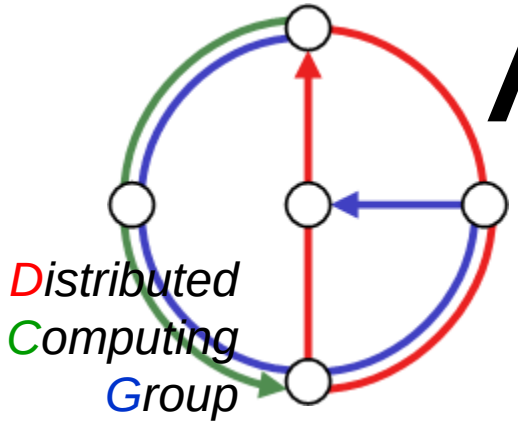
Chapter 2
SMARTER AUTOMATA



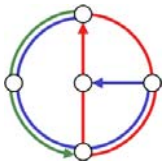
Distributed Computing Group

Discrete Event Systems

Fall 2008

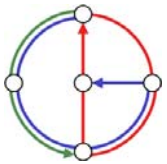
Overview

- Motivation
- Context free grammars (CFG)
- Derivations
- Parse trees
- Ambiguity
- Pushdown Automata
- Stacks and recursion
- Grammar Transforms
- Chomsky normal form
- CFG → PDA
- Context Sensitive Grammars
- PDA Transforms
- Pure Push and Pop machines
- PDA → CFG
- Context Free Pumping
- Transducers
- Turing Machines
- Mind-Body Problem



Motivation

- Why is a language such as $\{0^n1^n \mid n \geq 0\}$ not regular?!?
- It's really simple! All you need to keep track is the number of 0's…
- In this chapter we first study context-free grammars
  - More powerful than regular languages
  - Recursive structure
  - Developed for human languages
  - Important for engineers (parsers, protocols, etc.)



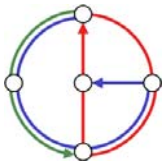
Motivating Example

- Palindromes, for example, are not regular.
- But there is a pattern.

- Q: If you have one palindrome, how can you generate another?
- A: Generate palindromes recursively as follows:
  - Base case: ε, 0 and 1 are palindromes.
  - Recursion: If *x* is a palindrome, then so are 0*x*0 and 1*x*1.

- Notation: $x \rightarrow \varepsilon \mid 0 \mid 1 \mid 0x0 \mid 1x1$.
  - Each pipe ("|") is an or, just as in UNIX regexp's.
  - In fact, all palindromes can be generated from ε using these rules.

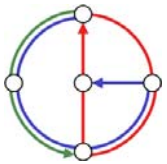
- Q: How would you generate 11011011?



Context Free Grammars (CFG): Definition

- Definition: A context free grammar consists of (V, Σ, R, S) with:
  - V: a finite set of variables (or symbols, or non-terminals)
  - Σ: a finite set set of terminals (or the alphabet)
  - R: a finite set of rules (or productions)
    of the form $v \rightarrow w$ with $v \in V$, and $w \in (\Sigma_\varepsilon \cup V)^*$
    (read: "v yields w" or "v produces w")
  - $S \in V$: the start symbol.

- Q: What are (V, Σ, R, S) for our palindrome example?
- A: $V = \{x\}$,
  $\Sigma = \{0,1\}$,
  $R = \{x \rightarrow \varepsilon, x \rightarrow 0, x \rightarrow 1, x \rightarrow 0x0, x \rightarrow 1x1\}$,
  $S = x$.

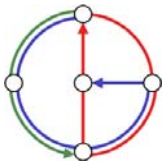


Derivations and Language

- Definition: The derivation symbol "$\Rightarrow$" (read "1-step derives" or "1-step produces") is a relation between strings in $(\Sigma \cup V)^*$. We write $x \Rightarrow y$ if x and y can be broken up as $x = svt$ and $y = swt$ with $v \rightarrow w$ being a production in R.

- Definition: The derivation symbol "$\Rightarrow^*$", (read "derives" or "produces" or "yields") is a relation between strings in $(\Sigma \cup V)^*$. We write $x \Rightarrow^* y$ if there is a sequence of 1-step productions from x to y. I.e., there are strings x_i with i ranging from 0 to n such that $x = x_0$, $y = x_n$ and $x_0 \Rightarrow x_1, x_1 \Rightarrow x_2, x_2 \Rightarrow x_3, \ldots, x_{n-1} \Rightarrow x_n$.

- Definition: Let G be a context-free grammar. The context-free language (CFL) generated by G is the set of all terminal strings which are derivable from the start symbol. Symbolically: $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$

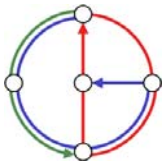


Example: Infix Expressions

- Infix expressions involving {+, ×, *a*, *b*, *c*, (,)}
- *E* stands for an expression (most general)
- *F* stands for factor (a multiplicative part)
- *T* stands for term (a product of factors)
- *V* stands for a variable: *a, b,* or *c*

- Grammar is given by:
  - $E \rightarrow T \mid E + T$
  - $T \rightarrow F \mid T \times F$
  - $F \rightarrow V \mid (E)$
  - $V \rightarrow a \mid b \mid c$

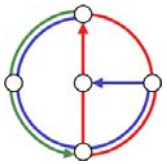
- Convention: Start variable is the first one in grammar (*E*)



Example: Infix Expressions

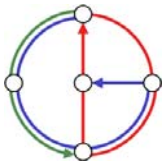
- Consider the string u given by $a \times b + (c + (a + c))$
- This is a valid infix expression. Can be generated from E.

1. A sum of two expressions, so first production must be $E \Rightarrow E + T$
2. Sub-expression $a \times b$ is a product, so a term so generated by sequence $E + T \Rightarrow T + T \Rightarrow T \times F + T \Rightarrow^* a \times b + T$
3. Second sub-expression is a factor only because a parenthesized sum. $a \times b + T \Rightarrow a \times b + F \Rightarrow a \times b + (E) \Rightarrow a \times b + (E + T)$ …
4. $E \Rightarrow E + T \Rightarrow T + T \Rightarrow T \times F + T \Rightarrow F \times F + T \Rightarrow V \times F + T \Rightarrow a \times F + T \Rightarrow a \times V + T \Rightarrow a \times b + T \Rightarrow a \times b + F \Rightarrow a \times b + (E) \Rightarrow a \times b + (E + T) \Rightarrow a \times b + (T + T) \Rightarrow a \times b + (F + T) \Rightarrow a \times b + (V + T) \Rightarrow a \times b + (c + T) \Rightarrow a \times b + (c + F) \Rightarrow a \times b + (c + (E)) \Rightarrow a \times b + (c + (E + T)) \Rightarrow a \times b + (c + (T + T)) \Rightarrow a \times b + (c + (F + T)) \Rightarrow a \times b + (c + (a + T)) \Rightarrow a \times b + (c + (a + F)) \Rightarrow a \times b + (c + (a + V)) \Rightarrow a \times b + (c + (a + c))$



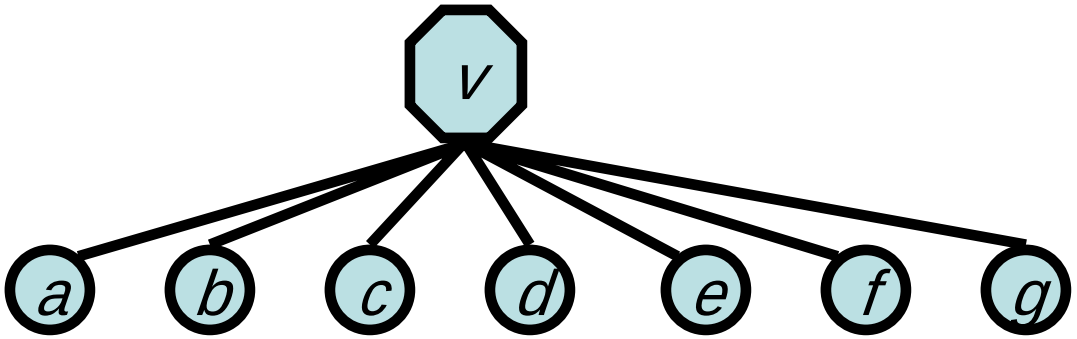
Left- and Right-most derivation

- The derivation on the previous slide was a so-called left-most derivation.
- In a right-most derivation, the variable most to the right is replaced.
  - $E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + (E) \Rightarrow E + (E + T) \Rightarrow$ etc.
- There can be a lot of ambiguity involved in how a string is derived.
- Another way to describe a derivation in a unique way is using derivation trees.

