CS 335 Test 1 [redacted]

1) $f = \sigma(PR(\pi_2^2, \pi_3^4))$

This should be F or something else, not f. $f = \sigma(PR(\pi_2^2, \pi_3^4))$

$f(n_1, n_2, 0) = g(n_2)$ $\quad g = n_2$

$f(n_1, n_2, m+1) = h(n_1, n_2, m, f(n_1, n_2, m))$

$g = n_2$

$h = m$

$f(3,3,0) = 3$
$f(3,3,1) = \cancel{1}\ 0$ $\quad m+1 = 1$ so $m = 0$.
$f(3,3,2) = 2$
$f(3,3,3) = 3$
$f(3,3,4) = \cancel{4}\ 3$

$$F(n_1,n_2,n_3) = \begin{cases} n_2 & \text{if } n_3 = 0 \\ n_3 - 1 & \text{otherwise} \end{cases}$$

$$f(n_1,n_2,n_3) = \begin{cases} n_2 + 1 & \text{if } n_3 = 0 \\ n_3 & \text{otherwise} \end{cases}$$

We still need the sigma,

$\boxed{f(3,3,4) = \cancel{3}\ 4}$

This function is just a for loop, initialized at n_2, then gets the iterator variable m each time. At the end of the for loop (P.R.), we add one (sigma).

2) $<$, $=$, and $<=$ are primitive recursive.

To prove this we start with the definition of a Primitive Recursive function.

- A Primitive Recursive function is any function which can be obtained from $0, \sigma, \pi_i^k$ by using composition "o" and Primitive Recursion (PR)

By showing that Relations are compositions of PR functions, by definition we prove that they themselves are PR functions.

$A \leq B \equiv a \dot{-} b = 0$

- We must show "difference" is PR

$diff(a, 0) = a$
$diff(a, m+1) = prev(diff(a, m))$

- We must show "previous" is PR

$prev(0) = 0$
$prev(m+1) = m$

$A < B \equiv B \dot{-} A > 0$

We already showed $\dot{-}$ is PR

We must show Greater than 0 is PR

greaterThan0 $\equiv$ not(eqTo0(n))

We must show eqTo0 is PR

eqTo0(0) = 1
eqTo0(m+1) = 0

- We must show "not" is PR

not(0) = 1
not(m+1) = 0

$A = B \equiv$ not($a<b$) and not($b<a$)

- We must show "not" is PR

not(0) = 1
not(m+1) = 0

- We must show "and" is PR

Just Multiplication!

$0 * 0 = 0$
$0 * 1 = 0$
$1 * 0 = 0$
$1 * 1 = 1$

- We must show "Multiplication" is PR

mult(a, 0) = 0
mult(a, b+1) = mult(a, b) + a

- We must show "addition" is PR

add(a, 0) = a
add(a, b+1) = ~~add~~ sigma(a, b);

By composing the relationship functions of of PR function, the relations are themselves PR functions

---

4) Every P.R. function is by definition μ-recursive.

A μ-recursive function is any function which can be obtained from 0, σ, and π_i^k by using composition "$\circ$", P.R. and μrecursion.

By definition, P.R. functions are μ-recursive functions.

Not every μ-recursive function is Primitive Recursive

Church-Turing Thesis: Anything that can be computed on a physical device can also be computed by a μrecursive function $\equiv$ Turing Machine $\equiv$ Java Program.

In essence, any computable function is μ-recursive.

To provide a counter example, we consider a while loop.

Primitive Recursive functions are always defined.

Consider the function

$$f(n) = \begin{cases} 3 & \text{if } n = 2 \\ 2 & \text{if } n = 5 \\ \text{undefined} & \text{otherwise} \end{cases}$$

This is clearly not a PR function, since it can be undefined for some inputs.

This is a μ recursive function, but not a P.R. function.

$$f(n) = \mu m. \left((m = 2 \,\&\, m = 3) \vee (n = 5 \,\&\, m = 2) \right)$$

$a-b = a-b, -\emptyset$

$\equiv \underbrace{eqTo\emptyset}_{PR}(\underbrace{sub(a,b)}_{PR})$

so $a=b$ is PR

$a \leq b \equiv \neg(a > b) = \neg(a-b > 0)$

$= \underbrace{neg}_{PR}(\underbrace{grThan\emptyset}_{PR}(\underbrace{sub(a,b)}_{PR}))$

