3 Turing Machines

We once again wish to generalize the languages we can analyse. Rather than starting on the syntactic side we look at more powerful automata, namely *Turing machines*. These have read/write memory which allows them to access items in an arbitrary order. With these machines there is a distinction to be made between languages which are *recognized* (accepted) and those which are *decided*. We finally give an overview of how the different kinds of languages covered so far fit together. This is the point where we lose connection with the other part of this course.

3.1 Automata with random-access memory—Turing machines

Clearly the question of recognizing the language $\{a^i b^i c^i \mid i \in \mathbb{N}\}$ (see page 40) is not in itself a difficult one. In most programming languages it would be easy to write a procedure which does this, and therefore pushdown automata are not as powerful as other computing devices. Before we go on to look at more expressive automata let us review the progression of such devices so far.

We can think of a DFA as a computing device without dedicated memory. However, such automata can remember a limited amount of information by making use of their states. We can think of a pushdown automaton as a computing device that has a dedicated unlimited memory[15] which can only be used in a very limited way, namely in the manner of a stack. The resulting machines are too weak even to be used to deal with programs on a level that a compiler has to do. Symbol tables are typically used to keep track of variables, and for those a random access memory of some form is required.

Turing machines were proposed as a theoretical device by Alan Turing[16]. Again they are idealized in that they have an infinite memory, which comes in the form of an 'infinite tape' (infinite on both sides). The idea is that this tape consists of cells each of which can take one symbol from some underlying alphabet Σ, and that there is a 'read/write head' which moves over the tape one position at a time and can either read from or write to the tape. It is practical to assume that the input string is written on the tape (from left to right). When the machine starts the read/write head is positioned on the cell holding the first symbol of the input string. We further assume that the tape contains no further symbols that is all remaining cells are *empty*. In order to describe what the machine does (that is, under which circumstances it moves the head one position to the left or the right, or it reads from the tape, or writes to the tape) we need to have a notion of state, just as we had with the other automata. As before there is a specific start state, and this time we need an action `stop` to tell us when the machine has finished its computation.

[15] This is not entirely realistic when thinking of actual computers, since those cannot have an infinite amount of memory. Automata are, after all, theoretical devices which help us reason about computations and as such are ideal and not 'realistic'. Models of the real world often are, as you may have seen in physics. There models are unrealistic in that they fail account for 'minor' matters—when considering a pendulum it is customary to neglect friction against the air, for example. It may help to think of computing devices with 'unlimited memories' as machines with an operator who, whenever the machine threatens to run out of memory, installs more.

[16] See A. Hodges, Alan Turing: The Enigma, Vintage, 1992 or visit `http://www.turing.org.uk/turing/` for an account of the life of this great British logician and computer scientist who at some point worked in the Maths Department at Manchester University. During the war Turing was part of the effort to decode the German Enigma code, and he designed mechanical and electronic devices to help with that task.

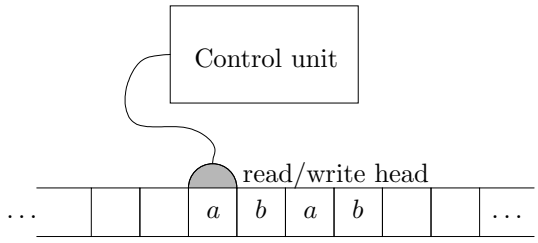

Figure 28: A Turing machine

We have to be able to refer to cells which do not contain a symbol, and to do so we introduce the special symbol $\sqcup$ for an empty cell.[17]. We therefore have to distinguish between the tape alphabet T and the alphabet of input strings Σ, the assumption being that T contains all of Σ and $\sqcup$ at the very least (there may be other symbols occurring in T but not in Σ). In pictures we typically leave cells containing no symbol empty. As for other automata there is a transition function which describes how the Turing machine acts. Given a state and a symbol for the cell where the head is currently positioned there are a number of actions a Turing machine will take at each step:

- it writes a symbol onto the tape;
- it then moves the head one cell to the left or to the right or leaves it were it is;
- it finally goes to a new state.

