# yz709-CT-sup1



# Question 1

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

(a) First projection function $p \in \mathbb{N}^2 \to \mathbb{N}$, where $p(x, y) \triangleq x$

(b) Constant function with value $n \in \mathbb{N}$, $c \in \mathbb{N} \to \mathbb{N}$, 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

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

(e) Integer remainder function, $\_mod\_ \in \mathbb{N}^2 \to \mathbb{N}$, where $x \; mod \; y \triangleq x \dot{-} y(x \; div \; y)$

(f) Exponentiation base 2, $e \in \mathbb{N} \to \mathbb{N}$, where $e(x) \triangleq 2^x$.

(g) Logarithm base 2, $\log_2 \in \mathbb{N} \to \mathbb{N}$, where $\log_2(x) \triangleq \begin{cases} greatest\ y\ such\ that\ 2^y \leq x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$

- (a):
  - Claim: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(3, x, 0, 0)$, so $p(x, y) \triangleq x$ is computable
  - For a general projection function, we can add another parameter $k \in \{0, 1\}$, when $k = 0$, we project to $x$ and when $k = 1$, we project to $y$.

projection $p(x,y) \triangleq x$

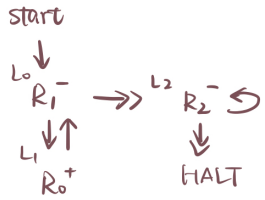
Assume $R_1 = x$, $R_2 = y$

$L_0$ : $R_1^- \to L_1, L_2$

$L_1$ : $R_0^+ \to L_0$

$L_2$ : $R_2^- \to L_2, L_3$

$L_3$ : HALT




---

In general, $p(x,y,k) \triangleq x \cdot I_{k=0} + y \cdot I_{k=1}$, where $k \in \{0,1\}$

Assume $R_1 = x$, $R_2 = y$, $R_3 = k$

$L_0$ : $R_3^- \to L_1, L_2$

$L_1$ : $R_2^- \to L_4, L_5$ # $k=1$, return $y$

$L_2$ : $R_1^- \to L_3, L_6$ # $k=0$, return $x$

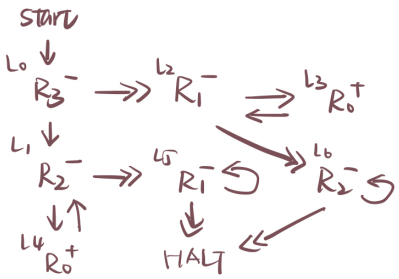
$L_3$ : $R_0^+ \to L_2$

$L_4$ : $R_0^+ \to L_1$

$L_5$ : $R_1^- \to L_5, L_7$

$L_6$ : $R_2^- \to L_6, L_7$

$L_7$ : HALT




- (b):
  - Claim: starting from initial configuration $(0, 0, x, n)$, this machine's computation halts with configuration $(3, n, 0, 0)$, so $c(x) \triangleq n$ is computable

