28/9/2018

Does a system do what its supposed to do?

SPEC (specification of behavior you want)
↳ written in natural language / logic

BEHAVIOR : sequence of actions

Are sequence of actions good enough to capture all aspects of behavior?

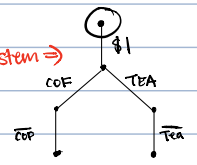
Model Checking
↳ check systems w/ 10^{600} states

Design of Vending Machine

$\Sigma = \{\$1, COF, TEA, \overline{COF}, \overline{TEA}\}$ (actions = alphabet)
↳ $1 = put in a dollar
↳ COF = button of coffee
↳ $\overline{COF}$ = dispense coffee

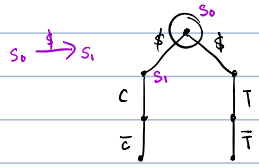
* actions may be rejected, unlike DFA
* systems may be indeterminate
↳ do action, & next state is not fixed, like NFA

Interactive system ⇒ (determinate)



2 possible behaviors: $\{(\$, c, \bar{c}), (\$, T, \bar{T})\}$
↳ w/o any money, those behaviors are rejected

Indeterminate Machine: (same state may lead to 2 actions)



- machine make a nondeterministic choice
- it chooses for you
$\{(C\$, c, \bar{c}), (\$, T, \bar{T})\}$
(same behaviors!)

Sequences do not capture all aspects of behavior. We need to capture branching

Definition of Bisimulation

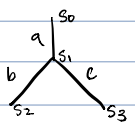
An equivalence relation on states "$\sim$"
ex. $s \sim t$ means you can't tell whether you start from s or from t
↳ ex can look @ start states of vending machine ☆

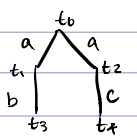
s & t are similar (behave same) if for any action on s to s' I should be able to match it w/ the same action of t & reach t' ⇒ s' & t' are still related.

$s \sim t$ (bisimular)
$\forall a$ (ACTION) $s \xrightarrow{a} s'$ $\exists t'$ s.t. $t \xrightarrow{a} t'$ and $s' \sim t'$
& $\forall a$ $t \xrightarrow{a} t''$ $\exists s''$ s.t $s \xrightarrow{a} s''$ & $s'' \sim t''$

- every state is bisimilar to itself
- bisimulation is co-inductive
↳ start from the dead states

ex. vending machine





Found:
* $s_2 \sim s_3 \sim t_3 \sim t_4$ ∵ dead states; last states & can't do anything
* $s_1 \overset{?}{\sim} t_1$ NO $s_1 \not\sim t_1$, $s_1 \not\sim t_2$
↳ $s_1 \xrightarrow{c} s_3$ & not possible for t_1 (i.e. only $t_1 \xrightarrow{b} t_3$)
* $s_0 \overset{?}{\sim} t_0$ NO
$s_0 \xrightarrow{a} s_1$ $t_0 \xrightarrow{a} t_1$ BUT $s_1 \not\sim t_1$
then, $t_0 \xrightarrow{a} t_2$ BUT $s_1 \not\sim t_2$
∴ matches for only one action but not continous matching

THEOREM If 2 states are bisimilar, then no interactive experiments can reveal the differences

15/10/2018

LINGUISTICS

- how can we define languages?
- other languages that cannot be defined

ex. regular languages
↳ defined by regex
↳ recognized by finite automata (recognize strings)
↳ pumping lemma to prove L is NOT regular

For linguistics, the goal is to generate strings by Grammar

Context-Free Grammar (Chomsky) (CFG)

Definition of CFG:

- consists of 4 things
① a set of symbols (Σ or T) called terminals
② another set of symbols (disjoint w/①) called non-terminals (V) ⇒ $V \cap T = \emptyset$
③ a special non terminal $S \in V$: the start symbol
④ a finite set of rules for productions

↳ Forms of production: (w/ a nonterminal, a rule allows you to write to the string α)

$A \in V \qquad A \xrightarrow{\text{rule}} \alpha \qquad \alpha \in (V \cup T)^*$

↳ string can have som variables & (terminal symbols ⇒ can't be rewritten)

↳ can only have T on the left side

Example:

$T = \{a, b\} \quad V = \{S\}$

$S \longrightarrow \varepsilon$ (take start symbol & erase it)

$S \longrightarrow aSb$ (to mixed string)

*example strings in the language

$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aaabbb$ (used $S \rightarrow \varepsilon$)

a & b remains — rewrite of S, ignoring context (∴ context free)

$\therefore L(G) = \{a^n b^n \mid n \geq 0\}$

All regular languages are context free

* $a^n b^n c^n$ is not CFG!

Language of Arithmetic Expressions

$T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *, (,)\}$

$V = \{\langle EXP\rangle, \langle NUM\rangle, \langle NZ\rangle, \langle N\rangle\}$ (start, num, nonzero digits, digits w/ zero)

* $\langle NUM\rangle \rightarrow 0 \mid \langle NZ\rangle$: <NUM> can be 0 or a nonzero number (2 rules, separated w/ '|')

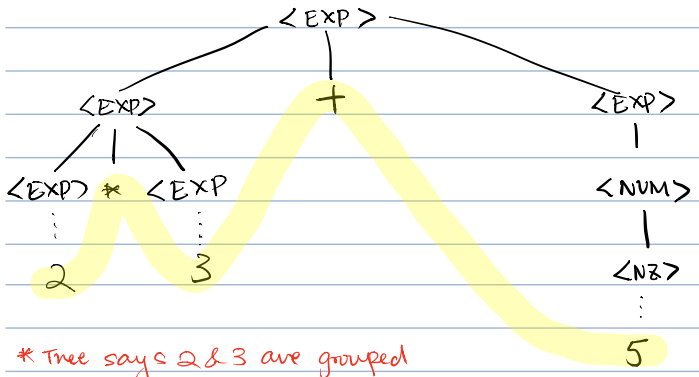
* $\langle NZ\rangle \rightarrow 1\langle N\rangle \mid 2\langle N\rangle \mid \dots \mid 9\langle N\rangle \mid \varepsilon$: <NZ> has 10 diff rules (<n>, 2<n>..., 9<n>, ε)

* $\langle N\rangle \rightarrow 0\langle N\rangle \mid 1\langle N\rangle \mid \dots \mid 9\langle N\rangle \mid \varepsilon$

* $\langle EXP\rangle \rightarrow \langle EXP\rangle + \langle EXP\rangle \mid \langle EXP\rangle * \langle EXP\rangle \mid (\langle EXP\rangle) \mid \langle NUM\rangle$

Sequences don't tell the story!

ex. $2 * 3 + 5 = 11$ ∵ BEDMAS (more in prof notes)



* Tree says 2 & 3 are grouped more tightly & 5 is more wisely bound.

↳ ∴ math grammar is ambiguous

↳ arith expressions itself don't tell us BEDMAS

↳ there's the underlying function of trees

Ambiguous Grammar

Definition: Grammar is ambiguous if there are 2 distinct parse trees for the SAME string

↳ property of Grammar, NOT language

* but there are language where its impossible to design ambiguous grammar

↳ called inherently-ambiguous languages (≠ NOT for us this class)

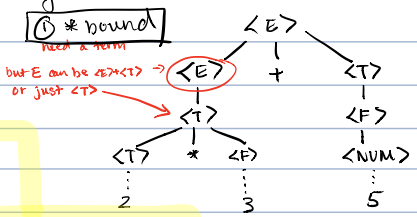
17/10/2018

Recap: CFG & CFL (a language that can be defined by a CFG)

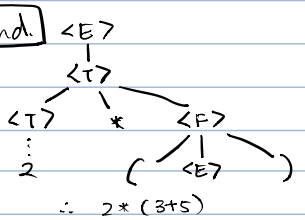
- no unique grammar for a language
- ex $2 * 3 + 5$ (want this part to be grouped more closely)
- $V = \{\langle E\rangle, \langle T\rangle, \langle F\rangle\}$ not showing <N> etc.

Start <E> Term Factor

① * bound (need a term) — but E can be <E>+<T> or just <T>



② + bound.



* but not the same string (i.e. has ())

$\langle E\rangle \rightarrow \langle E\rangle + \langle T\rangle \mid \langle T\rangle$ ⇒ an expression can be ① an exp plus a term or ② just a term

$\langle T\rangle \rightarrow \langle T\rangle * \langle F\rangle \mid \langle F\rangle$ ⇒ terms can have products; all left recurses

$\langle F\rangle \rightarrow (\langle E\rangle) \mid \langle NUM\rangle$ ⇒ factors can be an (expression) or a number

Tree structure is important for meaning!

ex. $\{a^n b^{2n} \mid n \geq 0\}$ NOT REGULAR

$S \to aSbb \mid \varepsilon$

ex. (((() ()) ()) ()))) how do you define a balanced sequence?

- ex in HTML ⇒ Every prefix has greater or equal # of left parenthesis "(" as right ")"
- CFG is good for matching tags

ex. $L = \{x \in \Sigma^* \mid x \text{ has as many a's as b's}\}$

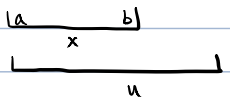
$d(x) = \#_b(x) - \#_a(x)$ ⇒ define function d(x) = (# of b's in x) − (# of a's in x)

$L = \{x \mid d(x) = 0\}$

Suppose $u \in L$ & u starts w/ an a

$u = aw$ $d(w) = 1$ (w has at least one b more than a)

Let x be the shortest prefix s.t. $d(x) = 0$



⇒ x starts w/ a ∴ u starts w/ a.

