yz709-ct-sup2

> Supervision work includes: Section 5 - Q1, Q2(Exercise 5), Q3(Exercise 6), Q4 (Exercise 4); Section 8 - Q1(Exercise 8). Could we also have a discussion on Q2 and Q3 in section 8 if we have time?

5 The Hating Problem and undecidability

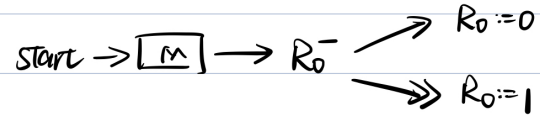
Question 1

1. Show that decidable sets are closed under union, intersection, and complementation. Do all of these closure properties hold for undecidable languages?

complementation:

S is decidable $\Leftrightarrow$ register machine M computes $\chi_S(x) \triangleq \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$

$\Leftrightarrow$ register machine NOT(M) computes $\chi_S'(x) \triangleq \begin{cases} 0 & \text{if } x \in S \\ 1 & \text{if } x \notin S \end{cases}$



$\Leftrightarrow$ $\neg S$ is decidable

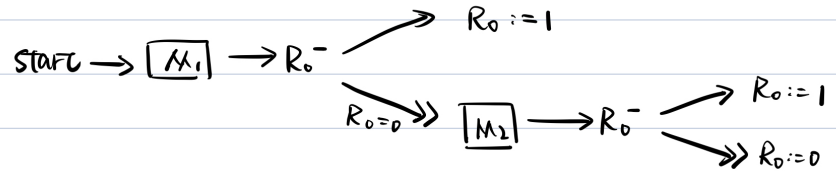
hence NOT(M) can compute $\chi_S'(x)$, decidable set S is closed under complementation

union:

S_1 is decidable $\Leftrightarrow$ RM M_1 computes $\chi_{S_1}(x) \triangleq \begin{cases} 1 & \text{if } x \in S_1 \\ 0 & \text{if } x \notin S_1 \end{cases}$

S_2 is decidable $\Leftrightarrow$ RM M_2 computes $\chi_{S_2}(x) \triangleq \begin{cases} 1 & \text{if } x \in S_2 \\ 0 & \text{if } x \notin S_2 \end{cases}$

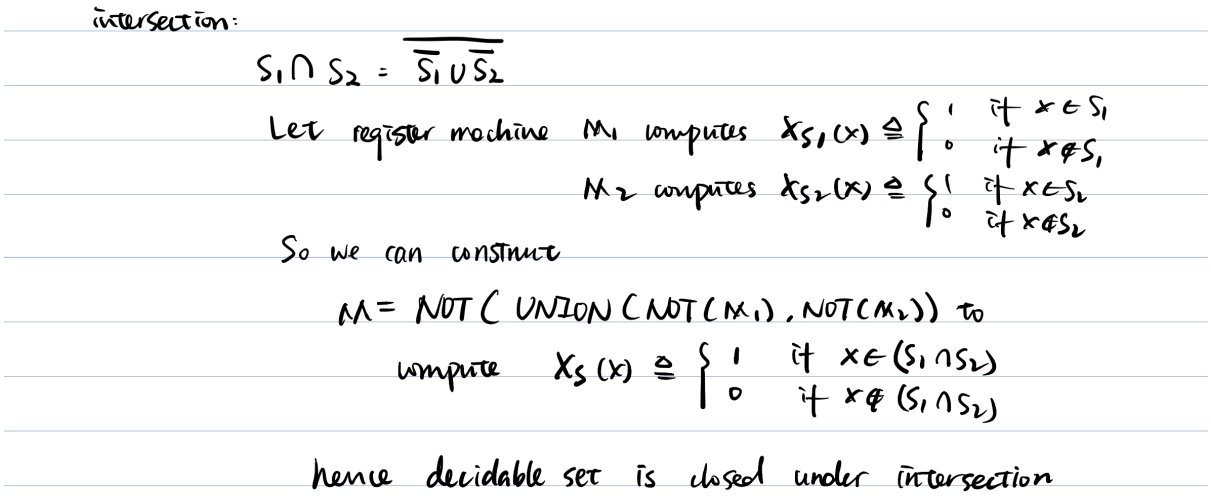
So we can construct a register machine M:



which computes $\chi_S(x) \triangleq \begin{cases} 1 & \text{if } x \in (S_1 \cup S_2) \\ 0 & \text{if } x \notin (S_1 \cup S_2) \end{cases}$

