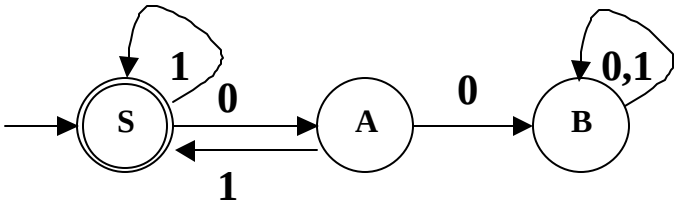
Main topics of the week:

- **Context Free Grammars and parse trees**
- **Regular Languages and CFGs**
- **Chomsky Normal Form for CFGs**
- **Pushdown Automata Definition and examples**
- **Construction of PDA from a CFG**

CFGs for Regular Languages.

A CFG for a regular language is easy if you have the DFA for the language. To create the CFG from the DFA, just make a variable for each state and have a rule of the form $R_i \rightarrow aR_j$ if there is a transition by input 'a' from R_i to R_j. Have a rule that takes each



S ® 0A | 1S | e
A ® 0B | 1S
B ® 0B | 1B

Simplified to:

S ® 01S | 1S | e

CFG from DFA for regular language { w Î {0,1}* | every 0 is followed by a 1}

variable corresponding to an accept state to ε.

A **right-linear** grammar is a grammar in which all productions are of the form

$A \rightarrow xB$ where x is a terminal

or $A \rightarrow \varepsilon$. The construction above shows that a DFA can be used to create a right-linear grammar. A left-linear grammar has a similar definition. A **regular** grammar is a right or left linear grammar, and corresponds to a regular language.

Example of Chomsky normalization.

This example is for a grammar which produces balanced parentheses. The right side at each stage shows the way the grammar has been changed. The highlighted text indicates what is being eliminated on the left and what are the corresponding additons on the right.

$S \rightarrow (S) \mid SS \mid \varepsilon$	

First we add a new start symbol.

$S \rightarrow (S) \mid SS \mid \varepsilon$	$S_0 \rightarrow S$ $S \rightarrow (S) \mid SS \mid \varepsilon$

Then we remove the ε rule $S \rightarrow \varepsilon$.

$S_0 \rightarrow S$ $S \rightarrow (S) \mid SS \mid \varepsilon$	$S_0 \rightarrow S \mid \varepsilon$ $S \rightarrow (S) \mid SS \mid ( ) \mid S$

Now we remove the unit rules $S \rightarrow S$ and $S_0 \rightarrow S$.

$S_0 \rightarrow S \mid \varepsilon$ $S \rightarrow (S) \mid SS \mid ( ) \mid S$	$S_0 \rightarrow (S) \mid SS \mid ( ) \mid \varepsilon$ $S \rightarrow (S) \mid SS \mid ( )$

Individual rules are added for terminals appearing in strings of length ≥ 2.

$S_0 \rightarrow (S) \mid SS \mid ( ) \mid \varepsilon$ $S \rightarrow (S) \mid SS \mid ( )$	$S_0 \rightarrow LSR \mid SS \mid LR \mid \varepsilon$ $S \rightarrow LSR \mid SS \mid LR$ $L \rightarrow ($ $R \rightarrow)$

Finally, reduce all strings of length ≥ 3, adding new variables as necessary.

$S_0 \rightarrow LSR \mid SS \mid LR \mid \varepsilon$ $S \rightarrow LSR \mid SS \mid LR$ $L \rightarrow ($ $R \rightarrow)$	$S_0 \rightarrow L_1R \mid SS \mid LR \mid \varepsilon$ $S \rightarrow L_1R \mid SS \mid LR$ $L_1 \rightarrow LS$ $L \rightarrow ($ $R \rightarrow)$

Pushdown Automata Examples

Let A be the language $\{ 0^n1^n \mid n \geq 0\}$, which we know is not a regular language. We show a pushdown automaton that recognizes A. For the formal specification, we need a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$:

1. $Q = \{q_1,q_2,q_3,q_4\}$
2. $\Sigma = \{0,1\}$
3. $\Gamma = \{0,\$\}$
4. $F = \{q_1,q_4\}$

