Auckland University of Technology
Theory of Computation - Semester 2, 2025
URM Exercises

Contributors:
Hayden Richard-Marsters
21152003
{ qjn45042@autuni.ac.nz }

1) 2.2. Exercise, page 14:

   Final config: [6, 8, 6, 0, 0]

2) 2.3. Exercise, page 16

   The program of example 2.1 with initial configuration 2, 3, 0, 0 is a non-terminating cycle, and will never stop. As explained why below:

   Set initial register configuration of 2, 3, 0, 0

   Current register configuration: [2, 3, 0, 0]

   The program begins with I_1: jump(1, 2, 6), with the instruction defined as $\mathrm{J}(m, n, q)$, If $r_m = r_n$ jump to the qth instruction; otherwise go on to the next instruction in the program. As $r_1 \neq r_2$, because 2 $(r_1) \neq 3$ (r_2), we go on to the next instruction I_2.

   Current register configuration: [2, 3, 0, 0]

   I_2: succ(2), with the instruction defined as S(n), Add 1 to n. Therefore we increment r_2 by 1, so [2, 3, 0, 0] to [2, 4, 0, 0].

   Current register configuration: [2, 4, 0, 0]

   I_3: succ(3), increment r_3 by 1.

   Current register configuration: [2, 4, 1, 0]

   I_4: jump(1, 2, 6), $r_1 \neq r_2$, we go on to the next instruction I_5.

   Current register configuration: [2, 4, 1, 0]

   I_5: jump(1, 1, 2), $r_1 = r_2$, jump to I_2

   Current register configuration: [2, 4, 1, 0]

   I_6: succ(2), increment r_2 by 1

Current register configuration: [2, 5, 1, 0]

…

This cycle will never end as it will continuously execute the sequence of instructions succ(2), succ(3), jump(1,2,6), jump(1,1,2) indefinitely.

This is because r_1 remains constant at 2 throughout the entire program – no instruction ever modifies it. However, r_2 is incremented by 1 on every pass through the loop. Therefore, r_2 will always be strictly greater than r_1 after the first iteration.

3. 3.3. Exercises 1, page 21

Show that the following functions are computable by devising programs that will compute them.

Each includes program (list of instructions), and test cases/examples (including input, expectation, and step through).

(a) $f(x) = \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{if } x \neq 0; \end{cases}$

Take input x (in register r_1) and output either 0 or 1 (in register r_1):

That means:

If x = 0: output should be 0.
If $x \neq$ 0: output should be 1.

A program that computes this:

Take input x as initial configuration for r_1.

```
I₁: zero(2)          ; clear register 2 (so r₂= 0).
I₂:  zero(3)         ; clear register 3 (register to store output temporarily = 0).
I₃:  jump(1, 2, 5)   ; if r₁= r₂ (i.e. x = 0) -> jump to 5th instruction (halt / end).
I₄:  succ(3)         ; otherwise, increment output storing register r₃= 1.
I₅:  transfer(3,1)   ; transfer r3 (temporary output) to  r₁.
I₆:  HALT            ; end program
```

Hence f is computable.

Below are two examples run on the URM implemented in the previous assignment.

Example 1:

```
~~~~~~~ Running Ex. 3.3. 1a -> initial config: x = 5 -> expectation: r1 = 1 ~~~~~~~
program size = 5
PC=1 regs=[5] next=1: zero(2)
PC=2 regs=[5, 0] next=2: zero(3)
PC=3 regs=[5, 0, 0] next=3: jump(1,2,5)
PC=4 regs=[5, 0, 0] next=4: succ(3)
PC=5 regs=[5, 0, 1] next=5: transfer(3,1)
CURRENT REGISTER CONFIG: regs =[1, 0, 1] next = HALT
```

Outputs r_1 = 1, as 5 ≠ 0.

Example 2:

```
~~~~~~~ Running Ex. 3.3. 1a -> initial config: x = 0 -> expectation: r1 = 0 ~~~~~~~
program size = 5
PC=1 regs=[0] next=1: zero(2)
PC=2 regs=[0, 0] next=2: zero(3)
PC=3 regs=[0, 0, 0] next=3: jump(1,2,5)
PC=5 regs=[0, 0, 0] next=5: transfer(3,1)
CURRENT REGISTER CONFIG: regs =[0, 0, 0] next = HALT
```

Outputs r_1 = 0, as 0 = 0.

