
PART IB

Computation Theory

Supervision Exercises

Lent Term 2027 · Supervisions 1–3

Michael Lee

Standard Libraries	1
A Register Machine Library	1
B Primitive Recursive Library	2
Refresher	3
1 Countability	3
Supervision 1	5
1 Computability	5
2 Algorithmically undecidable problems	5
3 Register machines	6
4 Gödel Encoding	12
5 Universal Register Machine	18
Supervision 2	20
1 The Halting Problem and Decidability	20
2 Turing Machines	22
3 Primitive Recursive Functions	24
4 The Lambda Calculus: Syntax	27
5 The Lambda Calculus: Equational Systems	28
Supervision 3	29
1 The Lambda Calculus: Normal Forms	29
2 Lambda-Definable Functions	32
Refresher — Solutions	37
1 Countability	37

★ A ★ in the margin means the question, or that part of the question, is optional.

Standard Libraries

Lent Term 2027 · Michael Lee

A Register Machine Library

RM

```
1  (* x++ *)
2  machine Inc(x) = struct
3    l0 : x+ -> l1
4    l1 : halt
5  end
6
7  (* x-- *)
8  machine Dec(x) = struct
9    l0 : x- -> l1, l1
10   l1 : halt
11 end
12
13 machine Halt() = struct
14   l0 : halt
15 end
16
17 (* r0 := r1 *)
18 machine Copy(r0, r1) = struct
19   reg t
20   l0 : r0- -> l0, l1
21   l1 : r1- -> l2, l4
22   l2 : r0+ -> l3
23   l3 : t+ -> l1
24   l4 : t- -> l5, l6
25   l5 : r1+ -> l4
26   l6 : halt
27 end
28
29 (* r0 := r1 + r2 *)
30 machine Add(r0, r1, r2) = struct
31   reg t
32   l0 : call Copy(r0, r1) -> l1
33   l1 : r2- -> l2, l4
34   l2 : r0+ -> l3
35   l3 : t+ -> l1
36   l4 : t- -> l5, l6
```

```

37   l5 : r2+ -> l4
38   l6 : halt
39 end
40
41 (* x += y *)
42 machine AddBy(x, y) = struct
43   reg t
44   l0 : call Add(t, x, y) -> l1
45   l1 : call Copy(x, t) -> l2
46   l2 : halt
47 end
48
49 (* r0 := r1 * r2 *)
50 machine Mul(r0, r1, r2) = for r1 do AddBy(r0, r2) done
51
52 (* x *= y *)
53 machine MulBy(x, y) = struct
54   reg t
55   l0 : call Mul(t, x, y) -> l1
56   l1 : call Copy(x, t) -> l2
57   l2 : halt
58 end

```

B Primitive Recursive Library

The library defines $add(x, y) = x + y$, $mult(x, y) = x \times y$, $pred(x) = x \dot{-} 1$, and $triangle(x) = 0 + 1 + \dots + x$.

```

1  let add = rho[1](proj[1, 1], succ . proj[3, 3])
2
3  let mult = rho[1](zero[1], add . (proj[3, 3], proj[3, 1]))
4
5  let pred = rho[0](zero[0], proj[2, 1])
6
7  let triangle = rho[0](zero[0], succ . add . (proj[2, 1], proj[2, 2]))

```

Refresher

Lent Term 2027 · Michael Lee

1 Countability

PREREQUISITES None · TIME 40 min

NOTE

Computation Theory relies heavily on the concepts of *countability* and *diagonalisation*, as taught in Part IA Discrete Mathematics.

This sheet is meant to refresh your knowledge. It is meant to be attempted over the Christmas vacation, and not meant to be submitted. Solutions have been provided.

QUESTION 1

CONCEPTUAL · 40 MIN

● ● ● ● ● BASIC

This question aims to refresh your memory of countability and diagonalisation arguments.

Recall from discrete mathematics the notion of a countable set. There are various definitions of countability, all equivalent.

1. A set S is countable if it is empty, or there exists a surjective function $\mathbb{N}$ to S .
2. A set S is countable if there exists an injective function S to $\mathbb{N}$.

An injective function $f : A \rightarrow B$ is one where $\forall a_1, a_2 \in A. f(a_1) = f(a_2) \implies a_1 = a_2$. A surjective function $g : A \rightarrow B$ is one where $\forall b \in B. \exists a \in A. g(a) = b$.

HINT The characterisation using injections is easier to work with when proving a true statement. The characterisation using surjections is easier to work with when disproving a false statement.

Prove or disprove the following statements.

- (a) Any finite set is countable.
- (b) $\mathbb{N}$, the set of natural numbers, is countable.
- (c) $\mathbb{Z}^+$, the positive integers, is countable.
- (d) $\mathbb{Z}$, the set of integers, is countable.
- (e) $\mathbb{N} \times \mathbb{N}$ is countable.
- (f) If A and B are countable, then $A \times B$ is countable.
- (g) For all natural numbers n , the iterated product $\prod_{i=1}^n A_i$ is countable if each A_i is countable.
- (h) $\mathbb{Q}$, the set of rational numbers, is countable.
- (i) If A and B are countable, then $A + B$ is countable.
- (j) For all natural numbers n , the iterated sum $\sum_{i=1}^n A_i$ is countable if each A_i is countable.

- (k) If A is countable, then the inductively defined set $\text{List } A$ is countable. *A definition of $\text{List } A$ is given in the Foundations of Computer Science exercises.*
 - (l) If A is countable, then $B \subseteq A$ is countable.
 - (m) If A is countable, then the powerset of A , $\mathcal{P}(A)$, is countable.
 - (n) If A and B are countable, then the set of functions $A \rightarrow B$ is countable.
 - (o) $\mathbb{R}$, the set of real numbers, is countable.
-

Supervision 1

Lent Term 2027 · Michael Lee

WORK DUE	A single PDF, 48 hours before the supervision
EXPECTED TIME	6 hours 20 min (+ 1 hour 25 min for the optional questions)
OPTIONAL	★ marks a question, or a part of one, you may skip

1 Computability

PREREQUISITES None · TIME 20 min

QUESTION 1

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

This course focuses on three different models of computation: register machines, partial recursive functions, and the untyped lambda calculus.

With any model, of any concept, it's important to think about what falls inside the model, and what falls outside the model.

Write the simplest OCaml program that exhibits a form of computation that is not captured by the models in this course.

HINT It's possible to do it in one line.

2 Algorithmically undecidable problems

PREREQUISITES Lecture 1 · TIME 10 min

QUESTION 2

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

The halting problem is an example of a function that is not computable. This question is meant to illustrate some other examples.

HINT Show that a solution to each of these implies a solution to the halting problem.

- (a) A compiler technique for equating functions based on extensional equality, regardless of the language. Two partial functions f and g are extensionally equal when

$$\forall x, y. f(x) \Downarrow y \iff g(x) \Downarrow y$$

It would be very useful to compare for extensional equality, since then slow functions could be replaced with much faster ones.

Explain why there is no function that takes f and g and returns true if they are extensionally equal, and false otherwise.

- (b) A compiler technique for performing complete dead code elimination, for any arbitrary language. Sometimes, some code cannot ever be reached during execution. For example,

```

1  def g():
2      return 3
3      h() # this code is dead

```

Python

Dead code elimination is an optimisation that eliminates dead code only if it is dead: that is, all code that is eliminated must be dead, but some dead code might not be eliminated. A compiler performs complete dead code elimination if it eliminates code if, and only if, the code is dead. Explain why such a compiler technique cannot exist.

- (c) Suggest a specific language in which functions can be equated extensionally, and complete dead code elimination is achievable.

3 Register machines

PREREQUISITES Lecture 2 · TIME 1 hour 40 min (+ 1 hour 20 min)

★ QUESTION 3

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

Define a syntax and small step operational semantics, as in Part IB Semantics, for register machines. Identify the halting configurations.

QUESTION 4

CONCEPTUAL · 30 MIN

● ● ● ● ● CORE

Explain how, given register machines M , M_1 , M_2 , we can build new register machines with the following semantics (these are known as register machine *combinators*):

1. Sequencing: do M_1 , then M_2
2. Conditional: if register R is not zero, do M_1 , otherwise, do M_2
3. While: while register R is non-zero, do M
4. For: do M R times, counting R down to zero

QUESTION 5

CONCEPTUAL · 10 MIN (+ 1 HOUR)

● ● ● ● ● CORE

I have built a register machine language in which you can write programs. You can install the compiler for this language by going to [this link](#), and following the instructions in the `README.md`.

The language lets you write, for example, the multiplication example from the notes.

RM

```

1  reg r0, r1 := 3, r2 := 4, r3
2
3  10 : r1- -> 11, 16
4  11 : r2- -> 12, 14
5  12 : r0+ -> 13
6  13 : r3+ -> 11
7  14 : r3- -> 15, 10
8  15 : r2+ -> 14
9  16 : halt

```

This code is kind of hard to read, or reason about. We shall now work towards extending this language with exactly the higher-level combinators described in the previous question.

First, we will extend the language so that we can define register machines, name them, and then call them within the body of another register machine.

RM

```

1  machine AddBy(x, y) = struct
2    reg t
3    10 : y- -> 11, 13
4    11 : x+ -> 12
5    12 : t+ -> 10
6    13 : t- -> 14, 15
7    14 : y+ -> 13
8    15 : halt
9  end
10
11 reg r0, r1 := 3, r2 := 4
12
13 10 : r1- -> 11, 12
14 11 : call AddBy(r0, r2) -> 10
15 12 : halt

```

Then, we will add register machine combinators, like sequencing, if, while, and for. With these combinators, we could write the multiplication program even more cleanly, like so:

RM

```

1  machine AddBy(x, y) = struct
2    reg t
3    10 : y- -> 11, 13
4    11 : x+ -> 12
5    12 : t+ -> 10
6    13 : t- -> 14, 15
7    14 : y+ -> 13
8    15 : halt
9  end
10
11 machine Mul(z, x, y) =
12   for x do AddBy(z, y) done

```

```

13
14 reg r0, r1 := 3, r2 := 4
15
16 10 : call Mul(r0, r1, r2) -> 11
17 11 : halt

```

(a) Consider extending the definition of a register machine with two additional instructions:

- (i) $\ell : \text{JUMP } \ell'$, which performs an unconditional jump to an instruction ℓ'
- (ii) $\ell : \text{CLEAR } r \rightarrow \ell'$, which zeros out a register, and then moves to ℓ'

Prove that these instructions add no extra expressiveness, that is, they can be expressed in terms of r^+ , r^- , and HALT.

