CS420

Context-Free Languages (CFLs)

Monday, March 1, 2021
Announcements

• HW4 in

• HW5 out
  • Due Sunday 3/7/2021 11:59pm EST

• **Reminder:** HW submissions must include README files
  • Cite your sources and collaborators
    • This is how (computer) scientists work
  • Answers must be written in your own words
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 $\{0^n1^n \mid n \geq 0\}$. 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)?

• (This language sort of resembles HTML/XML)
A Context-Free Grammar (CFG)

- Top variable is **Start variable**
- **Variables** (also called a nonterminal)
- **Terminals**
- **Substitution rules** (a.k.a., productions)

- $A \rightarrow 0A1$
- $A \rightarrow B$
- $B \rightarrow #$
- **Terminals** (analogous to a DFA’s alphabet)
CFGs: Formal Definition

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 (Regexp)</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>
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 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:

```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
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

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

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

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

A CFG represents a Language!

Strings in CFG's language = all possible generated strings

Stop when string is all terminals

Start variable

After applying 1st rule

Used 2nd rule

Used last rule
Formal Definition of a CFL

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:

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

\( L(G_1) \) is \( \{0^n\#1^n \mid n \geq 0\} \)
Formal Definition of a Derivation

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

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

This sequence is called a **derivation**

If \(u, v,\) and \(w\) are strings of variables and terminals, and \(A \rightarrow w\) is a rule of the grammar, we say that \(uAv\) **yields** \(uwv\), written \(uAv \Rightarrow uwv\). Say that \(u\) **derives** \(v\), written \(u \Rightarrow^* v\), if \(u = v\) or if a sequence \(u_1, u_2, \ldots, u_k\) exists for \(k \geq 0\) and

\[
u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.
\]

The **language of the grammar** is \(\{ w \in \Sigma^* | S \Rightarrow^* w \}\).
In-class exercise: derivations

\[
\begin{align*}
\langle \text{EXPR} \rangle & \rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\
\langle \text{TERM} \rangle & \rightarrow \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\
\langle \text{FACTOR} \rangle & \rightarrow ( \langle \text{EXPR} \rangle ) \mid a
\end{align*}
\]

• Come up with a derivation (a sequence of substs) for string:
  • a + a x a
A String Can Have Multiple Derivations

\[
\begin{align*}
\langle \text{EXPR} \rangle & \rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\
\langle \text{TERM} \rangle & \rightarrow \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\
\langle \text{FACTOR} \rangle & \rightarrow ( \langle \text{EXPR} \rangle ) \mid a
\end{align*}
\]

- \text{EXPR} =>
- \text{EXPR} + \text{TERM} =>
- \text{EXPR} + \text{TERM} \times \text{FACTOR} =>
- \text{EXPR} + \text{TERM} \times a =>
- ...

\textbf{RIGHTMOST DERIVATION}

- \text{EXPR} =>
- \text{EXPR} + \text{TERM} =>
- \text{TERM} + \text{TERM} =>
- \text{FACTOR} + \text{TERM} =>
- a + \text{TERM}
- ...

\textbf{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**

A Parse Tree gives “meaning” to a string
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 =>
...

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

grammar $G_5$:

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

Same string,
Different derivation,
and different parse tree!

\[\text{a} + \text{a} \times \text{a}\]

\[\text{a} + \text{a} \times \text{a}\]
**Definition 2.7**

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.** |
| There’s no guaranteed way to create an unambiguous grammar (just have to think about it) |
Designing Grammars: Basics

• Think about what you want to “link” together
• E.g., XML
  • ELEMENT $\to$ <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 grammar for lang \( \{0^n1^n | n \geq 0\} \cup \{1^n0^n | n \geq 0\} \)
    - 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: 
      \[
      S \rightarrow S_1 \mid S_2 \\
      S_1 \rightarrow 0S_11 \mid \varepsilon \\
      S_2 \rightarrow 1S_20 \mid \varepsilon
      \]

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

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

• “Or”: \[ S \rightarrow S_1 | S_2 \]

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

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

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

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

- $S \rightarrow 0C'0 \mid 1C'1 \mid \varepsilon$  
  “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 3/1

On gradescope