CS622

Context-Free Languages (CFLs)

Wednesday, September 29, 2021
Announcements

• First **in-person class**: next Monday 10/4 7pm
  • McCormack M01-0209

• HW3 due Sun 11:59pm EST

• HW2 grades released
HW2 Review

2 Exending the definition of "REACHABLE"

Define $\varepsilon$-REACHABLE$_{qs}$, which is like the $\varepsilon$-REACHABLE definition from class, but extended to sets of states. (Don’t forget to handle the empty set!)

$$\varepsilon\text{-REACHABLE}_{qs}(qs) = \bigcup_{q \in qs} \varepsilon\text{-REACHABLE}(q)$$
3. DFA→NFA

In class we showed how to convert an NFA into an equivalent DFA, but not a DFA to NFA. Do this now.

More specifically:

- Come up with a procedure **DFA→NFA** that converts DFAs to equivalent NFAs. In other words, given some DFA \( M = (Q, \Sigma, \delta, q_0, F) \) that satisfies the formal definition of DFAs from class, **DFA→NFA** should produce some NFA \( N = (Q', \Sigma, \delta', q'_0, F') \) that satisfies the formal definition of NFAs and accepts the same language as \( M \).

**A finite automaton** is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where

1. \( Q \) is a finite set called the **states**,
2. \( \Sigma \) is a finite set called the **alphabet**,
3. \( \delta: Q \times \Sigma \rightarrow Q \) is the **transition function**,
4. \( q_0 \in Q \) is the **start state**, and
5. \( F \subseteq Q \) is the **set of accept states**.

**A nondeterministic finite automaton** is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where

1. \( Q \) is a finite set of states,
2. \( \Sigma \) is a finite alphabet,
3. \( \delta: Q \times \Sigma \rightarrow \mathcal{P}(Q) \) is the transition function,
4. \( q_0 \in Q \) is the **start state**, and
5. \( F \subseteq Q \) is the **set of accept states**.
HW2 Review

⇒ If $M$ accepts $w$, then $N$ accepts $w$
  • If $M$ accepts $w$, then $\hat{\delta}_M(q_0, w) \in F$  
  • So $N$ accepts $w$ because $\hat{\delta}_N(q_0, w) = \{\hat{\delta}_M(q_0, w)\}$  
  thus $\hat{\delta}_N(q_0, w) \cap F_N \neq \emptyset$
⇐ If $N$ accepts $w$, the $M$ accepts $w$
  • (similar)

First assume: $\hat{\delta}_N(q_0, w) = \{\hat{\delta}_M(q_0, w)\}$
  • NOTE: This must match part 1’s answer!
  • Some invalid equalities:
    $\hat{\delta}_N(q_0, w) \neq \hat{\delta}_M(q_0, w)$
    $\hat{\delta}_N(q_0, w) \neq \hat{\delta}_M(\{q_0\}, w)$

Criteria for acceptance for DFAs / NFAs
So correctness proof must also have these parts
This says nothing about acceptance!
HW2 Review

Now prove: \( \hat{\delta}_N(q_0, w) = \{\hat{\delta}_M(q_0, w)\} \)

Proof: Using proof by induction on the length of string \( w \)

- **Base case:** We always start from the smallest string i.e., \( w = \varepsilon \)
  
  Applying this on the theorem, \( \hat{\delta}_N(q_0, \varepsilon) \) and \( \{\hat{\delta}_M(q_0, \varepsilon)\} \) we get \( \{q_0\} \) for both the cases.

- **Inductive case:** For this we will take \( w = xa \)
  
  - **Inductive hypothesis:** \( \hat{\delta}_N(q_0, x) = \{\hat{\delta}_M(q_0, x)\} \), call this set of states \( R \)
  
  - DFA last step from \( \delta_M \) definition is given as \( \{\delta_M(r, a)\} \)
  
  - NFA last step from DFA \( \rightarrow \) NFA definition is given as \( \{\delta_M(r, a)\} \)

Here, \( r \in R \) and \( a \) is the last alphabet of the string \( w \).

From definition of \( \hat{\delta} \) (base case)

From definition of \( \hat{\delta} \) (inductive case)

From our NFA \( \rightarrow \) DFA conversion
5. A Closure Operation

Let $\text{EXPAND}_c$ on a language $L$, where $\Sigma$ is the alphabet of $L$ and $c \in \Sigma$, be:

$$\text{EXPAND}_c(L) = \{wc \mid w \in L\}$$

Prove that, for any $c$, $\text{EXPAND}_c$ is closed for regular languages.