(b) $f(x) = 5;$

Constant function, an example of a total computable function, takes any input x, ignores it completely, and halts with 5 in $r_1$as output.

That means:
Any input x = 5.

A program that computes this function:

Take input x as initial configuration for r_1.

```
$I_1$: zero(1)      ; set $r_1$= 0 (clear it, any input $x$ = 0)
$I_2$:  succ(1)     ; $r_1$ = 1
$I_3$:  succ(1)     ; $r_1$ = 2
$I_4$:  succ(1)     ; $r_1$ = 3
$I_5$:  succ(1)     ; $r_1$ = 4
$I_6$:  succ(1)     ; $r_1$ = 5
$I_7$: HALT         ; end of program
```

Hence f is computable.

Below is an example run on the URM.

Example:

```
~~~~~~~ Running Ex. 3.3. 1b -> initial config: x = 72 -> expectation: r1 = 5 ~~~~~~~
program size = 6
PC=1 regs=[72] next=1: zero(1)
PC=2 regs=[0] next=2: succ(1)
PC=3 regs=[1] next=3: succ(1)
PC=4 regs=[2] next=4: succ(1)
PC=5 regs=[3] next=5: succ(1)
PC=6 regs=[4] next=6: succ(1)
CURRENT REGISTER CONFIG: regs =[5] next = HALT
```

Outputs r_1= 5, as any input x = 5.

(c) $f(x,y) = \begin{cases} 0 \text{ if } x = y, \\ 1 \text{ if } x \neq y; \end{cases}$

Take input x, y (register r_1, and $r_2$respectively) and output 0 if $x = y$, or 1 if $x \neq y$ (output in register r_1)

A program that computes this function:

Take input x, y as initial configuration for r_1, r_2.

```
I_1: zero(3)            ; clear temporary output register r_3.
I_2: jump(1, 2, 4)      ; if x = y, jump to 4th instruction (halt with 0)
I_3: succ(3)            ; otherwise increment r_3= 1
I_4: transfer(3, 1)     ; transfer temporary output from r_3to r_1.
I_5: HALT               ; end of program
```

Hence, f is computable.

Example 1:

```
~~~~~~~ Running Ex. 3.3. 1c -> initial config: x, y = 4, 4 -> expectation: r1 = 0
~~~~~~~
program size = 4
PC=1 regs=[4, 4] next=1: zero(3)
PC=2 regs=[4, 4, 0] next=2: jump(1,2,4)
PC=4 regs=[4, 4, 0] next=4: transfer(3,1)
CURRENT REGISTER CONFIG: regs =[0, 4, 0] next = HALT
```

Outputs r_1= 0, as 4 = 4.

Example 2:

```
~~~~~~~ Running Ex. 3.3. 1c -> initial config: x, y = 5, 4 -> expectation: r1 = 1
~~~~~~~
program size = 4
PC=1 regs=[5, 4] next=1: zero(3)
PC=2 regs=[5, 4, 0] next=2: jump(1,2,4)
PC=3 regs=[5, 4, 0] next=3: succ(3)
PC=4 regs=[5, 4, 1] next=4: transfer(3,1)
CURRENT REGISTER CONFIG: regs =[1, 4, 1] next = HALT
```

Outputs r_1 = 1, as $5 \neq 4$

$$\text{(d) } f(x,y) = \begin{cases} 0 \text{ if } x \leq y, \\ 1 \text{ if } x > y; \end{cases}$$

Take input x, y (register r_1, and $r_2$respectively) and output 0 if $x \leq y$, or 1 if $x > y$ (output in register r_1)

A program that computes this function:

Take input x, y as initial configuration for r_1, r_2.

