Lecture 7

Exercise 1
(a) Define $\text{proj}_n$, $\text{succ}$ and $\text{zero}$.
(b) Show that all of these are RM computable.

Exercise 2
(a) What is Kleene equivalence of two expressions?
(b) Define composition of multi-dimensional partial functions $f \in \mathbb{N}^n \to \mathbb{N}$ and $g_1, \ldots, g_n \in \mathbb{N}^m \to \mathbb{N}$.
(c) Show that composition of RM computable functions is RM computable.

Lecture 8

Exercise 3
(a) Define primitive recursion (See [2017P6Q4 (a)], [2014P6Q4 (a)], [1999P4Q1 (a)]).
(b) Define primitive recursive functions $\text{PRIM}$ (See [2017P6Q4 (b)(i)], [2014P6Q4 (b)], [2011P6Q4 (a)], [2006P4Q9 (a)], [1995P4Q9 (a)]).
(c) Prove that PRIM functions are total (See [2006P4Q9 (b)]). Deduce that there exist computable functions that are not PRIM.
(d) Are all total functions primitive recursive? (See [2017P6Q4 (b)(iii)])
(e) Show that the functions add, pred, mult, tsub, exp are primitive recursive (See [2014P6Q4 (c)], [2006P4Q9 (c)]).
(f) Show that the following functions are primitive recursive:
   i. $\text{Eq}_0(x, y, z) = \begin{cases} y & \text{if } x = 0 \\ z & \text{otherwise.} \end{cases}$
   ii. The bounded summation function for $g : \mathbb{N}^n+1 \to \mathbb{N}$ and $f : \mathbb{N}^{n+1} \to \mathbb{N}$,
      
      
      $$g(\vec{x}, x) = \begin{cases} 0 & \text{if } x = 0 \\ f(\vec{x}, 0) & \text{if } x = 1 \\ f(\vec{x}, 0) + \ldots + f(\vec{x}, x - 1) & \text{if } x > 1 \end{cases}$$
(g) Show that the functions square($x$) = $x^2$ and fact($x$) = $x!$ are primitive recursive functions (See [2011P6Q4 (c)])

Exercise 4 [RMs implement PRIM] Show that primitive recursion is implementable in RMs. Deduce that PRIM functions are computable.

Exercise 5 [Minimisation]
(a) Define minimisation.
(b) Why might we want to define minimisation?
(c) Implement div.
(d) Show that minimisation is implementable using RMs.
Exercise 6
(a) Define partial recursive (PR) functions. (See [2018P6Q5 (a)], [2016P6Q3 (a)], [2006P4Q9 (d)], [1995P4Q9 (a)]).
(b) Show that PR functions are RM computable. (See [2016P6Q3 (b)], [1999P4Q1 (b)])
(c) Describe in high-level terms why every computable function is also PR (See [1995P4Q9 (b),(c)]).

Exercise 7 Attempt [2018P6Q5].

Exercise 8 Attempt [2014P6Q4 (e)].

Exercise 9 (first part)

Exercise 9
(a) Define the Ackermann function.
(b) In what sense does it grow faster than any primitive recursive function?
(c) (optional - advanced) Read this proof for the Ackermann's function growing faster than any primitive recursive function.

Exercise 10 Attempt [2001P4Q8].

Exercise 11 [ack is RM computable] Recall the definition of Ackermann’s function $\text{ack}(x_1, x_2)$ (slide 102). Sketch how to build a register machine $M$ that computes $\text{ack}(x_1, x_2)$ in $R_0$ when started with $x_1$ in $R_1$ and $x_2$ in $R_2$ and all other registers zero. [$\text{Hint:}$ here’s one way; the next question steers you another way to the computability of $\text{ack}$. Call a finite list $L = [(x_1, y_1, z_1), (x_2, y_2, z_2), ...]$ of triples of numbers suitable if it satisfies

- if $(0, y, z) \in L$, then $z = y + 1$
- if $(x + 1, 0, z) \in L$, then $(x, 1, z) \in L$
- if $(x + 1, y + 1, z) \in L$, then there is some $u$ with $(x + 1, y, u) \in L$ and $(x, u, z) \in L$.

