CS241 Midterm Review

Winter 2020

University of Waterloo

Slides available at:
• Write the binary and the hexadecimal form of the MIPS instruction `lw $14, -421($27).`
• Write the binary and the hexadecimal form of the MIPS instruction
  `lw $14, -421($27)`.

• From the MIPS reference sheet, this is the format for `lw $t, i($s)`:
  
  100011 sssss ttttt iiii iiii iiii iiii
• Write the binary and the hexadecimal form of the MIPS instruction `lw $14, −421($27).

• From the MIPS reference sheet, this is the format for `lw $t, i($s):
  100011 sssss ttttt iiii iiii iiii iiii

• We need to convert 14 and 27 to 5-bit unsigned binary, and −421 to 16-bit signed two’s complement binary.
• Write the binary and the hexadecimal form of the MIPS instruction `lw $14, -421($27).

• From the MIPS reference sheet, this is the format for `lw $t, i($s):
  100011 sssss ttttt iiiiii iiiiii iiiiii

• We need to convert 14 and 27 to 5-bit unsigned binary, and −421 to 16-bit signed two’s complement binary.

• 14 is $8 + 4 + 2$ so we get $01110$, 27 is $16 + 8 + 2 + 1$ so we get $11011$. 

Write the binary and the hexadecimal form of the MIPS instruction `lw $14, -421($27).

From the MIPS reference sheet, this is the format for `lw $t, i($s):
```
100011 sssss ttttt iiiii iiiii iiiii iiiii
```

We need to convert 14 and 27 to 5-bit unsigned binary, and −421 to 16-bit signed two’s complement binary.

14 is $8 + 4 + 2$ so we get 01110, 27 is $16 + 8 + 2 + 1$ so we get 11011.

For −421, first convert 421 to unsigned 16-bit binary. We’ll use the repeated division trick:

\[
\begin{align*}
421/2 &= 210 \ r1 \\
210/2 &= 105 \ r0 \\
105/2 &= 52 \ r1 \\
52/2 &= 26 \ r0 \\
26/2 &= 13 \ r0 \\
13/2 &= 6 \ r1 \\
6/2 &= 3 \ r0 \\
3/2 &= 1 \ r1 \\
1/2 &= 0 \ r1
\end{align*}
\]
• Write the binary and the hexadecimal form of the MIPS instruction `lw $14, -421($27).
• From the MIPS reference sheet, this is the format for `lw $t, i($s):
100011 sssss ttttt iiiii iiiii iiiii iiiii
• We need to convert 14 and 27 to 5-bit unsigned binary, and -421 to 16-bit signed two’s complement binary.
• 14 is 8 + 4 + 2 so we get 01110, 27 is 16 + 8 + 2 + 1 so we get 11011.
• For -421, first convert 421 to unsigned 16-bit binary. We’ll use the repeated division trick:
  \[
  \begin{align*}
  421/2 &= 210 \ r1 \\
  210/2 &= 105 \ r0 \\
  105/2 &= 52 \ r1 \\
  52/2 &= 26 \ r0 \\
  26/2 &= 13 \ r0 \\
  13/2 &= 6 \ r1 \\
  6/2 &= 3 \ r0 \\
  3/2 &= 1 \ r1 \\
  1/2 &= 0 \ r1 \\
\end{align*}
\]
• Reading remainders bottom to top gives 110100101. We pad this to 16 bits: 0000 0001 1010 0101
• Write the binary and the hexadecimal form of the MIPS instruction `lw $14, -421($27).
• From the MIPS reference sheet, this is the format for `lw $t, i($s):
  100011 sssss ttttt iiii iiii iiii iiii
• We need to convert 14 and 27 to 5-bit unsigned binary, and −421 to 16-bit signed two’s complement binary.
• 14 is 8 + 4 + 2 so we get 01110, 27 is 16 + 8 + 2 + 1 so we get 11011.
• For −421, first convert 421 to unsigned 16-bit binary. We’ll use the repeated division trick:

\[
\begin{align*}
421/2 & = 210 \ r1 \\
210/2 & = 105 \ r0 \\
105/2 & = 52 \ r1 \\
52/2 & = 26 \ r0 \\
26/2 & = 13 \ r0 \\
13/2 & = 6 \ r1 \\
6/2 & = 3 \ r0 \\
3/2 & = 1 \ r1 \\
1/2 & = 0 \ r1 \\
\end{align*}
\]

• Reading remainders bottom to top gives 110100101. We pad this to 16 bits:
  0000 0001 1010 0101
• Now flip the bits and add one to get −421:
  1111 1110 0101 1011
Or equivalently, flip the bits to the left of the rightmost 1.
• Write the binary and the hexadecimal form of the MIPS instruction `lw $14, -421($27).
• From the MIPS reference sheet, this is the format for `lw $t, i($s):
  1000 11ss ssst tttt iiiii iiiii iiiii iiiii
• Using the numbers computed on the previous slide, the binary representation is: 100011 11011 01110 1111 1110 0101 1011
• Write the binary and the hexadecimal form of the MIPS instruction `lw $14, -421($27).
• From the MIPS reference sheet, this is the format for `lw $t, i($s):
  1000 11ss ssst tttt iiiii iiiii iiiii
• Using the numbers computed on the previous slide, the binary representation is: 100011 11011 01110 1111 1110 0101 1011
• To convert to hexadecimal, convert each 4-bit chunk (nibble).
• Write the binary and the hexadecimal form of the MIPS instruction
  lw $14, -421($27).
• From the MIPS reference sheet, this is the format for lw $t, i($s):
  1000 11ss ssst tttt iiiii iiiii iiiii iiiii
• Using the numbers computed on the previous slide, the binary
  representation is: 100011 11011 01110 1111 1110 0101 1011
• To convert to hexadecimal, convert each 4-bit chunk (nibble).
• You may want to convert the nibbles to decimal first, then figure out
  the hex value corresponding to the numbers larger than 9.
Binary, Decimal and Hexadecimal Conversions

- Write the binary and the hexadecimal form of the MIPS instruction `lw $14, -421($27)`.
- From the MIPS reference sheet, this is the format for `lw $t, i($s)`: 1000 11ss ssst tttt iiiii iiiii iiiii
- Using the numbers computed on the previous slide, the binary representation is: 100011 11011 01110 1111 1110 0101 1011
- To convert to hexadecimal, convert each 4-bit chunk (nibble).
- You may want to convert the nibbles to decimal first, then figure out the hex value corresponding to the numbers larger than 9.
- Binary: 1000 1111 0110 1110 1111 1110 0101 1011
  - Decimal Nibbles: 8 15 6 14 15 14 5 11
  - Hex Nibbles: 8 f 6 e f e 5 b
  - Hexadecimal: 0x8f6efe5b
Write the binary and the hexadecimal form of the MIPS instruction `lw $14, -421($27)`.

From the MIPS reference sheet, this is the format for `lw $t, i($s)`:

```
1000 11ss ssst tttt iiii iiii iiii iiii
```

Using the numbers computed on the previous slide, the binary representation is: `100011110110111101111111001011011`

To convert to hexadecimal, convert each 4-bit chunk (nibble).

You may want to convert the nibbles to decimal first, then figure out the hex value corresponding to the numbers larger than 9.

**Binary:**

```
1000 1111 0110 1110 1111 1110 0101 1011
```

**Decimal Nibbles:**

```
8  15  6  14  15  14  5  11
```

**Hex Nibbles:**

```
8  f  6  e  f  e  5  b
```

**Hexadecimal:** `0x8f6efe5b`

Remark: To construct this instruction with bitwise operations you would do the following:

```
(0b100011 << 26) | (27 << 21) | (14 << 16) | (-421 & 0xffffffff)
```
Recall the order of operations in the fetch-execute cycle:
- Fetch the instruction at the address in PC (the program counter) and store it in IR (the instruction register).
- Increment PC by 4.
- Execute the instruction in IR.

How does the `lis` instruction take advantage of this order?

The definition of `lis $d` on the MIPS reference sheet is $d = MEM[PC]; PC = PC + 4$.

Here `MEM[PC]` means "the value in memory at address PC".

<table>
<thead>
<tr>
<th>Memory</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>Data</td>
</tr>
<tr>
<td>PC = 0</td>
<td><code>lis $3</code></td>
</tr>
<tr>
<td>4</td>
<td><code>.word 55</code></td>
</tr>
<tr>
<td>8</td>
<td><code>jr $31</code></td>
</tr>
</tbody>
</table>
The Fetch-Execute Cycle and MIPS Instructions

- Recall the order of operations in the fetch-execute cycle:
  - Fetch the instruction at the address in PC (the program counter) and store it in IR (the instruction register).
  - Increment PC by 4.
  -Execute the instruction in IR.
- How does the `lis` instruction take advantage of this order?
- The definition of `lis $d` on the MIPS reference sheet is `d = MEM[PC] ; PC = PC + 4`.
- Here `MEM[PC]` means “the value in memory at address PC”.

### Memory

<table>
<thead>
<tr>
<th>Location</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC = 0</td>
<td><code>lis $3</code></td>
</tr>
<tr>
<td>4</td>
<td><code>.word 55</code></td>
</tr>
<tr>
<td>8</td>
<td><code>jr $31</code></td>
</tr>
</tbody>
</table>

### Registers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0</td>
</tr>
<tr>
<td>IR</td>
<td><code>lis $3</code></td>
</tr>
<tr>
<td>$3</td>
<td>??</td>
</tr>
</tbody>
</table>

Fetch the instruction.
Recall the order of operations in the fetch-execute cycle:
- Fetch the instruction at the address in PC (the program counter) and store it in IR (the instruction register).
- Increment PC by 4.
- Execute the instruction in IR.

How does the `lis` instruction take advantage of this order?
- The definition of `lis $d` on the MIPS reference sheet is $d = MEM[PC]; PC = PC + 4$.
- Here $MEM[PC]$ means “the value in memory at address PC”.

<table>
<thead>
<tr>
<th>Location</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>lis $3</code></td>
</tr>
<tr>
<td>PC = 4</td>
<td><code>.word 55</code></td>
</tr>
<tr>
<td>8</td>
<td><code>jr $31</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
</tr>
<tr>
<td>IR</td>
</tr>
<tr>
<td>$3</td>
</tr>
</tbody>
</table>

Increment the program counter.
The Fetch-Execute Cycle and MIPS Instructions

- Recall the order of operations in the fetch-execute cycle:
  - Fetch the instruction at the address in PC (the program counter) and store it in IR (the instruction register).
  - Increment PC by 4.
  - Execute the instruction in IR.

- How does the `lis` instruction take advantage of this order?

- The definition of `lis $d` on the MIPS reference sheet is $d = MEM[PC] ; PC = PC + 4$.

- Here $MEM[PC]$ means “the value in memory at address PC”.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>Data</td>
</tr>
<tr>
<td>0</td>
<td><code>lis $3</code></td>
</tr>
<tr>
<td>PC = 4</td>
<td><code>.word 55</code></td>
</tr>
<tr>
<td>8</td>
<td><code>jr $31</code></td>
</tr>
</tbody>
</table>

Execute the instruction: $3 = MEM[PC]$.
Recall the order of operations in the fetch-execute cycle:

- Fetch the instruction at the address in PC (the program counter) and store it in IR (the instruction register).
- Increment PC by 4.
- Execute the instruction in IR.

How does the `lis` instruction take advantage of this order?

The definition of `lis` $d$ on the MIPS reference sheet is $d = MEM[PC]; \ PC = \ PC + 4$.

Here $MEM[PC]$ means “the value in memory at address PC”.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>Data</td>
</tr>
<tr>
<td>0</td>
<td>lis $3$</td>
</tr>
<tr>
<td>4</td>
<td>.word 55</td>
</tr>
<tr>
<td>PC = 8</td>
<td>jr $31$</td>
</tr>
</tbody>
</table>

Execute the instruction: $PC = PC + 4$. 
Recall the order of operations in the fetch-execute cycle:

- Fetch the instruction at the address in PC (the program counter) and store it in IR (the instruction register).
- Increment PC by 4.
- Execute the instruction in IR.

How does the lis instruction take advantage of this order?
The definition of lis $d$ on the MIPS reference sheet is $d = \text{MEM}[PC]; PC = PC + 4$.
Here MEM[PC] means “the value in memory at address PC”.

<table>
<thead>
<tr>
<th>Location</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>lis $3$</td>
</tr>
<tr>
<td>4</td>
<td>.word 55</td>
</tr>
<tr>
<td>PC = 8</td>
<td>jr $31$</td>
</tr>
</tbody>
</table>

Next Fetch-Execute cycle will execute the jr $31$ instruction.
The Fetch-Execute Cycle and MIPS Instructions

- Consider the following MIPS program, where X and Y are integer constants.
  
  ```mips
  bne $1, $0, X
  lis $1
  add $2, $31, $0
  beq $2, $0, Y
  jalr $0
  jr $2
  jr $31
  .word -1
  ```

- Assume this program is loaded at address 0.

- Let's figure out what happens when this program is run with the values $X = 5, Y = -2, $1 = 0$ and $2 = 0$. 

Consider the following MIPS program, where X and Y are integer constants.

```
00 bne $1, $0, X ; X = -1  Y = -4
04 lis $1        ; X =  0  Y = -3
08 add $2, $31, $0 ; X =  1  Y = -2
12 beq $2, $0, Y ; X =  2  Y = -1
16 jalr $0      ; X =  3  Y =  0
20 jr $2        ; X =  4  Y =  1
24 jr $31       ; X =  5  Y =  2
28 .word -1      ; X =  6  Y =  3
```

Let’s figure out what happens when this program is run with the values X = 5, Y = -2, $1 = 0 and $2 = 0.

We have annotated the program to show instruction locations, and where each branch will go for given values of X and Y.

PC is incremented before the branch instruction executes.
Consider the following MIPS program.

```
00 bne $1, $0, 5 ; branch offset X = 5
04 lis $1
08 add $2, $31, $0 ; location corresponding to Y
12 beq $2, $0, -2 ; branch offset Y = -2
16 jalr $0
20 jr $2
24 jr $31 ; location corresponding to X
28 .word -1
```

We run this with $X = 5$, $Y = -2$, $\$1 = 0$ and $\$2 = 0$.

```
Instruction: bne $1, $0, 5
We don’t branch because $\$1 = 0.$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$$1$</td>
<td>0</td>
</tr>
<tr>
<td>$$2$</td>
<td>0</td>
</tr>
<tr>
<td>$$31$</td>
<td>loader return address</td>
</tr>
</tbody>
</table>
```
Consider the following MIPS program.

```
00 bne $1, $0, 5  ; branch offset X = 5
04 lis $1
08 add $2, $31, $0  ; location corresponding to Y
12 beq $2, $0, -2  ; branch offset Y = -2
16 jalr $0
20 jr $2
24 jr $31            ; location corresponding to X
28 .word -1
```

We run this with $X = 5$, $Y = -2$, $S1 = 0$ and $S2 = 0$.

**Instruction:** lis $1

We load the add $2, $31, $0 instruction into $1$ and skip over it!