Hence the transition function will take a pair consisting of a state and a symbol from the tape alphabet and map it to a new state, a symbol that will be written in the current location and one of L, R and N to indicate whether the head should move left, right, or not at all. Alternatively the machine can just stop. Hence the result of applying the transition function to an element of $Q \times T$ is an element of $(Q \times T \times \{L, R, N\}) \cup \{\mathsf{stop}\}$.[18]

Definition 15 *A* **Turing machine** *or* **TM** *over the tape alphabet* $\sqcup \in T \supseteq \Sigma$ *consists of*

- *a finite set of* states Q*;*
- *a* start state $q_\bullet \in Q$*;*
- *a set of* accepting states $F \subseteq Q$*;*
- *a* transition function *that maps* $Q \times T$ *to* $(Q \times T \times \{L, R, N\}) \cup \{\mathsf{stop}\}$.

Note that unlike a finite state machine or pushdown automaton a Turing machine can *manipulate* the input string, and it can *process the input string several times.* Like pushdown automata, Turing machines can run for ever on some input strings.

[17] This is similar to introducing symbols EOF and EOS for pushdown automata

[18] Note that even though we demand that the machine write something every time, this does not mean that it has to change what is on the tape—it can just write again whatever symbol there was. If we didn't do this then the target of the transition function would have the more complicated form $(Q \times T \times \{L, R, N\}) \cup (Q \times \{L, R, N\}) \cup \{\mathsf{stop}\}$.

We often tabulate the transition function δ in a table, and sometimes speak of the *transition table*. Assume that $\Sigma = \{a, b\}$, that the only additional symbol in T is $\sqcup$, and that the machine has only two states, the start state 0 and 1. Then the following describes a transition function for a Turing machine.

δ	a	b	$\sqcup$
0	$(0, a, R)$	$(1, b, R)$	stop
1	stop	stop	stop

Just as before we have to define when a Turing machine accepts a given input string.

Definition 16 *We say that a string α* **is accepted by a Turing machine** *if, when started in the initial state with the head positioned on the first (left-most) character of the input string and the tape otherwise empty, the Turing machine halts*[19] *and the state at that time is an accepting state. We say that a language is* **recognized by a Turing machine** *if it is the set of all words accepted by the machine.*

Assume that in the example above the only accepting state is 0. Clearly given some input string α, the machine reads the characters from left to right until it either finds a b or a blank cell. If it does find a b, it stops in state 1, otherwise in state 0. Hence the language accepted by the machine is $\{a^n \mid n \in \mathbb{N}\}$, the set of all strings over $\{a, b\}$ which consist entirely of as. Notice how the machine restores the input string as it examines it—this does not always have to happen. In fact the machine

δ	a	b	$\sqcup$
0	$(0, \sqcup, R)$	$(1, \sqcup, R)$	stop
1	stop	stop	stop

accepts the same language but destroys the input in the process.

Table 3 gives a more interesting example. The underlying alphabet is $\{a, b\}$ and we once again assume that 0 is the only accepting state.

δ	a	b	$\sqcup$
0	$(1, \sqcup, R)$	$(2, \sqcup, R)$	stop
1	$(1, a, R)$	$(1, b, R)$	$(3, \sqcup, L)$
2	$(2, a, R)$	$(2, b, R)$	$(4, \sqcup, L)$
3	$(5, \sqcup, L)$	stop	$(0, \sqcup, N)$
4	stop	$(5, \sqcup, L)$	$(0, \sqcup, N)$
5	$(5, a, L)$	$(5, b, L)$	$(0, \sqcup, R)$

Table 3: A Turing machine

This machine is is complicated enough that it is not clear at a glance which strings it will accept. We will go through its behaviour for the input string *ababa*.

The machine always starts with the input string being the only non-blank symbols on the tape, and with the head pointing at the first of these. It then starts 'running', and we can

[19] Remember that unlike DFAs, Turing machines do not always stop.

capture 'snapshots' of what it is doing by describing the *configurations* it goes through. A configuration describes the current content of the tape, the current position of the head, and the current state of the machine. This information is sufficient to be able to carry out the remainder of the computation (provided a description of the machine is available). Now at first sight this looks as if we were trying to describe something infinite here (the tape), but that is not true. When the machine starts there are only finitely many non-blank symbols on the tape. In each step of the computation the machine changes at most one symbol on the tape, and therefore at each stage of the overall computation there are only finitely many non-blank symbols on the tape. The most efficient way of presenting the desired information is as a string

$$x_0 x_1 \cdots x_i q x_{i+1} \cdots x_n,$$

where

- q is the current state of the machine;
- the x_j are tape symbols;
- the head of the machine points at the cell containing x_{i+1};
- to the immediate left of the head we find the symbols $x_0 \cdots x_i$ on the tape, and all cells to the left of the cell holding x_0 are blank;
- to the immediate right of the head we find the symbols $x_{i+2} \cdots x_n$, and all cells to the right of x_n hold blanks.

