Computation Theory

Solutions

2021

Contents

1. Algorithmically undecidable problems

1. Two important concepts in the theory of computability are *enumerations* and *diagonalisation*. Intuitively, an enumeration of a set S is an ordered, "exhaustive" listing of all elements. While this intuition works for finite sets, we need to be more formal to handle infinite sets. Thus, an *enumeration* of a finite or infinite set S is a surjective function from the natural numbers $\mathbb{N}$ to S, if it exists. If it does, the set S is called *countable*; if it doesn't, it is *uncountable*.

   Prove or disprove the following statements:

   a) The set of natural numbers is countable.

   > Yes, enumerated by the identity function $\mathrm{id}_{\mathbb{N}} : \mathbb{N} \twoheadrightarrow \mathbb{N}$.

   b) The set of integers is countable.

   > Yes, the enumeration alternates between positive and negative numbers: $0, 1, -1, 2, -2, 3, -3, \ldots$. Explicitly,
   >
   > $$\varphi(n) \triangleq \begin{cases} \frac{n+1}{2} & \text{if } n \text{ is odd} \\ -\frac{n}{2} & \text{if } n \text{ is even} \end{cases}$$

   c) The Cartesian product of two countable sets is countable.

   > Perhaps surprisingly, yes: the enumeration traverses the "multiplication table" from one corner diagonally. Given two countable sets S and T, they can be enumerated to create two coordinate axes, with the points representing elements of $S \times T$. The systematic, exhaustive enumeration of the table starts at the upper left corner with (s_0, t_0), moving to the right to (s_0, t_1), then diagonally down and to the left to (s_1, t_0), down to (s_2, t_0), diagonally up and to the right to (s_1, t_1), further to (s_0, t_2), and so on. Despite both dimensions being infinite, this method will cover every pair in $S \times T$.
   >
   > $$\begin{array}{cccccc} (s_0,t_0) & \to & (s_0,t_1) & & (s_0,t_2) \to (s_0,t_3) & \cdots \\ \swarrow & \nearrow & & \swarrow & & \\ (s_1,t_0) & & (s_1,t_1) & & (s_1,t_2) \quad (s_1,t_3) & \cdots \\ \downarrow & \nearrow & & \swarrow & & \\ (s_2,t_0) & & (s_2,t_1) & & (s_2,t_2) \quad (s_2,t_3) & \cdots \\ \swarrow & & & & & \\ (s_3,t_0) & & (s_3,t_1) & & (s_3,t_2) \quad (s_3,t_3) & \cdots \\ \vdots & & \vdots & & \vdots \quad \vdots & \ddots \end{array}$$

   d) The set of rational numbers is countable.

   > Yes, since any rational number can be represented by an ordered pair of the integer numerator and integer denominator, integers are countable, and the Cartesian product $\mathbb{Z} \times \mathbb{Z}$ is therefore also countable. Note that the enumeration will include every fraction an infinite number of times (since $\frac{2}{3}$, $\frac{4}{6}$, $\frac{8}{12}$ all denote the same fractions but correspond to different pairs), but since we only ask for a surjection $\mathbb{N} \twoheadrightarrow \mathbb{Z} \times \mathbb{Z}$, not a bijection,

this will not be an issue.

e) The finite n-ary product of countable sets is countable.

Any n-ary product $S_0 \times S_1 \times S_2 \times \cdots \times S_n$ is isomorphic/equivalent to a nested binary product $(\cdots((S_0 \times S_1) \times S_2) \times \cdots) \times S_n$. S_0 and S_1 are countable, so $S_0 \times S_1$ will be countable, and in turn, $(S_0 \times S_1) \times S_2$ will be countable, and so on. As long as the number of sets is finite, the product will be countable.

f) The set of polynomials with coefficients from a countable set is countable.

Polynomials are finite sums of terms consisting of a natural power of the variable and a countable (e.g. rational) coefficient. A polynomial $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$ of degree n (meaning that the highest exponent of the variable is n) can therefore be represented as an n-tuple of the coefficients $(a_n, a_{n-1}, \ldots, a_1, a_0)$, with $a_k = 0$ if a term of degree k does not appear in the polynomial. Thus, the set of all polynomials of degree n is isomorphic to the n-ary Cartesian product of the set of coefficients. If this set is countable, the product will also be countable from part (e) above.

g) The powerset of a countable set is countable.

This is false: there is no surjection from $\mathbb{N}$ to $\mathcal{P}(S)$ for a countable S, i.e. there are infinite sets that are "more infinite", than the countably infinite set of natural numbers. This deep result was established by Georg Cantor in 1891 using his famous *diagonal argument*, which has since been applied to many other nonexistence proofs. A more general version of the statement is known as Cantor's Theorem:

There is no surjection $f : A \to \mathcal{P}(A)$ from a set A to its powerset.

The proof proceeds by contradiction. Assume f is a surjection, i.e. for every subset $S \subseteq A$ there is an $a \in A$ such that $f(a) = S$. In particular, consider the subset $D \triangleq \{x \in A \mid x \notin f(x)\}$ of elements $x \in A$ which are not in $f(x)$. Since f is surjective, there exists an associated $a \in A$ such that $f(a) = D$. But then $a \in D$ would imply that $a \notin f(a)$, and $a \notin D$ would imply that $a \in D$ – $a \in D \iff a \notin D$ is a contradiction, so the assumption that f is surjective was wrong.

As a corollary of Cantor's theorem we get that there is no surjection $\mathbb{N} \to \mathcal{P}(\mathbb{N})$, so there are indeed countable sets whose powerset is not countable.

The intuition behind the construction of the *diagonal set* D for the case of $\mathbb{N}$ is the following. The contradictive assumption $f : \mathbb{N} \twoheadrightarrow \mathcal{P}(\mathbb{N})$ states that $\mathcal{P}(\mathbb{N})$ is countable, so we have an exhaustive listing of all subsets of the natural numbers. However, for any such listing we can construct a set of naturals that cannot be in the listing, so it couldn't have been exhaustive. To construct this set, we look at whether $n \in \mathbb{N}$ occurs in the n^{th} set of the enumeration. If $n \notin f(n)$, we include n in D, otherwise we don't. Thus, by construction, every set in the enumeration will differ from D in at least one element: there may be a $k \in \mathbb{N}$ such that $f(k)$ is nearly identical to D, but they will

certainly differ in their inclusion of k. This D can be constructed for any listing, so no listing can be exhaustive – $\mathcal{P}(\mathbb{N})$ is not countable.

$f(1) = \{$	1,											$\}$
$f(2) = \{$	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	$\dots\}$
$f(3) = \{$		2,	3,	4,		6,		8,		10,		$\dots\}$
$f(4) = \{$	1,		3,	4,	5,		7,		9,		11,	$\dots\}$
$f(5) = \{$	1,	2,		4,	5,	6,	7,		9,		11,	$\dots\}$
$f(6) = \{$			3,	4,		6,	7,		9,	10,		$\dots\}$
$f(7) = \{$	1,				5,		7,		9,			$\dots\}$
$f(8) = \{$			3,	4,			7,	8,			11,	$\dots\}$
$f(9) = \{$	1,	2,			5,	6,			9,	10,		$\dots\}$
$f(10) = \{$	1,	2,		4,	5,	6,			9,	10,	11,	$\dots\}$
$f(11) = \{$	1,	2,		4,		6,			9,		11,	$\dots\}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

$T = \{$	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	$\dots\}$

h) The set of real numbers is countable.

Real numbers have a countably infinite decimal expansion, so every real number corresponds to a subset of the natural numbers. Since subsets of naturals are not enumerable (Cantor's Theorem), the set of real numbers will not be enumerable either. Another similar diagonal construction creates a real number that cannot be part of any listing by setting the n^{th} digit of its decimal expansion to be different from the n^{th} digit of the n^{th} real number in the listing. By the same reasoning as above, this new real number will not be in the listing by construction, so it cannot be exhaustive.

Feel free to do some research if you are not familiar with these results.

2. Rephrase the proof of the undecidability of the Halting Problem (with an abstract definition of an algorithm) as a diagonal argument.

Any algorithm (implemented as a register machine, Turing machine, etc.) can be encoded as a natural number, so algorithms are enumerable – they can be exhaustively listed. Consider therefore the following table, where the vertical axis is the n^{th} algorithm A_n in the listing, and the horizontal axis is the *code* for the n^{th} algorithm $\ulcorner A_n \urcorner$. We can populate the $(n,k)^{\text{th}}$ cell of the table with the behaviour of A_n on $\ulcorner A_k \urcorner$: whether it halts (with value 0 or 1), or it doesn't (denoted by $\times$):

	$\ulcorner A_0 \urcorner$	$\ulcorner A_1 \urcorner$	$\ulcorner A_2 \urcorner$	$\ulcorner A_3 \urcorner$	$\cdots$
A_0	1	0	$\times$	1	$\cdots$
A_1	$\times$	$\times$	1	$\times$	$\cdots$
A_2	1	$\times$	0	1	$\cdots$
A_3	0	1	$\times$	$\times$	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The Halting Problem amounts to constructing an algorithm H such that $H(A, D) = 1$ if $A(D)$ halts, and $H(A, D) = 0$ otherwise. That is, the computation table of H for the inputs A_n and $\ulcorner A_k \urcorner$ would be the table above, with 0 and 1 replaced with 1 and $\times$ replaced with 0.

