Example CFG

1. $S' \rightarrow \vdash S \vdash$
2. $S \rightarrow AyB$
3. $A \rightarrow ab$
4. $A \rightarrow cd$
5. $B \rightarrow z$
6. $B \rightarrow wz$
Recall that a stack in LL/top-down parsing is used in the following way:

\[ \text{input processed} + \text{stack} = \text{current derivation} \]

(Note that the stack here is read from the top to bottom)

For LR/bottom-up parsing, we have

\[ \text{stack} + \text{input to be read} = \text{current derivation} \]

(stack is read from bottom to top here)
### A trace

<table>
<thead>
<tr>
<th>Derivation</th>
<th>Stack</th>
<th>Input read</th>
<th>Unread Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ abywz ⊥</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>⊢ abywz ⊥</td>
<td>Shift ⊥</td>
</tr>
<tr>
<td>⊢ abywz ⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊢ abywz ⊥</td>
<td>Shift a</td>
</tr>
<tr>
<td>⊢ abywz ⊥</td>
<td>⊥ a</td>
<td>⊥ a</td>
<td>bywz ⊥</td>
<td>Shift b</td>
</tr>
<tr>
<td>⊢ abywz ⊥</td>
<td>⊥ a b</td>
<td>⊥ ab</td>
<td>ywz ⊥</td>
<td>Reduce A $\rightarrow$ ab</td>
</tr>
<tr>
<td>⊢ Aywz ⊥</td>
<td>⊥ A</td>
<td>⊥ ab</td>
<td>ywz ⊥</td>
<td>Shift y</td>
</tr>
<tr>
<td>⊢ Aywz ⊥</td>
<td>⊥ A y</td>
<td>⊥ aby</td>
<td>wz ⊥</td>
<td>Shift w</td>
</tr>
<tr>
<td>⊢ Aywz ⊥</td>
<td>⊥ A y w</td>
<td>⊥ abyw</td>
<td>z ⊥</td>
<td>Shift z</td>
</tr>
<tr>
<td>⊢ Aywz ⊥</td>
<td>⊥ A y w z</td>
<td>⊥ abywz</td>
<td>⊥</td>
<td>Reduce B $\rightarrow$ w z</td>
</tr>
<tr>
<td>⊢ AyB ⊥</td>
<td>⊥ A y B</td>
<td>⊥ abywz</td>
<td>⊥</td>
<td>Reduce S $\rightarrow$ AyB</td>
</tr>
<tr>
<td>⊢ S ⊥</td>
<td>⊥ S</td>
<td>⊥ abywz</td>
<td>⊥</td>
<td>Shift ⊥</td>
</tr>
<tr>
<td>⊢ S ⊥</td>
<td>⊥ S ⊥</td>
<td>⊥ abywz ⊥</td>
<td>$\epsilon$</td>
<td>Accept</td>
</tr>
</tbody>
</table>

Shift: shifting a token from one place to another (push)

Reduce: size of the stack may be reduced (pop RHS, push LHS)
Somehow, we shifted at just the right time, and reduced just at the right time.

How did we know this?

- Recall that for LL(1) parsing, we had a predictor table.
- For LR(1) parsing, we have an oracle, in the form of a DFA.
An LR Oracle for the previous CFG
Constructing DFA oracle for LR(1) grammars

- This is difficult to do
  - Donald Knuth proved a theorem that we can construct a DFA (really, a transducer) for LR(1) grammars (1965)
  - This transducer tells us when to shift or reduce.
  - We will use the transducer (primarily) and build the DFA only for simple grammars which satisfy further restrictions of being LR(0) or SLR(1) (secondarily)
LR(0) $\epsilon$-NFA formal definition

Given a CFG $G = (N, T, P, S)$, construct an $\epsilon$-NFA $(Q, N \cup T, q_0, F, D)$ as follows:

- $Q = \{A \rightarrow \alpha \bullet \beta | A \rightarrow \alpha \beta \in P\}$
- $q_0 = \{S' \rightarrow \vdash \bullet S \vdash\}$
- $D[A \rightarrow \alpha \bullet X \beta, X] = \{A \rightarrow \alpha X \bullet \beta\}$
  - shift depending on whether $X$ in $T$ or $N$
- $D[A \rightarrow \alpha \bullet B \beta, \epsilon] = \{B \rightarrow \bullet \gamma | B \rightarrow \gamma \in P\}$
- $F = \{S' \rightarrow \vdash S \bullet \vdash\}$

The LR(0) DFA is the subset construction of this NFA, which recognizes a valid stack.

