Chapter 3

Automata Theory

3.1 Why Study Automata Theory?

The previous chapter provided an introduction into the theory of formal languages, a topic dealing with the syntax structures found in any text-based interaction.

The current chapter will study different kinds of abstract machines – so-called *automata* – able to recognize the elements of particular languages. As we will soon discover, these two chapters are profoundly coupled together, and constitute two parts of an almost indivisible whole.

On the one hand, this chapter will deal with many theoretical notions, such as the concept of *computation* which lies at the heart of automata theory. On the other hand, the exploration we're about to embark on will let us get to know several practical techniques and applications, such as taking advantages of compiler compilers, which will hopefully help extending one's panoply of essential tools.

3.2 Computation

The current chapter is basically concerned with the question:

> "Does a particular string w belongs to a given language L or not?"

In fact, this very question constitutes the foundational definition of *computation*, whose theory gave birth to computers – those devices that changed our world.

The term *computation* carries nowadays various meanings, such as any type of information processing that can be represented mathematically.

3.2.1 Definition of Computation

In theoretical computer science, computation refers to a mapping that associates to each element of an input set X exactly one element in an output set Y. Without loss of generality, the definition of computation can be defined using the simple alphabet $\Sigma = \{0, 1\}$. The input set X can thus be defined as the Kleene closure of the alphabet $\{0, 1\}^*$. Interestingly, restricting the output set Y to $\{0, 1\}$ – i.e. considering only *"yes or no" decision problems* – doesn't really make any difference: even if we allow more complicated answers, the resulting problems are still equivalent to "yes or no" decision problems.

Computation thus deals mainly with decision problems of the type "given an integer n, decise whether or not n is a prime number" rather than (equivalent) problems of the type "given two real numbers x and y, what is their product $x \times y$".

Definition 3.1. A *decision problem* is a mapping

$$\{0, 1\}^* \longmapsto \{0, 1\}$$

that takes as input any finite string of 0's and 1's and assign to it an output consisting of either 0 ("no") or 1 ("yes").

Obviously, the theory computation and the theory of formal language are just two sides of the same coin: solving a decision problem is the same as accepting strings of a language (namely, the language of all strings that are mapped to 1).

3.2.2 $\mathcal{O}$ Notation

The theory of computation is also closely related to the theory of computational complexity, which is concerned with the relative computational difficulty of computable functions. The question there is how the resource requirements grow with input of increasing length. *Time* resource refers to the number of steps required, whereas *space* resource refers to the size of the memory necessary to perform the computation.

In fact, we are not interested in the *exact* number of time steps (or bits of memory) a particular computation requires (which depends on what machine and language is being used), but rather in *characterizing* how this number increases with larger input. This is usually done with help of the $\mathcal{O}$ notation:

Definition 3.2. The $\mathcal{O}$ notation (pronounce: big oh) stands for "order of" and is used to describe an asymptotic upper bound for the magnitude of a function in terms of another, typically simpler, function.

$f(x)$ is $\mathcal{O}(g(x))$ if and only if $\exists x_0, \exists c > 0$ such that $|f(x)| \leq c \cdot |g(x)|$ for $x > x_0$.

In other words: for sufficiently large x, $f(x)$ does not grow faster than $g(x)$, i.e. $f(x)$ remains smaller than $g(x)$ (up to a constant multiplicative factor).

The statement "$f(x)$ is $\mathcal{O}(g(x))$" as defined above is usually written as $f(x) = \mathcal{O}(g(x))$. Note that this is a slight abuse of notation: for instance $\mathcal{O}(x) = \mathcal{O}(x^2)$ but $\mathcal{O}(x^2) \neq \mathcal{O}(x)$. For this reason, some literature prefers the set notation and write $f \in \mathcal{O}(g)$, thinking of $\mathcal{O}(g)$ as the set of all functions dominated by g.

Example 3.1. If an algorithm uses exactly $3n^2 + 5n + 2$ steps to process an input string of length n, its time complexity is $\mathcal{O}(n^2)$.

Example 3.2. Searching a word in a dictionary containing n entries is not a problem with linear time complexity $\mathcal{O}(n)$: since the words are sorted alphabetically, one can obviously use an algorithm more efficient than searching each page starting from the first one (for instance, opening the dictionary in the middle, and then again in the middle of the appropriate half, and so on). Since doubling n requires only one more time step, the problem has a time complexity of $\mathcal{O}(\log(n))$.

