The Australian National University
School of Computing
Dirk Pattinson and Victor Rivera

Semester 2, 2021
Practical Session 11

Foundations of Computation

The practical contains a number of exercises designed for the students to practice the course content. During the practical session, the tutor will work through some of these exercises while students will be responsible for completing the remaining exercises in their own time. There is no expectation that all the exercises will be covered in the practical session.

Covers: Lecture Material Weeks 11, and Weeks 5 – 11

```
Properties of recursive languages, co-recursive, show that recursive is closed
under complement, re and co-re imply recursive as warm-up exercises.
```

Exercise 1 **Recursive Enumerability** (practice)

Consider the sets

$$E = \{w \in \Sigma^* \mid w \text{ is a code of a TM that accepts } \epsilon\}$$

and

$$A = \{w \in \Sigma^* \mid w \text{ is a code of a TM that accepts at least one string}\}$$

Are the sets E and A recursive?

Solution. First consider the set E. We claim that it is not recursive. If it were recursive, there would be a total (i.e. it terminates on all inputs) Turing machine T such that $L(T) = E$. This would allow us to construct a new Turing machine H that would solve the halting problem.

The Turing machine H would implement the following algorithm.

1. Given input (M, w) to the halting problem, it constructs a new Turing machine M' that operates as follows:
   - If the input to M' is not ϵ, halt and reject
   - Otherwise, run M on input w and accept after the execution of M on w has terminated.
2. Run the Turing machine T on input M'.

Note that the TM H always terminates, as T always terminates by assumption. We then have that H accepts (M, w) iff T accepts M' iff $\epsilon \in L(M')$ iff the execution of M on w terminates. That is, H indeed solves the halting problem, contradiction.

For the set A we use a similar argument to show that A is not decidable. Suppose that A were decidable. Then there would be a total Turing machine T with $L(T) = A$. This would allow us to construct a Turing machine H that solves the Halting problem. The TM H operates precisely as above, and in particular terminates for all inputs. For the TM M' constructed above, note that $L(M') \subseteq \{\epsilon\}$ so that

$$M' \text{ accepts any string } \iff M' \text{accepts } \epsilon.$$

In summary we have that H accepts (M, w) iff T accepts M' iff M' accepts any string iff M' accepts ϵ iff the execution of M on input w terminates. In other words, H indeed solves the Halting Problem which is the intended contradiction.

Exercise 2 **Recursive Enumerability** (useful)

Consider the alphabet $\Sigma = \{0, 1\}$. Show that if $S \subseteq \{0, 1\}^*$ is recursively enumerable and nonempty, then there exists a TM E (for 'enumeration') such that

- E terminates on all inputs
- for every input w, the tape content after running E on w is an element of S.
- for each element $s \in S$, there is an input w such that running E on w gives the output s.

In other words, E generates all elements of S, possibly with repetitions.

Solution.

To see this, note that because S is recursively enumerable, there is a TM M with $S = L(M)$. As we just assume that S is recursively enumerable, the TM M does not necessarily terminate for all inputs.

Because S is not empty, there is an element $c_0 \in S$. We say that $w \in \{0,1\}^*$ is the n-th string in $\{0,1\}^*$ if it appears at the n-th position in the following enumeration

```
''
0  1
00 01 10 11
000 001 010 011 100 101 111 111
. . .
```

i.e. the 0-th string is the empty string ϵ, the third string is 1 and the 9th string is 001. Crucially, all $w \in \Sigma^*$ appear in this enumeration.

The TM E operates as follows. Given input string i that is not of the form 0^n1^k, the machine E replaces its input with c_0. Otherwise, if the input is of the form 0^n1^k, it runs the TM M on input w for k steps, where w is the n-th string in $\{0,1\}^*$ as given above. If M terminates on input w in k steps or fewer, the machine E replaces the input with w. Otherwise, the input is replaced with c_0.

From the construction, it is clear that E satisfies the two requirements above:

- the output of E is either $c_0 \in S$ or w where we know that w is accepted by M, i.e. $w \in L(M) = S$.
- if $w \in S = L(M)$, then M terminates on input w, say in k steps. If w is the n-th string in $\{0,1\}^*$, then E terminates with output w on the input 0^n1^k.