constant $c(x) \triangleq n$

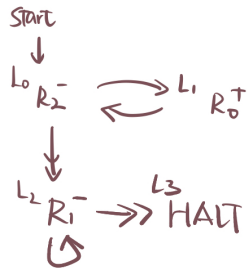
Assume $R_1 = x$, $R_2 = n$

$L_0$ : $R_2^- \to L_1, L_2$

$L_1$ : $R_0^+ \to L_0$

$L_2$ : $R_1^- \to L_2, L_3$

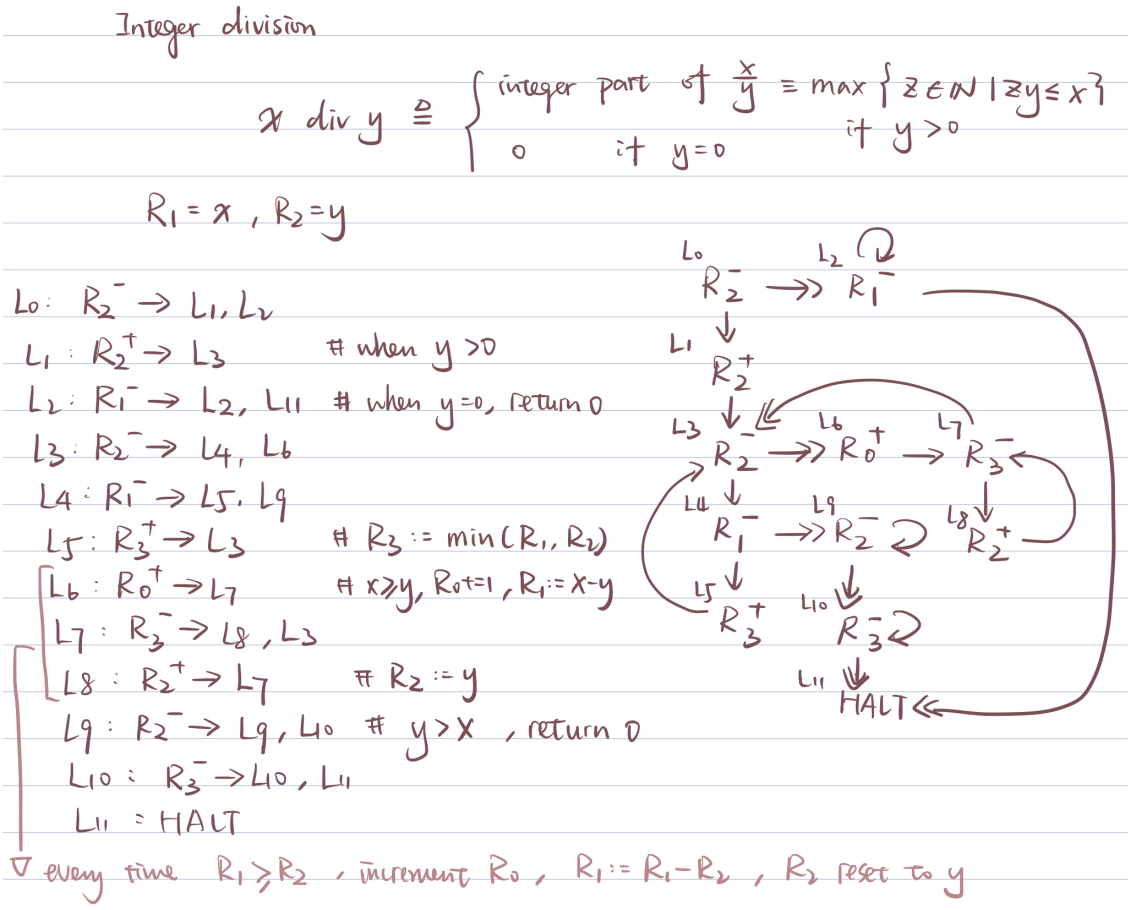
$L_3$ : HALT




- (c):
  - Claim: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(-1, (x-y) \times \mathbf{1}_{y \le x}, 0, 0)$, so this function is register machine computable.

```
-1: halt

0: dec(r2), 1, 4
1: dec(r1), 0, 2
2: dec(r0), 2, 3
3: dec(r2), 3, -1
4: dec(r1), 5, -1
5: inc(r0), 4
```

- (d):
  - Claim: starting from configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(11, \max\{z \in \mathbb{N} | zy \leq x\} \times \mathbf{1}_{y>0}, 0, 0)$, so integer division function defined here is register machine computable.




- (e):
  - Claim: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(-1, x \dot{-} y(x \text{ div } y), 0, 0)$, so this function is register machine computable.

```
-1: halt

# copy(r1, r3)
0: dec(r1), 1, 3
1: inc(r3), 2
2: inc(r4), 0
3: dec(r4), 4, 5
4: inc(r1), 3

# copy(r2, r4)
5: dec(r2), 6, 8
6: inc(r4), 7
7: inc(r5), 5
8: dec(r5), 9, 10
9: inc(r2), 8

# r5 := div(r3, r4)
10: dec(r4), 11, 14
11: inc(r4), 12
12: dec(r4), 13, 16
13: dec(r3), 15, 19
14: dec(r3), 14, 20
15: inc(r6), 12
16: inc(r5), 17
17: dec(r6), 18, 12
18: inc(r4), 17
19: dec(r4), 19, 110
110: dec(r6), 110, 20

# r3 := r2 * r5
20: dec(r2), 21, 26
21: dec(r5), 22, 24
22: inc(r3), 23
23: inc(r6), 21
24: dec(r6), 25, 20
25: inc(r5), 24
26: dec(r5), 26, 30

# r0 := r1 - r3
30: dec(r3), 31, 34
31: dec(r1), 30, 32
32: dec(r0), 32, 33
33: dec(r3), 33, -1
34: dec(r1), 35, -1
35: inc(r0), 34
```

- (f):
  - Claim: starting from initial configuration $(0, 0, x)$, this machine's computation halts with configuration $(2, 2^x, 0)$, so this function is register machine computable.

```
# loop x times
0: inc(r0), 1
1: dec(r1), 3, 2
2: halt
3: inc(r2), 4
4: inc(r2), 5
5: dec(r0), 6, 7
6: inc(r3), 5

# r0 := r2 * r3
7: dec(r2), 8, 13
8: dec(r3), 9, 10
9: inc(r0), 11
10: dec(r4), 12, 7
11: inc(r4), 8
12: inc(r3), 10
13: dec(r3), 13, 1
```

- (g):
  - Claim: starting from initial configuration $(0, 0, x)$, this machine's computation halts with configuration $(-1, \max\{y \in \mathbb{N} | 2^y \leq x\}, 0)$, so this function is register machine computable.

