# 50003 - Models Of Computation - (Prof Wiklicky) Lecture 1

Oliver Killane

28/03/22

Lecture Recording

Lecture recording is available here

# Algorithms

Hilbert's Entscheidungsproblem (Decision Problem)

A problem proposed by David Hilbert and Wilhem Ackermann in 1928. Considering if there is an algorithm to determine if any statement is universally valid (valid in every structure satisfying the axioms - facts within the logic system assumed to be true (e.g in maths $1+0=1$)).

This can be also be expressed as an algorithm that can determine if any first-order logic statement is provable given some axioms.

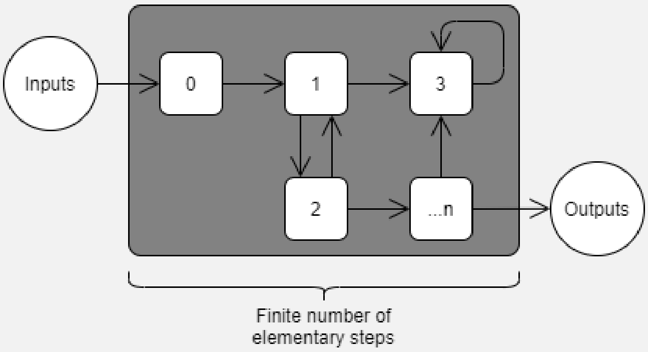
It was proven that no such algorithm exists by Alonzo Church and Alan Turing using their notions of Computing which show it is not computable.

Definition: Algorithms Informally

One definition is: *A finite, ordered series of steps to solve a problem.*

Common features of the many definitions of algorithms are:

- **Finite** Finite number of elementary (cannot be broken down further) operations.
- **Deterministic** Next step uniquely defined by the current.
- **Terminating?** May not terminate, but we can see when it does & what the result is.



## Register Machines

**Definition: Register Machine**

A turing-equivalent (same computational power as a turing machine) abstract machine that models what is computable.

- Infinitely many registers, each storing a natural number ($\mathbb{N} \triangleq \{0, 1, 2, \dots\}$)
- Each instruction has a label associated with it.
- **3 Instructions**

| | |
|---|---|
| ${R_i}^+ \to L_m$ | Add 1 to register $R_i$ and then jump to the instruction at $L_m$ |
| ${R_i}^- \to L_n, L_m$ | If $R_i > 0$ then decrement it and jump to $L_n$, else jump to $L_m$ |
| **HALT** | Halt the program. |

At each point in a program the registers are in a configuration $c = (l, r_0, \dots, r_n)$ (where $r_i$ is the value of $R_i$ and $l$ is the instruction label $L_l$ that is about to be run).

- $c_0$ is the initial configuration, next configurations are $c_1, c_2, \dots$.
- In a finite computation, the final configuration is the **halting configuration**.
- In a **proper halt** the program ends on a **HALT**.
- In an **erroneous halt** the program jumps to a non-existent instruction, the **halting configuration** is for the instruction immediately before this jump.

**Example: Sum of three numbers**

The following register machine computes:

$$R_0 = R_0 + R_1 + R_2 \quad R_1 = 0 \quad R_2 = 0$$

Or as a partial function:

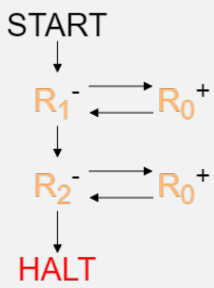
$$f(x, y, z) = x + y + z$$

**Registers**

$R_0 \quad R_1 \quad R_2$

**Program**

$L_0: \quad {R_1}^- \to L_1, L_2$
$L_1: \quad {R_0}^+ \to L_0$
$L_2: \quad {R_2}^- \to L_3, L_4$
$L_3: \quad {R_0}^+ \to L_2$
$L_4: \quad$ **HALT**




**Example Configuration**

| $L_i$ | $R_0$ | $R_1$ | $R_2$ |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 1 | 1 | 1 | 3 |
| 0 | 2 | 1 | 3 |
| 1 | 2 | 0 | 3 |
| 0 | 3 | 0 | 3 |
| 2 | 3 | 0 | 3 |
| 3 | 3 | 0 | 2 |
| 2 | 4 | 0 | 2 |
| 3 | 4 | 0 | 1 |
| 2 | 5 | 0 | 1 |
| 3 | 5 | 0 | 0 |
| 2 | 6 | 0 | 0 |
| 4 | 6 | 0 | 0 |