This concludes the proof.

Exercise 3 **Recursive Enumerability** (hard)

Consider the alphabet $\Sigma = \{0,1\}$. In the lectures, we have seen that the set

$$T = \{w \in \Sigma^* \mid w \text{ is the code of a TM that terminates on all inputs}\}$$

is not recursive. Is it recursively enumerable? '

Solution. It is not recursively enumerable. To see this, assume, for a contradiction, that T is recursively enumerable. As T is not empty, we obtain a TM E from the previous exercise that always terminates, and replaces its input with a binary string w such that:

- w is the code of a total TM, i.e. $w \in T$
- every code of a total TM arises in this way, i.e. for each $w \in T$ there is $i \in \Sigma^*$ such that running E on i gives w.

We write $f(w)$ for the output of the TM E on input w, and note that $f : \Sigma^* \to \Sigma^*$ is a total function, and $T = \{f(w) \mid w \in \Sigma^*\}$.

Now consider the TM P ('P' for paradox): given input w, run the (total) TM with code $f(w)$ on input w and flip the result, i.e. accept if the TM $f(w)$ rejects w, and accept if the TM $f(w)$ rejects w. Because f is a total function, and we can use the TM E to compute f, this defines a total TM P.

As the range $\{f(w) \mid w \in \Sigma^*\}$ contains all codes of total TMs, we have that $P = f(w)$, for some $w \in \Sigma^*$.

We can now ask whether P accepts w, or not.

Case 1. P accepts w. Then the TM with code $f(w)$ rejects w (because of the flipping). But P is the TM with code $f(w)$, so that P rejects w, contradiction.

Case 2. Similarly, if P rejects w, then the TM with code $f(w)$ accepts w, i.e. P accepts w, contradiction.

In conclusion, our construction was based on the existence of M, i.e. the fact that T is recursively enumerable, and the contradiction shows that M cannot exist, i.e. T is not recursively enumerable.

Exercise 4 **Total Programming Languages** (informal)

In view of the last exercise, do you agree or disagree with the statement

There is no programming language that only defines total functions, and allows to define all total functions.

and if so, why, and if not, why not?

Solution. The least requirement of a programming language would be that the set of syntactically correct programs in this language is recursively enumerable. As the language allows to define *all* total functions, this would mean, by the Church-Turing thesis, that the set of total recursive functions is recursively enumerable. The last exercise has demonstrated that this is not the case.

Alternatively, we can repeat the argument made in the previous exercise, but replace the concept of 'total TM" by the notion of definable functions in the given language.

Exercise 5 **Decimal Expansion of π** (insightful)

Consider the function

$$f(n) = \begin{cases} 1 & \text{if there are } n \text{ consecutive zeros in the decimal expansion of } \pi \\ 0 & \text{otherwise.} \end{cases}$$

Is this function computable, i.e. does there exist a TM M_f over the alphabet $\Sigma = \{0, 1\}$ such that:

- M_f terminates on all inputs
- after running f_M on the binary coding of n, the tape contains the binary coding of $f(n)$?

Solution. The function *is* computable. There are two cases.

Case 1. The binary expansion of π contains n consecutive zeros for all n. Then the TM that erases the input, and replaces it with 1 is a witness for the computability of f.

Case 2. The decimal expansion of π contains 0^m but not 0^{m+1}. Then the TM that

- replaces its input by 1 if the input is a binary coding of a number $\leq m$
- replaces its input by 0 if the input is a binary coding of a number $> m$

witnesses the computability of f.

The point here is that we don't need to know or be able to compute whether the decimal expansion of π contains n consecutive zeros or not. We know that there exists a TM for each case, we just don't know which one is correct.

Exercise 6 **Hoare Logic** (revision)

Consider the Hoare Triple:

$$[x > 0] \ \texttt{while (x} \neq \texttt{0) do x := x - 1} \ [true]$$

1. What is a suitable invariant P?

   Solution.

   $$x \leq 0$$

2. What is a suitable variant E?

   Solution.

   $$x$$

3. We wish to use Hoare Logic to prove the total correctness of the triple

   $$[x > 0] \ \texttt{while (x} \neq \texttt{0) do x := x - 1} \ [true]$$

   Solution.

