UMB CS 622

Context-Free Languages (CFLs)

Monday, March 18, 2024
Announcements

• HW 4 in
  • due Mon 3/18 12pm noon

• HW 5 out
  • due Mon 3/25 12pm noon
Non-Regular Languages

Example: 

\[ L = \{ \theta^n 1^n \mid n \geq 0 \} \]

- A DFA recognizing \( L \) would require infinite states! (impossible)
  - States representing: zero \( \theta \)s seen, one \( \theta \) seen, two \( \theta \)s, ...

- This language is the same as many PLs, e.g., HTML!
  - To better see this replace:
    - \( \theta \) with \( <\text{tag}>\) or \( “(\) “
    - \( 1 \) with \( </\text{tag}>\) or \( “)”
  
- The Problem: remembering nestedness
  - Need to count arbitrary nesting depths
    - E.g., \{ if \{ if \{ ... \} \} \}
  - Thus: most programming language syntax is not regular!

We can prove non-regularness ... with the Pumping Lemma (and proof by contradiction)

But ... what kind of language is it then?
A Context-Free Grammar (CFG)

Top variable is:

Start variable

Variables (a.k.a., non-terminals) on RHS of rules

terminals

$A \rightarrow 0A1$

$A \rightarrow B$

$B \rightarrow \#$

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

terminals (analogous to 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.
Java Syntax: Described with CFGs

Chapter 2. Grammars

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).

A CFG specifies a language!
# 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 describes a Regular lang</td>
<td>A CFG describes a CFL</td>
</tr>
</tbody>
</table>

*thm*

*def*
Python Syntax: Described with a CFG

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
Many Other Language (partially)

Python Syntax: Described with a CFG

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
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 (§2.5).

https://docs.oracle.com/javase/specs/jls/se7/html/jls-2.html
Generating Strings with a CFG

In-class exercise:
Write 3 more strings that can be generated by this grammar

1st rule
\[ A \rightarrow 0A1 \]

2nd rule
\[ A \rightarrow B \]

Last rule
\[ B \rightarrow \# \]

"Applying a rule" = replace LHS variable with RHS sequence
At each step, arbitrarily choose any variable to replace, and any rule to apply

Definition:
A CFG describes a context-free language but what strings are in the language?

Stop when: string is all terminals

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

Example:
\[ A \rightarrow 0A1 \rightarrow 00A11 \rightarrow 000A111 \rightarrow 000B111 \rightarrow 000\#111 \]
Generating Strings with a CFG

Definition:
A CFG describes a context-free language!
but what strings are in the language?

Strings in CFG’s language
= all possible generated / derived strings

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

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

Example:

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

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

**Single-step**

$$\alpha A \beta \xrightarrow{\alpha \gamma \beta} \quad (G)$$

Where:

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

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**.
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)

**Multi-step** (recursively defined)

**Base case:**

$\alpha \Rightarrow^*_G \alpha$ (0 steps)

**Recursive case:**

- If: $\alpha \Rightarrow G \beta$ and $\beta \Rightarrow^*_G \gamma$
  - Single step
- Then: $\alpha \Rightarrow G \gamma$ (smaller) Recursive “call”
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.

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

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

"all possible sequences of terminal symbols ..."

... "that can be generated with rules of grammar \(G\)"

"the language of a grammar \(G\) is ..."

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: not a regular language
- It's a context-free language!
  - Proof?
  - Key step: Come up with CFG describing it ...
  - Hint: It's similar to:

$$
\begin{align*}
A & \rightarrow 0A1 \\
A & \rightarrow B \\
B & \rightarrow \# \epsilon
\end{align*}
\quad L(G_1) \text{ is } \{0^n\#1^n \mid n \geq 0\}
$$

 Statements and Justifications?
Proof: $L = \{0^n1^n \mid n \geq 0\}$ is a CFL

<table>
<thead>
<tr>
<th>Statements</th>
<th>Justifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. If a CFG describes a language, then it is a CFL</td>
<td>1. Definition of CFL</td>
</tr>
<tr>
<td>2. CFG $G_1$ describes $L$</td>
<td>2. (Did you come up with examples???)</td>
</tr>
<tr>
<td>$A \rightarrow 0A1$</td>
<td></td>
</tr>
<tr>
<td>$A \rightarrow B$</td>
<td></td>
</tr>
<tr>
<td>$B \rightarrow \varepsilon$</td>
<td></td>
</tr>
<tr>
<td>3. $L = {0^n1^n \mid n \geq 0}$ is a CFL</td>
<td>3. By Statements #1 and #2</td>
</tr>
</tbody>
</table>
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: \( a + a \times a \)

