COMPUTATION THEORY

Supervision 1

Lent Term 2027 · Michael Lee

●●●○○ difficulty · ★ optional

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,

```python
def g():
  return 3
  h() # this code is dead
```

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

```
reg r0, r1 := 3, r2 := 4, r3

l0 : r1- -> l1, l6
l1 : r2- -> l2, l4
l2 : r0+ -> l3
l3 : r3+ -> l1
l4 : r3- -> l5, l0
l5 : r2+ -> l4
l6 : 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

```
machine AddBy(x, y) = struct
  reg t
  l0 : y- -> l1, l3
  l1 : x+ -> l2
  l2 : t+ -> l0
  l3 : t- -> l4, l5
  l4 : y+ -> l3
  l5 : halt
end

reg r0, r1 := 3, r2 := 4

l0 : r1- -> l1, l2
l1 : call AddBy(r0, r2) -> l0
l2 : 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

```
machine AddBy(x, y) = struct
  reg t
  l0 : y- -> l1, l3
```

```
    l1 : x+ -> l2
    l2 : t+ -> l0
    l3 : t- -> l4, l5
    l4 : y+ -> l3
    l5 : halt
  end

  machine Mul(z, x, y) =
    for x do AddBy(z, y) done

  reg r0, r1 := 3, r2 := 4

  l0 : call Mul(r0, r1, r2) -> l1
  l1 : halt
```

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

(i) ℓ : JUMP ℓ', which performs an unconditional jump to an instruction ℓ'

(ii) ℓ : CLEAR $r \longrightarrow \ell'$, 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.*

```
machine Copy(d, s) = struct
  reg t
  l0 : d- -> l0, l1
  l1 : s- -> l2, l4
  l2 : d+ -> l3
  l3 : t+ -> l1
  l4 : t- -> l5, l6
  l5 : s+ -> l4
  l6 : halt
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

```
l1 : call Copy(r3, r2) -> l2
```

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 `call`s to defined machines, into the extended register machine language with `JUMP`s and `CLEAR`s. 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.

```rm
machine ConstOne(r0) = struct
  reg r
  l0 : r+ -> l1
  l1 : r0- -> l1, l2
  l2 : call Copy(r0, r) -> l3
  l3 : halt
end
```

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

```rm
reg r0, r1 := 2

l0 : r1- -> l1, l2
l1 : call ConstOne(r0) -> l0
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

```rm
machine Mul(z, x, y) = struct
  machine AddBy(x, y) = struct
    reg t
    l0 : y- -> l1, l3
    l1 : x+ -> l2
    l2 : t+ -> l0
    l3 : t- -> l4, l5
    l4 : y+ -> l3
    l5 : halt
  end
```

```
RM
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,

```
RM
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 the **Register Machine Library** in the **Standard Libraries** sheet.*

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 (the **Register Machine Library** in the **Standard Libraries** sheet).

(a) First projection function $p \in \mathbb{N}^2 \to \mathbb{N}$, 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, _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, _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

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

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 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 prog(n), the program obtained by decoding n. You can use instruction(m) to mean the instruction obtained by decoding m. Describe also ϕ_n, the unary partial function $\mathbb{N} \rightharpoonup \mathbb{N}$ computed by 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 $f : \mathbb{N} \rightharpoonup \mathbb{N}$, 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 $\{a, b\}^*$:

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

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:

$$\frac{}{n : \mathbb{N}}\ \text{(T-Nat)} \qquad \frac{}{s : \mathbb{S}}\ \text{(T-Str)} \qquad \frac{e_1 : \mathbb{N} \quad e_2 : \mathbb{N}}{e_1 \circ e_2 : \mathbb{N}}\ \text{(T-Circ)} \qquad \frac{e_1 : \mathbb{S} \quad e_2 : \mathbb{S}}{e_1 \star e_2 : \mathbb{S}}\ \text{(T-Cat)}$$

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 $\mathbb{S}$:

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

Strings are encoded as

$$\ulcorner \texttt{""} \urcorner = 0, \quad \ulcorner a :: s \urcorner = 2\ulcorner s \urcorner + 1, \quad \ulcorner b :: s \urcorner = 2\ulcorner s \urcorner + 2$$

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

(a) An expression `"aba"` ⋆ `"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

$$[\texttt{"aba"};\star;\texttt{"bab"}]$$

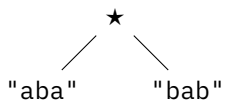
(i) Find a Gödel encoding for lists of tokens that establishes a bijection between lists of tokens and naturals. You do not need to prove that it is bijective.