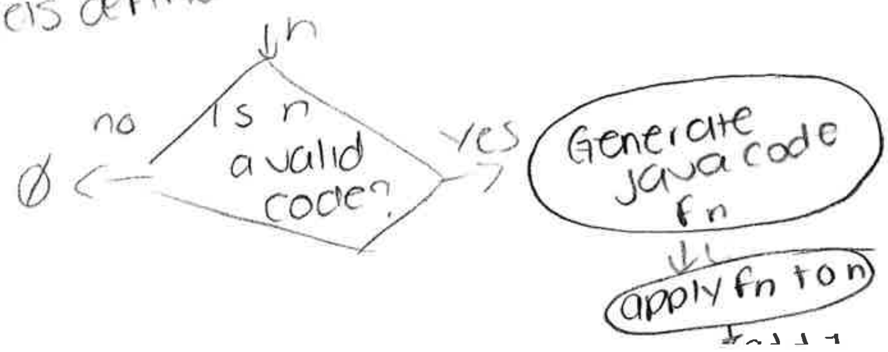
so less than equal is PR

4. - every PR function is μ recursive
By definition
μ m is any function which can be computed from ∅, c, π
by using ∅, PR & μ recursion
so every PR function is μ recursive

* not every μ recursive function is PR.
- we introduce the notion of a code of a PR function.
- a natural # can be assignated to every PR function.
- Every PR function is represented by an expression
- then using ASCII code to translate it into ∅s & 1s
- in order to make the expression of the PR unique we add 1 to the front

There exists an algorithm that given a code c
- checks whether is a valid code of a PR function
- If yes produces a java code f_c which computes this function.

Proof: let's define a function f as follows.



In particular
For $n = c$ $f_c(c) = f(c)$
By definition of f,
since c is a valid code
$f(c) = f_c(c) + 1$
$0 = 1$
contradiction

[redacted]

5 $\boxed{f(a,b,c) = \mu m. \left((a=b=c \,\&\, m=1) \vee (a\neq b \,\&\, b\neq c \,\&\, a\neq c \,\&\, m=0)\right)}$

$$f(a,b,c) = \begin{cases} 1 & \text{if } a=b=c \\ 0 & \text{if } a\neq b,\ b\neq c,\ a\neq c \\ \text{undefined elsewhere} \end{cases}$$

6) The union of two R.E. sets A and B is R.E. i.e. $A \cup B$ is R.E

Proof: We construct an algorithm to print every element in $A \cup B$

Since A is R.E., there exists an algorithm that eventually prints all members of A (alg A)
Since B 〃 〃 〃 〃 〃 〃 〃 〃 of B (alg B)

Algorithm:

Run alg A for 1hr
Run alg B for 1hr
Run alg A for 1hr
Run alg B for 1hr
⋮

We can see that their union will be printed, assuming we don't care for repeats

1	Alg A	1	
2	Alg B	1	
3	Alg A	2	
4	Alg B	2	
5	Alg A	3	— 5 printed — already at moment 5
6	Alg B	3	
7	Alg A	4	
8	Alg B	4	— 5 printed

It will be printed at moment $\boxed{8}$ corresponding to the union algorithm

7) set S & -S are both R.E. Then S is decidable [redacted]

Th141 This means there exists an algorithm that prints all elements of S & -S and since -S contains all elements not in S this mean a number n is either in S or -S. we can find out if n is in S or not this way. we can find this similar to problem 6.

run algo to print elements from S for time t
" " -S for time t.

has n been printed? repeat & print more till it does.
if so is it printed by S or -S?

number 7 queried

run set	times run	times t
S	1	1
-S	1	2
S	2	3
-S	2	4
S	3	5
7 generated → -S	3	6

This algo will find out 7 does not belong to S at time 6

[redacted]

8] we can show this by first testing program $p \le 1$, data $d \le 1$ for time $t=1$
if a (p,d,t) combination halts we print it. we then continue for $p \le 2, d \le 2, t=2$
testing all combinations a total of 2 time steps. Continue in this fashion.

$p \le 1, d \le 1, t=1$

(p,d)	t = time ran
(0,0)	
(0,1)	1
(1,0)	1
(1,1)	1

4

$p \le 2, d \le 2, t=2$

(0,0)	1
(0,1)	1
(0,2)	1
(1,0)	1
(1,1)	2
(1,2)	2
(2,0)	2
(2,1)	2
$\boxed{(2,2) \quad 2}$	

12

13

17 times t

	time ran	
(2,2)	1	First time t
(2,2)	1	← Second time t

the pair $\overset{(p,d,t)}{(2,2,1)}$ will be printed after 17 times at

7) S is R.E.
-S is R.E.
Then S is decidable

A set A is decidable if there exists an algorithm that, given a natural number n, checks whether $n \in A$