HINT You can always add a new register

- ★ (b) Consider allowing register machines to call other register machines. To enable this, we first need to invent some new syntax for defining (and naming) register machines. *Note: we will only support non-recursive definitions, so machines will not be allowed to call themselves in their definition.*

```

1 machine Copy(d, s) = struct
2   reg t
3   10 : d- -> 10, 11
4   11 : s- -> 12, 14
5   12 : d+ -> 13
6   13 : t+ -> 11
7   14 : t- -> 15, 16
8   15 : s+ -> 14
9   16 : halt
10 end

```

which defines a register machine Copy, which takes registers d, s as arguments, and also defines internal register t.

We can call this machine using the syntax

```
11 : call Copy(r3, r2) -> 12
```

The callee, Copy, receives its arguments in registers d and s. It is allowed to modify their values: the caller should not expect the values of the registers to be retained. By convention, Copy treats d as an output register: when it halts, it updates the value in d with the return value.

We will assume that all argument registers are different. The implementation checks for this property, and raises a compile-time error if the same register is passed twice as an argument.

Copy is allowed to define internal registers, but these registers must always be initialised to zero. We cannot write `reg t := 3`. Copy must be self contained: it cannot refer to registers defined outside its arguments and internal registers, or labels ℓ belonging to other machines.

- (i) Identify the *calling convention*, used by these register machines.
- (ii) Explain, informally, a procedure for eliminating the calls to defined machines, into the

extended register machine language with JUMPs and CLEARs. That is, you should seek to produce a program that has a list of registers, and a list of instructions, with no internal definitions.

- (iii) Consider a machine, `ConstOne`, that returns a constant one, by adding one to an internal register, R , zeroing out R_0 , and then copying R to R_0 .

```

1  machine ConstOne(r0) = struct
2      reg r
3      l0 : r+ -> l1
4      l1 : r0- -> l1, l2
5      l2 : call Copy(r0, r) -> l3
6      l3 : halt
7  end

```

Explain what your procedure returns when the machine is called twice, as such:

```

1  reg r0, r1 := 2
2
3  l0 : r1- -> l1, l2
4  l1 : call ConstOne(r0) -> l0
5  l2 : halt

```

How do you ensure that the internal register r is set to zero the second time the machine is called?

- ★ (c) We now wish to allow nested machine definitions, as in

```

1  machine Mul(z, x, y) = struct
2      machine AddBy(x, y) = struct
3          reg t
4          l0 : y- -> l1, l3
5          l1 : x+ -> l2
6          l2 : t+ -> l0
7          l3 : t- -> l4, l5
8          l4 : y+ -> l3
9          l5 : halt
10         end
11
12     l0 : x- -> l1, l2
13     l1 : call AddBy(z, y) -> l0
14     l2 : halt
15 end

```

Machine labels and registers are fully self contained. That is, in the earlier example, `AddBy` cannot refer to `Mul`'s z , x , y or $l0$, $l1$, $l2$, even though it is nested within `Mul`. Describe, informally, a procedure for unnesting nested definitions. At the end of your procedure, all machine definitions should be leaves: they contain no nested definitions within them.

- ★ (d) Finally, we wish to define machines using combinators `if`, `while`, `for`, and `;`. For example,

```

1  (* Multiplication using a for loop *)
2  machine Mul(z, x, y) =
3      for x do AddBy(z, y) done
4
5  (* Multiplication using a while loop *)
6  machine Mul2(z, x, y) =
7      while x do Dec(x); AddBy(z, y) done
8
9  (* Returns 1 if b > 0, else returns 0 *)
10 machine Sign(a, b) =
11     if b then Inc(a) else Halt() end

```

Note that *Inc*, *Dec*, and *AddBy* are defined in [Appendix A](#).

Explain how each of these four combinators can be expressed using (nested) machine definitions.

Note that we might not want to implement a for loop directly as specified, for *R* do *M*. Can you see why?

QUESTION 6

PRACTICAL · 1 HOUR

● ● ● ● ● CORE

Show that the following arithmetic functions are all register machine computable. You may submit either a hand-written register machine diagram, a register machine program that compiles and runs in the language provided, or both.

If you do write register machine programs in the language provided, note that you can rely on any of the functions defined in the library, at the start of these exercises ([Appendix A](#)).

- (a) First projection function $p \in \mathbb{N}^2 \rightarrow \mathbb{N}$, where $p(x, y) \triangleq x$
- (b) Constant function with value $n \in \mathbb{N}$, $c_n \in \mathbb{N} \rightarrow \mathbb{N}$ where $c(x) \triangleq n$
- (c) Truncated subtraction function, $_ \dot{-} _ \in \mathbb{N}^2 \rightarrow \mathbb{N}$, 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, $_ \text{div} _ \in \mathbb{N}^2 \rightarrow \mathbb{N}$ 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, $_ \text{mod} _ \in \mathbb{N}^2 \rightarrow \mathbb{N}$ with $x \text{ mod } y \triangleq x \dot{-} y \cdot (x \text{ div } y)$
- (f) Exponentiation base 2, $e \in \mathbb{N} \rightarrow \mathbb{N}$, where $e(x) = 2^x$
- (g) Logarithm base 2, $\log_2 \in \mathbb{N} \rightarrow \mathbb{N}$, where

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

4 Gödel Encoding

PREREQUISITES Lecture 3 · TIME 2 hours 40 min (+ 5 min)

QUESTION 7

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

For each of the following n , describe $\text{instruction}(n)$, the instruction obtained by decoding n .

- (a) 0
- (b) 1
- (c) 2
- (d) 3

QUESTION 8

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

For each of the following n , describe $\text{prog}(n)$, the program obtained by decoding n . You can use $\text{instruction}(m)$ to mean the instruction obtained by decoding m . Describe also ϕ_n , the unary partial function $\mathbb{N} \rightarrow \mathbb{N}$ computed by $\text{prog}(n)$.

- (a) 0
- (b) 1
- (c) 2
- (d) 3
- (e) 4
- (f) 5
- (g) 1544

QUESTION 9

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

Show that for any register machine computable unary partial function $f : \mathbb{N} \rightarrow \mathbb{N}$, there are infinitely many encodings $e \in \mathbb{N}$ such that ϕ_e is extensionally equal to f .

QUESTION 10

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

What can you say about ϕ_e when e is an odd natural?

★ QUESTION 11

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

Investigate Haskell's arith-encode package. Explain how Gödel encoding is useful in a compiler.

QUESTION 12

CONCEPTUAL · 40 MIN

● ● ● ● ● CORE

In this question, we will explore different possible Gödel encodings for the same language, and in doing so illustrate progressively useful ways of encoding the same language.

Consider a language $\mathcal{L}$ with two types, $\mathbb{N}$, the naturals, and $\mathbb{S}$, strings from the language $\{a, b\}^*$:

$$\begin{aligned} n \in \mathbb{N} &= \{0, 1, 2, \dots\} \\ s \in \mathbb{S} &= \{ "", "a", "aa", "ab", \dots \} \end{aligned}$$

Expressions in the language are defined recursively. There are four expression formers:

$$e ::= n \mid s \mid e_1 \star e_2 \mid e_1 \circ e_2$$

The language also has a type system, defined as follows:

$$\frac{}{n : \mathbb{N}} \text{ (T-NAT)} \quad \frac{}{s : \mathbb{S}} \text{ (T-STR)} \quad \frac{e_1 : \mathbb{N} \quad e_2 : \mathbb{N}}{e_1 \circ e_2 : \mathbb{N}} \text{ (T-CIRC)} \quad \frac{e_1 : \mathbb{S} \quad e_2 : \mathbb{S}}{e_1 \star e_2 : \mathbb{S}} \text{ (T-CAT)}$$

Finally, this language has a system of equations. Use $e_{\mathbb{N}}$ to mean an expression of type $\mathbb{N}$ and $e_{\mathbb{S}}$ to mean an expression of type $\mathbb{S}$:

$$\begin{aligned} 0 \circ e_{\mathbb{N}} &= e_{\mathbb{N}} & "" \star e_{\mathbb{S}} &= e_{\mathbb{S}} \\ e_{\mathbb{N}} \circ 0 &= e_{\mathbb{N}} & e_{\mathbb{S}} \star "" &= e_{\mathbb{S}} \\ (e_{\mathbb{N}} \circ e'_{\mathbb{N}}) \circ e''_{\mathbb{N}} &= e_{\mathbb{N}} \circ (e'_{\mathbb{N}} \circ e''_{\mathbb{N}}) & (e_{\mathbb{S}} \star e'_{\mathbb{S}}) \star e''_{\mathbb{S}} &= e_{\mathbb{S}} \star (e'_{\mathbb{S}} \star e''_{\mathbb{S}}) \end{aligned}$$

Strings are encoded as

$$\lceil "" \rceil = 0, \quad \lceil a :: s \rceil = 2^{\lceil s \rceil} + 1, \quad \lceil b :: s \rceil = 2^{\lceil s \rceil} + 2$$

We shall now consider Gödel encodings $\mathcal{L} \rightarrow \mathbb{N}$ of expressions in $\mathcal{L}$.

(a) An expression $"aba" \star "bab"$ can be seen as a list of characters

$$["; a; b; a; "; \star; "; b; a; b; "]$$

(i) Find a Gödel encoding for lists of characters that establishes at least an injection from lists of characters to naturals. You do not need to prove that it is injective.

HINT For an injection, encode each character, then encode the list of encodings.

A bijection is trickier, but note that there are 15 characters.

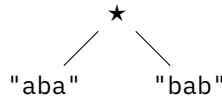
(ii) Give a number whose decoding corresponds to a gibberish sequence.

(iii) Give a number whose decoding corresponds to two different syntactic expressions (they may have the same meaning under $=$).

- (b) An expression "aba" ★ "bab" can be seen as a list of tokens

$$["aba"; \star; "bab"]$$

- (i) Find a Gödel encoding for lists of tokens that establishes a bijection between lists of tokens and naturals. You do not need to prove that it is bijective.
 - (ii) Explain whether a number can decode to a gibberish sequence containing non-token strings.
 - (iii) Give a number whose decoding corresponds to two different expression trees (even if they are equal under =).
 - (iv) Give a number whose decoding does not correspond to a grammatical expression in $\mathcal{L}$.
- (c) An expression "aba" ★ "bab" can be seen as an application of expression formers.