$\Leftrightarrow$ $S_1 \cup S_2$ is decidable

hence UNION(M_1, M_2) $= M$ computes $\chi_S(x)$, decidable sets are closed under union



- For undecidable sets, if set S is undecidable, then showing decidability of S implies the decidability of the Halting problem, hence there doesn't exist a register machine that can compute the characteristic function of set S.
- For complement, assume we have two sets S and $\neg S$: S is undecidable $\rightarrow \neg S$ is undecidable has contrapositive (e.g., $P \rightarrow Q$ has contrapositive $\neg Q \rightarrow \neg P$): $\neg S$ is decidable $\rightarrow S$ is decidable, since $S = \neg\neg S$, we can prove as decidable sets are closed under complementation, hence the undecidable set is closed under complementation.
- For union, suppose set S and set $\neg S$ are both undecidable (this is true because the undecidable set is closed under complementation), then the union of these two sets is Σ^* (all possible strings formed using the grammar), which is decidable, hence undecidable set is not closed under union.
- Similarly, for intersection, suppose set S and set $\neg S$ are both undecidable, then the intersection of these two sets is ϕ (empty set), which is decidable, hence undecidable set is not closed under intersection.

Comments:

Question 2 (Exercise 5)

2. Suppose S_1 and S_2 are subsets of $\mathbb{N}$. Suppose $r \in \mathbb{N} \to \mathbb{N}$ is a register machine computable function satisfying: for all n in $\mathbb{N}$, n is an element of S_1 if and only if $r(n)$ is an element of S_2. Show that S_1 is register machine decidable if S_2 is. Is the converse, inverse, or contrapositive of this statement true?

- Objective: if S_1 is register machine decidable, then S_2 is register machine decidable as well.
    - If S_1 is register machine decidable, then there exists a register machine M_1 computes χ_{S_1}.
    - Since $r \in \mathbb{N} \to \mathbb{N}$ is register machine computable, we can construct a register machine M_r starting with $R_0 := 0, R_1 := n$ and all other zeroed, will halt with $R_0 := r(n)$.
    - Then we can construct a register machine $M_{r^{-1}}$ to computes the inverse function r^{-1}, given input $y \in \mathbb{N}$, enumerate a counter x from 0 onwards, fed x into the register machine M_r, if there is a pair (x, y) such that $y = r(x)$, then we halt with $R_0 := x$, otherwise increment the counter. This procedure will produce $r^{-1}(y)$ for every y in the range of the computable function r, so it will compute r^{-1}.
    - Hence, we can construct a register machine M_2 for computing χ_{S_2}, fed $y \in \mathbb{N}$ into register machine $M_{r^{-1}}$, if it ever produces $R_0 := x \in \mathbb{N}$, then we set $R_0 := 1$. Hence this register machine M_2 computes χ_{S_2}, i.e., S_2 will be decidable.
- Converse: if when S_2 is register machine computable, then S_1 is register machine computable.
    - Build a register machine M_1 upon the register machine M_r, fed input $n \in \mathbb{N}$ into M_r, if it halts with $R_0 := r(n)$, then we set $R_0 := 1$, hence this register machine M_1 computes χ_{S_1}, i.e., S_1 will be decidable.
- Inverse: if S_1 is not register machine computable, then S_2 is not register machine computable.
    - The inverse is the contrapositive of the converse, since the converse of the original statement is true, the inverse statement is logically true.
- Contrapositive: if S_2 is not register machine computable, then S_1 is not register machine computable.
    - Since the original statement is true, its contrapositive statement is true.

💡 Comments:

Question 3 (Exercise 6)

3. Show that the set E of codes $\langle e, e' \rangle$ of pairs of numbers satisfying $\varphi_e = \varphi_{e'}$ is undecidable.