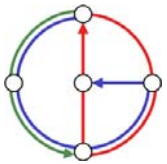


Derivation Trees

- In a derivation tree (or parse tree) each node is a symbol. Each parent is a variable whose children spell out the production from left to right. For, example $v \rightarrow abcdefg$:

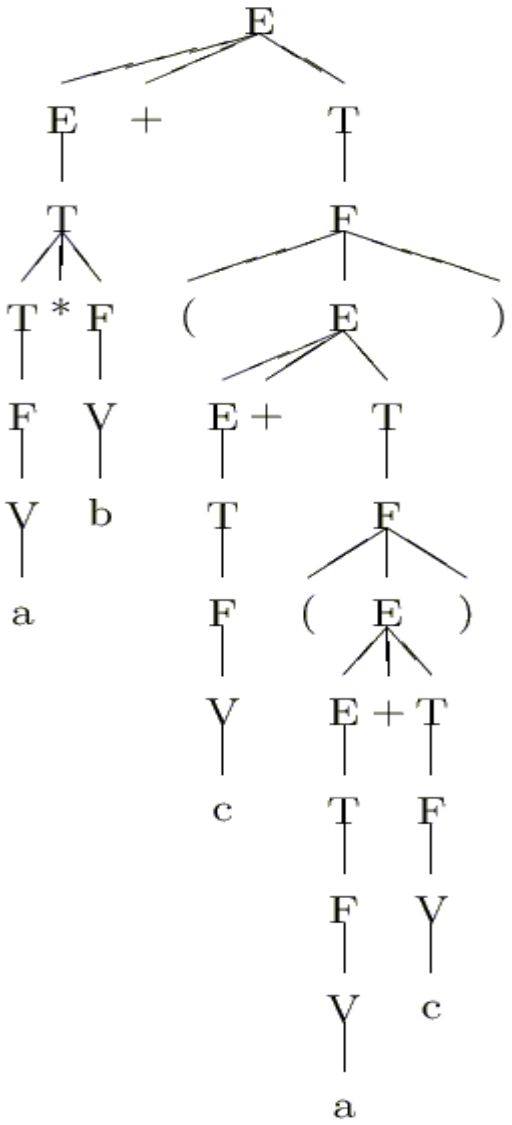


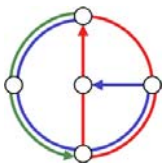
- The root is the start variable.
- The leaves spell out the derived string from left to right.



Derivation Trees

- On the right, we see a derivation tree for our string $a \times b + (c + (a + c))$
- Derivation trees help understanding semantics! You can tell how expression should be evaluated from the tree.



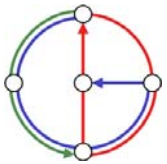


Ambiguity

<sentence>	→	***<action>* | *<action>*** with ***<subject>***
<action>	→	***<subject><activity>***
<subject>	→	***<noun>* | *<noun>*** and ***<subject>***
<activity>	→	***<verb>* | *<verb><object>***
<noun>	→	Hannibal | Clarice | rice | onions
<verb>	→	ate | played
<prep>	→	with | and | or
<object>	→	***<noun>* | *<noun><prep><object>***

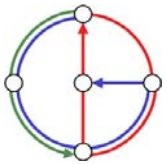
- Clarice played with Hannibal
- Clarice ate rice with onions
- Hannibal ate rice with Clarice

- Q: Are there any suspect sentences?

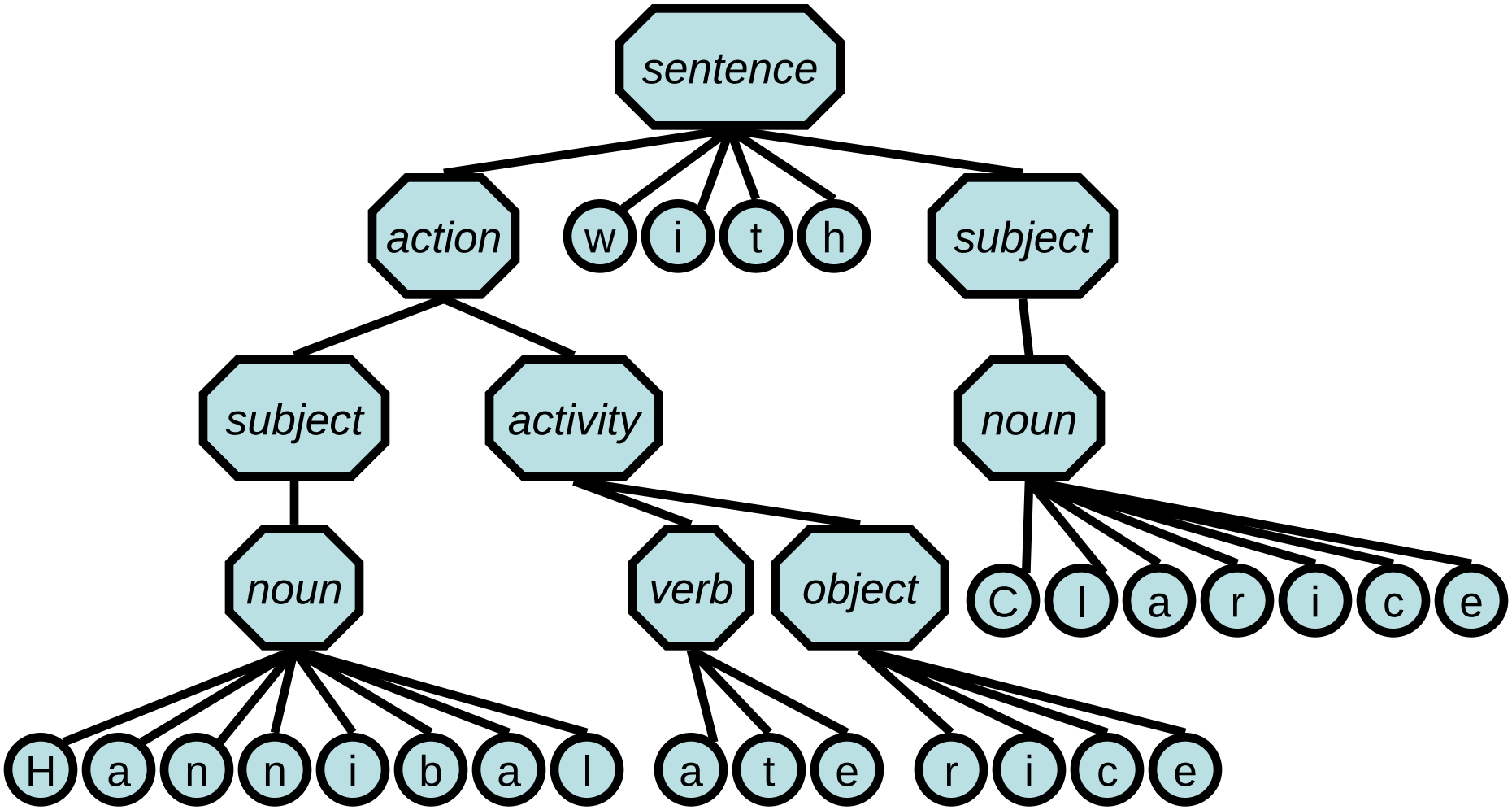


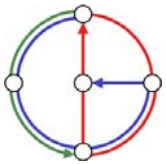
Ambiguity

- A: Consider “Hannibal ate rice with Clarice”
- This could either mean
  - Hannibal and Clarice ate rice *together*.
  - Hannibal ate rice and *ate* Clarice.
- This ambiguity arises from the fact that the sentence has two different parse-trees, and therefore two different interpretations:

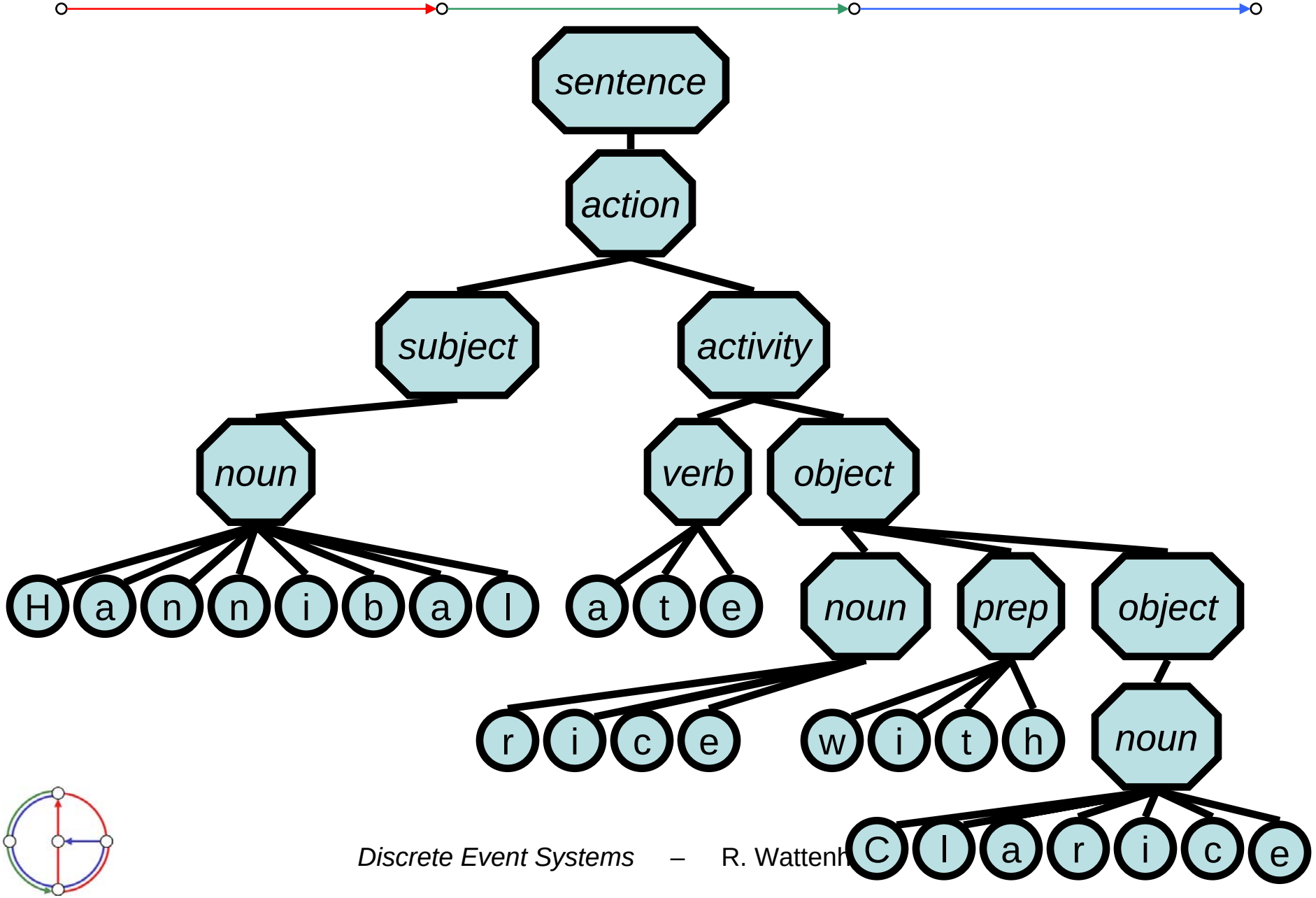


Hannibal and Clarice Ate





Hannibal the Cannibal

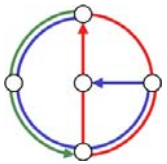


Ambiguity: Definition

- Definition: A string *x* is said to be ambiguous relative the grammar *G* if there are two essentially different ways to derive *x* in *G*. I.e. *x* admits two (or more) different parse-trees (equivalently, *x* admits different left-most [resp. right-most] derivations). A grammar *G* is said to be ambiguous if there is some string *x* in *L*(*G*) which is ambiguous.

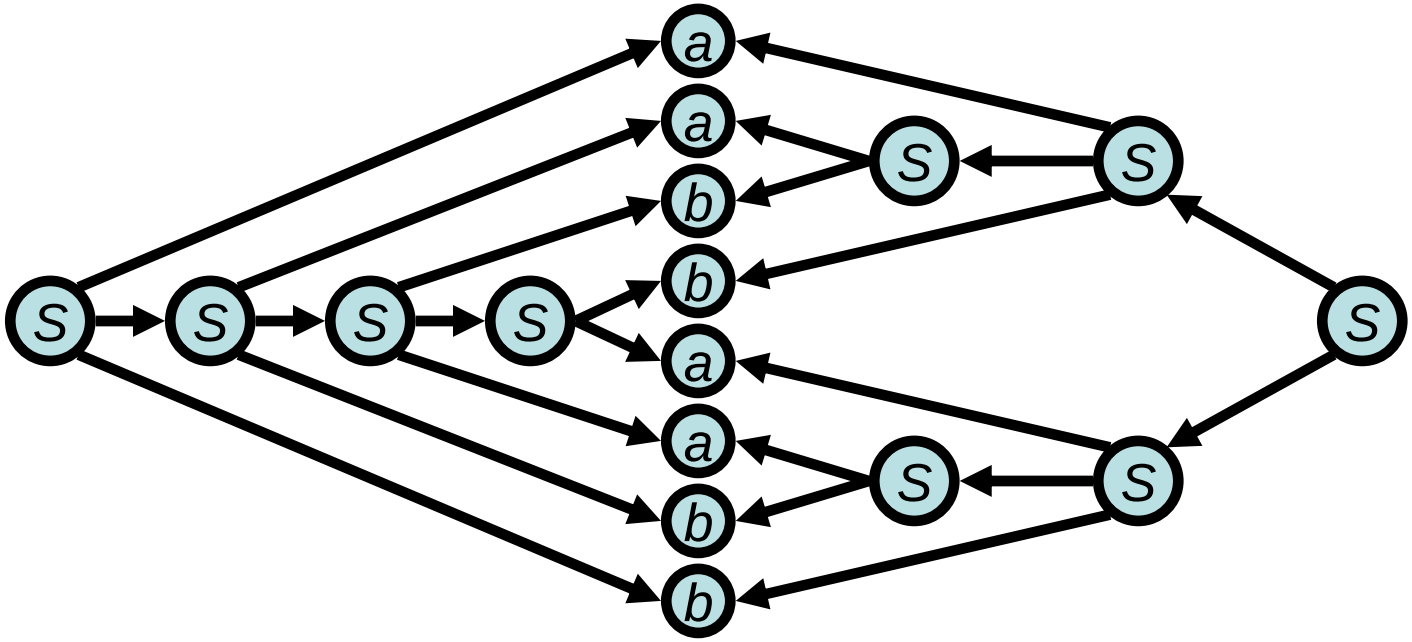
- Question: Is the grammar $S \rightarrow ab \mid ba \mid aSb \mid bSa \mid SS$ ambiguous?

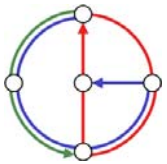
- What language is generated?



Ambiguity

- Answer: $L(G)$ = the language with equal no. of *a'* s and *b'* s
- Yes, the language is ambiguous:





CFG's: Proving Correctness

- The recursive nature of CFG's means that they are especially amenable to correctness proofs.

- For example let's consider the grammar

$$G = (S \rightarrow \varepsilon \mid ab \mid ba \mid aSb \mid bSa \mid SS)$$

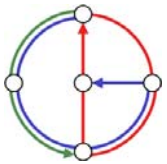
- We claim that L(G) = L = $\{ x \in \{a,b\}^* \mid n_a(x) = n_b(x) \}$, where $n_a(x)$ is the number of *a*'s in *x,* and $n_b(x)$ is the number of *b*'s.

- *Proof*: To prove that L = L(G) is to show both inclusions:

i. $L \subseteq L(G)$: Every string in L can be generated by *G*.

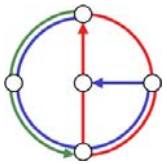
ii. $L \supseteq L(G)$: *G* only generate strings of L.

- This part is easy, so we concentrate on part i.



Proving $L \subseteq L(G)$

- $L \subseteq L(G)$: Show that every string x with the same number of a's as b's is generated by G. Prove by induction on the length $n = |x|$.
- Base case: The empty string is derived by $S \rightarrow \varepsilon$.
- Inductive hypothesis: Assume $n > 0$. Let u be the smallest non-empty prefix of x which is also in L.
  - Either there is such a prefix with $|u| < |x|$, then $x = uv$ whereas $v \in L$ as well, and we can use $S \rightarrow SS$ and repeat the argument.
  - Or $x = u$. In this case notice that u can't start and end in the same letter. If it started and ended with a then write $u = ava$. This means that v *must have* 2 more b's than a's. So somewhere in v the b's of u catch up to the a's which means that there's a smaller prefix in L, contradicting the definition of u as the *smallest* prefix in L. Thus for some string v in L we have $u = avb$ OR $u = bva$. We can use either $S \rightarrow aSb$ OR $S \rightarrow bSa$.



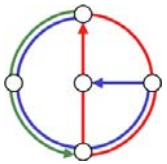
Designing Context-Free Grammars

- As for regular languages this is a creative process.

- However, if the grammar is the union of simpler grammars, you can design the simpler grammars (with starting symbols S_1, S_2, respectively) first, and then add a new starting symbol/production $S \rightarrow S_1 \mid S_2$.

- If the CFG happens to be regular as well, you can first design the FA, introduce a variable/production for each state of the FA, and then add a rule $x \rightarrow ay$ to the CFG if $\delta(x,a) = y$ is in the FA. If a state x is accepting in FA then add $x \rightarrow \varepsilon$ to CFG. The start symbol of the CFG is of course equivalent to the start state in the FA.

- There are quite a few other tricks. Try yourself…



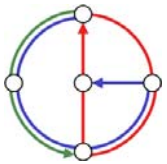
Push-Down Automata (PDA)

- As finite automata where the machine interpretation of regular languages, there is also a machine interpretation for grammars, called push-down automaton.

- The problem of finite automata was that they couldn't handle languages that needed some sort of unbounded memory, something that could be implemented easily by a single (unbounded) integer register!

- Example: To recognize the language $L = \{0^n 1^n \mid n \geq 0\}$, all you need is to count how many 0's you have seen so far…

- Push-Down Automata allow even more than a register: a full stack!

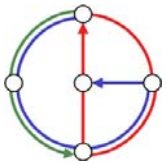


Recursive Algorithms and Stacks

- A stack allows the following basic operations
  - Push, pushing a new element on the top of the stack.
  - Pop, removing the top element from the stack (if there is one).
  - Peek, checking the top element without removing it.