<table>
<thead>
<tr>
<th></th>
<th>add $2, $31, $0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td>$2$</td>
<td>0</td>
</tr>
<tr>
<td>$31$</td>
<td>loader return address</td>
</tr>
</tbody>
</table>
Consider the following MIPS program.

```
00 bne $1, $0, 5    ; branch offset X = 5
04 lis $1
08 add $2, $31, $0  ; location corresponding to Y
12 beq $2, $0, -2   ; branch offset Y = -2
16 jalr $0
20 jr $2
24 jr $31            ; location corresponding to X
28 .word -1
```

We run this with $X = 5$, $Y = -2$, $\$1 = 0$ and $\$2 = 0$.

Instruction: `beq $2, $0, -2`

Since $\$2 = 0$ we branch back to the add instruction...

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td>add $2$, $31$, $0$</td>
</tr>
<tr>
<td>$2$</td>
<td>0</td>
</tr>
<tr>
<td>$31$</td>
<td>loader return address</td>
</tr>
</tbody>
</table>
Consider the following MIPS program.

```
00 bne $1, $0, 5 ; branch offset X = 5
04 lis $1
08 add $2, $31, $0 ; location corresponding to Y
12 beq $2, $0, -2 ; branch offset Y = -2
16 jalr $0
20 jr $2
24 jr $31 ; location corresponding to X
28 .word -1
```

We run this with $X = 5$, $Y = -2$, $\$1 = 0$ and $\$2 = 0$.

**Instruction:** add $\$2, $\$31, $\$0

We copy $\$31$ into $\$2.$

<table>
<thead>
<tr>
<th>$$1</th>
<th>$$2</th>
<th>$$31</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{add } $2, $31, $0$</td>
<td>$\text{loader return adddress}$</td>
<td>$\text{loader return address}$</td>
</tr>
</tbody>
</table>
Consider the following MIPS program.

```
00 bne $1, $0, 5 ; branch offset X = 5
04 lis $1
08 add $2, $31, $0 ; location corresponding to Y
12 beq $2, $0, -2 ; branch offset Y = -2
16 jalr $0
20 jr $2
24 jr $31 ; location corresponding to X
28 .word -1
```

We run this with $X = 5$, $Y = -2$, $S1 = 0$ and $S2 = 0$.

Instruction: beq $2$, $0$, $-2$

Now $S2$ is non-zero so we don't branch.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S1$</td>
<td>add $S2$, $S31$, $0$</td>
</tr>
<tr>
<td>$S2$</td>
<td>loader return address</td>
</tr>
<tr>
<td>$S31$</td>
<td>loader return address</td>
</tr>
</tbody>
</table>
Consider the following MIPS program.

```
00 bne $1, $0, 5       ; branch offset X = 5
04 lis $1
08 add $2, $31, $0     ; location corresponding to Y
12 beq $2, $0, -2      ; branch offset Y = -2
16 jalr $0
20 jr $2
24 jr $31              ; location corresponding to X
28 .word -1
```

We run this with X = 5, Y = -2, $1 = 0 and $2 = 0.

Instruction: jalr $0

$31 gets the address after jalr $0. Then jump to address 0.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$1</td>
<td>add $2, $31, $0</td>
</tr>
<tr>
<td>$2</td>
<td>loader return address</td>
</tr>
<tr>
<td>$31</td>
<td>20</td>
</tr>
</tbody>
</table>
Consider the following MIPS program.

```
00 bne $1, $0, 5 ; branch offset X = 5
04 lis $1
08 add $2, $31, $0 ; location corresponding to Y
12 beq $2, $0, -2 ; branch offset Y = -2
16 jalr $0
20 jr $2
24 jr $31 ; location corresponding to X
28 .word -1
```

We run this with $X = 5$, $Y = -2$, $S1 = 0$ and $S2 = 0$.

Instruction: `bne $1, $0, 5`

Now $S1$ is non-zero, so we branch.

<table>
<thead>
<tr>
<th>$S1$</th>
<th><code>add $2, $31, $0</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S2$</td>
<td>loader return address</td>
</tr>
<tr>
<td>$S31$</td>
<td>20</td>
</tr>
</tbody>
</table>
Consider the following MIPS program.

```
00 bne $1, $0, 5 ; branch offset X = 5
04 lis $1
08 add $2, $31, $0 ; location corresponding to Y
12 beq $2, $0, -2 ; branch offset Y = -2
16 jalr $0
20 jr $2
24 jr $31 ; location corresponding to X
28 .word -1
```

We run this with $X = 5$, $Y = -2$, $\$1 = 0$ and $\$2 = 0$.

Instruction: `jr $31`

We jump to location 20 where the `jr $2` instruction is...

| $1$    | add $2$, $31$, $0$ |
| $2$    | loader return address |
| $31$   | 20 |
- Consider the following MIPS program.

```mips
00 bne $1, $0, 5 ; branch offset X = 5
04 lis $1
08 add $2, $31, $0 ; location corresponding to Y
12 beq $2, $0, -2 ; branch offset Y = -2
16 jalr $0
20 jr $2
24 jr $31 ; location corresponding to X
28 .word -1
```

- We run this with X = 5, Y = -2, $1 = 0 and $2 = 0.

**Instruction:** `jr $2`

Jump to the address in $2. Somehow we returned to the loader!

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$1</td>
<td>add $2, $31, $0</td>
</tr>
<tr>
<td>$2</td>
<td>loader return address</td>
</tr>
<tr>
<td>$31</td>
<td>20</td>
</tr>
</tbody>
</table>
MIPS Procedures

```
main:    divideAndAdd:
00      00 sw $31, -4($30)  48     sw $1, -4($30)
04      04 lis $31          52     lis $3
08      08 .word 4          56     .word 4
12      12 sub $30, $30, $31 60     sub $30, $30, $3
16      16 lis $2           64     div $1, $3
20      20 .word divideAndAdd 68     mflo $1
24      24 jalr $2          72     add $3, $1, $3
28      28 lis $31          76     lis $1
32      32 .word 4          80     .word 4
36      36 add $30, $30, $31 84     add $30, $30, $1
40      40 lw $31, -4($30)  88     lw $1, -4($30)
44      44 jr $31           92     jr $31
```

- This procedure takes the number in register 1, divides it by 4, then adds 4, and returns the result in register 3.
MIPS Procedures

main:

00 sw $31, -4($30)
04 lis $31
08 .word 4
12 sub $30, $30, $31
16 lis $2
20 .word divideAndAdd
24 jalr $2
28 lis $31
32 .word 4
36 add $30, $30, $31
40 lw $31, -4($30)
44 jr $31

divideAndAdd:

48 sw $1, -4($30)
52 lis $3
56 .word 4
60 sub $30, $30, $3
64 div $1, $3
68 mflo $1
72 add $3, $1, $3
76 lis $1
80 .word 4
84 add $30, $30, $1
88 lw $1, -4($30)
92 jr $31

• We use register 3 to decrement the stack pointer (lines 52-60). Why can’t we use register 1, since we just saved the value?
MIPS Procedures

main:

```
00  sw $31, -4($30)
04  lis $31
08  .word 4
12  sub $30, $30, $31
16  lis $2
20  .word divideAndAdd
24  jalr $2
28  lis $31
32  .word 4
36  add $30, $30, $31
40  lw $31, -4($30)
44  jr $31
```

divideAndAdd:

```
48  sw $1, -4($30)
52  lis $3
56  .word 4
60  sub $30, $30, $3
64  div $1, $3
68  mflo $1
72  add $3, $1, $3
76  lis $1
80  .word 4
84  add $30, $30, $1
88  lw $1, -4($30)
92  jr $31
```

- We use register 3 to decrement the stack pointer (lines 52-60). Why can’t we use register 1, since we just saved the value? **Register 1 is the parameter, this would overwrite it.**
### MIPS Procedures

**main:**

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td><code>sw $31, -4($30)</code></td>
<td>48</td>
<td><code>sw $1, -4($30)</code></td>
</tr>
<tr>
<td>04</td>
<td><code>lis $31</code></td>
<td>52</td>
<td><code>lis $3</code></td>
</tr>
<tr>
<td>08</td>
<td><code>.word 4</code></td>
<td>56</td>
<td><code>.word 4</code></td>
</tr>
<tr>
<td>12</td>
<td><code>sub $30, $30, $31</code></td>
<td>60</td>
<td><code>sub $30, $30, $3</code></td>
</tr>
<tr>
<td>16</td>
<td><code>lis $2</code></td>
<td>64</td>
<td><code>div $1, $3</code></td>
</tr>
<tr>
<td>20</td>
<td><code>.word divideAndAdd</code></td>
<td>68</td>
<td><code>mflo $1</code></td>
</tr>
<tr>
<td>24</td>
<td><code>jalr $2</code></td>
<td>72</td>
<td><code>add $3, $1, $3</code></td>
</tr>
<tr>
<td>28</td>
<td><code>lis $31</code></td>
<td>76</td>
<td><code>lis $1</code></td>
</tr>
<tr>
<td>32</td>
<td><code>.word 4</code></td>
<td>80</td>
<td><code>.word 4</code></td>
</tr>
<tr>
<td>36</td>
<td><code>add $30, $30, $31</code></td>
<td>84</td>
<td><code>add $30, $30, $1</code></td>
</tr>
<tr>
<td>40</td>
<td><code>lw $31, -4($30)</code></td>
<td>88</td>
<td><code>lw $1, -4($30)</code></td>
</tr>
<tr>
<td>44</td>
<td><code>jr $31</code></td>
<td>92</td>
<td><code>jr $31</code></td>
</tr>
</tbody>
</table>

**divideAndAdd:**

- We use register 1 to increment the stack pointer (lines 76-84). Why can’t we use register 3 like we did before?
MIPS Procedures

main:
00  sw $31, -4($30)
04  lis $31
08  .word 4
12  sub $30, $30, $31
16  lis $2
20  .word divideAndAdd
24  jalr $2
28  lis $31
32  .word 4
36  add $30, $30, $31
40  lw $31, -4($30)
44  jr $31

divideAndAdd:
48  sw $1, -4($30)
52  lis $3
56  .word 4
60  sub $30, $30, $31
64  div $1, $3
68  mflo $1
72  add $3, $1, $3
76  lis $1
80  .word 4
84  add $30, $30, $1
88  lw $1, -4($30)
92  jr $31

• We use register 1 to increment the stack pointer (lines 76-84). Why can’t we use register 3 like we did before? Register 3 is the return value, this would overwrite it.
main:
00  sw $31, -4($30)
04  lis $31
08  .word 4
12  sub $30, $30, $31
16  lis $2
20  .word divideAndAdd
24  jalr $2
28  lis $31
32  .word 4
36  add $30, $30, $31
40  lw $31, -4($30)
44  jr $31

divideAndAdd:
48  sw $1, -4($30)
52  lis $3
56  .word 4
60  sub $30, $30, $3
64  div $1, $3
68  mflo $1
72  add $3, $1, $3
76  lis $1
80  .word 4
84  add $30, $30, $1
88  lw $1, -4($30)
92  jr $31

• Notice we only save and restore register 1 (lines 48 and 88). Why don’t we save and restore register 3?
MIPS Procedures

main:
00 sw $31, -4($30)
04 lis $31
08 .word 4
12 sub $30, $30, $31
16 lis $2
20 .word divideAndAdd
24 jalr $2
28 lis $31
32 .word 4
36 add $30, $30, $31
40 lw $31, -4($30)
44 jr $31

divideAndAdd:
48 sw $1, -4($30)
52 lis $3
56 .word 4
60 sub $30, $30, $3
64 div $1, $3
68 mflo $1
72 add $3, $1, $3
76 lis $1
80 .word 4
84 add $30, $30, $1
88 lw $1, -4($30)
92 jr $31

• Notice we only save and restore register 1 (lines 48 and 88). Why don’t we save and restore register 3? Again, it’s the return value, this would overwrite it.
MIPS Procedures

main:
00  sw $31, -4($30)
04  lis $31
08  .word 4
12  sub $30, $30, $31
16  lis $2
20  .word divideAndAdd
24  jalr $2
28  lis $31
32  .word 4
36  add $30, $30, $31
40  lw $31, -4($30)
44  jr $31

divideAndAdd:
48  sw $1, -4($30)
52  lis $3
56  .word 4
60  sub $30, $30, $3
64  div $1, $3
68  mflo $1
72  add $3, $1, $3
76  lis $1
80  .word 4
84  add $30, $30, $1
88  lw $1, -4($30)
92  jr $31

- The main program saves and restores register 31 (lines 0 and 40). Why is this necessary?
MIPS Procedures

main:

00 sw $31, -4($30)
04 lis $31
08 .word 4
12 sub $30, $30, $31
16 lis $2
20 .word divideAndAdd
24 jalr $2
28 lis $31
32 .word 4
36 add $30, $30, $31
40 lw $31, -4($30)
44 jr $31

divideAndAdd:

48 sw $1, -4($30)
52 lis $3
56 .word 4
60 sub $30, $30, $3
64 div $1, $3
68 mflo $1
72 add $3, $1, $3
76 lis $1
80 .word 4
84 add $30, $30, $1
88 lw $1, -4($30)
92 jr $31

- The main program saves and restores register 31 (lines 0 and 40). Why is this necessary? Using jalr to call a procedure overwrites register 31.
MIPS Procedures

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>sw $31, -4($30)</td>
</tr>
<tr>
<td>04</td>
<td>lis $31</td>
</tr>
<tr>
<td>08</td>
<td>.word 4</td>
</tr>
<tr>
<td>12</td>
<td>sub $30, $30, $31</td>
</tr>
<tr>
<td>16</td>
<td>lis $2</td>
</tr>
<tr>
<td>20</td>
<td>.word divideAndAdd</td>
</tr>
<tr>
<td>24</td>
<td>jalr $2</td>
</tr>
<tr>
<td>28</td>
<td>lis $31</td>
</tr>
<tr>
<td>32</td>
<td>.word 4</td>
</tr>
<tr>
<td>36</td>
<td>add $30, $30, $31</td>
</tr>
<tr>
<td>40</td>
<td>lw $31, -4($30)</td>
</tr>
<tr>
<td>44</td>
<td>jr $31</td>
</tr>
<tr>
<td>48</td>
<td>sw $1, -4($30)</td>
</tr>
<tr>
<td>52</td>
<td>lis $3</td>
</tr>
<tr>
<td>56</td>
<td>.word 4</td>
</tr>
<tr>
<td>60</td>
<td>sub $30, $30, $30, $3</td>
</tr>
<tr>
<td>64</td>
<td>div $1, $3</td>
</tr>
<tr>
<td>68</td>
<td>mflo $1</td>
</tr>
<tr>
<td>72</td>
<td>add $3, $1, $3</td>
</tr>
<tr>
<td>76</td>
<td>lis $1</td>
</tr>
<tr>
<td>80</td>
<td>.word 4</td>
</tr>
<tr>
<td>84</td>
<td>add $30, $30, $1</td>
</tr>
<tr>
<td>88</td>
<td>lw $1, -4($30)</td>
</tr>
<tr>
<td>92</td>
<td>jr $31</td>
</tr>
</tbody>
</table>

- The divideAndAdd procedure is not recursive. If it was recursive, what additional register would it need to save?
The divideAndAdd procedure is not recursive. If it was, what additional register would it need to save? It would need to save register 31, since it is calling a procedure (itself).
Building a Symbol Table

; charcount.asm: counts number of characters in a file read from standard input,  
; returns the count in $3. This procedure saves and restores all other registers,  
; but it does so without using the stack at all! Don’t do this on an exam!

charcount:
    lis $3
    .word returnAddress
    sw $31, 0($3) ; save return address in a "variable" at end of program
    lis $3
    .word count
    sw $0, 0($3) ; our "variables" are global, so we need to reinitialize

loop:
    lis $31
    .word 0xffff0004 ; address for reading from standard input
    lw $31, 0($31) ; $31 = next character from standard input
    lis $3
    .word -1 ; -1 represents end of file
    beq $31, $3, endOfFile
    lis $3
    .word count
    lw $3, 0($3) ; $3 = count
    lis $31
    .word 1
    add $3, $3, $31 ; $3 gets incremented
    lis $31
    .word count
    sw $3, 0($31) ; count gets incremented
    beq $0, $0, loop

endOfFile:
    lis $3
    .word count
    lw $3, 0($3) ; retrieve the count variable
    lis $31
    .word returnAddress
    lw $31, 0($31) ; restore the return address
    jr $31

Lines which do not contain an instruction are called null lines.

