Regular vs Context-Free Languages (and others?)

Monday, March 20, 2023

UMB CS 420
Announcements

• HW 5 in
  • Due Sunday March 19, 2023 11:59pm EST

• HW 6 out
  • Due Sunday March 26, 2023 11:59pm EST

Quiz Preview

• Is every CFL a regular language?
Is This Diagram “Correct”?
(What are the statements implied by this diagram?)

1. Every regular language is a CFL
2. Not every CFL is a regular language
How to **Prove** This Diagram “Correct”? 

1. **Every regular language is a CFL**

2. **Not every CFL is a regular language**

   - Find a CFL that is not regular

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

   - It’s a CFL
   
     *Proof:* CFG \( S \rightarrow 0S1 | \varepsilon\)

   - It’s not regular
     
     *Proof:* by contradiction using the Pumping Lemma
How to **Prove** This Diagram “Correct”?

1. Every regular language is a CFL
   - For any regular language $A$, show ...
     - it has a CFG or PDA

2. Not every CFL is a regular language
   - A regular language is represented by a:
     - DFA
     - NFA
     - Regular Expression
Regular Languages are CFLs: 3 Ways to Prove

• DFA → CFG or PDA

• NFA → CFG or PDA

• Regular expression → CFG or PDA

See HW 6!

Are there other interesting subsets of CFLs?
Deterministic CFLs and DPDAs
Previously: Generating Strings

Generating strings:
1. Start with start variable,
2. Repeatedly apply CFG rules to get string (and parse tree)

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

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

**Generating strings:**
1. Start with **start variable**,
2. Repeatedly apply CFG rules to get string (and parse tree)

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

In practice, **opposite** is more interesting:
1. **Start** with a **string**,
2. Then **parse** it into **parse tree**

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

- In practice, **parsing** a string more important than **generating** one
  - E.g., a **compiler** (first) parses source code into a parse tree
  - (Actually, *any* program with string inputs must first parse it)
Previously: Example CFG $\Rightarrow$ PDA

Example Derivation using CFG:
$S \Rightarrow aTb$ (using rule $S \rightarrow aTb$)
$\Rightarrow aTab$ (using rule $T \rightarrow Ta$)
$\Rightarrow aab$ (using rule $T \rightarrow \varepsilon$)

Machine is parsing:
1. Start with (input) string,
2. Find rules that generate string
Generating vs Parsing

- In practice, **parsing** a string more important than **generating** one
  - E.g., a **compiler** (first step) parses source code into a parse tree
  - (Actually, *any* program with string inputs must first parse it)

- But: the PDAs we’ve seen are **non-deterministic** (like NFAs)
Previously: (Nondeterministic) PDA

\[ S \rightarrow aTb \textbf{[b]} \]
\[ T \rightarrow Ta \mid \varepsilon \]

This PDA nondeterministically "tries all grammar rules at once"

A parser implementation can't do this!
Generating vs Parsing

• In practice, parsing a string more important than generating one
  • E.g., a compiler (first step) parses source code into a parse tree
  • (Actually, any program with string inputs must first parse it)

• But: the PDAs we’ve seen are non-deterministic (like NFAs)

• Compiler’s parsing algorithm must be deterministic

• So: to model parsers, we need a Deterministic PDA (DPDA)
A determinist pushdown automaton is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\), where \(Q, \Sigma, \Gamma,\) and \(F\) are all finite sets, and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet,
3. \(\Gamma\) is the stack alphabet,
4. \(\delta: \mathcal{Q} \times \Sigma \times \Gamma \rightarrow (\mathcal{Q} \times \Gamma) \cup \{\emptyset\}\) is the transition function,
5. \(q_0 \in Q\) is the start state, and
6. \(F \subseteq Q\) is the set of accept states.

The language of a DPDA is called a deterministic context-free language.

A pushdown automaton is a 6-tuple

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet,
3. \(\Gamma\) is the stack alphabet,
4. \(\delta: \mathcal{Q} \times \Sigma \times \Gamma \rightarrow \mathcal{P}(\mathcal{Q} \times \Gamma)\)
5. \(q_0 \in Q\) is the start state, and
6. \(F \subseteq Q\) is the set of accept states.

**Difference:** DPDA has only one possible action, for any given state, input, and stack op (similar to DFA vs NFA).

Must take into account \(\varepsilon\) reads or stack ops! E.g., if \(\delta(q, a, X)\) does “something”, then \(\delta(q, \varepsilon, X)\) must “do nothing”
DPDAs are **Not** Equivalent to PDAs!

- A PDA can non-deterministically **“try all rules”** (abandoning failed attempts);
- A DPDA must **choose one rule at each step!** (can't go back after reading input!)

\[
R \rightarrow S \mid T \\
S \rightarrow aSb \mid ab \\
T \rightarrow aTbb \mid abb
\]

**Parsing** = deriving reversed:
- **start** with string, **end** with parse tree

```
aaa bbb bbb \rightarrow aaSbb
```

When parsing this string, when does it know which rule was used, $S$ or $T$?

```
aaa bbbb bbb \rightarrow aaTbbb
```

Choosing “correct” rule depends on rest of the input!