- Objective: show $E \triangleq \{< e, e' > | \varphi_e = \varphi_{e'}\}$ is undecidable - there is no register machine that can decide for any two programs when they are fed with the same input, they halts with the same output.
- We try to show the decidability of E implies the decidability of the Halting problem.
- For all $< e, e' > \in E$, they satisfy $\varphi_e(x)\downarrow \Leftrightarrow \varphi_{e'}(x)\downarrow$ and $\varphi_e(x)\uparrow \Leftrightarrow \varphi_{e'}(x)\uparrow$, in other words, if we have a set $S \triangleq \{e | \varphi(0)\downarrow\}$ and a set $\neg S \triangleq \{e | \varphi(0)\uparrow\}$, either $e, e' \in S$ or $e, e' \in \neg S$.
- However the set S is undecidable because (1) we can construct a register machine H which encompasses a register machine M_0 which computes χ_{S_0}, H is based on the universal register machine (which has input $R_1 := e, R_2 := \lceil [a_1, a_2, ..., a_n] \rceil$) but add another instruction $R_2 := 0$ based on the def of S_0, then run M_0. (2) Since H is a register machine that decides whether the program halts or not, which can be used to solve the Halting problem, as there is no such machine exist, M_0 is not exist, hence S is undecidable.

```
H:
let e = R1 and ⌈[a1, ..., an]⌉ = R2 in
  R1 ::= ⌈(R1 ::= a1; ...; (Rn ::= an); prog(e)⌉;
R2 ::= 0;
run M0
```

- Since undecidable set is closed under complementation, $\neg S$ is undecidable. Hence we can not decide whether e, e' are both in S or $\neg S$. Thus the set E is undecidable.

💡 Comments: I have some problems understanding about proving the undecidable sets, are there any tips when relating a set with the halting problem?

Question 4 (Exercise 4)

4. Show that there is a register machine computable partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ such that both sets $\{ n \in \mathbb{N} \mid f(n)\downarrow \}$ and $\{ y \in \mathbb{N} \mid \exists n \in \mathbb{N}.\, f(n) = y \}$ are register machine undecidable.

- Let $S_1 = \{n \in \mathbb{N} | f(n) \downarrow\}$, $S_2 = \{y \in \mathbb{N} | \exists n \in \mathbb{N}. f(n) = y\}$.
- For the computable partial function $f \in \mathbb{N} \rightharpoonup \mathbb{N}$, there is a register machine M_f runs starting with $R_0 = 0, R_1 = x$ and all other registers set to 0 and halts with $R_0 = y$ if and only if $f(x) = y$, i.e., $f(x) \downarrow$. Define the program runs on M_f be $prog(e)$.
- Objective 1: S_1 is register machine undecidable, there does not exist a register machine M_1 such that it can compute whether $prog(e)$ halts with input $n \in \mathbb{N}$, i.e., computes χ_{S_1}
  - Construct H based on the universal register machine which has input $R_1 := e, R_2 := \ulcorner [a_1, a_2, ..., a_n] \urcorner$ but adds another instruction $R_2 := n$ based on the definition of S_1, then run M_1
  - So essentially H computes the halting problem, as we have already proved there does not exist such a machine, M_1 does not exist as well, i.e., χ_{S_1} is uncomputable, S_1 is undecidable

```
H:
let e = R1 and [[a1, ..., an]] = R2 in
  R1 ::= ⌈(R1 ::= a1; ...; (Rn ::= an); prog(e)⌉;
R2 ::= n;
run M1
```

- Objective 2: S_2 is register machine undecidable, there does not exist a register machine M_2 such that it can compute whether $prog(e')$ halts with input $y \in \mathbb{N}$, where $prog(e')$ computes the inverse function f^{-1} (we have proved the inverse function of a computable function is computable).

