Announcements

• HW 5 in
  • Due Mon 3/25 12pm noon

• HW 6 out
  • Due Mon 4/1 12pm noon
## Regular Language vs CFL Comparison

<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><strong>describes</strong> a Regular Lang</td>
<td><strong>describes</strong> a CFL</td>
</tr>
</tbody>
</table>

---

---
## Regular Language vs CFL Comparison

<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><em>describes a Regular Lang</em></td>
<td><em>describes a CFL</em></td>
</tr>
<tr>
<td>Finite State Automaton (FSM)</td>
<td>???</td>
</tr>
<tr>
<td><em>recognizes a Regular Lang</em></td>
<td><em>recognizes a CFL</em></td>
</tr>
</tbody>
</table>

*describes a Regular Lang* and *recognizes a Regular Lang* are used to describe the relationship between Regular Languages and their corresponding Finite Automata (regular expressions and Finite State Automata). Similarly, *describes a CFL* and *recognizes a CFL* describe the relationship between Context-Free Languages (Context-Free Grammar) and their corresponding Pushdown Automata.
# Regular Language vs CFL Comparison

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context-Free Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Regular Expression</strong></td>
<td><strong>Context-Free Grammar (CFG)</strong></td>
</tr>
<tr>
<td>describes a Regular Lang</td>
<td>describes a CFL</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Finite State Automaton (FSM)</strong></td>
<td><strong>Push-down Automata (PDA)</strong></td>
</tr>
<tr>
<td>recognizes a Regular Lang</td>
<td>recognizes a CFL</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Regular Language vs CFL Comparison

<table>
<thead>
<tr>
<th>Regular Languages</th>
<th>Context-Free Languages (CFLs)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Regular Expression</strong></td>
<td><strong>Context-Free Grammar (CFG)</strong></td>
</tr>
<tr>
<td><em>describes</em> a Regular Lang</td>
<td><em>describes</em> a CFL</td>
</tr>
<tr>
<td><strong>Finite State Automaton (FSM)</strong></td>
<td><strong>Push-down Automata</strong> <em>(PDA)</em></td>
</tr>
<tr>
<td><em>recognizes</em> a Regular Lang</td>
<td><em>recognizes</em> a CFL</td>
</tr>
</tbody>
</table>

**Proved:**
- Regular Lang $\Leftrightarrow$ Regular Expr ✓

**Must Prove:**
- CFL $\Leftrightarrow$ PDA ??
A lang is a CFL iff some PDA recognizes it

⇒ If a language is a CFL, then a PDA recognizes it
  • We know: A CFL has a CFG describing it (definition of CFL)
  • To prove this part, show: the CFG has an equivalent PDA

⇐ If a PDA recognizes a language, then it’s a CFL
Shorthand: Multi-Symbol Read Transition
Shorthand: Multi-Stack Push Transition

Note the reverse order of pushes
CFG to PDA (sketch)

- Construct PDA from CFG such that:
  - PDA accepts input only if CFG generates it
- PDA:
  - simulates generating a string with CFG rules
  - by (nondeterministically) trying all rules to find the right ones

```
q_{start}
  \rightarrow\varepsilon,\varepsilon \rightarrow S$

q_{loop}
  \rightarrow \varepsilon, A \rightarrow \varepsilon \quad \text{for rule } A \rightarrow w
  \rightarrow a, a \rightarrow \varepsilon \quad \text{for terminal } a

q_{accept}
  \rightarrow \varepsilon, $ \rightarrow \varepsilon
```
**CFG→PDA (sketch)**

- Construct PDA from CFG such that:
  - PDA accepts input only if CFG generates it
- PDA:
  - simulates generating a string with CFG rules
  - by (nondeterministically) trying all rules to find the right ones

Convert: every **CFG rule** to PDA “loop” transition(s) that:
- Pops LHS variable
- Pushes RHS

\[ \epsilon, A \rightarrow w \quad \text{for rule } A \rightarrow w \]
\[ a, a \rightarrow \epsilon \quad \text{for terminal } a \]

Convert: every **terminal** to “loop” transition that:
- Reads input char
- Pops matching char on stack
**CFG→PDA** (sketch)