## Partial Functions

**Definition: Partial Function**

A partial function maps some members of the domain $X$, with each mapped member going to at most one member of the codomain $Y$.

$$f \subseteq X \times Y \text{ and } (x, y_1) \in f \land (x, y_2) \in f \Rightarrow y_1 = y_2$$

| | |
|---|---|
| $f(x) = y$ | $(x, y) \in f$ |
| $f(x) \downarrow$ | $\exists y \in Y.[f(x) = y]$ |
| $f(x) \uparrow$ | $\neg \exists y \in Y.[f(x) = y]$ |
| $X \rightharpoonup Y$ | Set of all **partial functions** from $X$ to $Y$. |
| $X \to Y$ | Set of all **total functions** from $X$ to $Y$. |

A partial function from $X$ to $Y$ is total if it satisfies $f(x) \downarrow$.

Register machines can be considered as partial functions as for a given input/initial configuration, they produce at most one halting configuration (as they are deterministic, for non-finite computations/non-halting there is no halting configuration).

We can consider a register machine as a partial function of the input configuration, to the value of the first register in the halting configuration.

$$f \in \mathbb{N}^n \rightharpoonup \mathbb{N} \;\text{ and }\; (r_0, \ldots, r_n) \in \mathbb{N}^n, r_0 \in \mathbb{N}$$

Note that many different register machines may compute the same partial function.

## Computable Functions

The following arithmetic functions are computable. Using them we can derive larger register machines for more complex arithmetic (e.g logarithms making use of repeated division).

### Projection

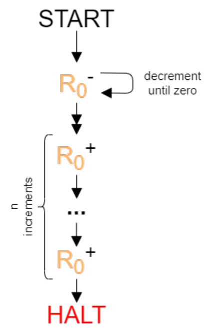
$$p(x, y) \triangleq x$$
$$(r_0, r_1) \to r_0$$

**HALT**


### Constant

$$c(x) \triangleq n$$
$$(r_0) \to n$$

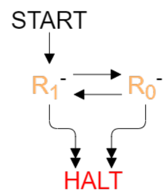
$$\begin{array}{ll} L_0: & {R_0}^- \to L_0, L_1 \\ L_1: & {R_0}^+ \to L_2 \\ \vdots & \vdots \\ L_n: & {R_0}^+ \to L_{n+1} \\ L_{n+1}: & \textbf{HALT} \end{array}$$




### Truncated Subtraction

$$x - y \triangleq \begin{cases} x - y & y \leq x \\ 0 & y > x \end{cases}$$
$$(r_0, r_1) \to r_0 - r_1$$

$$\begin{array}{ll} L_0: & {R_1}^- \to L_1, L_2 \\ L_1: & {R_0}^- \to L_0, L_2 \\ L_2: & \textbf{HALT} \end{array}$$

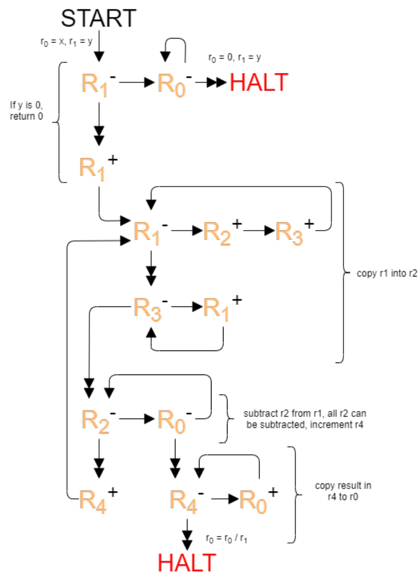



### Integer Division

Note that this is an inefficient implementation (to make it easy to follow) we could combine the halts and shortcut the initial zero check (so we don't increment, then re-decrement).

$$x\ div\ y \triangleq \begin{cases} \left\lfloor \dfrac{x}{y} \right\rfloor & y > 0 \\ 0 & y = 0 \end{cases}$$