- Reference to the proof of “the inverse function of a computable function is computable”:
  - construct a register machine $M_{f^{-1}}$ to computes the inverse function f^{-1}, given input $y \in \mathbb{N}$, enumerate a counter x from 0 onwards, fed x into the register machine M_f, if there is a pair (x, y) such that $y = f(x)$, then we halt with $R_0 := x$, otherwise increment the counter. This procedure will produce $f^{-1}(y)$ for every y in the range of the computable function f, so it will compute f^{-1}.
- Construct H based on the universal register machine which has input $R_1 := e, R_2 := \ulcorner [a_1, a_2, ..., a_n] \urcorner$ but adds another instruction $R_2 := y$ based on the definition of S_2, then run M_2
- So essentially H computes the halting problem, as we have already proved there does not exist such a machine, M_2 does not exist as well, i.e., χ_{S_2} is uncomputable, S_2 is undecidable

```
H:
let e = R1 and ⌜[a1, ..., an]⌝ = R2 in
  R1 ::= ⌜(R1 ::= a1; ...; (Rn ::= an); prog(e')⌝;
R2 ::= y;
run M2
```

💡 Comments:

6 Turing machines

Question 1 (Optional)

1. Compare and contrast register machines with Turing machines: how do they keep track of state, how are programs represented, what form do machine configurations and computations take?

- Similarity: Register machine computable = Turing computable

Differences	Register machine	Turing machine

Differences	Register machine	Turing machine
Tracking state	- The register machine configuration $c = (l, r_1, r_2, ..., r_n)$ indicates the state, l is the current label of instruction, and $r_1, r_2, ..., r_n$ are values stored in all registers. - State transits based on the instruction under the current label	- A Turing machine is specified by (Q, Σ, s, δ) where Q is a finite set of machine states, we use $q \in Q$ to represent the current state - State changes via the transition function δ
Represent program	- Encodes a program into a set of instructions of the form `label:body`, the body could be one of the elementary operations: $(1)\mathrm{HALT}, (2)R_i^+ \to L_j, (3)R_i^- \to L_j, L_k$	- A finite or infinite tape, each tape cell $\in \Sigma$, a tape head starts by pointing to $\triangleright$
Machine configuration	- The register machine configuration $c = (l, r_1, r_2, ..., r_n)$ indicates the state of the machine l is the current label of instruction, and $r_1, r_2, ..., r_n$ are values stored in all registers.	- Configuration $c = (q, w, u)$, starting from initial configuration $(s, \triangleright, u)$ - (1) $q \in Q \cup \{acc, rej\}$: current state; - (2) $w = va$ is an non-empty string to the left of the tape head (which points to a); - (3) u is possibly an empty string (all blanks $\sqcup$) to the right of the tape head
Machine computation	- A register machine computation is a finite or infinite sequence of configurations. - The computation may halt properly or erroneously, or never halt. - Computation is deterministic: the next configuration is uniquely determined by the program.	- Computation is a finite or infinite sequence of configurations c_i, starting from the initial $(s, \triangleright, u)$, we have transitions $c_i \to_M c_{i+1}$ holds for each $i = 0, 1, ...$ - Computation halts if the sequence is finite and last element is of the form (acc, w, u) or (rej, w, u) - A transition is depend on the input symbol and the current state, transit to another state and take corresponding actions: reads or writes tape cells, moving to the left or the right

💡 Comments:

Question 2 (Optional)

2. Familiarise yourself with the Chomsky hierarchy and explain the connection between regular expressions and Turing machines.

- Chomsky hierarchy divides grammar into 4 types, $T_3 \subset T_2 \subset T_1 \subset T_0$
  - T_0 are unrestricted grammars that could be recognised by a Turning machine
  - T_1 are context-sensitive grammar that could be accepted by linear bound automata
  - T_2 are context-free grammar that could be accepted by push-down automata
  - T_3 are regular grammar that can be accepted by finite automata
- Regular expressions can be recognised by Turing machines.
  - Regular expressions can be accepted by a DFA with definition $(Q, \Sigma, s, \delta, F)$ which are Q (a finite set of states), Σ (alphabet), s (an initial state), δ (transition function between states), and F (accepting states)
  - We can construct a register machine analogous to a DFA, with $Q' \cup \{acc, rej\}$ (a finite set of machine states), Σ' (a finite set of tape symbols), $s' \in Q'$ (an initial machine state), $\delta' \in (Q' \times \Sigma') \to (Q' \cup \{acc, rej\}) \times \Sigma \times \{L, R, S\}$ (a transition function), acc be the accepting state.
  - For each transition $\delta(p, a) = q$ in the DFA, the analogous in the Turing machine would be $\delta'(p, a) = (q, a, R)$.
  - From the Chomsky hierarchy, since $T_3 \subset T_0$, i.e., regular language is a subset of unrestricted language, since unrestricted grammar can be recognised by a Turning machine, any regular language can be accepted by a Turing machine.