The LR parser has 2 actions:

- If you have stack $K$ and input $a$, and $Ka$ is recognized, you can shift.
- If you have stack $K$ and input $a$, and the top of $K$ is a state containing $A \rightarrow \alpha \bullet$ and $a$ can follow $A$, reduce $A \rightarrow \alpha \bullet$
- If more than one of these is defined, you have a conflict.
Building an LR(0) automaton

Definition: An item is a production with a dot (\(\bullet\)) somewhere on the RHS (which indicates a partially completed rule)

How to construct the automaton:

- make the start state the first rule, with the dot (\(\bullet\)) in front of the left-most symbol of the RHS
- for each state, label an arc with the symbol that follows \(\bullet\) and advance the \(\bullet\) one position to the right in the next state.
- If the \(\bullet\) precedes a non-terminal (e.g., \(A\)) add all productions with that non-terminal \(A\) on the LHS to the current state, with the \(\bullet\) in the leftmost position
A sample construction of the DFA

Small example CFG:

1. $S' \rightarrow \epsilon \ E \ \epsilon$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow \text{id}$
For each input token

Start in the start state

Read the stack (from the bottom up) and read the current input, and do the action indicated for the current input

If there is a transition out of our current state on the current input, then *shift* (push) that input onto the stack

We know we can *reduce* if the current state has only one item and the • is the rightmost symbol

To *reduce*, pop the RHS off the stack, reread the stack (from the bottom-up), follow the transition for the LHS and push the LHS onto the stack

Accept if $S'$ on the stack when all input is read
Using the transducer

Example input: $id + id + id$

<table>
<thead>
<tr>
<th>Stack</th>
<th>States visited</th>
<th>Input read</th>
<th>Unread Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>1</td>
<td>$\epsilon$</td>
<td>$id + id + id$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash$</td>
<td>1 2</td>
<td>$\vdash$</td>
<td>$id + id + id$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash id$</td>
<td>1 2 6</td>
<td>$\vdash id$</td>
<td>$+ id + id$</td>
<td>reduce $T \rightarrow id$</td>
</tr>
<tr>
<td>$\vdash T$</td>
<td>1 2 5</td>
<td>$\vdash id$</td>
<td>$+ id + id$</td>
<td>reduce $E \rightarrow T$</td>
</tr>
<tr>
<td>$\vdash E$</td>
<td>1 2 3</td>
<td>$\vdash id$</td>
<td>$+ id + id$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash E +$</td>
<td>1 2 3 7</td>
<td>$\vdash id +$</td>
<td>$id + id$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash E + id$</td>
<td>1 2 3 7 6</td>
<td>$\vdash id + id$</td>
<td>$+ id$</td>
<td>reduce $T \rightarrow id$</td>
</tr>
<tr>
<td>$\vdash E + T$</td>
<td>1 2 3 7 8</td>
<td>$\vdash id + id$</td>
<td>$+ id$</td>
<td>reduce $E \rightarrow E + T$</td>
</tr>
<tr>
<td>$\vdash E$</td>
<td>1 2 3 8</td>
<td>$\vdash id + id$</td>
<td>$+ id$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash E +$</td>
<td>1 2 3 7</td>
<td>$\vdash id + id +$</td>
<td>$id$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash E + id$</td>
<td>1 2 3 7 6</td>
<td>$\vdash id + id + id$</td>
<td>$\vdash$</td>
<td>reduce $T \rightarrow id$</td>
</tr>
<tr>
<td>$\vdash E + T$</td>
<td>1 2 3 7 8</td>
<td>$\vdash id + id + id$</td>
<td>$\vdash$</td>
<td>reduce $E \rightarrow E + T$</td>
</tr>
<tr>
<td>$\vdash E$</td>
<td>1 2 3 8</td>
<td>$\vdash id + id + id$</td>
<td>$\vdash$</td>
<td>shift</td>
</tr>
<tr>
<td>$\vdash E$</td>
<td>1 2 3 4</td>
<td>$\vdash id + id + id$</td>
<td>$\epsilon$</td>
<td>reduce $S' \rightarrow \vdash E$</td>
</tr>
<tr>
<td>$S'$</td>
<td>1</td>
<td>$\vdash id + id + id$</td>
<td>$\epsilon$</td>
<td>accept</td>
</tr>
</tbody>
</table>
What can go wrong?

Two distinct problems:

Problem 1: What if the state looks like this?

\[
\begin{align*}
A & \rightarrow \alpha \bullet c \beta \\
B & \rightarrow \gamma \bullet
\end{align*}
\]

Do we try to shift the next character (as suggested by \(A \rightarrow \alpha \bullet c \beta\)) or do we reduce by \(B \rightarrow \gamma\) (as suggested by \(B \rightarrow \gamma \bullet\))?
This is known as a *shift-reduce conflict*.

