LR Parsing

Table Construction

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.

Outline

- Review of bottom-up parsing
- Computing the parsing DFA
  - Closures, LR(1) Items, States
  - Transitions
- Using parser generators
  - Handling Conflicts

Bottom-up Parsing (Review)

- A bottom-up parser rewrites the input string to the start symbol
- The state of the parser is described as

α ► γ

  - α is a **stack** of terminals and non-terminals
  - γ is the string of terminals not yet examined

- Initially: ► $x_1x_2 \ldots x_n$

Shift and Reduce Actions (Review)

- Recall the CFG: **E → int | E + (E)**
- A bottom-up parser uses two kinds of actions:

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

E + (► int) ⇒ E + (int ►)

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

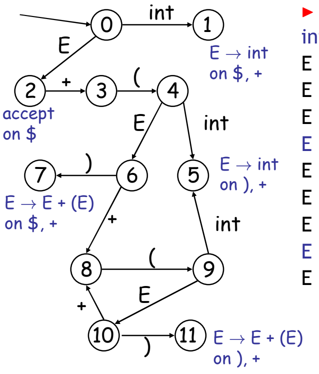
E + ($\underline{\text{E + (E)}}$ ►) ⇒ E +($\underline{\text{E}}$ ►)

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 ►
  - If X has a transition labeled **tok** then **shift**
  - If X is labeled with "**A → β on tok**" then **reduce**

LR(1) Parsing. An Example

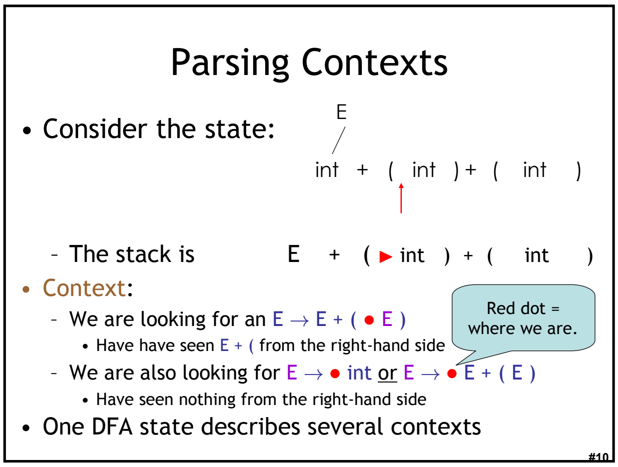


► int + (int) + (int)$	shift
int ► + (int) + (int)$	E → int
E ► + (int) + (int)$	shift(x3)
E + (int ►) + (int)$	E → int
E + (E ►) + (int)$	shift
E + (E) ► + (int)$	E → E+(E)
E ► + (int)$	shift (x3)
E + (int ►)$	E → int
E + (E ►)$	shift
E + (E) ► $	E → E+(E)
E ► $	accept

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



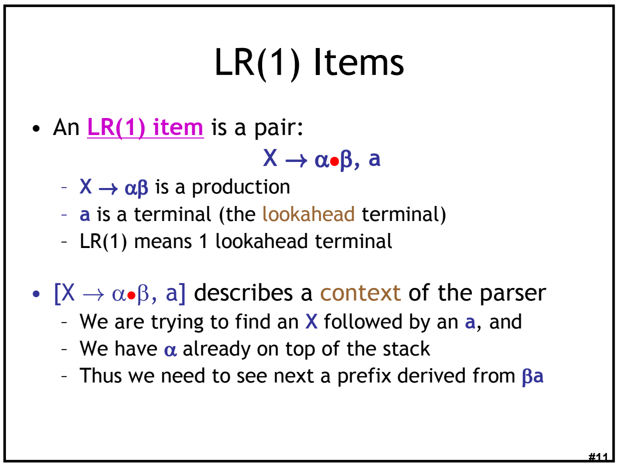
Parsing Contexts

- Consider the state:

$$\overset{E}{\text{int}} \; + \; (\; \overset{\uparrow}{\text{int}} \;) \; + \; (\; \text{int} \;)$$

  - The stack is E + (▶ int) + (int)
- Context:
  - We are looking for an E → E + (• E)
    - Have have seen E + (from the right-hand side
  - We are also looking for E → • int or E → • E + (E)
    - Have seen nothing from the right-hand side
- One DFA state describes several contexts

Red dot = where we are.

