n$, we can say even more:
- CFLs and regular languages are not equivalent.

Relationship between regular languages and CFLs



Section 2.3: Non-Context-Free Languages

Non-context-free languages

- Just as there are non-regular languages, there are also languages that are not context-free.
  - Hence they cannot be generated by a CFG.
  - Hence they cannot be recognized by a PDA.
- Just your luck! There is also a Pumping Lemma that can be used to show that a language is not context-free!

Pumping Lemma for CFGs

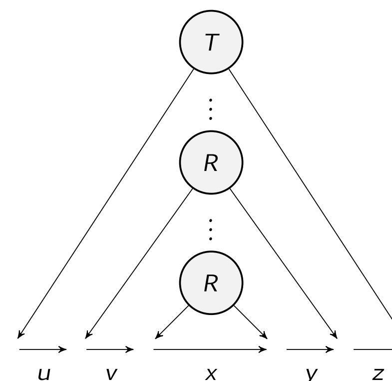
If A is a CFL, then there is a pumping length p such that any $s \in A$ with $|s| \geq p$ can be written in the form $s = uvxyz$, where:

- $uv^i xy^i z \in A$ for each $i \geq 0$.
- $|vy| > 0$.
- $|vxy| \leq p$.

Proof Idea

- For regular languages, we applied the pigeonhole principle to the number of *states*, to show that a *state* had to be repeated.
- Here, we apply the pigeonhole principle to the number of *variables* in the CFG, to show that some *variable* will need to be repeated within some path found in the parse tree of a sufficiently-long string.
- Suppose:
  - R is one such variable.
  - (The lowest appearance of) R (on this path) derives a string x.
- Then ...

Proof Idea (cont'd)



Have derivations $T \to uRz$, $R \to x$, $R \to vRy$.
Since we can keep applying rule $R \to vRy$, we can derive $uv^i xy^i z$ for any $i \geq 0$.

Example 1

- Let $B = \{a^n b^n c^n : n \geq 0\}$. Use the Pumping Lemma to show that B is not context-free (Ex. 2.36, pg. 127).
- Let p be the pumping length. Select the string $s = a^p b^p c^p$. Then $s \in B$ and $|s| > p$.
- Condition 2 says that v and y cannot both be empty.
- Consider two cases:
  - Both v and y contain only one kind of input symbol. Then uv^2xy^2z can't have equal number of a's, b's, c's.
  - Either v or y contains more than one kind of symbol. Pumping then violates separation of a's, b's, c's.
- In either case, conclusion of Pumping Lemma is false, and so B is not context-free.
- Alternative proof:
  - Since $|vxy| \leq p$, it follows that v and y contain at most two symbols. Hence at least one is left out when we pump up.
  - More precisely, at least one symbol is in v or y, and so pumping breaks the equality.

Example 2

- Let $D = \{ww : w \in \{0,1\}^*\}$. Use the Pumping Lemma to show that D is not context-free (Ex. 2.38, pg. 127).
- Let's choose $s = 0^p 10^p 1$.
  - This *can* be pumped!
  - Take $u = 0^{p-1}$, $v = 0$, $x = 1$, $y = 0$, and $z = 0^{p-1}1$.
  - We find $uv^2xy^2z \in D$, $uxz \in D$, ...
- Choose $s = 0^p 1^p 0^p 1^p$:
  - First, note that vxy must straddle midpoint. (Otherwise, pumping makes the two halves unequal.)
  - Since vxy must straddle midpoint, pumping down doesn't work: uxz is not of the form ww (neither 0's nor 1's match in each half).
- So D is not context-free.