	$\ulcorner A_0 \urcorner$	$\ulcorner A_1 \urcorner$	$\ulcorner A_2 \urcorner$	$\ulcorner A_3 \urcorner$	$\cdots$
A_0	1	1	0	1	$\cdots$
A_1	0	0	1	0	$\cdots$
A_2	1	0	1	1	$\cdots$
A_3	1	1	0	0	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The undecidability of the Halting Problem means that such a H does not exist. Assume, for contradiction, that we do have such a decider H. Then we can construct a new algorithm D that takes descriptions of algorithms $\ulcorner A_k \urcorner$ and operates the following way:

$$D(\ulcorner A_k \urcorner) \triangleq \begin{cases} 0 & \text{if } H(A_k, \ulcorner A_k \urcorner) = 0 \\ \uparrow & \text{otherwise} \end{cases}$$

The graph of D is precisely the diagonal of the computation table for H above, with 0 changed to 1 and 1 changed to $\uparrow$. D differs from every line of H in at least one position: if it didn't, there would be a $k \in \mathbb{N}$ such that $A_k = D$, and $A_k(\ulcorner A_k \urcorner) = D(\ulcorner D \urcorner)$ would halt if and only if $H(D, \ulcorner D \urcorner) = 0$, i.e. $D(\ulcorner D \urcorner)$ didn't halt. Thus D cannot be in the listing of computable algorithms, so it's not computable; but since it was computably constructed from H, this implies that the computable halting function H cannot exist.

Note the subtle difference between this proof and Cantor's Theorem: we're not trying to prove that there is no exhaustive listing of algorithms as we can always construct a new one which is not in the list, since we already know that algorithms are computable (due to a possibly bijective encoding scheme with natural numbers). Instead, we're proving that a specific machine cannot be computable because it cannot be in the exhaustive listing of computable functions.

2. Register machines

1. Show that the following arithmetic functions are all register machine computable.

a) First projection function $p \in \mathbb{N} \to \mathbb{N}$, where $p(x, y) \triangleq x$
b) Constant function with value $n \in \mathbb{N}$, $c_n \in \mathbb{N} \to \mathbb{N}$ where $c(x) \triangleq n$
c) Truncated subtraction function, $\_ \dot{-} \_ \in \mathbb{N}^2 \to \mathbb{N}$, where

$$x \dot{-} y \triangleq \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{if } y > x \end{cases}$$

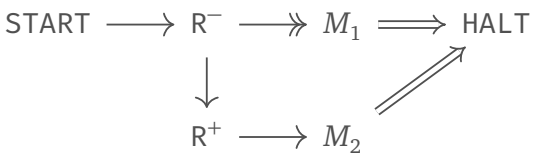
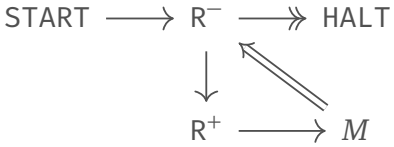
d) Integer division function, $\_\text{div}\_ \in \mathbb{N}^2 \to \mathbb{N}$ where

$$x \text{ div } y \triangleq \begin{cases} \text{integer part of } x/y & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases}$$

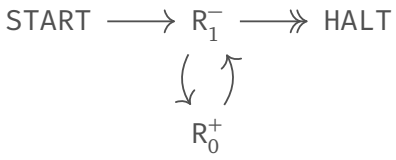
e) Integer remainder function, $\_\text{mod}\_ \in \mathbb{N}^2 \to \mathbb{N}$ with $x \bmod y \triangleq x \dot{-} y \cdot (x \text{ div } y)$
f) Exponentiation base 2, $e \in \mathbb{N} \to \mathbb{N}$, where $e(x) = 2^x$
g) Logarithm base 2, $\log_2 \in \mathbb{N} \to \mathbb{N}$, where

$$\log_2(x) \triangleq \begin{cases} \text{greatest } y \text{ such that } 2^y \leq x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

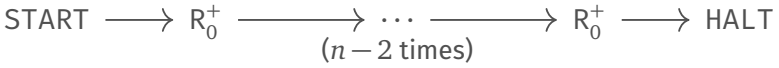
Hint: instead of defining everything from scratch, try implementing these machines with the help of general control flow components.

> As hinted in the question, we can save some effort by working at a higher level of abstraction rather than individual register operations. Specifically, we can create RM "combinators" corresponding to the three fundamental building blocks of programming: sequential composition, conditional branching and iteration. Any register machine is equivalent to one with a single halt state, so abstractly an RM program looks like
>
> $$\text{START} \longrightarrow M \Longrightarrow \text{HALT}$$
>
> Then, we have ways of combining RMs with sequential composition
>
> $$\text{START} \longrightarrow M_1 \Longrightarrow M_2 \Longrightarrow \text{HALT}$$
>
> conditional branching with `if R = 0 then` M_1 `else` M_2
>
> 
>
>
> and iteration with `while R ≠ 0 do` M
>
> 
>

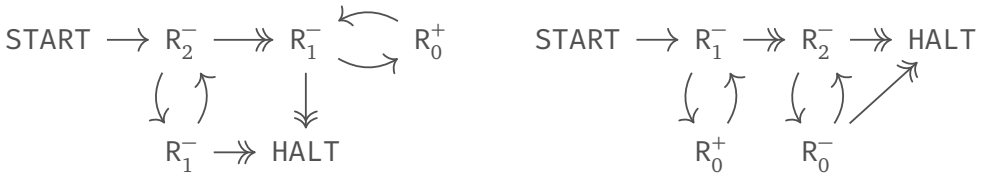
a) First projection: copy over R_1 to R_0.



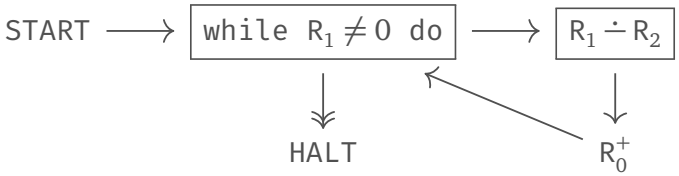
b) Constant function: the constant value is not an argument, it has to be "baked into" the register machine. We can illustrate this schematically – in principle this machine can be constructed for any finite n.



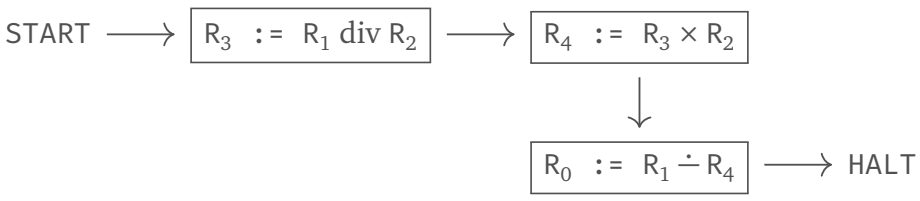
c) Truncated subtraction: subtract R_2 from R_1, then move R_1 to R_0. Alternatively, copy R_1 to R_0, and then subtract R_2 from R_0.



d) Integer division: repeated truncated subtraction. Abstractly, we can write this as:



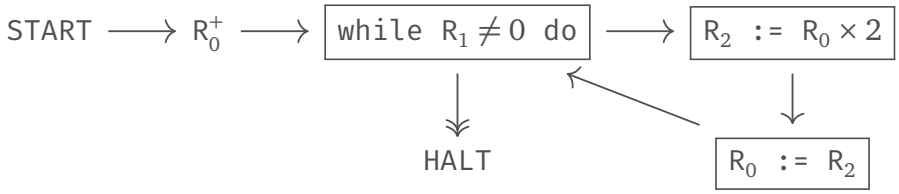
e) Integer remainder: compose previously defined computation blocks together according to the definition. Assignment and multiplication is defined in the notes; doing computation and assignment in the same expression is just a matter of using an auxiliary register to store the intermediate value.



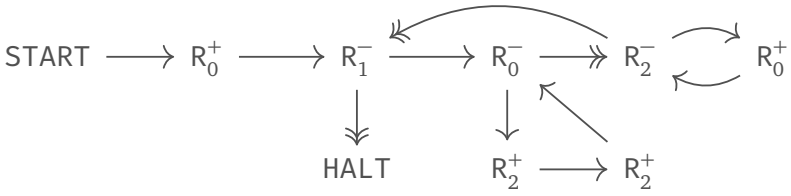
One might well argue that such programs can be written in a more efficient way from scratch, without using any higher-level abstractions. This might indeed be the case, but remember that for the purposes of computability we only care about whether performing the computation is possible *at all*, not necessarily if there is an efficient implementation. And given that our blocks are just shorthands for low-level register

machines, all of these high-level definitions can be expanded into individual register operations, just like in a real computer.

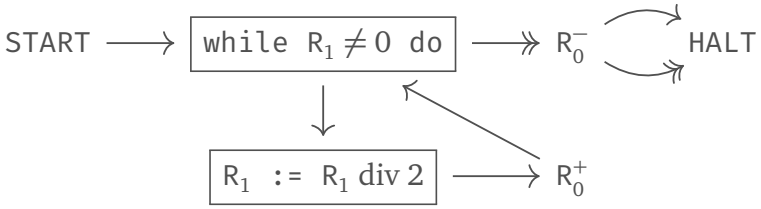
f) Exponentiation base 2: initialise R_0 to 1, then repeatedly multiply R_0 R_1 times.



As an example of the above remark, this program can also be written in a fairly terse and elegant form without using any high-level combinators. However, readability suffers somewhat, and as explained above, we do not get any conceptual benefits: if something is computable in a high-level, perhaps inefficient way, it is computable.



g) Logarithm base 2: repeated division by 2. Since the `while` loop tests for 0 instead of 1, we end up overshooting the result by one, so we decrement R_0 after the loop.



3. Coding programs as numbers

1. *Gödel numbering* is a general technique for assigning a natural number to some mathematical object (such as a well-formed formula in some formal language). The numbering is often computed by translating every symbol of a formula Φ to a natural number, then combining the codes to create a unique Gödel number $G(\Phi)$. For example, with the assignments $t(\text{`}\forall\text{'}) = 1, t(\text{`}x\text{'}) = 2, t(\text{`.'}) = 3$, and $t(\text{`}=\text{'}) = 4$, the Gödel number of the formula $\forall x.\ x = x$ with a particular combination function could be $G(\text{`}\forall x.\ x = x\text{'}) = 272794772250$.

   a) Is the Gödel numbering of register machines described in the notes a bijection, an injection, a surjection, a total function, a partial function, or a relation? Justify your answer.

