LR Parsing

Table Construction
Outline

• Review of bottom-up parsing

• Computing the parsing DFA
  - Closures, LR(1) Items, States
  - Transitions

• Using parser generators
  - Handling Conflicts
In One Slide

- An **LR(1) parsing table** can be constructed automatically from a CFG. An **LR(1) item** is a pair made up of a **production** and a **lookahead token**; it represents a possible parser **context**. After we **extend** LR(1) items by **closing** them they become LR(1) **DFA states**. Grammars can have **shift/reduce** or **reduce/reduce conflicts**. You can fix most conflicts with **precedence and associativity declarations**. **LALR(1) tables** are formed from LR(1) tables by **merging states** with similar **cores**.
Bottom-up Parsing (Review)

- A bottom-up parser rewrites the input string to the start symbol.
- The state of the parser is described as \( \alpha \rightarrow \gamma \).
  - \( \alpha \) is a stack of terminals and non-terminals.
  - \( \gamma \) is the string of terminals not yet examined.
- Initially: \( \rightarrow x_1 x_2 \ldots x_n \).
Shift and Reduce Actions (Review)

• Recall the CFG: $E \rightarrow \text{int} \mid E + (E)$

• A bottom-up parser uses two kinds of actions:

  • **Shift** pushes a terminal from input on the stack

    $E + (\uparrow \text{int}) \Rightarrow E + (\text{int} \uparrow)$

  • **Reduce** pops 0 or more symbols off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS)

    $E + (E + (E) \uparrow) \Rightarrow E +(E \uparrow)$
Key Issue: When to Shift or Reduce?

- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The input is the stack
  - The language consists of terminals and non-terminals

- We run the DFA on the stack and we examine the resulting state $X$ and the token $tok$ after $\rightarrow$
  - If $X$ has a transition labeled $tok$ then shift
  - If $X$ is labeled with “$A \rightarrow \beta$ on tok” then reduce
LR(1) Parsing. An Example

E → int

\[ E \rightarrow \text{int} \]

\[ E \rightarrow \text{int} + (\text{int}) + (\text{int}) \]

\[ \text{shift} \]

\[ \text{int} \rightarrow \text{int} + (\text{int}) + (\text{int}) \]

\[ \text{E} \rightarrow \text{int} \]

\[ E \rightarrow \text{int} + (\text{int}) + (\text{int}) \]

\[ \text{shift(x3)} \]

\[ E \rightarrow \text{int} + (\text{int}) + (\text{int}) \]

\[ \text{shift} \]

\[ E \rightarrow \text{int} + (\text{int}) + (\text{int}) \]

\[ \text{shift(x3)} \]

\[ E \rightarrow \text{E} + (\text{E}) \]

\[ \text{E} \rightarrow \text{E} + (\text{E}) \]

\[ \text{shift} \]

\[ E \rightarrow \text{E} + (\text{E}) \]

\[ \text{shift} \]

\[ E \rightarrow \text{E} + (\text{E}) \]

\[ \text{accept} \]
End of review
Key Issue: How is the DFA Constructed?

- The **stack** describes the **context** of the parse
  - What non-terminal we are looking for
  - What production rhs we are looking for
  - What we have seen so far from the rhs
Three hours later, you can finally parse $E \rightarrow E + E \mid \text{int}$.
Parsing Contexts

• Consider the state:

\[
\begin{align*}
E & \\
\text{int} + ( \text{int } ) + ( \text{int } ) & \\
\end{align*}
\]

- The stack is

\[
E + ( \text{int } ) + ( \text{int } )
\]

• **Context:**
  - We are looking for an \( E \rightarrow E + ( \bullet E ) \)
    - Have have seen \( E + ( \) from the right-hand side
  - We are also looking for \( E \rightarrow \bullet \text{int or } E \rightarrow \bullet E + ( E ) \)
    - Have seen nothing from the right-hand side

• **One DFA state must thus describe several contexts**
LR(1) Items

• An LR(1) item is a pair:
  \[ X \rightarrow \alpha \cdot \beta, \ a \]
  - \( X \rightarrow \alpha \beta \) is a production
  - \( a \) is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal

• \([X \rightarrow \alpha \cdot \beta, \ a]\) describes a context of the parser
  - We are trying to find an \( X \) followed by an \( a \), and
  - We have \( \alpha \) already on top of the stack
  - Thus we need to see next a prefix derived from \( \beta a \)
Note

- The symbol ▶ was used before to separate the stack from the rest of input
  - $\alpha ▶ \gamma$, where $\alpha$ is the stack and $\gamma$ is the remaining string of terminals
- In LR(1) items • is used to mark a prefix of a production rhs:
  \[ X \rightarrow \alpha\bullet\beta, a \]
  - Here $\beta$ might contain non-terminals as well
- In both case the stack is on the left
Convention

- We add to our grammar a fresh new start symbol $S$ and a production $S \rightarrow E$
  - Where $E$ is the old start symbol
  - No need to do this if $E$ had only one production

- The **initial parsing context** contains:
  $$ S \rightarrow \bullet E, \$ $$
  - Trying to find an $S$ as a string derived from $E\$$
  - The stack is empty
LR(1) Items (Cont.)

- In context containing
  \[ E \rightarrow E + \bullet \ ( E ) , + \]
  - If ( follows then we can perform a shift to context containing
  \[ E \rightarrow E + ( \bullet E ) , + \]

- In context containing
  \[ E \rightarrow E + \ ( E ) \bullet , + \]
  - We can perform a reduction with \[ E \rightarrow E + ( E ) \]
  - But only if a + follows
LR(1) Items (Cont.)

• Consider a context with the item

\[ E \rightarrow E + ( \bullet E ) , + \]

• We expect next a string derived from \( E ) + \)

• There are two productions for \( E \)

\[ E \rightarrow \text{int} \quad \text{and} \quad E \rightarrow E + ( E ) \]

• We describe this by extending the context with two more items:

\[ E \rightarrow \bullet \text{int}, ) \]

\[ E \rightarrow \bullet E + ( E ) , ) \]
The Closure Operation

• The operation of extending the context with items is called the closure operation

\[
\text{Closure(Items)} = \\
\text{repeat} \\
\quad \text{for each } [X \rightarrow \alpha \bullet Y\beta, a] \text{ in Items} \\
\quad \text{for each production } Y \rightarrow \gamma \\
\quad \text{for each } b \in \text{First}(\beta a) \\
\quad \text{add } [Y \rightarrow \bullet \gamma, b] \text{ to Items} \\
\text{until Items is unchanged}
\]
Constructing the Parsing DFA (1)

• Construct the start context:

$$\text{Closure}([S \rightarrow \bullet E, \$]) =$$

- $S \rightarrow \bullet E, \$
- $E \rightarrow \bullet E+(E), \$
- $E \rightarrow \bullet \text{int}, \$
- $E \rightarrow \bullet E+(E), +$
- $E \rightarrow \bullet \text{int}, +$

• We abbreviate as:

$$S \rightarrow \bullet E, \$
E \rightarrow \bullet E+(E), \$/+
E \rightarrow \bullet \text{int}, \$/+$$
PLANNING

You... have a plan, right?
Constructing the Parsing DFA (2)

• An LR(1) DFA state is a closed set of LR(1) items
  - This means that we performed Closure

• The start state contains \([S \rightarrow \bullet E, \$]\)

• A state that contains \([X \rightarrow \alpha \bullet, \ b]\) is labeled with “reduce with \(X \rightarrow \alpha\) on \(b\)”

• And now the transitions ...
The DFA Transitions

- A state “State” that contains $[X \rightarrow \alpha \cdot y\beta, \ b]$ has a transition labeled $y$ to a state that contains the items “Transition(State, y)”
  - $y$ can be a terminal or a non-terminal

Transition(State, y) =

$\text{Items} \leftarrow \emptyset$

for each $[X \rightarrow \alpha \cdot y\beta, \ b] \in \text{State}$

  add $[X \rightarrow \alpha y\cdot \beta, \ b]$ to $\text{Items}$

return Closure(Items)
LR(1) DFA Construction Example

S → • E, $
E → • E+(E), $/+ 
E → • int, $/+
LR(1) DFA Construction Example

S → E, $
E → E+(E), $/+ 
E → int, $/+
LR(1) DFA Construction Example

S → •E, $
E → •E+(E), $/+ 
E → •int, $/+ 

E → int•, $/+
LR(1) DFA Construction Example

\[
\begin{align*}
S &\rightarrow \cdot E, \$
E &\rightarrow \cdot E+(E), \$/+
E &\rightarrow \cdot \text{int}, \$/+
\end{align*}
\]
LR(1) DFA Construction Example

