Theory of Computations,
Test 2 for the course
CS 5315, Spring 2018

Name:

Up to 5 handwritten pages are allowed.

To prove that a function is computable by a Turing machine if and only if it is \( \mu \)-recursive, we proved that 0, \( \sigma \) and \( \pi \) can be implemented on a Turing machine, and that composition, primitive recursion and \( \mu \)-recursion of Turing-computable functions are also Turing computable.

1. Prove that the functions 0 and \( \sigma \) are Turing-computable, both for unary and for binary numbers. Test your Turing machines on the example of \( n = 3 \) (which is 111 in unary code and 11 in binary code).

---

file:///Q:/cs5315.18/test2.html 2/28/2018
2. Design a Turing machine that computes $f(n) = n + 2$ in binary code. Test in on the example of $n = 3_{10} = 11_2$. Hint: Ignore the last digit, and add 1 to the remaining number, and we know how to design a Turing machine that adds 1 to a binary number.

```
Start,    → 1st, R
1st,      → back, L
1st, 0    → rest, R
1st, 1    → rest, R
rest,     → 1, back, L
rest, 0    → 1, back, L
rest, 1    → 0, R
back, 0    → L
back, 1    → L
back,      → halt
```

$n = 3_{10} = 11_2$

$f(3) = 3 + 2 = 5_{10} = 101_2$
2. Design a Turing machine that computes \( f(n) = n + 2 \) in binary code. Test in on the example of \( n = 3_{10} = 11_2 \). Hint: Ignore the last digit, and add 1 to the remaining number, and we know how to design a Turing machine that adds 1 to a binary number.

Algorithm: Start with second lowest digit, replace every 1 with 0 until we reach 0 or blank, then replace that with 1.

\[
\begin{align*}
\text{Start, } &\rightarrow \text{ skip low, } R \\
\text{skip low, } 0 &\rightarrow \text{ replace, } R \\
\text{skip low, } 1 &\rightarrow \text{ replace, } R \\
\text{replace, } 1 &\rightarrow 0, R \\
\text{replace, } 0 &\rightarrow 1, \text{ back, } L \\
\text{replace, } - &\rightarrow 1, \text{ back, } L \\
\text{back, } 0 &\rightarrow L \\
\text{back, } 1 &\rightarrow L \\
\text{back, } - &\rightarrow \text{ halt}
\end{align*}
\]

Example: Test on \( n = 3_{10} = 11_2 \)

Reverse of 11 is 11

File: //Q:/cs5315.18/test2.html
3. Use a general algorithm for a Turing machine that represents composition to transform your design from Problem 1 into a Turing machine for computing \( f(f(n)) = n + 4 \).

\[ TM(n) = \]

\begin{align*}
\text{Start}_n, & \quad \rightarrow \text{working}_n, R \\
\text{working}_n, 1 & \quad \rightarrow R \\
\text{working}_n, & \quad \rightarrow 1, \text{back}_n, L \\
\text{back}_n, 1 & \quad \rightarrow L \\
\text{back}_n, & \quad \rightarrow \text{start}_{n-1} \\
\text{back}_0, & \quad \rightarrow \text{halt} \\
\end{align*}

\[ TM(4) = \]

\begin{align*}
\text{Start}_4, & \quad \rightarrow \text{working}_4, R \\
\text{working}_4, 1 & \quad \rightarrow R \\
\text{working}_4, & \quad \rightarrow 1, \text{back}_4, L \\
\text{back}_4, 1 & \quad \rightarrow L \\
\text{back}_4, & \quad \rightarrow \text{start}_3 \\
\text{back}_3, & \quad \rightarrow \text{halt} \\
\end{align*}
3. Use a general algorithm for a Turing machine that represents composition to transform your design from Problem 1 into a Turing machine for computing $f(f(n)) = n + 4$.