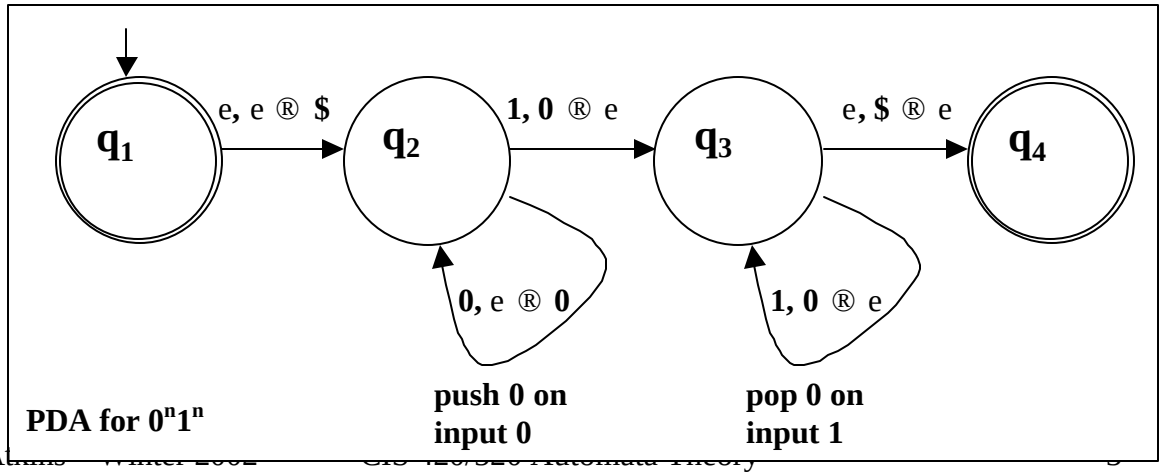
For δ we could use a 3-dimensional table for δ, but instead we'll give the non $\varnothing$ mappings:

$\delta(q_1,\varepsilon,\varepsilon) = \{(q_2,\$)\}$
$\delta(q2,0,\varepsilon) = \{(q_2,0)\}$
$\delta(q_2,1,0) = \{(q_3,\varepsilon)\}$
$\delta(q_3,1,0) = \{(q_3,\varepsilon)\}$
$\delta(q_3,\varepsilon,\$) = \{(q_4,\varepsilon)\}$

Notice that the first transition is really just for initialization since it happens with no input, and pushes the *$* on the stack. The *$* is basically going to be used as a marker for the bottom of the stack. This also advances us to the second state. Now when we see *0*, we stay in the second state, but push *0* on the stack. If we see *1* on input when we have *0* at the top of the stack, we pop it and advance to the third state. Note that if we saw a *1* first, we would halt as this transition is not defined since the stack would have *$* on the top. Once in the third state, with each *1* and still a *0* on the stack, we pop it but stay in the third state. When we see *$* on the stack (no input), we pop it and advance to the accept state. Note that the accept state has not transitions, so we cannot have any more input in order to be accepted. So we're stuck if we see the *$* and haven't seen any *1*'s, or we still have input. We're also stuck if we see any 0's after we have seen a *1*. And most importantly, the number of *0*'s and *1*'s must be the same.

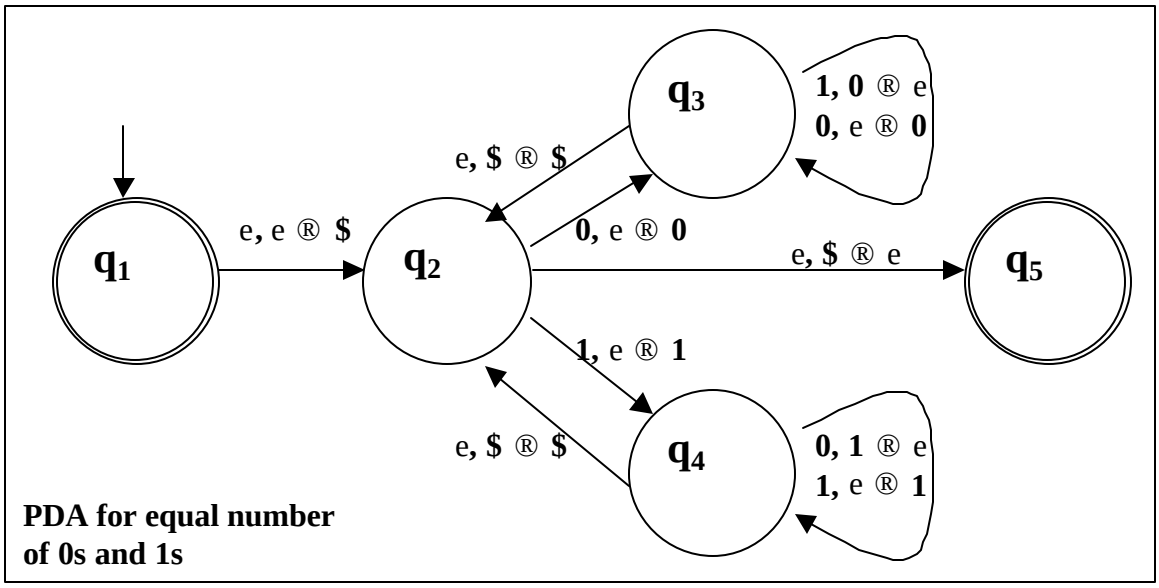
We can also draw state diagrams for PDAs. In addition to our usual conventions for drawing, we now add to our arrow label to indicate what is happening to the stack. This will just be the mapping as given in the definition, from Γ_ε to Γ_ε. A mapping $e \to b$ means to push b on the stack, $a \to e$ means to pop a from the stack, $e \to e$ means to do nothing to the stack, and $a \to b$ means to pop a and push b.