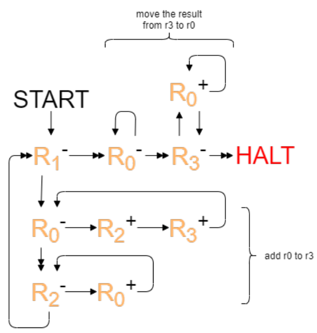
$$\begin{array}{ll}
L_0: & R_1{}^- \to L_3, L_2 \\
L_1: & R_0{}^- \to L_1, L_2 \\
L_2: & \textbf{HALT} \\
L_3: & R_1{}^+ \to L_4 \\
L_4: & R_1{}^- \to L_5, L_7 \\
L_5: & R_2{}^+ \to L_6 \\
L_6: & R_3{}^+ \to L_4 \\
L_7: & R_3{}^- \to L_8, L_9 \\
L_8: & R_1{}^+ \to L_9 \\
L_9: & R_2{}^- \to L_{10}, L_4 \\
L_{10}: & R_0{}^- \to L_9, L_{11} \\
L_{11}: & R_4{}^- \to L_{12}, L_{13} \\
L_{12}: & R_0{}^+ \to L_{11} \\
L_{13}: & \textbf{HALT}
\end{array}$$




## Multiplication

$$x \times y$$

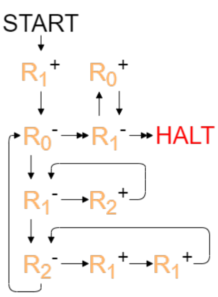
$$\begin{array}{ll}
L_0: & R_1{}^- \to L_5, L_1 \\
L_1: & R_0{}^- \to L_1, L_2 \\
L_2: & R_3{}^- \to L_3, L_4 \\
L_3: & R_0{}^+ \to L_2 \\
L_4: & \textbf{HALT} \\
L_5: & R_0{}^- \to L_6, L_8 \\
L_6: & R_2{}^+ \to L_7 \\
L_7: & R_3{}^+ \to L_5 \\
L_8: & R_2{}^- \to L_9, L_0 \\
L_9: & R_0{}^+ \to L_8
\end{array}$$




## Exponent of base 2

$$e(x) \triangleq 2^x$$

$$\begin{array}{ll}
L_0: & R_1{}^+ \to L_1 \\
L_1: & R_0{}^- \to L_5, L_2 \\
L_2: & R_1{}^- \to L_3, L_4 \\
L_3: & R_0{}^+ \to L_2 \\
L_4: & \textbf{HALT} \\
L_5: & R_1{}^- \to L_6, L_7 \\
L_6: & R_2{}^+ \to L_5 \\
L_7: & R_2{}^- \to L_8, L_1 \\
L_8: & R_1{}^+ \to L_9 \\
L_9: & R_1{}^+ \to L_7
\end{array}$$



# Encoding Programs as Numbers

**Definition: Halting Problem**

Given a set $S$ of pairs $(A, D)$ where $A$ is an algorithm and $D$ is some input data $A$ operates on $(A(D))$.

We want to create some algorithm $H$ such that:

$$H(A, D) \triangleq \begin{cases} 1 & A(D) \downarrow \\ 0 & otherwise \end{cases}$$

Hence if $A(D) \downarrow$ then $A(D)$ eventually halts with some result.

We can use proof by contradiction to show no such algorithm $H$ can exist.

Assume an algorithm $H$ exists:

$$B(p) \triangleq \begin{cases} halts & H(p(p)) = 0 \quad (p(p) \text{ does not halt}) \\ forever & H(p(p)) = 1 \quad (p(p) \text{ halts}) \end{cases}$$

Hence using $H$ on any $B(p)$ we can determine if $p(p)$ halts $(H(B(p)) \Rightarrow \neg H(p(p)))$.

Now we consider the case when $p = B$.

- $B(B)$ **halts** Hence $B(B)$ does not halt. Contradiction!
- $B(B)$ **does not halt** Hence $B(B)$ halts. Contradiction!

Hence by contradiction there is not such algorithm $H$.

In order to reason about programs consuming/running programs (as in the halting problem), we need a way to encode programs as data. Register machines use natural numbers as values for input, and hence we need a way to encode any register machine as a natural number.