Here $f(n) = n + 2$

$f(f(n)) = (n + 2) + 4$

**Algorithm**

The idea is to follow the rules of inner machine compute $f(n)$ first, rename state to state 1 and then $f(f(n))$ where state is start 2 instead of halt at $f(n)$ start second state to by start 2.

**Rules**

**Computing $f(n)$**

Start 1, $\rightarrow$ skip low, $R_1$

skip low, 0 $\rightarrow$ repls 1, $R_1$

skip low, 1 $\rightarrow$ repls 1, $R_1$

repls 1, 1 $\rightarrow$ 0, $R_1$

repls 1, 0 $\rightarrow$ 1, back 1, $L_1$

repls 1, $\rightarrow$ 1, back 1, $L_1$

back 1, 1 $\rightarrow$ 1, $L_1$

back 1, 0 $\rightarrow$ L, $L_1$

back 1, $\rightarrow$ Start 2

**Computing $f(f(n)) = n + 4$**

Start 2, $\rightarrow$ skip lst low, $R_2$

skip lst low, 0 $\rightarrow$ skip 2nd low, $R_2$

skip lst low, 1 $\rightarrow$ skip 2nd low, $R_2$

skip 2nd low, 0 $\rightarrow$ repls 2, $R_2$

skip 2nd low, 1 $\rightarrow$ repls 2, $R_2$

repls 2, 1 $\rightarrow$ 0, $R_2$

repls 2, 0 $\rightarrow$ 1, back 2, $L_2$

repls 2, $\rightarrow$ 1, back 2, $L_2$

back 2, 1 $\rightarrow$ L, $L_2$

back 2, 0 $\rightarrow$ L, $L_2$

back 2, $\rightarrow$ Halt
4. Design a Turing machine for computing $n^3$. Test it on the example of a triple (1,2,3).

$n^3$ for unary

Start, $\_\_\_\_\_ \to 1, \text{in}_1, R$
\text{in}_1, 1 \to \_\_\_\_\_ R$
\text{in}_1, \_\_\_\_\_ \to \text{in}_2, R$
\text{in}_2, 1 \to \_\_\_\_\_ R$
\text{in}_2, \_\_\_\_\_ \to \text{in}_3, R$
\text{in}_3, 1 \to R$
\text{in}_3, \_\_\_\_\_ \to \text{prep}, L$
\text{prep}, 1 \to \_\_\_\_\_ carry, L$
\text{carry}, 1 \to L$
\text{carry}, \_\_\_\_\_ \to \_\_\_\_\_ check, L$
\text{check}, \_\_\_\_\_ \to \text{in}_3, R$
\text{check}, 1 \to \_\_\_\_\_ halt

Extreme case when 3rd number is zero

\text{prep}, \_\_\_\_\_ \to 0, \text{L}$
0, \_\_\_\_\_ \to \text{L}$
0, \_\_\_\_\_ \to \_\_\_\_\_ halt

repeats...
4. Design a Turing machine for computing $\pi^3$. Test it on the example of a triple (1,2,3).

Rules: (We are using Unary Number)

- **Start**, $- \rightarrow 1$, erase 1st, R
- erase 1st, 1 $\rightarrow -$, R
- erase 1st, $- \rightarrow$ erase 2nd, R
- erase 2nd, 1 $\rightarrow -$, R
- erase 2nd, $- \rightarrow$ right, R
- right, 1 $\rightarrow$ right, R
- right, $- \rightarrow$ erasing, L
- erasing, 1 $\rightarrow -$, carry, L
- carry, 1 $\rightarrow$ L
- carry, $- \rightarrow$ 1, checking, L
- checking, $- \rightarrow$ right, R
- checking, 1 $\rightarrow -$, halt

Test on the exam

<table>
<thead>
<tr>
<th>Start</th>
<th>1 1 1 1 1 1 1 1 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elst</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Elst</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>2nd</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Right</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Right</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Erase</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Carry</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Carry</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Carry</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Checking</td>
<td>2/28/2018</td>
</tr>
</tbody>
</table>
Right
skipping

erase

carry

checking

Right
skipping similar steps
all the 1 go left

Checking

halt
5. Explain, step by step, how \( \mu \)-recursion \( \mu m. P(n_1, ..., n_k, m) \) can be implemented on a Turing machine. Why you need to copy a tuple \((n_1, ..., n_k, m)\)?

