Computation Theory

Supervision 1 – Solutions

1. Algorithmically undecidable problems

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

   Prove or disprove the following statements:

   a) The set of natural numbers is countable.

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

   b) The set of integers is countable.

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

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

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

   d) The set of rational numbers is countable.

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

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

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

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

> Polynomials are finite sums of terms consisting of a natural power of the variable and a countable (e.g. rational) coefficient. A polynomial $a_nx^n+a_{n-1}x^{n-1}+\cdots+a_2x^2+a_1x+a_0$ of degree n (meaning that the highest exponent of the variable is n) can therefore be represented as an n-tuple of the coefficients $(a_n, a_{n-1}, \ldots, a_1, a_0)$, with $a_k = 0$ if a term of degree k does not appear in the polynomial. Thus, the set of all polynomials of degree n is isomorphic to the n-ary Cartesian product of the set of coefficients. If this set is countable, the product will also be countable from part (e) above.

g) The powerset of a countable set is countable.

> This is false: there is no surjection from $\mathbb{N}$ to $\mathcal{P}(S)$ for a countable S, i.e. there are infinite sets that are "more infinite", than the countably infinite set of natural numbers. This deep result was established by Georg Cantor in 1891 using his famous *diagonal argument*, which has since been applied to many other nonexistence proofs. A more general version of the statement is known as Cantor's Theorem:
>
> *There is no surjection $f : A \to \mathcal{P}(A)$ from a set A to its powerset.*
>
> The proof proceeds by contradiction. Assume f is a surjection, i.e. for every subset $S \subseteq A$ there is an $a \in A$ such that $f(a) = S$. In particular, consider the subset $D \triangleq \{x \in A \mid x \notin f(x)\}$ of elements $x \in A$ which are not in $f(x)$. Since f is surjective, there exists an associated $a \in A$ such that $f(a) = D$. But then $a \in D$ would imply that $a \notin f(a)$, and $a \notin D$ would imply that $a \in D$; but $a \in D \iff a \notin D$ is a contradiction, so the assumption that f is surjective was wrong.
>
> As a corollary of Cantor's theorem we get that there is no surjection $\mathbb{N} \to \mathcal{P}(\mathbb{N})$, so there are indeed countable sets whose powerset is not countable.
>
> The intuition behind the construction of the *diagonal set D* for the case of $\mathbb{N}$ is the following. The contradictive assumption $f : \mathbb{N} \twoheadrightarrow \mathcal{P}(\mathbb{N})$ states that $\mathcal{P}(\mathbb{N})$ is countable, so we have an exhaustive listing of all subsets of the natural numbers. However, for

> any such listing we can construct a set of naturals that cannot be in the listing, so it couldn't have been exhaustive. To construct this set, we look at whether $n \in \mathbb{N}$ occurs in the n^{th} set of the enumeration. If $n \notin f(n)$, we include n in D, otherwise we don't. Thus, by construction, every set in the enumeration will differ from D in at least one element: there may be a $k \in \mathbb{N}$ such that $f(k)$ is nearly identical to D, but they will certainly differ in their inclusion of k. This D can be constructed for any listing, so no listing can be exhaustive – $\mathcal{P}(\mathbb{N})$ is not countable.
>
> $$\boxed{\begin{array}{llccccccccccccl}
f(1) & = \{ & 1, & & & & & & & & & & & & \} \\
f(2) & = \{ & 1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10, & 11, & \dots & \} \\
f(3) & = \{ & & 2, & 3, & 4, & & 6, & & 8, & & 10, & & \dots & \} \\
f(4) & = \{ & 1, & & 3, & 4, & 5, & & 7, & & 9, & & 11, & \dots & \} \\
f(5) & = \{ & 1, & 2, & & 4, & 5, & 6, & 7, & & 9, & & 11, & \dots & \} \\
f(6) & = \{ & & & 3, & 4, & & 6, & 7, & & 9, & 10, & & \dots & \} \\
f(7) & = \{ & 1, & & & & 5, & & 7, & & 9, & & & \dots & \} \\
f(8) & = \{ & & & 3, & 4, & & & 7, & 8, & & & 11, & \dots & \} \\
f(9) & = \{ & 1, & 2, & & & 5, & 6, & & & 9, & 10, & & \dots & \} \\
f(10) & = \{ & 1, & 2, & & 4, & 5, & 6, & & & 9, & 10, & 11, & \dots & \} \\
f(11) & = \{ & 1, & 2, & & 4, & & 6, & & & 9, & & 11, & \dots & \} \\
\vdots & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots &
\end{array}}$$
>
> $$\boxed{\begin{array}{llccccccccccccl}
T & = \{ & 1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10, & 11, & \dots & \}
\end{array}}$$

h) The set of real numbers is countable.