- General Principle in Programming: Any recursive algorithm can be turned into a non-recursive one using a stack and a while-loop which exits only when stack is empty.

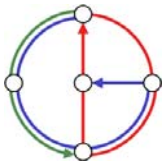
- It seems that with a stack at our fingertips we can even recognize palindromes! Yoo-hoo!
  - Palindromes are generated by the grammar S → ε | aSa | bSb.
  - Let's simplify for the moment and look at S → # | aSa | bSb.



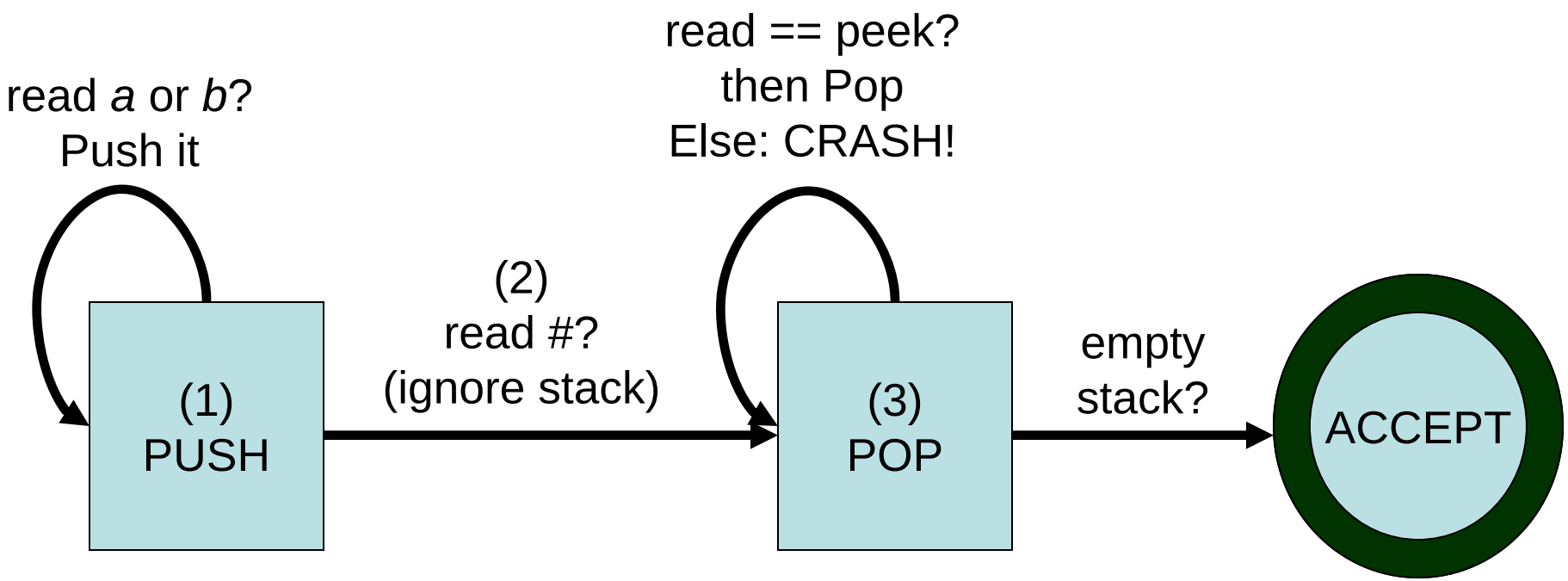
From CFG's to Stack Machines

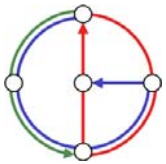
- The CFG S → # | aSa | bSb describes palindromes containing exactly one #-symbol. Using a stack, how can we recognize such strings?

- Answer: Use a three phase process:
  - Push mode: Before reading "#", push everything on the stack.
  - Reading "#" switches modes.
  - Pop mode: Read remaining symbols making sure that each new read symbol is identical to symbol popped from stack.
- Accept if able to empty stack completely. Otherwise reject. Also reject if could not pop somewhere.



From CFG's to Stack Machines

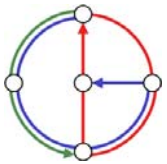




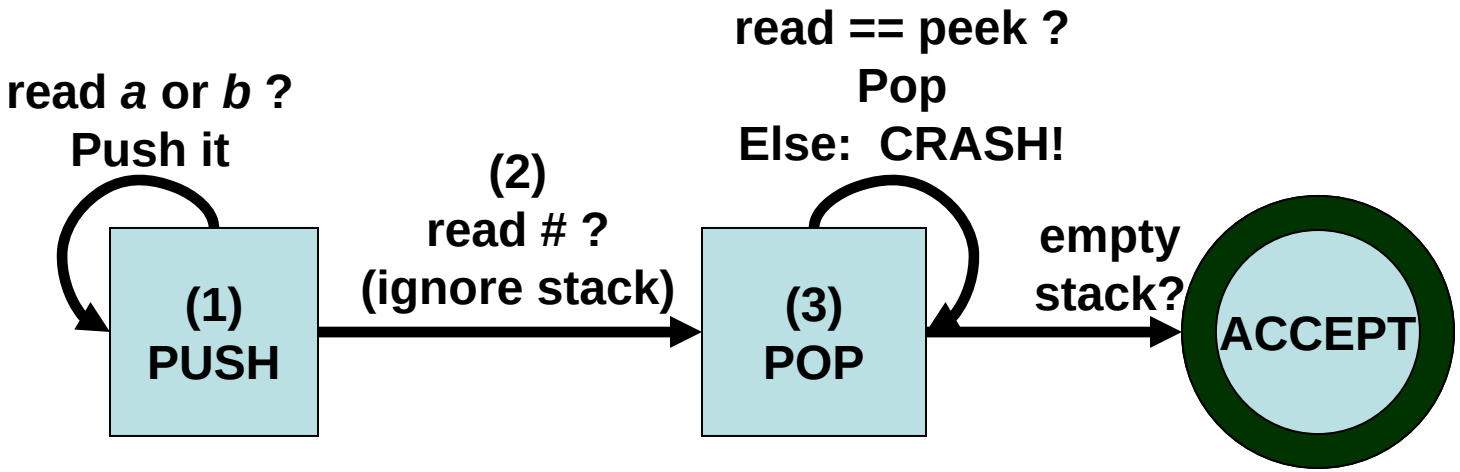
PDA's à la Sipser

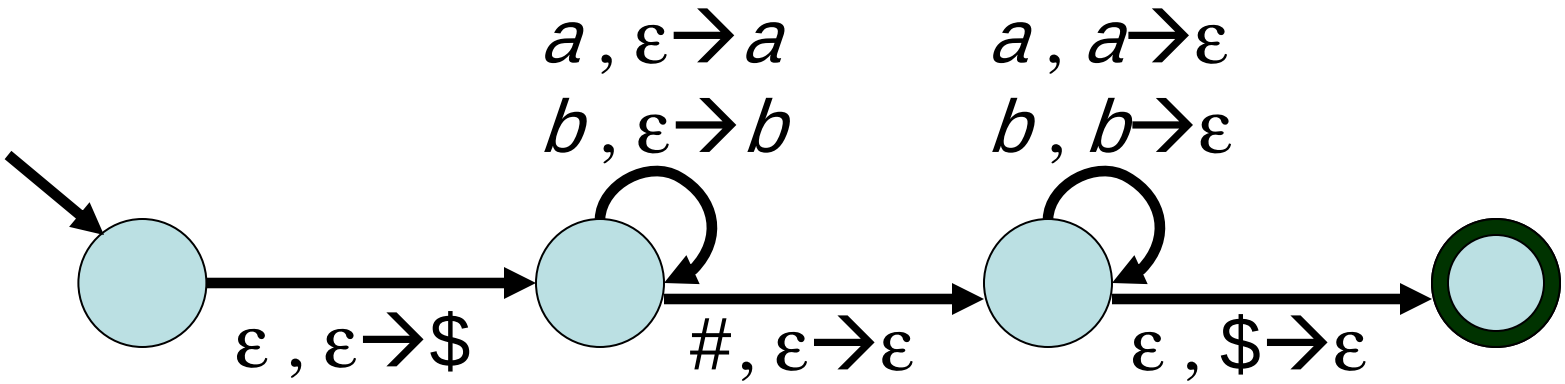
- To aid analysis, theoretical stack machines restrict the allowable operations. Each text-book author has his/her own version.

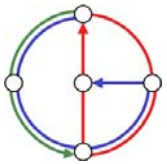
- Sipser's machines are especially simple:
  - Push/Pop rolled into a single operation: replace top stack symbol.
  - In particular, replacing top by ε is a pop.
- No intrinsic way to test for empty stack.
  - Instead often push a special symbol ("$") as the very first operation!
- Epsilon's used to increase functionality
  - result in default nondeterministic machines.



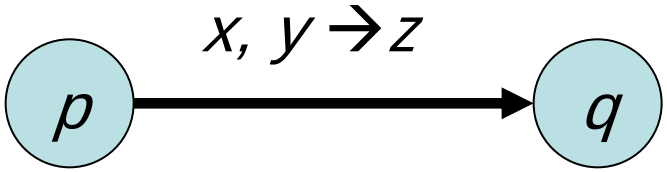
Sipser's Version







Sipser's Version



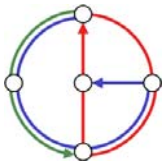
Meaning of labeling convention:

If at state *p* and next input is *x* and top stack is *y*,

then go to state *q* and replace *y* by *z* on stack.