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

[Exercise 9 in Lecturer's handout]

Exercise 12 Give an example of a function that is not in PRIM. (See [2014P6Q4 (d)])

Exercise 12

Lecture 9 (second part)

Further reading:

- Foundations of Functional Programming
Exercise 13
(a) How are $\lambda$-terms defined? (See [2016P6Q4 (a)])
(b) What notational conventions do we follow?
(c) **Exercise 1.4 in Hindley and Seldin (2008)** Insert the full amount of parentheses in the following abbreviated terms:
   i. $xyz(yx)$,
   ii. $\lambda x.uxy$,
   iii. $\lambda u.u(\lambda x.y)$,
   iv. $ux(yz)(\lambda v.vy)$,
   v. $(\lambda xyz.xz(yz))uvw$,
   vi. $w(\lambda xyz.xz(yz))uv$.
(d) What does $x \# M$ mean?
(e) What do the terms *bound variable*, *body*, *binding*, *bound*, *free*, $\text{FV}(\cdot)$, $\text{BV}(\cdot)$ and *closed term* mean?
(f) Determine the free variables and bound variables in the following expressions:
   i. $\lambda u.\lambda y.\lambda u.\lambda y.u$.
   ii. $(\lambda x\lambda u.y)((xx)x)((vy)\lambda u.u)$.
   iii. $(((\lambda z.z)\lambda y.\lambda v.v)(\lambda v.\lambda y.v)$.
   iv. $(\lambda x.(xx)u)((\lambda z.z)\lambda y.y))$.
   v. You can generate more practice questions [here].

Exercise 14 **[$\alpha$-equivalence]**
(a) Intuitively, what does $\alpha$-equivalence try to capture?
(b) Define what $M\{z/x\}$ means.
(c) Define formally $\alpha$-equivalence.
(d) Show that the following pairs are $\alpha$-equivalent:
   i. $A \equiv \lambda xy.x(xy)$ and $B \equiv \lambda uv.u(vw)$,
   ii. $A \equiv (\lambda x\lambda y.y)(xx)(\lambda k.k)\lambda x.yx$ and $B \equiv (\lambda k\ell.m.(mk(\lambda a.a)))(\lambda x.\lambda y.xy)$.
(e) (optional) Show that $\alpha$-equivalence is an equivalence relation.

Lecture 10

Exercise 15 **[Substitution]**
(a) Define the substitution operation $N[M/x]$.
(b) **Exercise 1.14 in Hindley and Seldin (2008)** Evaluate the following substitutions:
   i. $(\lambda y.x(\lambda w.v w x))(u v)/x$
   ii. $(\lambda y.x(\lambda x.x))(\lambda y.y)/x$
   iii. $(y(\lambda v.x v))(\lambda v.vy)/x$
   iv. $(\lambda x.z y)(u v)/x$

Exercise 16
(a) Define one-step $\beta$-reduction.
(b) Define the many-step $\beta$-reduction. (See [2015P6Q4 (a)])
(c) Define the $\beta$-conversion. (See [2019P6Q6 (a)(i)])
Exercise 17
(a) State the Church-Rosser Theorem and prove its corollary.
(b) Attempt [2019P6Q6 (a)(iii)].

Exercise 18
(a) Define the $\beta$-normal form. (See [2019P6Q6 (a)(ii)], [2013P6Q4 (a)(i)]
(b) What properties does this form have?
(c) Do all terms have a $\beta$-normal form? (See [2013P6Q4 (a)(iii)])
(d) Show that there exists $\lambda$-terms that have both a $\beta$-normal form and an infinite chain of reductions from it.
(e) **Exercise 1.28 in Hindley and Seldin (2008)** Find the $\beta$-normal form for the following terms (if it exists):
   i. $(\lambda x.x(xy))z$,
   ii. $(\lambda x.y)z$,
   iii. $(\lambda x.(\lambda y.xy))zv$,
   iv. $(\lambda x.xx)(\lambda x.xy)$,
   v. $(\lambda x.xy)(\lambda u.vu)$,
   vi. $(\lambda x.(x(yz))x)(\lambda u.uv)$,
   vii. $(\lambda xy.xy)(\lambda u.uv)$,
   viii. $(\lambda xyz.xz)((\lambda xy.yx)u)((\lambda xy.yx)v)w$.