Consider an encoding of grammatical expressions that gives each expression former its own remainder modulo 4, and encodes the subexpressions recursively:

$$\begin{aligned} \lceil n \rceil_{\mathcal{L}} &= 4n & \lceil e_1 \circ e_2 \rceil_{\mathcal{L}} &= 4\langle \lceil e_1 \rceil_{\mathcal{L}}, \lceil e_2 \rceil_{\mathcal{L}} \rangle + 2 \\ \lceil s \rceil_{\mathcal{L}} &= 4\lceil s \rceil + 1 & \lceil e_1 \star e_2 \rceil_{\mathcal{L}} &= 4\langle \lceil e_1 \rceil_{\mathcal{L}}, \lceil e_2 \rceil_{\mathcal{L}} \rangle + 3 \end{aligned}$$

For example,

$$\begin{aligned} \lceil "aba" \rceil_{\mathcal{L}} &= 4(9) + 1 = 37 \\ \lceil "bab" \rceil_{\mathcal{L}} &= 4(12) + 1 = 49 \\ \lceil "aba" \star "bab" \rceil_{\mathcal{L}} &= 4\langle 37, 49 \rangle + 3 = 2^{39} \times 99 - 1 \end{aligned}$$

- (i) Explain whether a number can decode to a non-grammatical expression.
 - (ii) Give a number whose decoding does not correspond to a well-typed expression.
- (d) Consider yet another encoding: An expression "aba" ★ "bab" can be seen as a typing derivation

$$\frac{\frac{}{"aba" : \mathbb{S}} \text{ (T-STR)}}{\text{ "aba" } \star \text{ "bab" : } \mathbb{S}} \text{ (T-CAT)}$$

- (i) Give a Gödel encoding of typing derivations that establishes a bijection between well-typed expressions and the naturals.
 - (ii) Can a number decode to an ill-typed expression?
 - (iii) Give an example of two expressions that are equal under = and encode differently.
- (e) Consider a final encoding: First, consider two interpretations, one on well-typed expressions of type $\mathbb{N}$, and one on well-typed expressions of type $\mathbb{S}$. Let ++ mean list append.

$$\begin{aligned} \llbracket 0 \rrbracket_{\mathbb{N}} &= [] & \llbracket "" \rrbracket_{\mathbb{S}} &= [] \\ \llbracket n \rrbracket_{\mathbb{N}} &= [n] \quad (n \neq 0) & \llbracket s \rrbracket_{\mathbb{S}} &= [s] \quad (s \neq "") \\ \llbracket e_1 \circ e_2 \rrbracket_{\mathbb{N}} &= \llbracket e_1 \rrbracket_{\mathbb{N}} ++ \llbracket e_2 \rrbracket_{\mathbb{N}} & \llbracket e_1 \star e_2 \rrbracket_{\mathbb{S}} &= \llbracket e_1 \rrbracket_{\mathbb{S}} ++ \llbracket e_2 \rrbracket_{\mathbb{S}} \end{aligned}$$

For example,

$$\llbracket "aba" \star (" " \star "bab") \rrbracket_{\mathbb{S}} = \llbracket "aba" \rrbracket_{\mathbb{S}} ++ (\llbracket "" \rrbracket_{\mathbb{S}} ++ \llbracket "bab" \rrbracket_{\mathbb{S}}) = \llbracket "aba"; "bab" \rrbracket_{\mathbb{S}}$$

Now consider the following encoding, where $\lceil l \rceil$ is the encoding of lists from the notes. An expression e of type $\mathbb{N}$ with $\llbracket e \rrbracket_{\mathbb{N}} = [n_1; \dots; n_k]$ encodes as $\lceil e \rceil_{\mathcal{L}/=} = 2^{\lceil n_1 - 1; \dots; n_k - 1 \rceil}$. An expression e of type $\mathbb{S}$ with $\llbracket e \rrbracket_{\mathbb{S}} = [s_1; \dots; s_k]$ encodes as $\lceil e \rceil_{\mathcal{L}/=} = 2^{\lceil \lceil s_1 \rceil - 1; \dots; \lceil s_k \rceil - 1 \rceil} + 1$. For example, $\text{"aba"} \star (\text{" "} \star \text{"bab"})$ encodes as $2^{\lceil 8; 11 \rceil} + 1 = 2^{21} + 2^9 + 1$.

- (i) Show that $e_{\mathbb{N}} \circ 0$ and $e_{\mathbb{N}}$ are interpreted as the same list.
- (ii) Show that $0 \circ e_{\mathbb{N}}$ and $e_{\mathbb{N}}$ are interpreted as the same list.
- (iii) Show that $(e_1 \circ e_2) \circ e_3$ and $e_1 \circ (e_2 \circ e_3)$ are interpreted as the same list, where e_1, e_2 and e_3 have type $\mathbb{N}$.
- (iv) If $e = e'$, what can you say about $\lceil e \rceil_{\mathcal{L}/=}$ and $\lceil e' \rceil_{\mathcal{L}/=}$?
- (v) If $\lceil e \rceil_{\mathcal{L}/=} = n = \lceil e' \rceil_{\mathcal{L}/=}$, what can you say about e and e' ?

In your answer, you can appeal for free to the following facts: for all lists l, l_1, l_2 and l_3 ,

$$\begin{aligned} [] \mathbin{++} l &= l \\ l \mathbin{++} [] &= l \\ (l_1 \mathbin{++} l_2) \mathbin{++} l_3 &= l_1 \mathbin{++} (l_2 \mathbin{++} l_3) \end{aligned}$$

- (f) Explain whether a procedure like the one in the previous part can be applied to register machines, or if not, why not.

QUESTION 13

CONCEPTUAL · 20 MIN

●●●●● CORE

Consider the encoding defined in [Part 12\(c\)](#).

- (a) Assume an AST e , encoded as a natural number $\lceil e \rceil_{\mathcal{L}}$. Write an OCaml function `splice` that takes $\lceil e \rceil_{\mathcal{L}}$ and returns $\lceil e \circ 4 \rceil_{\mathcal{L}}$. *Note: you may ignore integer overflow.*
- (b) Can $\mathcal{L}$ express `splice`?
- (c) Gödel encodings are an important tool in the field of [metamathematics](#). Metamathematics studies mathematical systems using other mathematical systems.

The mathematical system being studied tends to be called the *object* language. The mathematical system used to study the object language tends to be called the *meta* language.

Identify the object and meta language in this example.

QUESTION 14

CONCEPTUAL · 30 MIN

●●●●● CORE

This question will be easier after Lecture 5.

- (a) Consider $\phi_e : \mathbb{N}^2 \rightarrow \mathbb{N}$. That is, each $e \in \mathbb{N}$ encodes a register machine with two arguments, x and y . Show that there is a total register machine computable function $S : \mathbb{N}^2 \rightarrow \mathbb{N}$ such that

$$\forall e, x, y \in \mathbb{N}. \phi_{S(e,x)}(y) = \phi_e(x, y)$$

You may assume, in your construction, that $\text{prog}(e)$, for decoding an encoded program, and $\lceil e \rceil$, for encoding a program, are given. Assume also $\text{copy}(R, R')$, which copies the contents of R' to R , and $\text{const}_n(R)$, which fills the contents of a register R with a constant n .

You do not need to construct the full register machine, but you should explain what it computes, how to construct it, and why it is total.

- (b) We now generalise the finding in [Part 14\(a\)](#) to arbitrary arities. Assume positive integers m, n . Consider $\phi_e : \mathbb{N}^{m+n} \rightarrow \mathbb{N}$. That is, each $e \in \mathbb{N}$ encodes a register machine with $m + n$ arguments, $\vec{x}$ and $\vec{y}$. Show that there is a total register machine computable function $S_n^m : \mathbb{N}^{1+m} \rightarrow \mathbb{N}$ such that

$$\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n. \phi_{S_n^m(e, \vec{x})}(\vec{y}) = \phi_e(\vec{x}, \vec{y})$$

This result is known as the *S-m-n theorem*.

- (c) Relate your result in [Part 14\(b\)](#) to [Question 13](#). How does S act on e and $\vec{x}$?
 (d) Rather than considering computable partial functions ϕ_e , consider an OCaml function

`f: nat * nat -> nat.`

Show there exists an OCaml function s such that for all y , $s(f, x)(y) = f(x, y)$.

- (e) Assume $\vec{x}$ is known before $\vec{y}$. What are the benefits of using the function $\phi_{S_n^m(e, \vec{x})}$ over ϕ_e ?

QUESTION 15

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

In this question, we will see how, as computer scientists, we can use the concept of Gödel encoding to write fast but maintainable programs.

Gödel encoding may seem very abstract. One use case in programming languages is **metaprogramming**.

In [Question 13](#), the `splice` program is a metaprogram, or program generator: written in the metalanguage (OCaml), when `splice` is run, it generates code in the object language ($\mathcal{L}$). Metaprogramming is the act of writing code generators like `splice`. Many languages offer metaprogramming support, for example, [C macros](#), [C++ templates](#), and [MetaOCaml](#) quotes and splices.

The [Modular Macros](#) project, run by [Dr. Jeremy Yallop](#), aims to extend the OCaml compiler with metaprogramming facilities. We will investigate metaprogramming through the lens of `MacoCaml`. An online compiler and interpreter for `MacoCaml` is accessible [here](#).

One important use of metaprogramming is writing code that is both fast and maintainable.

Consider the following problem: you are asked to write, in OCaml, functions that compute the square of an integer, x^2 , and the cube of an integer, x^3 .

There are two ways to go about doing this. First, you can write the functions one by one:

```
1 let square x = x * x
2 let cube x = x * x * x
```

OCaml

This approach is generally criticised in software engineering as being not maintainable. Imagine someone now wants a `pow4` function, or a `pow5` function: you would have to re-write each, one by one.

A more maintainable approach is thus to write a power function `pow n x`, and instantiate it at 2 and 3:

```
1 let rec pow n x =
```

OCaml

OCaml

```

2   if n = 0 then 1
3   else x * pow (n-1) x
4
5   let square = pow 2
6   let cube = pow 3

```

This approach is more maintainable: it easily extends to powers 4, 5, ... However, the cost is *efficiency*: going through the `pow` function costs an indirection, a conditional, and several recursive calls.

With `MacoCaml`, we can use metaprogramming annotations `<<->` (“quote”) and `$-` (“splice”). These annotations transform the power function into a power program generator, or *macro*, which resembles the maintainable program. The call to the macro, `$(pow 2 <<x>>)`, runs at compile time. When it runs, it generates the code of `x * (x * 1)`, which resembles the fast program.