- *x* = ε: ignore input, don't read
- *y* = ε: ignore top of stack and push *z*
- *z* = ε: pop *y*

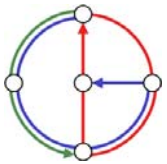
- In addition, push "$" initially to detect the empty stack.



PDA: Formal Definition

- Definition: A pushdown automaton (PDA) is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$:
  - Q, Σ, and q_0, and F are defined as for an FA.
  - Γ is the stack alphabet.
  - δ is as follows: Given a state p, an input letter x and a tape letter y, $\delta(p,x,y)$ gives all (q,z) where q is a target state and z a stack replacement for y.

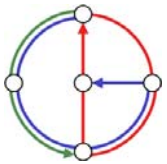
$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$$



PDA Exercises

- Draw the PDA $\{a^i b^j c^k \mid i,j,k \geq 0 \text{ and } i=j \text{ or } i=k\}$

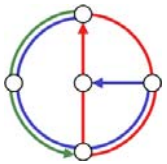
- Draw the PDA for $L = \{x \in \{a,b\}^* \mid n_a(x) = 2n_b(x)\}$



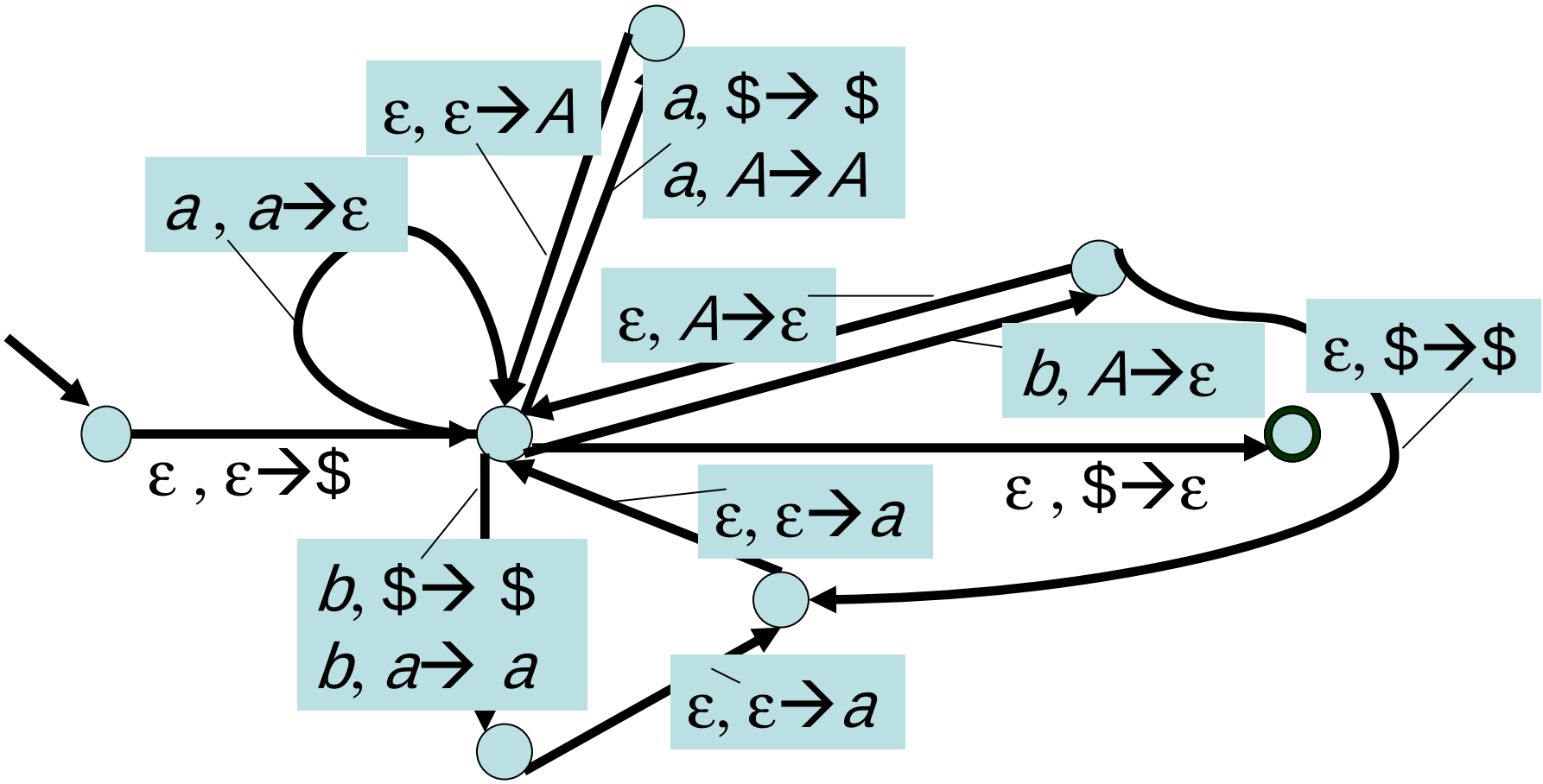
Solution 2

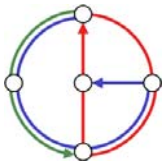
- The idea is to use the stack to keep count of the number of *a*'s and/or *b*'s needed to get a valid string. If we have a surplus of *b*'s thus far, we should have corresponding number of *a*'s (two for every *b*) on the stack. On the other hand, if we have a surplus of *a*'s we *cannot* put *b*'s on the stack since we can't split symbols. So instead, put two "negative" *a*-symbols, where a negative *a* will be denoted by capital *A*.

- Let's see how this looks on the next slide.



Solution 2

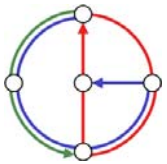




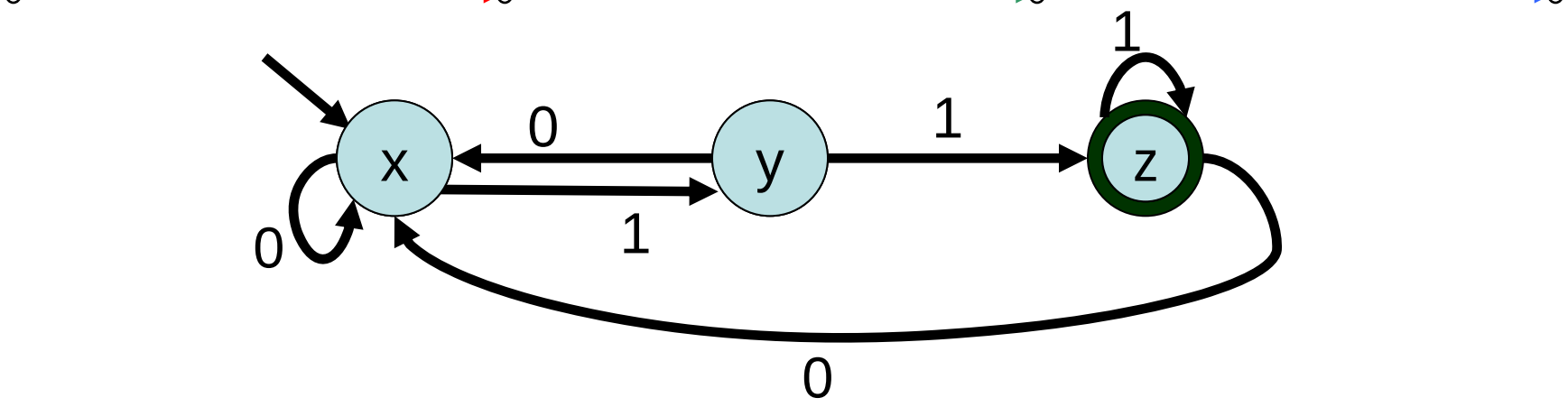
Model Robustness

- The class of regular languages was quite robust
  - Allows multiple ways for defining languages (automaton vs. regexp)
  - Slight perturbations of model do not change result (non-determinism)

- The class of context free languages is also robust, as you can use either PDA's or CFG's to describe the languages in the class. However, it is less robust when it comes to slight perturbations of the model:
  - Smaller classes
    - Right-linear grammars
    - Deterministic PDA's
  - Larger classes
    - Context Sensitive Grammars

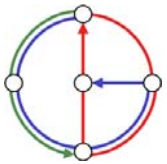


Right Linear Grammars and Regular Languages



- The DFA above can be simulated by the grammar
- $x \rightarrow 0x \mid 1y$
- $y \rightarrow 0x \mid 1z$
- $z \rightarrow 0x \mid 1z \mid \varepsilon$

- Definition: A right-linear grammar is a CFG such that every production is of the form $A \rightarrow uB$, or $A \rightarrow u$ where u is a terminal string, and A,B are variables.