⇒ last letter of x has to be b

$x = avb$

$u = avby$

∵ (avb) & u has an equal # of a's & b's

∴ y also has an equal # of a's & b's

∴ $v, y \in L$

∴ $S \to aSbS \mid bSaS \mid \varepsilon$, OR $S \to aSb \mid bSa \mid \varepsilon \mid SS$,

↳ are these 2 grammars equivalent?

Need to prove 2 things:

① Any word generated is in L

② Every word in L can be generated

} proof in prof notes

ex. palindromes $L = \{x \mid x = x^{REV}\}$ $\Sigma = \{a, b\}$

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

single word palindromes

If the word starts w/ an a or b, it also needs to end in a or b

① $S \to aSb \mid bSa \mid SS \mid \varepsilon$

Claim: Any word generated has $\#_a(w) = \#_b(w)$

⇒ pairs aSb & bSa says every a comes w/ b & every b comes w/ a (PROVED)

Any word w/ equal # of a's & b's can be generated.

Proof: By induction on the # of a's

Base: use ε string (i.e. w/ fewest # of a's) ∴ a# = b# = 0

Inductive Step: Assume any string w/ up to n a's can be generated & consider string w/ n+1 a's

ex. a w b $S \to aSb \to \cdots awb$ by inductive hypothesis

↳ if whole string has #a = #b, then so does w

∴ if whole string has (n+1) a's, then w has n a's

Suppose w begins & ends w/ a (same letter)

$w = aw'a$ w' has 2 more b's than a's

$\therefore S \to aw_1$, $S \to w_2a$

$\therefore S \to SS \to \cdots aw_1w_2a$

★ If you have a function like d which can only change by ±1 at each step & it starts out −ve & ends up +ve it must hit zero

Chomsky Normal Form

⇒ Every CFG has a grammar in which the rules are of a very restricted form.

$N \to a$ (one non-terminal go to one terminal)

$A \to BC$ (one nonterminal go to two nonterminals)

} so any grammar can be put into this form (CFG → CNF)

The only non-terminal allowed to go ε is the start terminal.

19/10/2018

- Recognizing context-free languages (Pushdown Automata)
  - ↳ CFL described through CFG
- ex. Regular language
  - describe through regex
  - recognize w/ DFA/NFA/NFA+ε

Pushdown Automata

- Its a stack machine
- composed of a DFA + 1 Stack

ex. How do we recognize $L = \{a^n b^n \mid n \geq 0\}$?

- nonregular language (via pumping lemma)
- can't used finite automatons b/c they can't do unbounded counting
  - ↳ by stacks are unbounded (~infinite memory)
- * CFL = Recognized by PDA
  - ⇒ default PDA is nondeterministic

Definition: PDA

Q: set of states

Σ: input alphabet

q_0: start state

Γ: stack alphabet (usually includes the Σ + some extra symbols)

↳ not actually reading these symbols, used to keep track of things on the stack

$F \subseteq Q$: a family of accept states

$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ $\quad \Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$

$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}_{fin}(Q \times \Gamma_\varepsilon)$: transition function

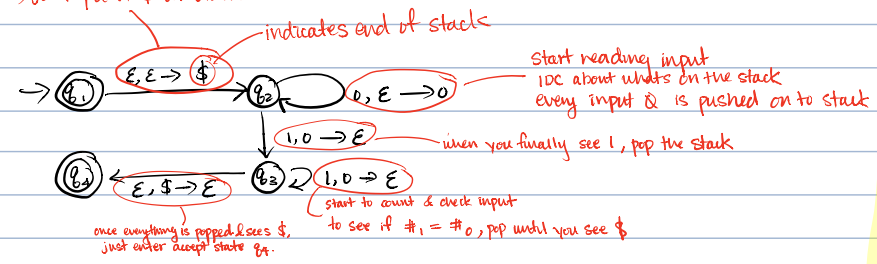
- looks @ state (Q) & input alphabet (Σ_ε) & top of stack (Γ_ε)
- the move is nondeterministic ($\to \mathcal{P}_{fin}$) [i.e. a finite set of things that can happen / outcomes]
  - ↳ these outcomes corresponds to a new state & new thing on stack ($Q \times \Gamma_\varepsilon$)
  - ↳ $\mathcal{P}$ stands for power set (finite); can be ε

Functions of PDA

- look @ input & top of stack & pop stack, push new symbol & change state
  & does not have to do all these functions (i.e. can just only change state)
- notion of acceptance: some path may lead to accept state, & just accept

Example PDA: $L = \{0^n 1^n \mid n \geq 0\}$

This machine is deterministic



Decoration of Arrows

a, b → c

MEANS: see a in the input & b ontop of stack
replace b w/ c on the stack

* a, b, c can be ε
  - ↳ if a = ε: don't look @ input
  - ↳ if b = ε: just push c
  - ↳ if c = ε: just pop b

Theorem:

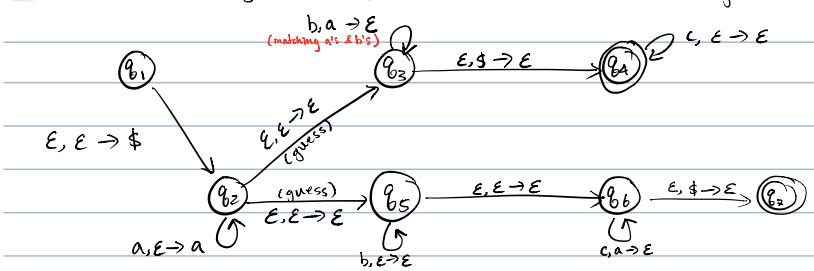
① Every CFL is recognised by some PDA
② Every language recognised by a PDA is a CFL

COR The intersection of a CFL & a regular language is a CFL

- ∵ CFL is recognised by PDA
- & regular language is recognised by a DFA w/ no stack
- ∴ you can run both in parallel

The intersection of 2 CFLs may not be context free / CFL

Ex. $L = \{a^i b^j c^k \mid i,j,k \geq 0 \text{ & } i=j \text{ OR } i=k\}$ Now design a CFL



22/10/2018

Important Recap for PDA: (more detailed in prof notes)

① Acceptance only happens @ the end of the input
⇒ have to reach the end of string & see if its at an accepts
② A PDA cannot decide to jam when there are moves possible
③ When there are choices, PDA can elect any choice it wants ⇒ includes ε

* PDAs are equivalent to CFLs (PDA ⟷ CFL)
  - ⇒ If L can be recognised by a PDA, then there is a CFG for the L
  - ⇒ If its a CFL, it can be recognised by a PDA

* DPDA ⟷ DCFL (deterministic CFLs)
  - ⇒ used to write parsing algorithms

PDA: 2 Distinct Notions of Recognition / Acceptance

① Accept by having accept states
② Accept by empty stack; @ end of string, is stack empty?
↳ There are NO accept states

DPDA Definition

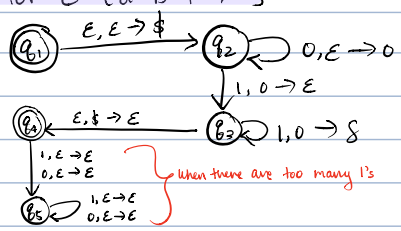
$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow (Q \times \Gamma_\varepsilon)$ OR $\emptyset$

(unlike PDAs, there can be at most ONE outcome)

☆ For every $q \in Q$, $a \in \Sigma$, $x \in \Gamma$ exactly one of

$\left.\begin{array}{l}\delta(q,a,x)\\ \delta(q,a,\varepsilon)\\ \delta(q,\varepsilon,x)\\ \delta(q,\varepsilon,\varepsilon)\end{array}\right\}$ is non-empty

DPDA for $L = \{a^n b^n \mid n \geq 0\}$



* In the PDA of L, there were no transitions coming out of q_4

Context Sensitive Languages (NOT STUDIED)

⇒ can only use rule, if within a certain context
... to then do a transition

ex. intersection of 2 CFLs ⟶ a CFL?