OCaml

```

1   macro pow (n : int) (x : int expr) =
2     if n = 0 then <<1>>
3     else << $x * $(pow (n-1) x) >>
4
5   let square x = $(pow 2 <<x>>) (* executes at compile-time,
6                                generates x * (x * 1) *)

```

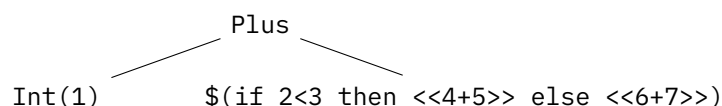
At the term level, `MacoCaml` extends OCaml in three ways:

1. An `'a expr` type. For example, `int expr` represents encodings of OCaml ints.
2. A `<<->` (pronounced “quote”) operator. For example, `<<3>>` has type `int expr`, and returns the encoding of 3.
3. A `$-` (pronounced “splice”) operator, that stops the encoding process. Sometimes, we would like to run some code to decide what to encode. A splice lets us run OCaml code under the `<<->`. For example, `<<1 + $(if true then <<2>> else <<3>>)>>` returns `<<1 + 2>>`.

For example, let us assume OCaml programs are encoded as ASTs. *Note: This is a simplification, conceptually, they behave more like typed ASTs.* Now assume we are executing OCaml at compile time, and consider the example

```
<<1 + $(if 2<3 then <<4+5>> else <<6+7>>)>>
```

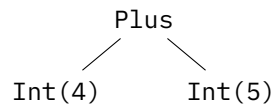
We first see the outer quote, so we switch into encoding mode, and encode `+` as the AST node `Plus`, and 1 as the AST node `Int(1)`.



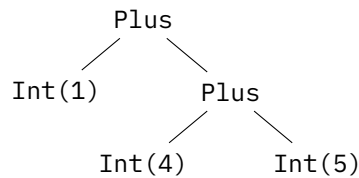
When we try to encode the right subtree, we see a splice, so we stop encoding, and treat the `if` statement as regular OCaml code. Running the OCaml code should give us an AST that we can splice in as the right subtree, much like you did in [Part 13\(a\)](#).

To evaluate the `if`, we first evaluate `2 < 3` to `true`, obtaining `<<4+5>>`.

The quotes tell us to switch back to encoding mode, so we get the AST



which we then splice into the larger AST



Thus, the program generates the AST (more generally, the encoding) of $1+(4+5)$, which we write $\ll 1+(4+5) \gg$. When we run this at runtime, we will get 10.

- Identify the metalanguage and object language in MacoCaml.
- Explain how `pow 2 <<x>>` generates $x * (x * 1)$ when run.
- C macros and C++ templates work very similarly, except that they respectively manipulate list of tokens and ASTs, rather than typed ASTs. With reference to your answer in [Part 12\(b\)](#), explain the problem with the approach adopted by C / C++.

5 Universal Register Machine

PREREQUISITES Lecture 4 · TIME 1 hour 30 min

QUESTION 16

CONCEPTUAL · 10 MIN

●●●●● CORE

A register machine computable partial function $\mathbb{N} \rightarrow \mathbb{B}$ can be encoded as a partial function $\mathbb{N} \rightarrow \{0, 1\}$, where 0 plays the role of $\top$ (true) and 1 plays the role of $\perp$ (false). Assume two such functions, $f : \mathbb{N} \rightarrow \{0, 1\}$ and $g : \mathbb{N} \rightarrow \{0, 1\}$. Show that the following partial function is register machine computable

$$(f \oplus g)(x) = \begin{cases} 0, & f(x) = 0 \vee g(x) = 0 \\ 1, & f(x) = 1 \wedge g(x) = 1 \\ \uparrow, & \text{otherwise} \end{cases}$$

HINT Be careful: What happens when $f(x)$ returns true, but $g(x) \uparrow$?

Supervision 2

Lent Term 2027 · Michael Lee

WORK DUE	A single PDF, 48 hours before the supervision
EXPECTED TIME	6 hours 5 min (+ 35 min for the optional questions)
OPTIONAL	★ marks a question, or a part of one, you may skip

1 The Halting Problem and Decidability

PREREQUISITES Lecture 5 · TIME 1 hour 50 min

QUESTION 1

CONCEPTUAL · 20 MIN

● ● ● ● ● BASIC

Consider the notion of semi-decidability, also known as recursive enumerability.

The semi-characteristic function of a set A , ψ_A , is defined as:

$$\psi_A(x) = \begin{cases} 1, & x \in A \\ \uparrow, & x \notin A \end{cases}$$

A set A is (register-machine) semi-decidable if its semi-characteristic function is (register machine) computable.

- (a) Show that all decidable sets are semi-decidable.
 - (b) Give an example of a semi-decidable set that is not decidable.
 - (c) Prove that if a set S is semi-decidable, and its complement S^c is also semi-decidable, then S is decidable.
 - (d) Give an example of a set that is not semi-decidable.
-

QUESTION 2

CONCEPTUAL · 30 MIN

● ● ● ● ● CORE

Show that decidable sets are closed under union, intersection, and complementation.

In other words, show that, for any decidable sets X and Y

- (a) $X \cup Y$ is decidable
- (b) $X \cap Y$ is decidable
- (c) X^c is decidable

Which of these statements is true for undecidable sets?

QUESTION 3

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

Assume a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$.

Define the graph of f , $\text{graph}(f)$, as the subset of $\mathbb{N} \times \mathbb{N}$

$$\{(x, y) \in \mathbb{N} \times \mathbb{N} \mid f(x) \downarrow \wedge f(x) = y\}$$

Define the domain of f , $\text{dom}(f)$, as the subset of $\mathbb{N}$ at which f is defined

$$\{x \in \mathbb{N} \mid f(x) \downarrow\}$$

Define the range of f , $\text{range}(f)$, as the subset of $\mathbb{N}$

$$\{y \in \mathbb{N} \mid \exists x \in \mathbb{N}. f(x) \downarrow \wedge f(x) = y\}$$

Note that decidability can also be defined over sets of pairs $\mathbb{N} \times \mathbb{N}$, since these are in bijection (using Gödel encoding and decoding) with numbers $\mathbb{N}$.

- Construct a function f such that $\text{dom}(f)$ and $\text{range}(f)$ are both decidable, but $\text{graph}(f)$ is not decidable.
- Construct a function f such that $\text{graph}(f)$ is decidable, but $\text{dom}(f)$ is not decidable.
- Construct a function f such that $\text{graph}(f)$ is decidable, but $\text{range}(f)$ is not decidable.
- Show that if $\text{graph}(f)$ is decidable, then f is register machine computable.
- Construct a function f which is register machine computable but where $\text{dom}(f)$ and $\text{range}(f)$ are both not decidable.

QUESTION 4

CONCEPTUAL · 20 MIN

● ● ● ● ● BASIC

Let A, B be subsets of $\mathbb{N}$. Write $A \leq B$ if there exists a total computable function $r : \mathbb{N} \rightarrow \mathbb{N}$ that satisfies the following property

$$\forall n \in \mathbb{N}. n \in A \iff r(n) \in B$$

- Assume $A \leq B$. Show that if B is decidable, then A is.
- Consider the converse: If $A \leq B$, and A is decidable, must B be?
- Explain, informally, how to read $\leq$.

QUESTION 5

CONCEPTUAL · 20 MIN

● ● ● ● ● BASIC

Show that the following set E is undecidable. $=$ is extensional equality.

$$E \triangleq \{(e, e') \mid \phi_e = \phi_{e'}\}$$

2 Turing Machines

PREREQUISITES Lecture 7 · TIME 45 min (+ 35 min)

QUESTION 6

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

Explain whether the Turing Machine introduced in the notes is deterministic, or non-deterministic.

QUESTION 7

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

The notes state

We have seen that a Turing machine's computation can be implemented by a register machine. The converse holds, too: the computation of a register machine can be implemented by a Turing machine. To make sense of this, **we first have to fix a tape representation of register machine configurations and hence of numbers and lists of numbers.**

Why do we need to fix a tape representation of numbers? Why can we not set $\Sigma = \mathbb{N} \cup \{\sqcup, \triangleright\}$?

QUESTION 8

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

Some Turing Machines require the head of the machine to move left or right, rather than stay still, during a state transition.

The formalisation in the course allows the head of the machine to remain still during a state transition. Explain how this choice adds no additional expressiveness.

★ QUESTION 9

CONCEPTUAL · 30 MIN

● ● ● ● ● CORE

Define a Turing Machine that accepts all binary palindromes, and rejects all other strings.

QUESTION 10

CONCEPTUAL · 20 MIN

●●●●● CORE

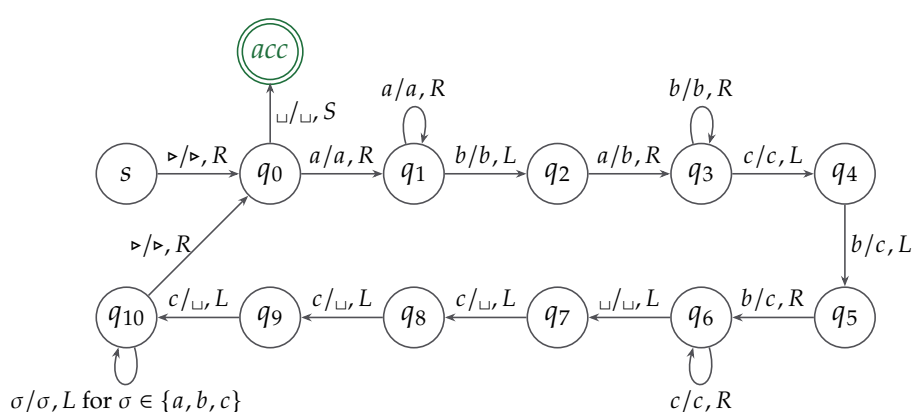
Explain what the following Turing Machine does. It is started in the configuration $(s, \triangleright, u)$ with $u \in \{a, b, c\}^*$.

The Turing Machine has many transitions: the ones that are unreachable are faded out.