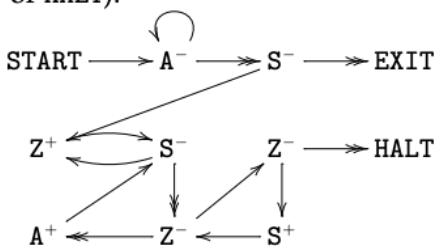
```
# copy(r1, r6)
100: dec(r1), 101, 103
101: inc(r6), 102
102: inc(r4), 100
103: dec(r4), 104, 14
104: inc(r1), 103

# copy(r0, r7)
200: dec(r0), 201, 203
201: inc(r7), 202
202: inc(r4), 200
203: dec(r4), 204, 1
204: inc(r0), 203

# r0 := y, r1 := x, r5 := 2^y, r2 := 2, r3 := 2^y
# loop y times
0: inc(r5), 200
1: dec(r0), 3, 100
2: halt
3: inc(r2), 4
4: inc(r2), 5
5: dec(r5), 6, 7
6: inc(r3), 5

# r5 := r2 * r3
7: dec(r2), 8, 13
8: dec(r3), 9, 10
9: inc(r5), 11
10: dec(r4), 12, 7
```

```
11: inc(r4), 8
12: inc(r3), 10
13: dec(r3), 13, 1

# compare 2^y ? x, r5 := 2^y, r1 := x, r6 := x, r0 := 0, r7 := y
14: dec(r5), 15, 16
15: dec(r6), 14, 20

# if 2^y <= x, r5 := 0, r6 := x - 2^y, should inc y
16: inc(r7), 17
17: dec(r6), 17, 18
18: dec(r7), 19, 0
19: inc(r0), 18

# if x < 2^y, r6 := 0, r5 := 2^y - x
20: dec(r7), 21, 2
21: dec(r5), 21, 22
22: dec(r1), 22, 23
23: dec(r7), 24, 2
24: inc(r0), 23
```

💡 Comments: I spent a long time on some parts of the question, is there a good technique?

# Question 2

**Exercise 3.** 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).




1. $A^-$: clears the contents inside register $A$
2. $S^-$: if $S = 0$, then exist (erroneous halt)
3. The loop $(Z^+, S^-)$ combined with the previous $S^-$: copy the contents of $S$ into $Z$
4. The loop $(Z^-, Z^-, S^+)$ would divide $Z$ by 2 and increment $S$ by 1 each time:

- If $Z$ is an even number, then $S' := \frac{Z}{2}, Z' := 0$, and increment $A$ by 1, then repeat the loop $(S^-, Z^+)$ to copy the contents of $S$ to $Z$
- If $Z$ is an odd number then $S' := \frac{Z-1}{2}, Z' := 0$, the program halts

Claim: starting from configuration $(A, S)$, where $S = 2^x(2i + 1)$. If $S = 0$, the program would halt erroneously; otherwise, the program would halt when $(A, S) := (x, i)$

> 💡 Comments:

# Question 3

### 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 $\Phi$ to a natural number, then combining the codes to create a unique Gödel number $G(\Phi)$. For example, with the assignments $t(\text{`}\forall\text{'}) = 1, t(\text{`}x\text{'}) = 2, t(\text{`.'}) = 3$, and $t(\text{`}=\text{'}) = 4$, the Gödel number of the formula $\forall x.\ x = x$ with a particular combination function could be $G(\text{`}\forall x.\ x = x\text{'}) = 272794772250$.
    a) Is the Gödel numbering of register machines described in the notes a bijection, an injection, a surjection, a total function, a partial function, or a relation? Justify your answer.
    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.
    c) Suggest one or more ways of combining the symbol codes of a formula $\Phi$ to generate a *unique* Gödel number for $\Phi$. Demonstrate your methods on the formula $\Phi = \text{`}\forall x.\ x = x\text{'}$ used above.