$\{a^n b^n c^m \mid m,n \geq 0\} \cap \{a^m b^n c^n \mid n,m \geq 0\}$

⇒ $\{a^n b^n c^n \mid n \geq 0\}$ NOT CFL

Algorithms for CFLs

Important Notes:

We say a NT X is generating if $X \xrightarrow{*} w \in \Sigma^*$ (or T^*)

(if X eventually produce some string w in Σ^*/T^*)

$L(G) \neq \emptyset$ iff S is generating

Call the generating set: GEN

ALG: (to check if L is empty)

Initialize GEN = T

Repeat until nothing changes

for each rule $X \to \alpha$ } α doesn't have to be all T symbols; can be some NT that is already generating

check if $\alpha \in GEN^*$ }

If so, add X to GEN

If $S \in GEN$ then say $L(G) \neq \emptyset$

Ex. $S \to aS$

GEN = {a}

- then look @ right side & see if everything is in GEN
  - ↳ NO, S is not in GEN, no update & terminate

EX. $S \to aXb \mid bYa$

$X \to aXb$

$Y \to bYa$

$Z \to ab$

1st Iteration: GEN = {a, b} (i.e. the NTs) ⇒ there is stn on the right for S, X & Y that's not in GEN (i.e. X & Y)

2nd Iteration: GEN = {a, b, Z} ∴ only add Z

No more changes & Terminate ⇒ is S in GEN ($S \in GEN$)? NO

∴ $L(G) = \emptyset$ (this G produces an EMPTY L)

How about $L(G) = \Sigma^*$ (complement)? ⇒ have a G & generates a L w/ every string possible

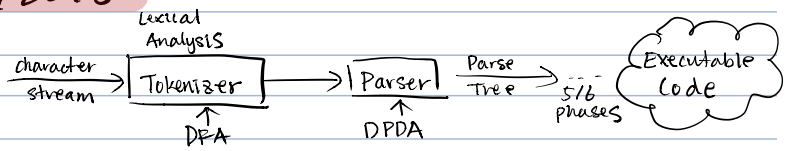
⇒ This is provably impossible ∴ The complement of a CFL may not be a CFL

The complement of a DCFL is a DCFL (∵ can recognize w/ DPDA & flip accept & reject S)

24/10/2018

Decision Procedures for CFLs : Parsing

Front-end Overview of Compiler



- when you write code, it looks like a sequence of char in a file (i.e. character stream)
- It then go into a tokenizer (recognize the function of sequences i.e. for is not a string but a function etc)
  - ↳ done w/ a DFA for lexical analysis (grouping things & add meaning)
- character stream is now more structured, it is then parsed (organized into tree structures)
  - ↳ check syntax & uses DPDA

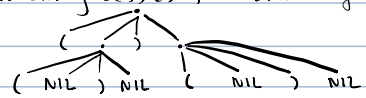
Parsing (more in prof notes)

Consider the following unambiguous grammar for parenthesis

$S \to (S)S \mid \varepsilon$

Should use a recursive descent parser to recognize grammar

For an input string '(())()', we should get



ALG Tree parseString(st) returns a Tree

{T = parseS(st)

If not eos(st), then error("Extra tokens", st)

else return(T);}

Tree parseS(st)

{ if first(st) = '(' then

{ advance(st);

Sleft = parseS(st);

If (first(st) = ')') then

{ advance(st);

Sright = parseS(st);}

else error('expecting closing paren');

→ return(makeTree('(', Sleft, ')', Sright))};

else

return(NIL);}

3 Important Grammars:

① LL(1) grammars use a one-symbol lookahead to decide which rule to use

⇒ ex. HTML & matching tages

② LR(1) grammars work bottom up & use a one symbol lookahead

③ LALR(1) is a hybrid & most popular

$w \in L(G)$? G → generic CFG, $w \in \Sigma^*$ (prof note Cocke-Kasami-Younger)

① Convert G in Chomsky normal form ($X \to AB$ $X \to a$)

② Use dynamic programming $O(n^3)$

③ Maintain a 2-indexed family of subsets of V

$X_{ij} = \{A \in V \mid A \xrightarrow{*} a_i \dots a_j\}$

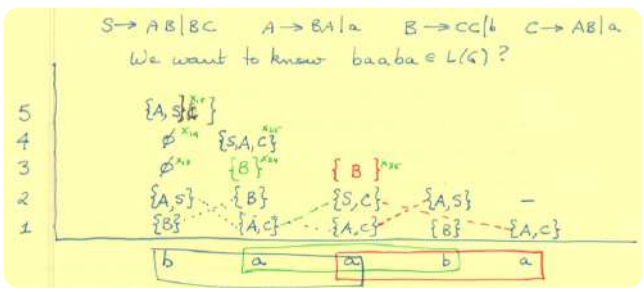
$X_{ii} = \{A \in V \mid A \to a_i\}$

Suppose $B \in X_{ik}$, $C \in X_{k+1,j}$, $A \to BC$

∴ ⇒ $A \in X_{ij}$

∵ $(A \to BC \xrightarrow{*} a_i \dots a_k \underbrace{a_{k+1} \dots a_j}_{C})$ (B: $a_i \dots a_k$)

Ultimately we check: $S \in X_{in}$



There is a decision procedure for any CFL whether w belongs to L or NOT?

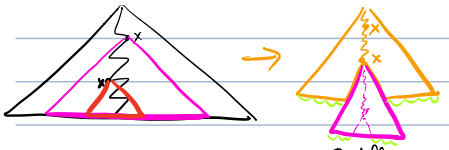
(T)

DONT PRACTICE EXECUTING just know this exists

Pumping Lemma for CFL (Important for Final Exam)

The machine (PDA) is not a finite memory machine, but we need sth finite to apply pumping lemma

* There are finite # of variables / NT in CFL



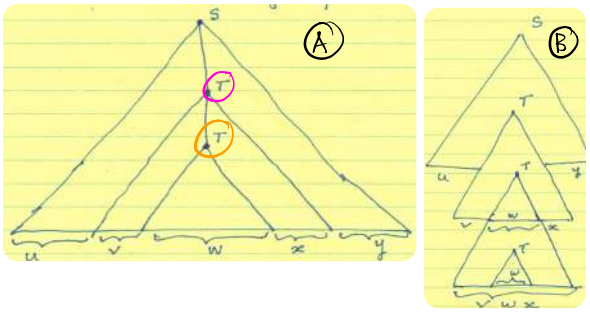
* ∴ pumping is happening in 2 diff places.
2 diff strings are getting pumped together

STEPS
- make parse tree (shortest one if ambiguous)
- somewhere along the way, I'll hit a NT twice (i.e. x)
  ↳ each NT generates a portion of the string
- we're going to cut off lower x branch
- insert the higher x tree into the pt removed
- can see repetition in 2 places

26/10/2018

Pumping Lemma for CFL (more in prof notes)

⇒ there are finite # of trees of a given height (∵ of finite # NT & rules)



(A) this whole string is generated from start symbol S
- going down tree & hit first T, see it generates 'vwx'
- 2nd T generates w

* both v & x cannot be empty, one of them can but NOT BOTH

(B) - once I do surgery, there are 2 v's & x's ; string = uv^2wx^2y
∴ $\forall i \geq 0 \quad uv^iwx^iy \in L$

$\forall$ CFLs L $\exists p \geq 0 > 0$
$\forall s \in L \quad |s| \geq p$
$\exists u,v,w,x,y \in \Sigma^*$ s.t
- $s = uvwxy$
- $|vx| \neq 0$
- $|vwx| \leq p$

$\forall i \geq 0 \quad uv^iwx^iy \in L$

for every CFL L, $\exists$ some num >0
if you take any string in the language, the length of string $\geq p$
then its possible possible to break the string into 5 pieces
↳ individual pieces may not be in string but concat of 5 is
$|vwx| \leq p \Rightarrow$ v & x can't be too far apart

Fix L some language
$\forall p > 0$
$\exists s \in L \quad |s| \geq p$
$\forall u,v,w,x,y \in \Sigma^*$
$s = uvwxy$
$|vx| > 0$
$|vwx| \leq p$
$\exists i \geq 0 \quad uv^iwx^iy \notin L$

Then L is not contex-free

ex. Show $L = \{a^n b^n a^n \mid n \geq 0\}$ is not CF
IN PROF NOTES
(more examples)

ex. $L = \{a^{i+j} b^{j+k} c^{i+k} \mid i,j,k > 0\}$ * final question

He went through all exs in prof notes

29/10/2018

$\Sigma = \{a\}$ very special

More examples Regular = CFL

Ex. $\{0^i 1^j \mid j = i^2\}$ Not CFL

$A \to p$
$I = 0^p 1^{p^2}$
$A = u,v,w,x,y. \quad uvwxy = 0^p 1^{p^2}$
$|vx| > 0 \qquad |vwx| \leq p$

cases: a) vwx → all zeros / pick $i \neq 1$
b) vwx → all ones / pick $i \neq 1$
c) v or x overlaps the boundary / pick $i = 2$
0's & 1's out of order
d) v is all 0's, x all 1's
If either $|v|=0$ or $|x|=0$, just pick $i=2$.
e) suppose $|v| \neq 0$, $|x| \neq 0$ $|v| = m$, $|x| = q$
$m > 0$, $q > 0$
pick $i=2 \Rightarrow$ New string is $0^{p+m} 1^{p^2+q}$

→ Question: is it possible that $(p+m)^2 = p^2 + q$?
$(p+m)^2 = p^2 + m^2 + 2pm$
$2pm > p \quad (\because m > 0, \therefore 2m > 1 \therefore 2pm > p)$
$\therefore 2pm + m^2 > p > q$
$\therefore p^2 + 2pm + m^2 > p^2 + q \Rightarrow$ can't be equal & out of L

Ex $\{a^p \mid p$ prime$\}$ → do directly using CFL pumping.