\[
\mu m. P(n, m) \\
\bar{\mu} = (n_1, ..., n_k)
\]

We have a Turing machine for checking \( P \)
We want a Turing machine for computing \( \bar{\mu} \)

We could implement a Turing machine as such to compute \( \bar{\mu} \)

However, we can only apply the Turing machine for checking \( P \) only once with this method. The values of \( \bar{\mu} \) would be overwritten.

Our solution is to instead copy the values \( \bar{\mu} \) onto the tape until the condition is satisfied and we have our smallest \( m \).

![Diagram showing Turing machine implementation]

Our condition is satisfied

Increment \( m \) by 1

Continue this pattern until applying the \( P \) returns true.
5. Explain, step by step, how μ-recursion \( \mu m. P(n_1, \ldots, n_k, m) \) can be implemented on a Turing machine. Why you need to copy a tuple \((n_1, \ldots, n_k, m)\)?

\[ \mu m. P(n, \ldots, n_k, m) = f(n) = \mu m. P(n, m) \]

We have a Turing Machine for checking \( P \). We want a TM for computing \( f \).

In general, \( m = 0 \)

\[ \uparrow P(n, 0) \]

If we apply TM for computing \( P \), here we will lose \( n = \{n_1, \ldots, n_k, m\} \).

That's why we need to copy the tuple \((n_1, \ldots, n_k, m)\).

So, after copy

\[ \uparrow \text{apply TM for computing } P \]

if true

\[ \text{halt} \]

if false

\[ \uparrow P(n, 0) = 0 \]

increase \( m \) by 1

\[ \text{copy} \downarrow \]
If true, apply $\mathcal{P}$.

If false, increase $m$ by 1 and continue.

Apply $\mathcal{R}_{k+1}^{k+2}$.
6. Prove that it is not possible, given a program \( p \) and data \( d \), to check whether \( p \) halts on \( d \).

Theorem: No algorithm is possible that given a program \( p \) and data \( d \), checks whether \( p \) halts on \( d \).

Let us first define the code of a program. A program is a sequence of ASCII symbols. From these symbols, we get a long sequence of bits which we append a leading 1. The resulting integer from interpreting this sequence is called the code of a program.

Lemma: There exists an algorithm that given a natural number \( c \), checks whether \( c \) is a code of a program. If yes, returns the executable file for computing function \( c \). This file is denoted as \( f_c \).

Let's define the following function,

\[
    f(n) = \begin{cases} 
    f_n(n) + 1 & \text{if } n \text{ represents a valid program and this program halts on } n. \quad (\text{halt-checker}(f_n, n) = \text{True}) 
    \end{cases}
\]

We assume a halt-checker exists. Thus, \( f \) is computable.

So \( f \) is computable.

Let us denote \( C_0 \) as the code of this program \( f \).

\[
    \forall n \left( f_{C_0}(n) = f(n) \right)
\]

In particular, for \( n = C_0 \) we get

\[
    f_{C_0}(C_0) = f(C_0)
\]

On the other hand, by definition of \( f \), \( C_0 \) is a valid code and \( f_{C_0} = f \) halts on \( C_0 \).

Thus,

\[
    f_{C_0}(C_0) + 1 = f(C_0)
\]

\[1 = 0\] is a contradiction.

So, our assumption that a halt-checker exists is not possible.
6. Prove that it is not possible, given a program p and data d, to check whether p halts on d.

Java program: Let's define a code of a java program p is a sequence of ASCII symbols, in keyword that has binary numbers. We will get a long sequence of 0's and 1's and append 1 in front and interpret the resulting binary sequence as an integer. This integer is a code of java program.