This includes blank lines, lines with only labels, and lines with only comments.

The first non-null line gets location 0, then we count up by 4 for each line.
Building a Symbol Table

; charcount.asm: counts number of characters in a file read from standard input, 
; returns the count in $3. this procedure saves and restores all other registers, 
; but it does so without using the stack at all! don't do this on an exam!

charcount:
000 lis $3
004 .word returnAddress
008 sw $31, 0($3)
012 lis $3
016 .word count
020 sw $0, 0($3)

loop:
024 lis $31
028 .word 0xffff0004 ; address for reading from standard input
032 lw $31, 0($31) ; $31 = next character from standard input
036 lis $3
040 .word -1 ; -1 represents end of file
044 beq $31, $3, endOfFile
048 lis $3
052 .word count
056 lw $3, 0($3) ; $3 = count
060 lis $31
064 .word 1
068 add $3, $3, $31 ; $3 gets incremented
072 lis $31
076 .word count
080 sw $3, 0($31) ; count gets incremented
084 beq $0, $0, loop

endOfFile:
088 lis $3
092 .word count
096 lw $3, 0($3) ; retrieve the count variable
100 lis $31
104 .word returnAddress
108 lw $31, 0($31) ; restore the return address
112 jr $31

globalVariables:
116 returnAddress: .word 0
120 count: .word 0
Building a Symbol Table

; charcount.asm: counts number of characters in a file read from standard input,
; returns the count in $3. this procedure saves and restores all other registers,
; but it does so without using the stack at all! don't do this on an exam!

charcount:
000  lis $3
004   .word returnAddress
008   sw $31, 0($3) ; save return address in a "variable" at end of program
012   lis $3
016   .word count
020   sw $0, 0($3) ; our "variables" are global, so we need to reinitialize

loop:
024   lis $31
028   .word 0xffff0004 ; address for reading from standard input
032   lw $31, 0($31) ; $31 = next character from standard input
036   lis $3
040   .word -1 ; -1 represents end of file
044   beq $31, $3, endOfFile
048   lis $3   .word count
052   lw $3, 0($3) ; $3 = count
056   lis $31
060   lw $3 $31
064   .word 1
068   add $3, $3, $31 ; $3 gets incremented
072   lw $3 $31
076   .word count
080   sw $3, 0($31) ; count gets incremented
084   beq $0, $0, loop

endOfFile:
088   lis $3
092   .word count
096   lw $3, 0($3) ; retrieve the count variable
100   lis $31
104   .word returnAddress
108   lw $31, 0($31) ; restore the return address
112   jr $31

globalVariables:
116 returnAddress: .word 0
120 count: .word 0

• If a label occurs on a null line, its value is the location of the next non-null line.
• If a label occurs on a non-null line (followed by an instruction) its value is the location of the line.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>charcount</td>
<td>0</td>
</tr>
<tr>
<td>loop</td>
<td>24</td>
</tr>
<tr>
<td>endOfFile</td>
<td>88</td>
</tr>
<tr>
<td>globalVariables</td>
<td>116</td>
</tr>
<tr>
<td>returnAddress</td>
<td>116</td>
</tr>
<tr>
<td>count</td>
<td>120</td>
</tr>
</tbody>
</table>
Building a Symbol Table

; charcount.asm: counts number of characters in a file read from standard input, ; returns the count in $3. this procedure saves and restores all other registers, ; but it does so without using the stack at all! don't do this on an exam!

charcount:
000 lis $3
004 .word returnAddress
008 sw $31, 0($3) ; save return address in a "variable" at end of program
012 lis $3
016 .word count
020 sw $0, 0($3) ; the count variable every time the procedure is called

loop:
024 lis $31
028 .word 0xffff0004 ; address for reading from standard input
032 lw $31, 0($31) ; $31 = next character from standard input
036 lis $3
040 .word -1 ; -1 represents end of file
044 beq $31, $3, endOfFile
048 lis $3
052 .word count
056 lw $3, 0($3) ; $3 = count
060 lis $31
064 .word 1
068 add $3, $3, $31 ; $3 gets incremented
072 lis $31
076 .word count
080 sw $3, 0($31) ; count gets incremented
084 beq $0, $0, loop

endOfFile:
088 lis $3
092 .word count
096 lw $3, 0($3) ; retrieve the count variable
100 lis $31
104 .word returnAddress
108 lw $31, 0($31) ; restore the return address
112 jr $31

globalVariables:
116 returnAddress: .word 0
120 count: .word 0

- Branch offsets are computed using the following formula: (val-loc-4)/4 where loc is the branch instruction location and val is the label value.

- For .word label just substitute the label value.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>val</th>
<th>loc</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq $31, $3, endOfFile</td>
<td>88</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>beq $0, $0, loop</td>
<td>24</td>
<td>84</td>
<td></td>
</tr>
</tbody>
</table>
Building a Symbol Table

```
; charcount.asm: counts number of characters in a file read from standard input,
; returns the count in $3. this procedure saves and restores all other registers,
; but it does so without using the stack at all! don't do this on an exam!

charcount:
000 lis $3
004 .word returnAddress
008 sw $31, 0($3)
012 lis $3
016 .word count
020 sw $0, 0($3)

loop:
024 lis $31
028 .word 0xffff0004
032 lw $31, 0($31)
036 lis $3
040 .word -1
044 beq $31, $3, endOfFile
048 lis $3
052 .word count
056 lw $3, 0($3)
060 lis $31
064 .word 1
068 add $3, $3, $31

endOfFile:
088 lis $3
092 .word count
096 lw $3, 0($3)
100 lis $31
104 .word returnAddress
108 lw $31, 0($31)
112 jr $31

globalVariables:
116 returnAddress: .word 0
120 count: .word 0
```

- Branch offsets are computed using the following formula: 
  \( (val - loc - 4)/4 \)
  where \( loc \) is the branch instruction location and \( val \) is the label value.

- For .word label just substitute the label value.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>val</th>
<th>loc</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq $31, $3, endOfFile</td>
<td>88</td>
<td>44</td>
<td>10</td>
</tr>
<tr>
<td>beq $0, $0, loop</td>
<td>24</td>
<td>84</td>
<td>-16</td>
</tr>
</tbody>
</table>
Building a Symbol Table

• Here is the program with all label operands replaced.
• Write a regular expression for the language of words over \( \{a, b, c\} \) where the number of \( a \)'s is even. No restriction on \( b \)'s or \( c \)'s.
Write a regular expression for the language of words over \( \{a, b, c\} \) where the number of \( a \)'s is even. No restriction on \( b \)'s or \( c \)'s.

Even number of \( a \)'s: \((aa)^*\).
Regular Expressions

- Write a regular expression for the language of words over \( \{a, b, c\} \) where the number of \( a \)'s is even. No restriction on \( b \)'s or \( c \)'s.
- Even number of \( a \)'s: \((aa)^*\).
- We can have any number of \( b \)'s and \( c \)'s in between each \( a \) and at the start and end of the word.
• Write a regular expression for the language of words over \{a, b, c\} where the number of a’s is even. No restriction on b’s or c’s.
• Even number of a’s: \((aa)^*\).
• We can have any number of b’s and c’s in between each a and at the start and end of the word.
• Solution: \((b|c)^*(a(b|c)*a(b|c)^*)^*, \) or \(((b|c)^*a(b|c)*a)^*(b|c)^*\).
Write a regular expression for the language of words over \{a, b, c\} where the number of a’s is even. No restriction on b’s or c’s.

- Even number of a’s: \((aa)^*\).
- We can have any number of b’s and c’s in between each a and at the start and end of the word.
- Solution: \((b|c)^*(a(b|c)^*)^*\), or \(((b|c)^*a(b|c)^*a)^*(b|c)^*\).
- Note: \(((b|c)^*a(b|c)^*a(b|c)^*)^*\) is incorrect (e.g., doesn’t match bc).
• Write a regular expression for the language of words over \{a, b, c\} where the number of a’s is even. No restriction on b’s or c’s.

• Even number of a’s: \((aa)^*\).

• We can have any number of b’s and c’s in between each a and at the start and end of the word.

• Solution: \((b|c)^*(a(b|c)^*a(b|c)^*)^*, or \(((b|c)^*a(b|c)^*a)^*(b|c)^*\).

• Note: \(((b|c)^*a(b|c)^*a(b|c)^*)^* is incorrect (e.g., doesn’t match bc).

• Write a regular expression for the language \{a^m b^n : m, n \in \mathbb{N}, m + n is even\}. 
Regular Expressions

• Write a regular expression for the language of words over \{a, b, c\} where the number of a’s is even. No restriction on b’s or c’s.
• Even number of a’s: \((aa)^*\).
• We can have any number of b’s and c’s in between each a and at the start and end of the word.
• Solution: \((b|c)^*(a(b|c)^*a(b|c)^*)^*, \text{ or } ((b|c)^*a(b|c)^*a)^*(b|c)^*\).
• Note: \(((b|c)^*a(b|c)^*a(b|c)^*)^* \text{ is incorrect (e.g., doesn't match } bc)\).
• Write a regular expression for the language \(\{a^mb^n : m, n \in \mathbb{N}, m + n \text{ is even}\}\).
• Notice that for \(m + n\) to be even, either \(m\) and \(n\) are both even, or \(m\) and \(n\) are both odd.
Regular Expressions

- Write a regular expression for the language of words over \(\{a, b, c\}\) where the number of \(a\)'s is even. No restriction on \(b\)'s or \(c\)'s.
  - Even number of \(a\)'s: \((aa)^*\).
  - We can have any number of \(b\)'s and \(c\)'s in between each \(a\) and at the start and end of the word.
  - Solution: \((b|c)^*(a(b|c)^*a(b|c)^*)^*\), or \(((b|c)^*a(b|c)^*a)^*(b|c)^*\).
  - Note: \(((b|c)^*a(b|c)^*a)^*\) is incorrect (e.g., doesn't match \(bc\)).
- Write a regular expression for the language \(\{a^m b^n : m, n \in \mathbb{N}, m + n \text{ is even}\}\).
  - Notice that for \(m + n\) to be even, either \(m\) and \(n\) are both even, or \(m\) and \(n\) are both odd.
  - Let's tackle these cases separately.
Regular Expressions

- Write a regular expression for the language of words over \{a, b, c\} where the number of a’s is even. No restriction on b’s or c’s.
  - Even number of a’s: \((aa)^*\).
  - We can have any number of b’s and c’s in between each a and at the start and end of the word.
  - Solution: \((b|c)^*(a(b|c)^*a(b|c)^*)^*, or \(((b|c)^*a(b|c)^*a)^*(b|c)^*\).
  - Note: \(((b|c)^*a(b|c)^*a)^*\) is incorrect (e.g., doesn’t match bc).
- Write a regular expression for the language \(\{a^m b^n : m, n \in \mathbb{N}, m + n \text{ is even}\}\).
  - Notice that for \(m + n\) to be even, either \(m\) and \(n\) are both even, or \(m\) and \(n\) are both odd.
  - Let’s tackle these cases separately.
  - Even number of a’s followed by even number of b’s: \((aa)^*(bb)^*\)
• Write a regular expression for the language of words over \{a, b, c\} where the number of a’s is even. No restriction on b’s or c’s.

  • Even number of a’s: \((aa)^*\).

  • We can have any number of b’s and c’s in between each a and at the start and end of the word.

  • Solution: \((b|c)^* (a(b|c)^*a(b|c)^*)^*\), or \(((b|c)^*a(b|c)^*a)^*(b|c)^*\).

  • Note: \(((b|c)^*a(b|c)^*a)^*\) is incorrect (e.g., doesn’t match bc).

• Write a regular expression for the language \(\{a^m b^n : m, n \in \mathbb{N}, m + n \text{ is even}\}\).

  • Notice that for \(m + n\) to be even, either \(m\) and \(n\) are both even, or \(m\) and \(n\) are both odd.

  • Let’s tackle these cases separately.