Ex. $L = \{a^i b^j c^k \mid i < j < k, \; i,j,k > 0\}$
$A \to p \quad I \to a^p b^{p+1} c^{p+2}$ [$uvwxy$; $|vx| > 0$; $|vwx| \leq p$]

case: a) if v or x straddle boundary, pick $i=2$, symbol out of order
b) suppose $v = a^k$ $x = b^j$ $k,j > 0$
choose $i=2 \Rightarrow a^{p+k} b^{p+1+j} c^{p+2}$
↳ condition is violated
c) $v = a^k$, $x = \varepsilon$
choose $i=1$, $|k| \geq 1$
$a^{p+k} b^{p+1} c^{p+1}$
↳ violation
d) similar cases when v consists of b's

☆ NEXT study limits of computability

d x consists of c's [pump down]

- complement of CFL is not necessarily CFL
- DCFL has complement that is also DCFL
  - ⇒ ⊗
- regular ∩ CFL = CFL

31/10/2018

An algorithm is of precise specification interms of basic steps & there should be termination proof
⇒ Turing Machine, λ-calculus, combinatory logic, primitive recursive functions
* Goal: To build one machine & read software

Turing Machines

Q : finite set of states
Σ : input alphabet ⇒ {␣} ∉ Σ
Γ : contains Σ ∪ {␣ : blank symbol}
q_a : special accept state
q_r : special reject state
} always only one of these
⇒ both q_a & q_r are stepping states (once reached, no transition/no leaving these states)
↳ machine can never halt (go back & forth)
Tape Alphabet (finite # symbols)

Is a finite state machine (FSM) + "tape"

Tape: - is semi-infinite (has starting point, no end pt)
- is divided into cells: |␣|␣|␣|␣|␣|␣|␣|␣|␣|␣|
- write a tape alpha in a cell (can't put real #'s; not finite)

Has a read head; reads one symbol / cell at a step
↳ has a transition state: $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ (move)
⇒ looks @ current state & sees current symbol
⇒ A transition can only depend on a finite amount of info (i.e. one symbol, memory etc.)
responds: ① change state
② new symbol in cell (overwrite)
③ move: L/R
Input is always a finite word!!
↳ only can see only one symbol @ a time

Halting Problem

⇒ There is no algorithm that can take a "description" & the input of the program & decide whether the program halts on the input (UNIVERSAL THEOREM)

Proof of Halting Problem

S → program (suppose S is a program)
#S → code of program S
S(x) : run program S on input x
S(x)↓ : terminates
S(x)↑ : diverges / runs forever

Halting program H takes #'S & input x & answers either halts or no halt

H(#S, x) → YES S(x)↓
H(#S, x) → NO S(x)↑

Assume H exists:
new program "P": P(x) := if H(x,x) then loop; else halt
P(#P) does it halt?
⇒ if it halts then H(#P, #P) → T
but then we end up looping ⊗
⇒ perhaps P(#P) does not halt
H(#P, #P) → else halt → F ⊗ Also CONTRA
∴ H exists cannot be true

Models of Computation

* A Turing Machine w/ extra tape is very useful (each tape w/ their own head)
↳ same power as an ordinary TM
* Adding non determinism does not increase the expressive power
* What if I had 2D tape? → same power as TM

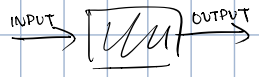
Better machines:
⇒ Random Access Machines (RAM) = TM = TM variants = λ-calculus = Java = C/C++/C# ...
= FSM + 2 stacks = 2-counter machines = while programs

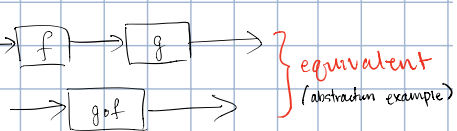
2/11/2018

TM example in prof notes (CFG two)

What is a language accepted by a TM?

Language of TM ⇒ $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ halts on } w \text{ & accepts}\}$
⇒ if a word is not in $\mathcal{L}(M)$, then the TM may reject or fail to halt
↳ can get yes but not always NO
⇒ Turing recognizable ≡ computably enumerable
if $\mathcal{L}$ can be recognised by a TM.
⇒ Turing decidable ≡ decidable ≡ computable (a set is decidable)
= is a subclass of turing recognizable
↳ the TM will always halt & give a yes/no answer
↳ all CFL are decidable
↳ always one output, can have multiple inputs





⇒ A computational device is abstracted as a function (i.e. looking only @ input(s) & ouput, not whats happenning in blackbox)

↳ however, we really need partial functions

If f is a partial function
$f(x)=y$ is one possible result
OR $f(x)$ may not be defined

$dom(f) = \{x \mid f(x) \text{ is defined}\}$ ⇒ domain of f are the possible inputs that may produce an answer
$range(f) = \{y \mid \exists x \text{ s.t. } f(x)=y\}$ ⇒ range = possible outputs
$dom(f) \subseteq X$ ⇒ total functions (special case of partial functions)
⇒ If $dom(f) = X$, then F is TOTAL

Datatypes & Coding

* Input may be ints, str or anything ⇒ DOES NOT MATTER
  ↳ ∵ any type can be encoded in other types.
* 2 integers can be coded up as 1
  ex. $(n,m) \to 2^n 3^m$ (if I have a pair of ints, I can represent it as $2^n 3^m$) (INJECTION)
  ex. $(n,m,p) \to (n,m),p) = 2^{2^n 3^m} 3^p$ (every pair is uniquely encoded)
  ∴ Any finite sequence can be coded this way!
  $\mathbb{N} \times \mathbb{N} \longleftrightarrow \mathbb{N}$ (bijection)
* Textual descriptions on transition tables can be encoded
  $\langle M \rangle$ → encoding of a TM
  ↳ not the machine but a description of the machine
  $\langle M, x \rangle$ → encoding of M with input x
  $ATM = \{\langle M,x \rangle \mid M \text{ accepts } x\}$
  ↳ This L is recognizable but not decidable
  ↳ set of all encodings of M,x pairs, such that M accepts x
  $HTM = \{\langle M,x \rangle \mid M \text{ halts on } x\}$

Technical Definition of Infinity (∞)

* X is ∞ if it is bijective with a proper subset of itself
  ex. $\mathbb{N}$ & even $\mathbb{N}$

0	1	2	3	4	...
0	2	4	6	8	...

} theres a one-to-one correspondence w/ each #
↳ this does NOT happen w/ finite sets

* A set is countable if it's bijective with $\mathbb{N}$
* Fact: there is no bijection b/w $\mathbb{N}$ & $2^{\mathbb{N}}$
  ↳ if I look @ sets of all subsets of $\mathbb{N}$, its strictly bigger than $\mathbb{N}$
  ↳ proof by diagonalization
* Fact: no bijection w/ $\mathbb{N}$ & $\mathbb{R}$
  no bijection w/ $\mathbb{R}$ & real value functions
* Fact: a countable union of countable sets is still countable
  ↳ have infinite many sets, # of sets is countable, each set is ∞, & the union of these sets are still countable

The space of functions are NOT countable

* Fact: There are countably many TM's ⇒ algorithms ⇒ computable functions

5/11/2018

Recap: ↙ = partial function ⇒ so might not even halt in finite time

A function is computable if there's an algorithm to compute it

A set is decidable if there is an alg A which always halts & answer Y/N to the membership query

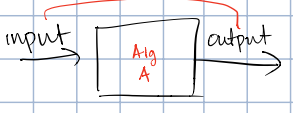
↳ an algorithm that can answer whether an element belongs to that set

↳ has to be an always terminating alg that gives Y/N

↳ # of alg is countable; each alg is written w/ finite string & # of diff finite string is countable

⇒ decidable ≡ computable, language ≡ set, computable function, decidable set

↳ = Turing Decidable (i.e. there ∃ a TM that answers the ques) ≡ generic algorithms (ex. Java)



↳ if we don't look inside, we have a purely functional view (f)

↳ if we look inside, we see algorithm A
(Algorithm A computes the function f)
(Function f is defined by algorithm A)

Turing Recognizable ≡ Computably Enumerable (CE) (with an input)

↳ means a TM will list all the members of a CE set in no particular order. If an element is in such a set, you'll see it eventually. If its not in the set, you might wait forever

↳ a Semi-decidable problem: can prove yes but not no

↳ The halting problem is a CE problem. Given a description of TM & input & ask does this TM halt w/ this input? It's CE or semi-decidable

⇒ decision procedure: run simulator & hope it eventually stops

Theorem (More detailed in proof notes)

An infinite set $X \subseteq \mathbb{N}$ is decidable (i.e. $\exists$ alg that answers Y/N) iff $\exists$ a total (i.e. always halts) computable function whose range is X.

(total computable function)

non-decreasing

Proof ⇒ Look @ output

Suppose X is the range of a TCF f with algorithm A (A always halts ∵ total)
there is a decision procedure for membership in X: (A is computing function f)
Given n, run A on $0, 1, 2, 3, \ldots$ look @ outputs from A
If one of these outputs is n, we say "YES" & $n \in X$
If we get output m, where $m > n$, then we know $n \notin X$ (∵ f is nondecreasing)

all outputs are non-decreasing (increasing or same #)

Output with element on the list

Suppose X is decidable, so B is an algorithm to decide it. B must always halt
What does B decides X mean?
⇒ give a # to B and ask if this # is in X, B will stop & give you answer (Y/N)
Now define f as follows:
Run B on $0, 1, 2, \ldots$ every time we get a # in X, add to list.
When we have n elements on list, we output it. Defined to be $f(n)$ (nth element is output)
↳ ∴ range of $f = X$

Suppose $X \subseteq \mathbb{N}$, we define an indicator function / characteristic function

$$1_X(n) = \begin{cases} 1 & \text{if } n \in X \\ 0 & \text{if } n \notin X \end{cases}$$

∴ A set is computable if it's indicator is computable

Semi-characteristic Function of X

$$\mathcal{G}_X(n) = \begin{cases} 1 & \text{if } n \in X \\ \text{undefined} & \text{if } n \notin X \end{cases}$$

Theorem: A set $X \subseteq \mathbb{N}$ is CE (iff) any one of the following equivalent conditions hold.

i) X is the domain of a computable f^n (not necessarily TOTAL) (There some computable function whose domain is X)
ii) $\mathcal{G}_X$ is computable (there a specific computable function, whose domain is X)
iii) X is the range of a computable f^n (print output rather than input)

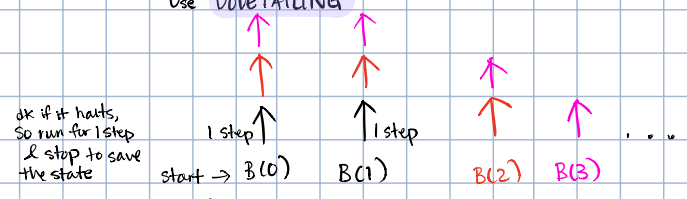
Proof: Clearly (ii) ⇒ (i)

Claim: CE ⇒ (ii) & CE ⇒ (i)

CE means I have an alg (not halting) that outputs members of X, not need increasing order (random order)
Given an enumerator, want to know if $\mathcal{G}_X(n) = 1$ (is n in set X)
↳ just wait for the enumerator.