To prove that for any $c$, $\text{EXPAND}_c$ is closed for regular languages, we need to create a DFA/NFA that recognizing it.

1. Let $L = (Q_1, \Sigma, \delta_1, q_1, F_1)$, we construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $\text{EXPAND}_c$.
2. The state $q_0$ is the same as the start state of $L$.
3. The accept state $F$ will be the new state $\{q_c\}$.
4. Define $\delta$ so that for any $q \in Q$, and any $a \in \Sigma$,
   
   $$\delta(q, a) = \begin{cases} 
   \delta_1(q, a) & q \in Q_1 \text{ and } a \notin F \\
   \delta_1(q, a) & q \in F_1 \text{ and } a \neq c \\
   \{q_c\} & q \in F_1 \text{ and } a = c 
   \end{cases}$$

$L$ is regular so it must have an NFA recognizing it (thm from class)

Extend $L$'s NFA to recognize $\text{EXPAND}_c(L)$

$\text{EXPAND}_c(L)$ must be regular if it has an NFA recognizing it (thm from class)

Therefore $\text{EXPAND}_c$ is closed for regular languages
Last Time:

The Pumping lemma states that if $A$ is a regular language, then there exists a number $p$ (the pumping length) such that for every string $s$ in $A$ with $|s| \geq p$, $s$ can be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. For each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Let $B$ be the language $\{0^n1^n \mid n \geq 0\}$. We use the pumping lemma to prove that $B$ is not regular. The proof is by contradiction.

If this language is not regular, then what is it???

Maybe? ... a context-free language (CFL)?
A Context-Free Grammar (CFG)

Top variable is **Start variable**

**Variables** (also called nonterminals)

**terminals**

**A** → **0A1**

**A** → **B**

**B** → **#**

**Substitution rules** (a.k.a., productions)

**terminals** (analogous to a DFA’s alphabet)
A context-free grammar is a 4-tuple $(V, \Sigma, R, S)$, where

1. $V$ is a finite set called the **variables**, 
2. $\Sigma$ is a finite set, disjoint from $V$, called the **terminals**, 
3. $R$ is a finite set of **rules**, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

**Examples:**

- $V = \{A, B\}$,
- $\Sigma = \{0, 1, \#\}$,
- $S = A$,
## Analogies

<table>
<thead>
<tr>
<th>Regular Language</th>
<th>Context-Free Language (CFL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expression</td>
<td>Context-Free Grammar (CFG)</td>
</tr>
<tr>
<td>A Reg expr <strong>describes</strong> a Regular lang</td>
<td>A CFG <strong>describes</strong> a CFL</td>
</tr>
</tbody>
</table>

*Practical application: Used to describe programming languages!*
Java Language Described with CFGs

Chapter 2. Grammars

This chapter describes the context-free grammars used in this specification to define the lexical and syntactic structure of a program.

2.1. Context-Free Grammars

A context-free grammar consists of a number of productions. Each production has an abstract symbol called a nonterminal as its left hand side, and a sequence of one or more nonterminal and terminal symbols as its right-hand side. For each grammar, the terminal symbols are drawn from a specified alphabet.

Starting from a sentence consisting of a single distinguished nonterminal, called the goal symbol, a given context-free grammar specifies a language, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

2.2. The Lexical Grammar

A lexical grammar for the Java programming language is given in §3. This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol input (§3.5), that describe how sequences of Unicode characters (§3.1) are translated into a sequence of input tokens (§3.6).

https://docs.oracle.com/javase/specs/jls/se7/html/jls-2.html
10. Full Grammar specification

This is the full Python grammar, as it is read by the parser generator and used to parse Python source files:

```
# Grammar for Python
# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
# single_input is a single interactive statement;
# file_input is a module or sequence of commands read from an input file;
# eval_input is the input for the eval() functions.
# func_type_input is a PEP 484 Python 2 function type comment
# NB: compound_stmt in single_input is followed by extra NEWLINE!
# NB: due to the way TYPE_COMMENT is tokenized it will always be followed by a NEWLINE
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER
```

https://docs.python.org/3/reference/grammar.html
10. Full Grammar specification

This is the full Python grammar, as it is read by the parser generator and used to parse Python source files:

```python
# Grammar for Python

# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
#    single_input is a single interactive statement;
#    file_input is a module or sequence of commands read from an input file;
#    eval_input is the input for the eval() functions.
#    func_type_input is a PEP 484 Python 2 function type comment
# NB: compound_stmt in single_input is followed by extra NEWLINE!
# NB: due to the way TYPE_COMMENT is tokenized it will always be followed by a NEWLINE
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER
```

https://docs.python.org/3/reference/grammar.html
Generating Strings with a CFG

A CFG \( G_1 \) represents a language:

\[
G_1 =
\begin{align*}
A & \rightarrow 0A1 \\
A & \rightarrow B \\
B & \rightarrow \# \\
\end{align*}
\]