- Construct PDA from CFG such that:
  - PDA accepts input only if CFG generates it
- PDA:
  - simulates generating a string with CFG rules
  - by (nondeterministically) trying all rules to find the right ones
Example **CFG→PDA**

- **Push start variable onto stack**
- **If: stack top is variable S, pop S and ...**
- **... push rule right-sides (in rev order)**

Production rules:

- $S \rightarrow aTb \mid b$
- $T \rightarrow Ta \mid \varepsilon$

States:

- $q_{start}$
- $q_{loop}$
- $q_{accept}$
Example CFG $\rightarrow$ PDA

\[
S \rightarrow aTb \mid b \\
T \rightarrow Ta \mid \varepsilon
\]
Example **CFG→PDA**

\[ S \rightarrow aTb \mid b \]
\[ T \rightarrow Ta \mid \varepsilon \]

If: stack top is **terminal**, **pop** and read matching input
Example Derivation using CFG:
\[ S \rightarrow aTb \text{ (using rule } S \rightarrow aTb) \]
\[ \Rightarrow aTab \text{ (using rule } T \rightarrow Ta) \]
\[ \Rightarrow aab \text{ (using rule } T \rightarrow \varepsilon) \]

Machine is doing reverse of grammar:
- start with the string,
- Find rules that generate string

PDA Example

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Stack</th>
<th>Equiv Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_{\text{start}})</td>
<td>(aab)</td>
<td>(\varepsilon)</td>
<td>(S \rightarrow aTb)</td>
</tr>
<tr>
<td>(q_{\text{loop}})</td>
<td>(aab)</td>
<td>(aTb)</td>
<td>(S \rightarrow aTb)</td>
</tr>
<tr>
<td>(q_{\text{loop}})</td>
<td>(aab)</td>
<td>(Ta)</td>
<td>(T \rightarrow Ta)</td>
</tr>
<tr>
<td>(q_{\text{loop}})</td>
<td>(ab)</td>
<td>(\varepsilon)</td>
<td>(T \rightarrow \varepsilon)</td>
</tr>
<tr>
<td>(q_{\text{loop}})</td>
<td>(b)</td>
<td>(\varepsilon)</td>
<td>(T \rightarrow \varepsilon)</td>
</tr>
<tr>
<td>(q_{\text{accept}})</td>
<td>$</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>
Example Derivation using CFG:

- \( S \rightarrow aTb \) (using rule \( S \rightarrow aTb \))
- \( \Rightarrow aTab \) (using rule \( T \rightarrow Ta \))
- \( \Rightarrow aab \) (using rule \( T \rightarrow \varepsilon \))

If stack top is variable \( S \), pop \( S \) and push rule right-sides (in rev order)

PDA Example

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Stack</th>
<th>Equiv Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_{start} )</td>
<td>aab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_{loop} )</td>
<td>aab</td>
<td>$</td>
<td>( S \rightarrow aTb )</td>
</tr>
<tr>
<td>( q_{loop} )</td>
<td>aab</td>
<td>aTb$</td>
<td></td>
</tr>
<tr>
<td>( q_{loop} )</td>
<td>ab</td>
<td>Tb$</td>
<td></td>
</tr>
<tr>
<td>( q_{loop} )</td>
<td>ab</td>
<td>Tab$</td>
<td>( T \rightarrow Ta )</td>
</tr>
<tr>
<td>( q_{loop} )</td>
<td>ab</td>
<td>ab$</td>
<td>( T \rightarrow \varepsilon )</td>
</tr>
<tr>
<td>( q_{loop} )</td>
<td>b</td>
<td>b$</td>
<td></td>
</tr>
<tr>
<td>( q_{loop} )</td>
<td></td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>( q_{accept} )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Derivation using CFG:

- $S \Rightarrow aTb$ (using rule $S \rightarrow aTb$)
- $\Rightarrow aTab$ (using rule $T \rightarrow Ta$)
- $\Rightarrow aab$ (using rule $T \rightarrow \varepsilon$)