   > The encoding of register machines is a *bijection*: every machine is associated with a unique natural number, and vice versa. Every encoding (e.g. pairs, lists and instructions) is a bijection by construction.

   b) In the example of first-order logic above, is a particular Gödel numbering a bijection, an

injection, a surjection, a total function, a partial function, or a relation? Justify your answer.

> Every formula can be encoded as a unique number, but not every number will be decoded as a well-formed logical formula – this is because we treat formulae as strings of symbols, rather than expression trees. Thus the encoding is not a bijection, but only an injection.

c) Suggest one or more ways of combining the symbol codes of a formula Φ to generate a *unique* Gödel number for Φ. Demonstrate your methods on the formula $\Phi = \text{`}\forall x.\, x = x\text{'}$ used above.

> (i) We can convert the list of symbols into a list of the individual symbol codes (such as `[1,2,3,2,4,2]`), then use the list encoding from the lecture notes to convert it into a number.
>
> $$G([1,2,3,2,4,2]) = \langle\langle 1, G([2,3,2,4,2])\rangle\rangle = \cdots = 592146$$
>
> (ii) Another common way to ensure a unique encoding is to make use of the Fundamental Theorem of Arithmetic, which states that every number has a unique prime decomposition. Thus, if the encoding is based on a prime decomposition, it can always be recovered from the code. In particular, we can encode a string of symbol codes $c_0\, c_1\, c_2\, c_3\, \ldots$ as the natural number $2^{c_0} \cdot 3^{c_1} \cdot 5^{c_2} \cdot 7^{c_3} \cdot \cdots$. To get back the original formula from a natural, we find its prime factorisation and read off the exponents of the (ordered) primes.
>
> $$G([1,2,3,2,4,2]) = 2^1 \cdot 3^2 \cdot 5^3 \cdot 7^2 \cdot 11^4 \cdot 13^2 = 272794772250$$

2. Let $\varphi_e \in \mathbb{N} \rightharpoonup \mathbb{N}$ denote the unary partial function from numbers to numbers computed by the register machine with code e. Show that for any given register machine computable unary partial function $f \in \mathbb{N} \rightharpoonup \mathbb{N}$, there are infinitely many numbers e such that $\varphi_e = f$. Two partial functions are equal if they are equal as sets of ordered pairs; equivalently, for all numbers $x \in \mathbb{N}$, $\varphi_e(n)$ is defined if and only if $f(x)$ is, and in that case they are equal numbers.

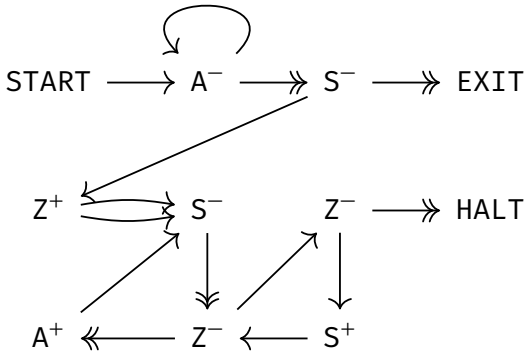
> This question shows that two functions with different codes can have the same behaviour, so while there is a bijection between the machine implementing the function and its code, there are infinitely many register machines implementing the same function. In practice, we can decode $f \in \mathbb{N} \rightharpoonup \mathbb{N}$ as a register machine program, then simply extend it with instructions that never get reached, such as any number of `HALT` instructions. This will change the code of the program without modifying its behaviour.

4. Universal register machine

1. What is the aim of the universal register machine U? How does it work? Annotate the diagram of the register machine with its major components, explaining what they accomplish in the bigger context of the operation of U.

> Th universal register machine U implements a register machine evaluator as a register machine. Its inputs are $R_1 = e$ and $R_2 = a$, where e is the code for a RM program, and a is the code for the list of arguments. The URM first decodes e as a program P, then decodes a as a list of register values $a_1, \ldots, a_n$, then executes the program P on the arguments $R_1 = a_1, \ldots, R_n = a_n$, storing the result in R_0.
>
> The implementation of the machine is similar to a rudimentary processor, keeping track of the currently executed instruction using a program counter and iteratively executing the commands on the arguments. A detailed, annotated analysis of the implementation can be found in the slides for Lecture 4, but it's worth analysing it yourself and getting a high-level grasp of its operation.

2. Consider the list of register machine instructions whose graphical representation is shown below. Assuming that register Z holds 0 initially, describe what happens when the code is executed (both in terms of the effect on registers A and S and whether the code halts by jumping to the label EXIT or HALT).



> This is just a rearranged and renamed version of the "pop" operation described on Slide 47.

Optional exercise

Write a register machine interpreter in a programming language you prefer (a functional language such as ML or Haskell is recommended). Implement a library of RM building blocks such as the ones appearing in the universal register machine or your answer for Ex. 2.1. You may try implementing the RM U as well, but don't worry if you run into resource constraints. The format of input and output is up to you but the RM representation and computation must conform to the theoretical definition.

5. The Halting Problem and undecidability

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

> Let S, T be decidable sets. Intuitively, this means that we can ask the question $n \in^? S$ and get an answer. Closure under union, intersection and complementation simply amounts

to asking the appropriate questions and combining the answers: $n \in^? S \cup T$ if $n \in^? S$ or $n \in^? T$, $n \in^? S \cap T$ if $n \in^? S$ and $n \in^? T$, $n \in^? S^\complement$ if not $n \in^? S$. The logical operations are of course computable: for instance, for deciding $S \cup T$, we run the machine deciding S, halt if the result register is 1, run the machine deciding T otherwise and return its result.

These constructions rely on the fact that the membership test returns an answer, so the reasoning can't be adapted for undecidable languages. The only somewhat obvious result is that the complement of an undecidable language is also undecidable: if it wasn't, we could just negate the answer to decide the original set. However, undecidable sets are *not* closed under union and intersection (and some other set operations): there are undecidable sets which combine to become decidable. There's nothing magical about this: if S is undecidable, then so is $S^\complement$, but $S \cup S^\complement = \mathbb{N}$ and $S \cap S^\complement = \emptyset$, which are of course decidable.

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?

   Such a mapping $r : \mathbb{N} \to \mathbb{N}$ between subsets is called a *reduction* $r : S_1 \leq S_2$: decidability of S_1 can be *reduced* to decidability of S_2. Intuitively, if S_2 is decidable, then we can turn the problem of decidability of S_1 into the decidability of S_2 via the reduction, and due to the assumption on r, the answer of $r(x) \in^? S_2$ gives us the answer for $x \in^? S_1$.

   We can express this formally by considering the characteristic functions of the subsets. The reduction property of r, $\forall n \in \mathbb{N}.\ n \in S_1 \iff r(n) \in S_2$, can be expressed as

   $$\forall n \in \mathbb{N}.\ \chi_{S_1}(n) \iff \chi_{S_2}(r(n))$$

   That is, the functions χ_{S_1} and $\chi_{S_2} \circ r$ are equal. If S_2 is decidable, χ_{S_2} is a computable function; composing it with another computable function r implies that χ_{S_1} is also computable, i.e. S_1 is a decidable subset of $\mathbb{N}$.

   This theorem gives us a useful proof technique: to show that a set S is decidable, we need to find a computable reduction to another set T which we know to be decidable. Such proofs will be common in Complexity Theory next term. In this course, we are more interested in the *contrapositive* of the statement: if S_1 is undecidable and there is a reduction r from S_1 to S_2, then S_2 is also undecidable. Reductions "propagate" undecidability: this is because, as the proof above shows, decidability of S_2 would imply decidability of S_1. Of course, the "de facto" undecidable problem is the Halting Problem; expressed as a set, it's defined as

   $$H \triangleq \{\, \langle e, x \rangle \mid \varphi_e(x)\downarrow \,\}$$

   To show that another set S is undecidable, it is enough to construct a reduction $r : H \leq S$, i.e. a function $r : \mathbb{N} \to \mathbb{N}$ which maps the code of a program e and an argument x to an element $s \in \mathbb{N}$ which is furthermore an element of $S \subseteq \mathbb{N}$ if and only if $\varphi_e(x)$ halts.

> Note that despite the reduction condition being a bi-implication, the reduction itself is still a one-sided function: in particular, neither the converse (if S_1 is decidable then S_2 is decidable) or inverse (if S_2 is undecidable then S_1 is undecidable) of the above statement hold. The theorem might seem a bit backwards, but as you can see from the proof, that is the only way it could work: the reduction translates the *input* to the problem (i.e. the natural $n \in \mathbb{N}$), not the *answer* (i.e. the Boolean $\chi_S(x) \in \mathbb{B}$).

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

> This corollary establishes that equality of programs is undecidable. There are several ways of establishing this: all we need to show is that some other undecidable set would be decidable if we could decide equality of programs. Consider, for instance, the set $S_0 = \{ e \mid \varphi_e(0)\downarrow \}$ from Slide 57: the set of program codes that halt with argument 0. To decide whether a function φ_e halts at 0, we can consider the partial function $c_e(x) = \varphi_e(0)$ which is the constant function with value $\varphi_e(0)$ when $\varphi_e(0)$ is defined, and the constant undefined function otherwise. To choose between the two, we can ask whether c_e is equal to the totally undefined function $\bot$ (where $\bot(x)\uparrow$ for all x) using our machine deciding E. If c_e is equal to the totally undefined function, $\varphi_e(0)$ is undefined so it shouldn't be an element of S_0. If E says that the two are not equal, then $\varphi_e(0)$ must be defined, so $e \in S_0$. Thus, if we could decide E, we could decide S_0, but that is a contradiction.
>
> We can present this reasoning more formally as a reduction proof from S_0 to $E^{\complement} = \{ \langle e, e' \rangle \mid \varphi_e \neq \varphi_{e'} \}$, then using the fact that undecidable languages are closed under complement. The reduction $r : S_0 \leq E$ maps the code e to the code of the pair consisting of the (code of the) function $x \mapsto \varphi_e(0)$, and the totally undefined function $\bot$. By the reasoning above, e will be in S_0 if and only if $x \mapsto \varphi_e(0)$ is not equal to the totally undefined function, i.e. both are in $E^{\complement}$. This implies that $E^{\complement}$ is undecidable, and so is E.