Strings in CFG’s language = all possible generated strings

\( L(G_1) \) is \( \{0^n#1^n \mid n \geq 0\} \)

A CFG \textit{generates} a string, by repeatedly applying substitution rules:

\[
A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000#111
\]

Stop when string is all terminals

Start variable  
After applying 1st rule  
Use 1st rule  
Use 1st rule  
Use 2nd rule  
Use last rule
Derivations: Formally

Let $G = (V, \Sigma, R, S)$

Single-step

$$\alpha A \beta \Rightarrow_G \alpha \gamma \beta$$

Where:

- $\alpha, \beta \in (V \cup \Sigma)^*$
- $A \in V$ (Variable)
- $A \rightarrow \gamma \in R$ (Rule)

Extended Derivation

Base case: $\alpha \Rightarrow_G^* \alpha$

Recursive case:

- If $\alpha \Rightarrow_G^* \beta$ and $\beta \Rightarrow_G \gamma$
- Then: $\alpha \Rightarrow_G^* \gamma$
Formal Definition of a CFL

A context-free grammar is a 4-tuple $(V, \Sigma, R, S)$, where
1. $V$ is a finite set called the variables,
2. $\Sigma$ is a finite set, disjoint from $V$, called the terminals,
3. $R$ is a finite set of rules, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

$$G = (V, \Sigma, R, S)$$

$$L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^*_G w \}$$

Any language that can be generated by some context-free grammar is called a context-free language
Flashback: \( \{0^n1^n \mid n \geq 0\} \)

- Pumping Lemma says it’s not a regular language
- It’s a context-free language!
  - Proof?
  - Come up with CFG describing it ...
  - It’s similar to:

\[
A \rightarrow 0A1 \\
A \rightarrow B \\
B \rightarrow \# \varepsilon
\]

\( L(G_1) \) is \( \{0^n\#1^n \mid n \geq 0\} \)
Proof of Correctness

Correctness statement: \( w \in L \) if and only if \( A \stackrel{*}{\rightarrow}_G w \)

\[ L = \{0^n1^n \mid n \geq 0\} \]

rules of \( G \):
\[ A \rightarrow 0A1 \mid \varepsilon \]

Base case \( w = \varepsilon \): if \( \varepsilon \in L \) then \( A \stackrel{*}{\rightarrow}_G \varepsilon \)
true, due to rule \( A \rightarrow \varepsilon \)

\[ \iff \text{if } A \stackrel{*}{\rightarrow}_G w \text{ then } w \in L \]
Proof of Correctness

Correctness statement: \( w \in L \) if and only if \( A \xrightarrow{G}^* w \)

\[ L = \{0^n1^n \mid n \geq 0\} \]

Rules of \( G \):
\[ A \rightarrow 0A1 \mid \varepsilon \]

\[ L \subseteq \{0^n1^n \mid n \geq 0\} \]

\[ L \cap \{0^n1^n \mid n \geq 0\} = \{0^n1^n \mid n \geq 0\} \]

Note the parts of the proof:
- Clear and precise correctness statement
- All cases covered (⇒ and ⇐, base and inductive cases)
- Every step logically follows from previous
- Every step has a justification
- Uses the given facts (IH, etc)

⇒ if \( w \in L \) then \( A \xrightarrow{G}^* w \)

Base case \( w = \varepsilon \):
- if \( \varepsilon \in L \) then \( A \xrightarrow{G}^* \varepsilon \)
  - true, due to rule \( A \rightarrow \varepsilon \)

Inductive case \( w = 0x1 \)

IH: if \( x \in L \) then \( A \xrightarrow{G}^* x \)

Need to prove: if \( 0x1 \in L \) then \( A \xrightarrow{G}^* 0x1 \)

if \( 0x1 \in L \) then \( x \in L \) (def of \( L \)) and \( A \xrightarrow{G}^* x \) (by IH)

if \( A \xrightarrow{G}^* x \) then \( A \xrightarrow{G}^* 0x1 \), by def of \( \xrightarrow{G}^* \) and rule \( A \rightarrow 0A1 \)

Therefore: if \( 0x1 \in L \) then \( A \xrightarrow{G}^* 0x1 \)

⇐ if \( A \xrightarrow{G}^* w \) then \( w \in L \)