FINAL (i) ⇒ CE Suppose we have an alg B to compute f & $\text{dom}(f) = X$
↳ We can't just run B on $0, 1, 2 \ldots$ b/c B may not halt on some of its inputs

Use DOVETAILING



Any given one of these may run forever but if any $B(n)$ halts, we will eventually find out
We advance computation on increasingly large # of inputs.

If B halts on any input, we print value & continue

Theorem The union & intersection of 2 CE sets is CE (HW QUES)

Union: To see this, run the enumerators for X, Y in parallel. If its in either X or Y, output value.
↳ for finite unions
* Infinite CE sets

Post Theorem:

1) If x is computable, it is CE
2) if x is CE & $\bar{x}$ is also CE, then x is decidable
↳ proof: run enumerats of x & $\bar{x}$ in parallel & will definetly get Y/N answer

☆ Remember pairs of numbers can be coded as a single #

$\mathbb{N} \times \mathbb{N} \leftrightarrow \mathbb{N}$ (This map is total computable)
$\langle n, m \rangle \in \mathbb{N}$ encode 2 # n, m into one # $\langle n, m \rangle$

Post Theorem 2: A set $X \subseteq \mathbb{N}$ is CE (iff) $\exists$ a computable set $Y \subseteq \mathbb{N} \times \mathbb{N}$ s.t.
$\forall x \in X \quad x \in X \Leftrightarrow \exists y \in \mathbb{N}$ such that $(x, y) \in Y$

Proof: If we have such a Y, we can see that X is CE (since Y is decidable)
Suppose X is CE, now do we construct Y? Y has to be total computable
Take $Y = \{\langle x,n\rangle \mid A$ enumerates x in n steps$\rangle$ ← Suppose A is the enumerator for X
We can count how long A runs. ∴ A is an algorithm, when run = outputs members of X
↳ Do you produce X in n steps?
Yes, then code up $\langle x,n\rangle$ & put in Y ⇒ Decidable! "Can it halt in n steps?"
∴ $\forall x \quad x \in X \Leftrightarrow \exists n \in \mathbb{N} \quad (x,n) \in Y$
↳ decidable + quantifier ⇒ CE

11/8/2018

Reduction:
- reduce hard problems to the halting problem
  ↳ if I solve this problem, I've secretly solved the halting problem

P reduces to Q : if solve Q, can solve P : Q is harder than P
$(P \leq Q)$

ex. want to prove that Q is undecidable, so take a problem K is undecidable (ex. halting problem)
① Take an instance of halting problem
② Modify to create an instance of Q
③ Assume Alg for Q exists, & can solve Q
④ ∴ able to solve halting problem.
But know halting p is impossible ⇒ alg of Q is impossible & Q DNE
* always start by assuming that Q exists

EIM, set of all TM descriptions such that it rejects every word
* write a TM program
① TM program ignores input
② Simulate M on w
③ If it halts, accept input

∴ $L(M') = \emptyset$ if $M(w) \in \uparrow$ (if M doesn't halt, you don't accept anything)
$L(M') \neq \emptyset \ (= \Sigma^*)$ if $M(w)\downarrow$ (if M halts on w, you accept everything)

CE → HP, AP, not in class of computable probs, EMPTY

coCE → able to give no. If y answer, may never halt
(complementary) ↳ EMPTY (run any input / # on empty ... it'll return no)
↳ dovetailing
— $\overline{HP}$, $\overline{AP}$

If prob is both CE & co-CE = decidable

$88.6 psych.
82 A-
$$ -791.8
3.7
3.0
0.11764706

Halting problem recap: Total = an output for every possible # in set

Assume $H(\#S, x)$ is the program for the halting program

Assume H exists

Now we define program $P(x) :=$ if $H(x,x)$ then loop; else halt

Assume P is a valid program, then according to the halting prob, P should halt

↳ $P(\#P)$ halts then $H(\#P, \#P)$ is true & will loop (contradiction)

Now assume P is an invalid program, so it shouldn't halt

↳ $P(\#P)$ does not halt, then $H(\#P, \#P)$ goes to else halt (contradiction)

∴ H cannot exist.

Language accepted by a TM: $L(M) = \{ w \in \Sigma^* \mid M \text{ halts on } w \text{ \& accepts} \}$

Turing recognizable / computably enumerable: if a language of the TM accepts for if a certain string is within that language, but not necessarily reject if that string is not in L.

Turing decidable / decidable: a subset of enumerable, where TM can give a yes & no answer of whether a string is in the L recognized by the TM.

∴ an enumerable L is not necessarily decidable

a decidable set is enumerable

dom(f) = the # of inputs that produce an output $\subseteq X$ (X is all possible inputs) ∴ not all inputs produce an output

range(f) = possible outputs

f is Total if dom(f) = X [all possible inputs produce an output]

ex. enumerable language that is not decidable

$A_{TM} = \{ \langle M, x \rangle \mid M \text{ accepts } x \}$

$H_{TM} = \{ \langle M, x \rangle \mid M \text{ halts on } x \}$

↳ H_{TM} is enumerable but not decidable

An ∞ set $x \subseteq \mathbb{N}$ is decidable if $\exists$ total, non-decreasing function f where range(f) = X

A set x is CE if one hold:

① $\exists$ a computable function w/ domain(f) = X

② $\forall x$ is computable

③ $\exists$ a computable function f w/ range(f) = X

① ⇒ ② ⇒ ③ ⇒ ① ...

proof ③ ⇒ ①

We want to find an output of f that is in X. But since the functions is not specified as non decreasing & computable only in the sense of whether for an input ∃ an output for f... it make take forever to actually find an output that's in X

proof ② ⇒ ③ : theorem of decidable.

proof ① ⇒ CE:

- have alg A to compute f w/ dom(f) = X
- run A on $\mathbb{N}$ (i.e. 0, 1, 2...) but it may take forever for B to find a # that is in X, ∴ may never halt

∴ use dovetailing on countable infinite set $\mathbb{N}$

The ① union & ② intersection of 2 CE sets is CE.

Assume 2 CE sets X & Y

Alg A computes computable function f_1, where $dom(f_1) = X$ & $range(f_1) = X$

Alg B computes computable f_2, where $dom(f_2) = y$ & $range(f_2) = y$

Run in parallel { Run A on 0, 1, 2... (i.e. $\mathbb{N}$), but make take forever to find # of X
Run B.

use dovetailing ⇒ CE

$X \cap Y$

$dom(x) \cap dom(y)$

Post Theorem:

① if x is computable, it's $CE\bar{}$

② if x is CE & $\bar{x}$ & $CE\bar{}$, then x is decidable

③ $\forall_x \; x \in X \Leftrightarrow \exists_n \in \mathbb{N} \; (x_n) \in Y$

x is $CE\bar{}$; Y is computable.

$P \leq Q \Rightarrow$ p reduces to $Q \Rightarrow Q$ is harder than P

claim: I have a alg to solve Q

Solution: make P be of sth unsolvable (ex. halting)

① Take an instance of P

② modify to create instance of Q

③ then wl supposed solution of Q can solve problem

BUT we know P doesn't exist!

Nov 12

$L = \{wwl w \in \Sigma^*\}$

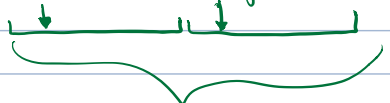
$\bar{L}$ is a CFL & have grammar for $\bar{L}$

How to recognize $\bar{L}$ w/ a PDA? (Yes, by one that can make guesses)

Sketch of how the PDA works

If the input has odd length, it's definitely in $\bar{L}$

Consider an even length word. To be not in L:



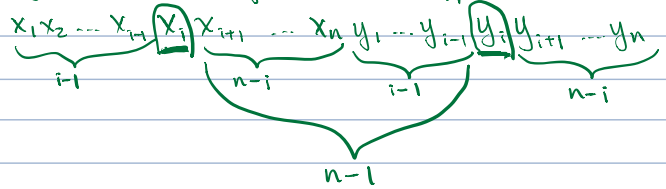
At the same corresponding position (↓), they must disagree.

split the word into equal length

oo guess a place where the mismatch occurs

guess the middle.

$x_i \neq y_i$ is where disagreement happens



stack x_1 to x_{i-1}, remembers x_i

pop the stack as it reads $x_{i+1} \cdots x_n y_1 \cdots y_{i-1}$

length $n-1 > i-1$

As soon as stack is empty, switch to stacking the letters again

then @ y_i it compares the letter w/ x_i.