**PDAs recognize CFLs, but DPDAs only recognize DCFLs!** (a subset of CFLs)
Subclasses of CFLs

Unambiguous CFLs / PDAs

DCFLs

Programming language parsers / compilers are ideally in here

All CFLs
Compiler Stages

A program string (chars) (e.g., `a := (5 + 3); ...`)

DFAs (recognizing regular languages) in here!

Lexer

Program “words”
(e.g., `ID(a) ASSIGN LPAREN NUM(5) PLUS NUM(3) RPAREN SEMI ...`)
A Lexer Implementation

DFA

Remember our analogy:
- DFAs are like programs
- All possible DFA tuples is like a programming language

This DFA is a real program!

A “lex” tool converts the program:
- from “DFA Lang” ...
- to an equivalent one in C!

DFAs (represented as regular expressions)!

```c
{%
/* C Declarations: */
#include "tokens.h" /* definitions of IF, ID, NUM, ... */
#include "errormsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
define ADJ (EM_tokPos=charPos, charPos+=yyleng)
%
/* Lex Definitions: */
digits [0-9]+%
/* Regular Expressions and Actions: */
if
[0-9][0-9-9]*
{ADJ; return IF;}
{ADJ; yylval.sval=String(yytext);
 return ID;}
digits
{ADJ; yylval.ival=atoi(yytext);
 return NUM:;}
digits
{ADJ; yylval.fval=atof(yytext);
 return REAL;}
"-"[a-z]*\n\n| \n| \n| \t
{ADJ;}
{ADJ; EM_error("illegal character");}
```
Compiler Stages

A program (chars) (e.g., \( a : = ( 5 + 3 ) ; \ldots \))

Lexer

DFAs (recognizing regular languages) in here!

Program “words” (e.g., \( \text{ID}(a) \ \text{ASSIGN} \ \text{LPAREN} \num{5} \ \text{PLUS} \ \num{3} \ \text{RPAREN} \ \text{SEMI} \ldots \))

Parser

DPDAs (recognizing DCFLs) in here!

Abstract Syntax tree (AST), i.e., a parse tree!
A Parser Implementation

```c
{%
int yylex(void);
void yyerror(char *s) { EM_error(EM_tokPos, "%s", s); }%
%}
%token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN
%start prog
%
prog: stmlist

stm: ID ASSIGN ID
   | WHILE ID DO stm
   | BEGIN stmlist END
   | IF ID THEN stm
   | IF ID THEN stm ELSE stm

stmlist: stm
    | stmlist SEMI stm

Remember our analogy: CFGs are like programs
This CFG is a real program!
A “yacc” tool converts the program:
- from “CFG Lang” ...
- to an equivalent one in C!
```

Just write the CFG!

130
DPDAs are **Not** Equivalent to PDAs!

\[ R \rightarrow S \mid T \]
\[ S \rightarrow aSb \mid ab \]
\[ T \rightarrow aTbb \mid abb \]

- **PDA**: can non-deterministically “try all rules” (abandoning failed attempts);
- **DPDA**: must choose one rule at each step!

**Should use S rule**

**Should use T rule**

When parsing reaches this position, does it know which rule, \( S \) or \( T \)?:

To choose “correct” rule, need to “look ahead” at rest of the input!

PDAs recognize CFLs, but **DPDAs only recognize DCFLs**! (a subset of CFLs)

Parsing = generating reversed:
- start with string
- end with parse tree
Subclasses of CFLs

2 parser design decisions:
1) Parse from left, or from right
2) choose “look ahead” amount

Programming language parsers / compilers are ideally in here
LL parsing

- L = left-to-right
- L = leftmost derivation

Game: “You’re the Parser”:
Guess which rule applies?

1 \[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \]
2 \[ S \rightarrow \text{begin } S \ L \]
3 \[ S \rightarrow \text{print } E \]

if 2 = 3 begin print 1; print 2; end else print 0

4 \[ L \rightarrow \text{end} \]
5 \[ L \rightarrow ; \ S \ L \]
6 \[ E \rightarrow \text{num } = \text{ num} \]
LL parsing

- L = left-to-right
- L = leftmost derivation

1 \( S \rightarrow \text{if } E \text{ then } S \text{ else } S \)
2 \( S \rightarrow \text{begin } S L \)
3 \( S \rightarrow \text{print } E \)
4 \( L \rightarrow \text{end} \)
5 \( L \rightarrow ; \ S \ L \)
6 \( E \rightarrow \text{num} = \text{num} \)

if 2 = 3 begin print 1; print 2; end else print 0
LL parsing

• L = left-to-right
• L = leftmost derivation

1 $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
2 $S \rightarrow \text{begin } S \text{ } L$
3 $S \rightarrow \text{print } E$
4 $L \rightarrow \text{end}$
5 $L \rightarrow ; \text{ } S \text{ } L$
6 $E \rightarrow \text{num } = \text{ num}$

if 2 = 3 begin print 1; print 2; end else print 0
LL parsing

• L = left-to-right
• L = leftmost derivation

1 \[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \]
2 \[ S \rightarrow \text{begin } S \text{ } L \]
3 \[ S \rightarrow \text{print } E \]