Here is the diagram for the PDA for 0^n1^n just described formally. Note that the start state is an accept state, and this allows empty strings. We see the transitions as discussed above, with the first being an initialization where *$* is pushed on the stack. Then we loop



on *0*'s, pushing *0* on the stack for each input *0*. *1*'s take us to the next state and also pop a *0*, which must be on the stack, else we would halt. We loop on *1*'s and popping *0*'s until the stack is empty (i.e., has our *$* marker).

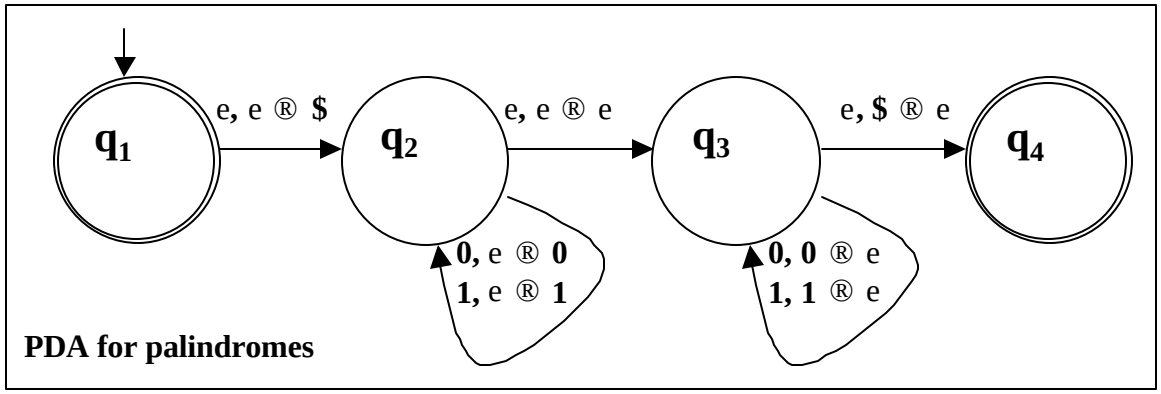
Here's another example, the language of all strings that have an equal number of 0s and 1s. In this case, we'll use the stack to match the 0s and 1s as we go along, which means we will be pushing 0s on until we see 1s at which point we'll pop 0s for each 1. When the stack is empty and we see 1s, we'll push 1s until we see 0s to match. The trick is noticing when the stack is empty so that we can know to switch to pushing. We do this by first initializing the stack as before (the transition to q_2) with the marker '$'. From this state, if we see a 0, we push 0s (state q_3) and also in this state, whenever we see 1s, we pop the 0s. If the stack ever gets empty, we return to q_2 and preserve our marker. We use state q_4 similarly to push 1s and match 0s against them. What makes this work is that whenever the stack is empty, we return to our neutral state so we can decide whether to push 1s or 0s, depending on the input. Notice that we are not requiring equal consecutive numbers of 0s and 1s, just that they ultimately balance.



PDA for equal number of 0s and 1s

In fact, we could have eliminated the last state q_5 by having the transition there go back to the start state instead. Essentially, this would mean that we would be in the accept state whenever the 0s and 1s were balanced, and if there is no more input, the string would be accepted. (If we had done this in the previous example, it would have extended the language to be all strings where 0s were followed by an equal number of 1s.)

One more example: we used the pumping lemma to show that the palindromes are not regular. Let's now construct a PDA to recognize palindromes. Notice that we really need nondeterminism here to "guess" when we are at the middle of the string. That is, we push 0s and 1s on the stack, then guess to begin popping as we match to the remaining input. Without nondeterminism to make this guess, this would be very hard to come up with a PDA



PDA for palindromes

Example of constructing a PDA from a CFG.

