COMP3803 — Introduction to Theory of Computation
Assignment 4

April 6, 2020

Name: ___________________________ Student Number: _______________________

Problem 1. (Universal Turing machines) [0 points] (Discarded) Let \( \Sigma \) be an alphabet with \( |\Sigma| \geq 2 \) and pick two distinct characters \( a, b \in \Sigma \). We could encode a natural number \( n \) as the string \( a^n b \), but there is a more compact way. We write \( n \) in binary (using \( a \) for zero and \( b \) for one) with each digit prefixed by an \( a \) and with a \( b \) at the end. For instance, we encode 0, 2 and 10 by \( b, abaab \) and \( abaaabaab \). More precisely, a natural number \( n \) is encoded as the string

\[
\langle n \rangle = \prod_{i=\lfloor \log_2(n+1) \rfloor}^{0} ax_i b
\]

where \( x_i = a \) if \( \lfloor n/2 \rfloor \) is even and \( x_i = b \) otherwise.

Let now \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R) \) be a Turing machine with the same \( \Sigma \) as above. We can assume, without loss of generality, that \( Q = \{q_0, \ldots, q_n, q_A = q_{n+1}, q_R = q_{n+2}\} \) with \( n \in \mathbb{N} \) and \( \Gamma = \{\gamma_0, \ldots, \gamma_{m-1}\} \) with \( m \in \mathbb{N} \) and \( m \geq 2 \) (recall \( \Sigma \subseteq \Gamma \)). We wish to encode \( M \) as a string in \( \{a,b\}^* \) and we begin by encoding the head movement directions: \( \langle \leftarrow \rangle = a \) and \( \langle \rightarrow \rangle = b \). Next we encode a triple \( (q, \gamma, d) \in Q \times \Gamma \times \{\leftarrow, \rightarrow\} \) as the string \( \langle q, \gamma, d \rangle = \langle i \rangle \langle j \rangle \langle d \rangle \), where \( q = q_i \) and \( \gamma = \gamma_j \). This is so we can encode the transition function \( \delta : (Q \setminus \{q_A, q_R\}) \times (\Gamma \cup \{\_\}) \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\} \) as the string

\[
\langle \delta \rangle = \prod_{i=0}^{n} \left( \prod_{j=0}^{m-1} (\delta(q_i, \gamma_j)) \right).
\]

Finally, we simply encode \( M \) as the string \( \langle n \rangle \langle m \rangle \langle \delta \rangle \).

A Universal Turing machine is a Turing machine \( U \) that checks its input is of the form \( \langle M \rangle w \) where \( M \) is a Turing machine and \( w \in \Sigma^* \) (rejecting otherwise) and then "emulates" \( M \) on input \( w \). More precisely, it accepts if \( M \) accepts \( w \), rejects if \( M \) rejects \( w \) and diverges if \( M \) diverges on \( w \).

In this problem, you are asked to construct such Universal Turing machine. Recall that we have (more or less) established that programming languages such as Python are not more expressive than Turing machines. Therefore, it is enough to write a program in such a programming language that: reads an input; checks it has the form \( \langle M \rangle w \); and "emulates" \( M \) on \( w \), returning, say, True if \( M \) accepts \( w \) and False if \( M \) rejects \( w \), and never returning, i.e., computing forever, if \( M \) diverges on \( w \). You do not need to show code in your solution, but you need to: briefly describe how you parse the input, describe which data structures your program uses, state its loop invariant and state its loop termination condition.

Solution: Imagine we have a function \texttt{read()} that returns the next character in the input, terminating the program with a rejection if no next character is available. Here is some pseudo-code to read a natural number from the input:

\begin{verbatim}
Procedure read-nat()
    x ← 0;
    while read() = a do
        x ← 2x;
        if read() = b then
            x ← x + 1;
    return x;
\end{verbatim}
Here is some pseudo-code that reads a triple in $Q \times \Gamma \times \{\leftarrow, \rightarrow\}$ from the input:

**Procedure read-transition()**

```pseudo
i ← read-nat();
j ← read-nat();
if read() = a then
    d ← “←”;
else
    d ← “→”;
return $(q, \gamma_j, d)$;
```

And here is more or less how we read the Turing machine from the input:

**Procedure read-tm()**

```pseudo
n ← read-nat();
m ← read-nat();
Create a matrix $D[0..n][0..m]$;
/* $D[i][j]$ will store $\delta(q_i, \gamma_j)$ if $0 \leq j < m$ and $\delta(q_i, \epsilon)$ if $j = m$. */
for $i$ from 0 to $n$ do
    $D[i][m] ←$ read-transition();
for $j$ from 0 to $m - 1$ do
    $D[i][j] ←$ read-transition();
```

Recall that we represented a configuration of a Turing machine as $t_0qht_1$ where $q \in Q$ is the current state, $h \in \Gamma \cup \{\leftarrow\}$ is the character under the head and $t_0, t_1 \in \Gamma^*$ are the tape characters respectively to the left of the head and to the right of the head ($t_0 = \epsilon$ or $t_1 = \epsilon$ whenever $h = \leftarrow$). Our algorithm represents $q$ by an index in $\{0, \ldots, n + 2\}$, $h$ by an index in $\{0, \ldots, m\}$ (the index $m$ is for the blank) and $t_0$ and $t_1$ by two stacks of indices in $\{0, \ldots, m-1\}$. The top of the stack representing $t_0$ is at its leftmost character and the top of the stack representing $t_1$ is at its rightmost character, so we can easily edit the characters close to the head. We saw in class how to initialize such a Turing machine with the remainder $w \in \Sigma^*$ of the input: if $w = \epsilon$, then $t_0 = t_1 = \epsilon$ and $h = \leftarrow$; if $w = \sigma w'$ where $\sigma \in \Sigma$ and $w' \in \Sigma^*$, then $t_0 = \epsilon$, $h = \sigma$ and $t_1 = w'$; in both cases, $q = q_0$.

The loop invariant is that, after $i$ repetitions of the loop, the Turing machine has performed $i$ transitions and is at configuration $t_0qht_i$. We saw in class how to update the Turing machine configuration, which is what a loop iteration does. Let us recall that here:

- If $\delta(q, h) = (q', \gamma', \leftarrow)$ but $t_0 = \epsilon$, then the new configuration is $q' \gamma' t_1$;
- If $\delta(q, h) = (q', \gamma', \leftarrow)$ and $t_0 = t'_0 \gamma$ with $t'_0 \in \Gamma^*$ and $\gamma \in \Gamma$, then the new configuration is $t'_0 q' \gamma' t_1$;
- If $\delta(q, h) = (q', \gamma', \rightarrow)$ but $t_1 = \epsilon$, then the new configuration is $t_0 \gamma' q' \leftarrow$; and
- If $\delta(q, h) = (q', \gamma', \rightarrow)$ and $t_1 = \gamma t'_1$ with $\gamma \in \Gamma$ and $t'_1 \in \Gamma^*$, then the new configuration is $t_0 \gamma' q' t'_1$.

The algorithm terminates as soon as $q$ becomes $q_A$ or $q_R$, respectively accepting and rejecting. If the Turing machine would diverge, the algorithm never terminates.
Problem 2. [25 points] Prove that a language \( A \) over an alphabet \( \Sigma \) is decidable if, and only if, \( A \) and \( \Sigma^* \setminus A \) are enumerable. If you construct a Turing machine, a broad overview of how it operates is enough. Hint: think about how you could adapt the machine \( U \) from Problem 1.

Solution: If \( A \) is decidable, then there is a Turing machine \( M \) deciding \( A \). As mentioned in class, it is clear that \( M \) enumerates \( A \). The following Turing machine clearly accepts exactly the strings in \( \Sigma^* \setminus A \), enumerating it:

\[
\text{Procedure } M'(w \in \Sigma^*)
\]

- Run Turing machine \( M \) on input \( w \);
- /* Since \( M \) decides \( A \), \( M \) never diverges. */
- \textbf{if } \( M \) accepted \( w \) \textbf{then}
  - Reject;
- \textbf{else}
  - Accept;

Suppose now that both \( A \) and \( \Sigma^* \setminus A \) are enumerable. There are then Turing machines \( M_0 \) and \( M_1 \) enumerating, respectively, \( A \) and \( \Sigma^* \setminus A \). We build a Turing machine \( M \) that simulates \( M_0 \) and \( M_1 \) in parallel and decides \( A \).

\[
\text{Procedure } M(w \in \Sigma^*)
\]

