UMB CS 420

Context-Free Languages (CFLs)

Wednesday, February 16, 2022
Announcements

• HW3, Problem 3 (rev strings) has a new requirement!
  • See piazza post and hw3 page

• HW3 due Sun 2/20 11:59pm EST

• Reminder: No class next Monday 2/21
Last Time:

Pumping lemma: If $A$ is a regular language, then there is a number $p$ (the pumping length) where if $s$ is any string in $A$ of length at least $p$, then $s$ may 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 $\{0^n1^n | n \geq 0\}$. We use the pumping lemma to prove that $B$ is not regular. The proof is by contradiction.

- **Assume** the language is regular

- So it **must follow** the Pumping Lemma:
  - All strings longer than length $p$ ...
  - ... must be splitable into $xyz$ ... where $y$ is “pumpable”

- **Find counterexample** where Pumping Lemma does not hold: $0^p1^p$

- Therefore, the language is **not regular**
  - This is the contrapositive of the Pumping Lemma
  - And also a **contradiction** of the assumption!
**Pumping lemma** If $A$ is a regular language, then there is a number $p$ (the pumping length) where if $s$ is any string in $A$ of length at least $p$, then $s$ may 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 $\{0^n1^n | n \geq 0\}$. 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 (a.k.a., nonterminals)

A → 0A1
A → B
B → #

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

Terminals (analogous to a DFA's alphabet)
A context-free grammar (CFG)

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.

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

**CFG Practical Application:**
Used to describe **programming language syntax**!
Java Syntax: 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 elements (§3.6).
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
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:

```plaintext
# 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

\[ G_1 = \]
\[
A \rightarrow 0A1 \\
A \rightarrow B \\
B \rightarrow \#
\]

1st rule

A CFG represents a context free language!

Strings in CFG’s language = all possible generated strings

\[ L(G_1) = \{0^n#1^n \mid n \geq 0\} \]

A CFG generates a string, by repeatedly applying substitution rules:

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

Start variable
After applying 1st rule
1st rule again
1st rule again
Use 2nd rule
Use last rule

Stop when string is all terminals
Derivations: Formally

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

**Single-step**

\[ \alpha A \beta \xrightarrow{G} \alpha \gamma \beta \]

Where:

\( \alpha, \beta \in (V \cup \Sigma)^* \)

\( A \in V \)

\( A \rightarrow \gamma \in R \)

**Extended Derivation**

**Base case:**

\[ \alpha \Rightarrow^* \alpha \]

(0 steps)

**Recursive case:**

- If \( \alpha \Rightarrow G \beta \) and \( \beta \Rightarrow^* \gamma \)
- Then:

\[ \alpha \Rightarrow^* \gamma \]

(multistep)
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**.

\[ G = (V, \Sigma, R, S) \]

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

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 ...
  - **Hint:** It's similar to:

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

\(L(G_1)\) is \(\{0^n\#1^n \mid n \geq 0\}\)
A String Can Have Multiple Derivations

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

Want to generate this string: \( a + a \times a \)

- **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?

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

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

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

This string has 2 parsings, and thus 2 meanings!

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

Problem is, there’s no guaranteed way to create an unambiguous grammar (up to language designers to “be careful”)

Designing Grammars: Basics

1. Think about what you want to “link” together

   - E.g., $0^n 1^n$
     - $A \rightarrow 0A1$
     - # 0s and # 1s are “linked”

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

2. 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 \( \{0^n1^n | n \geq 0\} \):
    \[
    S_1 \rightarrow 0S_11 \mid \varepsilon
    \]
  
  - Then create grammar for lang \( \{1^n0^n | n \geq 0\} \):
    \[
    S_2 \rightarrow 1S_20 \mid \varepsilon
    \]
  
  - Then combine:
    \[
    S \rightarrow S_1 \mid S_2
    S_1 \rightarrow 0S_11 \mid \varepsilon
    S_2 \rightarrow 1S_20 \mid \varepsilon
    \]

New start variable & rule combines two smaller grammars

"|" = “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”:
  \[ S' \rightarrow S' S_1 \mid \epsilon \]
In-class Example: 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”
- $C' \rightarrow C'C \mid \varepsilon$  
  “middle: all possible terminals, repeated (ie, all possible strings)”
- $C \rightarrow 0 \mid 1$  
  “all possible terminals”
Next Time:

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context-Free Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expression</td>
<td>Context-Free Grammar (CFG)</td>
</tr>
<tr>
<td>A Reg expr describes a Regular lang</td>
<td>A CFG describes a CFL</td>
</tr>
<tr>
<td>Finite automaton (FSM)</td>
<td>???</td>
</tr>
<tr>
<td>An FSM recognizes a Regular lang</td>
<td>???</td>
</tr>
</tbody>
</table>
Next Time:

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context-Free Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expression</td>
<td>Context-Free Grammar (CFG)</td>
</tr>
<tr>
<td>A Reg expr describes a Regular lang</td>
<td>A CFG describes a CFL</td>
</tr>
<tr>
<td>Finite automaton (FSM)</td>
<td>Push-down automaton (PDA)</td>
</tr>
<tr>
<td>An FSM recognizes a Regular lang</td>
<td>A PDA recognizes a CFL</td>
</tr>
</tbody>
</table>


### Next Time:

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context-Free Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Regular Expression</strong></td>
<td>Context-Free Grammar (CFG)</td>
</tr>
<tr>
<td>A Reg expr describes a Regular lang</td>
<td>A CFG describes a CFL</td>
</tr>
<tr>
<td><strong>Finite automaton (FSM)</strong></td>
<td>Push-down automaton (PDA)</td>
</tr>
<tr>
<td>An FSM recognizes a Regular lang</td>
<td>A PDA recognizes a CFL</td>
</tr>
<tr>
<td><strong>DIFFERENCE:</strong></td>
<td><strong>DIFFERENCE:</strong></td>
</tr>
<tr>
<td>A Regular lang is defined with a FSM</td>
<td>A CFL is defined with a CFG</td>
</tr>
</tbody>
</table>

**Proved:** Reg expr $\Leftrightarrow$ Reg lang

**Must prove:** PDA $\Leftrightarrow$ CFL
Check-in Quiz 2/16

On gradescope