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.

> We are asked to define a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ such that the sets $S_1 \triangleq \{ n \in \mathbb{N} \mid f(n)\downarrow \}$ and $S_2 \triangleq \{ y \in \mathbb{N} \mid \exists n \in \mathbb{N}.\, f(n) = y \}$ are undecidable. Undecidability means that we have two reductions $r_1 : H \leq S_1$ and $r_2 : H \leq S_2$, mapping pairs $\langle e, x \rangle$ to natural numbers such that
>
> $$\varphi_e(x)\downarrow \iff f(r_1(\langle e, x \rangle))\downarrow \ \wedge\ \exists n \in \mathbb{N}.\, f(n) = r_2(\langle e, x \rangle)$$
>
> The first condition suggests that f should preserve halting, while the second one suggests that its return value should have something to do with the pair $\langle e, x \rangle$. Therefore a good first guess is a function whose domain of definition (the set of naturals where it is defined) is the set of pairs $\langle e, x \rangle$ where $\varphi_e(x)\downarrow$, which is precisely H. Given a natural n, f should decode it as a pair $\langle e, x \rangle$, and return a value only if $\varphi_e(x)$ halts. What should be the return value? From the second condition we see that $f(\langle e, x \rangle) = \langle e, x \rangle$, which certainly holds for

the identity function. Hence our guess is the "partial identity function"

$$f(n) \triangleq \begin{cases} n & \text{if } n = \langle e, x \rangle \text{ and } \varphi_e(x)\downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

Using the RM components for decoding n as the pair, and the universal RM to run the computation, we can show that f is computable. However, the set S_1 is equal to $\{ n \in \mathbb{N} \mid n = \langle e, x \rangle \wedge \varphi_e(x)\downarrow \}$, and S_2 is $\{ y \in \mathbb{N} \mid \exists n \in \mathbb{N}.\ n = y \wedge n = \langle e, x \rangle \wedge \varphi_e(x) \}$, both of which are precisely the set $\{ \langle e, x \rangle \mid \varphi_e(x)\downarrow \} = H$. The sets S_1 and S_2 are equal to the set associated with the Halting Problem, and therefore are undecidable.

6. Turing machines

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?

   - A register machine stores data as natural numbers in its registers, while a Turing machine writes symbols on a tape. Since there can only be a finite number of symbols, natural numbers (and other data) have to be encoded explicitly on the tape, using e.g. unary encoding. The state of program execution (program counter) corresponds to the label of the currently executed register machine instruction; in TMs, it is a function of the symbol under the current tape head, and the internal state of the machine.
   - RM programs are a finite list of RM instructions: increment, conditional decrement, and halt. In Turing machines, the program is the transition function $\delta : (Q \times \Sigma) \to (Q \cup \{ \texttt{acc}, \texttt{rej} \}) \times \Sigma \times \{ L, R, S \}$ which assigns a state, symbol, and movement direction to every pairing of the current TM state and symbol under the tape head.
   - The configuration $(\ell, r_0, r_1, \ldots, r_n)$ of a RM is the current instruction label ℓ and the contents of the registers. A computation $c_0, c_1, \ldots$ is a sequence of configurations starting with the initial configuration c_0 (containing the initial register contents and instruction label 0), with each $c_{n+1} determined from $c_n = (\ell, r_0, r_1, \ldots, r_n)$ by executing the instruction at label ℓ on the registers $r_0, \ldots, r_n$. The computation halts if the sequence of configurations is finite (and ends at a HALT instruction), and doesn't halt if the sequence is infinite.

     A TM configuration (q, w, u) consists of the current machine state q, string of symbols w up to and including the tape head, and (finite) string of symbols to the right of the tape head. A computation starts from the initial configuration $(s, \triangleright, u)$ and transitions to new configuration based on the transition function δ. The computation halts of the sequence of configurations is finite (and ends in a `acc` or `rej` state), and doesn't halt if the sequence is infinite.

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

The Chomsky hierarchy is a classification of formal languages of which regular languages and (semi)decidable languages are the two extremes. Each type of language/grammar in the hierarchy is associated with a machine that can recognise membership for a language: for example, regular languages are accepted by finite state automata (recall the last part of the *Discrete Mathematics* course last year), while semidecidable languages are recognised by Turing machines. The precise distinction between recognising and deciding a language (and semidecidable vs. decidable languages) will be discussed in *Complexity Theory*; for now it's worth noting that decidable languages require a machine computing the (total) characteristic function (i.e. they must reject the string explicitly, if it's not a member of a set), while semidecidable languages only need firm acceptance for elements of the set, but can reject or diverge if the element is not in the set.

The full Chomsky hierarchy for demonstration purposes (more detail in the *Formal Models of Language* course next term):

Grammar	Languages	Machine
Type-0	Semidecidable	Turing machine
Type-1	Context-sensitive	Linear-bounded nondeterministic TM
Type-2	Context-free	Nondeterministic pushdown automaton
Type-3	Regular	Finite state automaton

7. Notions of computability

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?

   The Church–Turing thesis states that formal models of computation exactly characterise the nature of effective computation: a function on the natural numbers is effectively computable if and only if it is Turing-computable. This is just a hypothesis, and since there is no formal definition of effective computability, it cannot be formally proven; however, the thesis is universally accepted as identifying the classes of formally and effectively computable functions. In addition, the thesis also states that all (sufficiently strong) models of computation are in fact equivalent: Church and Turing proved this for the independently developed models of Turing machines, partial recursive functions and the λ-calculus, and it has since been reinforced through many new models of computation that are all equivalent to Turing machines. In summary, the Church–Turing thesis gives a formal definition of *computable functions* (functions that are computed by a Turing machine, or any other model of computation), and equates this definition with the informal notion of *effective computability*. A useful

> consequence of this is that it gives us a shortcut to establishing the computability of functions: instead of giving a full, formal specification of an operation as a Turing/register machine program or lambda-expression, we can write an informal, English description of the algorithm, and as long as there are no dubious steps (such as *if the computation halts, do …, else do …*), we have strong reasons to believe that the description corresponds to a computable function.

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.

> The power of a model of computation simply refers to the class of functions that it can compute: if that class is as big as the class of Turing-computable function, the model of computation is as powerful as a Turing machine. To establish Turing-completeness it is sufficient to encode a Turing machine (or any other Turing-complete model of computation) in the system; to compute a computable function, we can simply simulate the associated Turing machine computation.

c) Can our new model be even more powerful?

> The most likely answer is no: so far, every new model of computation was proved to be computationally equivalent to a Turing machine, meaning that both can simulate each other. The Church–Turing thesis implies that any model of computation that can simulate a Turing machine is computationally equivalent to a Turing machine (as a TM can simulate any other model of computation), and all known models of computation support this. However, we do not have a definitive proof of this (because the Church–Turing thesis cannot be formally proved), so in principle there may be a model of computation that is more powerful than a Turing machine and cannot be simulated by one. This is the realm of *hypercomputation* or super-Turing computation, and while there are some theoretical models of hypercomputation (using random oracles or infinite time), there is little hope in discovering a "practical" model that would invalidate the Church–Turing thesis.

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

> There are many examples: abstract rewriting systems, combinatory logic, Kahn process networks, some cellular automata, etc. Sometimes Turing-completeness arises unintentionally in a system not necessarily developed as a model of computation: many games and software have been shown to be Turing-complete by enthusiastic users. Some unsurprising examples are *Minecraft, LittleBigPlanet, Excel, Factorio, Opus Magnum*; some slightly more surprising ones are C++ templates, Java generics, SQL, *Magic: The Gathering*, PowerPoint, and of course the sewage-based 4-bit adder in *Cities: Skylines*. It's a fun internet hole to get lost in.

8. Partial recursive functions

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, $minus : \mathbb{N}^2 \to \mathbb{N}$, where

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

   > The recursive definition of the function is as follows:
   >
   > $$\begin{cases} minus(x, 0) = x \\ minus(x, y + 1) = pred(minus(x, y)) \end{cases}$$
   >
   > We know that $pred$ is primitive recursive, and thus so is $minus$. Explicitly, we have
   >
   > $$minus \triangleq \rho^1\left(\mathtt{proj}_1^1,\ pred \circ \mathtt{proj}_3^3\right) = \rho^1\left(\mathtt{proj}_1^1,\ \rho^0\left(\mathtt{zero}^0,\ \mathtt{proj}_1^2\right) \circ \mathtt{proj}_3^3\right)$$

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

   > The recursive definition is:
   >
   > $$\begin{cases} exp(x, 0) = 1 \\ exp(x, y + 1) = mult(x, exp(x, y)) \end{cases}$$
   >
   > $$\begin{aligned} exp &\triangleq \rho^1\left(\mathtt{succ} \circ \mathtt{zero}^1,\ mult \circ \left[\mathtt{proj}_3^3, \mathtt{proj}_1^3\right]\right) \\ &= \rho^1\left(\mathtt{succ} \circ \mathtt{zero}^1,\ \rho^1\left(\mathtt{zero}^1,\ plus \circ \left[\mathtt{proj}_3^3, \mathtt{proj}_1^3\right]\right) \circ \left[\mathtt{proj}_3^3, \mathtt{proj}_1^3\right]\right) \\ &= \rho^1\Big(\mathtt{succ} \circ \mathtt{zero}^1, \rho^1\left(\mathtt{zero}^1,\ \rho^1\left(\mathtt{proj}_1^1,\ \mathtt{succ} \circ \mathtt{proj}_3^3\right) \circ \left[\mathtt{proj}_3^3, \mathtt{proj}_1^3\right]\right) \\ &\qquad \circ \left[\mathtt{proj}_3^3, \mathtt{proj}_1^3\right]\Big) \end{aligned}$$

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

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

   > The definition is simple, but we need to swap the arguments of the function so the Boolean condition is the last argument (since we only pattern-match on the last argument). Define $C : \mathbb{N}^3 \to \mathbb{N}$ as:
   >
   > $$\begin{cases} C(x_1, x_2, 0) = x_1 \\ C(x_1, x_2, x + 1) = x_2 \end{cases}$$
   >
   > Then $ifzero = \rho^2\left(\mathtt{proj}_1^2,\ \mathtt{proj}_2^4\right) \circ \left[\mathtt{proj}_2^3, \mathtt{proj}_3^3, \mathtt{proj}_1^3\right]$.