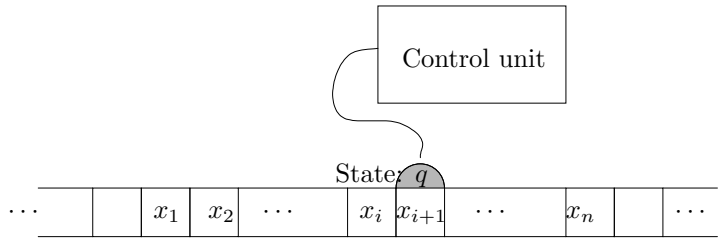

Figure 29: Configuration

Picture 29 gives an idea for how the current status of the machine maps onto the configuration.

The *start configuration* therefore clearly is of the form $q_\bullet \alpha$ where $q_\bullet$ is the start state and α is the input string.

Back to the example. The machine starts in configuration $0ababa$. We look up the transition table for state 0 and input symbol a and find $(1, \sqcup, R)$, so the machine overwrites the a with a blank, moves the head one to the right and goes to state 1, giving configuration $1baba$. We look up the table for state 1 and symbol b and find $(1, b, R)$, so the machine writes a b (thus in effect just leaving the cell as it is) and the head moves one to the right while the machine remains in state 1. Hence the next configuration is $b1aba$. We notice that this is, in fact, all that's ever going to happen in state 1 until the machine finds a blank cell. The machine will go through configurations $ba1ba$, $bab1a$ and $baba1$. Now the head is over a

cell which contains a blank and the corresponding entry in the table reads $(3, \sqcup, L)$, so the resulting configuration is $bab3a$. We once again look up the corresponding table entry and find $(5, \sqcup, L)$, so the next configuration is $ba5b$. The following few configurations are $b5ab$, $5bab$, $5\sqcup bab$ and $0bab$, which is somewhat reminiscent of the configuration in which we started. We give the remaining configurations in less detail.

$2ab$	$a2b$	$ab2$	$a4b$	$5a$
$5\sqcup a$	$0a$	1	3	0.

Hence the state when the machine stops is 0 which is an accepting state and the string $ababa$ is therefore accepted by the machine. Exercise 21 is a good way of familiarizing yourself with the notion of a configuration.

After solving Exercise 21, you should have a good idea regarding which words the machine accepts. Can you spot some kind of pattern?

Here is an alternative, informal way of describing the same Turing machine.

- Remember the first symbol of the input string (by going to state 1 if it an a and to state 2 if it is a b) and erase it.
- Go to the right until you find a blank.
- If the symbol to the left of that blank is the one remembered, erase it. If it is a blank, stop in an accepting state (0). Otherwise stop (in a non-accepting state).
- Go to the left until you find a blank.
- Start over with the head pointing at the symbol to the right of that blank. (This means the machine will start over with an input string where the two outermost characters have been erased, provided they were equal. If they weren't, the machine stopped in a non-accepting state. If there was just one character, the machine stops in an accepting state.)

So the machine works on the given string 'from both ends', and it is checking whether the symbols on each end are the same. In other words, the machine accepts all *palindromes* over the alphabet.

Exercises 22 and 23 invite you to create your own Turing machines. You are now ready to do Assessed Exercise 2.4.

3.2 Languages accepted by Turing machines

It is easy to describe a machine which recognizes the language of all strings consisting of as whose length is even, that is $\{a^{2n} \mid n \in \mathbb{N}\}$. It moves to the right until it finds a blank, switching between two states which code 'odd' and 'even'. It stops when it encounters a blank. The only accepting state is the one that codes for 'even'. Here is the corresponding table.