Problem 2: What if the state looks like this?

\[
\begin{align*}
A & \rightarrow \alpha \bullet \\
B & \rightarrow \beta \bullet
\end{align*}
\]

Do we reduce by \(A \rightarrow \alpha\) or by \(B \rightarrow \beta\)?
This is known as a *reduce-reduce conflict*.

If any item \(A \rightarrow \alpha \bullet\) occurs in a state in which it is not alone, then there is a shift-reduce or reduce-reduce conflict and the grammar is not LR(0).
Example with conflicts

Consider right-associative expressions. Modify our grammar slightly to allow (reverse RHS of second-rule).

1. \( S' \rightarrow \top E \bot \)
2. \( E \rightarrow T + E \)
3. \( E \rightarrow T \)
4. \( T \rightarrow \text{id} \)

DFA:
Parsing with conflicts

Suppose we are parsing a string that looks like \( \text{\texttt{id...}} \)

Picture of the stack:

Question: Should we reduce \( E \rightarrow T \)?
Answer: It depends.

- If input is \( \text{\texttt{id\texttt{\ldots}}} \), then yes.
- If input is \( \text{\texttt{id+\ldots}} \), then no.
Looking ahead

If we add a lookahead token to the automaton, we can fix the conflict.

For each $A \rightarrow \alpha \bullet$, attach $\text{Follow}(A)$.

For our grammar:

$\text{Follow}(E) =$

$\text{Follow}(T) =$

Consider our conflicting state:

Interpretation: A reduce action $A \rightarrow \alpha \bullet F$ (where $F$ is the $\text{Follow}(A)$) applies only if the next character is in $F$. 
When we add this one character of lookahead, we have an SLR(1) (Simple LR with 1 character of lookahead) parser. SLR(1) resolves many, but not all, conflicts.

- LR(1) parsing is more sophisticated than SLR(1) parsers
- LR(1) parses strictly more grammars
- LR(1) automaton is more complex
- LR(1) and SLR(1) are identical as parsing algorithms: the only difference is in the respective automaton they create

There is also a parser called LALR(1) (lookahead LR(1)), which falls between SLR(1) and LR(1).
- this is what Yacc and Bison use
Making this more efficient

Current running time of this algorithm:

Instead of scanning the stack each time...

Start the transducer in....

Running time:
Outputting a derivation

- Easy: each time we do a reduction, output the rule
- But, this isn’t quite right. Derivations should start with the start symbol. Bottom-up parsing doesn’t.
A simple observation

- Didn’t we say that this was LR(1) parsing?

- Doesn’t the “R” mean rightmost derivation?

- Aren’t we always reducing the leftmost nonterminal?

- But notice the direction we are creating the derivation. Write the derivation in reverse.
Outputting the parse tree

Algorithm

▷ Create a “tree stack”
▷ Each time we reduce, pop the right hand side nodes from tree stack
▷ Push the left hand side node and make its children the nodes we just popped
▷ Example:
How the tree is actually built in LR parsing
How the tree is actually built in LL parsing
Assignment 8 hints

Note that the automaton in cfg-r format specifies:

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>shift/reduce</th>
<th>next state / production number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(terminal</td>
<td>(for shifts) / (for reductions)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or non-terminal</td>
<td></td>
</tr>
</tbody>
</table>

P1, P2: write a cfg-r derivation by hand

P3: Write a parser
   - Read a CFG, the DFA and input
   - Output cfg-r (derivation) if input is in the language, ERROR otherwise

P4: Write a parser for WLM
   - Your parser will read tokens, output as it shifts.
   - Find a way to embed the WLM grammar and DFA table in your program.

P5: Redo P4 to produce the output in .wlmi format.

Build a parse tree! Build a parse tree! Build a parse tree!
Going back

Looking at: \( L = \{a^n b^m : n \geq m \geq 0\} \) (non-LL(1) language)

1. \( S' \rightarrow \vdash S \vdash \)
2. \( S \rightarrow a \ S \)
3. \( S \rightarrow T \)
4. \( T \rightarrow a \ T \ b \)
5. \( T \rightarrow \epsilon \)

What to do when you see the symbol:

- \( \vdash \)
- \( a \)
- \( b \)
- \( \vdash \)
Final fun facts

- Theorem: For any augmented LR(1) grammar, there is an equivalent LR(0) grammar.
- Theorem: The class of languages that can be parsed deterministically with a stack can be represented with an LR(1) grammar.
- Comparing LL(1) vs. LR(1)