💡 Comments:

7 Notions of computability

Question 1 (Optional)

1. Before the formal development of the field of computation theory, mathematicians often used the term *effectively computable* to describe functions that can – in principle – be computed using mechanical, pen-and-paper methods.
   a) How was the notion of effective computability formalised by Church and Turing, and generalised to other models of computation?
   b) Suppose we invented a new model of computation. How can we establish that it is as "powerful" as mechanical methods? Make sure to formally explain what "power" means in this case.
   c) Can our new model be even more powerful?

- (a):
  - The Church and Turing Thesis states that any real-world computation can be translated into an equivalent computation involving a Turing machine.
  - Although it is not proved, the fact that every realistic model of computation has shown to be equivalent has been strong evidence of its validity.
- (b): assume the new model is called modelX
  - Power equivalence means every Turning computable function is modelX computable and vice versa.
  - We have to prove in both directions: (1) construct a modelX machine to simulate the computation of a Turing machine; (2) construct a Turing machine to simulate the computation of a modelX machine.
- (c): No, if our model is more powerful than a Turing machine, then it is a Turing oracle, i.e., we can use this model to solve undecidable problems such as the Halting problem.

💡 Comments:

Question 2 (Optional)

2. Briefly describe of three Turing-complete models of computation not covered in the course.

- Despite the Turing machine, λ-calculus, register machines and partial recursive functions, the Church-Turing thesis encompasses other kinds of computations such as cellular automata, random access machines, and rewrite systems.
  - Cellular automata: (1) a collection of coloured cells on a grid of specified shape (e.g., 1-dimension lines, 2-dimension square/triangle/hexagonal grid); (2) the colour of the cells would change according to a set of rules based on the states of neighbouring cells; (3) the rules are applied iteratively for a specified number of time steps.
  - Random access machines: (1) An infinite input tape I whose cells can hold a natural number of arbitrary size with a read head position $i \in \mathbb{N}$; (2) An infinite output tape O whose cells can hold a natural number of arbitrary size with a write head position $o \in \mathbb{N}$; (3) An accumulator A which can hold a natural number of arbitrary size; (4) A program counter C which can hold an arbitrary natural number; (5) A program consisting of a finite number of instructions $P[1], .., P[m]$; (6) A memory consisting of a countably infinite sequence of registers $R[1], R[2], ...$ each of which can hold an arbitrary natural number.
    - A Turing machine can use five tapes to simulate the RAM; every instruction of the RAM is simulated by a sequence of steps of the Turing machine; hence every RAM can be simulated by a Turing machine.
  - Rewrite systems: every computable function can be represented by a term rewriting system: (1) a set of rules of the form $L \to R$, where L, R are terms such that L is not a variable and every variable that appears in R must also appear in L; (2) a rewriting step $T \to T'$ means T' can be derived from T: if we have some rule $L \to R$, and a substitution σ such that $U = L\sigma$, then we can replace U with $R\sigma$ to derive T'; (3) a rewriting sequence would be the form $T_1 \to^* T_2$ where T_2 is in normal form, and no further reduction is possible.

- Moreover, it also applies to most programming languages (with unlimited memory assumed) and other kinds of computations found in theoretical computer science, such as quantum computing and probabilistic computing.
    - Quantum computers are more efficient; they can perform many common tasks with less time complexity than modern ones. But it is still equivalent to a Turing machine, thus unable to answer questions like the Halting problem.