- \( \text{EXPR} \Rightarrow \)
- \( \text{EXPR} + \text{TERM} \Rightarrow \)
- \( \text{EXPR} + \text{TERM} \times \text{FACTOR} \Rightarrow \)
- \( \text{EXPR} + \text{TERM} \times a \Rightarrow \)

... 

**RIGHTMOST DERIVATION**

- \( \text{EXPR} \Rightarrow \)
- \( \text{EXPR} + \text{TERM} \Rightarrow \)
- \( \text{TERM} + \text{TERM} \Rightarrow \)
- \( \text{FACTOR} + \text{TERM} \Rightarrow \)
- \( a + \text{TERM} \)

... 

**LEFTMOST DERIVATION**

“|” symbol = Shorthand for multiple rules with same LHS variable
Derivations and Parse Trees

\[ A \Rightarrow OA1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111 \]

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

Leftmost derivation

- `EXPR =>`
- `EXPR + TERM =>`
- `TERM + TERM =>`
- `FACTOR + TERM =>`
- `a + TERM`
- `...`

Rightmost derivation

- `EXPR =>`
- `EXPR + TERM =>`
- `EXPR + TERM x FACTOR =>`
- `EXPR + TERM x a =>`
- `...`

A parse tree represents a CFG computation ... like a sequence of states represents a DFA computation

A Parse Tree gives “meaning” to a string
Ambiguity grammar $G_{5}$:

\[
(EXPR) \rightarrow (EXPR) + (EXPR) \mid (EXPR) \times (EXPR) \mid (EXPR) \mid a
\]

Same string, different derivation, and different parse tree!

So this string has two meanings!
Ambiguity

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*, i.e., represent *two different computations!* (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");
```

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

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

This string has 2 parsings, and thus 2 meanings!

Ambiguous grammars are confusing. A computation on a string should ideally have only one result.

Thus in practice, we typically focus on the unambiguous subset of CFGs (CFLs) (more on this later)

Problem is, there’s no easy way to create an unambiguous grammar (it’s up to language designers to “be careful”)
Designing Grammars: Basics

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

   - E.g., $0^n1^n$
     - $A \rightarrow 0A1$
     - # 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 and rule combines two smaller grammars
“\(\mid\)” = “or” = union (combines 2 rules with same left side)
(Closed) Operations for 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 Example: Designing grammars

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

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

1) come up with examples: In the language: 010, 101, 11011 1, 0 ?
Not in the language: 10, 01, 110 $\varepsilon$ ?

2) Create CFG:

$$S \rightarrow 0C'0 \mid 1C'1 \mid 0 \mid 1$$

“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”

3) Check CFG: generates examples in the language; does not generate examples not in language
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>A ??? 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>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><strong>Push-down Automaton</strong> (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>thm</strong></td>
<td><strong>def</strong></td>
</tr>
<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></td>
<td></td>
</tr>
<tr>
<td><strong>def</strong></td>
<td></td>
</tr>
<tr>
<td>Finite Automaton (FSM)</td>
<td><strong>thm</strong></td>
</tr>
<tr>
<td>An FSM recognizes a Regular Lang</td>
<td>Push-down Automaton (PDA)</td>
</tr>
<tr>
<td><strong>DIFERENCE:</strong></td>
<td></td>
</tr>
<tr>
<td>A Regular Lang is defined with a FSM</td>
<td>A PDA recognizes a CFL</td>
</tr>
<tr>
<td><em>Proved:</em> Reg Expr ⇔ Reg Lang</td>
<td><strong>Must prove:</strong> PDA ⇔ CFL</td>
</tr>
<tr>
<td><strong>DIFERENCE:</strong></td>
<td></td>
</tr>
</tbody>
</table>
Submit in-class work 3/18

On gradescope