  • Even number of a’s followed by even number of b’s: \((aa)^*(bb)^*\)

  • Odd number of a’s: \(a(aa)^*\)
• Write a regular expression for the language of words over \( \{a, b, c\} \) where the number of \( a \)'s is even. No restriction on \( b \)'s or \( c \)'s.
• Even number of \( a \)'s: \((aa)^*\).
• We can have any number of \( b \)'s and \( c \)'s in between each \( a \) and at the start and end of the word.
• Solution: \((b|c)^*(a(b|c)^*a(b|c)^*)^*, \text{ or } ((b|c)^*a(b|c)^*a)^*(b|c)^*\).
• Note: \(((b|c)^*a(b|c)^*a(b|c)^*)^* \text{ is incorrect (e.g., doesn’t match } bc)\).
• Write a regular expression for the language \( \{a^m b^n : m, n \in \mathbb{N}, m + n \text{ is even}\} \).
• Notice that for \( m + n \) to be even, either \( m \) and \( n \) are both even, or \( m \) and \( n \) are both odd.
• Let's tackle these cases separately.
• Even number of \( a \)'s followed by even number of \( b \)'s: \((aa)^*(bb)^*\)
• Odd number of \( a \)'s: \(a(aa)^*\)
• Odd number of \( a \)'s followed by odd number of \( b \)'s: \(a(aa)^*b(bb)^*\)
Regular Expressions

• Write a regular expression for the language of words over \{a, b, c\} where the number of a’s is even. No restriction on b’s or c’s.
• Even number of a’s: \((aa)^*\).
• We can have any number of b’s and c’s in between each a and at the start and end of the word.
• Solution: \((b|c)^*(a(b|c)^*a(b|c)^*)^*,\) or \(((b|c)^*a(b|c)^*a)^*(b|c)^*\).
• Note: \(((b|c)^*a(b|c)^*a)^*\) is incorrect (e.g., doesn’t match bc).
• Write a regular expression for the language \(\{a^m b^n : m, n \in \mathbb{N}, m + n \text{ is even}\}\).
• Notice that for \(m + n\) to be even, either \(m\) and \(n\) are both even, or \(m\) and \(n\) are both odd.
• Let’s tackle these cases separately.
• Even number of a’s followed by even number of b’s: \((aa)^*(bb)^*\)
• Odd number of a’s: \(a(aa)^*\)
• Odd number of a’s followed by odd number of b’s: \(a(aa)^*b(bb)^*\)
• Putting it all together: \((aa)^*(bb)^*|a(aa)^*b(bb)^*\).
• Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).
Regular Expressions

- Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).
- Excluding substrings in regular expressions is hard.
- Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).
- Excluding substrings in regular expressions is hard.
- Sometimes it helps to look at a DFA.
• Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).
• Excluding substrings in regular expressions is hard.
• Sometimes it helps to look at a DFA.

\[
\begin{array}{c}
0 \\
\downarrow a \quad b \\
1 \\
\end{array}
\]

• We can split this regular expression into two parts \( R \) and \( S \):
Regular Expressions

- Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).
- Excluding substrings in regular expressions is hard.
- Sometimes it helps to look at a DFA.

\[
\begin{array}{c}
0 \\
\downarrow b \\
1 \\
\downarrow a \\
\end{array}
\]

- We can split this regular expression into two parts \( R \) and \( S \):
  - \( R \) matches paths in the DFA that start at 0 and end at 0.
Regular Expressions

• Write an expression for words over \{a, b\} that do not contain $aa$.
• Excluding substrings in regular expressions is hard.
• Sometimes it helps to look at a DFA.

We can split this regular expression into two parts $R$ and $S$:
• $R$ matches paths in the DFA that start at 0 and end at 0.
• $S$ matches paths in the DFA that start at 0 and end at 1.
- Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).
- Excluding substrings in regular expressions is hard.
- Sometimes it helps to look at a DFA.

\[
\begin{array}{c}
0 \quad \quad \quad \quad 1 \\
\text{\begin{tikzpicture}
\node (0) at (0,0) [circle,draw] {$0$};
\node (1) at (1,0) [circle,draw] {$1$};
\draw [->, thick] (0) edge [loop above] node {$b$} (0);
\draw [->, thick] (0) edge [bend left] node {$a$} (1);
\draw [->, thick] (1) edge [bend left] node {$b$} (0);
\end{tikzpicture}}
\end{array}
\]

- We can split this regular expression into two parts \( R \) and \( S \):
  - \( R \) matches paths in the DFA that start at 0 and end at 0.
  - \( S \) matches paths in the DFA that start at 0 and end at 1.
- Then \( R(\varepsilon|S) \) is an expression for the language we want, since we can end at state 0 (\( R \)) or end at state 1 (\( RS \)).
• Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).

\[ \begin{align*}
\text{0} & \quad a \quad \text{1} \\
\text{b} & \quad \text{b}
\end{align*} \]

• We can split this regular expression into two parts \( R \) and \( S \):
  • \( R \) matches paths in the DFA that start at 0 and end at 0.
  • \( S \) matches paths in the DFA that start at 0 and end at 1.

• Then \( R(\varepsilon|S) \) is an expression for the language we want, since we can end at state 0 (\( R \)) or end at state 1 (\( RS \)).
• Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).

We can split this regular expression into two parts \( R \) and \( S \):

• \( R \) matches paths in the DFA that start at 0 and end at 0.
• \( S \) matches paths in the DFA that start at 0 and end at 1.

Then \( R(\varepsilon|S) \) is an expression for the language we want, since we can end at state 0 (\( R \)) or end at state 1 (\( RS \)).

To figure out \( R \), look at paths with different numbers of \( a \)'s:
• Write an expression for words over \{a, b\} that do not contain \textit{aa}.

\begin{center}
\begin{tikzpicture}[node distance=2cm,auto,>=latex]
    
    
    \node[state] (0) {0};
    \node[state] (1) [right of=0] {1};

    
    
    \path[->]
    (0) edge [loop right] node {$b$} (0)
    (0) edge [bend left] node {$a$} (1)
    (1) edge [bend left] node {$a$} (0)
    (1) edge [loop below] node {$b$} (1);

\end{tikzpicture}
\end{center}

• We can split this regular expression into two parts \textit{R} and \textit{S}:
  
  \begin{itemize}
    \item \textit{R} matches paths in the DFA that start at 0 and end at 0.
    \item \textit{S} matches paths in the DFA that start at 0 and end at 1.
  \end{itemize}

• Then \textit{R(ε|S)} is an expression for the language we want, since we can end at state 0 (\textit{R}) or end at state 1 (\textit{RS}).

• To figure out \textit{R}, look at paths with different numbers of \textit{a}'s:
  
  \begin{itemize}
    \item Zero \textit{a}'s: \textit{b}*
  \end{itemize}
Write an expression for words over \{a, b\} that do not contain \textit{aa}.

We can split this regular expression into two parts \textit{R} and \textit{S}:

- \textit{R} matches paths in the DFA that start at 0 and end at 0.
- \textit{S} matches paths in the DFA that start at 0 and end at 1.

Then \textit{R(\varepsilon|S)} is an expression for the language we want, since we can end at state 0 (\textit{R}) or end at state 1 (\textit{RS}).

To figure out \textit{R}, look at paths with different numbers of \textit{a}'s:

- Zero \textit{a}'s: \textit{b}^*
- One \textit{a}: \textit{b}^*abb^*
• Write an expression for words over \{a, b\} that do not contain aa.

\[
\begin{align*}
0 & \quad \text{(State 0)} \\
1 & \quad \text{(State 1)}
\end{align*}
\]

\[
\xrightarrow{a} \\
\xrightarrow{b}
\]

• We can split this regular expression into two parts \(R\) and \(S\):
  • \(R\) matches paths in the DFA that start at 0 and end at 0.
  • \(S\) matches paths in the DFA that start at 0 and end at 1.

• Then \(R(\epsilon|S)\) is an expression for the language we want, since we can end at state 0 (\(R\)) or end at state 1 (\(RS\)).

• To figure out \(R\), look at paths with different numbers of a’s:
  • Zero a’s: \(b^*\)
  • One a: \(b^*abb^*\)
  • Two a’s: \(b^*abb^*abb^*\)
● Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).

\[
\begin{align*}
\text{0} & \quad \text{1} \\
\text{b} & \quad \text{a} \\
\end{align*}
\]

● We can split this regular expression into two parts \( R \) and \( S \):
  ● \( R \) matches paths in the DFA that start at 0 and end at 0.
  ● \( S \) matches paths in the DFA that start at 0 and end at 1.

● Then \( R(\varepsilon|S) \) is an expression for the language we want, since we can end at state 0 (\( R \)) or end at state 1 (\( RS \)).

● To figure out \( R \), look at paths with different numbers of \( a \)'s:
  ● Zero \( a \)'s: \( b^* \)
  ● One \( a \): \( b^*abb^* \)
  ● Two \( a \)'s: \( b^*abb^*abb^* \)
  ● Three \( a \)'s: \( b^*abb^*abb^*abb^* \)
• Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).

\[
\begin{align*}
\text{0} & \xrightarrow{a} \text{1} \\
\text{0} & \xrightarrow{b} \text{0}
\end{align*}
\]

• We can split this regular expression into two parts \( R \) and \( S \):
  - \( R \) matches paths in the DFA that start at 0 and end at 0.
  - \( S \) matches paths in the DFA that start at 0 and end at 1.

• Then \( R(\varepsilon|S) \) is an expression for the language we want, since we can end at state 0 (\( R \)) or end at state 1 (\( RS \)).

• To figure out \( R \), look at paths with different numbers of \( a \)'s:
  - Zero \( a \)'s: \( b^* \)
  - One \( a \): \( b^*abb^* \)
  - Two \( a \)'s: \( b^*abb^*abb^* \)
  - Three \( a \)'s: \( b^*abb^*abb^*abb^* \)

• We notice a pattern: \( R = b^*(abb^*)^* \).
Regular Expressions

- Write an expression for words over \{a, b\} that do not contain \textit{aa}.

\[R\]

- We can split this regular expression into two parts \(R\) and \(S\):
  - \(R\) matches paths in the DFA that start at 0 and end at 0.
  - \(S\) matches paths in the DFA that start at 0 and end at 1.
- Then \(R(\varepsilon|S)\) is an expression for the language we want, since we can end at state 0 (\(R\)) or end at state 1 (\(RS\)).
- We have \(R = b^*(abb^*)^*\).
- Write an expression for words over \{a, b\} that do not contain \textit{aa}.

- We can split this regular expression into two parts \(R\) and \(S\):
  - \(R\) matches paths in the DFA that start at 0 and end at 0.
  - \(S\) matches paths in the DFA that start at 0 and end at 1.

- Then \(R(\varepsilon | S)\) is an expression for the language we want, since we can end at state 0 (\(R\)) or end at state 1 (\(RS\)).

- We have \(R = b^*(abb^*)^*\).

- For \(S\), notice we can exclude paths which cycle back to 0 before reaching 1, since these are already covered by \(R\).
Regular Expressions

- Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).

\[
\begin{array}{c}
0 \quad 1 \\
\downarrow & \downarrow \\
a & b \\
b & b
\end{array}
\]

- We can split this regular expression into two parts \( R \) and \( S \):
  - \( R \) matches paths in the DFA that start at 0 and end at 0.
  - \( S \) matches paths in the DFA that start at 0 and end at 1.
- Then \( R(\varepsilon|S) \) is an expression for the language we want, since we can end at state 0 (\( R \)) or end at state 1 (\( RS \)).
- We have \( R = b^*(abb^*)^* \).
- For \( S \), notice we can exclude paths which cycle back to 0 before reaching 1, since these are already covered by \( R \).
- So we simply need to consider paths which go from 0 to 1 and never return to 0.
Write an expression for words over \{a, b\} that do not contain \textit{aa}.

\[ 0 \xrightarrow{a} 1 \xrightarrow{b} 0 \]

We can split this regular expression into two parts \( R \) and \( S \):
- \( R \) matches paths in the DFA that start at 0 and end at 0.
- \( S \) matches paths in the DFA that start at 0 and end at 1.

Then \( R(\varepsilon|S) \) is an expression for the language we want, since we can end at state 0 (\( R \)) or end at state 1 (\( RS \)).

We have \( R = b^*(abb^*)^* \).

For \( S \), notice we can exclude paths which cycle back to 0 before reaching 1, since these are already covered by \( R \).

So we simply need to consider paths which go from 0 to 1 and never return to 0.

The only such path is \( a \)! So \( S = a \).
• Write an expression for words over \( \{a, b\} \) that do not contain \( aa \).

\[
\begin{align*}
\text{\begin{tikzpicture}[baseline=(current bounding box.center)]
  \node (0) at (0,0) [shape=circle] {0};
  \node (1) at (1,0) [shape=circle] {1};
  \draw[->] (0) edge [loop above] node {b} (0);
  \draw[->] (0) edge [bend left=45] node {a} (1);
  \draw[->] (1) edge [loop right] node {b} (1);
\end{tikzpicture}}
\end{align*}
\]

• We can split this regular expression into two parts \( R \) and \( S \):
  • \( R \) matches paths in the DFA that start at 0 and end at 0.
  • \( S \) matches paths in the DFA that start at 0 and end at 1.

• Then \( R(\varepsilon|S) \) is an expression for the language we want, since we can end at state 0 (\( R \)) or end at state 1 (\( RS \)).

• We have \( R = b^*(abb^*)^* \).

• For \( S \), notice we can exclude paths which cycle back to 0 before reaching 1, since these are already covered by \( R \).

• So we simply need to consider paths which go from 0 to 1 and never return to 0.

• The only such path is \( a \)! So \( S = a \).

• Our final expression is \( b^*(abb^*)^*(a|\varepsilon) \).
• Give the formal definition of this DFA.
• Give the formal definition of this DFA.

A DFA is a 5-tuple \((\Sigma, Q, q_0, A, \delta)\).
Formal Definitions of Automata

- Give the formal definition of this DFA.

- A DFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$.
- $\Sigma = \{a, b, c\}$ is the alphabet.
Formal Definitions of Automata

- Give the formal definition of this DFA.

- A DFA is a 5-tuple \((\Sigma, Q, q_0, A, \delta)\).
- \(\Sigma = \{a, b, c\}\) is the alphabet.
- \(Q = \{0, 1, 2, 3, 4, \text{ERROR}\}\) is the state set.
Formal Definitions of Automata

- Give the formal definition of this DFA.

- A DFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$.
  - $\Sigma = \{a, b, c\}$ is the alphabet.
  - $Q = \{0, 1, 2, 3, 4, \text{ERROR}\}$ is the state set.
  - Why did we include ERROR? This DFA has an implicit error state it goes to when there is no transition out of a state on a given letter.
Formal Definitions of Automata

- Give the formal definition of this DFA.

A DFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$.

- $\Sigma = \{a, b, c\}$ is the alphabet.
- $Q = \{0, 1, 2, 3, 4, \text{ERROR}\}$ is the state set.
- Why did we include \text{ERROR}? This DFA has an implicit error state it goes to when there is no transition out of a state on a given letter.
- The transition function $\delta: Q \times \Sigma \rightarrow Q$ must be defined for every state and symbol in the formal definition of a DFA, so we need somewhere to go to for the missing transitions.
Formal Definitions of Automata

• Give the formal definition of this DFA.

\[
\begin{align*}
0 & \xrightarrow{b} 1 \\
1 & \xrightarrow{a} 2 \\
2 & \xrightarrow{b, c} 3 \\
3 & \xrightarrow{a} 4 \\
4 & \xrightarrow{b, c} 3
\end{align*}
\]

A DFA is a 5-tuple \((\Sigma, Q, q_0, A, \delta)\).

• \(\Sigma = \{a, b, c\}\) is the alphabet.

• \(Q = \{0, 1, 2, 3, 4, \text{ERROR}\}\) is the state set.

• Why did we include \text{ERROR}? This DFA has an implicit error state it goes to when there is no transition out of a state on a given letter.

• The transition function \(\delta: Q \times \Sigma \rightarrow Q\) must be defined for every state and symbol in the formal definition of a DFA, so we need somewhere to go to for the missing transitions.

• \(q_0 = 0\) is the initial state and \(A = \{2, 4\}\) is the accepting state set.
Formal Definitions of Automata

- Give the formal definition of this DFA.

- The transition function $\delta: Q \times \Sigma \rightarrow Q$ is defined as follows:
• Give the formal definition of this DFA.

\[
\begin{array}{c}
1 & 2 & 3 & 4 \\
b & a & b, c & b, c \\
0 & 1 & 2 & 3 \\
a & b, c & a & a \\
\end{array}
\]

• The transition function \( \delta : Q \times \Sigma \rightarrow Q \) is defined as follows:
  • Whenever there is an arrow from state \( q \) to \( q' \) labelled with symbol \( \sigma \), we define \( \delta(q, \sigma) = q' \).
Formal Definitions of Automata

- Give the formal definition of this DFA.