PDA Example

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Stack</th>
<th>Equiv Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_{start}$</td>
<td>aab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>aab</td>
<td>$S$</td>
<td>$S \rightarrow aTb$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>aab</td>
<td>a$Tb$</td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>ab</td>
<td>$7b$</td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>ab</td>
<td>$Tb$</td>
<td></td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>ab</td>
<td>$ab$</td>
<td>$T \rightarrow Ta$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td>b</td>
<td>$b$</td>
<td>$T \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$q_{loop}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_{accept}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If stack top is terminal, pop and read matching input.
Example CFG→PDA

Example Derivation using CFG:
\[ S \rightarrow aTb \] (using rule \( S \rightarrow aTb \))
\[ \Rightarrow aTab \] (using rule \( T \rightarrow Ta \))
\[ \Rightarrow aab \] (using rule \( T \rightarrow \varepsilon \))
A lang is a CFL iff some PDA recognizes it

⇒ If a language is a CFL, then a PDA recognizes it
  • Convert CFG⇒PDA

⇐ If a PDA recognizes a language, then it’s a CFL
  • To prove this part: show PDA has an equivalent CFG
PDA→CFG: Prelims

Before converting PDA to CFG, modify it so:

1. It has a single accept state, $q_{accept}$.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a push move) or pops one off the stack (a pop move), but it does not do both at the same time.

Important:
This doesn’t change the language recognized by the PDA
PDA $P$ -> CFG $G$ : Variables

\[ P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\}) \]

variables of $G$ are \( \{A_{pq} \mid p, q \in Q\} \)

- **Want:** if $P$ goes from state $p$ to $q$ reading input $x$, then some $A_{pq}$ generates $x$

- **So:** For every pair of states $p, q$ in $P$, add variable $A_{pq}$ to $G$

- **Then:** connect the variables together by,
  - Add rules: $A_{pq} \to A_{pr}A_{rq}$, for each state $r$
  - These rules allow grammar to simulate every possible transition
  - (We haven’t added input read/generated terminals yet)

- The Key IDEA

- To add terminals: pair up stack pushes and pops (essence of a CFL)
PDA $P \rightarrow$ CFG $G$: Generating Strings

$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$

variables of $G$ are $\{A_{pq} \mid p, q \in Q\}$

- The key: pair up stack pushes and pops (essence of a CFL)

if $\delta(p, a, \varepsilon)$ contains $(r, u)$ and $\delta(s, b, u)$ contains $(q, \varepsilon)$,

put the rule $A_{pq} \rightarrow aA_{rs}b$ in $G$
PDA $P \rightarrow$ CFG $G$: Generating Strings

$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$

variables of $G$ are $\{A_{pq} | p, q \in Q\}$

- The key: pair up stack pushes and pops (essence of a CFL)

if $\delta(p, a, \varepsilon)$ contains $(r, u)$ and $\delta(s, b, u)$ contains $(q, \varepsilon)$,

put the rule $A_{pq} \rightarrow aA_{rs}b$ in $G$
PDA $P \rightarrow$ CFG $G$: Generating Strings

$P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$

variables of $G$ are $\{A_{pq} | p, q \in Q\}$

- **The key**: pair up stack pushes and pops (essence of a CFL)

  if $\delta(p, a, \varepsilon)$ contains $(r, u)$ and $\delta(s, b, u)$ contains $(q, \varepsilon)$,

  put the rule $A_{pq} \rightarrow aA_{rs}b$ in $G$
A language is a CFL $\iff$ A PDA recognizes it

$\Rightarrow$ If a language is a CFL, then a PDA recognizes it
  • Convert CFG $\rightarrow$ PDA

$\Leftarrow$ If a PDA recognizes a language, then it’s a CFL
  • Convert PDA $\rightarrow$ CFG
Regular vs Context-Free Languages
(and others?)
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

   - **Proof**: CFG $S \rightarrow 0S1 \mid \varepsilon$
   - **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!

context-free languages (CFLs)

regular languages

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 **string**,  
2. **Then** parse 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→PDA

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

This 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

\[
S \rightarrow \textcolor{green}{aTb} \quad \textcolor{red}{b} \\
T \rightarrow Ta \mid \varepsilon
\]

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 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. \( \delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow (Q \times \Gamma \epsilon ) \cup \{ \emptyset \} \) 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. \( \delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow \mathcal{P}(Q \times \Gamma \epsilon) \)
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.