   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

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

> The function g can be defined recursively as:
>
> $$\begin{cases} g(\vec{x}, 0) = 0 \\ g(\vec{x}, x+1) = add(g(\vec{x}, x), f(\vec{x}, x)) \end{cases}$$
>
> The explicit definition depends on the number of arguments, but it's easy to see that the function is primitive recursive because both f and add are.

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.

> Given a partial function $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$, the minimisation of f, denoted $\mu^n f(\vec{x})$ is the least argument $x \in \mathbb{N}$ such that $f(\vec{x}, x) = 0$ while $f(\vec{x}, i)$ for all $i = 0, \ldots, x-1$ is defined and strictly greater than 0. This looks like a very specific operation of seemingly limited applicability, however, it is the operator that expands the class of primitive recursive functions to the class of partial recursive, or computable functions. The way to think about minimisation is as *unbounded search*: we're looking for a least value x satisfying some decidable property P. The key is that the functions we try to minimise are not some traditional, "naturally arising" functions, but custom-made ones defined specifically to encapsulate the property P that we want satisfied after minimisation. Quite often, the function f will simply be a step function that switches from 1 to 0 as soon as the property holds:
>
> $$f(\vec{x}, x) \triangleq \begin{cases} 1 & \text{if } \neg P(\vec{x}, x) \\ 0 & \text{if } P(\vec{x}, x) \end{cases}$$
>
> Minimising such a function will give us the least x that satisfies the required property. As an example, we can compute the integer division function $div : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ using minimisation by noticing that the output of $div(x_1, x_2)$ has the unique property that it is the least x such that $x_1 < x_2(x+1)$. Extracting this into a helper function:
>
> $$f(x_1, x_2, x) \triangleq \begin{cases} 1 & \text{if } x_1 \geq x_2(x+1) \\ 0 & \text{if } x_1 < x_2(x+1) \end{cases}$$
>
> This function will switch to 0 exactly when $x = \lfloor x_1 / x_2 \rfloor$, so $div \triangleq \mu^2 f$.
>
> Still, many of the functions that are expressible as a minimisation problem can be defined using primitive recursion. The only definite non-example is Ackermann's function, which can be proved to grow faster than any definable primitive recursive function, but has

> a valid definition using minimisation. This does not mean that the only examples of computable, non-primitive-recursive functions are variations on Ackermann, however. Primitive recursive functions are all total by construction, so any *partial* function is also not primitive recursive – not because of some "limitation" in the power of primitive recursion, but because it can never diverge. Minimisation gives us partiality because there may not be a least element satisfying our property; for instance, in the integer division function above, f never becomes 0 if x_2 is 0, so $\mu^2 f(x_1, 0)$ is undefined, as expected. The Ackermann function is still special, however, as it is one of the simplest *total* computable functions that is not primitive recursive.

3. Use minimisation to show that the following functions are partial recursive:

   a) the binary maximum function $max\colon \mathbb{N}^2 \to \mathbb{N}$.

   > The maximum of two numbers $x_1, x_2 \in \mathbb{N}$ is the least x greater than or equal to both x_1 and x_2 – clearly this will return one of the two numbers. We can encapsulate this with the helper function:
   >
   > $$f(x_1, x_2, x) \triangleq \begin{cases} 1 & \text{if } x < x_1 \text{ or } x < x_2 \\ 0 & \text{if } x \geq x_1 \text{ and } x \geq x_2 \end{cases}$$
   >
   > As before, this switches to 0 when we reached the maximum, so $max \triangleq \mu^2 f$.

   b) the integer square root function $sqrt\colon \mathbb{N} \rightharpoonup \mathbb{N}$ which is only defined if its argument is a perfect square.

   > Unlike before, now we ask for the function to be undefined whenever its argument is not of the right form. The usual helper step-function still works here, with $f(x, y) = 1$ if $x = y^2$ and 0 otherwise. However, we can often define the helper function in a more natural way. Consider $f(x, y) = |x - y^2|$. This will be zero only when $x = y^2$ and positive otherwise; moreover, it will never reach 0 if x is not a perfect square. This is precisely what we were looking for, so $sqrt \triangleq \mu^1 f$.

Optional exercises

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.
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.
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.

9. Lambda calculus

1. Given a set $\mathcal{V} = \{x, y, \ldots\}$ of variables, define the set $\mathcal{T}$ of λ-terms

   a) as an inductively defined set (see IA Formal Languages course).

   > One axiom for variables, and two rules:
   >
   > $$\frac{}{x \in \mathcal{T}}\ (x \in \mathcal{V}) \qquad \frac{M \in \mathcal{T}}{(\lambda x.\, M) \in \mathcal{T}}\ (x \in \mathcal{V}) \qquad \frac{M \in \mathcal{T} \qquad N \in \mathcal{T}}{(MN) \in \mathcal{T}}$$
   >
   > The $\in \mathcal{T}$ is often omitted.

   b) using Backus–Naur form (see IB Semantics course).

   > BNF often makes the set where variables come from, and set of terms being constructed implicit.
   >
   > $$M, N ::= x \mid (\lambda x.\, M) \mid (MN)$$

   c) using a recursive set comprehension (see Lecture 7 of the IB Logic course).

   > Hierarchical construction:
   >
   > $$\mathcal{T}_0 \triangleq \mathcal{V}, \quad \mathcal{T}_{k+1} \triangleq \{(\lambda x.\, M) \mid x \in \mathcal{V} \wedge M \in \mathcal{T}_k\} \cup \{(MN) \mid M, N \in \mathcal{T}_k\}, \quad \mathcal{T} \triangleq \bigcup_{k \in \mathbb{N}} \mathcal{T}_k$$
   >
   > Recursive set comprehension:
   >
   > $$\mathcal{T} \triangleq \mathcal{V} \cup \{(\lambda x.\, M) \mid x \in \mathcal{V} \wedge M \in \mathcal{T}\} \cup \{(MN) \mid M, N \in \mathcal{T}\}$$

2. a) Simplify the following λ-terms (as much as possible, but without evaluating them) using the notational conventions described on Slide 105:

   $$(\lambda x.\, ((ux)y)) \qquad (((\lambda u.\, (\lambda v.\, (vu)u))z)y) \qquad ((((\lambda x.\, (\lambda y.\, (\lambda z.\, ((xz)(yz)))))u)v)w)$$

   > Application is left-associative, dot after binding extends as far to the right as possible.
   >
   > $$\lambda x.\, uxy \qquad (\lambda uv.\, vuu)zy \qquad (\lambda xyz.\, xz(yz))uvw$$

b) Expand the following simplified λ-terms (as much as possible) using the notational conventions described on Slide 105, inserting all parentheses and λ's:

$$xyz(yx) \qquad \lambda u.\, u(\lambda x.\, y) \qquad \lambda xy.\, ux(yz)(\lambda v.\, vy)$$

> Note that the syntax includes parentheses around every compound lambda term, so the whole expression should be wrapped in parentheses.
>
> $$(((xy)z)(yx)) \qquad (\lambda u.\, (u(\lambda x.\, y))) \qquad (\lambda x.\, (\lambda y.\, (((ux)(yz))(\lambda v.\, (vy)))))$$

3. Give a recursive definition of the function $len(M)$ denoting the *length* of the λ-term M given by the total number of variables in M. For example, $len(x(\lambda y.\, yux)) = 5$.

> Straightforward recursive function that pattern-matches on the shape of the argument.
>
> $$\begin{aligned} len(x) &\triangleq 1 \\ len(\lambda x.\, M) &\triangleq 1 + len(M) \\ len(MN) &\triangleq len(M) + len(N) \end{aligned}$$

4. a) Define the *subterm relation* $M \sqsubseteq N$ by recursion on N. For example,

$$x \sqsubseteq \lambda y.\, ux \qquad \lambda x.\, y \sqsubseteq \lambda x.\, y \qquad xy \sqsubseteq (\lambda x.\, xy)z \qquad z \sqsubseteq x(\lambda z.\, y)$$

but $uv \not\sqsubseteq \lambda x.\, xu(vy)$.

> By recursion on N, we have three cases:
>
> - $M \sqsubseteq x$ if $M = x$ (and x is a variable)
> - $M \sqsubseteq \lambda x.\, N$ if $M = x$ or $M \sqsubseteq N$
> - $M \sqsubseteq NN'$ if $M \sqsubseteq N$ or $M \sqsubseteq N'$ The relation is also reflexive, so we have $M \sqsubseteq M$ for all M.

b) We say there is *an occurrence of* M *in* N if $M \sqsubseteq N$.