## Pairs

| | | | |
|---|---|---|---|
| $\langle\langle x, y\rangle\rangle$ | $= 2^x(2y+1)$ | $y\ 1\ 0_1 \dots 0_x$ | Bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}^+ = \{n \in \mathbb{N} \mid n \neq 0\}$ |
| $\langle x, y\rangle$ | $= 2^x(2y+1) - 1$ | $y\ 0\ 1_1 \dots 1_x$ | Bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$ |

## Lists

We can express lists and right-nested pairs.

$$[x_1, x_2, \dots, x_n] = x_1 : x_2 : \dots : x_n = (x_1, (x_2, (\dots, x_n) \dots))$$

We use zero to define the empty list, so must use a bijection that does not map to zero, hence we use the pair mapping $\langle\langle x, y\rangle\rangle$.

$$l : \begin{cases} \ulcorner [] \urcorner \triangleq 0 \\ \ulcorner x_1 :: l_{inner} \urcorner \triangleq \langle\langle x, \ulcorner l_{inner} \urcorner \rangle\rangle \end{cases}$$

Hence:

$$\ulcorner x_1, \ldots, x_n \urcorner = \langle\langle x_1, \langle\langle \ldots, x_n \rangle\rangle \ldots \rangle\rangle$$

### Instructions

$$\ulcorner R_i^{+} \to L_n \urcorner = \langle\langle 2i, n \rangle\rangle$$

$$\ulcorner R_i^{-} \to L_n, L_m \urcorner = \langle\langle 2i+1, \langle n, m \rangle \rangle\rangle$$

$$\ulcorner \textbf{HALT} \urcorner = 0$$

### programs

Given some program:

$$\ulcorner \begin{pmatrix} L_0 : & instruction_0 \\ \vdots & \vdots \\ L_n : & instruction_n \end{pmatrix} \urcorner = \ulcorner [\ulcorner instruction_0 \urcorner, \ldots, \ulcorner instruction_n \urcorner] \urcorner$$

## Tools

In order to simplify checking workings, I have created a basic python script for running, encoding and decoding register machines.

It is designed to be used in the python shell, to allow for easy manipulation, storing, etc of register machines, encoding/decoding results.

It also produces latex to show step-by-step workings for calculations.

```python
from typing import List, Tuple
from collections import namedtuple

# Register Instructions
Inc = namedtuple('Inc', 'reg label')
Dec = namedtuple('Dec', 'reg label1 label2')
Halt = namedtuple('Halt', '')

'''
 --           -------- -      ----- ----  --  -- ------
 \ \       / /  ----| |    / ----/ -- \|  \/  |  ----|
  \ \  /\  / /| |--  | |   | |   | |  | | \  / | |--
   \ \/  \/ / |  --| | |   | |   | |  | | |\/| |  --|
    \  /\  /  | |----| |---| |---| |--| | |  | | |----
     \/  \/   |------|------\-----\----/|-|  |-|------|

This file can be used to quickly create, run, encode & decode register machine
programs. Furthermore it prints out the workings as formatted latex for easy
use in documents.

Here making use of python's ints as they are arbitrary size (Rust's bigInts
are 3rd party and awful by comparison).

To create register Instructions simply use:
Dec(reg, label 1, label 2)
Inc(reg, label)
Halt()
```

```

To ensure your latex will compile, make sure you have commands for, these are
available on my github (Oliver Killane) (Imperial-Computing-Year-2-Notes):

% register machine helper commands:
\newcommand{\instrlabel}[1]{\text{\textcolor{teal}{$L_{#1}$}}}
\newcommand{\reglabel}[1]{\text{\textcolor{orange}{$R_{#1}$}}}
\newcommand{\instr}[2]{\instrlabel{#1} : & $#2$ \\}
\newcommand{\dec}[3]{\reglabel{#1}^- \to \instrlabel{#2}, \instrlabel{#3}}
\newcommand{\inc}[2]{\reglabel{#1}^+ \to \instrlabel{#2}}
\newcommand{\halt}{\text{\textcolor{red}{\textbf{HALT}}}}

To see examples, go to the end of this file.
'''

# for encoding numbers as <a,b>
def encode_large(x: int, y: int) -> int:
    return (2 ** x) * (2 * y+1)

# for decoding n -> <a,b>
def decode_large(n: int) -> Tuple[int, int]:
    x = 0;

    # get zeros from LSB
    while (n % 2 == 0 and n != 0):
        x += 1
        n /= 2
    y = int((n - 1) // 2)
    return (x,y)

# for encoding <<a,b>> -> n
def encode_small(a: int, b: int) -> int:
    return encode_large(a,b) - 1

# for decoding n -> <<a,b>>
def decode_small(n: int) -> Tuple[int, int]:
    return decode_large(n+1)

# for encoding [a0,a1,a2,...,an] -> <<a0, <<a1, <<a2, <<... <<an, 0 >>...>> >> >> >> -> n
def encode_large_list(lst: List[int]) -> int:
    return encode_large_list_helper(lst, 0)[0]

def encode_large_list_helper(lst: List[int], step: int) -> Tuple[int, int]:
    buffer = r"\to" * step
    if (step == 0):
        print(r"\begin{center}\begin{tabular}{r l l}")
    if len(lst) == 0:
        print(f"{step} &" + rf"$ {buffer} 0$ & (No more numbers in the list, can unwrap recursion) \\")
        return (0, step)
    else:

        print(rf"{step} & $ {buffer} \langle \langle {lst[0]}, \ulcorner {lst[1:]} \urcorner \rangle \rangle $ & (Take next element {lst[0]}, and encode the rest {lst[1:]}) \\")

        (b, step2) = encode_large_list_helper(lst[1:], step + 1)
        c = encode_large(lst[0], b)