3.3 Finite State Automata

Imagine that you have to design a machine that, when given an input string consisting only of a's, b's and c's, tells you if the string contains the sequence "abc".

At first sight, the simplest algorithm could be the following:

1. Start with first character of the string.
2. Look if the three characters read from the current position are a, b and c.
3. If so, stop and return "yes".
4. Otherwise, advance to the next character and repeat step 2.
5. If the end of the string is reached, stop and return "no".

In some kind of pseudo-code, the algorithm could be written like this (note the two `for` loops):

```
input = raw_input("Enter a string: ")
sequence = "abc"
found = 0
for i in range(1, len(input) - 2):
    flag = 1
    for j in range(1, 3):
        if input[i + j] != sequence[j]:
           flag = 0
    if flag:
         found = 1

if found:
    print "Yes"
else:
    print "No"
```

If the length of the input string is n, and the length of the sequence to search for is k (in our case: $k = 3$), then the time complexity of the algorithm is $\mathcal{O}(n \cdot k)$.

Yet, there's room for improvement... Consider the following algorithm:

0. Move to the next character.
   If the next character is a, go to step 1.
   Otherwise, remain at step 0.

1. Move to the next character.
   If the next character is b, go to step 2.
   If the next character is a, remain at step 1.
   Otherwise, go to step 0.

2. Move to the next character.
   If the next character is c, go to step 3.
   If the next character is a, go to step 1.
   Otherwise, go to step 0.

3. Move to the next character.
   Whatever the character is, remain at step 3.

Listing 3.1: A more efficient algorithm to search for the sequence "abc".

If the end of the input string is reached on step 3, then the sequence "abc" has been found. Otherwise, the input string does not contain the sequence "abc".

Note that with this algorithm, each character of the input string is only read once. The time complexity of this algorithm is thus only $\mathcal{O}(n)$! In fact, this algorithm is nothing else than a *finite state automaton*.

3.3.1 Definition

A *finite state automaton* (plural: finite state automata) is an abstract machine that successively reads each symbols of the input string, and changes its state according to a particular control mechanism. If the machine, after reading the last symbol of the input string, is in one of a set of particular states, then the machine is said to *accept* the input string. It can be illustrated as follows:

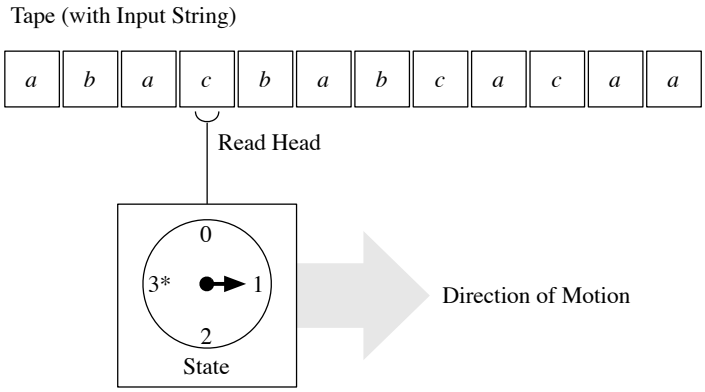


Figure 3.1: Illustration of a finite state automaton.

Definition 3.3. A *finite state automaton* is a five-tuple

$$(Q, \Sigma, \delta, q_0, F)$$

consisting of:

1. Q: a finite set of states.
2. Σ: a finite set of input symbols.
3. $\delta : (q, s) \in Q \times \Sigma \longmapsto q' \in Q$: a transition function (or transition table) specifying, for each state q and each input symbol s, the next state q' of the automaton.

4. $q_0 \in Q$: the initial state.

5. $F \subset Q$: a set of accepting states.

The literature abounds with TLA's[1] to refer to finite state automata (FSA): they are also called *finite state machines* (FSM) or *deterministic finite automata* (DFA).

A finite state automaton can be represented with a table. The rows indicate the states Q, the columns the input symbols Σ, and the table entries the transition function δ. The initial state q_0 is indicated with an arrow $\rightarrow$, and the accepting states F are indicated with a star *.