δ	a	$\sqcup$
0	$(1, a, R)$	stop
1	$(0, a, R)$	stop

But we can have another Turing machine which recognizes this language. Just like the first one, it moves to the right along the input string, switching between two states which code for 'odd' and 'even'. If it finds a blank it will do one of two things:

- it will stop if the current state is the one which is for 'even';
- it will keep moving to the right forever if the current state is the one for 'odd'.

Once again here is a transition table for such a machine.

δ	a	$\sqcup$
0	$(1, a, R)$	stop
1	$(0, a, R)$	$(1, \sqcup, R)$

Which of the two machines is more useful? If you cannot watch the machine work, but just observe whether or not it halts, and in which state, then clearly the second machine is not as useful as the first one. You might wait and wait, but you can never be sure whether that is because the input string is very long or whether the machine has slipped into this endless 'loop'.

We will distinguish these two ideas by having two definitions of the language recognized by a Turing machine.

Definition 17 *A language is* **Turing-recognizable** *(or* **recursively enumerable***) if there is a Turing machine which recognizes it.*

A language is **Turing-decidable** *(or* **recursive***) if there is a Turing machine which recognizes it which halts for all inputs.*

Every Turing-decidable language is Turing-recognizable, but the converse is false. An example for a language that is Turing-recognizable but not Turing-decidable is provided in Theorem 4.1. Figure 30 shows how the two concepts relate. An example for a language which is not even Turing-recognizable is given on Page 67.

The main difference between the two concepts is this: To show that a language is Turing-decidable, we have to find a Turing machine which recognizes it which *halts for all inputs.* For Turing-recognizability it is sufficient if the machine only halts for those inputs which is accepts, that is which belong to the language.

If a language is Turing-decidable then we can find a Turing machine which, when given an input string, will give us a *definite* answer whether the string belongs to the language: 'yes' (if it halts in an accepting state) or 'no' (if it halts in a non-accepting state).

If a language is Turing-recognizable then we can find a Turing machine which, given an input string, will give us a definite answer *only when the string does belong to the language.* The problem is that we have no way of knowing how long to wait for the answer. If the language is very complicated (or the string very long), it might just take a very long time to work out that it does belong.

Proposition 3.1 *Every context-free language is Turing-decidable.*

Proof sketch. This proof requires some technical material on context-free grammars that we have not covered. Therefore we will only give a quick summary of the proof. The 'obvious' thing to try is to take a (possibly non-deterministic) PDA and mimic its behaviour using a

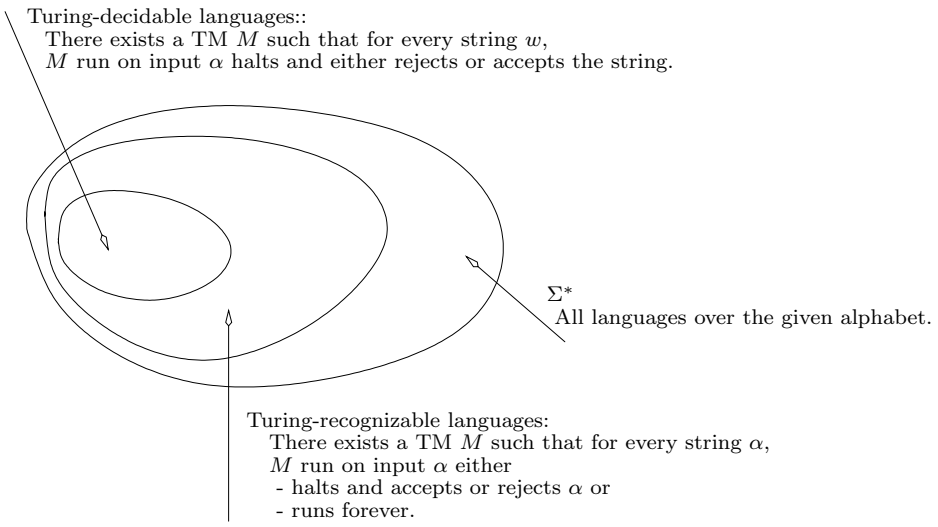

Figure 30: Turing-decidable *versus* Turing-recognizable

Turing machine. However, a PDA might loop forever (see the example in Section 2), whereas we claim that our Turing machine will halt for every input. The alternative would be to take the generating grammar G and keep generating words of the language, but that suffers from a similar flaw: When do we decide that a given word will not be generated, that is, when do we stop trying? Hence we have to be more clever than that.