- Let \( C_0 \) be the initial configuration of Turing machine \( M_0 \) on input \( w \);
- Let \( C_1 \) be the initial configuration of Turing machine \( M_1 \) on input \( w \);
- /* Refer to Problem 1. */
- \textbf{while} neither \( C_0 \) nor \( C_1 \) are in the accepting or rejecting state \textbf{do}
  - Change \( C_0 \) into the next configuration for machine \( M_0 \);
  - Change \( C_1 \) into the next configuration for machine \( M_1 \);
  - /* Again, see Problem 1. */
- \textbf{if } \( C_0 \) is at the accepting state or \( C_1 \) is at the rejecting state \textbf{then}
  - Accept;
- \textbf{else}
  - Reject;

Note than on any string \( w \in \Sigma^* \), either machine \( M_0 \) or machine \( M_1 \) (but not both) must accept \( w \). Therefore machine \( M \) is guaranteed to converge. If \( M_0 \) accepted or \( M_1 \) rejected, then the string must be in \( A \). If \( M_0 \) rejected or \( M_1 \) accepted, then the string must be in \( \Sigma^* \setminus A \).
Problem 3. (Regular operations on decidable and enumerable languages) [60 points] Prove that the following statements are all true. A high-level description of any Turing machine you employ is enough, but make sure you show you understand what they do.

- The union of two decidable languages is decidable;
- The union of two enumerable languages is enumerable;
- The concatenation of two decidable languages is decidable;
- The concatenation of two enumerable languages is enumerable;
- If $A$ is a decidable language, then so is $A^*$; and
- If $A$ is an enumerable language, then so is $A^*$.

**Solution:** Throughout the following, let $\Sigma$ be the languages’ alphabet.

Let $M_0$ and $M_1$ be Turing machines deciding, respectively, languages $A$ and $B$. We can build a Turing machine $M$ that, on input $w \in \Sigma^*$, simulates $M_0$ and $M_1$ on input $w$, accepting if, and only if, either $M_0$ or $M_1$ accepts $w$. As both $M_0$ and $M_1$ always converge, $M$ always converges and accepts precisely the strings in $L(M_0) \cup L(M_1) = A \cup B$.

Let again $M_0$ and $M_1$ be Turing machines deciding, respectively, languages $A$ and $B$. As a string $w \in \Sigma^*$ is in $AB$ if, and only if, we can write $w = w AwB$ with $w_A \in A$ and $w_B \in B$, the following Turing machine, which checks every possible way this can be done, decides $AB$.

**Procedure $M(w \in \Sigma^*)$**

Write $w = a_0 \cdots a_{|w|-1}$ where, for all $i$, $a_i \in \Sigma$;

for $i$ from 0 to $|w|$ do

- if simulating $M_0$ on input $a_0 \cdots a_{i-1}$ accepts then

  - if simulating $M_1$ on input $a_i \cdots a_{|w|-1}$ accepts then

    - Accept;

  - Reject;

- Reject;

Let now $M_0$ be a Turing machine deciding language $A$. As a string $w \in \Sigma^*$ is in $A^*$ if, and only if, $w$ can be written as $w_0 \cdots w_{n-1}$ where $n \in \mathbb{N}$ and, for all $i$, $w_i \in A \setminus \{\varepsilon\}$, the following Turing machine, which considers each of these finitely many ways to partition $w$ decides $A^*$. Note: enumerating all such partitions can be done in many ways, one of which is backtracking.

**Procedure $M(w \in \Sigma^*)$**

for each way we can write $w = w_0 \cdots w_{n-1}$ with each $w_i \in \Sigma^* \setminus \{\varepsilon\}$ do

for $i$ from 0 to $n-1$ do

- if simulating $M_0$ on $w_i$ rejects then

  - Continue with the next iteration of the outer-for-loop;

- Accept;

- Reject;

Moving on to enumerable languages, let $M_0$ and $M_1$ be Turing machines enumerating, respectively, languages $A$ and $B$. By simulating $M_0$ and $M_1$ in parallel (akin to Problem 2) and accepting as soon as at least one of them accepts (and diverging otherwise), we accept exactly the strings in $L(M_0) \cup L(M_1) = A \cup B$.