(i) Mark all occurrences of xy in $(xy)(\lambda x.\, xy)$.
(ii) Mark all occurrences of x in $(xy)(\lambda x.\, xy)$.
(iii) Mark all occurrences of xy in $\lambda xy.\, xy$.
(iv) Mark all occurrences of uv in $x(uv)(\lambda u.\, v(uv))uv$.
(v) Does $\lambda u.\, u$ occur in $\lambda u.\, uv$?

> 1) $(\underline{xy})(\lambda x.\, \underline{xy})$
> 2) $(\underline{x}y)(\lambda \underline{x}.\, \underline{x}y)$
> 3) Since $\lambda xy.\, xy = \lambda x.\, \lambda y.\, xy$, we only have one occurrence: $\lambda xy.\, \underline{xy}$.
> 4) If we add one omitted pair of parentheses, we get $(x(uv)(\lambda u.\, v(uv))u)v$, so the last "$uv$" is not actually an occurrence: $x(\underline{uv})(\lambda u.\, v(\underline{uv}))uv$.
> 5) No, since $\lambda u.\, uv = \lambda u.\, (uv)$, not $(\lambda u.\, u)v$.

5. Let M be the λ-term $\lambda xy.\, x(\lambda z.\, zu)y$.

a) What is the β-normal form of the term $N = M(\lambda vw.\ v(wb))(\lambda xy.\ yaz)$?

> We calculate the normal form by successive β reductions and substitutions. We may start by α-renaming the second argument term, but performing both substitutions at once in the first step, any name clashes are avoided.
>
> $$\begin{aligned} M(\lambda vw.\ v(wb))(\lambda xy.\ yaz) &=_\beta (\lambda xy.\ x(\lambda z.\ zu)y)(\lambda vw.\ v(wb))(\lambda xy.\ yaz) \\ &=_\beta (\lambda vw.\ v(wb))(\lambda z.\ zu)(\lambda xy.\ yaz) \\ &=_\beta (\lambda z.\ zu)((\lambda xy.\ yaz)b) \\ &=_\beta ((\lambda xy.\ yaz)b)u \\ &=_\beta (\lambda y.\ yaz)u \\ &=_\beta uaz \end{aligned}$$

b) Apply the simultaneous substitution $\sigma = [x/y, (\lambda xy.\ zy)/u]$ to M and N, and find the β-normal form of $N[\sigma]$.

> Substitutions are only performed for *free* variables: even if the names match, bound variables are unaffected by substitutions and can always be arbitrarily renamed. To avoid the capture of z, we rename the bound z to w in M first.
>
> $$M[\sigma] = (\lambda xy.\ x(\lambda w.\ w\underline{u})y)[x/y, (\lambda xy.\ zy)/u] = \lambda xy.\ x(\lambda w.\ w\underline{(\lambda xy.\ zy)})y$$
>
> Since substitution distributes over application, we can apply it to the two arguments of N first – however, neither of them contain free occurrences of y or u, so the substitution is a no-op. Hence:
>
> $$\begin{aligned} N[\sigma] &= M[\sigma](\lambda vw.\ v(wb))(\lambda xy.\ yaz) \\ &= (\lambda xy.\ x(\lambda w.\ w(\lambda xy.\ zy))y)(\lambda vw.\ v(wb))(\lambda xy.\ yaz) \end{aligned}$$
>
> To find the normal form of $N[\sigma]$, we could evaluate the above expression. However, we can once again make use of the fact that substitution distributes over application, so the normal form of $N[\sigma]$ will be the normal form of N with σ applied to it.
>
> $$N[\sigma] =_\beta (uaz)[x/y, (\lambda xy.\ zy)/u] = (\lambda xy.\ zy)az = zz$$

c) Give 2 terms α-equivalent to M. Give 2 other terms β-equivalent to M.

> The α-equivalent terms will have the same structure, but renamed (bound) variables.
>
> $$\lambda fa.\ f(\lambda z.\ zu)a \qquad \lambda xy.\ x(\lambda ☺.\ ☺u)y$$
>
> The β-equivalent terms all evaluate to M (since M is in β-nf, there are no shorter β-equivalent terms). Remember that the rules allow for evaluation inside the body of a λ-binding. An infinite number of possibilities here of course – for “yeah okay fair

> enough" points you can recall that $=_\beta$ is reflexive so M is β-equivalent to M.
>
> $$(\lambda x.\, x)M \qquad \lambda xy.\, (\lambda abc.\, bca)(\lambda z.\, zu)yx$$

h. We define η-equivalence as: $M =_\eta \lambda x.\, Mx$ for any λ-term M. Give a shorter and a longer term η-equivalent to M. What is the use of η-equivalence in functional programming?

> A longer term can simply surround M with an application and a binding: $\lambda w.\, (\lambda xy.\, x(\lambda z.\, zu)y)w$. For a shorter term we notice that $M = \lambda x.\, \lambda y.\, x(\lambda z.\, zu)y$, and the body of the outermost binding has the right shape to be η-reduced: $\lambda y.\, x(\lambda z.\, zu)y =_\eta x(\lambda z.\, zu)$. Hence $M =_\eta \lambda x.\, x(\lambda z.\, zu)$. This is an example of a term which is in β-normal form, but not in η-normal form, but $\lambda x.\, x(\lambda z.\, zu)$ cannot be reduced any further with either method, so it is in $\beta\eta$-normal form.
>
> In functional programming η-reduction corresponds to *pointfree programming*: a style of writing functions without mentioning (all of the) arguments, and instead concentrating on how the functions can be combined and composed. For instance, we can write a function to get the last element of a list as the head of its reverse:
>
> ```
> let last xs = hd (rev xs)
> ```
>
> However, we can use the `o` operator to compose `hd` and `rev`:
>
> ```
> let last xs = (hd o rev) xs
> ```
>
> We can rewrite this as a value equalling an anonymous function
>
> ```
> let last = fun xs -> (hd o rev) xs
> ```
>
> But the anonymous function is precisely of the form that admits η-reduction: it is a function that takes an argument and applies another function to the argument. Extensionally, the two functions are equal, so we can also write
>
> ```
> let last = hd o rev
> ```

6. What are some differences between the lambda calculus as defined in this course, and the functional subset of L2 from the IB Semantics course?

> The basic syntactic constructs are the same: we have terms e made up of variables x, functional bindings $\mathbf{fn}\ x : T \Rightarrow e$ and application $e_1\ e_2$. The main difference is that L2 is *typed* while the (untyped) λ-calculus presented in this course is not. This has many consequences, most notably a severe restriction on what kinds of terms we may construct: while the grammar of λ-terms is unrestricted (so e.g. xx is a perfectly reasonable term), in a simply typed setting every subterm is required to have a unique type (so self-application is not possible). Another difference is the evaluation strategy: in L2 we use call-by-value

> (reduce argument first, then substitute). In UTLC, the β-reduction is nondeterministic (albeit confluent), and also allows reduction in the body of a lambda term. We can restrict it to normal-order reduction (left-most, outer-most), which corresponds to call-by-name evaluation: application is performed without evaluating the argument.

10. Lambda-definable functions

1. Give a complete proof of the correctness of Church addition from Slide 119. *Hint:* a formal justification of one of the steps will require mathematical induction.

$$\mathbf{Plus}\,\underline{m}\,\underline{n} =_\beta \underline{m+n}$$

> The full calculation is given in the notes, the only step that requires a more formal proof is $f^m(f^n x) = f^{m+n}$ – we remedy that now.
>
> We prove that for all $m, n \in \mathbb{N}$ and λ-terms F and M, we have that $F^m(F^n M) = F^{m+n}M$ by mathematical induction on m.
>
> **Base case**: $m = 0$. We have $F^0(F^n M) = F^n M = F^{0+n}M$, as required.
>
> **Inductive step**: $m = k+1$. Assume that the IH holds for $m = k$: $F^k(F^n M) = F^{k+n}M$. Then we have: $F^{k+1}(F^n M) = F(F^k(F^n M)) = F(F^{k+n}M) = F^{(k+1)+n}M$, where we applied the IH in the second equality.

2. Define the λ-terms **Times** and **Exp** representing multiplication and exponentiation of Church numerals respectively. Prove the correctness of your definitions.

$$\mathbf{Times}\,\underline{m}\,\underline{n} =_\beta \underline{m \times n} \qquad \mathbf{Exp}\,\underline{m}\,\underline{n} =_\beta \underline{m^n}$$

> Addition applies the successor function n then m times. Multiplication applies the "apply the successor function n times" function m times.
>
> $$\mathbf{Times} \triangleq \lambda mn.\ \lambda f\, x.\ m(nf)x \qquad (=_\eta \lambda mn.\ \lambda f.\ m(nf))$$
>
> We can calculate (noting the difference between definitional equality and β-equivalence):
>
> $$\begin{aligned}\mathbf{Times}\,\underline{m}\,\underline{n} &=_\beta \lambda f\, x.\ \underline{m}(\underline{n}f)x \\ &=_\beta \lambda f\, x.\ (f^n)^m x \\ &= \lambda f\, x.\ f^{n\times m}\, x \\ &= \underline{n \times m}\end{aligned}$$
>
> The property we use in the third step is $(F^n)^m M = F^{n\times m}M$ for all terms F, M and naturals m, n. We prove it by induction on m:
>
> **Base case**: $m = 0$. We have $(F^n)^0 M = M = F^{n\times 0}$, as required.

Inductive step: $m = k+1$. Assume that the IH holds for $m = k$: $(F^n)^k M = F^{n\times k} M$. Then:

$$(F^n)^{k+1} M = F^n((F^n)^k M) =_{\mathsf{IH}} F^n(F^{n\times k} M) = F^{n\times k+n} M = F^{n\times(k+1)} M$$

where we made use of the addition property above in the third step.