Every grammar can be converted into a so-called Chomsky-normal form, which has the advantage that every string belonging to the language has a unique parse tree.[20] It also has the advantage that a string of length n will be generated after at most $2n - 1$ applications of a production rule.

This allows us to decide when a given string is *not* in the language: We let the Turing machine convert the given grammar to its normal form.[21] Given an input of length n, we let the machine generate all the strings in the language that require no more than $2n - 1$ applications of a production rule, comparing each to the input string as it is created. If the input string occurs as a generated word the machine stops in an accepting state. If it does not occur among these finitely many words, it stops in a non-accepting state. □

Exercise 24 invites you to draw further connections between Turing machines and material from previous sections.

Theorem 3.2 *A language is Turing-decidable if and only if that holds true for its complement.*

Proof. This is an exercise. □

Recall that context-free languages are not closed under complementation, so this result is more remarkable than you might think at first.

[20] The grammar for arithmetic expression in Section 2.1 has two parse trees for $2 + 3 \times 4$.

[21] There is an algorithm for doing that.

When we defined Turing-decidable languages we said that they were also called 'recursively enumerated' languages. So far we have only been concerned with a Turing machine accepting an input string, that is, recognizing it as a word in some language. Turing machines can do more, however, and this is another way in which they are much more flexible than PDAs.

Let us assume that we have a Turing machine which, given the empty input tape, starts producing strings by writing them on the tape (each separated by a blank). We can then define a language as the set of all strings the Turing machine produces in this way (if it is allowed to go on forever). We say that *the Turing machine enumerates the language*, because the Turing machine lists all the words of the language. We can see this as giving them all numbers: There is a first word, a second word, a third word, etc, and each word in the language will appear at some place and thus get a number as the nth word. Note that we have not insisted on every word appearing only once on the list—so, in fact, a word might get more than one number. However, one could easily clean that up by insisting that the machine first compare the new word with each word created so far (which would slow the machine down a lot, but that is not our concern here), and therefore nothing is lost or gained by adding this stipulation. These languages are, in fact, nothing new as is shown in the following theorem.

Theorem 3.3 *A language is Turing-recognizable if and only if there is a Turing machine enumerating it.*

Proof. This is an exercise (albeit one marked with a $*$). □

closed under	complement	intersection	union	reversal	concatenation
regular	✓	✓	✓	✓	✓
context-free			✓	✓	✓
Turing-decidable	✓	✓	✓	✓	✓
Turing-recognizable		✓	✓	✓	✓

Table 4: Properties of languages

3.3 Chomsky's hierarchy of grammars

Before we leave the subject of different kinds of languages and machines which generate them we want to summarize our findings. The following hierarchy was proposed by the linguist Noam Chomsky in the nineteen-fifties. He was interested in finding classes of languages which generated by various grammars, and whether machines can be identified which recognize precisely the languages in one of these classes. We have only covered some of the entries in the table below in any detail. It is given here in full for completeness' sake.

Languages	Production rules for grammars $\Gamma \to \Delta$ such that	recognizing machines
regular	$\Gamma = R$ any one non-terminal $\Delta = xR'$ or $\Delta = x$ or $\Delta = \epsilon$ where R' non-terminal, x terminal	finite automata
context-free	$\Gamma = R$ any one non-terminal Δ any string	non-deterministic pushdown automata
context-sensitive	Γ any string containing non-terminals Δ any string at least as long as Γ	Turing machines with bounded tape
Turing-recognizable	Γ any string containing non-terminals Δ any string	Turing machines