Now it reads the rest & pop the stack.

If the stack is empty @ the end, we know all the guesses have been reindicated.

To see if w is in L'(complement of L) when the word is even. It is trivial the word is in L' when word is of odd length.

The word should be able to split into 2 halves of equal length.

We chose a letter within the first half xi. We get the same letter positioned at i in the second half: yi

If this word w is in L1, then xi != yi

We do this put reading the word and putting the letters in a stack until we reach xi and remember its location. Once we reached xi (I.e. the number we want to compare with the other half of the word), we start popping letters of the stack for each letter we read. The stack will obviously eventually get empty. Once its empty, we continue to stack letters until we reach yi. We now compare xi and yi and see if they're mismatched. If they are, we now have to confirm that they are at the right locations (I.e. i). So we continue reading the word and pop. The stack should be empty at the end.

Computability Theory Review

All functions in computability theory are partial functions

Computable: a function is computable if there is an algorithm to compute it

PARTIAL FUNCTIONS

We chose to have **N** natural numbers as our data type (it can be anything actually)

- Its just prettier and in increasing order if we use N
- So a partial function f takes a natural number as an argument and MAY produce an outcome that is natural
  - If f produces an outcome m with argument n, then function f is **defined**
    - **f(n) = m**
- **Domain** of function: set of all values where partial function f is **defined**
- **Range** of function: set of values that f can return
- g extends f if:
  - **Dom(g) = dom(f), for all arguments in dom(f): f(n) = g(n)**
  - g can be defined for arguments where f is not!
- *An empty function is a **computable** function because it is nowhere defined*

COMPUTABLE AND COMPUTABLY ENUMERABLE SETS

Suppose set X is a subset of N, and X has a total characteristic function

- 1x(n): 1 if n is in x, 0 if n is not in x

A set is **decidable/computable** if it's total characteristic function is computable

- Computable sets have a **decision procedure**: a way which always terminates with a definite answer
  - Ex. Prime numbers is computable (have alg that takes input and decide whether is prime or not)
    - Since Y/N answer = always terminate
- <M, x> means the encoding of information of a TM and an input string
  - Sets where a M accepts x or M halts on x are NOT COMPUTABLE
- *An INFINITE set X, of natural numbers is computable if and only if it is the range of some **total non-decreasing** computable function f.* **Expected to halt**
  - Proof:
    - Assume X is infinite, so range of f is infinite
    - Suppose we have X and A an algorithm that computes it
    - So decision procedure for X:
      - Is a given x in X?
      - Run A successively on 0, 1, 2,... and look at outputs
      - If the output is x, we stop and sat yes x is in X
      - If output <x, we continue to next input
      - If output >x, we stop and x is not in X

      - Now suppose X is computable and B algorithm decides whether x is in X
      - Define f: given n as the input for f, run B successively on 0, 1, 2...
        - Every time we find a number in X, we store it in a list and count how many numbers we have seen
        - When we hit the nth number, return this output
        - Thus, f is computable and non-decreasing and has **range = X**

A set X is **computably enumerable (CE)** if there is an algorithm A that lists all the members of X, and only the numbers of X, in some arbitrary order . (*Not like computable sets where they're ordered in non-decreasing order*)

- **Not expected to halt**, halts only if its a FINITE set (bc it has to list all the elements in X)
- A set is CE if there is an algorithm for:
  - If an element is in set, algorithm is guaranteed to terminate and **say element is in set**
  - If element is not in set, it may terminate or loop forever
- If we wait long enough, we will get a YES for every element in the set
  - But elements will output in **arbitrary order**, so unsure of waiting time and presence

Set X is CE IIF any of the conditions hold

1. X is the domain of a computable function
2. The **semi-characteristic** function of X is computable (Sx(n) = {1| if n is in X; undefined| if n is not in X}
3. X is the range of computable function

(2) immediately implies (1)

CE implies (2) Proof:
- Suppose A is an enumerating algorithm A for X
- To compute Sx(n), a semi-characteristic function:
    - Run A and inspect list as it comes out, checking for n
    - If n appear on list, output 1
    - If n does not appear, we may run forever

(1)[and so (2)] implies CE Proof:
- Suppose B algorithm compute f and dom(f) = X
- We CANNOT run B on 0, 1, 2... and output any numbers for which it terminates
    - So use **dovetailing**: for every phase we run the computation longer and include more arguments in our simulation
    - If B terminates on an input, we print it and continue
    - If n is in dom(f), B will run on long enough on n and n will get enumerated
- X is CE

(3) implies X is CE Proof:
- If we run dovetailing to print outputs that B terminates (instead of inputs)

CE implies (3) Proof: (??)
- Know Sx(n) is computable by some algorithm C
- Define a partial function g:
    - G(n) = {n | if C terminates on n; undefined otherwise}
- Clearly g is computable and range is X

Theorem: The union and intersection of 2 (hence any finite number of) CE sets is CE
Proof: suppose X and Y are 2 CE sets with enumerating algorithms A and B respectively. Run A and B in parallel, and if an item appears on the list, enumerate it.

Post's Theorem:
1. If X is computable, it is also CE
2. If X is CE and its complement X' is also CE, then X is computable

Proof (1):

Proof (2):
- Assume A enumerates X & B enumeratees X'
- To show that X is computable, we need a decision procedure
    - Run A and B both in parallel and soon, one of them will enumerate n
        - We know whether n is in X or not
    - This will have to terminate bc n is in X or not (I.e. X') and will terminate in finite time

Projection functions: Pi1(<n,m>) = n and Pi2(<n,m>) = m

Theorem: set X is CE IIF there is a computable set Y of pairs of natural numbers such that

$$\forall x, x \in X \Leftrightarrow \exists y \in \mathbf{N}(\langle x,\ y\rangle \in Y).$$

Proof: if Y is computable or CE, we can enumerate its element and use pi1 to extract the first component
- If such Y exists, then X is CE
- If X is CE, take enumerating algorithm A, and construct Y as: **Y = {<x,n>|A enumerates x within n steps}**
- Its is obvious that Y is decidable bc we put a time blunt on the algorithm run time
- If x is in X, then a enumerates x after some number of steps so the theorem above is true.

Notes of Reductions

Overview:

- What does "P reduces to Q" mean?
- 2 notions of reduction: (1) many-one reduction and (2) Turing reduction

"P reduces to Q" = if you can solve problem Q, then you can solve problem P = **P<= Q**

- Suggests that Q is more difficult than P, so if u solve Q, you can solve P
  - So if P is undecidable, then we know Q is undecidable
- **Relations are <u>transitive</u>**

HP <= Q, show Q is undecidable

- **HP = halting problem**
- Knowing that Q is undecidable we can write another reduction: Q <= R, so R is also undecidable

Working with fixed alphabet Σ and considering strings in Σ^*

- We can now also reduce languages
  - A reduced to B
    - Is the word in B, then word is also in A
- **Definition**: A language A is mapping reducible to B, written $A \leq_m B$, if there is a total computable function $f : \Sigma* \longrightarrow \Sigma *$such that x is in A IIF f(x) is in B. Function f is called the **reduction**
  - Works only to guarantee membership in A only if there is membership in B
  - Membership in A does NOT guarantee membership in B
  - If we wish to find "is w in B?"
    - Need to find an x such that f(x) = w; not guaranteed to find an x

Theorem: if A<=m B and B is decidable, then A is also decidable. If A is undecidable, then B is undecidable

Theorem: if A <=m B and B is CE, then A is CE. If A is not CE, then B is not CE.

Proof: Suppose B is CE, then there is an algorithm isinB where input x will halt and say TRUE if x is in B.

- May loop forever if x is not in B
- Algorithm isinA:
  - Read input x
  - Apply f to x to produce y
  - Run isinB on y
  - Output whatever the previous line produces
- Line 2 must terminate because f is total
- If x is in A, then reduction is correct, then f(x) = y is in B.
- isinB is guaranteed to terminate, then we know that isinA terminates and say true if x is in A
- If x is not in A, then f(x) will not be in B and line 3 will run forever.
- A is CE, if A is not CE, then B cannot be CE (controposing)

A set is **co-CE** if its complement is CE

- So complement of HP is a co-CE set : $\overline{H}_{TM} \stackrel{def}{=} \{\langle M, x\rangle \mid M(x)\uparrow\}$
- Use mapping reduces to show that sets are not CE by showing reductions from this set.
- **Observation**: If $A \leq_m B$ then $A' \leq_m B'$
- **Corollary**: if $A \leq_m B$ and A is CE but not decidable, then B is not co-CE
- Proof: if B is co-CE, then B'is also CE, and the mapping reduction will be written as:
  - $A' \leq_m B'$, so A' is also CE
  - If A is CE and A' is also CE, then A is decidable.
    - Contradicts assumption that A is undecidable

Example:

Language of TM that accepts nothing: $EMP_{TM} \stackrel{\text{def}}{=} \{\langle M\rangle | L(M) = \emptyset\}.$

This is UNDECIDABLE

Proof: by showing that ATM <=m EMPTM'

- Define reduction function that is **total and computable:** $f : \Sigma* \longrightarrow \Sigma *$
- try to solve: given encoding <M,w>, accept IIF M accepts w
  - Reduction function will work with ay string, not just strings that are encodings of acceptance problems
  - We will map any string that is not of the right form to the string fubar
  - The string is certainly not part of EMPTM
- We will define a new TM M'
  - Input x
  - Simulate M on w

    - If M accepts w accept x
- If M accepts w, M' will accept every input and if M does not accept w, then M' will not accept anything
-
- WE DO NOT RUN M'
    - We write code of M' and feed <M'> to the decision algorithm for EMPTM'

Halting problem: Im defining a language that consists of words or numbers. These words are encoding a description of machine and description of an input. This word is in the language, if that machine halts on that input.
HTM is the halting language: HTM = {<M,w> | M halts on w }

- Language of all TM descriptions and input descriptions such that that machine halts on that word
- Not important whether they accept or reject; care about HALT
- It might halt & accept or halt & reject. All we know is that it halts

ATM is a set of words that encode the description of machine and input where the machine ACCEPTS and halts on that input.