- The transition function $\delta : Q \times \Sigma \rightarrow Q$ is defined as follows:
  - Whenever there is an arrow from state $q$ to $q'$ labelled with symbol $\sigma$, we define $\delta(q, \sigma) = q'$.
  - Whenever there is a state $q$ and a symbol $\sigma$ such that there is no arrow out from $q$ on the letter $\sigma$, we define $\delta(q, \sigma) = \text{ERROR}$.
Formal Definitions of Automata

- Give the formal definition of this DFA.

- The transition function $\delta : Q \times \Sigma \to Q$ is defined as follows:
  - Whenever there is an arrow from state $q$ to $q'$ labelled with symbol $\sigma$, we define $\delta(q, \sigma) = q'$.
  - Whenever there is a state $q$ and a symbol $\sigma$ such that there is no arrow out from $q$ on the letter $\sigma$, we define $\delta(q, \sigma) = \text{ERROR}$.
  - For all symbols $\sigma$ we define $\delta(\text{ERROR}, \sigma) = \text{ERROR}$. 
Formal Definitions of Automata

• Give the formal definition of this DFA.

![DFA Diagram]

• The transition function $\delta: Q \times \Sigma \rightarrow Q$ is defined as follows:
  
  - Whenever there is an arrow from state $q$ to $q'$ labelled with symbol $\sigma$, we define $\delta(q, \sigma) = q'$.
  
  - Whenever there is a state $q$ and a symbol $\sigma$ such that there is no arrow out from $q$ on the letter $\sigma$, we define $\delta(q, \sigma) = \text{ERROR}$.
  
  - For all symbols $\sigma$ we define $\delta(\text{ERROR}, \sigma) = \text{ERROR}$.

• Here is the transition function in table form.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>ERROR</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>ERROR</td>
<td>3</td>
<td>ERROR</td>
<td>3</td>
<td>ERROR</td>
</tr>
<tr>
<td>c</td>
<td>ERROR</td>
<td>ERROR</td>
<td>3</td>
<td>ERROR</td>
<td>3</td>
<td>ERROR</td>
</tr>
</tbody>
</table>
• Give the formal definition of this NFA.
• Give the formal definition of this NFA.

• An NFA is a 5-tuple \((\Sigma, Q, q_0, A, \delta)\).
• Give the formal definition of this NFA.

• An NFA is a 5-tuple \((\Sigma, Q, q_0, A, \delta)\).

• \(\Sigma = \{a, b, c\}\) is the alphabet.
Formal Definitions of Automata

- Give the formal definition of this NFA.

- An NFA is a 5-tuple \((\Sigma, Q, q_0, A, \delta)\).

- \(\Sigma = \{a, b, c\}\) is the alphabet.

- \(Q = \{0, 1, 2\}\) is the state set. For NFAs and \(\varepsilon\)-NFAs we don't need error states. Instead, since the transition function produces sets, we can just produce the empty set when there are missing transitions.
• Give the formal definition of this NFA.

- An NFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$.
- $\Sigma = \{a, b, c\}$ is the alphabet.
- $Q = \{0, 1, 2\}$ is the state set. For NFAs and $\varepsilon$-NFAs we don’t need error states. Instead, since the transition function produces sets, we can just produce the empty set when there are missing transitions.
- $q_0 = 0$ is the initial state and $A = \{2\}$ is the set of accepting states.
• Give the formal definition of this NFA.

\[ \delta : Q \times \Sigma \rightarrow 2^Q \] is constructed similarly to a DFA, but the outputs are sets of states, corresponding to all the different states reached by transitions on the given letter.

\[
\begin{array}{c|ccc}
\delta & 0 & 1 & 2 \\
\hline
a & \{1\} & \{2\} & \{0\} \\
b & \{0\} & \{2\} & \{1\} \\
c & \{0, 1\} & \emptyset & \{2\} \\
\end{array}
\]
• Give the formal definition of this $\varepsilon$-NFA.
• Give the formal definition of this $\varepsilon$-NFA.

• An $\varepsilon$-NFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$.
• Give the formal definition of this $\varepsilon$-NFA.

$$\varepsilon$$-NFA is a 5-tuple $$(\Sigma, Q, q_0, A, \delta)$$. 

- $$\Sigma = \{a, b\}$$
- $$Q = \{0, 1, 2, 3\}$$
- $$q_0 = 0$$
- $$A = \{0, 3\}$$
- Give the formal definition of this $\varepsilon$-NFA.

- An $\varepsilon$-NFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$.
- $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$ and $A = \{0, 3\}$.
- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ is the transition function:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$\emptyset$</td>
<td>${2}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$b$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>${1}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
</tr>
</tbody>
</table>
Converting an $\varepsilon$-NFA to a DFA

- Convert this $\varepsilon$-NFA to a DFA.
- $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$ and $A = \{0, 3\}$.
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ is the transition function:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$\emptyset$</td>
<td>${2}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$b$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>${1}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
</tr>
</tbody>
</table>

- The states of the DFA are sets of states which are reachable in $\varepsilon$-NFA from the initial state $\{0\}$.
- We will construct the DFA transition table by starting from set $\{0\}$ and successively finding reachable states.
- Every time we add a set to the table we take the $\varepsilon$-closure first, that is, we find all states which are reachable from the set via one or more $\varepsilon$-transitions and add them in.
- This includes the initial set $\{0\}$ whose $\varepsilon$-closure is $\{0, 1\}$!
Converting an $\varepsilon$-NFA to a DFA

- Convert this $\varepsilon$-NFA to a DFA.
- $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$ and $A = \{0, 3\}$.
- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ is the transition function:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$\emptyset$</td>
<td>{2}</td>
<td>{3}</td>
<td>{1}</td>
</tr>
<tr>
<td>$b$</td>
<td>$\emptyset$</td>
<td>{1}</td>
<td>{3}</td>
<td>{1}</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>{1}</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>{1}</td>
</tr>
</tbody>
</table>

- Here is the DFA transition table:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>${0, 1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td></td>
</tr>
</tbody>
</table>
Converting an $\varepsilon$-NFA to a DFA

- Convert this $\varepsilon$-NFA to a DFA.
- $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$ and $A = \{0, 3\}$.
- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ is the transition function:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$\emptyset$</td>
<td>${2}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$b$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>${1}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
</tr>
</tbody>
</table>

- Here is the DFA transition table:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>${0, 1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>${2}$</td>
</tr>
<tr>
<td>$b$</td>
<td>${1}$</td>
</tr>
</tbody>
</table>
Converting an $\varepsilon$-NFA to a DFA

• Convert this $\varepsilon$-NFA to a DFA.

• $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$ and $A = \{0, 3\}$.

• $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is the transition function:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$\emptyset$</td>
<td>${2}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$b$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>${1}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
</tr>
</tbody>
</table>

• Here is the DFA transition table:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>${0, 1}$</th>
<th>${2}$</th>
<th>${1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>${2}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>${1}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Converting an \( \varepsilon \)-NFA to a DFA

- Convert this \( \varepsilon \)-NFA to a DFA.
- \( \Sigma = \{a, b\}, \ Q = \{0, 1, 2, 3\}, \ q_0 = 0 \) and \( A = \{0, 3\} \).
- \( \delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q \) is the transition function:

\[
\begin{array}{|c|c|c|c|}
\hline
\delta & 0 & 1 & 2 & 3 \\
\hline
a & \emptyset & \{2\} & \{3\} & \{1\} \\
b & \emptyset & \{1\} & \{3\} & \{1\} \\
\varepsilon & \{1\} & \emptyset & \emptyset & \{1\} \\
\hline
\end{array}
\]

- Here is the DFA transition table:

\[
\begin{array}{|c|c|c|}
\hline
\delta & \{0, 1\} & \{2\} & \{1\} \\
\hline
a & \{2\} & \{1, 3\} & \{1\} \\
b & \{1\} & \{1, 3\} & \{1\} \\
\hline
\end{array}
\]

We must include 1 because there is a \( \varepsilon \)-transition from 3 to 1.
Converting an $\varepsilon$-NFA to a DFA

- Convert this $\varepsilon$-NFA to a DFA.

- $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$ and $A = \{0, 3\}$.

- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ is the transition function:

$$
\begin{array}{c|ccc}
\delta & 0 & 1 & 2 & 3 \\
\hline
a & \emptyset & \{2\} & \{3\} & \{1\} \\
b & \emptyset & \{1\} & \{3\} & \{1\} \\
\varepsilon & \{1\} & \emptyset & \emptyset & \{1\} \\
\end{array}
$$

- Here is the DFA transition table:

$$
\begin{array}{c|c|c|c}
\delta & \{0, 1\} & \{2\} & \{1\} \\
\hline
a & \{2\} & \{1, 3\} & \{2\} \\
b & \{1\} & \{1, 3\} & \{1\} \\
\end{array}
$$
Converting an ε-NFA to a DFA

- Convert this ε-NFA to a DFA.
- \( \Sigma = \{a, b\}, \ Q = \{0, 1, 2, 3\}, \ q_0 = 0 \) and \( A = \{0, 3\} \).
- \( \delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q \) is the transition function:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>( \emptyset )</td>
<td>{2}</td>
<td>{3}</td>
<td>{1}</td>
</tr>
<tr>
<td>b</td>
<td>( \emptyset )</td>
<td>{1}</td>
<td>{3}</td>
<td>{1}</td>
</tr>
<tr>
<td>\varepsilon</td>
<td>{1}</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>{1}</td>
</tr>
</tbody>
</table>

- Here is the DFA transition table:

<table>
<thead>
<tr>
<th></th>
<th>{0, 1}</th>
<th>{2}</th>
<th>{1}</th>
<th>{1, 3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{2}</td>
<td>{1, 3}</td>
<td>{2}</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>{1}</td>
<td>{1, 3}</td>
<td>{1}</td>
<td></td>
</tr>
</tbody>
</table>
Converting an \( \varepsilon \)-NFA to a DFA

- Convert this \( \varepsilon \)-NFA to a DFA.
- \( \Sigma = \{a, b\} \), \( Q = \{0, 1, 2, 3\} \), \( q_0 = 0 \) and \( A = \{0, 3\} \).
- \( \delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q \) is the transition function:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>( \emptyset )</td>
<td>{2}</td>
<td>{3}</td>
<td>{1}</td>
</tr>
<tr>
<td>b</td>
<td>( \emptyset )</td>
<td>{1}</td>
<td>{3}</td>
<td>{1}</td>
</tr>
<tr>
<td>\varepsilon</td>
<td>{1}</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>{1}</td>
</tr>
</tbody>
</table>

- Here is the DFA transition table:

<table>
<thead>
<tr>
<th></th>
<th>{0, 1}</th>
<th>{2}</th>
<th>{1}</th>
<th>{1, 3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{2}</td>
<td>{1, 3}</td>
<td>{2}</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>b</td>
<td>{1}</td>
<td>{1, 3}</td>
<td>{1}</td>
<td>{1}</td>
</tr>
</tbody>
</table>
Converting an \( \varepsilon \)-NFA to a DFA

- Convert this \( \varepsilon \)-NFA to a DFA.
- \( \Sigma = \{a, b\} \), \( Q = \{0, 1, 2, 3\} \), \( q_0 = 0 \) and \( A = \{0, 3\} \).
- \( \delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q \) is the transition function:

\[
\begin{array}{c|cccc}
\delta & 0 & 1 & 2 & 3 \\
\hline
a & \emptyset & \{2\} & \{3\} & \{1\} \\
b & \emptyset & \{1\} & \{3\} & \{1\} \\
\varepsilon & \{1\} & \emptyset & \emptyset & \{1\} \\
\end{array}
\]

- Here is the DFA transition table:

\[
\begin{array}{c|ccccc}
\delta & \{0, 1\} & \{2\} & \{1\} & \{1, 3\} & \{1, 2\} \\
\hline
a & \{2\} & \{1, 3\} & \{2\} & \{1, 2\} & \\
b & \{1\} & \{1, 3\} & \{1\} & \{1\} & \\
\end{array}
\]
Converting an $\varepsilon$-NFA to a DFA

- Convert this $\varepsilon$-NFA to a DFA.
  - $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$ and $A = \{0, 3\}$.
  - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ is the transition function:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$\emptyset$</td>
<td>${2}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$b$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>${1}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
</tr>
</tbody>
</table>

- Here is the DFA transition table:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>${0, 1}$</th>
<th>${2}$</th>
<th>${1}$</th>
<th>${1, 3}$</th>
<th>${1, 2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>${2}$</td>
<td>${1, 3}$</td>
<td>${2}$</td>
<td>${1, 2}$</td>
<td>${1, 2, 3}$</td>
</tr>
<tr>
<td>$b$</td>
<td>${1}$</td>
<td>${1, 3}$</td>
<td>${1}$</td>
<td>${1}$</td>
<td>${1, 3}$</td>
</tr>
</tbody>
</table>

We must include 1 because there is a $\varepsilon$-transition from 3 to 1.
Converting an $\varepsilon$-NFA to a DFA

- Convert this $\varepsilon$-NFA to a DFA.
- $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$ and $A = \{0, 3\}$.
- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ is the transition function:

  $\begin{array}{c|cccc}
  \delta & 0 & 1 & 2 & 3 \\
  \hline
  a & \emptyset & \{2\} & \{3\} & \{1\} \\
  b & \emptyset & \{1\} & \{3\} & \{1\} \\
  \varepsilon & \{1\} & \emptyset & \emptyset & \{1\} \\
  \end{array}$

- Here is the DFA transition table:

  $\begin{array}{c|cccc}
  \delta & \{0, 1\} & \{2\} & \{1\} & \{1, 3\} & \{1, 2\} & \{1, 2, 3\} \\
  \hline
  a & \{2\} & \{1, 3\} & \{2\} & \{1, 2\} & \{1, 2, 3\} & \\
  b & \{1\} & \{1, 3\} & \{1\} & \{1\} & \{1, 3\} & \\
  \end{array}$
Convert this $\varepsilon$-NFA to a DFA.