(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:

$$\ulcorner n \urcorner_{\mathcal{L}} = 4n \qquad \ulcorner e_1 \circ e_2 \urcorner_{\mathcal{L}} = 4\langle \ulcorner e_1 \urcorner_{\mathcal{L}}, \ulcorner e_2 \urcorner_{\mathcal{L}} \rangle + 2$$
$$\ulcorner s \urcorner_{\mathcal{L}} = 4\ulcorner s \urcorner + 1 \qquad \ulcorner e_1 \star e_2 \urcorner_{\mathcal{L}} = 4\langle \ulcorner e_1 \urcorner_{\mathcal{L}}, \ulcorner e_2 \urcorner_{\mathcal{L}} \rangle + 3$$

For example,

$$\ulcorner \texttt{"aba"} \urcorner_{\mathcal{L}} = 4(9) + 1 = 37$$
$$\ulcorner \texttt{"bab"} \urcorner_{\mathcal{L}} = 4(12) + 1 = 49$$
$$\ulcorner \texttt{"aba"} \star \texttt{"bab"} \urcorner_{\mathcal{L}} = 4\langle 37, 49 \rangle + 3 = 2^{39} \times 99 - 1$$

(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

$$\dfrac{\dfrac{}{\texttt{"aba"} : \mathbb{S}}\ (\text{T-Str}) \qquad \dfrac{}{\texttt{"bab"} : \mathbb{S}}\ (\text{T-Str})}{\texttt{"aba"} \star \texttt{"bab"} : \mathbb{S}}\ (\text{T-Cat})$$

(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.

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

For example,

$$[\![\texttt{"aba"} \star (\texttt{""} \star \texttt{"bab"})]\!]_{\mathbb{S}} = [\texttt{"aba"}] +\!\!+ ([] +\!\!+ [\texttt{"bab"}]) = [\texttt{"aba"};\texttt{"bab"}]$$

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

(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 $\ulcorner e \urcorner_{\mathcal{L}/=}$ and $\ulcorner e' \urcorner_{\mathcal{L}/=}$?

(v) If $\ulcorner e \urcorner_{\mathcal{L}/=} = n = \ulcorner e' \urcorner_{\mathcal{L}/=}$, 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,

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

(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 $\ulcorner e \urcorner_{\mathcal{L}}$. Write an OCaml function `splice` that takes $\ulcorner e \urcorner_{\mathcal{L}}$ and returns $\ulcorner e \circ 4 \urcorner_{\mathcal{L}}$. *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 $\phi_e : \mathbb{N}^2 \rightharpoonup \mathbb{N}$. 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 $S : \mathbb{N}^2 \to \mathbb{N}$ 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 $prog(e)$, for decoding an encoded program, and $\ulcorner e \urcorner$, for encoding a program, are given. Assume also $copy(R, R')$, which copies the contents of R' to R, and $const_n(R)$, 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 $\phi_e : \mathbb{N}^{m+n} \rightharpoonup \mathbb{N}$. 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 $S^m_n : \mathbb{N}^{1+m} \to \mathbb{N}$ such that

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

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 $\phi_{S^m_n(e,\vec{x})}$ 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:

```OCaml
let square x = x * x
let cube x = x * x * x
```

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`:

```OCaml
let rec pow n x =
  if n = 0 then 1
  else x * pow (n-1) x

let square = pow 2
let cube = pow 3
```