```

```python
        step2 += 1

        print(f"{step2} & $ {buffer} \langle \langle {lst[0]}, {b} \\rangle \\rangle = {c} $ & (Can now encode) \\\\")

        if (step == 0):
            print(r"\end{tabular}\end{center}")
        return (encode_large(lst[0], b), step2)

# decode a list from an integer
def decode_large_list(n : int) -> List[int]:
    return decode_large_list_helper(n, [], 0)

def decode_large_list_helper(n : int, prev : List[int], step : int = 0) -> List[int]:
    if (step == 0):
        print(r"\begin{center}\begin{tabular}{r l l l}")
    if n == 0:
        print(rf"{step} & $0$ & ${prev}$ & (At the list end) \\")
        return prev
    else:
        (a,b) = decode_large(n)
        prev.append(a)
        print(rf"{step} & ${n} = \langle \langle {a}, {b} \rangle \rangle \ \ $&$ {prev}$ & (Decode into two integers) \\ ")

        next = decode_large_list_helper(b, prev, step + 1)

        if (step == 0):
            print(r"\end{tabular}\end{center}")

        return next

# For encoding register machine instructions
# R+(i) -> L(j)
def encode_inc(instr: Inc) -> int:
    encode = encode_large(2 * instr.reg, instr.label)
    print(rf"$\ulcorner \inc{{{instr.reg}}}{{{instr.label}}} \urcorner = \langle \langle 2 \times {instr.reg}, {instr.label} \rangle \rangle = {encode}$")
    return encode

# R-(i) -> L(j), L(k)
def encode_dec(instr: Dec) -> int:
    encode: int =  encode_large(2 * instr.reg + 1, encode_small(instr.label1 ,instr.label2))
    print(rf"$\ulcorner \dec {{{instr.reg}}}{{{instr.label1}}}{{{instr.label2}}} \urcorner = \langle \langle 2 \times {instr.reg} + 1, \langle {instr.label1}, {instr.label2} \rangle \rangle \rangle = {encode}$")
    return encode

# Halt
def encode_halt() -> int:
    print(rf"$\ulcorner \halt \urcorner = 0 $")
    return 0

# encode an instruction
def encode_instr(instr) -> int:
    if type(instr) == Inc:
        return encode_inc(instr)
    elif type(instr) == Dec:
        return encode_dec(instr)
```

```
    else:
        return encode_halt()

# display register machine instruction in latex format
def instr_to_str(instr) -> str:
    if type(instr) == Inc:
        return rf"\inc{{{instr.reg}}}{{{instr.label}}}"
    elif type(instr) == Dec:
        return rf"\dec{{{instr.reg}}}{{{instr.label1}}}{{{instr.label2}}}"
    else:
        return r"\halt"

# decode an instruction
def decode_instr(x: int) -> int:
    if x == 0:
        return Halt()
    else:
        assert(x > 0)
        (y,z) = decode_large(x)
        if (y % 2 == 0):
            return Inc(int(y / 2), z)
        else:
            (j,k) = decode_small(z)
            return Dec(y // 2, j, k)

# encode a program to a number by encoding instructions, then list
def encode_program_to_list(prog : List) -> List[int]:
    encoded = []
    print(r"\begin{center}\begin{tabular}{r l l}")
    for (step, instr) in enumerate(prog):
        print(f"{step} & ")
        encoded.append(encode_instr(instr))
        print(r"& \\")
    print(r"\end{tabular}\end{center}")
    print(f"\[{encoded}\]")
    return encoded

# encode a program as an integer
def encode_program_to_int(prog: List) -> int:
    return encode_large_list(encode_program_to_list(prog))

# decode a program by decoding to a list, then decoding each instruction
def decode_program(n : int):
    decoded = decode_large_list(n)
    prog = []
    prog_str = []
    for num in decoded:
        instr = decode_instr(num)
        prog_str.append(instr_to_str(instr))
        prog.append(instr)
    print(f"\[ [ {', '.join(prog_str)} ] \]")
    return prog

# print program in latex form
def program_str(prog) -> str:
    prog_str = []
    for (num, instr) in enumerate(prog):
        prog_str.append(rf"\instr{{{num}}}{{{instr_to_str(instr)}}}")