```
I_1:  jump(1, 2, 9)      ; if x = y, jump to output 0
I_2:  transfer(2, 4)     ; r_4 = y (counter)
I_3:  jump(4, 6 , 7)     ; if counter = 0, jump to check if x > 0
I_4:  succ(6)            ; r_6 = r_6 + 1
I_5: jump(6, 1, 9)       ; if comparison = x, jump to output 0 (x ≤ y)
I_6:  jump(1, 1, 3)      ; loop back
I_7:  succ(3)            ; r_3 = 1 (x > y)
I_8:  jump(1, 1, 10)     ; jump to cleanup (transfer output
I_9:  zero(3)            ; r_3 = 0
I_10: transfer(3,1)      ; transfer output from r_3 to r_1
I_11: HALT               ; end of program
```

Hence, f is computable.

Below are three examples of the program run on the URM.

Example 1:

```
~~~~~~~ Running Ex. 3.3. 1d -> initial config: x, y = 5, 4 -> expectation: r1 = 1
~~~~~~~
program size = 10
PC=1 regs=[5, 4] next=1: jump(1,2,9)
```

```
PC=2 regs=[5, 4] next=2: transfer(2,4)
PC=3 regs=[5, 4, 0, 4] next=3: jump(4,6,7)
PC=4 regs=[5, 4, 0, 4, 0, 0] next=4: succ(6)
PC=5 regs=[5, 4, 0, 4, 0, 1] next=5: jump(6,1,9)
PC=6 regs=[5, 4, 0, 4, 0, 1] next=6: jump(1,1,3)
PC=3 regs=[5, 4, 0, 4, 0, 1] next=3: jump(4,6,7)
PC=4 regs=[5, 4, 0, 4, 0, 1] next=4: succ(6)
PC=5 regs=[5, 4, 0, 4, 0, 2] next=5: jump(6,1,9)
PC=6 regs=[5, 4, 0, 4, 0, 2] next=6: jump(1,1,3)
PC=3 regs=[5, 4, 0, 4, 0, 2] next=3: jump(4,6,7)
PC=4 regs=[5, 4, 0, 4, 0, 2] next=4: succ(6)
PC=5 regs=[5, 4, 0, 4, 0, 3] next=5: jump(6,1,9)
PC=6 regs=[5, 4, 0, 4, 0, 3] next=6: jump(1,1,3)
PC=3 regs=[5, 4, 0, 4, 0, 3] next=3: jump(4,6,7)
PC=4 regs=[5, 4, 0, 4, 0, 3] next=4: succ(6)
PC=5 regs=[5, 4, 0, 4, 0, 4] next=5: jump(6,1,9)
PC=6 regs=[5, 4, 0, 4, 0, 4] next=6: jump(1,1,3)
PC=3 regs=[5, 4, 0, 4, 0, 4] next=3: jump(4,6,7)
PC=7 regs=[5, 4, 0, 4, 0, 4] next=7: succ(3)
PC=8 regs=[5, 4, 1, 4, 0, 4] next=8: jump(1,1,10)
PC=10 regs=[5, 4, 1, 4, 0, 4] next=10: transfer(3,1)
CURRENT REGISTER CONFIG: regs =[1, 4, 1, 4, 0, 4] next = HALT
```

Outputs $r_1 = 1$, as $5 > 4$.

Example 2:

```
~~~~~~~ Running Ex. 3.3. 1d -> initial config: x, y = 4, 4 -> expectation: r1 = 0
~~~~~~~
program size = 10
PC=1 regs=[4, 4] next=1: jump(1,2,9)
PC=9 regs=[4, 4] next=9: zero(3)
PC=10 regs=[4, 4, 0] next=10: transfer(3,1)
CURRENT REGISTER CONFIG: regs =[0, 4, 0] next = HALT
```

Outputs $r_1 = 0,$ as $4 \le 4$.

Example 3:

```
~~~~~~~ Running Ex. 3.3. 1d -> initial config: x, y = 4, 5 -> expectation: r1 = 0
~~~~~~~
program size = 10
PC=1 regs=[4, 5] next=1: jump(1,2,9)
PC=2 regs=[4, 5] next=2: transfer(2,4)
PC=3 regs=[4, 5, 0, 5] next=3: jump(4,6,7)
PC=4 regs=[4, 5, 0, 5, 0, 0] next=4: succ(6)
PC=5 regs=[4, 5, 0, 5, 0, 1] next=5: jump(6,1,9)
PC=6 regs=[4, 5, 0, 5, 0, 1] next=6: jump(1,1,3)
```