   1. $x \geq 0 \wedge x \neq 0 \rightarrow x \geq 0$ (Trivial Fact)
   2. $[x - 1 \geq 0 \wedge x - 1 < n]$ $\texttt{x := x - 1}$ $[x \geq 0 \wedge x < n]$ (Assignment)
   3. $[x \geq 0 \wedge x \neq 0 \wedge x = n]$ $\texttt{x := x - 1}$ $[x \geq 0 \wedge x < n]$ (2, Precondition Strengthening)
   4. $[x \geq 0]$ $\texttt{while (x} \neq \texttt{0) do x := x - 1}$ $[x \geq 0 \wedge \neg(x \neq 0)]$ (1, 3, While)
   5. $[x \geq 0]$ $\texttt{while (x} \neq \texttt{0) do x := x - 1}$ $[true]$ (4, Postcondition Weakening)

6. $[x > 0]$ `while (x` $\neq$ `0) do x := x - 1` $[true]$ (5, Precondition Strengthening)

For your reference the while-rule for total correctness is as follows:

$$\frac{P \wedge b \rightarrow E \geq 0 \qquad [P \wedge b \wedge E = n]\ S\ [P \wedge E < n]}{[P]\ \textbf{while}\ b\ \textbf{do}\ S\ [P \wedge \neg b]}$$

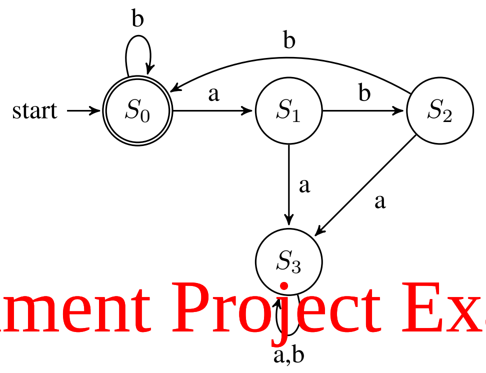
where n is an auxiliary variable not appearing anywhere else.

Exercise 7 **Deterministic Finite Automata** (revision)

Design a minimal DFA that recognises the language

$$\{w \in \{a, b\}^* \mid \text{ in } w, \text{ every } a \text{ is followed by } bb\}$$

Solution.



To see that this automaton is minimal, we just need to demonstrate that the three non-accepting states are not equivalent.

- S_1 and S_2 are not equivalent, as $N^*(S_1, b) \notin F$ but $N^*(S_2, b) \in F$;
- S_1 and S_3 are not equivalent, as $N^*(S_1, bb) \in F$ but $N^*(S_3, bb) \notin F$;
- S_2 and S_3 are not equivalent, as $N^*(S_2, b) \in F$ but $N^*(S_3, b) \notin F$.

It is easy to see that the automaton recognises the given language.

Exercise 8 **Push-down Automata** (revision)

1. Design a (deterministic or non-deterministic) pushdown automaton that recognises precisely all odd length palindromes over the alphabet $\Sigma = \{a, b, c\}$. A palindrome is a word that reads the same backwards and forwards, e.g. "racecar" or "reviver". Briefly explain the design of your automaton.

   Solution. We design the following PDA where q_0 is the starting state, and q_3 is the only final state.

$(q_0, \Diamond, Z)$	$\mapsto q_1, \Diamond Z$	start push phase, all $\Diamond \in \Sigma$
$(q_1, \square, \Diamond)$	$\mapsto q_1/\square\Diamond,$	continue push phase, all $\square, \Diamond \in \Sigma$
$(q_1, \square, \Diamond)$	$\mapsto q_2, \Diamond$	guess middle of string
$(q_2, \square, \square)$	$\mapsto q_2, \epsilon$	match reverse
(q_2, ϵ, Z)	$\mapsto \underline{q_3}, \epsilon)$	accept if fully read

2. Is your PDA deterministic or non-deterministic? Justify your answer.

   Solution. It is non-deterministic, as there are two possible transitions from state q_1 reading the same symbol with the same top of stack.