$M = (Q, \Sigma, s, \delta)$ where $Q = \{s, q_0, \dots, q_{10}\}$ (s initial), $\Sigma = \{\triangleright, \sqcup, a, b, c\}$, and δ is

δ	$\triangleright$	$\sqcup$	a	b	c
s	$\langle q_0, \triangleright, R \rangle$	$\langle rej, \sqcup, R \rangle$	$\langle rej, a, R \rangle$	$\langle rej, b, R \rangle$	$\langle rej, c, R \rangle$
q_0	$\langle rej, \triangleright, R \rangle$	$\langle acc, \sqcup, S \rangle$	$\langle q_1, a, R \rangle$	$\langle rej, b, S \rangle$	$\langle rej, c, S \rangle$
q_1	$\langle rej, \triangleright, R \rangle$	$\langle rej, \sqcup, S \rangle$	$\langle q_1, a, R \rangle$	$\langle q_2, b, L \rangle$	$\langle rej, c, S \rangle$
q_2	$\langle rej, \triangleright, R \rangle$	$\langle rej, \sqcup, R \rangle$	$\langle q_3, b, R \rangle$	$\langle rej, b, R \rangle$	$\langle rej, c, R \rangle$
q_3	$\langle rej, \triangleright, R \rangle$	$\langle rej, \sqcup, S \rangle$	$\langle rej, a, S \rangle$	$\langle q_3, b, R \rangle$	$\langle q_4, c, L \rangle$
q_4	$\langle rej, \triangleright, R \rangle$	$\langle rej, \sqcup, R \rangle$	$\langle rej, a, R \rangle$	$\langle q_5, c, L \rangle$	$\langle rej, c, R \rangle$
q_5	$\langle rej, \triangleright, R \rangle$	$\langle rej, \sqcup, R \rangle$	$\langle rej, a, R \rangle$	$\langle q_6, c, R \rangle$	$\langle rej, c, R \rangle$
q_6	$\langle rej, \triangleright, R \rangle$	$\langle q_7, \sqcup, L \rangle$	$\langle rej, a, S \rangle$	$\langle rej, b, S \rangle$	$\langle q_6, c, R \rangle$
q_7	$\langle rej, \triangleright, R \rangle$	$\langle rej, \sqcup, R \rangle$	$\langle rej, a, R \rangle$	$\langle rej, b, R \rangle$	$\langle q_8, \sqcup, L \rangle$
q_8	$\langle rej, \triangleright, R \rangle$	$\langle rej, \sqcup, R \rangle$	$\langle rej, a, R \rangle$	$\langle rej, b, R \rangle$	$\langle q_9, \sqcup, L \rangle$
q_9	$\langle rej, \triangleright, R \rangle$	$\langle rej, \sqcup, R \rangle$	$\langle rej, a, R \rangle$	$\langle rej, b, R \rangle$	$\langle q_{10}, \sqcup, L \rangle$
q_{10}	$\langle q_0, \triangleright, R \rangle$	$\langle rej, \sqcup, R \rangle$	$\langle q_{10}, a, L \rangle$	$\langle q_{10}, b, L \rangle$	$\langle q_{10}, c, L \rangle$

You may prefer to view the Turing Machine as a state transition diagram. Rejecting edges are implicit, and have been left out.



QUESTION 11

CONCEPTUAL · 5 MIN (+ 5 MIN)

●●●●● BASIC

(a) Construct a notion of Turing-Machine decidability.

HINT You may find it helpful to fix a concrete Σ , like $\{0, 1, \triangleright, \sqcup\}$.

- ★ (b) List other forms of decidability that are not equivalent to Turing-Machine decidability. You should feel free to use your answers to [Question 9](#) and [Question 10](#).

3 Primitive Recursive Functions

PREREQUISITES Lecture 9 · TIME 2 hours 30 min

QUESTION 12

CONCEPTUAL · 30 MIN

● ● ● ● ● CORE

Recall the fold function for lists, in OCaml.

```
1 let rec fold_r b f = function
2   | [] -> b
3   | x::xs -> f x (fold_r b f xs)
```

OCaml

A very similar concept to a fold is a *recursor*.

```
1 let rec recurse_r b f = function
2   | [] -> b
3   | x::xs -> f xs x (recurse_r b f xs)
```

OCaml

- (a) Show that every fold is a recursor.
 (b) Show that every recursor is a fold.

Consider the OCaml datatype for natural numbers, defined as follows:

```
type nat = Z | S of nat
```

With the following functions given for free:

```
1 let succ n = S n
2 let zero = Z
3 let comp g f = fun x -> g (f x)
```

OCaml

- (c) Design a fold function for naturals. It should have the following type

```
val fold : nat -> (nat -> nat) -> nat -> nat
```

Note that an implementation may have a more general type, but we will ignore this difference for now.

- (d) Design a recurse function for naturals. It should have the following type

```
val recurse : nat -> (nat -> nat -> nat) -> nat -> nat
```

Note that an implementation may have a more general type, but we will ignore this difference for now.

- (e) By instantiating your recurse function in [Part 12\(d\)](#), write the following functions

(i) `val add : nat -> nat -> nat`

- (ii) `val triangle : nat -> nat`
- (iii) `val pred : nat -> nat`
- (iv) `val mul : nat -> nat -> nat`
- (f) Modify the statement of Theorem 19 in the notes to drop the additional parameters $\vec{x}$.
- (g) Compare your answers in [Part 12\(d\)](#) and [Part 12\(f\)](#).
- (h) Extend your recursor over the naturals in [Part 12\(d\)](#) to take a list of additional arguments $\vec{x}$ (call it `xs`). Call this alternative recursor `recurse'`. Explain why, in OCaml, every use of `recurse'` is equivalent to a use of `recurse`.
- (i) Explain why the definition of primitive recursion in the course uses (the equivalent of) `recurse'`, rather than `recurse`.

QUESTION 13

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

The notes define the set of primitive recursive functions, PRIM , as follows:

A (partial) function f is primitive recursive ($f \in \text{PRIM}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition and primitive recursion.

- (a) Rephrase this definition as an *inductive* definition, as in Part IB Semantics of Programming Languages.
- (b) Write an OCaml ADT that captures the syntax of primitive recursion

QUESTION 14

CONCEPTUAL · 30 MIN

● ● ● ● ● CORE

- (a) Assume there is a function `ackm` that computes $A_m = \text{ack}(m, -)$. By instantiating `recurse`, write an OCaml function `ackm'` that computes $A_{m+1} = \text{ack}(m+1, -)$

```
val ackm' : nat -> nat
```

The existence of `ackm'` shows that A_{m+1} can be defined by applying primitive recursion to A_m and the basic functions.

- (b) Now make the dependency on `ackm` explicit, by modifying your definition such that `m` and `ackm` are passed as an argument

```
val ackm' : nat -> (nat -> nat) -> nat -> nat
              (*m      ackm          n      ack(m+1, n) *)
```

Convince yourself that `ackm'`, as defined, computes A_{m+1} from A_m

- (c) Consider generalising the return type of the recursor from `nat` to `'a`

```
val recurse : 'a -> (nat -> 'a -> 'a) -> nat -> 'a
```

By choosing suitable `'a`, use `recurse` to write the Ackermann function

```
val ack : nat -> nat -> nat
```

in a primitive recursive style.

HINT How can you use your answer to the previous parts of the question?

- (d) Prove that, for all m , A_m is primitive recursive.
- (e) Explain what your answer to [Part 14\(c\)](#) implies about the relationship between the Ackermann function, recursors, and *primitive recursive functions*.

QUESTION 15

CONCEPTUAL · 40 MIN

● ● ● ● ● CORE

Show that the following functions are all primitive recursive.

You can do this abstractly, by composing together primitive recursive functions (the notes define a set of primitive recursive functions: you can assume these are primitive recursive for free).

You can also do this concretely. I have defined a language for writing primitive recursive functions [here](#). Each of the building blocks of primitive recursion has a corresponding construct in the language:

1. $\text{proj}[n, i]$ is the projection proj_i^n .
2. $\text{zero}[n]$ is the zero function zero^n .
3. succ is the successor function succ .
4. $\text{f} \cdot (g_1, \dots, g_m)$ is the composition $f \circ [g_1, \dots, g_m]$. When $m = 1$, you may write $\text{f} \cdot g$.
5. $\text{rho}[n](\text{f}, g)$ is the primitive recursion $\rho^n(f, g)$.
6. $\text{let } h = e$ names the function e as h .
7. An expression $h(x_1, \dots, x_n)$ at the end of the file applies h to the numbers $x_1, \dots, x_n$. When $n = 1$, you may write $h x$.

I have also provided a standard library of primitive recursive functions, corresponding to each of the primitive recursive functions defined by the notes ([Appendix B](#)). You may use any of these “standard library” functions.

- (a) Truncated subtraction function, $\text{minus}: \mathbb{N}^2 \rightarrow \mathbb{N}$, where

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

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

- (c) Conditional branch on zero, $\text{ifzero}: \mathbb{N}^3 \rightarrow \mathbb{N}$, where

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

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

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

QUESTION 16

CONCEPTUAL · 30 MIN

● ● ● ● ● BASIC

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

- (a) the binary maximum function $\text{max}: \mathbb{N}^2 \rightarrow \mathbb{N}$.
- (b) the integer square root function $\text{sqrt}: \mathbb{N} \rightarrow \mathbb{N}$ which is only defined if its argument is a perfect square.

Once again, you may do this abstractly. Alternatively, you can use the [programming language](#) given in [Question 15](#). You can write μ^n as `mu[n]`.

Minimisation can result in non-termination. Thus, you need to run the programs with minimisation explicitly turned on, using the `-m` flag, as in `pr -m [FILE]`.

4 The Lambda Calculus: Syntax

PREREQUISITES Lecture 10 · TIME 20 min

QUESTION 17

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

Give the syntax of the lambda calculus as an inductive definition, as in the Part IB Semantics of Programming Languages course.

QUESTION 18

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

Draw the abstract syntax trees corresponding to each of the following concrete expressions. You can use `@` to mean application.

- (a) $\lambda x. u \ x \ y$
 - (b) $\lambda x. u \ (x \ y)$
 - (c) $\lambda x. x \ \lambda y. y$
 - (d) $(\lambda x. x) (\lambda y. y)$
-

5 The Lambda Calculus: Equational Systems

PREREQUISITES None · TIME 40 min

QUESTION 19

CONCEPTUAL · 40 MIN

● ● ● ● ● CORE

This question encourages you to explore the relationship between equational systems and reduction systems.

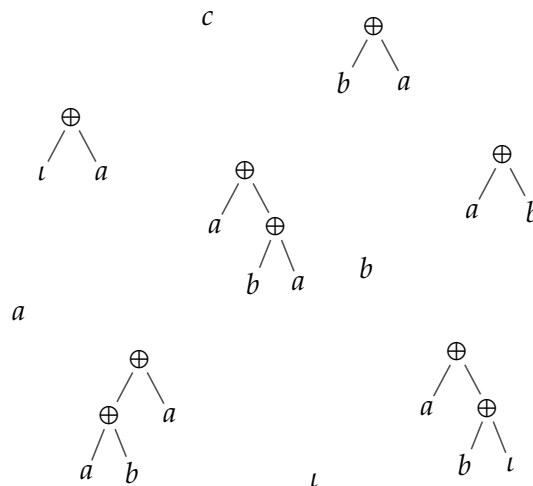
Consider a programming language whose syntax is defined as follows:

$$\begin{aligned} x &::= a \mid b \mid c \\ e &::= \iota \mid x \mid e \oplus e' \end{aligned}$$

with the equations

$$\begin{aligned} \iota \oplus e &= e \\ e \oplus \iota &= e \\ e \oplus (e' \oplus e'') &= (e \oplus e') \oplus e'' \end{aligned}$$

(a) Consider the following abstract syntax trees.