HW4
A String Can Have Multiple Derivations

\[
\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle
\]
\[
\langle \text{TERM} \rangle \rightarrow \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle
\]
\[
\langle \text{FACTOR} \rangle \rightarrow ( \langle \text{EXPR} \rangle ) \mid a
\]

String to generate: \( a + a \times a \)

- **EXPR** ⇒
- **EXPR** + **TERM** ⇒
- **EXPR** + **TERM** \( \times \) **FACTOR** ⇒
- **EXPR** + **TERM** \( \times \) \( a \) ⇒

**RIGHTMOST** DERIVATION

- **EXPR** ⇒
- **EXPR** + **TERM** ⇒
- **TERM** + **TERM** ⇒
- **FACTOR** + **TERM** ⇒
- \( a \) + **TERM**

**LEFTMOST** DERIVATION
Derivations and Parse Trees

\[ A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111 \]

A derivation may also be represented as a **parse tree**
Multiple Derivations, Single Parse Tree

- **Leftmost** derivation
  - \( \text{EXPR} \Rightarrow \)
  - \( \text{EXPR} + \text{TERM} \Rightarrow \)
  - \( \text{TERM} + \text{TERM} \Rightarrow \)
  - \( \text{FACTOR} + \text{TERM} \Rightarrow \)
  - \( a + \text{TERM} \)
  - ...

- **Rightmost** derivation
  - \( \text{EXPR} \Rightarrow \)
  - \( \text{EXPR} + \text{TERM} \Rightarrow \)
  - \( \text{EXPR} + \text{TERM} \times \text{FACTOR} \Rightarrow \)
  - \( \text{EXPR} + \text{TERM} \times a \Rightarrow \)
  - ...

Since the “meaning” (i.e., parse tree) is same, by convention we just use **leftmost** derivation.

A Parse Tree gives “meaning” to a string.
Ambiguity

grammar $G_5$:

\[ \langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid ( \langle \text{EXPR} \rangle ) \mid \text{a} \]

Same string, Different derivation, and different parse tree!
A string $w$ is derived *ambiguously* in context-free grammar $G$ if it has two or more different leftmost derivations. Grammar $G$ is *ambiguous* if it generates some string ambiguously.

An ambiguous grammar can give a string multiple meanings! (why is this bad?)
Real-life Ambiguity (“Dangling” else)

• What is the result of this C program?
  • if (1) if (0) printf("a"); else printf("2");

```
if (1)
  if (0)
    printf("a");
  else
    printf("2");
```

```
if (1)
  if (0)
    printf("a");
  else
    printf("2");
```

Ambiguous grammars are confusing. In a language, a string (program) should have only one meaning.

Problem is, there’s no guaranteed way to create an unambiguous grammar (so language designers must be careful)
Designing Grammars : Basics

• Think about what you want to “link” together

• E.g., XML
  • ELEMENT $\rightarrow$ <TAG>CONTENT</TAG>
  • Start and end tags are “linked”

• Start with small grammars and then combine (just like FSMs)
Designing Grammars: Building Up

• Start with small grammars and then combine (just like FSMs)
  
  • To create a grammar for the language \( \{0^n1^n|n \geq 0\} \cup \{1^n0^n|n \geq 0\} \)

• First create grammar for lang \( \{0^n1^n|n \geq 0\} \):
  \[
  S_1 \rightarrow 0S_11 | \varepsilon
  \]

• Then create grammar for lang \( \{1^n0^n|n \geq 0\} \):
  \[
  S_2 \rightarrow 1S_20 | \varepsilon
  \]

• Then combine:
  \[
  S \rightarrow S_1 | S_2
  S_1 \rightarrow 0S_11 | \varepsilon
  S_2 \rightarrow 1S_20 | \varepsilon
  \]

“\(|\)” = “or” = union (combines 2 rules with same left side)
Closed Operations on CFLs

• Start with small grammars and then combine (just like FSMs)

• “Or”: \[ S \rightarrow S_1 \mid S_2 \]

• “Concatenate”: \[ S \rightarrow S_1 S_2 \]

• “Repetition”: \[ S' \rightarrow S'S_1 \mid \varepsilon \]
In-class exercise: Designing grammars

alphabet $\Sigma$ is \{0,1\}

\{w \mid w \text{ starts and ends with the same symbol}\}

• $S \rightarrow 0C'0 \mid 1C'1 \mid \varepsilon$ “string starts/ends with same symbol, middle can be anything”
• $C' \rightarrow C'C \mid \varepsilon$ “all possible terminals, repeated (ie, all possible strings)”
• $C \rightarrow 0 \mid 1$ “all possible terminals”
Check-in Quiz 9/29

On gradescope