Exercise 19
(a) Define normal-order reduction.
(b) Is it similar to call-by-name?
(c) Is there an evaluation analogous to call-by-value? Which one is preferred?

Exercise 20 [Lambda functions in OCaml] (optional) In this exercise, you will implement $\beta$-reduction for lambda terms in OCaml.
(a) Define a type for lambda terms in OCaml.
(b) Define the function `substitute n m x` that replaces all occurrences of variable x with m inside n.
(c) Define the function `single_step_reduce m` that returns $(m', \text{reduced})$ the reduced term (or the original term) and whether a reduction was applied.
(d) Define the function `multi_step_reduce m` that calls `single_step_reduce` until `reduced` is false. Verify the reduction works as expected by applying it on the above examples.

Lecture 11

Exercise 21
(a) Define Church’s numerals. (See [2020P6Q6 (a)], [2016P6Q4 (b)], [2010P6Q4 (a)])
(b) What is the difference between $ffx$ and $f(f(x))$?
(c) Show that $n M N =_\beta M^n N$.
(d) Prove that $(\lambda x_1 x_2 M x_1 f(x_2 f x)) n m$ represents addition.

Exercise 22 Define $\lambda$-definable functions. (See [2020P6Q6 (c)], [2018P6Q6 (c)], [2010P6Q4 (b)])
Exercise 23
(a) Show that proj, succ and zero are λ-definable. (See [2020P6Q6 (d)], [2010P6Q4 (c)])
(b) Show how to represent composition. What is the problem here? (See [2013P6Q4 (b)(ii),(iii)])
(c) Define λ-terms for True, False and If. (See [2020P6Q6 (b)], [2019P6Q6 (b)])
(d) Prove that If True MN ≡β M and If False M N≡β N.
(e) Define λ-terms for And, Or and Not.
(f) Show that testing for equality with 0 is λ-definable.
(g) Define λ-terms for Pair, Fst and Snd. Show that Fst (Pair M N)≡β M (See [2020P6Q6 (e)]).
(h) Define the pred function and prove by induction that it works.
(i) Attempt [2020P6Q6 (f),(g)].
(j) Attempt [2016P6Q4 (c)].

Exercise 24 If you are still not fed up with Ackermann’s function ack ∈ N^2 → N, show that the λ-term
Ack ≜ λx.x (λfy.y f (f 1)) Succ represents ack (where Succ is as on slide 123).
[Exercise 11 in Lecturer’s handout]

Exercise 25 Give a definition of a function that is λ-definable but not primitive recursive. [2011P6Q4 (d)].

Exercise 26 Attempt [2010P6Q4 (d)].

Exercise 27 [Correct composition] Let I be the λ-term λx.x.
(a) Show that n I≡β I holds for every Church numeral n.
(b) Now consider B ≜ λfgx.g x I (f (g x)). Assuming the fact about normal order reduction mentioned on L10S35, show that if partial functions f, g ∈ N → N are represented by closed λ-terms F and G respectively, then their composition (f o g)(x)≡ f(g(x)) is represented by B F G.
(c) How does this solve the problem mentioned on L11S14?
[Exercise 12 in Lecturer’s handout]

Lecture 12

Exercise 28
(a) Why do we need the fixed point combinator in showing that primitive recursion is λ-definable? How is it used?
(b) Define Curry’s fixed point combinator Y and show that it satisfies the desired property.
(c) Define Turing’s combinator and show that it satisfies the desired property. (See [2015P6Q4 (c)])
(d) Attempt [2019P6Q6 (d),(e)].
(e) Show that the square and fact are λ-definable. (See [2011P6Q4 (c)])

Exercise 29
(a) Explain how fixed-point combinators are used in the λ-definition of minimisation.
(b) Deduce that every total recursive function is λ-definable. Collect the arguments and make an outline of the proof.
**Exercise 30** Give a high-level argument for why every $\lambda$-definable function is RM computable. (See [2018P6Q6 (d)])

**Exercise 31** Describe the *Church-Turing thesis*. Why is this not called a theorem? What examples did you come across in the lectures?