Comments:

8 Partial recursive functions

Question 1 (Exercise 8)

1. Show that the following functions are all primitive recursive. Make sure to give the final form of the function as a composition of primitive functions and projection.

    a) Truncated subtraction function, $minus\colon \mathbb{N}^2 \to \mathbb{N}$, where

    $$minus(x,y) \triangleq \begin{cases} x-y & \text{if } y \leq x \\ 0 & \text{if } y > x \end{cases}$$

    b) Exponentiation, $exp\colon \mathbb{N}^2 \to \mathbb{N}$, where $exp(x,y) = x^y$.

    c) Conditional branch on zero, $ifzero\colon \mathbb{N}^3 \to \mathbb{N}$, where

    $$ifzero(x,y,z) \triangleq \begin{cases} y & \text{if } x = 0 \\ z & \text{if } x > 0 \end{cases}$$

    d) Bounded summation: if $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is primitive recursive, then so is $g : \mathbb{N}^{n+1} \to \mathbb{N}$ where where

    $$g(\vec{x},x) \triangleq \begin{cases} 0 & \text{if } x = 0 \\ f(\vec{x},0) & \text{if } x = 1 \\ f(\vec{x},0) + \cdots + f(\vec{x},x-1) & \text{if } x > 1 \end{cases}$$

Q1

a) $minus(x,0) = x \Rightarrow f(x) \triangleq x \Rightarrow f = proj_1^1$

$minus(x,y+1) = pred(minus(x,y)) \Rightarrow g(x,y,h) \triangleq Pred(h) \Rightarrow g = Pred \circ Proj_3^3$

hence $minus \in \mathbb{N}^2 \to \mathbb{N} = \rho^1(proj_1^1, Pred \circ Proj_3^3)$

* $|x-y| = minus(x,y) + minus(y,x)$ is also primitive recursive

b) $\begin{cases} exp(x,0) = 1 \Rightarrow f(x) \triangleq 1 \Rightarrow f = succ \circ zero^1 \\ exp(x,y+1) = x * exp(x,y) \Rightarrow g(x,y,w) \triangleq x*h \Rightarrow g = mult \circ (proj_1^3, proj_3^3) \end{cases}$

where $mult = \rho^1(zero^1, add \circ (proj_3^3, proj_1^3))$

hence $exp \in \mathbb{N}^2 \to \mathbb{N} = \rho^1(succ \circ zero^1, mult \circ (proj_1^3, proj_3^3))$

c)

$\begin{cases} ifzero(0,y,z) = y \\ ifzero(x+1,y,z) = z \end{cases}$ $\quad f(y,z) \triangleq y \Rightarrow f = proj_1^2$ $\quad g(x,y,z,h) \triangleq z \Rightarrow g = proj_3^4$

$\Rightarrow ifzero \in \mathbb{N}^3 \to \mathbb{N} = \rho^2(f,g) = \rho^2(proj_1^2, proj_3^4)$

d)

$\begin{cases} g(\vec{x},0) = 0 \Rightarrow f_g(\vec{x}) \triangleq 0 \Rightarrow f_g = zero^n \\ g(\vec{x},x+1) = g(\vec{x},x) + f(\vec{x},x) \end{cases}$ where $f(\vec{x},x) = \rho^n(f_f, g_f)$

$\Rightarrow g_g(\vec{x},x,h) \triangleq h + f(\vec{x},x)$

$\Rightarrow g_g = add \circ (proj_{n+2}^{n+2}, \rho^n(f_f, g_f))$

hence $g = \rho^n(zero^n, add \circ (proj_{n+2}^{n+2}, \rho^n(f_f, g_f)))$

Intuition:

$g(\vec{x},0) = 0$

$g(\vec{x},1) = f(\vec{x},0)$

$g(\vec{x},2) = f(\vec{x},0) + f(\vec{x},1)$

$g(\vec{x},3) = f(\vec{x},0) + f(\vec{x},1) + f(\vec{x},2)$

$= g(\vec{x},2) + f(\vec{x},2)$

Question 2 (Optional)

2. Explain the motivation and intuition behind *minimisation*. How does it extend the set of functions computable using primitive recursion? Give three examples of computable partial functions that are not definable using primitive recursion, justifying your answer in each case.

- Motivation and how it extends primitive recursion:

- Since primitive recursion only formalises bounded repetition (e.g., if-else condition), we need minimisation to formalise unbound repetition (e.g., while condition) to define all computable functions. However, it also introduces the possibility of functions not terminating (e.g., not total).