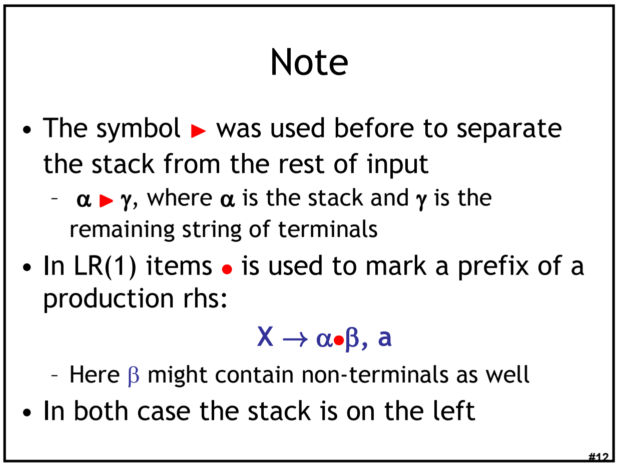


LR(1) Items

- An **LR(1) item** is a pair:

$$X \rightarrow \alpha\bullet\beta,\ a$$

  - $X \rightarrow \alpha\beta$ is a production
  - a is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal
- $[X \rightarrow \alpha\bullet\beta,\ a]$ describes a context of the parser
  - We are trying to find an X followed by an a, and
  - We have α already on top of the stack
  - Thus we need to see next a prefix derived from βa



Note

- The symbol ▶ was used before to separate the stack from the rest of input
  - α ▶ γ, where α is the stack and γ 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 β 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 → 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 → 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 → int and E → 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

```
Closure(Items) =
  repeat
    for each $[X \to \alpha \bullet Y\beta, a]$ in Items
      for each production $Y \to \gamma$
        for each $b \in First(\beta a)$
          add $[Y \to \bullet\gamma, b]$ to Items
  until Items is unchanged
```

Constructing the Parsing DFA (1)

- Construct the start context:

Closure({$S \to \bullet E$, \$}) =

$S \to \bullet E$, \$
$E \to \bullet E+(E)$, \$
$E \to \bullet int$, \$
$E \to \bullet E+(E)$, +
$E \to \bullet int$, +

  - We abbreviate as:

$S \to \bullet E$, \$
$E \to \bullet E+(E)$, \$/+
$E \to \bullet int$, \$/+

Constructing the Parsing DFA (2)

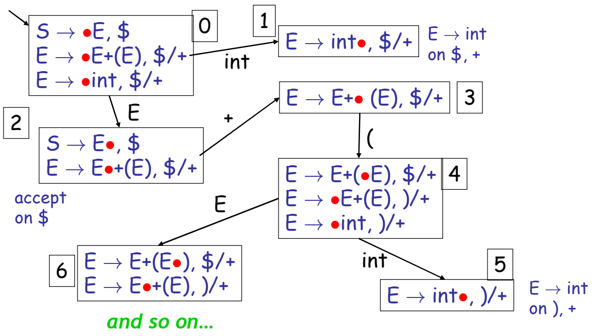
- An **LR(1) DFA state** is a closed set of LR(1) items
  - This means that we performed Closure
- The start state contains **[$S \to \bullet E$, \$]**
- A state that contains [$X \to \alpha\bullet$, b] is labeled with "**reduce with $X \to \alpha$ on b**"
- And now the transitions …