Group ASTs such that all ASTs in a group are equal to one another, and not equal to any AST in any other group.

- Each group in your answer to [Part 19\(a\)](#) contains a selection of elements from the same equivalence class. Why is the notion of a *normal form* useful? What properties should a normal form have?
- Suggest two distinct normal forms for this language. Convert $(a \oplus b) \oplus (\iota \oplus (c \oplus b))$ to each normal form.
- For each of the normal forms in [Part 19\(c\)](#), propose an algorithm for converting an arbitrary expression e into that normal form. Relate your answer to the notion of a *reduction system*.
- Suggest an algorithm for determining if two expressions e and e' are in the same equivalence class.
- Does your algorithm always terminate with a yes/no answer? What can you conclude about the decidability of $=$ as defined above?

Supervision 3

Lent Term 2027 · Michael Lee

WORK DUE	A single PDF, 48 hours before the supervision
EXPECTED TIME	5 hours 50 min (+ 10 min for the optional questions)
OPTIONAL	★ marks a question, or a part of one, you may skip

PREREQUISITES Lecture 11 · TIME 2 hours 10 min

1 The Lambda Calculus: Normal Forms

(+ 10 min)

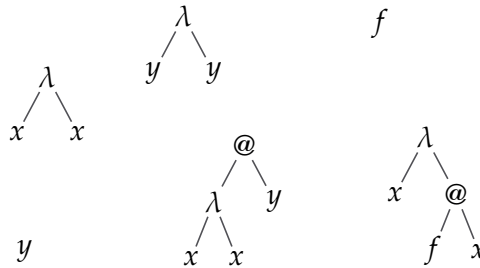
QUESTION 1

CONCEPTUAL · 40 MIN

● ● ● ● ● CORE

This question considers different notions of equivalence in the lambda calculus

Consider the following lambda terms.



- Divide these lambda terms into groups that share the same β -normal form, up to α -equivalence.
- Divide these lambda terms into equivalence classes, where all lambda terms in the same equivalence class are contextually equivalent — they behave the same way.
- Another notion of equivalence is α -equivalence. Formally, α -equivalence is defined as follows:

$$\begin{array}{c}
 \frac{}{x \equiv_{\alpha} x} \text{ (VAR)} \qquad \frac{z \# (M N) \quad M\{z/x\} \equiv_{\alpha} N\{z/y\}}{\lambda x. M \equiv_{\alpha} \lambda y. N} \text{ (LAM)} \\
 \\
 \frac{M \equiv_{\alpha} M' \quad N \equiv_{\alpha} N'}{M N \equiv_{\alpha} M' N'} \text{ (APP)}
 \end{array}$$

Divide these lambda terms into α -equivalence classes.

- A third notion of equivalence is β -equivalence, defined as follows:¹

¹Use the notes' definition of $M[N/x]$ without the α -renaming step, so (BETA) applies only when no free variable of N is captured.

$$\begin{array}{c}
\frac{}{(\lambda x. M) N \equiv_{\beta} M[N/x]} \text{ (BETA)} \quad \frac{M \equiv_{\beta} M'}{\lambda x. M \equiv_{\beta} \lambda x. M'} \text{ (LAM)} \\
\frac{M \equiv_{\beta} M' \quad N \equiv_{\beta} N'}{M N \equiv_{\beta} M' N'} \text{ (APP)} \\
\frac{}{M \equiv_{\beta} M} \text{ (REFL)} \quad \frac{M \equiv_{\beta} M'}{M' \equiv_{\beta} M} \text{ (SYM)} \quad \frac{M \equiv_{\beta} M' \quad M' \equiv_{\beta} M''}{M \equiv_{\beta} M''} \text{ (TRANS)}
\end{array}$$

Divide these lambda terms into β -equivalence classes.

- (e) A fourth notion of equivalence is η -equivalence, defined as follows:

$$\begin{array}{c}
\frac{x \notin \text{fv}(M)}{\lambda x. M x \equiv_{\eta} M} \text{ (ETA)} \quad \frac{M \equiv_{\eta} M'}{\lambda x. M \equiv_{\eta} \lambda x. M'} \text{ (LAM)} \quad \frac{M \equiv_{\eta} M' \quad N \equiv_{\eta} N'}{M N \equiv_{\eta} M' N'} \text{ (APP)} \\
\frac{}{M \equiv_{\eta} M} \text{ (REFL)} \quad \frac{M \equiv_{\eta} M'}{M' \equiv_{\eta} M} \text{ (SYM)} \quad \frac{M \equiv_{\eta} M' \quad M' \equiv_{\eta} M''}{M \equiv_{\eta} M''} \text{ (TRANS)}
\end{array}$$

Divide these lambda terms into η -equivalence classes.

- (f) A fifth notion of equivalence is $\alpha\beta$ -equivalence, which relates terms that are related under either α , or β , or some combination of the two. Formally:

$$\frac{M \equiv_{\alpha} M'}{M \equiv_{\alpha\beta} M'} \text{ (ALPHA)} \quad \frac{M \equiv_{\beta} M'}{M \equiv_{\alpha\beta} M'} \text{ (BETA)} \quad \frac{M \equiv_{\alpha\beta} M' \quad M' \equiv_{\alpha\beta} M''}{M \equiv_{\alpha\beta} M''} \text{ (TRANS)}$$

Divide these lambda terms into $\alpha\beta$ -equivalence classes.

- (g) A sixth notion of equivalence is $\alpha\beta\eta$ -equivalence, which relates terms that are related under α , β , or η , or any combination of the three. Formally:

$$\begin{array}{c}
\frac{M \equiv_{\alpha} M'}{M \equiv_{\alpha\beta\eta} M'} \text{ (ALPHA)} \quad \frac{M \equiv_{\beta} M'}{M \equiv_{\alpha\beta\eta} M'} \text{ (BETA)} \quad \frac{M \equiv_{\eta} M'}{M \equiv_{\alpha\beta\eta} M'} \text{ (ETA)} \\
\frac{M \equiv_{\alpha\beta\eta} M' \quad M' \equiv_{\alpha\beta\eta} M''}{M \equiv_{\alpha\beta\eta} M''} \text{ (TRANS)}
\end{array}$$

Divide these lambda terms into $\alpha\beta\eta$ -equivalence classes.

- (h) Explain which notion of equivalence is used by the notes.

QUESTION 2

CONCEPTUAL · 10 MIN

●●●●● BASIC

Suggest an algorithm for deciding if two terms are α -equivalent.

HINT The simplest approach involves defining a new representation in which you get α -equivalence for free. You have been taught one such representation in Part IB Semantics of Programming Languages.

QUESTION 3

CONCEPTUAL · 30 MIN

● ● ● ● ● BASIC

- (a) Fully reduce the following lambda term to its β -normal form using normal-order reduction

$$(\lambda x y. x) y ((\lambda z. z z) (\lambda z. z z))$$

- (b) Fully reduce the following lambda term to its β -normal form using normal-order reduction

$$(\lambda x y. (\lambda z. x) ((\lambda z. z z) (\lambda z. z z))) y$$

- (c) In what sense is β -reduction non-deterministic? In what sense is it deterministic?
- (d) Is normal-order reduction deterministic?
- (e) Using an example, demonstrate the difference between call-by-value and normal-order reduction.
- (f) Using an example, demonstrate the difference between call-by-name and normal-order reduction.

QUESTION 4

CONCEPTUAL · 30 MIN

● ● ● ● ● CORE

- (a) Explain why β -normal forms, as defined in the notes, are not a good normal form for $\alpha\beta\eta$ -equivalence.
- (b) Suggest a better normal form for $\alpha\beta\eta$ -equivalence.
- (c) Modify β -reduction, as defined by the notes, to obtain a different reduction system that reduces to the normal form identified in [Part 4\(b\)](#).

QUESTION 5

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

- (a) Find a context C where $C[\lambda x. x \Omega]$ has a β -normal form but $C[\lambda x. \Omega]$ does not.
- (b) Are all terms with no β -normal form contextually equivalent?
- (c) Can a term have no head normal form but have a β -normal form?
- (d) Can a term with no β -normal form have a head normal form?

QUESTION 6

CONCEPTUAL · 10 MIN (+ 10 MIN)

● ● ● ● ● CORE

This question is about **head order reduction**. Head order reduction reduces the head redex until the term is in head normal form, at which point it stops, even if the entire term is not in β -normal form. Let $\rightarrow_h$ represent one step of head order reduction.

You can use the following fact:

FACT

If a term M has a head normal form, then reducing the term with head order reduction will eventually terminate, and will transform the term into a head normal form.

- (a) Using $\lambda x. (\lambda y. y) x \Omega$ as an example, contrast head order reduction with normal-order reduction (you can write one step of normal-order reduction as $\rightarrow_n$).
- ★ (b) Assume $M \rightarrow_h M'$. Show that $M[N/x] \rightarrow_h M'[N/x]$.
- ★ (c) Show that, if M has no head normal form, then $M[N/x]$ has no head normal form either.
- (d) Assume M has no head normal form. Let M occur in head position in P

$$P = \lambda x_1 \dots x_k. M N_1 N_2 \dots N_m.$$

Does P have a head normal form? Why, or why not?

HINT You do not need to give a proof, only an informal explanation. If you want to prove this statement, proceed by induction on m . Also do [Parts 6\(b\)](#) and [6\(c\)](#), since their results are used in the proof.

2 Lambda-Definable Functions

PREREQUISITES Lecture 12 · TIME 3 hours 40 min

QUESTION 7

CONCEPTUAL · 30 MIN

● ● ● ● ● CORE

Consider the OCaml ADT for natural numbers

```
type nat = Z | S of nat
```

In this question, treat OCaml as a pure, total language: every function terminates and has no side effects. Assume also that encode and decode can be given the types

```
val encode : nat -> ('a. ('a -> 'a) -> 'a -> 'a)
val decode : ('a. ('a -> 'a) -> 'a -> 'a) -> nat
```

which cannot be written directly in OCaml.