3. Give an execution trace of your PDA that shows that the word "accca" is accepted by your automaton.

   Solution.

$$\begin{aligned}(q_0, accca, Z) &\Rightarrow (q_1, ccca, aZ)\\ &\Rightarrow (q_1, cca, caZ)\\ &\Rightarrow (q_2, ca, caZ)\\ &\Rightarrow (q_2, a, aZ)\\ &\Rightarrow (q_2, \epsilon, Z)\\ &\Rightarrow (\underline{q_3}, \epsilon, \epsilon)\end{aligned}$$

4. Argue why there *cannot* be an execution trace that accepts the word "acca" (which is an even-length palindrome).

   Solution. The automaton pops one symbol for each symbol it pushes, with the exception of the second transition from state q_1 (which consumes precisely one symbol of the input string). This transition occurs precisely once, so that the total number of characters in an accepted string must be odd.

Exercise 9 **Regular Expressions** (revision)

In this exercise, we conceptualise regular expressions as string, and we add parentheses. Over the finite alphabet $\Sigma = \{a, b, \ldots, z\}$, a *regular expression with parentheses* is defined by the following:

- $\epsilon, \emptyset$ and each $a \in \Sigma$ is a regular expression
- if r and s are regular expressions, the so are $r|s$, rs, $r*$, and (r).

That is, the strings "abc", "$a|(bcd)*e$" and "$a*(b|c)d$" are regular expressions with parentheses, but "$**|a|$" and "$(ba(*$" are not.

1. Design a context-free grammar that generates *precisely* all strings over the alphabet $\Sigma \cup \{(,), *, |, \emptyset, \epsilon\}$ that are regular expressions with parentheses.

   Solution. We have the following grammar that is directly derived from the definition of regular expressions with parentheses. Note that the symbol "ϵ" carries a double meaning here: we use ϵ as a symbol in our grammar, and as a notation for the empty string. To avoid the confusion, we underline $\underline{\epsilon}$ to mean the regular expression that matches the empty string.

$$\begin{aligned}S &\to \underline{\epsilon}\\ S &\to \emptyset\\ S &\to a\\ &\ldots\\ S &\to z\\ S &\to SS\\ S &\to S|S\\ S &\to S*\\ S &\to (S)\end{aligned}$$

2. Hence, or otherwise, construct a pushdown automaton that accepts precisely all regular expressions with parentheses.

   Solution. We use the construction from the lectures to turn the grammar into a non-deterministic PDA. The PDA is given as $A = (Q, q_0, F, \Sigma, \Gamma, Z, \delta)$ where the components are as follows:

   - $Q = \{q_0, q_1, q_2\}$
   - $F = \{q_1\}$
   - $\Sigma = \{\underline{\epsilon}, \emptyset, (,), *, |, a, \ldots, z\}$
   - $\Gamma = \Sigma \cup \{S, Z\}$

   and δ is the transition relation contains the initialisation transition

$$(q_0, \epsilon, Z) \quad \mapsto q_1/SZ$$

   the termination transition:

$$(q_1, \epsilon, Z) \quad \mapsto q_2/\epsilon$$

the transition that expand the non-terminals:

$$\begin{array}{ll}
(q_1, \epsilon, S) & \mapsto q_1/\underline{\epsilon} \\
(q_1, \epsilon, S) & \mapsto q_1/\underline{\emptyset} \\
(q_1, \epsilon, S) & \mapsto q_1/a \\
 & \ldots \\
(q_1, \epsilon, S) & \mapsto q_1/z \\
(q_1, \epsilon, S) & \mapsto q_1/SS \\
(q_1, \epsilon, S) & \mapsto q_1/S|S \\
(q_1, \epsilon, S) & \mapsto q_1/S* \\
(q_1, \epsilon, S) & \mapsto q_1/(S)
\end{array}$$

and the transitions that match the terminal symbols

$$\begin{array}{ll}
(q_1, \underline{\epsilon}, \underline{\epsilon}) & \mapsto q_1/\epsilon \\
(q_1, \underline{\emptyset}, \underline{\emptyset}) & \mapsto q_1/\epsilon \\
(q_1, (, () & \mapsto q_1/\epsilon \\
(q_1,),)) & \mapsto q_1/\epsilon \\
(q_1, |, |) & \mapsto q_1/\epsilon \\
(q_1, *, *) & \mapsto q_1/\epsilon \\
(q_1, a, a) & \mapsto q_1/\epsilon \\
 & \ldots \\
(q_1, z, z) & \mapsto q_1/\epsilon
\end{array}$$