The DFA Transitions

- A state "**State**" that contains **[X → α•yβ, 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) =
  Items ← ∅
  for each [X → α•yβ, b] ∈ State
      add [X → αy•β, b] to Items
  return Closure(Items)
```

LR(1) DFA Construction Example



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

1
E → int•, $/+
E → int
on $, +

2
S → E•, $
E → E•+(E), $/+
accept
on $

3
E → E+• (E), $/+

4
E → E+(•E), $/+
E → •E+(E),)/+
E → •int,)/+

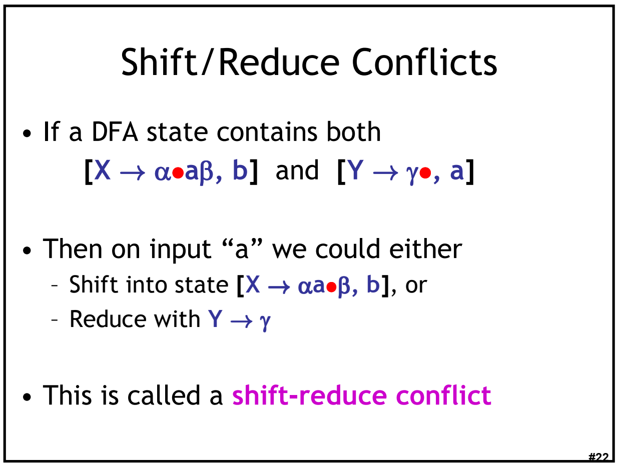
5
E → int•,)/+
E → int
on), +

6
E → E+(E•), $/+
E → E•+(E),)/+

and so on…

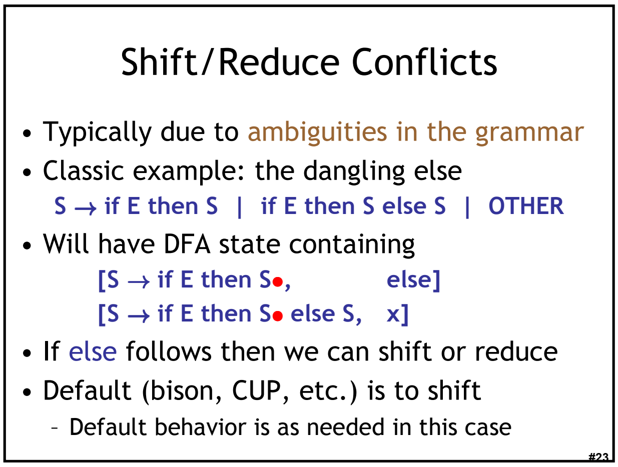
LR Parsing Tables. Notes

- 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?



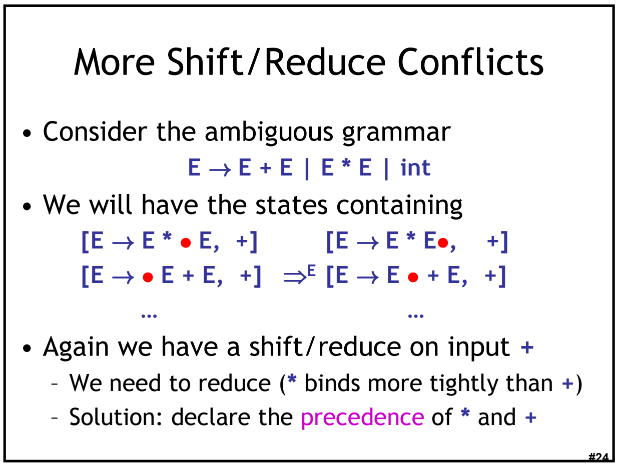
Shift/Reduce Conflicts

- If a DFA state contains both
  $[X \to \alpha \bullet a\beta,\ b]$ and $[Y \to \gamma \bullet,\ a]$
- Then on input “a” we could either
  - Shift into state $[X \to \alpha a \bullet \beta,\ b]$, or
  - Reduce with $Y \to \gamma$
- This is called a **shift-reduce conflict**



Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else
  $S \to \text{if } E \text{ then } S \ \mid\ \text{if } E \text{ then } S \text{ else } S \ \mid\ \text{OTHER}$
- Will have DFA state containing
  $[S \to \text{if } E \text{ then } S \bullet,\quad \text{else}]$
  $[S \to \text{if } E \text{ then } S \bullet \text{ else } S,\quad x]$
- 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
  $E \to E + E \mid E * E \mid \text{int}$
- We will have the states containing
  $[E \to E * \bullet E,\ +] \qquad [E \to E * E \bullet,\ +]$
  $[E \to \bullet E + E,\ +] \ \Rightarrow^{E}\ [E \to E \bullet + E,\ +]$
  … …
- 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:

[E → E * • E, +] [E → E * E•, +]

[E → • E + E, +] $\Rightarrow^{E}$ [E → E • + E, +]

… …

- Will choose **reduce** on input + because precedence of rule E → E * E is higher than of terminal +

Using Precedence to Solve S/R Conflicts

- Same grammar as before

E → E + E | E * E | int

- We will also have the states

[E → E + • E, +] [E → E + E•, +]

[E → • E + E, +] $\Rightarrow^{E}$ [E → E • + E, +]

… …

- Now we also have a shift/reduce on input +
  - We choose **reduce** because E → 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 → if E then S•, else]
  [S → if E then S• else S, x]
- Can eliminate conflict by declaring **else** with higher precedence than **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 → α•, a] and **[Y → β•, 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 → ε | id | id S

- There are **two parse trees** for the string id
  S → id
  S → id S → id