S → •E, $
E → •E+(E), $/+ 
E → •int, $/+ 

E → int•, $/+ 

E → int on $, + 

S → E•, $
E → E•+(E), $/+
LR(1) DFA Construction Example

S → \bullet E, $  
E → \bullet E+(E), $/+  
E → \bullet \text{int}, $/+  

S → E \bullet, $  
E → E \bullet+(E), $/+  

E → \text{int} \bullet, $/+  
E → \text{int} on $, +  

accept on $
LR(1) DFA Construction Example

S → •E, $
E → •E+(E), $/+
E → •int, $/+  \hspace{2cm} 0
S → E•, $
E → E•+(E), $/+  \hspace{2cm} 2
E → int•, $/+ \hspace{2cm} 1
E → E+• (E), $/+ \hspace{2cm} 3

int
+
accept

on $

E → int

on $, +
LR(1) DFA Construction Example

S → E, $
E → E+(E), $/+ 
E → int, $/+ 

E → int, $/+ 
S → E•, $
E → E+(E), $/+ 

E → E+(E), $/+ 

E → int, $/+ 

accept on $
LR(1) DFA Construction Example

S → •E, $
E → •E+(E), $/+ 0
E → •int, $/+ int

E → int•, $/+ 1
E → E+• (E), $/+ +
E → int, $/+ ( 3

S → E•, $
E → E•+(E), $/+ 2

accept on $

E → E+(•E), $/+ 4
E → •E+(E), )/+
LR(1) DFA Construction Example

\[ S \rightarrow \mathit{E}, \mathit{$} \]
\[ E \rightarrow \mathit{E}+(\mathit{E}), \mathit{$}/+ \]
\[ E \rightarrow \mathit{\mathit{int}}, \mathit{$}/+ \]

\[ S \rightarrow \mathit{E}, \mathit{$} \]
\[ E \rightarrow \mathit{E}+(\mathit{E}), \mathit{$}/+ \]

**0**
\[ E \rightarrow \mathit{\mathit{int}}, \mathit{$}/+ \]
\[ E \rightarrow \mathit{\mathit{E}+(\mathit{E})}, \mathit{$}/+ \]

**1**
\[ E \rightarrow \mathit{\mathit{E}+(\mathit{E})}, \mathit{$}/+ \]

**2**
\[ S \rightarrow \mathit{\mathit{E}}, \mathit{$} \]
\[ E \rightarrow \mathit{\mathit{E}+(\mathit{E})}, \mathit{$}/+ \]

**3**
\[ E \rightarrow \mathit{\mathit{E}+(\mathit{E})}, \mathit{$}/+ \]

**4**
\[ E \rightarrow \mathit{\mathit{E}+(\mathit{E})}, \mathit{)}+/ \]
\[ E \rightarrow \mathit{\mathit{\mathit{int}, }}, \mathit{)}+/ \]

accept on \$
LR(1) DFA Construction Example

S → •E, $  
E → •E+(E), $/+  
E → •int, $/+  

S → E•, $  
E → E•+(E), $/+  

E → int•, $/+  
E → int, $/+  

E → E+(E), $/+  
E → E+(•E), $/+  
E → •int, )/+  

E → E+(E), )/+  
E → •int, )/+  

E → int•, )/+  
E → int, $/+  

accept on $  
E → int on $, +  

2  
0  
1  
3  
4  
5  

states  
production rules  
transitions  
accept states  

LR(1) DFA Construction Example

S → •E, $
E → •E+(E), $/+ 
E → •int, $/+ 

S → E•, $
E → E•+(E), $/+ 

E → •int•, $/+ 
E → •E+(E), $/+ 
E → •int, $/+ 

E → •E+(E), $/+ 
E → •E+(E), )/+ 
E → •int, )/+ 

E → int•, )/+ 
E → int on ), +
LR(1) DFA Construction Example

S → •E, $
E → •E+(E), $/+ 
E → •int, $/+ 

S → E•, $
E → E•+(E), $/+ 

E → E•+$/+ 
E → •int+$/+ 

E → E•+(E), $/+ 
E → •int+$/+ 

E → E•+(E), $/+ 
E → •int+$/+ 

E → E•+(E), $/+ 
E → •int+$/+ 

E → •int+$/+ 
E → int+$/+ 

and so on...
This post-apocalyptic 1984 animated film by Studio Ghibli features a peace-loving, wind-riding princess who attempts to understand the apparently-evil insects and spreading fungi of her world while averting a war.
• The 1995 comedy film Clueless starring Alicia Silverstone was based on this Jane Austen novel.
Q: Books (736 / 842)