- (a) Write the number 3 as an OCaml datatype of type nat


```
val three : nat
```
- (b) Write a fold function for nat, with signature


```
val fold : nat -> ('a -> 'a) -> 'a -> 'a
```
- (c) Let f be a function of type $'a \rightarrow 'a$ and x be some arbitrary value of $'a$. What does `fold three f x` evaluate to?
- (d) Explain what `fold` does to each `S` and `Z`.
- (e) Show there is a function `encode` such that `encode three = e`, where e is your answer to [Part 7\(c\)](#), abstracted over f and x .
- (f) Show there is a function `decode` such that `decode e = three`, where e is your answer to [Part 7\(c\)](#), abstracted over f and x .

- (g) Explain what your answers to [Part 7\(e\)](#) and [Part 7\(f\)](#) imply about the relationship between naturals and folds over naturals.
- (h) Explain the relationship between folds over naturals and the Church encoding of naturals given in the notes.

QUESTION 8

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

- (a) Define a fold for booleans.

```
val fold : bool -> 'a -> 'a -> 'a
```

HINT A fold over type T is a general way to use / consume type T .

- (b) Using a similar procedure to [Part 7\(e\)](#) of [Question 7](#), encode true and false as functions of type $'a \rightarrow 'a \rightarrow 'a$.
- (c) Compare your answer to the encoding of booleans in the notes.

QUESTION 9

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

- (a) Write a fold (fold right) for lists of type t , where t is some concrete type.

```
type 'a list = Nil | Cons of 'a * 'a list
val fold : t list -> (t -> 'a -> 'a) -> 'a -> 'a
```

- (b) Using a similar procedure to [Part 7\(e\)](#) of [Question 7](#), encode Nil as a function of type $(t \rightarrow 'a \rightarrow 'a) \rightarrow 'a \rightarrow 'a$.
- (c) Using a similar procedure to [Part 7\(e\)](#) of [Question 7](#), and assuming xs is encoded as a function of type $(t \rightarrow 'a \rightarrow 'a) \rightarrow 'a \rightarrow 'a$, encode $\text{Cons}(x, xs)$ as a function of type $(t \rightarrow 'a \rightarrow 'a) \rightarrow 'a \rightarrow 'a$.
- (d) Derive the Church encoding of lists: Nil and Cons.

QUESTION 10

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

Let H , G , and G' be lambda terms. Assume

$$\forall A. H (G A) =_{\beta} G' (H A)$$

Prove that, for any natural n , for all lambda terms B

$$H (\underline{n} G B) =_{\beta} \underline{n} G' (H B)$$

QUESTION 11

CONCEPTUAL · 40 MIN

● ● ● ● ● CORE

- (a) Define a lambda term **Add** representing addition of Church numerals, **Add** $\underline{m} \underline{n} \equiv_{\alpha\beta\eta} \underline{m + n}$.
- (b) Define a lambda term **Times** representing multiplication of Church numerals, **Times** $\underline{m} \underline{n} \equiv_{\alpha\beta\eta} \underline{m \times n}$.
- (c) Define a lambda term **Exp** representing exponentiation of Church numerals, **Exp** $\underline{m} \underline{n} \equiv_{\alpha\beta\eta} \underline{m^n}$.

Optionally prove the correctness of your definitions.

You may take as given two important encodings, **Zero**, or $\underline{0}$, which encodes 0, and **Succ**, which encodes S .

$$\begin{aligned}\mathbf{Zero} &= \lambda f x. x \\ \mathbf{Succ} &= \lambda k f x. f (k f x)\end{aligned}$$

HINT It is easiest to first write it out as a fold on plain naturals, and then encode it piece by piece. There is a useful rule of thumb: **To encode a function defined by a fold, replace `fold n` by $\underline{n}$ and encode the other arguments.**

This is likely to not produce the simplest definition. You can use the law you proved in the previous question to simplify your initial definitions.

QUESTION 12

CONCEPTUAL · 30 MIN

● ● ● ● ● CORE

CORRECTION

Definition 32 of the notes reads as follows:

$f \in \mathbb{N}^n \rightarrow \mathbb{N}$ is λ -definable if there is a closed λ -term F that represents it: for all $(x_1, \dots, x_n) \in \mathbb{N}^n$ and $y \in \mathbb{N}$:

- if $f(x_1, \dots, x_n) = y$, then $F \underline{x_1} \cdots \underline{x_n} =_{\beta} \underline{y}$;
- if $f(x_1, \dots, x_n) \uparrow$, then $F \underline{x_1} \cdots \underline{x_n}$ has no β -normal form.

This definition is very slightly wrong: when $f(x_1, \dots, x_n)$ is undefined, we want $F \underline{x_1} \cdots \underline{x_n}$ to have no *head normal form*, not just no β -normal form.

For the following question, use the corrected definition:

$f \in \mathbb{N}^n \rightarrow \mathbb{N}$ is λ -definable if there is a closed λ -term F that represents it: for all $(x_1, \dots, x_n) \in \mathbb{N}^n$ and $y \in \mathbb{N}$:

- if $f(x_1, \dots, x_n) = y$, then $F \underline{x_1} \cdots \underline{x_n} =_{\beta} \underline{y}$;
- if $f(x_1, \dots, x_n) \uparrow$, then $F \underline{x_1} \cdots \underline{x_n}$ has no **head normal form**.

Consider the following λ -terms:

$$\mathbf{I} \triangleq \lambda x. x \quad \mathbf{C} \triangleq \lambda g f x. g (f x) \quad \mathbf{B} \triangleq \lambda g f x. f x \mathbf{I} (g (f x))$$

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

- (b) Assume partial functions $f, g: \mathbb{N} \rightarrow \mathbb{N}$ are represented by closed lambda terms F and G respectively. By choosing f and g , show that $\mathbf{C} G F$ does not represent $g \circ f$.
- (c) In this part, you may assume the following fact, and the result of [Part 6\(d\)](#).

FACT

If $M =_{\beta} N$, M has a head normal form if, and only if, N has a head normal form.

Assume partial functions $f, g: \mathbb{N} \rightarrow \mathbb{N}$ are represented by closed lambda terms F and G respectively. Show that their composition $(g \circ f)(x) \triangleq g(f(x))$ is represented by $\mathbf{B} G F$.

QUESTION 13

CONCEPTUAL · 40 MIN

● ● ● ● ● BASIC

- (a) Give a λ -term **Not** representing Boolean negation.
- (b) Give λ -terms **And** and **Or** representing Boolean conjunction and disjunction.
- (c) Give a λ -term **Minus** representing truncated subtraction (i.e. $\mathbf{Minus} \underline{m} \underline{n} \equiv_{\alpha\beta\eta} \underline{0}$ if $m < n$).
- (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.
- (e) Define the λ -term **UCr** that represents the uncurrying higher-order function: if $\langle M, N \rangle$ denotes **Pair** $M N$, then $\mathbf{UCr} F \langle M, N \rangle =_{\beta} F M N$.
- (f) Give a λ -term **MapPair** that applies a function to both elements of a pair: that is,

$$\mathbf{MapPair} F \langle M, N \rangle =_{\beta} \langle F M, F N \rangle.$$

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

QUESTION 14

CONCEPTUAL · 30 MIN

● ● ● ● ● CORE

- (a) Write the `triangle_g` function below as a lambda term, **TriangleG**. You may use the following encoded terms: **If**, **n**, **Eq₀**, **Add**, **Pred**.

```
1 let triangle_g f n =
2   if n = 0 then 0
3   else n + f (n - 1)
```

OCaml

- (b) What is the relationship between `triangle`, defined below, and `triangle_g`?

```
1 let rec triangle n =
2   if n = 0 then 0
3   else n + triangle (n - 1)
```

OCaml

- (c) Write `triangle` as a lambda term, **Triangle**.
 - (d) Given a lambda term M , what does $Y\ M$ compute, where Y is the Y combinator?
 - (e) Prove that **Triangle** $\underline{3} =_{\beta} \underline{6}$.
 - (f) Can you encode, directly, the Y combinator in OCaml, without any tricks? Why, or why not?²
-

²Assume the `-rectypes` flag is off.

Refresher — Solutions

Lent Term 2027 · Michael Lee

1 Countability

PREREQUISITES None · TIME 40 min

NOTE

Computation Theory relies heavily on the concepts of *countability* and *diagonalisation*, as taught in Part IA Discrete Mathematics.

This sheet is meant to refresh your knowledge. It is meant to be attempted over the Christmas vacation, and not meant to be submitted. Solutions have been provided.

QUESTION 1

CONCEPTUAL · 40 MIN

● ● ● ● ● BASIC

This question aims to refresh your memory of countability and diagonalisation arguments.

Recall from discrete mathematics the notion of a countable set. There are various definitions of countability, all equivalent.

1. A set S is countable if it is empty, or there exists a surjective function $\mathbb{N}$ to S .
2. A set S is countable if there exists an injective function S to $\mathbb{N}$.

An injective function $f : A \rightarrow B$ is one where $\forall a_1, a_2 \in A. f(a_1) = f(a_2) \implies a_1 = a_2$. A surjective function $g : A \rightarrow B$ is one where $\forall b \in B. \exists a \in A. g(a) = b$.

HINT The characterisation using injections is easier to work with when proving a true statement. The characterisation using surjections is easier to work with when disproving a false statement.

Prove or disprove the following statements.

- (a) Any finite set is countable.
- (b) $\mathbb{N}$, the set of natural numbers, is countable.
- (c) $\mathbb{Z}^+$, the positive integers, is countable.
- (d) $\mathbb{Z}$, the set of integers, is countable.
- (e) $\mathbb{N} \times \mathbb{N}$ is countable.
- (f) If A and B are countable, then $A \times B$ is countable.
- (g) For all natural numbers n , the iterated product $\prod_{i=1}^n A_i$ is countable if each A_i is countable.
- (h) $\mathbb{Q}$, the set of rational numbers, is countable.
- (i) If A and B are countable, then $A + B$ is countable.
- (j) For all natural numbers n , the iterated sum $\sum_{i=1}^n A_i$ is countable if each A_i is countable.

- (k) If A is countable, then the inductively defined set $\text{List } A$ is countable. *A definition of $\text{List } A$ is given in the Foundations of Computer Science exercises.*
- (l) If A is countable, then $B \subseteq A$ is countable.
- (m) If A is countable, then the powerset of A , $\mathcal{P}(A)$, is countable.
- (n) If A and B are countable, then the set of functions $A \rightarrow B$ is countable.
- (o) $\mathbb{R}$, the set of real numbers, is countable.

SOLUTION

- (a) This statement is true. Assume S is finite, thus, $|S| = n$ for some natural number n . Label the elements of S $s_0, s_1, s_2, \dots, s_{n-1}$. Define the function

$$f(s_m) = m$$

f is clearly injective.