- (a): The function $G(prog(e)) = e$ is a partial bijective function
    - Given a register machine program $prog(e)$ with $n$ instructions, labelled from $L_0$ to $L_n$, each label has a corresponding body content $x_0$ to $x_1$.
    - Using the Gödel numbering scheme, we can encode the program $prog(e)$ into $e = \lceil [x_1, x_2, ..., x_n] \rceil$ .
    - The scheme is a partial function because if the instructions are not defined as one of the register machine operations, its encoding is undefined.

- For all programs with instructions defined as register machine operations, each instruction would be uniquely encoded as a number (shown in the following picture). Since $\ll x, y \gg$ is a bijection between $\mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0}$ and $\mathbb{N}_{>0}$, so there is a bijection between the instruction and the number encoded for that instruction.

$$\lceil R_i^+ \to L_j \rceil \triangleq \ll 2i, j \gg$$
$$\lceil R_i^- \to L_j, L_k \rceil \triangleq \ll 2i+1, < j, k > \gg$$
$$\lceil HALT \rceil \triangleq 0$$

- The program is a list of instructions. For a list of instructions, we can encode the list uniquely as a number, since $\ll x, y \gg$ is a bijection between $\mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0}$ and $\mathbb{N}_{>0}$, so there is a bijection between the program and the number encoded for that program.

$$\ulcorner [] \urcorner \triangleq 0 \qquad \ulcorner x :: \ell \urcorner \triangleq \langle\langle x, \ulcorner \ell \urcorner \rangle\rangle = 2^x \left(2 \cdot \ulcorner \ell \urcorner + 1\right)$$

- (b): The function is a partial injective function
  - Since only 4 symbols can be encoded in the scheme, some FOL formulas with other symbols can not be represented, hence the function is partial as it cannot encode all formulas
  - It is not clear from the question what the particular combination function is, but from $G(\Phi) = 272794772250$, we can see that for any valid inputs, there is only one valid output mapped to it, hence the function is injective.
- (c):
  - Simple concatenation to create a partial injective function
    - For a FOL formula, we can define a unique natural number for each logical connective, predicate symbol, universal ($\forall$) and existential ($\exists$) symbol and finite constant symbols, say using $1 \sim k$ to define them. Then for variable symbols, starting from $k+1$, each of them will be defined as $k+i$, $i \in \mathbb{N}_{\geq 1}$ following the order of their occurrences.
    - After defining a number for each symbol, we can concatenate the symbols together to form a unique Gödel numbering code.
    - This scheme is partial because any symbols that are not defined in it cannot be represented as a number; It is injective because some

symbols cannot be placed adjacently to each other, so we cannot decode a random string of numbers into a valid logical formula

```
Φ = ∀x.x=x
Assignment: t(∀) = 1, t(.) = 2, t(=) = 3, t(x) = 4
G(Φ) = 142434
```

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.

- A register machine computable unary function $f$ indicates for any input $x \in \mathbb{N}$ stored in register $R_1$, and all other registers zeroed, we can get an output $y \in \mathbb{N}$ stored in register $R_0$ when the program halts on the register machine. We are required to show that there are infinitely many programs $e$ which when run on the register machine with input $x \in \mathbb{N}$ can give output $y \in \mathbb{N}$
- One informal method of proving is to have a register machine computes the function $f$, then add one or more side-effect-free instructions at the end, which have no impacts on the final output $y$. Since the number of registers is infinite, we can have infinite instruction blocks that do not affect the final output but change the program code.

💡 Comments: I have problems sketching out the formal proof for question 2. Could you give me some hints?

# 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.

💡 Comments:

# Questions

- For undecidable set $S_0 \triangleq \{e|\varphi_e(0)\downarrow\}$, I don't understand why when constructing the register machines, we have $R_1 ::= \ulcorner(R_1 ::= a_1); ...; (R_n := a_n); prog(e)\urcorner$, surely $S_0$ means we want to construct a register machine that can decide whether with program $e$ and input $0$, the program will halt or not; why can't we just say $R_1 ::= \lceil prog(e)\rceil$, $R_2 ::= 0$?

**Proof (sketch):** Suppose $M_0$ is a register machine computing $\chi_{S_0}$. From $M_0$'s program (using the same techniques as for constructing a universal register machine) we can construct a register machine $H$ to carry out:

```
let e = R_1 and ⌜[a_1, ..., a_n]⌝ = R_2 in
    R_1 ::= ⌜(R_1 ::= a_1); ··· ; (R_n ::= a_n); prog(e)⌝;
R_2 ::= 0;
run M_0
```

Then by assumption on $M_0$, $H$ decides the Halting Problem—contradiction. So no such $M_0$ exists, i.e. $\chi_{S_0}$ is uncomputable, i.e. $S_0$ is undeciable. ■