Deterministic CFLs, PDAs, and Parsing

Wednesday, October 6, 2021
Announcements

• Reminder: no class next Monday 10/11

• HW4 due Sunday 10/17 11:59pm
  • second Sunday from today
Previously: CFLs, CFGs, and Parse Trees

**Generating** strings: start with start variable, Apply rules to get a string (and parse tree)

\[
\begin{align*}
A & \rightarrow 0A1 \\
A & \rightarrow B \\
B & \rightarrow \# \\
A & \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111
\end{align*}
\]
Today: Generating vs Parsing

**Generating** strings: start with **start variable**, then apply rules to get a string and parse tree

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

In practice, the opposite is more interesting: start with a string, 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 is 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)

- But a compiler / parser (algorithm) must be deterministic

- The PDAs we’ve seen are non-deterministic (like NFAs)

- **So:** to model parsers, we need a **Deterministic** PDA (DPDA)
Last time: (Nondeterministic) PDA

\[ S \rightarrow aTb \]
\[ T \rightarrow Ta | \varepsilon \]

This PDA nondeterministically “tries all grammar rules at once”

A parser implementation can’t do this!
DPDA: Formal Definition

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

A **deterministic 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. \( \delta: Q \times \Sigma \times \Gamma \rightarrow (Q \times \Gamma) \cup \{\emptyset\} \) is the transition function,
5. \( q_0 \in Q \) is the start state, and
6. \( F \subseteq Q \) is the set of accept states.

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. \( \delta: Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma) \) is the transition function,
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)

This must take into account \( \varepsilon \) reads or stack ops! E.g., if \( \delta(q, a, X) \) is valid, then \( \delta(q, \varepsilon, X) \) must not be
DPDAs are **Not** Equivalent to PDAs!

\[
\begin{align*}
R & \rightarrow S \mid T \\
S & \rightarrow aSb \mid ab \\
T & \rightarrow aTbb \mid abb
\end{align*}
\]

- **Should use S rule**
- **Should use T rule**
- **Parsing is deriving reversed:** start with string, end with parse tree
- **When parsing reaches this input position, which rule should it use, S or T?**
- **Don’t know which rule to use because we can’t see rest of the input!**

A PDA non-deterministically “tries all rules” (abandons failed attempts) but a DPDA gets only one try!

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

Programming language parsers / compilers are ideally in here

DCFLs
Compiler Stages

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

Performed by Regular expressions and DFAs!

Lexer

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

```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 [a-z][a-z0-9]*
{adj; return IF;}
digits
{adj; return ID;}
{adj; yylval.sval=String(yytext);
return NUM;}
({digits}"\n\n")|([0-9]*\{digits\})
{adj;
return REAL;}
("-"[a-z]*\"\n")|(" "|"\n"|"\t")+{adj;}
{adj; EM_error("illegal character");}
```
Compiler Stages

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

Performed by Regular expressions and DFAs!

DCFLs and DPDAs

Lexer

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

Parser

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

```
{%
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
```
Parsing

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

A parser must be able to choose one correct rule, when reading input left-to-right
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. \( E \rightarrow \text{num = num} \)

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. $S \rightarrow \text{begin } S \ L$
3. $S \rightarrow \text{print } E$
4. $L \rightarrow \text{end}$
5. $L \rightarrow ; \ S \ L$
6. $E \rightarrow \text{num = num}$

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

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. $S \rightarrow \text{begin } S \ L$
3. $S \rightarrow \text{print } E$
4. $L \rightarrow \text{end}$
5. $L \rightarrow ; \ S \ L$
6. $E \rightarrow \text{num = num}$

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

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

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

\[
\begin{align*}
S & \rightarrow S; S \\
S & \rightarrow \text{id} := E \\
S & \rightarrow \text{print} ( L ) \\
E & \rightarrow \text{id} \\
E & \rightarrow \text{num} \\
E & \rightarrow E + E
\end{align*}
\]

\[
a \ := \ 7 \\
b \ := \ c \ + \ ( d := 5 + 6, \ d )
\]

When parse is here, can’t determine whether it’s an assign or a plus

Need to save input somewhere, like a stack: this is a job for a (D)PDA!!