Exponentiation of Church numerals is remarkably simple, as their definition automatically performs the "exponentiation":

$$\mathbf{Exp} \triangleq \lambda mn.\, \lambda f\, x.\, n\, m f\, x \qquad (=_\eta \lambda mn.\, n\, m)$$

We can calculate:

$$\begin{aligned}
\mathbf{Exp}\, \underline{m}\, \underline{n} &=_\beta \lambda f\, x.\, \underline{n}\, \underline{m}\, f\, x \\
&=_\beta \lambda f\, x.\, (\underline{m})^n\, f\, x \\
&= \lambda f\, x.\, (\underline{m^n})\, f\, x \\
&= \lambda f\, x.\, f^{m^n}\, x \\
&= \underline{m^n}
\end{aligned}$$

The property we use in the third step is $(\underline{m})^n\, M = \underline{m^n}\, M$ for all terms M and $m, n \in \mathbb{N}$. We proceed by induction on n:

Base case: $n = 0$. We have $(\underline{m})^0\, M = M = \underline{1}\, M = \underline{m^0}\, M$, as required.

Inductive step: $n = k+1$. Assume that the IH holds for $n = k$: $(\underline{m})^k\, M = \underline{m^k}\, M$. Then:

$$(\underline{m})^{k+1} M = \underline{m}\,((\underline{m})^k M) =_{\mathsf{IH}} \underline{m}\,((\underline{m^k})\, M) = \mathbf{Times}\, \underline{m}\, \underline{m^k}\, M = \underline{m \times m^k}\, M = \underline{m^{k+1}}\, M$$

3. Show that the λ-term $\mathbf{Ack} \triangleq \lambda x.\, x\, T\, \mathbf{Succ}$, where $T \triangleq (\lambda f\, y.\, y f\, (f\, \underline{1}))$ represents Ackermann's function $ack \in \mathbb{N}^2 \to \mathbb{N}$. *Hint*: you will need to use nested induction; consider deriving a simplified form for the outer inductive case before starting the nested proof.

For brevity I will write A for $\mathbf{Ack}$ and S for $\mathbf{Succ}$. We need to show that for all naturals $m, n \in \mathbb{N}$, $A\, \underline{m}\, \underline{n} =_\beta \underline{ack(m,n)}$ by induction on m.

Base case: $m = 0$.

$$A\, \underline{0}\, \underline{n} = \underline{0}\, T\, S\, \underline{n} = T^0\, S\, \underline{n} = S\, \underline{n} = \underline{n+1} = \underline{ack(0,n)}$$

Inductive step. Assume

$$\forall n \in \mathbb{N}.\, A\, \underline{m}\, \underline{n} =_\beta \underline{ack(m,n)} \tag{IH1}$$

and prove that for all $n \in \mathbb{N}$, $A\, \underline{m+1}\, \underline{n} =_\beta \underline{ack(m+1,n)}$. Before attempting this proof, it's worth showing some simple lemmas. First, we calculate the partial application:

$$A\, \underline{m+1} =_\beta \underline{m+1}\, T\, S =_\beta T\, (T^m\, S) =_\beta T\, (\underline{m}\, T\, S) =_\beta T\, (A\, \underline{m}) \tag{†}$$

This gives us

$$\begin{aligned} A\underline{m+1}\,\underline{n} &=_\beta T\,(A\underline{m})\,\underline{n} && \text{(by †)}\\ &= (\lambda f\,y.\,yf(f\,\underline{1}))\,(A\underline{m})\,\underline{n}\\ &=_\beta \underline{n}\,(A\underline{m})\,(A\underline{m}\,\underline{1})\\ &=_\beta \underline{n}\,(A\underline{m})\,\underline{ack(m,1)} && \text{(by IH1)} \end{aligned}$$

Thus we have that $A\underline{m+1}\,\underline{n} =_\beta \underline{n}\,(A\underline{m})\,\underline{ack(m,1)}$ and can proceed with a proof of

$$\forall n \in \mathbb{N}.\, A\underline{m+1}\,\underline{n} =_\beta \underline{ack(m+1,n)}$$

by (nested) induction on $n \in \mathbb{N}$.

Base case: $n = 0$. We have

$$A\underline{m+1}\,\underline{0} =_\beta \underline{0}\,(A\underline{m})\,\underline{ack(m,1)} =_\beta \underline{ack(m,1)}$$

and $ack(m+1,0) \triangleq ack(m,1)$ by definition.

Inductive step. Assume

$$A\underline{m+1}\,\underline{n} =_\beta \underline{ack(m+1,n)} \qquad \text{(IH2)}$$

We need to prove $A\underline{m+1}\,\underline{n+1} =_\beta \underline{ack(m+1,n+1)}$:

$$\begin{aligned} A\underline{m+1}\,\underline{n+1} &=_\beta \underline{n+1}\,(A\underline{m})\,\underline{ack(m,1)} && \text{(by †)}\\ &=_\beta (A\underline{m})^{n+1}\,\underline{ack(m,1)} && \text{(def. of Church numerals)}\\ &= A\underline{m}\,(\underline{n}\,(A\underline{m})\,\underline{ack(m,1)}) && \text{(by def. of exponentiation)}\\ &=_\beta A\underline{m}\,(A\underline{m+1}\,\underline{n}) && \text{(by †)}\\ &=_\beta A\underline{m}\,\underline{ack(m+1,n)} && \text{(by IH2)}\\ &=_\beta \underline{ack(m,ack(m+1,n))} && \text{(by IH1 with } n = ack(m+1,n))\\ &=_\beta \underline{ack(m+1,n+1)} && \text{(by def. of Ackermann)} \end{aligned}$$

All cases are covered, so the proof is finished. And what a proof it was.

4. Consider the following λ-terms:

$$\mathbf{I} \triangleq \lambda x.\, x \qquad \mathbf{B} \triangleq \lambda g f x.\, f\,x\,\mathbf{I}\,(g(f\,x))$$

a) Show that $\underline{n}\,\mathbf{I} =_\beta \mathbf{I}$ for every $n \in \mathbb{N}$.

Unsurprisingly, we prove this by induction on $n \in \mathbb{N}$. We also introduce an arbitrary λ-term M as the argument.

Base case: $n = 0$. We have $\underline{0}\,\mathbf{I}\,M =_\beta \mathbf{I}^0\,M = M =_\beta \mathbf{I}\,M$, as required.

Inductive step: $n = k+1$. Assume the IH holds for $n = k$: $\underline{k}\,\mathbf{I}\,M = \mathbf{I}\,M$. Now

$$\underline{k+1}\,\mathbf{I}\,M =_\beta \mathbf{I}\,(\underline{k}\,\mathbf{I}\,M) =_\beta \mathbf{I}\,(\mathbf{I}\,M) =_\beta \mathbf{I}\,M$$

b) Assuming the fact about normal order reduction mentioned on Slide 115, show that if partial functions $f, g : \mathbb{N} \rightharpoonup \mathbb{N}$ are represented by closed λ-terms F and G respectively, then their composition $(g \circ f)(x) \triangleq g(f(x))$ is represented by $\mathbf{B}\,G\,F$. Explain how this avoids the discrepancy with partial functions mentioned in Slide 125.

> Let F and G be closed λ-terms representing partial functions $f, g : \mathbb{N} \rightharpoonup \mathbb{N}$. We need to show that for all $n \in \mathbb{N}$ where $g(f(n))$ is defined, $\mathbf{B}\,G\,F\,\underline{n} =_\beta \underline{g(f(n))}$. If $g(f(n)) = m$ for some $m \in \mathbb{N}$, we must have a $k \in \mathbb{N}$ such that $f(n) = k$ and $g(k) = m$, so $F\,\underline{n} =_\beta \underline{k}$ and $G\,\underline{k} =_\beta \underline{m}$ by assumption. Then, using $\underline{k}\,\mathbf{I} =_\beta \mathbf{I}$ from (a):
>
> $$\mathbf{B}\,G\,F\,\underline{n} =_\beta F\,\underline{n}\,\mathbf{I}\,(G\,(F\,\underline{n})) =_\beta \underline{k}\,\mathbf{I}\,(G\,\underline{k}) =_\beta \mathbf{I}\,(G\,\underline{k}) =_\beta G\,\underline{k} =_\beta \underline{m}$$
>
> Now, if k exists but $g(k)$ is undefined, the last step will diverge. Since normal-order reduction will eventually try evaluating $G\,\underline{k}$ in $F\,\underline{n}\,\mathbf{I}\,(G\,\underline{k})$, the evaluation of $\mathbf{B}\,G\,F\,\underline{n}$ will also diverge. On the other hand, if $f(n)$ is undefined, then $F\,\underline{n}$ will diverge; since this is the outermost subterm of $F\,\underline{n}\,\mathbf{I}\,(G\,(F\,\underline{n}))$, normal-order reduction will also diverge. This differs from simply defining the composition as $G\,(F\,\underline{n})$, since normal-order reduction may not reach the evaluation of $(F\,\underline{n})$, e.g. if G is a constant function. The $\mathbf{B}$ combinator forces the evaluation of $F\,\underline{n}$ even if it wouldn't get encountered otherwise, and therefore it corresponds to composition of partial functions.

5. In the following questions you may use all of the λ-definable functions presented in the notes, as well as the terms you define as part of this exercise. You should explain your answers (possibly using some examples), but don't need to prove their correctness.

a) Give a λ-term **Not** representing Boolean negation.

> Since a Church Boolean corresponds to a branching operation, all we need to do is return false in the true branch, and true in the false branch:
>
> $$\mathbf{Not} \triangleq \lambda p.\ p\ \mathbf{False}\ \mathbf{True}$$

b) Give λ-terms **And** and **Or** representing Boolean conjunction and disjunction.