- Examples:
  - The integer part of logarithmic function $\log_2 x$ where $x \neq 0$, we need to find the least y such that $2^y \geq x \rightarrow y = \mu^1 f(x)$, where $f \in \mathbb{N}^2 \rightharpoonup \mathbb{N}$:

$$f(x,y) = \begin{cases} 1 & \text{if } x > 2^y \\ 0 & \text{if } x \leq 2^y \end{cases}$$

    - If we defined the function such that it is 0 when $x = 0$, then it can be shown to be in the set $PRIM$
  - The integer part of the division $\frac{x_1}{x_2}$ where $x_2 \neq 0$, we need to find the least x_3 such that $x_1 < x_2(x_3 + 1) \rightarrow x_3 = \mu^2 f(x_1, x_2)$ where $f \in \mathbb{N}^3 \rightharpoonup \mathbb{N}$

$$f(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } x_1 \geq x_2(x_3 + 1) \\ 0 & \text{if } x_1 < x_2(x_3 + 1) \end{cases}$$

    - If we define the function such that it is 0 when $x_2 = 0$ (because otherwise, division by 0 is undefined), then it can be shown to be in the set $PRIM$
  - The integer part of the function $\tan(x)$ where $x \neq \frac{\pi}{2} + n\pi$ for $n \in \mathbb{N}$, we need to find the least y such that $\tan(x) \leq y \rightarrow y = \mu^1 f(x)$, where $f \in \mathbb{N}^2 \rightharpoonup \mathbb{N}$:

$$f(x,y) = \begin{cases} 1 & \text{if } \tan(x) > y \\ 0 & \text{if } \tan(x) \leq y \end{cases}$$

> 💡 Comments:

Question 3 (Optional)

3. Use minimisation to show that the following functions are partial recursive:
    a) the binary maximum function $max\colon \mathbb{N}^2 \to \mathbb{N}$.
    b) the integer square root function $sqrt\colon \mathbb{N} \rightharpoonup \mathbb{N}$ which is only defined if its argument is a perfect square.

- (a) Maybe I didn't understand the question, but surely we can represent max as a primitive recursive function?
    - $\max(x, y) = monus(x, y) + y$, where $monus$ is the truncated subtraction
    - Hence we can represent it as $\max = add \circ (monus, proj_2^2)$ where $monus = \rho^1(proj_1^1, pred \circ proj_3^3)$
- (b)
    - For all perfect square x, we need to find the least y such that $y^2 \geq x$. Let $y = \mu^1 f(x)$, where $f \in \mathbb{N}^2 \rightharpoonup \mathbb{N}$:

$$f(x, y) = \begin{cases} 1 & \text{if } y^2 < x \\ 0 & \text{if } y^2 \geq x \end{cases}$$

    - Hence we have shown the partial function f is closed under minimisation.

> 💡 Comments:

Optional exercises

Question 1

1. Write a Turing machine simulator in a programming language you prefer (a functional language such as ML or Haskell is recommended). Implement the machine described on Slide 64.

Question 2

2. For the example Turing machine given on Slide 64, give the register machine program implementing $(S, T, D) := \delta(S, T)$, as described on Slide 70.

Question 3

3. Recall the definition of Ackermann's function ack. Sketch how to build a register machine M that computes $ack(x_1, x_2)$ in R_0 when started with x_1 in R_1 and x_2 in R_2 and all other registers zero. (E9)

   Hint: Call a finite list $L = [(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots]$ of triples of numbers *suitable* if it satisfies

   a) if $(0, y, z) \in L$, then $z = y + 1$

   b) if $(x + 1, 0, z) \in L$, then $(x, 1, z) \in L$

   c) if $(x + 1, y + 1, z) \in L$, then there is some u with $(x + 1, y, u) \in L$ and $(x, u, z) \in L$.

The idea is that if $(x, y, z) \in L$ and L is suitable then $z = ack(x, y)$ and L contains all the triples $(x', y', ack(x, y'))$ needed to calculate $ack(x, y)$. Show how to code lists of triples of numbers as numbers in such a way that we can (in principle, no need to do it explicitly!) build a register machine that recognises whether or not a number is the code for a suitable list of triples. Show how to use that machine to build a machine computing $ack(x, y)$ by searching for the code of a suitable list containing a triple with x and y in its first two components.