These so-called 'types' get bigger as we move down the table, and each type is strictly contained in the one below it, that is there are languages in the type below which are not in the one above. Regular languages are the subject of Section 1. Context-free languages are the subject of Section 2. At the time, we did not discuss the origin of the phrase 'context-free'. A context-free grammar has production rules by which one non-terminal is replaced by some string (of terminals and non-terminals), and we may then make use of this production rule whenever that non-terminal symbol occurs in a string—the rule is not sensitive to the *context* in which it is used. Clearly natural languages are not context-free—just because I may replace one word by another in some sentences (contexts) does not mean that I can do this everywhere the first word appears. Context-sensitive grammars allow rules like $aAb \to aBcc$, which say that if we have a non-terminal symbol A which has an a to the left of it and a b to the right of it then we may replace that whole string aAb by $aBcc$. These languages are not of particular interest in computer science which is why we are not discussing them in any detail. Turing-recognizable languages are the most general of all, and the associated grammars are also called 'unrestricted grammars'. We will not provide a proof here that these languages are indeed generated by a grammar, nor for the fact that for any such grammar we can find a Turing machine which recognizes the language. The former is done by creating a Turing machine which generates all the words for the language in question and compares them with the given word[22]. (This is most easily done if the Turing machine has three tapes (one holding the word to be recognized, one holding the rules of the grammar, and one for the creation of words) and is non-deterministic. See the following section for why this is valid.) The latter is done by using configurations. There is a production rule which transfers one configuration into another if and only if there one step in the Turing machine's computation transfers the first configuration into the second.

Summary. In this section we have introduced a notion of an automaton with random-access memory, a Turing machine. Turing machines are more powerful than pushdown automata. The languages recognized by a Turing machine are called Turing-recognizable, and having that property is equivalent to there being a Turing machine which enumerates the language in question. There is a stricter concept, namely that of Turing-decidability, where the deciding Turing machine is also required to effectively reject words which do not belong to the language in question by halting in a non-accepting state.

[22]Compare this with a Turing machine enumerating a language outline above.

Summary of Sections 1-3. We have discussed a number of theoretical devices to investigate a hierarchy of increasingly powerful languages. Figure 31 gives an overview over these languages for the alphabet $\{0, 1\}$, exhibiting examples that separate the different kinds of languages from each other. It also specifies which abstract machines decide or recognize a language, and how the language can be generated. Note that some of the examples given here will be discussed in Section 4 and therefore may not mean very much to you at this point.

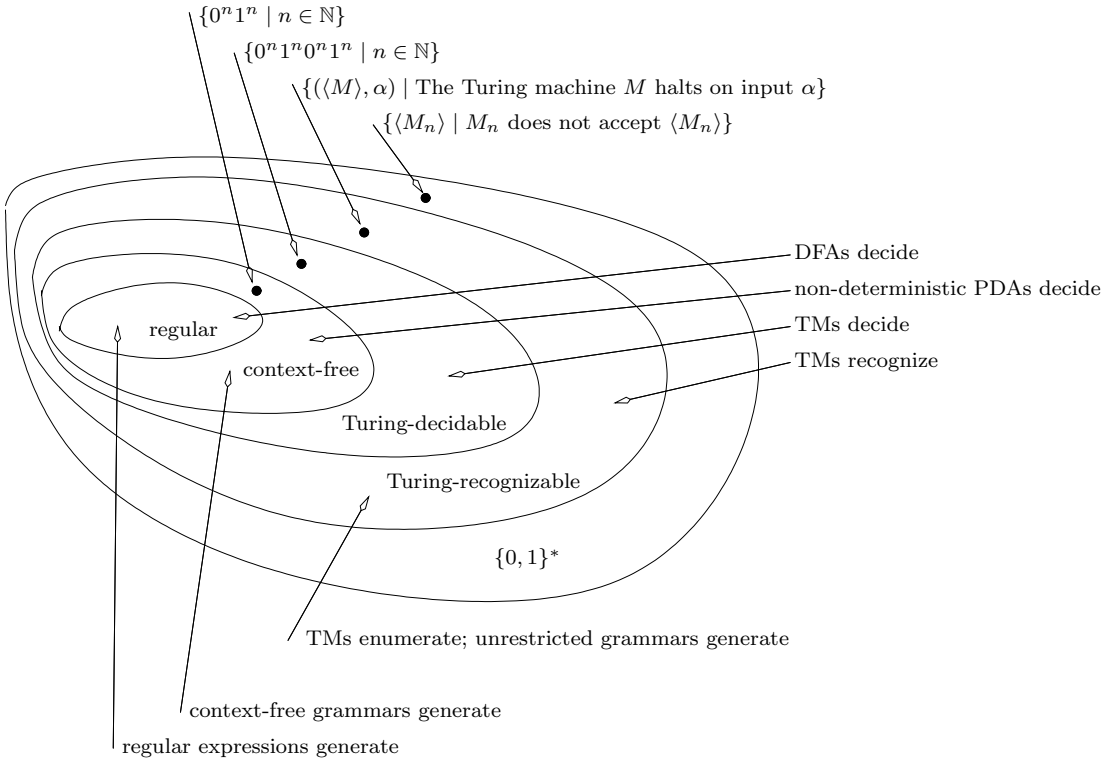

Figure 31: A hierarchy of languages

3.4 Exercises for Section 3

Exercise 21 (a) Write out the configurations the machine in Table 3 goes through when started with input strings *baab* and *baba*.

