UMB CS 420

Non-CFLs

Tuesday, October 25, 2022
Announcements

• HW 5 in
  • Due 10/23 11:59pm EST

• HW 6 out
  • Due 10/30 11:59pm EST
Last Time: Generating vsParsing

• 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:

• PDAs are **non-deterministic** (like NFAs)
• Compiler’s parsing algorithm must be **deterministic**

• **So**: to model parsers, we need a **Deterministic PDA (DPDA)**
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. $\delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow (Q \times \Gamma) \cup \{\emptyset\}$ 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. $\delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow \mathcal{P}(Q \times \Gamma)$
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).

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

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

$R \rightarrow S \mid T$

$S \rightarrow \textcolor{green}{aSb} \mid \text{ab}$

$T \rightarrow \textcolor{red}{aTbb} \mid \text{abb}$

**Should use $S$ rule**

$aaabbb \rightarrow aaSbb$

**Should use $T$ rule**

$aaabbb \rightarrow aaTbbb$

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)**
Subclasses of CFLs

- DCFLs
- Programming language parsers / compilers are ideally in here

2 parser design decisions:
1) Parse from **left**, or from **right**
2) choose “look ahead” amount
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 \[ S \rightarrow \text{print } E \]

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

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

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. $E \rightarrow \text{num } = \text{num}$

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

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

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

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

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

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!

<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>shift</td>
</tr>
<tr>
<td>1 id(_4)</td>
<td></td>
<td>push</td>
</tr>
<tr>
<td>1 id(_4) := 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 id(<em>4) := 6 num(</em>{10})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 id(<em>4) := 6 ( E</em>{11} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ( S_2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 7 ; b := c + ( d := 5 + 6 , d ) )</td>
<td>reduce ( E \rightarrow \text{num} )</td>
</tr>
<tr>
<td>( ; b := c + ( d := 5 + 6 , d ) )</td>
<td>reduce ( S \rightarrow \text{id} := E )</td>
</tr>
<tr>
<td>( ; b := c + ( d := 5 + 6 , d ) )</td>
<td>shift</td>
</tr>
</tbody>
</table>

State name
LR parsing

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

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

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

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

1
1 id4
1 id4 := 6
1 id4 := 6 num10
1 id4 := 6 E11
1 S2

Input

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

Action

shift
shift
shift
reduce \( E \rightarrow \text{num} \)
reduce \( S \rightarrow \text{id} := E \)
shift

Can determine (rightmost) rule
LR parsing

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

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

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

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

This phase needs computation that goes beyond CFLs
**Flashback:** Pumping Lemma for Regular Langs

• Pumping Lemma describes how strings **repeat**

• Regular language strings repeat using Kleene start operation
  • substrings are independent!

• A non-regular language:
  \[
  \{0^n1^n \mid n \geq 0\}
  \]

  *Kleene star can’t express this pattern: 2nd part depends on (length of) 1st part*

• Q: 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

\[ A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111 \]
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 \( i \geq 0 \), \( uv^i xy^i z \in A \),
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 conditions:

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

Two pumpable parts, pumped together
A Non CFL example

\[ B = \{a^n b^n c^n \mid n \geq 0\} \text{ is not context free} \]

Intuition

• Strings in CFLs can have two parts that are “pumped” together
• This language requires three parts to be “pumped” together
• So it’s not a CFL!
Want to prove: $a^n b^n c^n$ is not a CFL

Proof (by contradiction):

- **Assume**: $a^n b^n c^n$ is a CFL
  - So it must satisfy the pumping lemma for CFLs
  - i.e., all strings $\geq$ length $p$ are pumpable
- **Counterexample** = $a^p b^p c^p$

Now we must find a contradiction ...

Contradiction if: string $\geq$ length $p$ that is not splittable into $uvxyz$ where $v$ and $y$ are pumpable

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 $i \geq 0$, $uv^i x y^i z \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Reminder: CFL Pumping lemma says: all strings $a^n b^n c^n \geq$ length $p$ are splittable into $uvxyz$ where $v$ and $y$ are pumpable
Want to prove: \( a^n b^n c^n \) is not a CFL

**Possible Splits**

**Proof (by contradiction):**

- **Assume:** \( a^n b^n c^n \) is a CFL
  - So it must satisfy the pumping lemma for CFLs
  - I.e., all strings \( \geq \) length \( p \) are pumpable

- **Counterexample:** \( a^p b^p c^p \)

- **Possible Splits** (using condition \# 3: \( |vxy| \leq p \))
  - \( vyx \) is all \( a \)s
  - \( vyx \) is all \( b \)s
  - \( vyx \) is all \( c \)s
  - \( vyx \) has \( a \)s and \( b \)s
  - \( vyx \) has \( b \)s and \( c \)s

So \( a^n b^n c^n \) is not a CFL

(justification: contrapositive of CFL pumping lemma)

---

**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 \( i \geq 0 \), \( uv^i x y^i z \in A \),
2. \( |vxy| > 0 \), and
3. \( |vxy| \leq p \).
Another Non-CFL \( D = \{ww \mid w \in \{0,1\}^* \} \)

Be careful when choosing counterexample \( s: 0^p 1 0^p 1 \)
This \( s \) can be pumped according to CFL pumping lemma:

\[
\begin{array}{c}
0^p 1 \\
\{000 \cdots 000\} \quad 0 \quad 1 \\
u \quad v \quad x \\
\{000 \cdots 0001\} \quad 0^p 1
\end{array}
\]

- 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 doesn’t prove that the language is a CFL! It only means that this attempt to prove that the language is not a CFL failed.
Another Non-CFL \( D = \{ww | w \in \{0,1\}^*\} \)

- Need another counterexample string \( s \):

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

  \[
  0^p 1^p 0^p 1^p
  \]

  - 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. \( |vxy| \leq p \).

Now we have proven that this language is not a CFL!
A Practical Non-CFL

- **XML**
  - ELEMENT $\rightarrow$ `<TAG>CONTENT</TAG>`
  - Where TAG is any string

- **XML also looks like this non-CFL:**
  $$ D = \{ww | w \in \{0,1\}^* \} $$

- 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: $B = \{w\#w | w \in \{0,1\}^*\}$

$M_1 = "On$ $input$ $string$ $w$:"

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 10/25

See gradescope