Example 3.3. Consider the example used in the introduction of this section, namely the search for the sequence "abc" (see Listing 3.1). The corresponding finite state automaton is:

	a	b	c
$\rightarrow q_0$	q_1	q_0	q_0
q_1	q_1	q_2	q_0
q_2	q_1	q_0	q_3
$^*q_3$	q_3	q_3	q_3

3.3.2 State Diagrams

A finite state automaton can also be represented graphically with a *state* or *transition diagram*. Such as diagram is a graph where the nodes represent the states and the links between nodes represent the transitions. The initial state is usually indicated with an arrow, and the accepting states are denoted with double circles.

Example 3.4. The automata given in the previous Example 3.3 can be graphically represented with the state diagram shown in Figure 3.2.

3.3.3 Nondeterministic Finite Automata

The deterministic finite automata introduced so far are clearly an efficient way of searching some sequence in a string. However, having to specify all possible transitions is not extremely convenient. For instance, Figure 3.2 (especially among

[1] Three-Letter Abbreviations

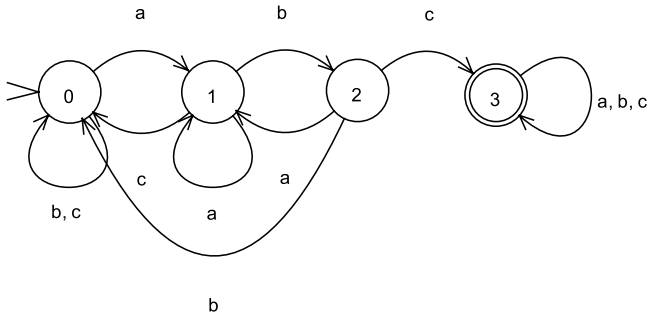


Figure 3.2: State diagram of a finite state automaton accepting the strings containing "abc".

the states q_0, q_1 and q_2) illustrates already with a simple case how this task can quickly become tedious.

A *nondeterministic* finite automaton (NFA) has the power to be in several states at once. For instance, if one state has more than one transition for a given input symbol, then the automaton follows simultaneously all the transitions and gets into several states at once. Moreover, if one state has no transition for a given input symbol, the corresponding state ceases to exist.

Figure 3.3 illustrates how a nondeterministic finite automaton can be substantially simpler than an equivalent deterministic finite automaton.

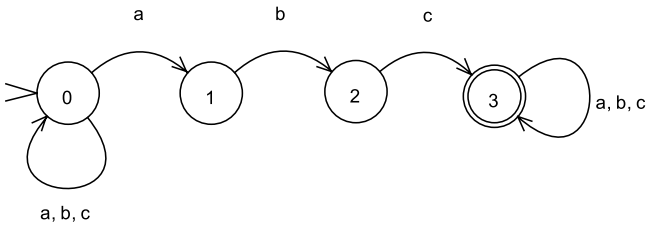


Figure 3.3: State diagram of a nondeterministic finite automaton accepting the strings containing "abc".

Definition 3.4. A *nondeterministic finite automaton* is a five-tuple

$$(Q, \Sigma, \hat{\delta}, q_0, F)$$

consisting of:

1. Q: a finite set of states.
2. Σ: a finite set of input symbols.
3. $\hat{\delta} : (q, s) \in Q \times \Sigma \longmapsto \{q_i, q_j, \ldots\} \subseteq Q$: a transition function (or transition table) specifying, for each state q and each input symbol s, the next state(s) $\{q_i, q_j, \ldots\}$ of the automaton.
4. $q_0 \in Q$: the initial state.
5. $F \subset Q$: a set of accepting states. An input w is accepted if the automaton, after reading the last symbol of the input, is in *at least one* accepting state.

Example 3.5. The transition table $\hat{\delta}$ of the nondeterministic finite automaton shown in Figure 3.3 is the following:

	a	b	c
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\{q_3\}$
$^*q_3$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$

3.3.4 Equivalence of Deterministic and Nondeterministic Finite Automata

Clearly, each deterministic finite automaton (DFA) is already a *nondeterministic* finite automaton (NFA), namely one that happens to always be in just one state. A surprising fact, however, is the following theorem:

Theorem 3.1. *Every language that can be described by an NFA (i.e. the set of all strings accepted by the automaton) can also be described by some DFA.*

Proof. The proof that DFA's can do whatever NFA's can do consists in showing that for every FNA, a DFA can be constructed such that both accepts the same languages. This so-called "subset construction proof" involves constructing all subset of the set of state of the NFA.