4 \[ L \rightarrow \text{end} \]
5 \[ L \rightarrow ; \text{ } S \text{ } L \]
6 \[ E \rightarrow \text{num} = \text{num} \]

if 2 = 3 begin print 1; print 2; end else print 0

“Prefix” languages (Scheme/Lisp) are easily parsed with LL parsers (zero lookahead)
LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

1. $S \to S \; ; \; S$
2. $S \to id \; := \; E$
3. $S \to print \left( L \right)$
4. $E \to id$
5. $E \to num$
6. $E \to E \; + \; E$

```
    a := 7;
    b := c + (d := 5 + 6, d)
```

When parse is here, can't determine whether it's an assign ($:=)$ or addition ($+$)

Need to *save* input (lookahead) to some memory, like a stack! This is a job for a (D)PDA!
LR parsing

• L = left-to-right
• R = rightmost derivation

\[ S \rightarrow S ; \ S \]
\[ S \rightarrow \text{id} := E \]
\[ E \rightarrow \text{id} \]
\[ E \rightarrow \text{num} \]
\[ S \rightarrow \text{print} \ ( \ L \ ) \]
\[ E \rightarrow E + E \]

\[
\begin{align*}
a & := 7 ; \\
b & := c + ( \ d := 5 + 6 , \ d )
\end{align*}
\]

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a := 7 ; b := c + ( d := 5 + 6 , d ) $</td>
<td>push = “push”</td>
</tr>
</tbody>
</table>

State name
LR parsing

- $L = \text{left-to-right}$
- $R = \text{rightmost derivation}$

$$S \rightarrow S \; ; \; S \quad E \rightarrow \text{id}$$
$$S \rightarrow \text{id} \; := \; E \quad E \rightarrow \text{num}$$
$$S \rightarrow \text{print} \; ( \; L \; ) \quad E \rightarrow E \; + \; E$$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 \ id_4</td>
<td>a := 7 ; b := c + ( d := 5 + 6 , d ) $</td>
<td>shift</td>
</tr>
<tr>
<td>1 \ id_4</td>
<td>:= 7 ; b := c + ( d := 5 + 6 , d ) $</td>
<td>shift</td>
</tr>
<tr>
<td>1 \ id_4 := 6</td>
<td>7 ; b := c + ( d := 5 + 6 , d ) $</td>
<td>shift</td>
</tr>
</tbody>
</table>
LR parsing

- L = left-to-right
- R = rightmost derivation

\[
S \rightarrow S ; S \\
S \rightarrow id := E \\
E \rightarrow num \\
S \rightarrow print ( L ) \\
E \rightarrow E + E
\]
LR parsing

- **L** = left-to-right
- **R** = rightmost derivation

1. \( S \rightarrow S ; S \)
2. \( S \rightarrow \text{id} := E \)
3. \( S \rightarrow \text{print} ( L ) \)
4. \( E \rightarrow \text{id} \)
5. \( E \rightarrow \text{num} \)
6. \( E \rightarrow E + E \)

---

**Stack**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>id(_4)</td>
<td>shift</td>
</tr>
<tr>
<td>1</td>
<td>id(_4) := 6</td>
<td>shift</td>
</tr>
<tr>
<td>1</td>
<td>num(_{10})</td>
<td>reduce ( E \rightarrow \text{num} )</td>
</tr>
</tbody>
</table>

Can determine (rightmost) rule
LR parsing

- \( L = \) left-to-right
- \( R = \) rightmost derivation

\[
\begin{align*}
1 & \quad S \rightarrow S ; S \\
2 & \quad S \rightarrow id := E \\
3 & \quad S \rightarrow \text{print} ( L ) \\
4 & \quad E \rightarrow \text{id} \\
5 & \quad E \rightarrow \text{num} \\
6 & \quad E \rightarrow E + E
\end{align*}
\]
LR parsing

- L = left-to-right
- R = rightmost derivation

\[
\begin{align*}
S & \rightarrow S ; S \\
S & \rightarrow \text{id} ::= E \\
S & \rightarrow \text{print} \,( L ) \\
E & \rightarrow \text{id} \\
E & \rightarrow \text{num} \\
E & \rightarrow E + E
\end{align*}
\]
To learn more, take a Compilers Class!

A program (string of chars)

Lexer
(DFAs / NFAs)

Program “words”

Parser
(DPDAs)

Abstract Syntax tree (AST)

???

This phase needs computation that goes beyond CFLs
In-class quiz 3/20

See gradescope