• Give the last word in 2 of the following 4 young adult book titles:
  - Beverly Cleary Ramona Quimby, Age
  - Judy Blume's Tales of a Fourth Grade
  - Lynne Reid Banks's The Indian in the
  - Lloyd Alexander's The High
In this 1982 arcade game features lance-wielding knights mounted on giant flying birds and dueling over a pit of lava. Destroying an enemy knight required ramming it such that your lance was higher than the enemy's.
LR Parsing Tables. Notes

- Parsing tables (= the DFA) can be constructed **automatically** for a CFG
  - “The tables which cannot be constructed are constructed automatically in response to a CFG input. You asked for a miracle, Theo. I give you the L-R-1.” - Hans Gruber, *Die Hard*

- But we still need to understand the construction to work with parser generators
  - e.g., they report errors in terms of sets of items

- What kind of errors can we expect?
PARTY CONFLICT

Sometimes, you should back down.
Shift/Reduce Conflicts

• If a DFA state contains both
  \[ X \rightarrow \alpha a \beta, b \] and \[ Y \rightarrow \gamma \dot{\,}, a \]

• Then on input “a” we could either
  - Shift into state \[ X \rightarrow \alpha a \beta, b \], or
  - Reduce with \[ Y \rightarrow \gamma \]

• This is called a **shift-reduce conflict**
Shift/Reduce Conflicts

- Typically due to *ambiguities in the grammar*
- Classic example: the dangling else
  
  \[ S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER} \]

- Will have DFA state containing
  
  \[ [S \rightarrow \text{if } E \text{ then } S\bullet, \text{ else}] \]
  
  \[ [S \rightarrow \text{if } E \text{ then } S\bullet \text{ else } S, \text{ x}] \]

- If *else* follows then we can shift or reduce

- Default (bison, CUP, etc.) is to shift
  
  - Default behavior is as needed in this case
More Shift/Reduce Conflicts

• Consider the ambiguous grammar

\[ E \rightarrow E + E \mid E \times E \mid \text{int} \]

• We will have the states containing

\[ [E \rightarrow E \times \bullet E, +] \quad [E \rightarrow E \times E \bullet, +] \]
\[ [E \rightarrow \bullet E + E, +] \Rightarrow^E [E \rightarrow E \bullet + E, +] \]
\[ \ldots \quad \ldots \]

• Again we have a shift/reduce on input +
  - We need to reduce (* binds more tightly than +)
  - Solution: declare the precedence of * and +
More Shift/Reduce Conflicts

- In bison declare **precedence** and **associativity**:

  ```
  %left +
  %left * // high precedence
  ```

- **Precedence** of a rule = that of its last terminal
  - See bison manual for ways to override this default

- Resolve shift/reduce conflict with a **shift** if:
  - no precedence declared for either rule or terminal
  - input terminal has higher precedence than the rule
  - the precedences are the same and right associative
Using Precedence to Solve S/R Conflicts

• Back to our example:

\[
\begin{align*}
[E \rightarrow E \cdot E, +] & \quad [E \rightarrow E \cdot E \cdot, +] \\
[E \rightarrow E \cdot E + E, +] & \Rightarrow^E [E \rightarrow E \cdot + E, +] \\
\cdots & \quad \cdots
\end{align*}
\]

• Will choose **reduce** on input + because precedence of rule \(E \rightarrow E \cdot E\) is higher than of terminal +
Using Precedence to Solve S/R Conflicts

• Same grammar as before

\[ E \rightarrow E + E \mid E \ast E \mid \text{int} \]