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 id4</td>
<td>a := 7 ; b := c + ( d := 5 + 6, d ) $</td>
<td>shift</td>
</tr>
<tr>
<td>1 id4 := 6</td>
<td>7 ; b := c + ( d := 5 + 6, d ) $</td>
<td>shift</td>
</tr>
<tr>
<td>1 id4 := 6 num10</td>
<td>7 ; b := c + ( d := 5 + 6, d ) $</td>
<td>shift</td>
</tr>
<tr>
<td>1 id4 := 6 E11</td>
<td>b := c + ( d := 5 + 6, d ) $</td>
<td>reduce E → num</td>
</tr>
<tr>
<td>1 S2</td>
<td>b := c + ( d := 5 + 6, d ) $</td>
<td>reduce S → id := E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shift</td>
</tr>
</tbody>
</table>
LR parsing

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

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

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

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

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

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

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

\[
\begin{align*}
S & \rightarrow S \; ; \; S \\
S & \rightarrow id := E \\
S & \rightarrow \text{print} \; ( \; L \; ) \\
E & \rightarrow id \\
E & \rightarrow \text{num} \\
E & \rightarrow E + E
\end{align*}
\]
LR parsing

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

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 id₄</td>
<td>a := 7 ; b := c + ( d := 5 + 6 , d ) $</td>
<td>shift</td>
</tr>
<tr>
<td>1 id₄ :=6</td>
<td>:= 7 ; b := c + ( d := 5 + 6 , d ) $</td>
<td>shift</td>
</tr>
<tr>
<td>1 id₄ :=6 num₁₀</td>
<td>7 ; b := c + ( d := 5 + 6 , d ) $</td>
<td>shift</td>
</tr>
<tr>
<td>1 id₄ :=6 E₁₁</td>
<td>; b := c + ( d := 5 + 6 , d ) $</td>
<td>reduce E → num</td>
</tr>
<tr>
<td>1 S₂</td>
<td>; b := c + ( d := 5 + 6 , d ) $</td>
<td>reduce S → id := E</td>
</tr>
<tr>
<td></td>
<td>; b := c + ( d := 5 + 6 , d ) $</td>
<td>shift</td>
</tr>
</tbody>
</table>
To learn more, take a Compilers Class!

A program (string of chars)

Lexer
(DFAs / NFAs)

Program “words”

Parser
(DPDAs)

Abstract Syntax tree (AST)

Need computation that goes beyond CFLs
Non-CFLs
Flashback: Pumping Lemma for Reg Langs

- The Pumping Lemma describes how strings repeat

- Regular language strings can (only) repeat using Kleene pattern
  - But the substrings are independent!

- A non-regular language:
  \[ \{0^n1^n | n \geq 0\} \]
  - Kleene star can’t express this pattern: 2nd part depends on (length of) 1st part

- How do CFLs repeat?
Repetition and Dependency in CFLs

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

\{ 0^n#1^n | n \geq 0 \}

Parts before/after repetition point are linked

Repetition

Repetition
How Do Strings in CFLs Repeat?

- Strings in regular languages repeat states

- Strings in CFLs repeat subtrees in the parse tree
Pumping Lemma for CFLS

Pumping lemma for context-free languages

If $A$ is a context-free 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 five pieces $s = uvxyz$ satisfying the conditions:

1. for each $i \geq 0$, $uv^ixyz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Now there are two pumpable parts. But they must be pumped together!

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. $|xy| \leq p$. 
Non CFL example: \( D = \{ww \mid w \in \{0,1\}^*\} \)

Previously: \( D \) is nonregular: unpumpable counterexample \( s: 0^p 1 \)
Now: this \( s \) can be pumped according to CFL pumping lemma:

\[
\begin{align*}
0^p 1 & \\
000 \ldots 000 & 0 & 1 & 000 \ldots 0001 \\
& u & v & x & y & z
\end{align*}
\]

\( \text{Pumping } v \text{ and } y \text{ (together) produces string still in } D \)

• CFL Pumping Lemma conditions:
  \( \checkmark \) 1. for each \( i \geq 0 \), \( uv^i xy^i z \in A \),
  \( \checkmark \) 2. \( |vy| > 0 \), and
  \( \checkmark \) 3. \( |vxy| \leq p \).

This doesn't prove that the language is a CFL!
It only means that the attempt to prove that
the language is not a CFL failed.
Non CFL example: \( D = \{ww \mid w \in \{0,1\}^*\} \)

- Choose another string \( s \):

\[
0^p1^p0^p1^p
\]

If \( vyx \) is contained in first or second half, then any pumping will break the match.