- $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$ and $A = \{0, 3\}$.
- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ is the transition function:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$\emptyset$</td>
<td>${2}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$b$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>${1}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
</tr>
</tbody>
</table>

Here is the DFA transition table:

<table>
<thead>
<tr>
<th></th>
<th>${0, 1}$</th>
<th>${2}$</th>
<th>${1}$</th>
<th>${1, 3}$</th>
<th>${1, 2}$</th>
<th>${1, 2, 3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>${2}$</td>
<td>${1, 3}$</td>
<td>${2}$</td>
<td>${1, 2}$</td>
<td>${1, 2, 3}$</td>
<td>${1, 2, 3}$</td>
</tr>
<tr>
<td>$b$</td>
<td>${1}$</td>
<td>${1, 3}$</td>
<td>${1}$</td>
<td>${1}$</td>
<td>${1, 3}$</td>
<td>${1, 3}$</td>
</tr>
</tbody>
</table>

We’re done because we didn’t reach any new states. State $\{0, 1\}$ is initial and states $\{0, 1\}$, $\{1, 3\}$ and $\{1, 2, 3\}$ are accepting since they contain accepting states.
Converting an $\varepsilon$-NFA to a DFA

- Convert this $\varepsilon$-NFA to a DFA.
- $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $q_0 = 0$ and $A = \{0, 3\}$.
- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is the transition function:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$\emptyset$</td>
<td>${2}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$b$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
<td>${3}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>${1}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${1}$</td>
</tr>
</tbody>
</table>

- Here is the DFA transition table:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>${0, 1}$</th>
<th>${2}$</th>
<th>${1}$</th>
<th>${1, 3}$</th>
<th>${1, 2}$</th>
<th>${1, 2, 3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>${2}$</td>
<td>${1, 3}$</td>
<td>${2}$</td>
<td>${1, 2}$</td>
<td>${1, 2, 3}$</td>
<td>${1, 2, 3}$</td>
</tr>
<tr>
<td>$b$</td>
<td>${1}$</td>
<td>${1, 3}$</td>
<td>${1}$</td>
<td>${1}$</td>
<td>${1, 3}$</td>
<td>${1, 3}$</td>
</tr>
</tbody>
</table>

Drawing a diagram of this DFA is left as an exercise. State $\{0, 1\}$ is initial and states $\{0, 1\}$, $\{1, 3\}$ and $\{1, 2, 3\}$ are accepting since they contain accepting states.
Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$.
Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$.

The word $0w0$ cannot be scanned into tokens from $L$. 
Scanning

- Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$. 

- The word $0w0$ cannot be scanned into tokens from $L$. 
- What about $0xbox$?
Let the set of tokens \( L \) be the language recognized by the following DFA. The alphabet is \( \{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\} \).

The word 0w0 cannot be scanned into tokens from \( L \).

What about 0xbox?

Yes, it can be scanned as \underline{NUM} 0 \underline{ID} \underline{xbox}.
Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$.

Can $0\text{xbox}$ be scanned with Maximal Munch?
Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$.

Can $0\text{xbox}$ be scanned with Maximal Munch?

No. Maximal Munch will find the token $\text{HEX 0xb}$, then get stuck while trying to scan $0x$. 
Let the set of tokens \( L \) be the language recognized by the following DFA. The alphabet is \( \{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\} \).

Maximal Munch can scan 0xyzzy. It will read 0x, get stuck because there is no transition on y, then backtrack and output NUM 0. It will then scan xyzzy successfully and output ID xyzzy.
Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$.

Can $0\text{xyzzy}$ be scanned with Simplified Maximal Munch?
Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$.

Can $0\text{xyzzy}$ be scanned with Simplified Maximal Munch?

No. It will get stuck in a non-accepting state after reading $0x$, and then simply produce an error and stop. Simplified Maximal Munch does no backtracking.
Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$.

The word 00x00 has two ways to scan it:
Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$.

The word 00x00 has two ways to scan it:
- NUM 0 HEX 0x0 NUM 0
Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$.

The word $00x00$ has two ways to scan it:
- NUM 0 HEX 0x0 NUM 0
- NUM 0 HEX 0x00
Scanning

- Let the set of tokens $L$ be the language recognized by the following DFA. The alphabet is $\{0, 1, \ldots, 9\} \cup \{a, b, c, \ldots, x, y, z\}$.

- The word 00x00 has two ways to scan it:
  - $\underline{NUM} \ 0 \ \underline{HEX} \ 0x0 \ \underline{NUM} \ 0$
  - $\underline{NUM} \ 0 \ \underline{HEX} \ 0\times00$

- Maximal Munch and Simplified Maximal Munch will both return the second one, since they always “munch” the longest possible token.
Here are some basic context-free grammars you should know.
Here are some basic context-free grammars you should know.

\[ \{a^n b^n : n \in \mathbb{N}\} : \quad S \rightarrow aSb \mid \varepsilon \]
Here are some basic context-free grammars you should know.

- \( \{ a^n b^n : n \in \mathbb{N} \} \): \( S \to aSb | \epsilon \)
- Palindromes over \( \{ a, b \} \): \( S \to aSa | bSb | a | b | \epsilon \)
Here are some basic context-free grammars you should know.

1. \( \{a^n b^n : n \in \mathbb{N}\} \): \( S \rightarrow aSb | \varepsilon \)

2. Palindromes over \( \{a, b\} \): \( S \rightarrow aSa | bSb | a | b | \varepsilon \)

3. Sequences of balanced parentheses over \( \{(,\}\} \): \( S \rightarrow (S) | SS | \varepsilon \)
   Alternative grammar: \( S \rightarrow (S)S | \varepsilon \)
Here are some basic context-free grammars you should know.

- \( \{a^n b^n : n \in \mathbb{N}\} \): \( S \rightarrow aSb \mid \varepsilon \)
- Palindromes over \( \{a, b\} \): \( S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon \)
- Sequences of balanced parentheses over \( \{(, )\} \): \( S \rightarrow (S) \mid SS \mid \varepsilon \)
  Alternative grammar: \( S \rightarrow (S)S \mid \varepsilon \)
Constructing Context-Free Grammars

- Here are some basic context-free grammars you should know.
- \( \{a^n b^n : n \in \mathbb{N}\} \): \( S \rightarrow aSb \mid \varepsilon \)
- Palindromes over \( \{a, b\} \): \( S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon \)
- Sequences of balanced parentheses over \( \{(, )\} \): \( S \rightarrow (S) \mid SS \mid \varepsilon \)
  Alternative grammar: \( S \rightarrow (S)S \mid \varepsilon \)
  The key is your grammar should support nesting and sequencing. You should be able to nest sequences of balanced parentheses within each other to form words like \((())\), and you should also be able to put them side-by-side to form words like \(()()\).
Here are some basic context-free grammars you should know.

- \( \{ a^n b^n : n \in \mathbb{N} \} : \ S \rightarrow aSb \ | \ \varepsilon \)
- Palindromes over \( \{ a, b \} \) : \ S \rightarrow aSa \ | \ bSb \ | \ a \ | \ b \ | \ \varepsilon \)
- Sequences of balanced parentheses over \( \{ (, ) \} \) : \ S \rightarrow (S) \ | \ SS \ | \ \varepsilon \)
  
  Alternative grammar: \( S \rightarrow (S)S \ | \ \varepsilon \)

The key is your grammar should support nesting and sequencing. You should be able to nest sequences of balanced parentheses within each other to form words like \( (()) \), and you should also be able to put them side-by-side to form words like \( ()() \).

- Words with an equal number of \( a \)'s and \( b \)'s over \( \{ a, b \} \):

  \[
  S \rightarrow aSb \ | \ bSa \ | \ SS \ | \ \varepsilon
  \]

  Alternative grammar: \( S \rightarrow aSbS \ | \ bSaS \ | \ \varepsilon \)
Here are some basic context-free grammars you should know.

- \( \{a^n b^n : n \in \mathbb{N}\} \): \( S \rightarrow aSb \mid \varepsilon \)

- Palindromes over \( \{a, b\} \): \( S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon \)

- Sequences of balanced parentheses over \( \{(,),\}\) \( S \rightarrow (S) \mid SS \mid \varepsilon \)
  Alternative grammar: \( S \rightarrow (S)S \mid \varepsilon \)
  The key is your grammar should support nesting and sequencing. You should be able to nest sequences of balanced parentheses within each other to form words like (()), and you should also be able to put them side-by-side to form words like ()().

- Words with an equal number of a’s and b’s over \( \{a, b\} \):

  \[
  S \rightarrow aSb \mid bSa \mid SS \mid \varepsilon
  \]

  Alternative grammar: \( S \rightarrow aSbS \mid bSaS \mid \varepsilon \)
Here are some basic context-free grammars you should know.

- \( \{a^n b^n : n \in \mathbb{N}\} \) : \( S \rightarrow aSb | \varepsilon \)
- Palindromes over \( \{a, b\} \) : \( S \rightarrow aSa | bSb | a | b | \varepsilon \)
- Sequences of balanced parentheses over \( \{(, )\} \) : \( S \rightarrow (S) | SS | \varepsilon \)
  Alternative grammar: \( S \rightarrow (S)S | \varepsilon \)
  The key is your grammar should support nesting and sequencing. You should be able to nest sequences of balanced parentheses within each other to form words like \((())\), and you should also be able to put them side-by-side to form words like \((())\).
- Words with an equal number of a’s and b’s over \( \{a, b\} \) :

\[
S \rightarrow aSb | bSa | SS | \varepsilon
\]

Alternative grammar: \( S \rightarrow aSbS | bSaS | \varepsilon \)
Again the key is to support nesting and sequencing of these words.
Here are some basic context-free grammars you should know.

- \( \{ a^n b^n : n \in \mathbb{N} \} \): \( S \rightarrow aSb | \varepsilon \)
- Palindromes over \( \{a, b\} \): \( S \rightarrow aSa | bSb | a | b | \varepsilon \)
- Sequences of balanced parentheses over \( \{(, )\} \): \( S \rightarrow (S) | SS | \varepsilon \)
  
  Alternative grammar: \( S \rightarrow (S)S | \varepsilon \)
  
  The key is your grammar should support **nesting** and **sequencing**. You should be able to nest sequences of balanced parentheses within each other to form words like \(((())\), and you should also be able to put them side-by-side to form words like \((())\).

- Words with an equal number of a’s and b’s over \( \{a, b\} \):

  \[
  S \rightarrow aSb | bSa | SS | \varepsilon
  \]

  Alternative grammar: \( S \rightarrow aSbS | bSaS | \varepsilon \)
  
  Again the key is to support **nesting** and **sequencing** of these words.

- Don’t just memorize these – try to understand how they work.
• Write a context-free grammar for palindromes over \( \{a, b\} \) at least three letters long.
• Write a context-free grammar for palindromes over \( \{a, b\} \) at least three letters long.
• You can just use intuition to solve this problem, but there is also a more structured approach.
• Write a context-free grammar for palindromes over \{a, b\} at least three letters long.
• You can just use intuition to solve this problem, but there is also a more structured approach.
• The structured approach sometimes produces larger or more complicated grammars than an intuitive approach, but may be useful if you are stuck.
• Write a context-free grammar for palindromes over \( \{a, b\} \) at least three letters long.

• You can just use intuition to solve this problem, but there is also a more structured approach.

• The structured approach sometimes produces larger or more complicated grammars than an intuitive approach, but may be useful if you are stuck.

• Key Idea: Notice this language is the intersection of “palindromes over \( \{a, b\} \)” and “words over \( \{a, b\} \) at least three letters long”. The latter is a regular language.
• Write a context-free grammar for palindromes over \( \{a, b\} \) at least three letters long.

• You can just use intuition to solve this problem, but there is also a more structured approach.

• The structured approach sometimes produces larger or more complicated grammars than an intuitive approach, but may be useful if you are stuck.

• Key Idea: Notice this language is the intersection of “palindromes over \( \{a, b\} \)” and “words over \( \{a, b\} \) at least three letters long”. The latter is a regular language.

• We will refer to conditions like “at least three letters long” as regular restrictions since they can be described in terms of regular languages.
To deal with regular restrictions, draw a DFA for the regular language.

```
0 -> 1 (a, b)
1 -> 2 (a, b)
2 -> 3 (a, b)
3 (accept state)
```

- 0: Initial state
- 1: State after reading 'a' or 'b'
- 2: State after reading 'a' or 'b' again
- 3: Accept state
• To deal with regular restrictions, draw a DFA for the regular language.

• For each state \( q \) of the DFA, think about this: what language would be recognized if we made \( q \) into the initial state of the DFA?
Constructing Context-Free Grammars

- To deal with regular restrictions, draw a DFA for the regular language.

- For each state $q$ of the DFA, think about this: what language would be recognized if we made $q$ into the initial state of the DFA?
  
- Each choice of initial state gives rise to a different regular restriction:
• To deal with regular restrictions, draw a DFA for the regular language.
  \[ a, b \]

  \[ 0 \rightarrow a, b \rightarrow 1 \rightarrow a, b \rightarrow 2 \rightarrow a, b \rightarrow 3 \]

• For each state \( q \) of the DFA, think about this: what language would be recognized if we made \( q \) into the initial state of the DFA?

• Each choice of initial state gives rise to a different regular restriction:
  • State 0: Words must be length at least 3.
• To deal with regular restrictions, draw a DFA for the regular language. Draw a DFA for the regular language.

For each state $q$ of the DFA, think about this: what language would be recognized if we made $q$ into the initial state of the DFA?

Each choice of initial state gives rise to a different regular restriction:

- State 0: Words must be length at least 3.
- State 1: Words must be length at least 2.
• To deal with regular restrictions, draw a DFA for the regular language.

![DFA Diagram]

• For each state $q$ of the DFA, think about this: what language would be recognized if we made $q$ into the initial state of the DFA?

• Each choice of initial state gives rise to a different regular restriction:
  • State 0: Words must be length at least 3.
  • State 1: Words must be length at least 2.
  • State 2: Words must be length at least 1.
To deal with regular restrictions, draw a DFA for the regular language. For each state $q$ of the DFA, think about this: what language would be recognized if we made $q$ into the initial state of the DFA?

Each choice of initial state gives rise to a different regular restriction:

- State 0: Words must be length at least 3.
- State 1: Words must be length at least 2.
- State 2: Words must be length at least 1.
- State 3: No restriction (all words over $\{a, b\}$ are accepted).
- To deal with regular restrictions, draw a DFA for the regular language.

  - For each state $q$ of the DFA, think about this: what language would be recognized if we made $q$ into the initial state of the DFA?
  - Each choice of initial state gives rise to a different regular restriction:
    - State 0: Words must be length at least 3.
    - State 1: Words must be length at least 2.
    - State 2: Words must be length at least 1.
    - State 3: No restriction (all words over $\{a, b\}$ are accepted).
  - Make a non-terminal $Q_q$ for each state $q$, which generates palindromes over $\{a, b\}$ subject to the regular restriction of state $q$. 
Constructing Context-Free Grammars

- To deal with regular restrictions, draw a DFA for the regular language. For each state $q$ of the DFA, think about this: what language would be recognized if we made $q$ into the initial state of the DFA?
  - Each choice of initial state gives rise to a different regular restriction:
    - State 0: Words must be length at least 3.
    - State 1: Words must be length at least 2.
    - State 2: Words must be length at least 1.
    - State 3: No restriction (all words over $\{a, b\}$ are accepted).
- Make a non-terminal $Q_q$ for each state $q$, which generates palindromes over $\{a, b\}$ subject to the regular restriction of state $q$.
- The start symbol will be $Q_0$ which should generate palindromes over $\{a, b\}$ of length at least 3.