In short, since an NFA can only be in a finite number of states simultaneously, it can be seen as a DFA where each "superstate" of the DFA corresponds to a set of states of the NFA.

Let $N = \{Q_N, \Sigma, \hat{\delta}_N, q_0, F_N\}$ be an NFA. We can construct a DFA $D = \{Q_D, \Sigma, \delta_D, q_0, F_D\}$ as follows:

- $Q_D = 2^{Q_N}$. That is, Q_D is the *power set* (the set of all subsets) of Q_N.
- $\delta_D(S, a) = \bigcup_{p \in S} \hat{\delta}_N(p, a)$
- $F_D = \left\{ S \subset Q_N \;\middle|\; S \cap F_N \neq \emptyset \right\}$. That is, F_D is all sets of N's states that include at least one accepting state of N.

It can "easily" be seen from the construction that both automata accept exactly the same input sequences. □

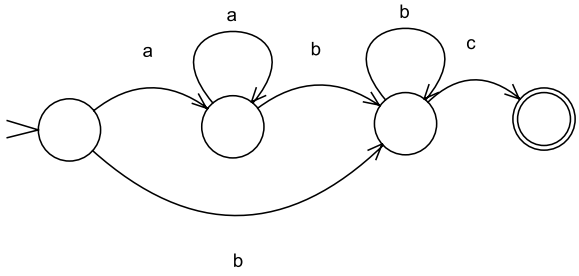
Example 3.6. Figure 3.4 shows an NFA that accepts strings over the alphabet $\Sigma = \{0, 1, 2, \ldots, 9\}$ containing dates of the form "19?0", where ? stands for any possible digit. Figure 3.5 shows an equivalent DFA.

3.3.5 Finite Automata and Regular Languages

We have seen so far different examples of finite automata that can accept strings that contain one or more substrings, such as "abc" or "19?0". Recalling what wew have learned in Chapter 2, we note that the corresponding languages are in fact regular, such as the language corresponding to the regular expression

$$(a \mid b \mid c)^* \; abc \; (a \mid b \mid c)^*$$

More generally, it can be proven that for each regular expression, there is a finite automaton that defines the same regular language. Rather than providing a proof, which can be found for instance in Hopcroft et al. (2001), we only give here an illustrating example. It can be seen that the following NFA defines the same language as the regular expression a^*b^+c:



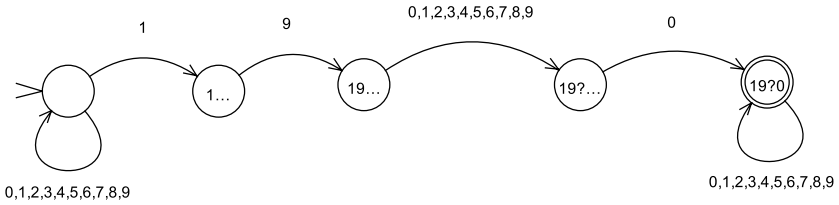


Figure 3.4: An NFA that searches for dates of the form "19?0".