```
PC=3 regs=[4, 5, 0, 5, 0, 1] next=3: jump(4,6,7)
PC=4 regs=[4, 5, 0, 5, 0, 1] next=4: succ(6)
PC=5 regs=[4, 5, 0, 5, 0, 2] next=5: jump(6,1,9)
PC=6 regs=[4, 5, 0, 5, 0, 2] next=6: jump(1,1,3)
PC=3 regs=[4, 5, 0, 5, 0, 2] next=3: jump(4,6,7)
PC=4 regs=[4, 5, 0, 5, 0, 2] next=4: succ(6)
PC=5 regs=[4, 5, 0, 5, 0, 3] next=5: jump(6,1,9)
PC=6 regs=[4, 5, 0, 5, 0, 3] next=6: jump(1,1,3)
PC=3 regs=[4, 5, 0, 5, 0, 3] next=3: jump(4,6,7)
PC=4 regs=[4, 5, 0, 5, 0, 3] next=4: succ(6)
PC=5 regs=[4, 5, 0, 5, 0, 4] next=5: jump(6,1,9)
PC=9 regs=[4, 5, 0, 5, 0, 4] next=9: zero(3)
PC=10 regs=[4, 5, 0, 5, 0, 4] next=10: transfer(3,1)
CURRENT REGISTER CONFIG: regs =[0, 5, 0, 5, 0, 4] next = HALT
```

Outputs $r_1 = 0$, as $4 \leq 5$.

(e) $f(x) = \begin{cases} \frac{1}{3}x \text{ if } x \text{ is a multiple of 3,} \\ \text{undefined otherwise;} \end{cases}$

Take input x (in register r_1), and output $1/3x$ if x is a multiple of 3 (output in register r_1), otherwise undefined and the program does not terminate.

A program that computes this function:

Take input x as initial configuration for r_1.

```
zero(2)
zero(3)
jump(1,2,9)  // jump to output if equal
succ(3)
succ(2)
succ(2)
succ(2)
jump(1,1,3)  // loop back
transfer(3,1) // output
```

Hence, f is computable.

Below are test cases of the program run on the URM (non-terminating examples not included due to nature of non-termination).

Test case 1:

```
~~~~~~~ Running Ex. 3.3. 1e -> initial config: x = 6 -> expectation: r1 = 2 ~~~~~~~
program size = 9
```

```
PC=1 regs=[6] next=1: zero(2)
PC=2 regs=[6, 0] next=2: zero(3)
PC=3 regs=[6, 0, 0] next=3: jump(1,2,9)
PC=4 regs=[6, 0, 0] next=4: succ(3)
PC=5 regs=[6, 0, 1] next=5: succ(2)
PC=6 regs=[6, 1, 1] next=6: succ(2)
PC=7 regs=[6, 2, 1] next=7: succ(2)
PC=8 regs=[6, 3, 1] next=8: jump(1,1,3)
PC=3 regs=[6, 3, 1] next=3: jump(1,2,9)
PC=4 regs=[6, 3, 1] next=4: succ(3)
PC=5 regs=[6, 3, 2] next=5: succ(2)
PC=6 regs=[6, 4, 2] next=6: succ(2)
PC=7 regs=[6, 5, 2] next=7: succ(2)
PC=8 regs=[6, 6, 2] next=8: jump(1,1,3)
PC=3 regs=[6, 6, 2] next=3: jump(1,2,9)
PC=9 regs=[6, 6, 2] next=9: transfer(3,1)
CURRENT REGISTER CONFIG: regs =[2, 6, 2] next = HALT
```

Outputs r_1 = 2, as 6 is a multiple of 3, and 1/3 of 6 = 2.