Exercise 10 **Turing Machine** (revision)

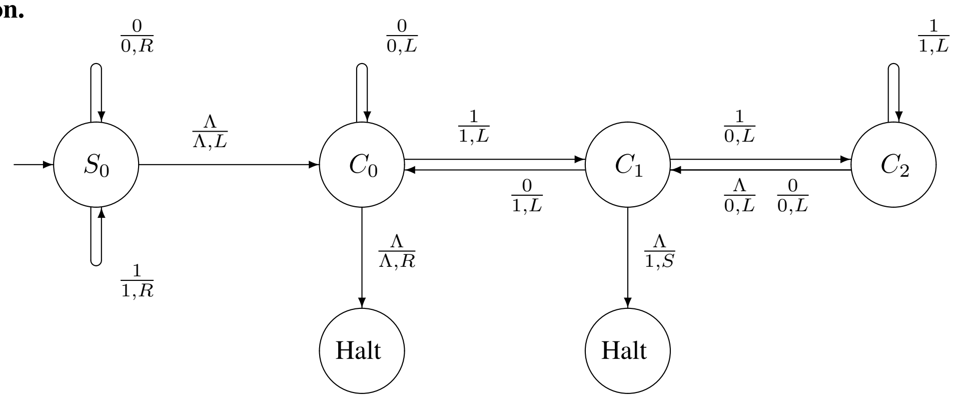
Specify a Turing machine which will multiply the binary number on its tape by 3. Assume (as above) that the head is initially somewhere over the data.

When doing this operation manually, we would work from right to left and compute a new value and a carry at each bit position. The carry can be 0, 1, or 2. The pair (new bit, carry-out) is a simple function of old bit value and carry-in. The following table captures this function

	0	1	2
0	0,0	1,0	0,1
1	1,1	0,2	1,2
Λ	0,0	1,0	0,1

We suggest that you try multiplying a few numbers by hand, using the above table, before starting on the Turing machine.

Solution.



- **Overview:** The TM works from right to left along the binary number, multiplying each digit by 3, adding the "carry-in" from the previous step, and passing on the "carry-out" to the next step. Thus when $3n + i = k + 2j$, and n is on the tape in state C_i, write k and go to state C_j; carry-in is i and carry-out is j.
- **State S_0:** In this state the TM scans right to find the least significant bit.
- **State C_i:** This state indicates that the carry currently is i. The TM is to multiply the number whose rightmost digit (blank treated as 0) is under the head by 3, and add i (which is the "carry" from the previous step of the multiplication).

The important lesson here is that the state of the finite control is being used as memory.

Exercise 11 **Another Undecidable Problem** (revision)

Show that the language

$$L = \{w \mid w \text{ is an encoding of a TM that accepts all strings in } \Sigma^*\}$$

is undecidable.

Solution. Suppose the language L were decidable, so that we would have a TM T (for trivial) that decides L. We argue that this is impossible, as we could then decide the halting problem.

First note that L contains the code of (at least) one Turing machine: any Turing machine without transitions and where the initial state is final accepts all input strings. Pick any such machine M_0.

To see that T would allow us to decide the halting problem, consider an arbitrary TM M and an input string w. We understand (M, w) as an input to the halting problem, and show that we can decide whether or not M terminates on input w, or not.

For this, we convert the pair (M, w) into a Turing machine M_w that we can then test to see whether it accepts all strings or not.

The machine M_w performs the following computation on an input string x.

1. run M on the string w
2. if the above step terminates, accept the string x.

We now claim that M terminates on w if and only if M_w accepts all strings in Σ^*.

First assume that M terminates on w. Then the computation in step 1 terminates, and the string x is accepted in step 2, i.e the machine M_w accepts all strings (as x was arbitrary).

Now suppose that M does *not* terminate on w. Then the computation gets stuck in point 1, hence M_w does *not* accept x, and is therefore *not* a machine that accepts all strings.

Note that the construction of M_w from M and w is clearly effective, and allows us to decide the halting problem:

To decide whether machine M halts on w, check whether the machine M_w accepts all strings.

- if yes, then M terminates on w
- if no, then M does not terminate on w.