So \( vyx \) must straddle the middle But any pumping still breaks the match because order is wrong.

- CFL Pumping Lemma conditions:

1. for each \( i \geq 0 \), \( uv^i xy^i z \in A \),
2. \( |vy| > 0 \), and
3. \( |vxy| \leq p \).

This language is not a CFL!
The **Pumping lemma for context-free languages** states that if $A$ is a context-free 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 five pieces $s = uvxyz$ satisfying the conditions:

1. for each $i \geq 0$, $uv^i xy^i z \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$. 
Review: Regular Language Pumping Lemma

• The pumping length $p$ for a language $L$ is ...
  ... the # of states in that language’s NFA!

• If string length > # of states, then some state must repeat

• If a state is repeated once, then it can repeat multiple times
Repeating Pattern in CFL Strings?

• When are we guaranteed to have a repeated subtree?
  • When height of parse tree > # of rules!

• Let \( k = \# \text{ of rules} \) and \( b = \text{longest rule RHS length} \)
  • Then the length string where we know there’s a repeat is \( b^k \)
  • I.e., pumping length = \( b^k \)??

Pumping lemma for context-free languages. If \( A \) is a context-free 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 five pieces \( s = uvxyz \) satisfying the conditions

1. for each \( i \geq 0, u v^i x y^i z \in A, \)
2. \( |vxy| > 0, \) and
3. \( |xy| \leq p. \)

Pumping Length could be too short!
A Pumpable Non-CFL?

- CFL Pumping Lemma says:
  - “All CFLs are pumpable”
  - So if we find a non-pumpable language ... it’s not a CFL!

- Pumping Lemma does not say:
  - “All non-CFLs are not pumpable”
  - (statement != it’s inverse)
  - So Pumping Lemma might not be able to prove some non-CFLs!

Example:

\[ L = \{a^i b^j c^k d^l \mid i = 0 \text{ or } j = k = l \} \]

- For any counterexample, split into \( uvxyz \) where,
  - \( v = \) first char
  - \( z = \) remaining chars
  - \( u = x = y = \varepsilon \)

- If there are \( a \)s ...
  - ... it’s pumpable bc # of \( a \)s is arbitrary
- There there are no \( a \)s
  - ... it’s pumpable bc # of other chars is arbitrary

This language is pumpable ... but not a CFL! (can’t come up with a CFG)
Ogden’s Lemma (generalizes pumping lemma)

Ogden’s lemma is: If $L$ is a CFL, then there is a constant $n$, such that if $z$ is any string of length at least $n$ in $L$, in which we select at least $n$ positions to be distinguished, then we can write $z = uvwxy$, such that:

1. $vwx$ has at most $n$ distinguished positions.
2. $vx$ has at least one distinguished position.
3. For all $i$, $uv^iwx^iy$ is in $L$.

Says that every long enough segment must be pumpable

Example:

$$L = \{a^ib^jc^kd^l \mid i = 0 \text{ or } j = k = l\}$$

This language is not a CFL because it doesn’t satisfy Ogden’s Lemma

Counterexample: $ab^n c^n d^n$

- $n$ “distinguished” positions must include non-$a$ character
  - Impossible to pump no matter which $n$ chars are chosen
A Practical Non-CFL

• **XML**
  - `ELEMENT → <TAG>CONTENT</TAG>`
  - Where `TAG` is any string

• **XML also looks like this non-CFL:** \[ D = \{ww \mid w \in \{0,1\}^*\} \]

• **This means XML is not context-free!**
  - **Note:** HTML is context-free because ...
  - ... there are only a finite number of tags,
  - so they can be embedded into a finite number of rules.

• **In practice:**
  - XML is parsed as a CFL, with a CFG
  - Then matching tags checked in a 2\textsuperscript{nd} pass with a more powerful machine ...
Next Time: A More Powerful Machine...

\( M_1 \) accepts its input if it is in language: \( B = \{ w#w \mid w \in \{0,1\}^* \} \)

\( M_1 = \text{"On input string } w:\)

1. **Zig-zag** across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

Infinite memory, initially starts with input

Can move to, and read/write from, arbitrary memory locations
In-class quiz 10/6

See gradescope