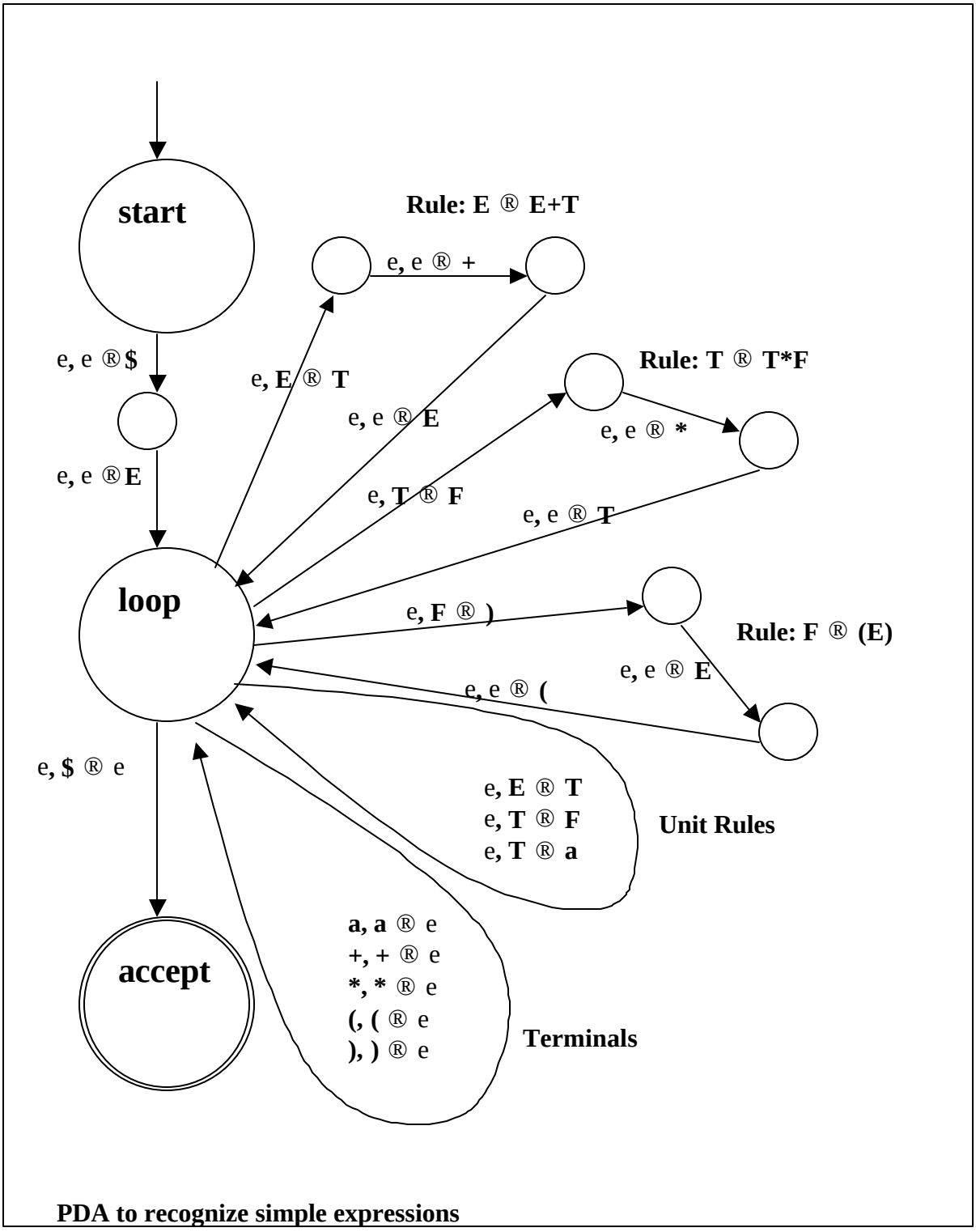
Here is a concrete example of using this construction of Lemma 2.13 on a CFL. We try this for the simple expression language defined by the grammar we have seen before:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid a$

Here the variables are E, T, and F, and the terminals are a, +, *, (, and). There are six rules in all. The following diagram shows the PDA constructed for this language. We have shown the longhand transitions for the non unit rules using intermediate states. Note that the symbols are pushed on the stack in the reverse order from how they occur in the rule. Three of the rules are unit rules, so require no intermediate state. Finally, there are five terminals, so five corresponding transitions on matching input.



PDA to recognize simple expressions