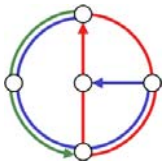


Right Linear Grammars vs. Regular Languages

- Theorem: If $M = (Q, \Sigma, \delta, q_0, F)$ is an NFA then there is a right-linear grammar $G(M)$ which generates the same language as M.

- *Proof*:
  - Variables are the states: $V = Q$
  - Start symbol is start state: $S = q_0$
  - Same alphabet of terminals Σ
  - A transition $q_1 \rightarrow a \rightarrow q_2$ becomes the production $q_1 \rightarrow a q_2$
  - Accept states $q \in F$ define the ε-productions $q \rightarrow \varepsilon$

- Question: Can every CFG be converted into a right-linear grammar?
- Answer: No! This would mean that all context free languages are regular. But, for example, $\{a^n b^n\}$ is not regular.

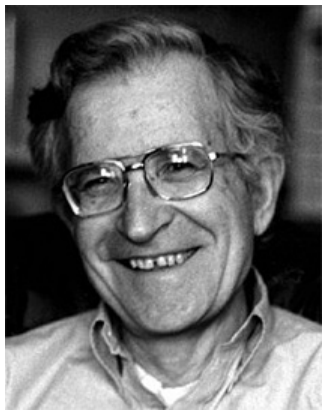


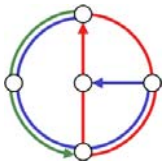
Chomsky Normal Form

- Chomsky came up with an especially simple type of context free grammars which is able to capture all context free languages, the Chomsky normal form (CNF).

- Chomsky's grammatical form is particularly useful when one wants to prove certain facts about context free languages. This is because assuming a much more restrictive kind of grammar can often make it easier to prove that the generated language has whatever property you are interested in.

- Noam Chomsky, linguist at MIT, creator of the Chomsky hierarchy, a classification of formal languages. Chomsky is also widely known for his left-wing political views and his criticism of the foreign policy of U.S. government.



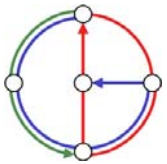


Chomsky Normal Form

- Definition: A CFG is said to be in Chomsky Normal Form if every rule in the grammar has one of the following forms:

  - $S \rightarrow \varepsilon$ (ε for epsilon's sake only)
  - $A \rightarrow BC$ (dyadic variable productions)
  - $A \rightarrow a$ (unit terminal productions)

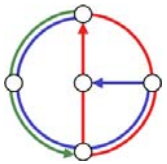
  where S is the start variable, A,B,C are variables and a is a terminal. Thus epsilons may only appear on the right hand side of the start symbol and other rights are either 2 variables or a single terminal.



CFG → CNF

- Converting a general grammar into Chomsky Normal Form works in four steps:

1. Ensure that the start variable doesn't appear on the right hand side of any rule.
2. Remove all epsilon productions, except from start variable.
3. Remove unit variable productions of the form $A \rightarrow B$ where A and B are variables.
4. Add variables and dyadic variable rules to replace any longer non-dyadic or non-variable productions



CFG → CNF: Example

$$S \to \varepsilon \mid a \mid b \mid aSa \mid bSb$$

1. No start variable on right hand side

$$S' \to S$$

$$S \to \varepsilon \mid a \mid b \mid aSa \mid bSb$$

2. Only start state is allowed to have ε

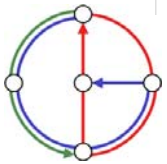
$$S' \to S \mid \varepsilon$$

$$S \to \cancel{\varepsilon} \mid a \mid b \mid aSa \mid bSb \mid aa \mid bb$$

3. Remove unit variable productions of the form $A \to B$.

$$S' \to \cancel{S} \mid \varepsilon \mid a \mid b \mid aSa \mid bSb \mid aa \mid bb$$

$$S \to a \mid b \mid aSa \mid bSb \mid aa \mid bb$$



CFG → CNF: Example continued

$$S' \to \cancel{S} \mid \varepsilon \mid a \mid b \mid aSa \mid bSb \mid aa \mid bb$$

$$S \to a \mid b \mid aSa \mid bSb \mid aa \mid bb$$

4. Add variables and dyadic variable rules to replace any longer productions.

$$S' \to \varepsilon \mid a \mid b \mid \cancel{aSa} \mid \cancel{bSb} \mid \cancel{aa} \mid \cancel{bb} \mid AB \mid CD \mid AA \mid CC$$

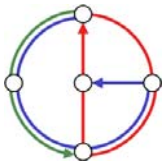
$$S \to a \mid b \mid \cancel{aSa} \mid \cancel{bSb} \mid \cancel{aa} \mid \cancel{bb} \mid AB \mid CD \mid AA \mid CC$$

$$A \to a$$

$$B \to SA$$

$$C \to b$$

$$D \to SC$$

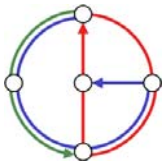


CFG → PDA

- CFG's can be converted into PDA's.

- In "NFA → REX" it was useful to consider GNFA's as a middle stage. Similarly, it's useful to consider Generalized PDA's here.

- A Generalized PDA (GPDA) is like a PDA, except it allows the top stack symbol to be replaced by a whole string, not just a single character or the empty string. It is easy to convert a GPDA's back to PDA's by changing each compound push into a sequence of simple pushes.



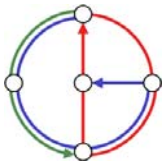
CFG → GPDA Recipe

1. Push the marker symbol $ and the start symbol S on the stack.
2. Repeat the following steps forever

   a. If the top of the stack is the variable symbol A, nondeterministically select a rule of A, and substitute A by the string on the right-hand-side of the rule.

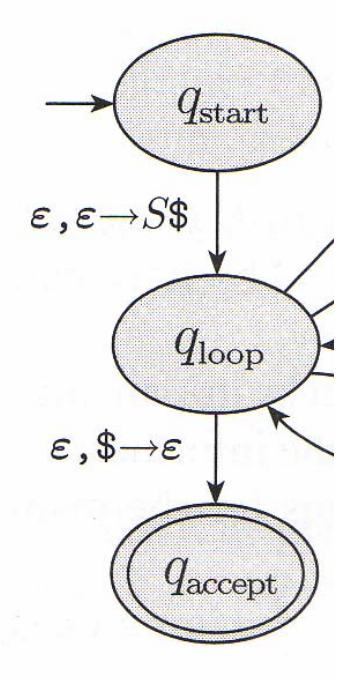
   b. If the top of the stack is a terminal symbol a, then read the next symbol from the input and compare it to a. If they match, continue. If they do not match reject this branch of the execution.

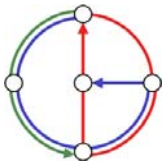
   c. If the top of the stack is the symbol $, enter the accept state. (Note that if the input was not yet empty, the PDA will still reject this branch of the execution.)



CFG → PDA: Example

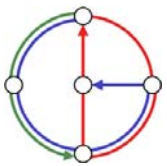
- S → aTb | b
- T → Ta | ε





CFG → PDA: Now you try!

- Convert the grammar $S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$



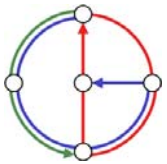
Context Sensitive Grammars

- An even more general form of grammars exists. In general, a non-context free grammar is one in which whole mixed variable/terminal substrings are replaced at a time. For example with $\Sigma = \{a,b,c\}$ consider:

$S \rightarrow \varepsilon \mid ASBC$
$A \rightarrow a$
$CB \rightarrow BC$
$aB \rightarrow ab$
$bB \rightarrow bb$
$bC \rightarrow bc$
$cC \rightarrow cc$

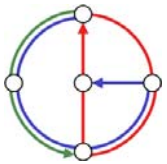
What language is generated by this non-context-free grammar?

- For technical reasons, when length of LHS always ≤ length of RHS, these general grammars are called context sensitive.



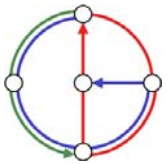
PDA → CFG

- To convert PDA's to CFG's we'll need to simulate the stack inside the productions.
- Unfortunately, in contrast to our previous transitions, this is not quite as constructive. We will therefore only state the theorem.
- Theorem: For each push-down automation there is a context-free grammar which accepts the same language.
- Corollary: PDA ≈ CFG.



Are all languages context-free?

- Design a CFG (or PDA) for the following languages:
- L = { w $\in$ {0,1,2}* | there are *k* 0's, *k* 1's, and *k* 2's for *k* $\geq$ 0 }
- L = { w $\in$ {0,1,2}* | with |0| = |1| or |0| = |2| or |1| = |2| }
- L = { $0^k1^k2^k$ | k $\geq$ 0 }

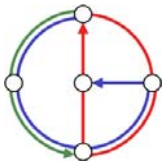


Tandem Pumping

- Analogous to regular languages there is a pumping lemma for context free languages. The idea is that you can pump a context free language at two places (but not more).

- Theorem: Given a context free language L, there is a number p (tandem-pumping number) such that any string in L of length $\geq p$ is tandem-pumpable within a substring of length p. In particular, for all $w \in L$ with $|w| \geq p$ we we can write:
  - $w = uvxyz$
  - $|vy| \geq 1$ (pumpable areas are non-empty)
  - $|vxy| \leq p$ (pumping inside length-p portion)
  - $uv^ixy^iz \in L$ for all $i \geq 0$ (tandem-pump v and y)

- If there is no such p the language is not context-free.