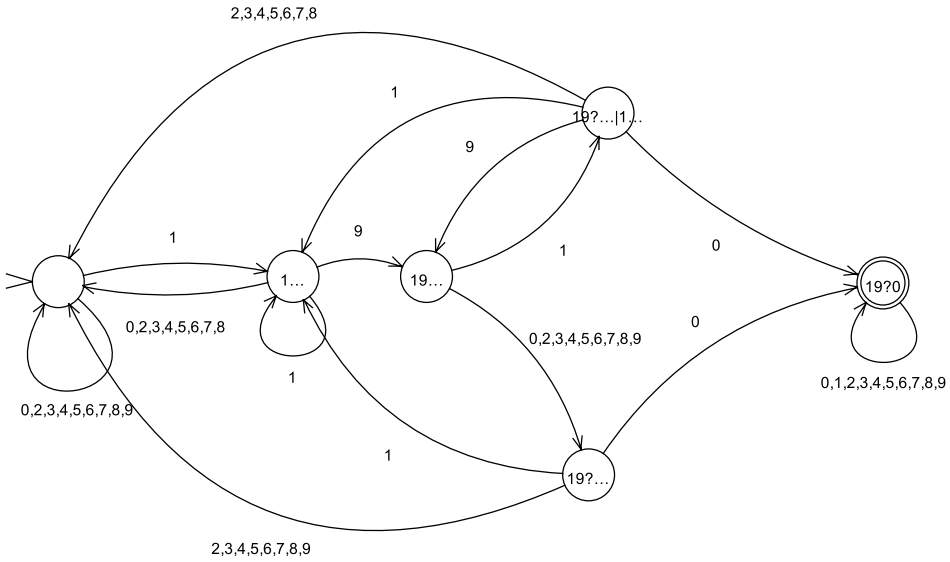


Figure 3.5: Conversion of the NFA from Figure 3.4 to a DFA.

In summary, it can be shown that any regular language satisfies the following equivalent properties:

- It can be generated by a *regular grammar*.
- It can be described by a *regular expression*.
- It can be accepted by a *deterministic finite automaton*.
- It can be accepted by a *nondeterministic finite finite automaton*.

3.3.6 The "Pumping Lemma" for Regular Languages

In the theory of formal languages, a *pumping lemma* for a given class of languages states a property that all languages in the class have – namely, that they can be "pumped". A language can be pumped if any sufficiently long string in the language can be broken into pieces, some of which can be repeated arbitrarily to produce a longer string that is still in the language.

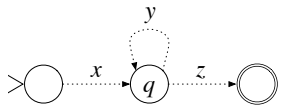
These lemmas can be used to determine if a particular language is not in a given language class. One of the most important examples is the pumping lemma for regular languages, which is primarily used to prove that there exist languages that are not regular (that's why it's called a "lemma" – it is a useful result for proving other things).

Lemma 3.1. *Let L be a regular language. Then, there exists a constant n (which depends on L) such that for every string $w \in L$ with n or more symbols ($|w| \geq n$), we can break w into three strings, $w = xyz$, such that:*

1. $|y| > 0$ *(i.e. $y \neq \epsilon$)*
2. $|xy| \leq n$
3. *For all $k \geq 0$, the string xy^kz is still in L. ($y^k := \underbrace{yy\ldots y}_{k \text{ times}}$)*

In other words, for every string w of a certain minimal length, we can always find a nonempty substring y of w that can be "pumped"; that is, repeating y any number of times, or deleting it (the case of $k = 0$), keeps the resulting string in the language L.

Proof. The proof idea uses the fact that for every regular language there is a finite state automaton (FSA) that accepts the language. The number of states in this FSA are counted, and then, that count is used as the pumping length n. If the string's length is longer than n, then there must be at least one state that is repeated (which we will call state q):



The transitions that take the automaton from state q back to state q match some string. This string is called y in the lemma, and since the machine will match a string without the y portion, or the string y can be repeated, the conditions of the lemma are satisfied. □

We're going to use this lemma to prove that a particular language L_{eq} is not regular. L_{eq} is defined over the alphabet $\{0, 1\}$ as the set of all strings that contain as many 0's and 1's:

$$L_{\text{eq}} = \{\epsilon, 01, 10, 0011, 0101, 1010, 1100, 000111, 001011, \ldots\}$$

We'll use this lemma in a *proof by contradiction*. If we assume that L_{eq} is regular, then the pumping lemma for regular languages must hold for all strings in L_{eq}. By showing that there exists an instance where the lemma does not hold, we show that our assumption is incorrect: L_{eq} cannot be regular.

Statements with multiple quantifiers

Let's first look at what it means for the lemma not to hold:[2]

1. There exists a language L_{eq} assumed to be regular. Then...
2. for all n...
3. there exists a string $w \in L_{\text{eq}}$, $|w| \geq n$ such that...
4. for all possible decompositions $w = xyz$ with $|y| > 0$ and $|xy| \leq n$...
5. there is a $k \geq 0$ so that the string $xy^k z$ is *not* in L_{eq}.

[2]Remember that the opposite of "for all x, y" is "there exists an x so that not y", and vice versa: $\neg(\forall x \Rightarrow y) \equiv (\exists x \Rightarrow \neg y)$ and $\neg(\exists x \Rightarrow y) \equiv (\forall x \Rightarrow \neg y)$.