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

Must take into account \( \epsilon \) reads or stack ops!
E.g., if \( \delta(q, a, X) \) does “something”, then \( \delta(q, \epsilon, X) \) must “do nothing”.

A deterministic 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. \( \delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow (Q \times \Gamma \epsilon) \cup \{ \emptyset \} \) 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.

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

Must take into account \( \epsilon \) reads or stack ops!
E.g., if \( \delta(q, a, X) \) does “something”, then \( \delta(q, \epsilon, X) \) must “do nothing”.

A deterministic 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. \( \delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow (Q \times \Gamma \epsilon) \cup \{ \emptyset \} \) 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.

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

Must take into account \( \epsilon \) reads or stack ops!
E.g., if \( \delta(q, a, X) \) does “something”, then \( \delta(q, \epsilon, X) \) must “do nothing”.

A deterministic 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. \( \delta: Q \times \Sigma \epsilon \times \Gamma \epsilon \rightarrow (Q \times \Gamma \epsilon) \cup \{ \emptyset \} \) 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.

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

Must take into account \( \epsilon \) reads or stack ops!
E.g., if \( \delta(q, a, X) \) does “something”, then \( \delta(q, \epsilon, X) \) 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!)

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

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

When parsing this string, when does it know **which rule** was used, \( S \) or \( T \)?

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**
- **LL(k)**
- **LR(k)**
- **LL(1)**
- **LR(1)**
- **LALR(1)**
- **SLR**
- **LL(0)**
- **LR(0)**

Programming language parsers / compilers are ideally in here.

All CFLs
Compiler Stages

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

DFAs (recognizing regular languages) in here!

Lexer

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

DFAs (represented as regular expressions)!

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

DFAs (recognizing regular languages) in here!

Lexer

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

DPDAs (recognizing DCFLs) in here!

Parser

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

\[
\begin{align*}
\text{AssignStm} & \quad a \quad \text{OpExp} \\
\text{OpExp} & \quad \text{NumExp} \quad \text{Plus} \quad \text{NumExp} \\
\text{NumExp} & \quad 5 \\
\text{Plus} & \quad 3
\end{align*}
\]
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 !
```
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!

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

Should use \( S \) rule

\[ \text{aaa} \]
\[ \text{aaabbb} \rightarrow \text{aaSbb} \]

\[ \text{aaabbb} \rightarrow \text{aaaSbb} \]
\[ \text{aaabbbbbb} \rightarrow \text{aaTbbbbb} \]

Should use \( T \) rule

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

DCFLs
LL parsing

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

Game: “You’re the Parser”: Guess which rule applies?
(and how much did you have to “look ahead”?)

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. $E \rightarrow \text{num = num}$
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 \( S \rightarrow \) if \( E \) then \( S \) else \( S \)
2 \( S \rightarrow \) begin \( S \ L \)
3 \( S \rightarrow \) print \( E \)

4 \( L \rightarrow \) end
5 \( L \rightarrow ; \) \( S \ L \)
6 \( E \rightarrow \) 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 \( S \rightarrow \textbf{print } E \)

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

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

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

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

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

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 id := E
S \rightarrow print ( L )
E \rightarrow id
E \rightarrow num
E \rightarrow E + E
```

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

*Stack*

```
1
push

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

*Input*

```
<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift</td>
</tr>
<tr>
<td>= “push”</td>
</tr>
</tbody>
</table>
```

*State name*
LR parsing

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

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

<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 id4</td>
<td>:= 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>
</tbody>
</table>
LR parsing

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

\[ S \to S ; S \]
\[ S \to \text{id} := E \]
\[ E \to \text{id} \]
\[ E \to \text{num} \]
\[ E \to E + E \]

Stack

<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 id4</td>
<td>:= 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>; b := c + ( d := 5 + 6 , d ) $</td>
<td>reduce E → num</td>
</tr>
</tbody>
</table>
LR parsing

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

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

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

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

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

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

Stack

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

A program (string of chars) → Lexer (DFAs / NFAs) → Parser (DPDAs) → Abstract Syntax tree (AST) → ???

This phase needs computation that goes beyond CFLs