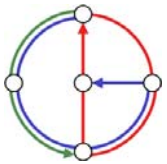


Proving Non-Context Freeness: Example

- $L = \{1^n 0^n 1^n 0^n \mid n \text{ is non-negative}\}$
- Let's try $w = 1^p 0^p 1^p 0^p$. Clearly $w \in L$ and $|w| \geq p$.
- With $|vxy| \leq p$, there are only three places where the "sliding window" *vxy* could be:

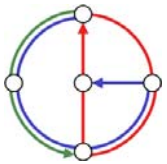
$$\underbrace{1 \ldots 1}_{\text{I}}\underbrace{0 \ldots 0}_{\text{II}}\underbrace{1 \ldots 1}_{\text{III}}0 \ldots 0$$

- In all three cases, pumping up such a case would only change the number of 0s and 1s in that part and not in the other two parts; this violates the language definition.



Proving Non-Context Freeness: You try!

- L = { x=y+z | x, y, and z are binary bit-strings satisfying the equation }
- The hard part is to come up with a word which cannot be pumped, such as

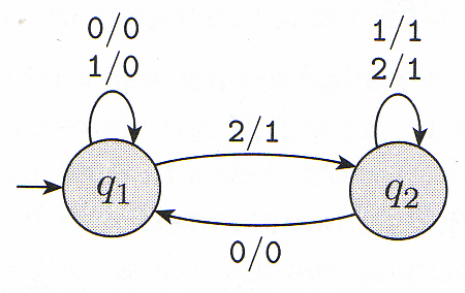


Transducers

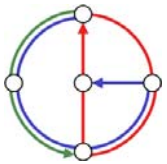
- Definition: A finite state transducer (FST) is a type of finite automaton whose output is a string and not just accept or reject.

- Each transition of an FST is labeled with two symbols, one designating the input symbol for that transition (as for automata), and the other designating the output symbol.
  - We allow ε as output symbol if no symbol should be added to the string.

- The figure on the right shows an example of a FST operating on the input alphabet {0,1,2} and the output alphabet {0,1}



- Exercise: Can you design a transducer that produces the inverted bit-string of the input string (e.g. 01001 → 10110)?

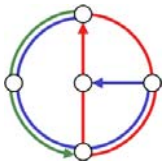


Even smarter automata…

- Even though the PDA is more powerful than the FA, Arnold Schwarzenegger would probably still call it a "girlie-machine," since it doesn't understand a lot of important languages.

- Let's try to make it more powerful by adding a second stack
  - You can push or pop from either stack, also there's still an input string
  - Clearly there are quite a few "implementation details"
  - It seems at first that it doesn't help a lot to add a second stack, but…

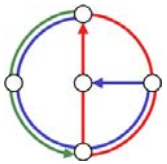
- Lemma: A PDA with two stacks is as powerful as a machine which operates on an infinite tape (restricted to read/write only "current" tape cell at the time – known as "Turing Machine").
  - Still that doesn't sound very exciting, does it…?!?



Turing Machine

- A Turing Machine (TM) is a device with a finite amount of *read-only* "*hard*" memory (states), and an unbounded amount of read/write tape-memory. There is no separate input. Rather, the input is assumed to reside on the tape at the time when the TM starts running.

- Just as with Automata, TM's can either be input/output machines (compare with Finite State Transducers), or yes/no decision machines.

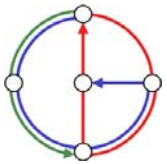


Turing Machine: Example Program

- Sample Rules:
  - If read 1, write 0, go right, repeat.
  - If read 0, write 1, HALT!
  - If read , write 1, HALT! (the symbol stands for the blank cell)

- Let's see how these rules are carried out on an input with the *reverse* binary representation of 47:

1	1	1	1	0	1				

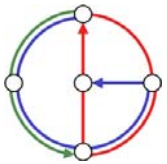


Turing Machine: Formal Definition

- Definition: A Turing machine (TM) consists of a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$.
  - Q, Σ, and q_0, are the same as for an FA.
  - q_{acc} and q_{rej} are accept and reject states, respectively.
  - Γ is the tape alphabet which necessarily contains the blank symbol , as well as the input alphabet Σ.
  - δ is as follows:

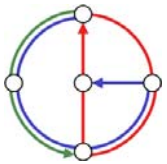
$$\delta : (Q - \{q_{acc}, q_{rej}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

  - Therefore given a non-halt state *p,* and a tape symbol *x*, $\delta(p,x)$ = (*q,y,*D) means that TM goes into state *q,* replaces *x* by *y*, and the tape head moves in direction D (left or right).
- A string *x* is accepted by *M* if after being put on the tape with the Turing machine head set to the left-most position, and letting *M* run, *M* eventually enters the accept state. In this case *w* is an element of *L*(*M*) – the language accepted by *M.*



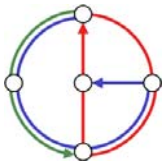
Comparison

Device	Separate Input?	Read/Write Data Structure	Deterministic by default?
FA	Yes	None	Yes
PDA	Yes	LIFO Stack	No
TM	No	1-way infinite tape. 1 cell access per step.	Yes (but will also allow crashes)



Alan Turing

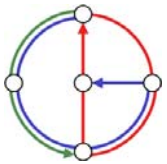
- 1912 – 1954, British mathematician
- one of the fathers of computer science
- his computer model – the Turing Machine – was inspiration/premonition of the electronic computer that came two decades later
- during World War II he worked on breaking German cyphers, particularly the Enigma machine.
- Invented the “Turing Test” used in Artificial Intelligence
- Legacy: The Turing Award.
  “Nobel prize” in computer science



Back to the Turing Machine

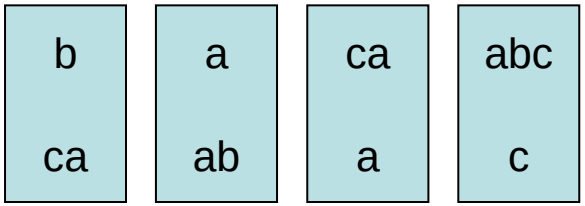
- First Goal of Turing's Machine: A "computer" which is as powerful as any real computer / programming language
  - As powerful as C, or "Java++"
  - Can execute all the same algorithms / code
  - Not as fast though (move the head left and right instead of RAM)
  - Historically: A model that can compute anything that a human can compute. Before invention of electronic computers the term "computer" actually referred to a *person* who's line of work is to calculate numerical quantities!
  - This is known as the [Church-[Post-]] Turing thesis, 1936.

- Second Goal of Turing's Machine: And at the same time a model that is simple enough to actually prove interesting epistemological results.

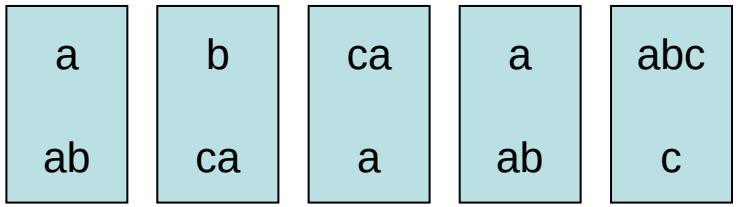


Can a computer compute anything…?!?

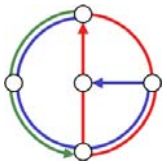
- Given collection of dominos, e.g.



- Can you make a list of these dominos (repetitions are allowed) so that the top string equals the bottom string, e.g.

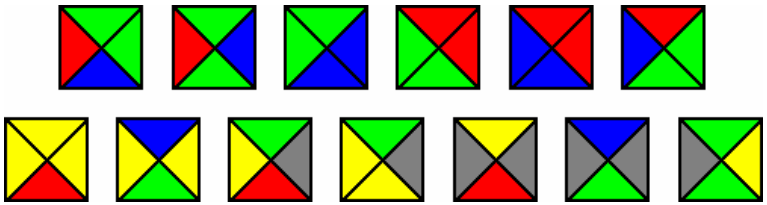


- This problem is known as Post-Correspondance-Problem.
- It is provably unsolvable by computers!

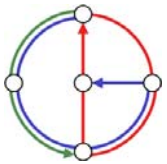


Also the Turing Machine (the Computer) is limited

- Similary it is undecidable whether you can cover a floor with a given set of floor tiles (famous examples are Penrose tiles or Wang tiles)



- Examples are leading back to Kurt Gödel's incompleteness theorem
  - "Any powerful enough axiomatic system will allow for propositions that are undecidable."