    print(r"\begin{center}\begin{tabular}{l l}")
    print("\n".join(prog_str))
```

```python
    print(r"\end{tabular}\end{center}")

# run a register machine with an input:
def program_run(prog, instr_no : int, registers : List[int])-> Tuple[int, List[int]]:
    # step instruction label R0 R1 R2 ... (info)
    print(rf"\begin{{center}}\begin{{tabular}}{{l l l c" + " c" * len(registers) + " }")
    print(r"\textbf{Step} & \textbf{Instruction} & \instrlabel{{i}} &" + " & ".join([rf"$\reglabel{{{n}}}$" for n in range(0, len(registers))]) + r" & \textbf{Description}\\")
    print(r"\hline")
    step = 0
    while True:
        step_str = rf"{step} & ${instr_to_str(prog[instr_no])}$ & ${instr_no}$ & " + "&".join([f"${n}$" for n in registers]) + "&"
        instr = prog[instr_no]
        if type(instr) == Inc:
            if (instr.reg >= len(registers)):
                print(step_str + rf"(register {instr.reg} is does not exist)\\")
                break
            elif instr.label >= len(prog):
                print(step_str + rf"(label {instr.label} is does not exist)\\")
                break
            else:
                registers[instr.reg] += 1
                instr_no = instr.label
                print(step_str + rf"(Add 1 to register {instr.reg} and jump to instruction {instr.label})\\")
        elif type(instr) == Dec:
            if (instr.reg >= len(registers)):
                print(step_str + rf"(register {instr.reg} is does not exist)\\")
                break
            elif registers[instr.reg] > 0:
                if instr.label1 >= len(prog):
                    print(step_str + rf"(label {instr.label1} is does not exist)\\")
                    break
                else:
                    registers[instr.reg] -= 1
                    instr_no = instr.label1
                    print(step_str + rf"(Subtract 1 from register {instr.reg} and jump to instruction {instr.label1})\\")
            else:
                if instr.label2 >= len(prog):
                    print(step_str + rf"(label {instr.label2} is does not exist)\\")
                    break
                else:
                    instr_no = instr.label2
                    print(step_str + rf"(Register {instr.reg} is zero, jump to instruction {instr.label2})\\")
        else:
            print(step_str + rf"(Halt!)\\")
            break
        step += 1
    print(r"\end{tabular}\end{center}")
    print("\[(" + ", ".join([str(instr_no)] + list(map(str, registers))) + ")\]")
    return (instr_no, registers)

# Basic tests for program decode and encode
def test():
    prog_a = [
```

```
        Dec(1,2,1),
        Halt(),
        Dec(1,3,4),
        Dec(1,5,4),
        Halt(),
        Inc(0,0)]

    prog_b = [
        Dec(1,1,1),
        Halt()
    ]

    # set R0 to 2n for n+3 instructions
    prog_c = [
        Inc(1,1),
        Inc(0,2),
        Inc(0,3),
        Inc(0,4),
        Inc(0,5),
        Inc(0,6),
        Inc(0,7),
        Dec(1, 0, 9),
        Halt()
    ]

    assert decode_program(encode_program_to_int(prog_a)) == prog_a
    assert decode_program(encode_program_to_int(prog_b)) == prog_b
    assert decode_program(encode_program_to_int(prog_c)) == prog_c

# Examples usage
def examples():
    program_run([
        Dec(1,2,1),
        Halt(),
        Dec(1,3,4),
        Dec(1,5,4),
        Halt(),
        Inc(0,0)
    ], 0, [0,7])

    encode_program_to_list([
        Inc(1,1),
        Inc(0,2),
        Inc(0,3),
        Inc(0,4),
    ])

    encode_program_to_int([
        Dec(1,2,1),
        Halt(),
        Dec(1,3,4),
        Dec(1,5,4),
        Halt(),
        Inc(0,0)
    ])

    decode_program((2 ** 46) * 20483)

examples()
```