This approach is more maintainable: it easily extends to powers $4, 5, \ldots$. 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
macro pow (n : int) (x : int expr) =
  if n = 0 then <<1>>
  else << $x * $(pow (n-1) x) >>

let square x = $(pow 2 <<x>>)  (* executes at compile-time,
                                  generates x * (x * 1) *)
```

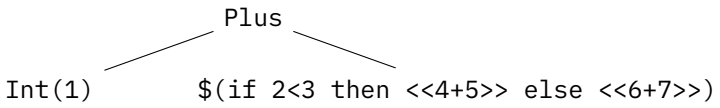
At the term level, MacoCaml extends OCaml in three ways:

1. An `'a expr` type. For example, `int expr` represents encodings of OCaml `int`s.
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>>)>>
```

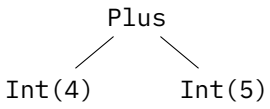
We first see the outer quote, so we switch into encoding mode, and encode `+` as the AST node `Plus`, and `1` as the AST node `Int(1)`.



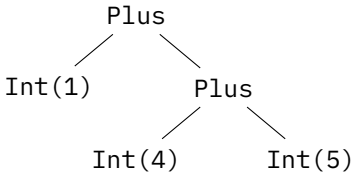
When we try to encode the right subtree, we see a splice, so we stop encoding, and treat the `if` statement as regular OCaml code. Running the OCaml code should give us an AST that we can splice in as the right subtree, much like you did in Part 13(a).

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 `<<1+(4+5)>>`. When we run this at runtime, we will get `10`.

(a) Identify the metalanguage and object language in MacoCaml.

(b) Explain how `pow 2 <<x>>` generates `x * (x * 1)` when run.

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

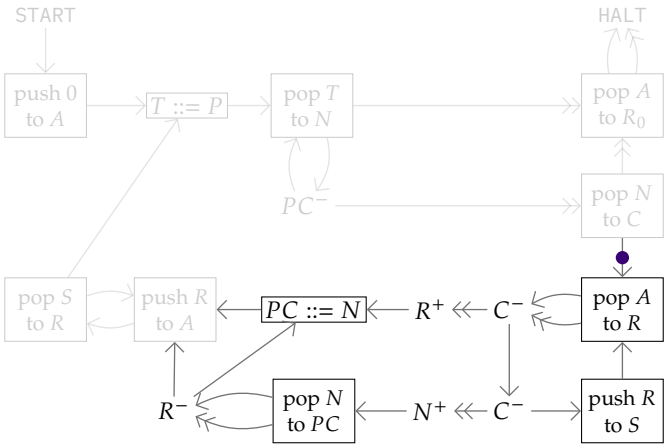
$$(f \oplus g)(x) = \begin{cases} 0, & f(x) = 0 \ \vee \ g(x) = 0 \\ 1, & f(x) = 1 \ \wedge \ g(x) = 1 \\ \uparrow, & \text{otherwise} \end{cases}$$

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

QUESTION 17

CONCEPTUAL · 30 MIN ●●●●● **CORE**

Consider starting the universal register machine at the marked point. Consider only the widget in the universal register machine that is not faded.



Assume register A is set to the encoding of $[1;2;3;0;0;0]$. Assume register C holds the natural 7, and register N holds the natural 15. Assume S is set to 0.

We do not care about any other register.

(a) What instruction should the universal register machine execute, and on which register?

(b) Explain carefully how the universal register machine decides which instruction it needs to execute, and finds the register on which it should execute the instruction.

(c) What label should the program jump to after executing the instruction?

(d) Explain carefully how the universal register machine executes the instruction, and determines which label to jump to.

(e) Immediately after applying the instruction to register R, what lists are stored in A, R, and S?

(f) Repeat the analysis in Part 17(d), but with A initially set to $[1;2;3;2;0;0]$.

QUESTION 18

CONCEPTUAL · 50 MIN ●●●○○ **CORE**

Trace the action of the universal register machine when register P is set to 24 and register A is set to $1 = \ulcorner[0]\urcorner$.

You do not have to give every step of the register machine (if you do, note that there are around 780 steps, so I won't mark it).

The point of this exercise is to familiarise yourself with how the machine runs, so give enough detail to show the execution of the machine.