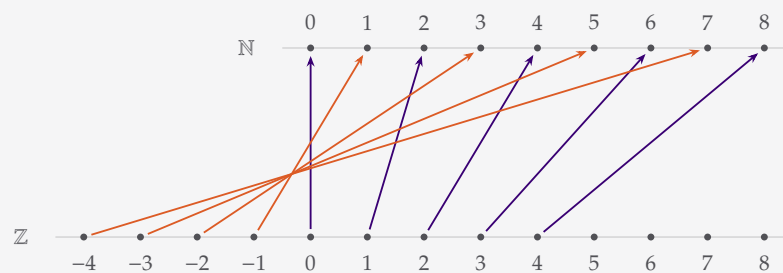
- (b) This statement is true. The identity function is an injective mapping $\mathbb{N} \rightarrow \mathbb{N}$.
- (c) This statement is true. The encoding function is

$$\begin{array}{ll} \text{pred} & : \mathbb{Z}^+ \rightarrow \mathbb{N} \\ \text{pred}(n) & \mapsto n - 1 \end{array}$$

- (d) This statement is true. Consider the function

$$f(z) = \begin{cases} 2z, & z \geq 0 \\ -2z - 1, & z < 0 \end{cases}$$

Graphically, it looks as follows:



This function is obviously injective.

- (e) This statement is true. Consider traversing the $\mathbb{N} \times \mathbb{N}$ grid diagonally.

$p \backslash q$	0	1	2	3	4
0	0	1	3	6	10
1	2	4	7	11	16
2	5	8	12	17	23
3	9	13	18	24	31
4	14	19	25	32	40

This corresponds to the encoding

$$\text{triangle}(p + q) + p$$

where $\text{triangle}(k) = \frac{k(k+1)}{2}$ is the k^{th} triangular number.

This encoding is injective (actually bijective). Assume

$$\text{triangle}(p + q) + p = \text{triangle}(p' + q') + p'$$

Notice that

$$\text{triangle}(p + q) \leq \text{triangle}(p + q) + p < \text{triangle}(p + q + 1)$$

Given any number n , there exists a unique greatest triangle number at most n . Call it $\Delta(n)$. Notice if n is triangular, then $n = \Delta(n)$. Then we can determine that

$$\begin{aligned}
 p &= \text{triangle}(p + q) + p - \text{triangle}(p + q) \\
 &= \text{triangle}(p + q) + p - \Delta(\text{triangle}(p + q) + p) \\
 &= \text{triangle}(p' + q') + p' - \Delta(\text{triangle}(p' + q') + p') \\
 &= \text{triangle}(p' + q') + p' - \text{triangle}(p' + q') \\
 &= p'
 \end{aligned}$$

So $p = p'$, and $\text{triangle}(p + q) = \text{triangle}(p' + q')$. By the injectivity of triangle,

$$p + q = p' + q'$$

and since $p = p'$, $q = q'$. The corresponding (surjective) decoding is

$$f(n) = (m, w - m)$$

where $m = n - \frac{w(w+1)}{2}$, and

$$w = \left\lfloor \frac{\sqrt{8n+1}-1}{2} \right\rfloor$$

An alternative encoding is

$$2^p \times (2q + 1) - 1$$

This encoding is also injective.

The corresponding decoding is

$$f(n) = \left(v_2(n+1), \frac{(n+1)/2^{v_2(n+1)} - 1}{2} \right)$$

where $v_2(n)$ is defined as the largest natural number m such that $2^m \mid n$ (divides n).

- (f) This statement is true. Since A and B are countable, there are injective functions $f : A \rightarrow \mathbb{N}$ and $g : B \rightarrow \mathbb{N}$. Thus, there is an injective function

$$f \times g : A \times B \rightarrow \mathbb{N} \times \mathbb{N}$$

given by

$$(f \times g)(a, b) = (f(a), g(b))$$

By the answer to the last question, there is an injective function

$$h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

Since the composition of injective functions is injective, we have our desired encoding

$$h \circ (f \times g)$$

- (g) This statement is true. Proof is by induction. The base case is [Part 1\(a\)](#). The inductive case is [Part 1\(f\)](#).

- (h) This statement is true. A rational number has the form $\frac{p}{q}$, where $p \in \mathbb{Z}$ and $q \in \mathbb{Z}^+$, and p and q are co-prime.

Thus, there is an injection

$$f : \mathbb{Q} \rightarrow \mathbb{Z} \times \mathbb{Z}^+$$

Since $\mathbb{Z}$ and $\mathbb{Z}^+$ are both countable, there exists an injective function

$$g : \mathbb{Z} \times \mathbb{Z}^+ \rightarrow \mathbb{N}$$

Since the composition of injective functions is injective, our encoding function is

$$g \circ f$$

- (i) This statement is true. Assume A is countable, that is, there exists an injection $f : A \rightarrow \mathbb{N}$. Assume B is countable, that is, there exists an injection $g : B \rightarrow \mathbb{N}$. Then consider the function

$$h(x) = \begin{cases} 2f(a), & x = \iota_1(a) \\ 2g(b) + 1, & x = \iota_2(b) \end{cases}$$

- (j) This statement is true. One can either prove it via induction, as before, or by constructing the encoding directly. Since there are n countable sets, $A_1 \dots A_n$, let $f_i : A_i \rightarrow \mathbb{N}$ be an injective encoding function for each A_i . We can then construct the encoding

$$h(x) = \begin{cases} n \times f_1(a_1), & x = \iota_1(a_1) \\ n \times f_2(a_2) + 1, & x = \iota_2(a_2) \\ \dots \\ n \times f_n(a_n) + (n - 1), & x = \iota_n(a_n) \end{cases}$$

which is obviously injective.

- (k) This statement is true.

Since A is countable, there is an injective function $h : A \rightarrow \mathbb{N}$.

Recall also that we can construct an injective encoding of pairs of naturals $e : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

We aim to prove an injection from List A to the naturals $\mathbb{N}$.

Consider the function f defined recursively as

$$\begin{aligned} f([]) &= 0 \\ f(x :: xs) &= e(h(x), f(xs)) + 1 \end{aligned}$$

This function is injective.

ASSUMPTIONS

- A-1** $h : A \rightarrow \mathbb{N}$ is injective
- A-2** $e : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is injective

GOALS

- G-1** $\forall ys. f(xs) = f(ys) \implies xs = ys$

PROOF

Proof is by structural induction on the definition of lists.

CASE $xs = []$

WANT TO SHOW

- G-1** $\forall ys. f([]) = f(ys) \implies [] = ys$

PROOF

Then $f(xs) = 0$. Assume arbitrary ys . If $f(ys) = 0$, then ys must also be $[]$.

CASE $xs = x :: xs'$

INDUCTIVE HYPOTHESIS

- IH(xs')** $\forall ys. f(xs') = f(ys) \implies xs' = ys$

WANT TO SHOW

- G-1** $\forall ys. f(x :: xs') = f(ys) \implies x :: xs' = ys$

PROOF

Then $f(xs) = e(h(x), f(xs')) + 1$. Assume arbitrary ys such that $f(ys) = f(xs)$.

ys must be of the form $y :: ys'$, since $f([]) = 0 < f(xs)$. So $f(ys) = e(h(y), f(ys')) + 1$.

Since successor is injective, we know

- D-1** $e(h(y), f(ys')) = e(h(x), f(xs'))$

Since e is injective, by **A-2**, we know

- D-2** $h(y) = h(x)$
- D-3** $f(ys') = f(xs')$

Since h is injective, by **A-1**, we know

- D-4** $y = x$

By $\text{IH}(xs')$ instantiated at ys' , with **D-3**, we know

$$\text{D-5} \quad ys' = xs'$$

And so, by **D-4** and **D-5**, we know that $y :: ys' = x :: xs'$, as desired.

- (l) This statement is true. Assume A is countable.

There is thus an injection from A to the natural numbers $\mathbb{N}$, g . If B is a subset of A , there is an injection $f : B \rightarrow A$. The composition of injections is an injection, so $g \circ f$ is our desired function.

- (m) This statement is false. $\mathbb{N}$ is countable, but $\mathcal{P}(\mathbb{N})$ is not countable.

The disproof is by diagonalisation.

Assume, for the sake of diagonalisation, a surjective decoding

$$f : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N})$$

This means we can enumerate the subsets

$$S_0, S_1, \dots, S_i, \dots$$

Now consider the subset $\hat{S}$ constructed as follows: for each i ,

$$i \in \hat{S} \iff i \notin S_i$$

Since $\hat{S}$ is an element of $\mathcal{P}(\mathbb{N})$, there exists a natural k such that $S_k = \hat{S}$.

$$k \in \hat{S} \iff k \notin \hat{S}$$

This is a contradiction.

- (n) This statement is false. Assume A is $\mathbb{N}$ and B is $\mathbb{B}$, the booleans. $\mathbb{B}$ is finite, so also countable. $\mathbb{N} \rightarrow \mathbb{B}$ is in bijection with $\mathcal{P}(\mathbb{N})$, which is uncountable. So $\mathbb{N} \rightarrow \mathbb{B}$ is also uncountable.

- (o) This statement is false. First, we show that the reals in range $(0, 1)$ are uncountable. This is proved via a diagonalisation argument. Once we have proved $(0, 1)$ is uncountable, we simply note an injection from $(0, 1)$ into $\mathbb{R}$, and then we are done.

Assume the reals in range $(0, 1)$ are countable, then we can list them

$$a_0, a_1, a_2, \dots$$

Further, every a_i is of the form $0.d_{i0}d_{i1}d_{i2}\dots$. We can thus expand the enumeration

$$\begin{array}{ccccccc} d_{00} & d_{01} & d_{02} & d_{03} & \dots & & \\ d_{10} & d_{11} & d_{12} & d_{13} & \dots & & \\ d_{20} & d_{21} & d_{22} & d_{23} & \dots & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \end{array}$$

Now consider the real number $\hat{d}$ defined such that the i^{th} digit of $\hat{d}$ is

$$\hat{d}_i = \begin{cases} 5, & d_{ii} \neq 5 \\ 4, & d_{ii} = 5 \end{cases}$$

Note that we choose 4 and 5 such that every syntactic choice must produce a different number. There are cases where two syntactic numbers have the same semantics, for example, $1 = 0.9999999999999999 \dots$

$\hat{d}$ must be given some number by the enumeration, call it k . Consider the k^{th} digit of $\hat{d}$, $\hat{d}_k$:

$$\begin{array}{rcl} \hat{d}_k & = & d_{kk} \\ & \neq & d_{kk} \end{array}$$

This is a contradiction.

PICKED UP IN

PART IA Discrete Mathematics · *countable sets; diagonalisation*

PART IA Foundations of Computer Science · *lists*