Winter 2020
CS241 Midterm Review
University of Waterloo
• Each choice of initial state gives rise to a different regular restriction:
  • State 0: Words must be length at least 3.
  • State 1: Words must be length at least 2.
  • State 2: Words must be length at least 1.
  • State 3: No restriction (all words over \{a, b\} are accepted).
• Make a non-terminal $Q_q$ for each state $q$, which generates palindromes over \{a, b\} subject to the regular restriction of state $q$.
• The start symbol will be $Q_0$ which should generate palindromes over \{a, b\} of length at least 3.
Constructing Context-Free Grammars

- Each choice of initial state gives rise to a different regular restriction:
  - State 0: Words must be length at least 3.
  - State 1: Words must be length at least 2.
  - State 2: Words must be length at least 1.
  - State 3: No restriction (all words over \{a, b\} are accepted).
- Make a non-terminal \(Q_q\) for each state \(q\), which generates palindromes over \{a, b\} subject to the regular restriction of state \(q\).
- The start symbol will be \(Q_0\) which should generate palindromes over \{a, b\} of length at least 3.
- A palindrome of length at least 3 is a palindrome of length at least 1, surrounded by the same beginning and ending letter.

\[
Q_0 \rightarrow aQ_2a \mid bQ_2b
\]
Constructing Context-Free Grammars

- Each choice of initial state gives rise to a different regular restriction:
  - State 0: Words must be length at least 3.
  - State 1: Words must be length at least 2.
  - State 2: Words must be length at least 1.
  - State 3: No restriction (all words over \{a, b\} are accepted).

- Make a non-terminal $Q_q$ for each state $q$, which generates palindromes over \{a, b\} subject to the regular restriction of state $q$.

- The start symbol will be $Q_0$ which should generate palindromes over \{a, b\} of length at least 3.

- A palindrome of length at least 3 is a palindrome of length at least 1, surrounded by the same beginning and ending letter.

  $$Q_0 \rightarrow aQ_2a \mid bQ_2b$$

- What about words generated by $Q_2$? A palindrome of length at least 1 is any non-empty palindrome. We can express this as a palindrome of length at least 3, or a palindrome of length between 1 and 2.

  $$Q_2 \rightarrow Q_0 \mid aa \mid bb \mid a \mid b$$
• Write a context-free grammar for palindromes over \( \{a, b\} \) at least three letters long.
• Write a context-free grammar for palindromes over \( \{a, b\} \) at least three letters long.

• Our analysis gives the following grammar:

\[
S \rightarrow aCa \mid bCb \\
C \rightarrow S \mid aa \mid bb \mid a \mid b
\]
• Write a context-free grammar for palindromes over \( \{a, b\} \) at least three letters long.

• Our analysis gives the following grammar:

\[
S \rightarrow aCa \mid bCb
\]

\[
C \rightarrow S \mid aa \mid bb \mid a \mid b
\]

• Another example: words over \( \{a, b, c\} \) of even length with an equal number of \( a \)'s and \( b \)'s.
• Write a context-free grammar for palindromes over \( \{a, b\} \) at least three letters long.

• Our analysis gives the following grammar:

\[
S \rightarrow aCa \mid bCb \\
C \rightarrow S \mid aa \mid bb \mid a \mid b
\]

• Another example: words over \( \{a, b, c\} \) of even length with an equal number of \( a \)'s and \( b \)'s.

• The regular restriction is “even length” corresponding to a two-state DFA. Make non-terminals for even length and odd length.
• Write a context-free grammar for palindromes over \{a, b\} at least three letters long.

• Our analysis gives the following grammar:

\[
S \rightarrow aCa \mid bCb \\
C \rightarrow S \mid aa \mid bb \mid a \mid b
\]

• Another example: words over \{a, b, c\} of even length with an equal number of a’s and b’s.

• The regular restriction is “even length” corresponding to a two-state DFA. Make non-terminals for even length and odd length.

• Here is one possible solution:

\[
E \rightarrow aEbE \mid bEaE \mid aObO \mid bOaO \mid cO \mid \varepsilon \\
O \rightarrow aObE \mid bOaE \mid aEbO \mid aEbO \mid cE \mid c
\]

Similar to \( S \rightarrow aSbS \mid bSaS \mid \varepsilon \) but we account for word length. For example, for an even-length word, instead of \( aSbS \) we have \( aEbE \) or \( aObO \) (two even subwords or two odd subwords make an even word).
• Compute Nullable, First, Follow and Predict for the following grammar.

\[
S \rightarrow \epsilon A \epsilon \\
A \rightarrow aCD \mid caBA \\
B \rightarrow b \mid \epsilon \\
C \rightarrow AB \mid B \\
D \rightarrow d
\]

• Let’s number each of the rules:

\[
\begin{align*}
S & \rightarrow \epsilon A \epsilon & (1) \\
A & \rightarrow aCD & (2) \\
A & \rightarrow caBA & (3) \\
B & \rightarrow b & (4) \\
B & \rightarrow \epsilon & (5) \\
C & \rightarrow AB & (6) \\
C & \rightarrow B & (7) \\
D & \rightarrow d & (8)
\end{align*}
\]
• First we compute Nullable. Start by assuming nothing is nullable.
  • Look for non-terminals which either directly derive $\varepsilon$, or derive a sequence of nullable non-terminals,
  • Iterate through rules until nothing changes on an iteration.

<table>
<thead>
<tr>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
</tr>
<tr>
<td>$A$</td>
</tr>
<tr>
<td>$B$</td>
</tr>
<tr>
<td>$C$</td>
</tr>
<tr>
<td>$D$</td>
</tr>
</tbody>
</table>
LL(1) Parsing

- First we compute Nullable. Start by assuming nothing is nullable.
- Look for non-terminals which either directly derive $\varepsilon$, or derive a sequence of nullable non-terminals,
- Iterate through rules until nothing changes on an iteration.

<table>
<thead>
<tr>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
</tr>
<tr>
<td>$A$</td>
</tr>
<tr>
<td>$B$</td>
</tr>
<tr>
<td>$C$</td>
</tr>
<tr>
<td>$D$</td>
</tr>
</tbody>
</table>

Iteration 1: $B$ is nullable by rule (5).
First we compute Nullable. Start by assuming nothing is nullable.

Look for non-terminals which either directly derive $\varepsilon$, or derive a sequence of nullable non-terminals,

Iterate through rules until nothing changes on an iteration.

<table>
<thead>
<tr>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
</tr>
<tr>
<td>$A$</td>
</tr>
<tr>
<td>$B$</td>
</tr>
<tr>
<td>$C$</td>
</tr>
<tr>
<td>$D$</td>
</tr>
</tbody>
</table>

Iteration 1: $C$ is nullable by rule (7).
First we compute Nullable. Start by assuming nothing is nullable.

Look for non-terminals which either directly derive $\varepsilon$, or derive a sequence of nullable non-terminals,

Iterate through rules until nothing changes on an iteration.

<table>
<thead>
<tr>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
</tr>
<tr>
<td>$A$</td>
</tr>
<tr>
<td>$B$</td>
</tr>
<tr>
<td>$C$</td>
</tr>
<tr>
<td>$D$</td>
</tr>
</tbody>
</table>

Iteration 2: Nothing changes, we’re done.
Now we compute First sets. Look at the first symbol on the right-hand-side of each rule.

- If it’s a terminal, add it. If it’s a non-terminal, add the corresponding First set, and if it’s nullable, look at the symbol after it.
Now we compute First sets. Look at the first symbol on the right-hand-side of each rule.

If it’s a terminal, add it. If it’s a non-terminal, add the corresponding First set, and if it’s nullable, look at the symbol after it.

Iteration 1: Add \( \vdash \) to First\( (S) \) by rule (1).
Now we compute First sets. Look at the first symbol on the right-hand-side of each rule.

- If it’s a terminal, add it. If it’s a non-terminal, add the corresponding First set, and if it’s nullable, look at the symbol after it.

<table>
<thead>
<tr>
<th>Nullable</th>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>${\epsilon}$</td>
</tr>
<tr>
<td>$A$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$B$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$C$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$D$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Iteration 1: Add $a$ to First($A$) by rule (2).
Now we compute First sets. Look at the first symbol on the right-hand-side of each rule.

If it’s a terminal, add it. If it’s a non-terminal, add the corresponding First set, and if it’s nullable, look at the symbol after it.

### Nullable

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$F$</td>
</tr>
<tr>
<td>$A$</td>
<td>$F$</td>
</tr>
<tr>
<td>$B$</td>
<td>$T$</td>
</tr>
<tr>
<td>$C$</td>
<td>$T$</td>
</tr>
<tr>
<td>$D$</td>
<td>$F$</td>
</tr>
</tbody>
</table>

### First

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>${\epsilon}$</td>
</tr>
<tr>
<td>$A$</td>
<td>${a, c}$</td>
</tr>
<tr>
<td>$B$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$C$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$D$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Iteration 1: Add $c$ to First($A$) by rule (3).
LL(1) Parsing

$S \rightarrow \emptyset A \emptyset$ (1)

$A \rightarrow aCD$ (2)

$A \rightarrow caBA$ (3)

$B \rightarrow b$ (4)

$B \rightarrow \epsilon$ (5)

$C \rightarrow AB$ (6)

$C \rightarrow B$ (7)

$D \rightarrow d$ (8)

- Now we compute First sets. Look at the first symbol on the right-hand-side of each rule.
- If it’s a terminal, add it. If it’s a non-terminal, add the corresponding First set, and if it’s nullable, look at the symbol after it.

<table>
<thead>
<tr>
<th>Nullable</th>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$A$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$B$</td>
<td>${b}$</td>
</tr>
<tr>
<td>$C$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$D$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Iteration 1: Add $b$ to First($B$) by rule (4).
Now we compute First sets. Look at the first symbol on the right-hand-side of each rule.

If it’s a terminal, add it. If it’s a non-terminal, add the corresponding First set, and if it’s nullable, look at the symbol after it.

### First

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>${\text{\textbackslash{}-}}$</td>
</tr>
<tr>
<td>$A$</td>
<td>${a, c}$</td>
</tr>
<tr>
<td>$B$</td>
<td>${b}$</td>
</tr>
<tr>
<td>$C$</td>
<td>${a, c}$</td>
</tr>
<tr>
<td>$D$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Iteration 1: Add First($A$) to First($C$) by rule (6).
Since $A$ is not nullable, we don’t need to add anything else for this rule.
Now we compute First sets. Look at the first symbol on the right-hand-side of each rule.

- If it’s a terminal, add it. If it’s a non-terminal, add the corresponding First set, and if it’s nullable, look at the symbol after it.

### First

<table>
<thead>
<tr>
<th></th>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>${\dagger}$</td>
</tr>
<tr>
<td>$A$</td>
<td>${a, c}$</td>
</tr>
<tr>
<td>$B$</td>
<td>${b}$</td>
</tr>
<tr>
<td>$C$</td>
<td>${a, b, c}$</td>
</tr>
<tr>
<td>$D$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

**Iteration 1:** Add First($B$) to First($C$) by rule (7). Although $B$ is nullable, there are no symbols after it, so we’re done with this rule.
Now we compute First sets. Look at the first symbol on the right-hand-side of each rule.

- If it’s a terminal, add it. If it’s a non-terminal, add the corresponding First set, and if it’s nullable, look at the symbol after it.

### First

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>${\text{\texttt{\textbar}}}$</td>
<td></td>
</tr>
<tr>
<td>$A$</td>
<td>${a, c}$</td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td>${b}$</td>
<td></td>
</tr>
<tr>
<td>$C$</td>
<td>${a, b, c}$</td>
<td></td>
</tr>
<tr>
<td>$D$</td>
<td>${d}$</td>
<td></td>
</tr>
</tbody>
</table>

Iteration 1: Add $d$ to First($D$) by rule (8).
Now we compute First sets. Look at the first symbol on the right-hand-side of each rule.

If it’s a terminal, add it. If it’s a non-terminal, add the corresponding First set, and if it’s nullable, look at the symbol after it.

Iteration 2: Nothing changes on this iteration, so we’re done.
Now we compute Follow sets. For Follow(A), look at what follows A on the right-hand-side of a rule.

If a terminal follows, add it. If a non-terminal B follows, add First(B), and if B is nullable, look at the next symbol. If nothing follows A or there is no next symbol after B, then add Follow(C) where C is the left-hand-side of the rule.

### Nullable

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

### First

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>{⊥}</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>{a, c}</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>{b}</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>{a, b, c}</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>{d}</td>
<td></td>
</tr>
</tbody>
</table>
Now we compute Follow sets. For Follow(A), look at what follows A on the right-hand-side of a rule.

If a terminal follows, add it. If a non-terminal B follows, add First(B), and if B is nullable, look at the next symbol. If nothing follows A or there is no next symbol after B, then add Follow(C) where C is the left-hand-side of the rule.

Iteration 1: Add \( \vdash \) to Follow(A) by rule (1).
Now we compute Follow sets. For Follow(A), look at what follows A on the right-hand-side of a rule.

If a terminal follows, add it. If a non-terminal B follows, add First(B), and if B is nullable, look at the next symbol. If nothing follows A or there is no next symbol after B, then add Follow(C) where C is the left-hand-side of the rule.

Winter 2020
CS241 Midterm Review
University of Waterloo
Now we compute Follow sets. For $\text{Follow}(A)$, look at what follows $A$ on the right-hand-side of a rule.

If a terminal follows, add it. If a non-terminal $B$ follows, add $\text{First}(B)$, and if $B$ is nullable, look at the next symbol. If nothing follows $A$ or there is no next symbol after $B$, then add $\text{Follow}(C)$ where $C$ is the left-hand-side of the rule.

### Nullable

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$F$</td>
</tr>
<tr>
<td>$A$</td>
<td>$F$</td>
</tr>
<tr>
<td>$B$</td>
<td>$T$</td>
</tr>
<tr>
<td>$C$</td>
<td>$T$</td>
</tr>
<tr>
<td>$D$</td>
<td>$F$</td>
</tr>
</tbody>
</table>

### First

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>${\epsilon}$</td>
</tr>
<tr>
<td>$A$</td>
<td>${a, c}$</td>
</tr>
<tr>
<td>$B$</td>
<td>${b}$</td>
</tr>
<tr>
<td>$C$</td>
<td>${a, b, c}$</td>
</tr>
<tr>
<td>$D$</td>
<td>${d}$</td>
</tr>
</tbody>
</table>

### Follow

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$A$</td>
<td>${\epsilon}$</td>
</tr>
<tr>
<td>$B$</td>
<td>${a, c}$</td>
</tr>
<tr>
<td>$C$</td>
<td>${d}$</td>
</tr>
<tr>
<td>$D$</td>
<td>${\epsilon}$</td>
</tr>
</tbody>
</table>

Iteration 1: Add $\text{First}(A)$ to $\text{Follow}(B)$ by rule (3). Since $A$ is not nullable we don’t add anything else.
Now we compute Follow sets. For \text{Follow}(A), look at what follows \(A\) on the right-hand-side of a rule.