- How does this confuse the parser?

More on Reduce/Reduce Conflicts

- Consider the states $[S \to id \bullet, \$]$

$$
\begin{array}{lcl}
[S' \to \bullet S, \$] & & [S \to id \bullet S, \$] \\
[S \to \bullet, \$] & \Rightarrow^{id} & [S \to \bullet, \$] \\
[S \to \bullet id, \$] & & [S \to \bullet id, \$] \\
[S \to \bullet id\ S, \$] & & [S \to \bullet id\ S, \$]
\end{array}
$$

- Reduce/reduce conflict on input $

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.

1
$\boxed{E \to \text{int}\bullet,\ \$/+}$ E → int on $, +

and

5
$\boxed{E \to \text{int}\bullet,\ )/+}$ E → int on), +

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

1'
$\boxed{E \to \text{int}\bullet,\ \$/+/)}$ E → int on $, +,)

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

$$\{\ [X \to \alpha\bullet\beta,\ b],\ [Y \to \gamma\bullet\delta,\ d]\}$$

is

$$\{X \to \alpha\bullet\beta,\ Y \to \gamma\bullet\delta\}$$

LALR States

- Consider for example the LR(1) states

$$\{[X \to \alpha\bullet,\ a],\ [Y \to \beta\bullet,\ c]\}$$
$$\{[X \to \alpha\bullet,\ b],\ [Y \to \beta\bullet,\ d]\}$$

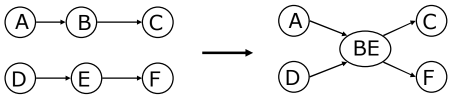
- They have the same core and can be merged
- And the merged state contains:

$$\{[X \to \alpha\bullet,\ a/b],\ [Y \to \beta\bullet,\ c/d]\}$$

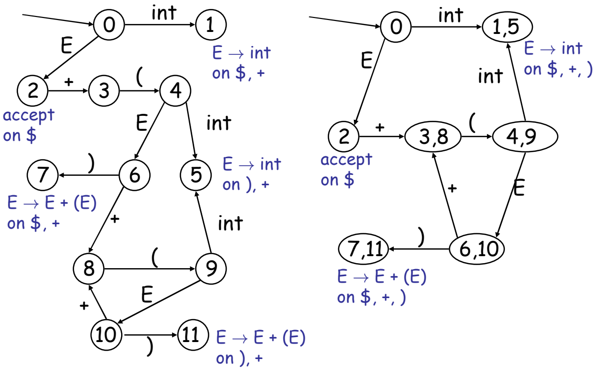
- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10x fewer LALR(1) states than LR(1)

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

  $\{[X \to \alpha\bullet, a], [Y \to \beta\bullet, b]\}$

  $\{[X \to \alpha\bullet, b], [Y \to \beta\bullet, a]\}$
- And the merged LALR(1) state

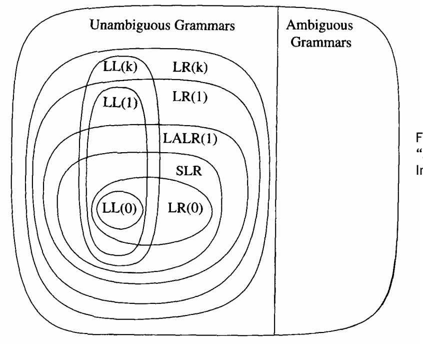
  $\{[X \to \alpha\bullet, a/b], [Y \to \beta\bullet, a/b]\}$
- Has a new reduce-reduce conflict

- 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
- 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

Supplement to LR Parsing

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

Strange Reduce/Reduce Conflicts

- Consider the grammar

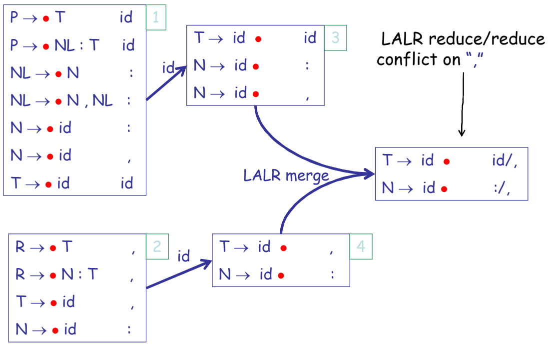
  S → P R ,   NL → N | N , NL
  P → T | NL : T   R → T | N : T
  N → id   T → id

- 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



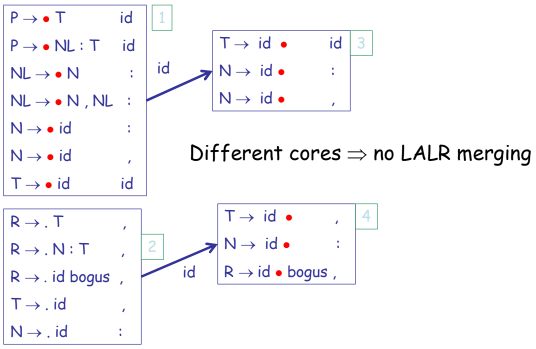
What Happened?

- 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 → id bogus

  - bogus is a terminal not used by the lexer
  - This production will never be used during parsing
  - But it distinguishes R from P

A Few LR(1) States After Fix



Homework

- Today: WA2 Due
- Tuesday: Chapter 3.1 – 3.6
  - Optional Wikipedia Article
- Next Friday: PA3 due
  - Parsing!
- **Tuesday Feb 27 - Midterm 1 in Class**