(b) Which of the following strings are accepted by the machine from Table 3: *baab*, *baba*, *a*, *b*, *babaab*, *bbabb*?

Exercise 22 Extend the Turing machine from Table 3 to one which accepts all palindromes over the alphabet $\{a, b, c\}$.

Exercise 23 (a) Design a Turing machine which recognizes the language $\{(ab)^n \mid n \in \mathbb{N}\}$.

(b) Design a Turing machine which recognizes the language $\{a^n b^n \mid n \in \mathbb{N}\}$.

Exercise 24 (a) Given a DFA, design a Turing machine which simulates that DFA. This is much more straight-forward than mimicking a PDA since the problem of the PDA looping forever does not arise.

(b) *Design a TM which decides whether a word is of the form $a^n b^n c^n$. Recall from Section 2 that this language is not context-free, so Turing-recognizable is indeed more general a concept than than context-free.*

Exercise 25 *Prove Theorem 3.2. (Hint: Given a Turing-decidable language we know that there exists a Turing machine which decides it. From this, argue that there is a Turing machine deciding the complement.) What happens if you try to apply the same technique to a Turing-recognizable language?*

Exercise 26 (*) *Can you sketch a proof of Theorem 3.3? Think about how, you might use a Turing machine that recognizes a language to design one which enumerates it, and* vice versa*. If you get totally stuck you might try consulting [Sip97], p. 140ff.*

Assessed exercises, due in week 12

Note that some of these exercises require knowledge from Section 4!

Exercise 3.1 Consider the following Turing machine with sole accepting state 0.

δ	a	b	c	$\sqcup$
0	$(1, \sqcup, R)$	$(5, b, N)$	$(10, c, N)$	stop
1	$(2, a, R)$	$(5, \sqcup, R)$	stop	$(1, \sqcup, R)$
2	$(2, a, R)$	$(3, \sqcup, L)$	stop	$(2, \sqcup, R)$
3	$(4, a, L)$	stop	stop	$(3, \sqcup, L)$
4	$(4, a, L)$	stop	stop	$(0, \sqcup, R)$
5	stop	$(6, \sqcup, R)$	stop	$(0, \sqcup, N)$
6	stop	$(7, b, R)$	$(10, \sqcup, R)$	$(7, \sqcup, R)$
7	stop	$(7, b, R)$	$(8, \sqcup, L)$	$(7, \sqcup, R)$
8	stop	$(9, b, L)$	stop	stop
9	stop	$(9, b, L)$	stop	$(5, \sqcup, R)$
10	stop	stop	stop	$(0, \sqcup, N)$

Give the configurations the machine goes through when processing the string *abbc*.

Which of the following strings is accepted by the machine: *ab*, *bc*, *abc*, *abbc*, *aabbbc*, *aabbbcc*?

What is the language accepted by the machine?

Exercise 3.2 (a) Design a Turing machine which decides whether a string of 0s and 1s, considered as a binary number, is odd or divisible by 8. You may assume that at the start, the head is on the left most, that is the most significant, digit. Do this by either giving a transition table or by describing the action of your machine along the lines of that on page 57.

(b) *Argue that the intersection of two Turing-decidable languages is decidable.*

Exercise 3.3 *Show that there is no program which reliably checks for division by 0. Hint: Assume you have such a program which takes two files as its input and prints 'Divides by 0' when it detects that the first file, interpreted as a program, will divide by* 0 *if run on the second file, and otherwise prints 'Does not divide by 0'. Now create a new program similar to* `DHalt`*, but instead of just reversing what is printed, replace all occurrences of the instruction to print 'Does not divide by 0' by an instruction which produces the desired contradiction.*