• We will also have the states

\[
\begin{align*}
[E \rightarrow E + \bullet E, +] & \quad [E \rightarrow E + E \bullet, +] \\
[E \rightarrow \bullet E + E, +] & \Rightarrow^E [E \rightarrow E \bullet + E, +] \\
\vdots & \quad \vdots
\end{align*}
\]

• Now we also have a shift/reduce on input +
  - We choose reduce because \( E \rightarrow E + E \) and + have the same precedence and + is left-associative
Using Precedence to Solve S/R Conflicts

• Back to our dangling else example
  
  \[ S \rightarrow \text{if } E \text{ then } S, \text{ else} \]
  
  \[ S \rightarrow \text{if } E \text{ then } S\bullet \text{ else } S, \ x \]

• Can eliminate conflict by declaring \textit{else} with higher precedence than \textit{then}
  
  - Or just rely on the default shift action

• But this starts to look like “hacking the parser”

• Avoid overuse of precedence declarations or you’ll end with unexpected parse trees
  
  - The kiss of death ...
Reduce/Reduce Conflicts

- If a DFA state contains both
  \[ X \to \alpha\bullet, a \] and \[ Y \to \beta\bullet, a \]
  - Then on input “a” we don’t know which production to reduce

- This is called a **reduce/reduce conflict**
Reduce/Reduce Conflicts

- Usually due to **gross ambiguity** in the grammar
- Example: a sequence of identifiers
  \[ S \to \varepsilon \mid id \mid id \, S \]

- There are **two parse trees** for the string **id**
  \[ S \to id \]
  \[ S \to id \, S \to id \]

- How does this confuse the parser?
More on Reduce/Reduce Conflicts

- Consider the states

\[
\begin{align*}
[S' \rightarrow \cdot S, \ $] & \quad [S \rightarrow \cdot S, \ $] \\
[S \rightarrow \cdot, \ $] & \quad \Rightarrow_{\text{id}} \quad [S \rightarrow \cdot, \ $] \\
[S \rightarrow \cdot \ \text{id}, \ $] & \quad [S \rightarrow \cdot \ \text{id}, \ $] \\
[S \rightarrow \cdot \ \text{id} \ S, \ $] & \quad [S \rightarrow \cdot \ \text{id} \ S, \ $]
\end{align*}
\]

- Reduce/reduce conflict on input $\$

\[
\begin{align*}
S' \rightarrow S \rightarrow \text{id} \\
S' \rightarrow S \rightarrow \text{id} \ S \rightarrow \text{id}
\end{align*}
\]

- Better rewrite the grammar: $S \rightarrow \varepsilon \mid \text{id} \ S$
Can’s someone learn this for me?

No, you can't have a neural network.
Using Parser Generators

- **Parser generators** construct the parsing DFA given a CFG
  - Use precedence declarations and default conventions to resolve conflicts
  - The parser algorithm is the same for all grammars (and is provided as a library function)

- But most parser generators do not construct the DFA as described before
  - Why might that be?
Using Parser Generators

• **Parser generators** construct the parsing DFA given a CFG
  - Use precedence declarations and default conventions to *resolve conflicts*
  - The **parser algorithm is the same** for all grammars (and is provided as a library function)

• But most parser generators do not construct the DFA as described before
  - Because the LR(1) parsing DFA has 1000s of states even for a simple language
LR(1) Parsing Tables are Big

- But many states are similar, e.g.

\[
\begin{align*}
E &\rightarrow \text{int}\bullet, \$, + & E &\rightarrow \text{int} & 1 \\
E &\rightarrow \text{int} & \text{on } \$, + & & E &\rightarrow \text{int} & 5 \\
E &\rightarrow \text{int}\bullet, )/+ & \text{and} & & E &\rightarrow \text{int} & \text{on } ), +
\end{align*}
\]

- Idea: **merge** the DFA states whose items differ only in the lookahead tokens
  - We say that such states have the same **core**

- We obtain

\[
\begin{align*}
E &\rightarrow \text{int}\bullet, \$, + & E &\rightarrow \text{int} & 1'
\end{align*}
\]
The Core of a Set of LR Items

• Definition: The core of a set of LR items is the set of first components
  - Without the lookahead terminals

• Example: the core of

\[
\{ [X \rightarrow \alpha\cdot\beta, b], [Y \rightarrow \gamma\cdot\delta, d]\}
\]

is

\[
\{X \rightarrow \alpha\cdot\beta, Y \rightarrow \gamma\cdot\delta\}
\]
LALR States