Again let $M_0$ and $M_1$ be Turing machines enumerating, respectively, languages $A$ and $B$. An implementation of a Turing machine $M$ enumerating $AB$ follows. For each possible way to partition its input $w \in \Sigma^*$ into $w_0w_1$ with $w_0, w_1 \in \Sigma^*$, we “spawn” two Turing machine simulations, one for $M_0$ on input $w_0$ and one for $M_1$ on input $w_1$. If one such pair of simulations has both components accepting, then the corresponding $w_0$ is in $A$ and the corresponding $w_1$ is in $B$, so $w = w_0w_1 \in AB$. If no such double acceptance ever happens, we diverge, but since we tried every possible combination, the string $w$ is not in $AB$. 
**Procedure** $M(w \in \Sigma^*)$

$L \leftarrow$ an empty list of pairs of Turing machine configurations;

Write $w = a_0 \cdots a_{|w|-1}$ with each $a_i \in \Sigma$;

for $i$ from 0 to $|w|$ do

$C_0 \leftarrow$ the initial configuration of $M_0$ on input $w_0 \cdots w_{i-1}$;

$C_1 \leftarrow$ the initial configuration of $M_1$ on input $w_i \cdots w_{|w|-1}$;

Add the pair $(C_0, C_1)$ to $L$;

end do

for every do

for each entry in $L$ do

Advance the configurations in the entry that are not in final states (see Problem 1);

if both configurations are in the accepting states then

Accept;

end if

end for

end do

The way we handle the star operation is very similar. Let $M_0$ be a Turing machine enumerating a language $A$. For each way to write the input $w \in \Sigma^*$ as $w_0 \cdots w_{n-1}$ with each $w_i \in \Sigma^* \setminus \{\varepsilon\}$, we have $n$ simulations of $M_0$ on each $w_i$ running. If eventually all simulations for one such partition accept, then each $w_i$ is in $A$, so $w \in A^*$ and we accept. If this never happens, we diverge, but then $w \notin A^*$ as we tried all possible partitions.

**Procedure** $M(w \in \Sigma^*)$

$L \leftarrow$ an empty list of lists of Turing machine configurations;

for each way we can write $w = w_0 \cdots w_{n-1}$ with each $w_i \in \Sigma^* \setminus \{\varepsilon\}$ do

$C \leftarrow$ an empty list of Turing machine configurations;

for $i$ from 0 to $n-1$ do

Add the initial configuration of $M_0$ on input $w_i$ to $C$;

end for

Add $C$ to $L$;

end for

for each entry in $L$ do

Advance the configurations in the entry that are not in final states;

if all configurations in the entry are in the accepting state then

Accept;

end if

end for
Problem 4. [15 points] Suppose we wish to show that all context-free languages are decidable by implementing the CYK algorithm on the $\lambda$-calculus. An implementation of matrices might then make itself useful. Show a correct implementation of matrices on the $\lambda$-calculus, i.e., define the following $\lambda$-terms (the precise specification is omitted, but your answer must make sense):

- **MakeMatrix**: takes as arguments the number of rows and columns as Church numerals and an initial value; returns a matrix with that many rows and columns and with each cell initialized to the given value.

- **MatrixGet**: takes as arguments a matrix and the (0-based) row and column indices of a cell as Church numerals; if the row and column indices are within range, returns the value at the indexed cell; if the indices are out of range, you are allowed to have it perform however you like (undefined behavior).

- **MatrixSet**: takes as arguments a matrix, the (0-based) row and column indices of a cell as Church numerals and a new value; if the row and column indices are within range, returns a matrix that equals the input one on every cell but the indexed one, where it has the new value; if the indices are out of range, you are allowed to have it perform however you like.

Solution: We choose a straightforward implementation of matrices. To us a matrix is a term $M$ such that the term $(M I J)$ normalizes to the entry at row $i$ and column $j$ whenever the terms $I$ and $J$ normalize to Church numerals representing a valid row index $i$ and a valid column index $j$. Therefore we let

$$\text{MatrixGet} = \lambda m, i, j. m_{i\!j}$$

and

$$\text{MakeMatrix} = \lambda m, n, v. \lambda i, j. v.$$

To update a matrix, we return an abstraction that returns the new value when both indices match the updated indices and consults the previous matrix otherwise. To do this, first we introduce the term

$$\text{Eq} = \lambda m, n. \text{And} (\text{AtMost} m n) (\text{AtMost} n m)$$

for Church numeral equality and then we let

$$\text{MatrixSet} = \lambda m, i, j, v. \lambda a, b. \text{If} (\text{And} (\text{Eq} a i) (\text{Eq} b j)) v (m a b).$$