Proof: By contradiction, let's assume that there exists a halt checker:

\[
\text{halt-checker}(p, d) = \begin{cases} 
1 & \text{if } p \text{ halts on } d \\
0 & \text{if } p \text{ does not halt on } d \\
\text{whatever if } p \text{ is not a java program.}
\end{cases}
\]

We have,

\[
f(c) = \begin{cases} 
c(c) + 1 & \text{if a integer c represent a valid java program and this program halts on c, } \text{halt-checker}(c, c) = \text{true} = 1 \\
0 & \text{otherwise}
\end{cases}
\]

We want to prove, \(f_c\) is computable.

\[
\begin{array}{c}
c \\
\text{compile if c is valid java} \\
\text{Yes} \\
\text{Check whether } f_c \text{ halts on } c \\
\text{Closing halt checker} \\
\text{add 1} \\
\text{return } c(c) + 1
\end{array}
\]

file:///Q:/cs5315.18/test2.html  2/28/2018
So, \( f \) is computable.

So \( c_0 \rightarrow \text{code of a java program computing } f \)
\[ \forall n (f(n)) = f_{c_0}(n) \]

In particular for \( n = c_0 \)
\[ f(c_0) = f_{c_0}(c_0) \]

On the other hand by definition of \( f \), \( c_0 \) is valid code
and \( f_{c_0} = f \) halts on \( c_0 \)

Thus, \( f(c_0) = f_{c_0}(c_0) + 1 \)
\[ f_{c_0}(c_0) = f(c_0) + 1 \]

\( 0 = 1 \) \( \text{contradiction} \)

So our assuming halt checker exist not possible
7. Prove that it is not possible, given a program \( p \) that always halts and a computable function \( g(n) \), to check whether \( p \) always computes \( g(n) \), i.e., where \( p(n) = g(n) \) for all \( n \).

**Th.** No algorithm is possible that, given a program \( p \) that always halts, checks whether \( \forall n \, (p(n) = 0) \).

**Th.** No algorithm is possible that, given a program \( p \) that always halts, checks whether \( \forall n \, (p(n) = g(n)) \).

**Proof** Suppose that we have a checker function

\[
\text{checker}(n) = \begin{cases} 
1 & \text{if } \forall n \, (p(n) = g(n)) \\
0 & \text{if } \forall n \, (p(n) \neq g(n))
\end{cases}
\]

We have \( P(n) \).

We want to check whether \( \forall n \, (p(n) = 0) \).

So we make a new program,

```java
public static void p(int n) {
    return g(n) + F(n);
}
```

Apply checker to \( p(n) \).

\[
\text{checker}(p) = \text{true} \iff \forall n \, (p(n) = g(n)) \iff \forall n \, (g(n) + F(n) = g(n)) \iff \forall n \, (F(n) = 0)
\]

A zero-checker is not possible so a checker function is not possible.
7. Prove that it is not possible, give a program $p$ that always halts and a computable function $g(n)$, to check whether $p$ always computes $g(n)$, i.e., where $p(n) = g(n)$ for all $n$.

Proof: By contradiction let's assume there exist a $g$-function checker, $(P)$

$$
((rove) \iff \forall n \ (p(n) = g(n))
$$

we will build a zero checker

$$
\begin{align*}
q 
\; \xrightarrow{?} 
\forall n \ (q(n) = 0) \\
q(n) = q(n) + g(n) \\
& \quad \text{g-function checker} \\
\forall n \ (p(n) = g(n)) \\
\text{zero_checker}(q) = g\text{-function_checker}(q(n) + g(n))
\end{align*}
$$

$0 = 0 \iff q(n) + g(n) = g(n)$

Case 1: $q(n) = 0$ then $p(n) = q(n) + g(n) = g(n)$

Case 2: $\exists (q(n) \neq 0)$ then $p(n) = q(n) + g(n) \neq g(n)$

So we have built a $g$-function checker assuming $\text{zero_checker}$ exist. But $\text{zero checker}$ not possible. So, $g$-function checker not possible.