Test Case 2:

```
~~~~~~~ Running Ex. 3.3. 1e -> initial config: x = 9 -> expectation: r1 = 3 ~~~~~~~
program size = 9
PC=1 regs=[9] next=1: zero(2)
PC=2 regs=[9, 0] next=2: zero(3)
PC=3 regs=[9, 0, 0] next=3: jump(1,2,9)
PC=4 regs=[9, 0, 0] next=4: succ(3)
PC=5 regs=[9, 0, 1] next=5: succ(2)
PC=6 regs=[9, 1, 1] next=6: succ(2)
PC=7 regs=[9, 2, 1] next=7: succ(2)
PC=8 regs=[9, 3, 1] next=8: jump(1,1,3)
PC=3 regs=[9, 3, 1] next=3: jump(1,2,9)
PC=4 regs=[9, 3, 1] next=4: succ(3)
PC=5 regs=[9, 3, 2] next=5: succ(2)
PC=6 regs=[9, 4, 2] next=6: succ(2)
PC=7 regs=[9, 5, 2] next=7: succ(2)
PC=8 regs=[9, 6, 2] next=8: jump(1,1,3)
PC=3 regs=[9, 6, 2] next=3: jump(1,2,9)
PC=4 regs=[9, 6, 2] next=4: succ(3)
PC=5 regs=[9, 6, 3] next=5: succ(2)
PC=6 regs=[9, 7, 3] next=6: succ(2)
PC=7 regs=[9, 8, 3] next=7: succ(2)
PC=8 regs=[9, 9, 3] next=8: jump(1,1,3)
PC=3 regs=[9, 9, 3] next=3: jump(1,2,9)
PC=9 regs=[9, 9, 3] next=9: transfer(3,1)
CURRENT REGISTER CONFIG: regs =[3, 9, 3] next = HALT
```

Outputs r_1 = 3, as 9 is a multiple of 3, and 1/3 of 9 = 3.

(f) $f(x) = \left[\frac{2x}{3}\right]$. ($[\mathcal{Z}]$ denotes the greatest integer $\leq \mathcal{Z}$).

Take input x (in register r_1) and output $\frac{2x}{3}$ (output in r_1)

A program that computes this function:

Take input x as initial configuration for r_1.

```
zero(2)
zero(3)
transfer(1,4)
jump(3,4,9)
succ(2)
succ(2)
succ(3)
jump(1,1,4)
zero(8)
succ(8)
succ(8)
succ(8)
zero(5)
zero(6)
zero(7)
jump(5,2,24)
succ(5)
succ(7)
jump(7,8,21)
jump(1,1,16)
zero(7)
succ(6)
jump(1,1,16)
transfer(6,1)
```

Hence, f is computable.

Test cases below, both excluding the extensively long step through:

Test Case 1:

```
~~~~~~~ Running Ex. 3.3. 1f -> initial config: x = 3 -> expectation: r1 = 2 ~~~~~~~
CURRENT REGISTER CONFIG: regs =[2, 6, 3, 3, 6, 2, 0, 3] next = HALT
```

Outputs $r_1 = 2$, as $2 * 3/3 = 2$.

Test Case 2:

```
~~~~~~~ Running Ex. 3.3. 1f -> initial config: x = 7 -> expectation: r1 = 4 ~~~~~~~
CURRENT REGISTER CONFIG: regs =[4, 14, 7, 7, 14, 4, 2, 3] next = HALT
```

Outputs $r_1 = 4$, as since the machine computes the integer part of $\frac{2x}{3}$. For input x = 7, this gives $7 * 2/3 = 4.66$, so the floor value is 4.

4. 3.3. Exercises 4, page 22

The transfer instruction $\mathrm{T}(m, n)$ means to replace r_n by r_m. We can show that T(m, n) is redundant by using zero, succ, and jump. To replace r_n by r_m we clear r_n (so it starts at 0), use a temporary register r_p as a counter. Increment both r_p and r_n until r_p = r_m. When the two registers are equal, the loop stops and the program halts, at which point $r_m = r_n$. Hence, the transfer instruction is redundant in the URM formulation, it adds convenience but not computational power.

The program to simulate the transfer instruction can be outlined as below:

```
zero(n)                ; clear destination register.
zero(p)                ; clear temporary counter.
jump(p, m, 7(halt))    ; check if comparison is complete, if yes – halt.
succ(n)                ; increment n.
succ(p)                ; increment counter.
jump(1, 1, 3)          ; unconditional loop back to comparison.
zero(p)                ; clear counter register.
```

Outputs $r_m = r_n$.

To illustrate an example can be run to simulate transfer(1, 2):