Ex. Reduction of HTM < ATM

- Assuming that we have solution to ATM, then there should be an algorithm for HTM
- We assume that HTM exists
- will simulate an instance of the halting problem.
- Input is <M,w>
- Algorithm will be slightly modified.
    - We will create a new machine M': on input x, it simulates M with input x. If M halts on X, then M' accepts X.
- Output <M', x> and input into the solution u supposedly had for ATM
    - So does M' accept w: if yes, then M halted on w
    - We have answer to halting problem.
- BUT we know HTM doesn't work, so ATM doesn't work.
- So since HTM is undecidable, then ATM is also undecidable

HTM < ETM
ETM: only has description of a TM and no input. Does this accept anything at all

- So the language of ETM is empty; such that it rejects everything

Proof:
Assume we have an algorithm for the HTM

- We feed in input <M,w> to the algorithm for HTM
- We define a new TM: M'
    - Ignore input
    - Simulate M on w.
    - If simulates halts, then M' accepts that input.
- Output is <M'> and feed into alg for ETM.
    - Alg for ETM
        - The language of M' TM is empty, if M doesn't halt on w
            - If M halts, then M' will accept that input
        - If the language of TM M' is not empty, then M halted on w.

So HTM is undecidable, so ETM is also undecidable
Try ATM < ETM

CE = HTM and ATM and EMPTY complement = are not decidable as they can only give yes answer
Co-CE = ETM, complement of HTM and ATM = are not decidable as they can only give NO answer

Is this language empty, if you run dovetailing and halts on something. Find its not empty and say NO. **No to what????** Is the language empty? Its easy to answer no, if we just run into one that accepts. But to see if its empty, the yes answer may take forever to run through all of the inputs.

If algorithms are both CE and co-CE (I.e. have both a yes and no answer), they are DECIDABLE

Mapping reduction
Suppose we have 2 languages and L1 *mapping reduces* to L2
If there is a total computable function (always halt) f.
If there is any word w in Sigma*, whether w is in L1 is equivalent to asking if f(w) is in L2
f is called the mapping reduction.
So if X is not in L1, then f(X) is not in L2
Implies complement of L1 and L2

Rice's theorem
TM = Program = algorithm etc.
The set of all programs is the same as the set of all TM's
Extensional property of a program: a property of a function it computes
- Describing input and output behaviour of function
- So all sorting algorithms are all the same

Two programs are extensionally equal if they implement the SAME FUNCTION
P1 ~ P2 if they compute the same function - PROGRAMS
M1 ~ M2 if L(M1) = L(M2) : TM (do they recognize the same language)
- Language is a extensional property

An extensional property Q is a family of CE sets or a predicate on TM's or on programs s.t.
If Q(M1) & M1~M2, *then Q(M2)*
If M1 satisfies the property Q and M1 and M2 are extensionally equal, then M2 also satisfies the property Q
You can have TRIVIAL properties:
Q: false = property that nothing satisfies
Q: true = everything satisfies
Trivial properties are decidable
RICE: Every non trivial extensional property of TM's is undecidable.
Proof:
Let Q be a non-trivial property of TM's or CE sets or programs. Assume Q is extensional
Assume L(M) = 0, then Q(M) does not hold.
Since Q is non trivial, there is some M0 such that Q(M0) holds and hence L(M0) != 0
ATM <m LQ = { <M> | Q(M) holds }

A language with FOR loops only can can express terminating algorithms
A typed high-order **functional program** language without recursion can only express terminating programs.
Any such language will also fail to express a properly terminating program
Suppose we have such a program, and we enumerate all programs in HALT
List of halting programs: P1, P2,...,Pn
We define a new program P(n) = Pn(n) +1
On input n, its runs program Pn on input n and add 1
So P cannot be on the list! And P is always halting.
So you say give me a language that only give you terminating programs but also gives me all terminating programs...this is not possible.
You can only have (1) I want all the halting programs, but some programs may not halt
(2) ??put more restrictions so everything terminates. But you cant write shit???

Valid Computations of a TM

- Allows you to reduce some reductions of a halting problem and show undecidability result with CFLs
- Many CFLs are undecidable

Given <M,w>. Is this accepted?
Given M, w, we can effectively construct a CFL or equivalent PDA such that it generates or recognizes the complement of the set of valid computations.
We can do this without knowing if M accepts w.
What is a valid computation?

- Important components of TM: state, tape and head position(which cell you are looking at in the tape)
- A configuration of a TM is a description of the state, the tape, and the head position.
- Suppose table contains **abbaab**. Suppose we are looking at the 2nd b and is in state q
- So just before the where the head is looking at, I write down the name of the state: **abqbaab**
  - The q tells u that its looking at the cell to its right
  - Abqbaab is the encoding of looking at the 2nd in state q
- How do we show moves?
  - We put a #; not part of table alphabet
  - # separates configurations
  - $\delta(q,b) = (q', a, R)$ ⇐ transition rule
  - Machine is in state q and looking at c with b inside
  - Move to state q', replace b with a and move to right

abbaab state q

encoding / configuration: abqbaab

abqbaab# aba q'aab# ----- shows transition

A valid computation is a description of all the steps form the start until the machine accepts.
Accept = accept in finite many steps.
How do we know the machine accepts the word? IDK, there might be no valid computations.

- we can use pda if something is NOT a valid computation

Define computation:
Start config: #q0a1...an# [start state and some symbols until #]

A valid comp: $\#\alpha_0\#\alpha_1\#\dots\#\alpha_N\#$
① α_0 must be START config
② α_N must be an accept config
③ α_{n+1} must follow from α according to the transition rule of M

☆ For a non deterministic M, there may be many valid comps.

VALCOMPS(M,w) ← not a CFL
If M does not halt/accept on w, then VALCOMPS(M,w) = ∅

want to show $\overline{\text{VALCOMPS}(M,w)}$ is a CFL.
∴ $\overline{\text{VALCOMPS}(M,w)} = \Sigma^*$
∴ Given G, $L(G) = \Sigma^*$ must be undecidable

$L = \{ww \mid w \in \Sigma^*\}$ not a CFL.
$\bar{L}$ is CFL
If some sequence $z \in$ VALCOMPS(M,w). Then 5 things must hold.
① z must begin & end w/ #, #
& btw every pair, we must have a non-empty string
ex. $z = \#\alpha_0\#\alpha_1\#\alpha_2\#\dots\#\alpha_N\#$
② Each α_i must contain exactly one letter from Q.
Q is a description of the states. Position of q lets you know where head is.
③ α_0 must be start config
④ α_N must be accept config.

⑤ for each i, the diff btw α_i & α_{i+n} must be according to rules of M

①-④ can be checked w/ DFA
⑤ check w/ PDA

* The diff btw α_i & α_{i+n} should be confined to 3 consecutive symbols.
⇒ the possible values of 3-symbol sequence can be remembered in finite memory

ex. $\delta(q,a) = (p,b,2)$

config: abaq abba

① $\delta(q,a)$: currently q is looking @
② first change a to b
abaq bbba
③ move head left

abaqabba ⟶ abpabbba ⇒ 2 pairs of 3 symbol sequences are consistent if they follow the rules of M
* parts changed.

Its PDA guesses a pair (ai, ai+1) where the rules of M are violated and it guesses a 3 symbol sequence where the violation occurs.

... # w_1 xxx w_2 # w_1' YYY w_2' #

Stacks w1, memorizes xxx, ignores w2, after#, starts popping stack with w1' and compare YYY
If find that YYY and XXX are wrong, then not in VALCOMPS

VALCOMPS 2 need 2 PDA to check $\exists \in$ VALCOMPS2(M,w)
VALCOMS2(M,w) i

VALCOMPS2(M,w) = $L(G_1) \cap L(G_2)$
VALCOMPS 2 ≠ 0 then M accepts w.
① $L(G_1) = \Sigma^*$ undecidable
¬ATM $\leq_m$ ($L(G_1) = \Sigma^*$)
ATM ≤ ($L(G_1) \cap L(G_2) = \emptyset$)
② $L(G_1) \cap L(G_2) = \emptyset$ undecidable.

Post's Correspondence Problem

- We have a finite set of pairs of strings.

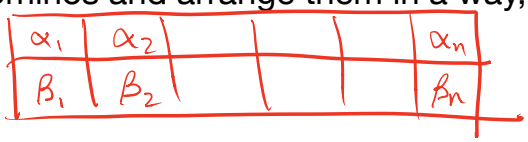
  $S = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \dots (\alpha_n, \beta_n)\}$

- A solution is a finite sequence of integers/natural numbers s.t.