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

To prove we build an algorithm that computes χ_S characteristic function of S

S Run 1st Alg S for 1hr
Run 2nd Alg -S for 1hr } if n appeared, we know if $n \in A$ or $n \notin A$

1	Alg S	1
2	Alg -S	1
3	Alg S	2
4	Alg -S	2
5	Alg S	3
6	Alg -S	3

$\longrightarrow$ 7 printed

The algorithm will decide whether 7 belongs to S just after moment 6

---

8) The halting set is R.E., meaning there is an algorithm that each element of the halting set will eventually be printed and only elements of the halting set are printed

H = the halt

We denote the halting set as

$H = \{(p,d) \mid \text{program } p \text{ halts on data } d\}$

To prove H is R.E. we build an algorithm that prints all elements of H

Take all $p \le 1, d \le 1, t = 1$, Run for 1hr, If any of them halts, print (p,d)
Take all $p \le 2, d \le 2, t = 2$ Run for 2hrs.
Take all $p \le 3, d \le 3, t = 3$ Run for 3hrs

p	d
0	0
0	1
1	0
1	1
0	2
1	2
2	0
2	1

4hrs + 4hrs = 8hrs $\longrightarrow$ 18 hrs

10 hrs

It will be printed at moment of time 18

4.- A → decidable
B → RE

By theorem 11: every decidable set is R.E
SO A IS RE.

Then By Theorem 12: union OF RE sets IS RE

A ∪ B IS RE.

IF B = H and A = Ø
Then B IS RE But not decidable. (By theorem 45)
then A ∪ B IS not decidable
It is a counter example Because we are giving a decidable set A = Ø (By th 4 it is decidable) and a RE But not decidable set B = H Therefore the union of A → decidable and B - RE sets is not always decidable

10.- Theorem no algorithm is possible that given a program checks wether p always returns the nth prime.

$$f_c(p) = \begin{cases} 1 & \text{if for every } n \; P(n) \text{ halts } \& \; P(n) = n^{th} \text{ prime} \\ \emptyset & \text{if for every } n \; P(n) \text{ halts } \& \text{ for some } n \; P(n) \neq n^{th} \text{ prime} \end{cases}$$

PROOF We can prove that if f exists we can use f_c to design a zero checker

$P(n) = \emptyset \leftrightarrow q(n) = n^{th}$ pr
then $q(n)$

$$P \longrightarrow \forall n (P(n) = \emptyset)$$
$$\downarrow \qquad \qquad \uparrow$$
$$q \longrightarrow \forall n (q(n) = n^{th} \text{ prime})$$

$P(n) + n^{th}\text{prime} = q$

n^{th}prime checker$(q) = 1$
$\forall n (q(n) = n^{th}$ prime$) \leftrightarrow \forall n (P(n) + n^{th}$ prime $= n^{th}$ prime
$\leftrightarrow \forall n (P(n) = \emptyset$
zerochecker$(p) = n^{th}$primechecker$(p + n^{th}$ prime$)$

[redacted]

9) The union of a decidable set and a R.E. set is always R.E.

Proof

Let set A be decidable

Let set B be R.E.

- According to Theorem 12, the union of R.E. sets is R.E. and we proved this in problem 6 of this test.

- A, being decidable, is also R.E. (Theorem 11)

  Proof: Basic algorithm

```
int n=0;
while(true) {
  if(n∈A)
    print n;
  n++;
}
```

$\xrightarrow{Th\ 11}$ A is RE, B is RE $\xrightarrow{Th 12}$ $A \cup B$ is R.E.

Thus A and B are R.E. ∴ By Theorem $A \cup B$ is R.E.

Such a union i.e. $A \cup B$ is not always decidable. To show this we provide a counter example.

By theorem, $\emptyset$ is decidable, Let $A = \emptyset$

We showed that H is R.E. (problem 8), Let $B = H$

By Theorem 4, we know that Union of decidable sets is decidable

$n \in A \cup B \iff n \in A \vee n \in B$

So for $A \cup B$ to be decidable, we would need B to be decidable. we know is impossible (H is not decidable)

Thus, $A \cup B$ is not always decidable

10) It is not possible to have a primechecker, a program that checks wether an input program always computhes the nth prime number. i.e. $p(n) =$ n-th prime number for all n

$$\text{prime-checker}(p) = \begin{cases} 1 & \text{if } \forall n\ p(n) \text{ halts and } p(n) = \text{n-th prime number} \\ 0 & \text{if } \forall n\ p(n) \text{ halts and } \exists n:\ p(n) \neq \text{n-th prime number} \\ \text{whatever} & \text{otherwise} \end{cases}$$

We will prove prime checkers don't exist by applying problem reduction and building a zero checker out of it.