```
zero(2)
zero(3)
jump(3, 1, 7)
succ(2)
succ(3)
jump(1, 1, 3)
zero(3)
```

Which run on the URM outputs:

```
Running Ex. 3.3. 4 example 1 -> T(1, 2) -> initial config: [3, 7] -> expectation: [3, 3, 0]
program size = 7
PC=1 regs=[3, 7] next=1: zero(2)
PC=2 regs=[3, 0] next=2: zero(3)
PC=3 regs=[3, 0, 0] next=3: jump(3,1,7)
PC=4 regs=[3, 0, 0] next=4: succ(2)
PC=5 regs=[3, 1, 0] next=5: succ(3)
PC=6 regs=[3, 1, 1] next=6: jump(1,1,3)
PC=3 regs=[3, 1, 1] next=3: jump(3,1,7)
PC=4 regs=[3, 1, 1] next=4: succ(2)
PC=5 regs=[3, 2, 1] next=5: succ(3)
PC=6 regs=[3, 2, 2] next=6: jump(1,1,3)
PC=3 regs=[3, 2, 2] next=3: jump(3,1,7)
PC=4 regs=[3, 2, 2] next=4: succ(2)
PC=5 regs=[3, 3, 2] next=5: succ(3)
PC=6 regs=[3, 3, 3] next=6: jump(1,1,3)
PC=3 regs=[3, 3, 3] next=3: jump(3,1,7)
PC=7 regs=[3, 3, 3] next=7: zero(3)
CURRENT REGISTER CONFIG: regs =[3, 3, 0] next = HALT
```

A second example to further exemplify on any configuration of the URM, simulating transfer(3, 4):

```
zero(4)
zero(5)
jump(5, 3, 7)
succ(4)
succ(5)
jump(1, 1, 3)
zero(5)
```

Which run on the URM outputs:

```
Running Ex. 3.3. 4 example 2 -> T(3, 4) -> initial config: [3, 7, 4, 6] -> expectation: [3,
7, 4, 4, 0]
program size = 7
PC=1 regs=[3, 7, 4, 6] next=1: zero(4)
PC=2 regs=[3, 7, 4, 0] next=2: zero(5)
PC=3 regs=[3, 7, 4, 0, 0] next=3: jump(5,3,7)
PC=4 regs=[3, 7, 4, 0, 0] next=4: succ(4)
PC=5 regs=[3, 7, 4, 1, 0] next=5: succ(5)
PC=6 regs=[3, 7, 4, 1, 1] next=6: jump(1,1,3)
PC=3 regs=[3, 7, 4, 1, 1] next=3: jump(5,3,7)
PC=4 regs=[3, 7, 4, 1, 1] next=4: succ(4)
PC=5 regs=[3, 7, 4, 2, 1] next=5: succ(5)
PC=6 regs=[3, 7, 4, 2, 2] next=6: jump(1,1,3)
PC=3 regs=[3, 7, 4, 2, 2] next=3: jump(5,3,7)
```

```
PC=4 regs=[3, 7, 4, 2, 2] next=4: succ(4)
PC=5 regs=[3, 7, 4, 3, 2] next=5: succ(5)
PC=6 regs=[3, 7, 4, 3, 3] next=6: jump(1,1,3)
PC=3 regs=[3, 7, 4, 3, 3] next=3: jump(5,3,7)
PC=4 regs=[3, 7, 4, 3, 3] next=4: succ(4)
PC=5 regs=[3, 7, 4, 4, 3] next=5: succ(5)
PC=6 regs=[3, 7, 4, 4, 4] next=6: jump(1,1,3)
PC=3 regs=[3, 7, 4, 4, 4] next=3: jump(5,3,7)
PC=7 regs=[3, 7, 4, 4, 4] next=7: zero(5)
CURRENT REGISTER CONFIG: regs =[3, 7, 4, 4, 0] next = HALT
```

Therefore, the transfer instruction is redundant and the function of replacing r_n by r_m, can be computed using only zero, succ, and jump on any configuration of the URM. Hence, the transfer instruction is technically redundant, however as Cutland expresses it is important to note, the transfer instruction is nevertheless natural and convenient relative to the above 7-instruction program to compute the same thing.

End of Exercises