If a terminal follows, add it. If a non-terminal \(B\) follows, add \text{First}(B), and if \(B\) is nullable, look at the next symbol. If nothing follows \(A\) or there is no next symbol after \(B\), then add \text{Follow}(C) where \(C\) is the left-hand-side of the rule.

\begin{center}
\begin{tabular}{c|c}
\text{Nullable} & \text{Follow} \\
\hline
\(S\) & \emptyset \\
\(A\) & \{\(b, d, \epsilon\)\} \\
\(B\) & \{\(a, c, d\)\} \\
\(C\) & \{\(d\)\} \\
\(D\) & \{\(\epsilon\)\} \\
\end{tabular}
\end{center}

Iteration 1: Add \text{First}(B) to \text{Follow}(A) by rule (6). Since \(B\) is nullable and there is nothing after \(B\), add \text{Follow}(C) to \text{Follow}(A) and \text{Follow}(B).
• Now we compute Follow sets. For Follow(A), look at what follows A on the right-hand-side of a rule.

• If a terminal follows, add it. If a non-terminal B follows, add First(B), and if B is nullable, look at the next symbol. If nothing follows A or there is no next symbol after B, then add Follow(C) where C is the left-hand-side of the rule.

<table>
<thead>
<tr>
<th>Nullable</th>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>First</th>
<th>S</th>
<th>{⊥}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{a, c}</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>{b}</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>{a, b, c}</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>{d}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>

Iteration 1: We add Follow(C) to Follow(B) by rule (7) since there is nothing after B, but this doesn’t change anything since we added it for rule (6).
LL(1) Parsing

$S \rightarrow \emptyset A \emptyset$ (1)
$A \rightarrow aCD$ (2)
$A \rightarrow caBA$ (3)
$B \rightarrow b$ (4)
$B \rightarrow \varepsilon$ (5)
$C \rightarrow AB$ (6)
$C \rightarrow B$ (7)
$D \rightarrow d$ (8)

• Now we compute Follow sets. For Follow($A$), look at what follows $A$ on the right-hand-side of a rule.

• If a terminal follows, add it. If a non-terminal $B$ follows, add First($B$), and if $B$ is nullable, look at the next symbol. If nothing follows $A$ or there is no next symbol after $B$, then add Follow($C$) where $C$ is the left-hand-side of the rule.

<table>
<thead>
<tr>
<th>Nullable</th>
<th>$S$</th>
<th>$F$</th>
<th>$A$</th>
<th>$F$</th>
<th>$B$</th>
<th>$T$</th>
<th>$C$</th>
<th>$T$</th>
<th>$D$</th>
<th>$F$</th>
</tr>
</thead>
</table>

| First   | $S$ | $\{\emptyset\}$ | $A$ | $\{a, c\}$ | $B$ | $\{b\}$ | $C$ | $\{a, b, c\}$ | $D$ | $\{d\}$ |

| Follow   | $S$ | $\emptyset$ | $A$ | $\{b, d, \emptyset\}$ | $B$ | $\{a, c, d\}$ | $C$ | $\{d\}$ | $D$ | $\{b, d, \emptyset\}$ |

Iteration 2: Update Follow($D$) with the new version of Follow($A$).
Now we compute Follow sets. For $\text{Follow}(A)$, look at what follows $A$ on the right-hand-side of a rule.

If a terminal follows, add it. If a non-terminal $B$ follows, add $\text{First}(B)$, and if $B$ is nullable, look at the next symbol. If nothing follows $A$ or there is no next symbol after $B$, then add $\text{Follow}(C)$ where $C$ is the left-hand-side of the rule.

<table>
<thead>
<tr>
<th>Nullable</th>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

| First   | S | a, c |
|---------|---+------|
| A | b |      |
| B | a, c, d |
| C | d |      |
| D | d |      |

Follow

<table>
<thead>
<tr>
<th>S</th>
<th>$\emptyset$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>${b, d, \text{−}}$</td>
</tr>
<tr>
<td>B</td>
<td>${a, c, d}$</td>
</tr>
<tr>
<td>C</td>
<td>${d}$</td>
</tr>
<tr>
<td>D</td>
<td>${b, d, \text{−}}$</td>
</tr>
</tbody>
</table>

Iteration 3: Nothing changes on this iteration.
LL(1) Parsing

$S \rightarrow \epsilon A \epsilon$ (1)
$A \rightarrow aCD$ (2)
$A \rightarrow caBA$ (3)
$B \rightarrow b$ (4)
$B \rightarrow \epsilon$ (5)
$C \rightarrow AB$ (6)
$C \rightarrow B$ (7)
$D \rightarrow d$ (8)

- Finally, we compute the Predict table.
- For each rule $A \rightarrow \gamma$, compute First($\gamma$). For each $a \in \text{First}(\gamma)$, add the rule to Predict($A, a$).
- Also, if $\gamma$ is nullable, then for each $a \in \text{Follow}(A)$, add the rule to Predict($A, a$).

<table>
<thead>
<tr>
<th>Nullable</th>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>First</th>
<th>S</th>
<th>{\epsilon}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{a, c}</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>{b}</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>{a, b, c}</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>{d}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Follow</th>
<th>S</th>
<th>{}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{b, d, \epsilon}</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>{a, c, d}</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>{d}</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>{b, d, \epsilon}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Predict</th>
<th>\epsilon</th>
<th>\epsilon</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finally, we compute the Predict table.

- For each rule $A \rightarrow \gamma$, compute $\text{First}(\gamma)$. For each $a \in \text{First}(\gamma)$, add the rule to $\text{Predict}(A, a)$.
- Also, if $\gamma$ is nullable, then for each $a \in \text{Follow}(A)$, add the rule to $\text{Predict}(A, a)$.

<table>
<thead>
<tr>
<th>Nullable</th>
<th>$S$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
<td>$F$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>First</th>
<th>$S$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${\cdot}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>${a, c}$</td>
<td>${b}$</td>
<td>${a, b, c}$</td>
<td>${d}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Follow</th>
<th>$S$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>${b, d, \cdot}$</td>
<td>${a, c, d}$</td>
<td>${d}$</td>
<td>${b, d, \cdot}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Predict</th>
<th>$\cdot$</th>
<th>$\cdot$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$\cdot$</td>
<td></td>
<td>$b$</td>
<td>$c$</td>
<td>$d$</td>
<td></td>
</tr>
<tr>
<td>$A$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rule (1): $\text{First}(\cdot, A, \cdot) = \{\cdot\}$ so add (1) to $\text{Predict}(S, \cdot)$.
LL(1) Parsing

- Finally, we compute the Predict table.
- For each rule \( A \rightarrow \gamma \), compute First(\( \gamma \)). For each \( a \in \text{First}(\gamma) \), add the rule to Predict(\( A, a \)).
- Also, if \( \gamma \) is nullable, then for each \( a \in \text{Follow}(A) \), add the rule to Predict(\( A, a \)).

Rule (2): First(\( aCD \)) = \{a\} so add (2) to Predict(\( A, a \)).
Finally, we compute the Predict table.

For each rule $A \rightarrow \gamma$, compute $\text{First}(\gamma)$. For each $a \in \text{First}(\gamma)$, add the rule to $\text{Predict}(A, a)$.

Also, if $\gamma$ is nullable, then for each $a \in \text{Follow}(A)$, add the rule to $\text{Predict}(A, a)$.

Rule (3): $\text{First}(caBA) = \{c\}$ so add (3) to $\text{Predict}(A, c)$. 
LL(1) Parsing

- Finally, we compute the Predict table.
- For each rule $A \rightarrow \gamma$, compute $\text{First}(\gamma)$. For each $a \in \text{First}(\gamma)$, add the rule to $\text{Predict}(A, a)$.
- Also, if $\gamma$ is nullable, then for each $a \in \text{Follow}(A)$, add the rule to $\text{Predict}(A, a)$.

\[
\begin{array}{c|c|c|c|c|c}
\text{Predict} & \text{\textbackslash{}} & \text{\textbackslash{}} & a & b & c & d \\
\hline
S & 1 \\
A & 2 & 3 \\
B & 4 \\
C & \\
D & \\
\end{array}
\]

Rule (4): $\text{First}(b) = \{b\}$ so add (4) to $\text{Predict}(B, b)$.
Finally, we compute the Predict table.

For each rule \( A \rightarrow \gamma \), compute First(\( \gamma \)). For each \( a \in \text{First}(\gamma) \), add the rule to \( \text{Predict}(A, a) \).

Also, if \( \gamma \) is nullable, then for each \( a \in \text{Follow}(A) \), add the rule to \( \text{Predict}(A, a) \).

Rule (5): First(\( \varepsilon \)) = \{\}. However \( \varepsilon \) is nullable and \( \text{Follow}(B) = \{a, c, d\} \), so add (5) to \( \text{Predict}(B, a) \), \( \text{Predict}(B, c) \) and \( \text{Predict}(B, d) \).
LL(1) Parsing

\[ S \rightarrow \epsilon A \epsilon \] (1)
\[ A \rightarrow aCD \] (2)
\[ A \rightarrow caBA \] (3)
\[ B \rightarrow b \] (4)
\[ B \rightarrow \epsilon \] (5)
\[ C \rightarrow AB \] (6)
\[ C \rightarrow B \] (7)
\[ D \rightarrow d \] (8)

- Finally, we compute the Predict table.
- For each rule \( A \rightarrow \gamma \), compute \( \text{First}(\gamma) \). For each \( a \in \text{First}(\gamma) \), add the rule to \( \text{Predict}(A, a) \).
- Also, if \( \gamma \) is nullable, then for each \( a \in \text{Follow}(A) \), add the rule to \( \text{Predict}(A, a) \).

<table>
<thead>
<tr>
<th>Nullable</th>
<th>Predict</th>
<th>( \Downarrow )</th>
<th>( \Downarrow )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A )</td>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( B )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( D )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
</tr>
<tr>
<td>( A )</td>
</tr>
<tr>
<td>( B )</td>
</tr>
<tr>
<td>( C )</td>
</tr>
<tr>
<td>( D )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
</tr>
<tr>
<td>( A )</td>
</tr>
<tr>
<td>( B )</td>
</tr>
<tr>
<td>( C )</td>
</tr>
<tr>
<td>( D )</td>
</tr>
</tbody>
</table>

Rule (6): \( \text{First}(AB) = \text{First}(A) = \{a, c\} \) since \( A \) is not nullable. Add (6) to \( \text{Predict}(C, a) \) and \( \text{Predict}(C, b) \).
Finally, we compute the Predict table.

For each rule $A \rightarrow \gamma$, compute $\text{First}(\gamma)$. For each $a \in \text{First}(\gamma)$, add the rule to $\text{Predict}(A, a)$.

Also, if $\gamma$ is nullable, then for each $a \in \text{Follow}(A)$, add the rule to $\text{Predict}(A, a)$.

Rule (7): $\text{First}(B) = \{b\}$ so add (7) to $\text{Predict}(C, b)$. Also $B$ is nullable and $\text{Follow}(C) = \{d\}$, so add (7) to $\text{Predict}(C, d)$. 
Finally, we compute the Predict table.

For each rule $A \rightarrow \gamma$, compute $\text{First}(\gamma)$. For each $a \in \text{First}(\gamma)$, add the rule to $\text{Predict}(A, a)$.

Also, if $\gamma$ is nullable, then for each $a \in \text{Follow}(A)$, add the rule to $\text{Predict}(A, a)$.

Rule (8): $\text{First}(D) = \{d\}$ so add (8) to $\text{Predict}(D, d)$.
LL(1) Parsing

\[ S \rightarrow \dashv A \dashv \] (1)
\[ A \rightarrow aCD \] (2)
\[ A \rightarrow caBA \] (3)
\[ B \rightarrow b \] (4)
\[ B \rightarrow \varepsilon \] (5)
\[ C \rightarrow AB \] (6)
\[ C \rightarrow B \] (7)
\[ D \rightarrow d \] (8)

- Finally, we compute the Predict table.
- For each rule \( A \rightarrow \gamma \), compute First(\( \gamma \)). For each \( a \in \text{First}(\gamma) \), add the rule to Predict(\( A, a \)).
- Also, if \( \gamma \) is nullable, then for each \( a \in \text{Follow}(A) \), add the rule to Predict(\( A, a \)).

<table>
<thead>
<tr>
<th>Nullable</th>
<th>S</th>
<th>F</th>
<th>A</th>
<th>F</th>
<th>B</th>
<th>T</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predict</td>
<td>(\vdash)</td>
<td>(\vdash)</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table is complete and we see the grammar is LL(1).
Let’s parse the word $\vdash acabadd \vdash$.

Top of the stack is on the right.

When applying a rule: pop LHS off the stack, push RHS from rightmost symbol to leftmost.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Next Symbol</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$\vdash$</td>
<td>Apply rule (1)</td>
</tr>
<tr>
<td>$\vdash A$</td>
<td>$\vdash$</td>
<td>Match $\vdash$</td>
</tr>
<tr>
<td>$\vdash A$</td>
<td>$a$</td>
<td>Apply rule (2)</td>
</tr>
<tr>
<td>$\vdash DCa$</td>
<td>$a$</td>
<td>Match $a$</td>
</tr>
<tr>
<td>$\vdash DC$</td>
<td>$c$</td>
<td>Apply rule (6)</td>
</tr>
<tr>
<td>$\vdash DBA$</td>
<td>$c$</td>
<td>Apply rule (3)</td>
</tr>
<tr>
<td>$\vdash DBABac$</td>
<td>$c$</td>
<td>Match $c$</td>
</tr>
<tr>
<td>$\vdash DBABA$</td>
<td>$a$</td>
<td>Match $a$</td>
</tr>
<tr>
<td>$\vdash DBAB$</td>
<td>$b$</td>
<td>Apply rule (4)</td>
</tr>
<tr>
<td>$\vdash DBAb$</td>
<td>$b$</td>
<td>Match $b$</td>
</tr>
<tr>
<td>$\vdash DBA$</td>
<td>$a$</td>
<td>Apply rule (2)</td>
</tr>
<tr>
<td>$\vdash DBDCa$</td>
<td>$a$</td>
<td>Match $a$</td>
</tr>
<tr>
<td>$\vdash DBDC$</td>
<td>$d$</td>
<td>Apply rule (7)</td>
</tr>
<tr>
<td>$\vdash DBDB$</td>
<td>$d$</td>
<td>Apply rule (5)</td>
</tr>
<tr>
<td>$\vdash DBD$</td>
<td>$d$</td>
<td>Apply rule (8)</td>
</tr>
<tr>
<td>$\vdash DBd$</td>
<td>$d$</td>
<td>Match $d$</td>
</tr>
<tr>
<td>$\vdash DB$</td>
<td>$d$</td>
<td>Apply rule (5)</td>
</tr>
<tr>
<td>$\vdash D$</td>
<td>$d$</td>
<td>Apply rule (8)</td>
</tr>
<tr>
<td>$\vdash d$</td>
<td>$d$</td>
<td>Match $d$</td>
</tr>
<tr>
<td>$\vdash \epsilon$</td>
<td>None</td>
<td>Accept</td>
</tr>
</tbody>
</table>