• Consider for example the LR(1) states
  \[
  \begin{align*}
  &\{[X \rightarrow \alpha\cdot, a], [Y \rightarrow \beta\cdot, c]\} \\
  &\{[X \rightarrow \alpha\cdot, b], [Y \rightarrow \beta\cdot, d]\}
  \end{align*}
  \]

• They have the **same core** and can be merged

• And the merged state contains:
  \[
  \{[X \rightarrow \alpha\cdot, a/b], [Y \rightarrow \beta\cdot, c/d]\}
  \]

• These are called **LALR(1)** states
  - Stands for **LookAhead LR**
  - Typically 10x fewer LALR(1) states than LR(1)
LALR(1) DFA

• **Repeat** until all states have distinct core
  - Choose two distinct states with same core
  - Merge the states by creating a new one with the union of all the items
  - Point edges from predecessors to new state
  - New state points to all the previous successors
Example LALR(1) to LR(1)
The LALR Parser Can Have Conflicts

• Consider for example the LR(1) states
  \[
  \{[X \rightarrow \alpha \cdot, a], [Y \rightarrow \beta \cdot, b]\}
  \{[X \rightarrow \alpha \cdot, b], [Y \rightarrow \beta \cdot, a]\}
  \]

• And the merged LALR(1) state
  \[
  \{[X \rightarrow \alpha \cdot, a/b], [Y \rightarrow \beta \cdot, a/b]\}
  \]

• Has a new reduce-reduce conflict

• In practice such cases are rare
LALR vs. LR Parsing

• LALR languages are not natural
  - They are an efficiency hack on LR languages

• Any “reasonable” programming language has a LALR(1) grammar
  - Java and C++ are presumed unreasonable …

• LALR(1) has become a standard for programming languages and for parser generators
A Hierarchy of Grammar Classes

From Andrew Appel, "Modern Compiler Implementation in Java"
Notes on Parsing

• Parsing
  - A solid foundation: context-free grammars
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficiency hack: LALR(1)
  - LALR(1) parser generators

• Now we move on to semantic analysis
Take a bow, you survived!
Supplement to LR Parsing

Strange Reduce/Reduce Conflicts Due to LALR Conversion
(from the bison manual)
Strange Reduce/Reduce Conflicts

- Consider the grammar

\[
\begin{align*}
S & \rightarrow P \ R \ , \\
NL & \rightarrow N \ | \ N \ , \ NL \\
P & \rightarrow T \ | \ NL : T \\
R & \rightarrow T \ | \ N : T \\
N & \rightarrow \text{id} \\
T & \rightarrow \text{id}
\end{align*}
\]

- **P** - parameters specification
- **R** - result specification
- **N** - a parameter or result name
- **T** - a type name
- **NL** - a list of names
Strange Reduce/Reduce Conflicts

• In $P$ an id is a
  - $N$ when followed by $,$ or $:$
  - $T$ when followed by id

• In $R$ an id is a
  - $N$ when followed by $:$
  - $T$ when followed by $,$

• This is an LR(1) grammar.

• But it is not LALR(1). Why?
  - For obscure reasons
A Few LR(1) States

P → • T id
P → • NL : T id
NL → • N :
NL → • N, NL :
N → • id :
N → • id ,
T → • id id

R → • T ,
R → • N : T ,
T → • id ,
N → • id :

T → id • id
N → id • :
N → id • ,

T → id • id /
N → id • :/

LALR reduce/reduce conflict on " , "
LALR merge

#67
What Happened?

- Two distinct states were confused because they have the same core
- Fix: add dummy productions to distinguish the two confused states
- E.g., add

  \[
  R \rightarrow \text{id bogus}
  \]

  - \text{bogus} is a terminal not used by the lexer
  - This production will never be used during parsing
  - But it distinguishes \text{R} from \text{P}
A Few LR(1) States After Fix

Different cores $\Rightarrow$ no LALR merging
Homework

- Today: WA2 Was Due
- Thursday: Chapter 3.1 - 3.6
  - Optional Wikipedia Article
- Tuesday Sep 29 - Midterm 1 in Class
- Wednesday: PA3 due
  - Parsing!
- Thursday: WA3 due