> Again, we use the standard definition of conjunction and disjunction in terms of Boolean branching, also exploiting the fact that Booleans are encoded as their own eliminators. For example, in $p \wedge q$, if p is true then we return q (and the truth value of q will determine the truth value of $\top \wedge q$), otherwise we return false. But, since p already denotes false (in its own 'else' branch), we just need to return p itself.
>
> $$\mathbf{And} \triangleq \lambda pq.\ p\,q\,p$$
> $$\mathbf{Or} \triangleq \lambda pq.\ p\,p\,q$$

c) Give a λ-term **Minus** representing truncated subtraction (i.e. $\mathbf{Minus}\,\underline{m}\,\underline{n} = 0$ if $m < n$).

> Subtraction is repeated application of the predecessor function. Since $\mathbf{Pred}\,\underline{0} = \underline{0}$, it

> already performs the truncation.
>
> $$\mathbf{Minus} \triangleq \lambda mn.\ n\ \mathbf{Pred}\ m$$

d) Give λ-terms **Eq, NEq, LT, LEq, GT, GEq,** representing the numeric comparison operations $=, \neq, <, \leq, >, \geq$ respectively. You can define them in any order you find most convenient.

> Most of these relations can be defined in terms of each other, so it's possible to define one as a starting point, then construct the rest from it. A good starting point is $\leq$ or $\geq$ which give you both the notion of ordering and equality. These are also simple to define in terms of truncated subtraction and zero check: $m \leq n$ iff $m \dot{-} n = 0$.
>
> $$\mathbf{LEq} \triangleq \lambda mn.\ \mathbf{Eq}_0\ (\mathbf{Minus}\ m\ n)$$
>
> From here, the remaining operations can be defined using Boolean operators. Here is one possible ordering, several others are possible:
>
> $$\begin{aligned}
> \mathbf{Eq} &\triangleq \lambda mn.\ \mathbf{And}\ (\mathbf{LEq}\ m\ n)\ (\mathbf{LEq}\ n\ m) \\
> \mathbf{NEq} &\triangleq \lambda mn.\ \mathbf{Not}\ (\mathbf{Eq}\ m\ n) \\
> \mathbf{GT} &\triangleq \lambda mn.\ \mathbf{Not}\ (\mathbf{LEq}\ m\ n) \\
> \mathbf{GEq} &\triangleq \lambda mn.\ \mathbf{Or}\ (\mathbf{GT}\ m\ n)\ (\mathbf{Eq}\ m\ n) \\
> \mathbf{LT} &\triangleq \lambda mn.\ \mathbf{Not}\ (\mathbf{GEq}\ m\ n)
> \end{aligned}$$

e) Define the λ-term **UCr** that represents the uncurrying higher-order function: if $\langle M, N\rangle$ denotes $\mathbf{Pair}\, M\, N$, then $\mathbf{UCr}\ F\ \langle M, N\rangle =_\beta F\ M\ N$.

> Nothing surprising here – use projections to get the two elements of the pair.
>
> $$\mathbf{UCr} \triangleq \lambda f\, p.\ f\, (\mathbf{Fst}\ p)\ (\mathbf{Snd}\ p)$$

f) Give a λ-term **MapPair** that applies a function to both elements of a pair: that is, $\mathbf{MapPair}\, F\, \langle M, N\rangle =_\beta \langle F\ M, F\ N\rangle$.

> Still nothing surprising!
>
> $$\mathbf{MapPair} \triangleq \lambda f\, p.\ \mathbf{Pair}\ (f\, (\mathbf{Fst}\ p))\ (f\, (\mathbf{Snd}\ p))$$

g) Give a λ-term **SqSum** which represents the function $\langle m, n\rangle \mapsto m^2 + n^2$.

> We can combine many of the functions we defined already to give a relatively clean, compositional definition.
>
> $$\mathbf{SqSum} \triangleq \lambda p.\ \mathbf{UCr\ Plus}\ (\mathbf{MapPair}\ (\lambda k.\ \mathbf{Exp}\ k\ \underline{2})\ p)$$
>
> We can even go point-free by using the composition combinator **B** from above:
>
> $$\mathbf{SqSum} \triangleq \mathbf{B}\ (\mathbf{UCr\ Plus})\ (\mathbf{MapPair}\ (\lambda k.\ \mathbf{Exp}\ k\ \underline{2}))$$

6. a) Explain why Curry's **Y** combinator is needed and how it works.

Term definitions in the λ-calculus are simply abbreviations for larger λ-terms, and therefore they cannot be self-referential (otherwise trying to expand the term in its own definition would lead to an infinite expansion). This prevents us from writing "recursive" definitions such as $\mathbf{Fact}\ n \triangleq n \times (\mathbf{Fact}\ (n-1))$: all subterms of a term should be well-defined. However, this doesn't mean that we cannot define recursive functions in the λ-calculus. We first note that a recursively defined term would have the general form $M \triangleq F\ M$, where the body of the definition of M itself takes the term M as an argument (e.g. $\mathbf{Fact} \triangleq (\lambda f.\ \lambda m.\ m \times f\,(m-1))\ \mathbf{Fact}$[a]). We can instead state this in terms of β-equivalence (as a *specification*, rather than *definition*):

$$M =_\beta F\ M$$

This form characterises M as a *fixed point* of F, where a fixed point of a function f is an argument x such that $f(x) = x$. Fixed points and recursion go hand-in-hand, since $M =_\beta F\ M =_\beta F(F\ M) =_\beta F(F \cdots (F\ M) \cdots)$, giving rise to the repetitive behavior that we are trying to capture. Hence, to define a term M that satisfies the "recursive" specification $M =_\beta F\ M$, we need to find a fixed point of F. The fixed point of the λ-term $(\lambda f\, m.\ m \times f\,(m-1))$ is precisely the factorial function.

The next question is: how do we find the fixed point of a λ-term? We can use *fixed-point combinators*, which are higher-order λ-terms that take a function and return its fixed point. One of the simplest fixed-point combinators is Curry's $\mathbf{Y}$ combinator, which satisfies the required fixed point property (stating that $\mathbf{Y}\,F$ is a FP of F):

$$\mathbf{Y}\,F =_\beta F(\mathbf{Y}\,F)$$

Curry defined $\mathbf{Y}$ as follows:

$$\mathbf{Y} \triangleq \lambda f.\ (\lambda x.\ f\,(x\,x))\ (\lambda x.\ f\,(x\,x))$$

It is crucially dependent on *self-application*, which is possible in the absence of types, but becomes impossible in a typed setting. $\mathbf{Y}$ is therefore slightly "magical" and doesn't correspond obviously to a natural computational construction, but it nevertheless satisfies the fixed point property above (see Slide 135). The $\mathbf{Y}$ combinator now allows us to define recursion indirectly: given the "intended" definition $M \triangleq F\ M$, we can define the λ-term that satisfies this as the fixed point $\mathbf{Y}\,F$ of F. For example,

$$\mathbf{Fact} \triangleq \mathbf{Y}\ (\lambda f\, m.\ m \times f\,(m-1))$$

[a] The base case in the definitions will be omitted for brevity – of course they are important, and are easy to define with $\mathbf{If}$ and $\mathbf{Eq}_0$.

b) Give a λ-term which is β-equivalent to the $\mathbf{Y}$ combinator, but only uses it's f argument once. *Hint*: see if you can exploit the symmetry of the $\mathbf{Y}$ combinator.

> We can notice that the body of $\mathbf{Y} \triangleq \lambda f.\ (\lambda x.\ f(xx))\ (\lambda x.\ f(xx))$ is the same term $(\lambda x.\ f(xx))$ repeated twice. We can exploit this using the "self-application combinator" $\lambda x.\ xx$ to get $\mathbf{Y}'$, which reduces to $\mathbf{Y}$ in one step.
>
> $$\mathbf{Y}' \triangleq \lambda f.\ (\lambda x.\ xx)\ (\lambda x.\ f(xx))$$

c) Define the λ-term **Fact** that computes the factorial of a Church numeral.

> We expand our previous, informal example of fixpoint recursion. The λ-term **Fact** should satisfy the following equivalence:
>
> $$\mathbf{Fact} =_\beta \lambda n.\ \mathbf{If}\ (\mathbf{Eq}_0\ n)\ \underline{1}\ (\mathbf{Times}\ n\ (\mathbf{Fact}\ (\mathbf{Pred}\ n)))$$
>
> To make this into a well-formed term definition, we need to abstract out the recursive call, then "tie the knot" using the $\mathbf{Y}$ combinator.
>
> $$\mathbf{Fact} \triangleq \mathbf{Y}\ \big(\lambda f.\ \lambda n.\ \mathbf{If}\ (\mathbf{Eq}_0\ n)\ \underline{1}\ (\mathbf{Times}\ n\ (f\ (\mathbf{Pred}\ n)))\big)$$

d) Define the λ-term **Fib** such that $\mathbf{Fib}\ \underline{n} =_\beta \underline{F_n}$ where F_n is the n^{th} Fibonacci number defined recursively as $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$.

> Similarly, we start with the recursive specification, then turn it into a definition.
>
> $$\mathbf{Fib} =_\beta \lambda n.\ \mathbf{If}\ (\mathbf{LEq}\ n\ \underline{1})\ n\ (\mathbf{Plus}\ (\mathbf{Fib}\ (\mathbf{Pred}\ n))\ (\mathbf{Fib}\ (\mathbf{Minus}\ n\ \underline{2})))$$
>
> Rewriting with the fixed point combinator, we notice that the recursive call is made twice, which is captured by the repeated use of the higher-order argument f.
>
> $$\mathbf{Fib} \triangleq \mathbf{Y}\ \big(\lambda f.\ \lambda n.\ \mathbf{If}\ (\mathbf{LEq}\ n\ \underline{1})\ n\ (\mathbf{Plus}\ (f\ (\mathbf{Pred}\ n))\ (f\ (\mathbf{Minus}\ n\ \underline{2})))\big)$$