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

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

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

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

$$\begin{array}{c|ccccc} & \ulcorner A_0 \urcorner & \ulcorner A_1 \urcorner & \ulcorner A_2 \urcorner & \ulcorner A_3 \urcorner & \cdots \\ A_0 & 1 & 0 & \times & 1 & \cdots \\ A_1 & \times & \times & 1 & \times & \cdots \\ A_2 & 1 & \times & 0 & 1 & \cdots \\ A_3 & 0 & 1 & \times & \times & \cdots \\ & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

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

$$\begin{array}{c|ccccc} & \ulcorner A_0 \urcorner & \ulcorner A_1 \urcorner & \ulcorner A_2 \urcorner & \ulcorner A_3 \urcorner & \cdots \\ A_0 & 1 & 1 & 0 & 1 & \cdots \\ A_1 & 0 & 0 & 1 & 0 & \cdots \\ A_2 & 1 & 0 & 1 & 1 & \cdots \\ A_3 & 1 & 1 & 0 & 0 & \cdots \\ & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

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

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

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

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

2. Register machines

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

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

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

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

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

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

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

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

As hinted in the question, we can save some effort by working at a higher level of abstraction rather than individual register operations. Specifically, we can create RM "combinators" corresponding to the three fundamental building blocks of programming: sequential composition, conditional branching and iteration. Any register machine is equivalent to one with a single halt state, so abstractly an RM program looks like

$$\begin{tikzcd} \text{START} \arrow[r] & M \arrow[r, Rightarrow] & \text{HALT} \end{tikzcd}$$

Then, we have ways of combining RMs with sequential composition

$$\begin{tikzcd} \text{START} \arrow[r] & M_1 \arrow[r, Rightarrow] & M_2 \arrow[r, Rightarrow] & \text{HALT} \end{tikzcd}$$

conditional branching with `if R = 0 then` M_1 `else` M_2

$$\begin{tikzcd} \text{START} \arrow[r] & \text{R}^- \arrow[r, twoheadrightarrow] \arrow[d] & M_1 \arrow[r, Rightarrow] & \text{HALT} \\ & \text{R}^+ \arrow[r] & M_2 \arrow[ur, Rightarrow] & \end{tikzcd}$$

and iteration with `while R ≠ 0 do` M

$$\begin{tikzcd} \text{START} \arrow[r] & \text{R}^- \arrow[r, twoheadrightarrow] \arrow[d] & \text{HALT} \\ & \text{R}^+ \arrow[r] & M \arrow[ul, Rightarrow] \end{tikzcd}$$

a) First projection: copy over R_1 to R_0.

$$\begin{tikzcd}
\text{START} \arrow[r] & R_1^- \arrow[r, two heads] \arrow[d, bend right] & \text{HALT} \\
& R_0^+ \arrow[u, bend right] &
\end{tikzcd}$$

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

$$\begin{tikzcd}
\text{START} \arrow[r] & R_0^+ \arrow[r] & \cdots \arrow[r] \arrow[phantom, ""{name=x}] & R_0^+ \arrow[r] & \text{HALT} \\
& & (n-2 \text{ times}) & &
\end{tikzcd}$$

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

$$\begin{tikzcd}
\text{START} \arrow[r] & R_2^- \arrow[r, two heads] \arrow[d, bend right] & R_1^- \arrow[r, bend right] \arrow[d, two heads] & R_0^+ \arrow[l, bend right] \\
& R_1^- \arrow[u, bend right] \arrow[r, two heads] & \text{HALT} &
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{START} \arrow[r] & R_1^- \arrow[r, two heads] \arrow[d, bend right] & R_2^- \arrow[r, two heads] \arrow[d, bend right] & \text{HALT} \\
& R_0^+ \arrow[u, bend right] & R_0^- \arrow[u, bend right] \arrow[ur, two heads] &
\end{tikzcd}$$

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

$$\begin{tikzcd}
\text{START} \arrow[r] & \boxed{\texttt{while } R_1 \neq 0 \texttt{ do}} \arrow[r] \arrow[d, two heads] & \boxed{R_1 \dot{-} R_2} \arrow[d] \\
& \text{HALT} & R_0^+ \arrow[ul]
\end{tikzcd}$$