Lemma: Zero checkers are not possible

Let's assume that prime checker exists. Let's build a zerochecker out of it.

$$p \xrightarrow[\text{function}]{\text{some}} \forall n\,(p(n) = 0)$$

To build it, we want a function q, which, given input n, its output is the n-th prime number and also that $p(n) = 0$, such that $p(n) = 0 \iff q(n) =$ n-th prime number

We define a specific auxilrary program q

$$q(n) = p(n) + \text{n-th prime number}$$

Using our "prime checker" on q, we can see that

$$\text{primechecker}(q) = 1 \iff \forall n\,(q(n) = \text{n-th prime number}) \iff \forall n\,(p(n) + \text{n-th prime number} = \text{n-th prime number})$$

$$\iff \forall_n\,(p(n) = 0)$$

$$\text{primechecker}(q) = 1 \iff \forall_n\,(p(n) = 0)$$
$$\text{prime checker}(q) = 0 \iff \exists n\,(p(n) \neq 0)$$

Thus

$$\underline{\text{zerochecker}(p) = \text{primechecker}(p + \text{n-th prime number})}$$

We have effectively built a zerochecker out of a primechecker. We know that zerochecker don't exist, so this leads to a contradiction, and the assumption is false.

[redacted]

11-12 $f(n) = n+1$

we have:

$\# \underbrace{|| \ldots ||}_{n} \# | \ldots$

we want

$\# \underbrace{|| \ldots ||}_{n} | \# | \ldots$

(start, #) → (add, R)
(add, #) → (gohome, 1, L)
(add, 1) → (add, R)

(gohome, 1) → (gohome, L)
(gohome, #) → (halt)

$g(0) = 0$
$g(n) = n-1$

we have either
$\# | \# | \ldots$
we want
$\# | \# | \ldots$
or
$\# \underbrace{|| \ldots |||}_{n} \# | \ldots$
we want
$\# \underbrace{|| \ldots ||}_{n-1} \# | \ldots$

(start, #) → (check, R)
(Check, #) → (gohome, L)
(Check, 1) → (Sub, R)
(Sub, 1) → (Sub, R)

(Sub, #) → (erase, L)
(erase, 1) → (gohome, #, L)
(gohome, 1) → (gohome, L)
(gohome, #) → (halt)

$f(g(n))$

($start_2$, #) → ($Check_2$, R)
($Check_2$, #) → ($gohome_2$, L
($Check_2$, 1) → (Sub_2, R)
(Sub_2 1) → (Sub_2, R)
(Sub_2, #) → ($erase_2$, L)
($erase_2$, 1) → ($gohome_2$ #, L)
($gohome_2$ 1) → ($gohome_2$ L)
($gohome_2$ #) → (Start)

n=1
$f(g(1))$

in subtract

#11# $start_2$, # → $check_2$, R
Step 1 Step 2 $Check_2$, 1 → Sub_2 R

#11# Sub_2, # → $erase_2$, L
Step 3

#1#1# $erase_2$, 1 → $gohome_2$, #, L
Step 4

#1#1# $gohome_2$, #, start
Step 5

in add

#1#1# Start, # → add, R
Step 6

#11# add, #, gohome, 1, L
Step 7

gohome, #, halt

#11#
Step 8

[redacted]

1) $f(n) = n+1$

(start, #) → (checkBit, R)
(checkBit, 1) → (checkBit, 0, R)
(checkBit, 0) → (goBack, 1, L)
(check, #) → (goBack, L)
(goBack, 0) → (goBack, L)
(goBack, 1) → (goBack, L)
(goBack, #) → halt

$g(0) = 0$
$g(n) = n-1$

Assuming Numbers are layed from Rightmost bit to Leftmost Bit

(start, #) → (lookforBit, R)
(lookforBit, 0) → (lookforBit, R)
(lookforBit, #) → (return, L)
(lookforBit, 1) → (fondBit, L)
(fondBit, 0) → (foundBit, L)
(fondBit, 1) → (foundBit, L)
(fondBit, #) → (flipBit, R)
(flipBit, 0) → (flipBit, 1, R)
(flipBit, 1) → (return, 0, L)
(return, 1) → (return, L)
(return, 0) → (return, L)
(return, #) → Halt.

For general algorithm of composition, instead of halting, jump to start2, (change state names as well).

Could be unary but ok