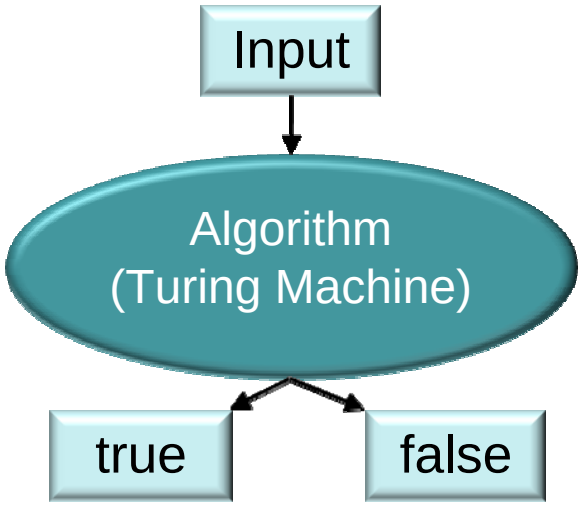


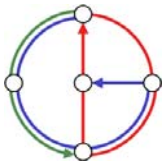
Decidability

- A function is computable if there is an algorithm (according to the Church-Turing-Thesis a turing machine is sufficient) that computes the function (in finite time).

- A subset T of a set M is called decidable (or recursive), if the function f: M → {true, false} with f(m) = true if m∈T, is computable.

- A more general class are the semi-decidable problems, for which the algorithm must only terminate in finite time in either the true or the false branch, but not the other.



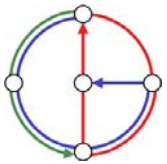


Halting Problem

- The halting problem is a famous example of an undecidable (semi-decidable) problem. Essentially, you cannot write a computer program that decides whether another computer program ever terminates (or has an infinite loop) on some given input.

- In pseudo code, we would like to have:

```
procedure halting(program, input) {
     if program(input) terminates
     then return true
     else return false
}
```



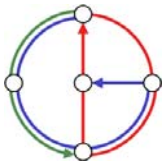
Halting Problem Proof

- Now we write a little wrapper around our halting procedure

```
procedure test(program) {
     if halting(program,program)=true
     then loop forever
     else return
}
```

- Now we simply run: `test(test)`! Does it halt?!?
  - If halting(test,test)=true, test(test) should terminate, but it does not!
  - If halting(test,test)=false, test(test) should not terminate, but it does!

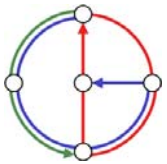
- Wicked! Our (generic) halting procedure is wrong, no matter what!
- We have a contradiction. No halting procedure can exist.



Excursion: P and NP

- P is the complexity class containing decision problems which can be solved by a Turing machine in time polynomial of the input size.

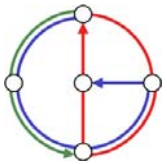
- NP is the class of decision problems solvable by a non-deterministic polynomial time Turing machine such that the machine answers "yes," if at least one computation path accepts, and answers “no,” if all computation paths reject.
  - Quite similarly to the nondeterministic finite automaton from Chapter 1.
  - Informally, there is a Turing machine which can check the correctness of an answer in polynomial time.
  - E.g. one can check in polynomial time whether a traveling salesperson path connects *n* cities with less than a total distance *d*.
  - Or one can check in polynomial time whether two big numbers are the factors of an even bigger number (with *n* digits).



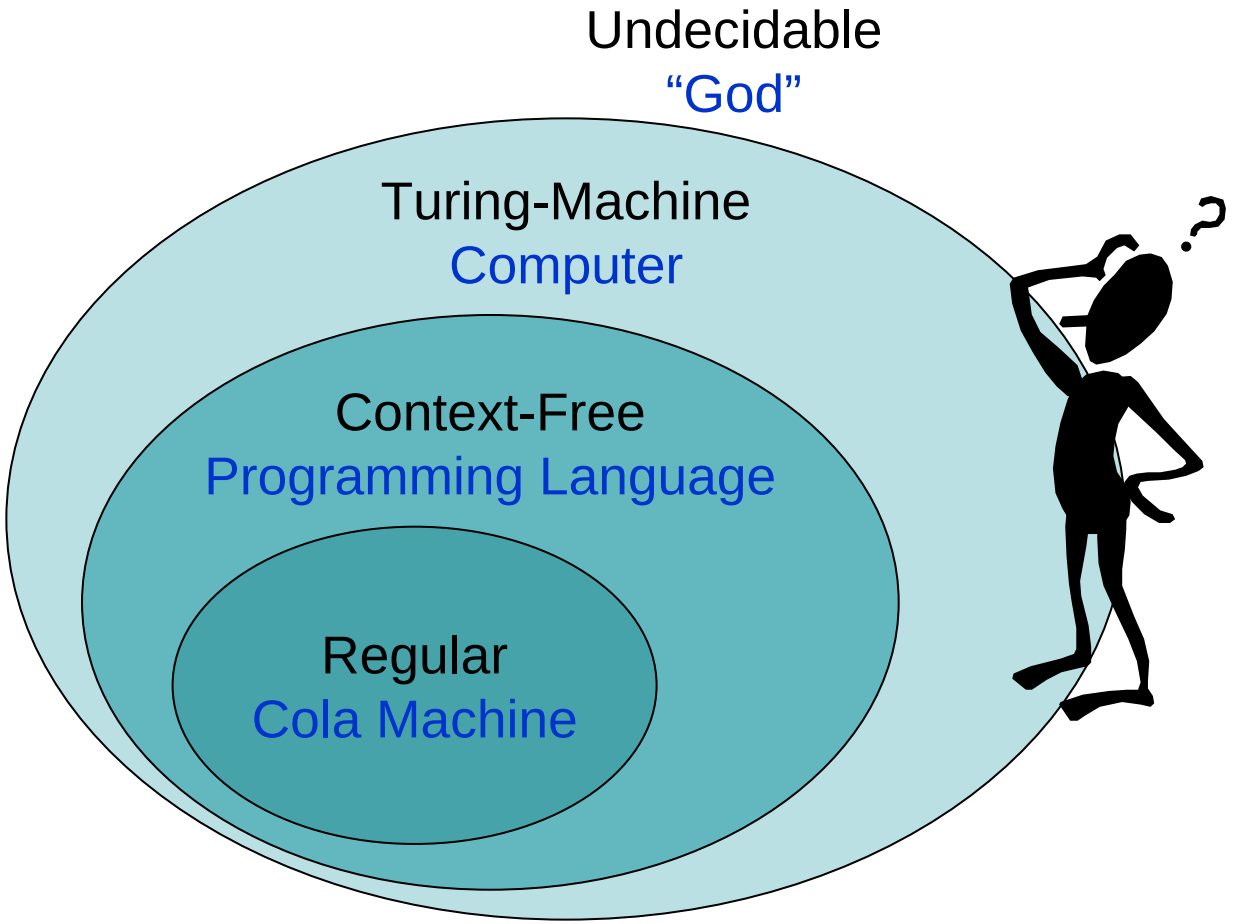
P vs. NP

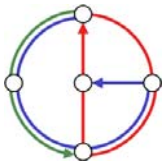
- An important notion in this context is the large set of NP-complete decision problems, which is a subset of NP and might be informally described as the "hardest" problems in NP. If there is a polynomial-time algorithm for even one of them, then there is a polynomial-time algorithm for all the problems in NP.
  - E.g. Given a set of n integers, is there a non-empty subset which sums up to 0? This problem was shown to be NP-complete.
  - Also the traveling salesperson problem is NP-complete, or Tetris, or Minesweeper.

- One of the big questions in Math and CS: Is P = NP?
  - Or are there problems which cannot be solved in polynomial time.
  - Big practical impact (e.g. in Cryptography).
  - One of the seven $1M problems by the Clay Mathematics Institute of Cambridge, Massachusetts.



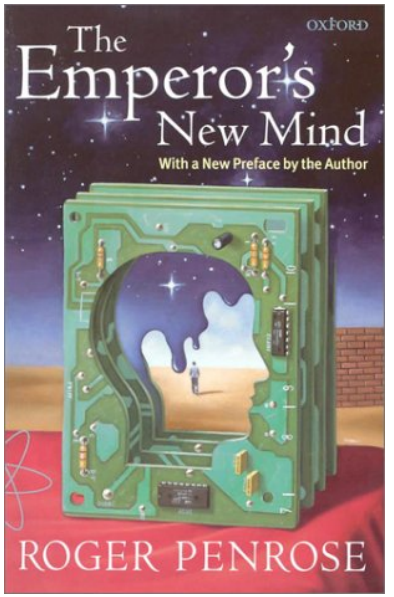
Summary

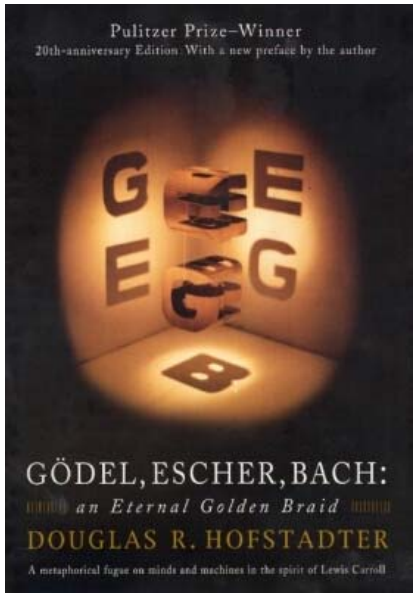

Undecidable
“God”
Turing-Machine
Computer
Context-Free
Programming Language
Regular
Cola Machine



Bedtime Reading

If you're leaning towards "human = machine"





If you're leaning towards "human $\supset$ machine"