  $i_1, i_2, \dots i_k$ s.t.
  $\alpha_{i_1}, \alpha_{i_2}, \alpha_{i_3} \dots \alpha_{i_k} = \beta_{i_1}, \beta_{i_2} \dots \beta_{i_k}$ } imagine these strings as dominos

I can have as many of each domino
ex. have 4 [$\alpha_1\beta_1$] domino & 3 [$\alpha_n\beta_n$] dominos

Can I chose dominos and arrange them in a way, such that the top string = bottom string?



$\therefore \alpha_1\alpha_2 \dots \alpha_n = \beta_1\beta_2 \dots \beta_n$

Example: 1 2 3
S = { (b, bbb), (babbb, ba), (ba, a) }, is this solvable? YES
Solution 2 1 1 3

babbb	b	b	ba	= babbbbbba
ba	bbb	bbb	a	= babbbbbba

Example: Impossible, no solution
S = { (ab, abb), (b,ba) , (b,bb) }
There is no algorithm to solve this problem!

The problem of finding a match is equivalent whether a TM accepts a word.

- We have seen valid computations of a TM
- I'm going to design my dominos, such that if I have a match, the string is a valid computation of a TM on a word; Irrelevant whether the TM halts.

Modified PCP:
There is a special domino with which you must start.
An accepting computation looks like #q0w1...w1n#...#x1...xkqnx(k+1)...#

- Sequence of words separate by #
- Each b/w #,# is a description of the configuration of a TM (state, head, content)
- Follow the 5 rules of VALCOMP and then its valid

Design dominos
Top: #
Bottom: #q0w1....wn#
If we have rule $\delta(q,a) = (r, b, R)$, then
put $\begin{bmatrix} qa \\ br \end{bmatrix}$ into the set (unless $r = q_r$)

Example: $w = 0100$, q_0
$\delta(q_0, 0) = (q_7, 2, R)$
Start domino: $\begin{bmatrix} \# & q_0 0 & 100 & \# \\ \# & q_0 0 & 100 & \# \end{bmatrix}$ $\begin{bmatrix} 2 q_7 & 100 \# \end{bmatrix}$

$\begin{bmatrix} q_0 0 \\ 2 q_7 \end{bmatrix}$

if we have rule: $\delta(q,a) = (r,b,L)$

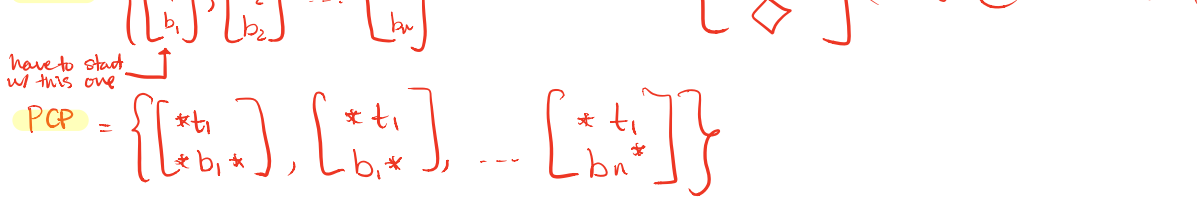
$\begin{bmatrix} c q a \\ r c b \end{bmatrix}$ for ever $c \in \Gamma$

$\forall a \in \Gamma$ $\begin{bmatrix} q \\ a \end{bmatrix}$, $\begin{bmatrix} \# \\ \# \end{bmatrix}$ $\begin{bmatrix} \# \\ \sqcup \# \end{bmatrix}$

Given a munch of dominos, how do I ensure that I start with $\begin{bmatrix} t_1 \\ b_1 \end{bmatrix}$?
Use an extra symbol * which does not appear in any string.

$\vec{u} = u_1 u_2 \dots u_n$
$*\vec{u} = *u_1 * u_2 \dots * u_n$ (stick a * to the left of every u)
$\vec{u}* = u_1 * u_2 * \dots u_n *$
$*\vec{u}* = * u_1 * u_2 * \dots * u_n *$
MPCP = $\{ [t_1], [t_2], \dots [t_n] \}$

$\begin{bmatrix} * & \diamond \end{bmatrix}$ ⇐ use @ the end



This PCP is equivalent to MPCP

$\therefore$ $ATM \leq_m PCP$

↳ undecidable & CE

If there is a match, you will eventually find it.

ATM <m PCP <M AMBIG

AMBIG: give a grammar and ask if there is a alg to find whether this is ambiguous?

HTM <m ATM (BOTH CE)
ATM <m HTM (BOTH CE)

Are all ce problems reducible to each other? NO
All ce problems can be solved if you can solve ATM or HTM

Syntax of a first-order language

- Syntax: proper notations and formulas
- We have a set of constants: a, b, c,...
- We have a set of variables: x, y, z, ...
- Both these sets of countable
- A set of function symbols: f, g, h,...
- A set of predicate symbols: P, Q, R...
- Functions and predicts come with arities (type information: each function takes a fixed number of args etc.)

- TERMS: defined by induction

1. Any constant is a term
2. Any variable is a term
3. if f is a function symbol of arity n, and t1,...,tn are terms, then f(t1,...,tn) is a term

Example first order language: Arithmetic

- Constants: 0, 1, 2, 3...
- Variables: x, y, z,...
- Function symbols: +, *, succ (number after)
  - + has arity 2, * has arty 2, and succ has arity 1
- Predicate symbols: =, <
- Example terms we can build:
  - +(3,0) and *(+(3,0),x) and succ(7)
  - (3+0) (3+0)*X
- Terms describe what we are talking about (nouns)
- Formulas are what we say about terms:
- If P is a predicate of arity n and t1,...,tn are terms. Then P(t1,...tn) is a formula.
- If phi1, phi2 are formulas

- If phi is a formula, and xi is a variable then

Example: Plus(0,3) < 2 is a formula

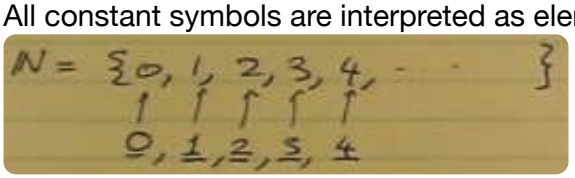
- 4 = 5 is a formula
- 3-3 = 0 is a formula

SEMANTICS

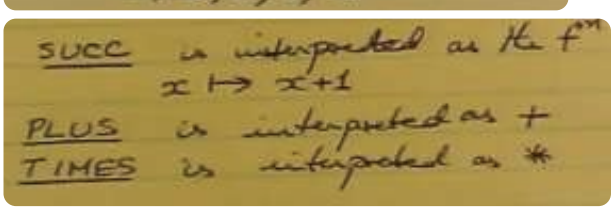
Interpretation:

D = a set called the domain of interpretation

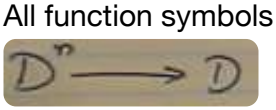
All constant symbols are interpreted as elements of D



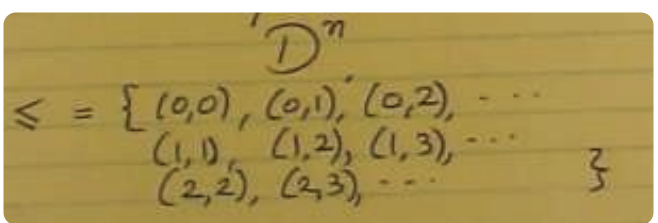
The zero symbol means value zero etc.



All function symbols of arity n are interpreted as functions:

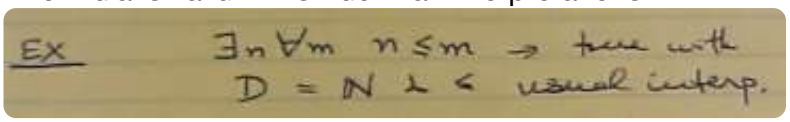


Expecting function to take in n arguments and producing something new



All predicate symbols of rarity n are interpreted as subsets of D^n.

A formula is valid if it is true in all interpretations



NOT true is D = Z (integers)
There is no smallest integer
NOT VALID

PROOF
We can define axioms and rule of interface and proofs
A formula which can be proved is called a theorem
Soundness: any theorem is valid
Different proof systems have different power.
Proof systems may not be able to show ALL true things.

- If we have a proof system s.t. every valid formula is a theorem .
- Then the system is called **complete**. (If it shows all true things)

FOL has COMPLETE
ARITHMETIC has NO COMPLETE PROOF SYSTEM.
a formula is provable = there is a proof of a formula

- A proof is a finite object
- A proof can be checked mechanically by terminating algorithm
- A formula that is true in an interpretation is called **satisfiable.**
- A formula true in every interpretation is said to be **valid**.
  -
  - $\forall x \, P(x) \Rightarrow P(x)$ Validity is undecidable
- A formula is **invalid** if there is at least one interpretation in which it is not true.
- A formula is **unsatisfiable** if there is no interpretation in which it is true.
  - Doesn't matter how you interpret it, it is be false

VALIDITY IS UNDECIDABLE BUT IS CE

- We have a complete proof system for FOL
- Any valid formula has a proof
- Soundness is anything that you prove is valid
- Completeness, anything that is valid, there is a proof for it.

Want to check for validity: generate proofs and see if one is a proof for this formula
If a formula is valid, bc of completeness theory, it must have a proof.

- You will find it.
- Guaranteed to say yes if its valid.

PCP <m VALIDITY (dont need to know details of proof)

- Have to take an example or instance of PCP and show how to transform it into a formula, in such a way that the original PCP has a solution IIF the formula I produce is valid