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

$$\begin{tikzcd}
\text{START} \arrow[r] & \boxed{R_3 \ := \ R_1 \operatorname{div} R_2} \arrow[r] & \boxed{R_4 \ := \ R_3 \times R_2} \arrow[d] & \\
& & \boxed{R_0 \ := \ R_1 \dot{-} R_4} \arrow[r] & \text{HALT}
\end{tikzcd}$$

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

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

$$\begin{tikzcd}
\mathtt{START} \arrow[r] & R_0^+ \arrow[r] & \boxed{\mathtt{while}\ R_1 \neq 0\ \mathtt{do}} \arrow[r] \arrow[d, two heads] & \boxed{R_2\ \mathtt{:=}\ R_0 \times 2} \arrow[d] \\
& & \mathtt{HALT} & \boxed{R_0\ \mathtt{:=}\ R_2} \arrow[ul]
\end{tikzcd}$$

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

$$\begin{tikzcd}
\mathtt{START} \arrow[r] & R_0^+ \arrow[r] & R_1^- \arrow[r] \arrow[d, two heads] & R_0^- \arrow[r, two heads] \arrow[d] & R_2^- \arrow[ll, bend right, two heads] \arrow[r, bend left] & R_0^+ \arrow[l, bend left] \\
& & \mathtt{HALT} & R_2^+ \arrow[r] & R_2^+ \arrow[ul]
\end{tikzcd}$$

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

$$\begin{tikzcd}
\mathtt{START} \arrow[r] & \boxed{\mathtt{while}\ R_1 \neq 0\ \mathtt{do}} \arrow[r, two heads] \arrow[d] & R_0^- \arrow[r, bend left] \arrow[r, bend right, two heads] & \mathtt{HALT} \\
& \boxed{R_1\ \mathtt{:=}\ R_1\ \mathrm{div}\ 2} \arrow[r] & R_0^+ \arrow[ul]
\end{tikzcd}$$

3. Coding programs as numbers

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

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

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

   b) In the example of first-order logic above, is a particular Gödel numbering a bijection, an injection, a surjection, a total function, a partial function, or a relation? Justify your answer.

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

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

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

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

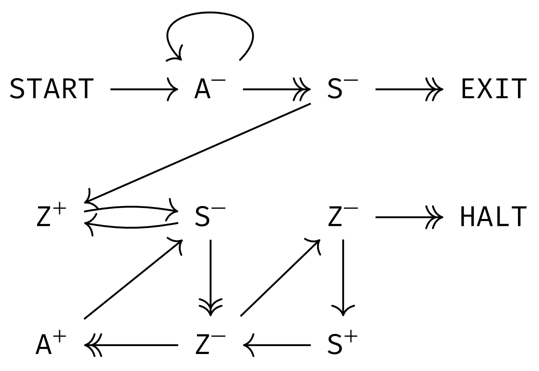
> This question shows that two functions with different codes can have the same behaviour, so while there is a bijection between the machine implementing the function and its code, there are infinitely many register machines implementing the same function. In practice, we can decode $f \in \mathbb{N}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

4. Universal register machine

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

> Th universal register machine U implements a register machine evaluator as a register machine. Its inputs are $\mathsf{R}_1 = e$ and $\mathsf{R}_2 = a$, where e is the code for a RM program, and a is the code for the list of arguments. The URM first decodes e as a program P, then decodes a as a list of register values $a_1, \ldots, a_n$, then executes the program P on the arguments $\mathsf{R}_1 = a_1, \ldots, \mathsf{R}_n = a_n$, storing the result in R_0.
>
> The implementation of the machine is similar to a rudimentary processor, keeping track of the currently executed instruction using a program counter and iteratively executing the commands on the arguments. A detailed, annotated analysis of the implementation can be found in the slides for Lecture 4, but it's worth analysing it yourself and getting a high-level grasp of its operation.

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



$$\begin{tikzcd}
\mathsf{START} \arrow[r] & \mathsf{A}^- \arrow[loop above] \arrow[r, two heads] & \mathsf{S}^- \arrow[r, two heads] \arrow[dll] & \mathsf{EXIT} \\
\mathsf{Z}^+ \arrow[r, bend left] & \mathsf{S}^- \arrow[l, bend left] \arrow[d, two heads] & \mathsf{Z}^- \arrow[r, two heads] \arrow[d] & \mathsf{HALT} \\
\mathsf{A}^+ \arrow[ur] & \mathsf{Z}^- \arrow[l, two heads] \arrow[ur] & \mathsf{S}^+ \arrow[l] &
\end{tikzcd}$$

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

Optional exercise

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