If is often helpful to see statements with more than one quantifier as a game between two players – one player A for the "there exists", and another player B for the "for all" – who take turns specifying values for the variables mentioned in the theorem. We'll take the stand of player A – being smart enough to find, for each "there exists" statement, a clever choice – and show that it can beat player B for any of his choices.

Player A:	We choose L_{eq} and assume it is regular.
Player B:	He returns some n.
Player A:	We choose the following, particular string: $w = 0^n 1^n$. Obviously, $w \in L_{\text{eq}}$ and $\|w\| \geq n$.
Player B:	He returns a particular decomposition $w = xyz$ with $\|y\| > 0$ and $\|xy\| \leq n$.
Player A:	Since $\|xy\| \leq n$ and xy is at the front of our w, we known that x and y only consist of 0's. We choose $k = 0$, i.e. we remove y from our string w. Now, the remaining string xz does obviously contain less 0's than 1's, since it "lost" the 0's from y (which is not the empty string). Thus, for $k = 0$, $xy^k z \notin L_{\text{eq}}$.

The lemma has been proven wrong. Thus, it can only be our initial assumption which is wrong: L_{eq} cannot be a regular language. □

3.3.7 Applications

Before moving to the next kind of abstract machine, let us briefly list a number of applications of finite state automata:

- Regular expressions
- Software for scanning large bodies of text, such as collections of web pages, to find occurrences of words or patterns
- Communication protocols (such as TCP)
- Protocols for the secure exchange of information
- Stateful firewalls (which build on the 3-way handshake of the TCP protocol)
- Lexical analyzer of compilers

3.4 Push-Down Automata

We have seen in the previous section that finite state automata are capable of recognizing elements of a language such as $L = \{a^n \mid n \text{ even}\}$ – for instance simply by switching between an "odd" state and an "even" state.

Yet they are also proven to be too simple to recognize the elements of a language such as $L = \{a^n b^n \mid n > 0\}$. At least for this language, the reason can be intuitively understood. n can be arbitrarily large, but the finite state automaton only has a finite number of states: it thus cannot "remember" the occurrences of more than a certain number of a's.

The next kind of abstract machine we will study consists of finite state automata endowed with simple external memory – namely a *stack*, whose top element can be manipulated and used by the automaton to decide which transition to take:

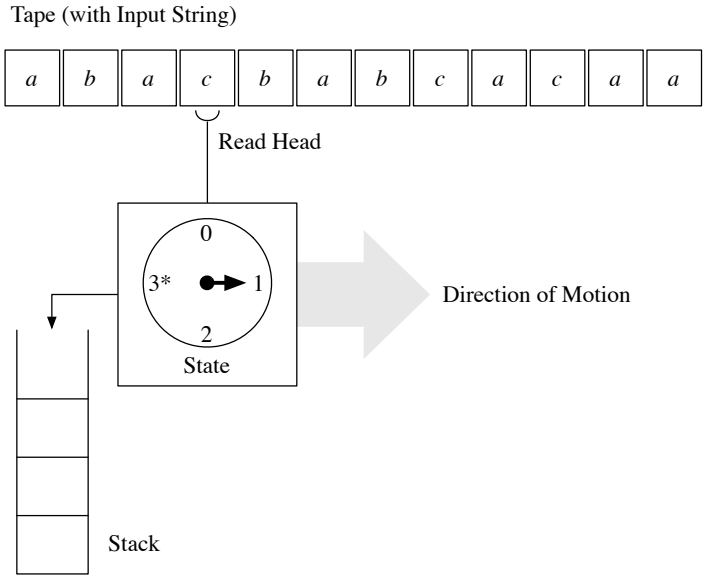


Figure 3.6: Illustration of a push-down automaton.

Push-down automata *per se* offer only little interest: what use is a machine that only recognize whether or not a text – such as the source code of a program – has a correct syntax? Yet, the concept of a push-down automaton lies very close to the concept of an *interpreter*, which not only checks the syntax, but also interprets a text or executes the instructions of a source code.

3.4.1 Parser Generator

Section 2.5.3 introduces the concept of *parsing*, i.e. the process of transforming input strings into their corresponding parse trees according to a given grammar. A *parser* is a component that carries out this task:

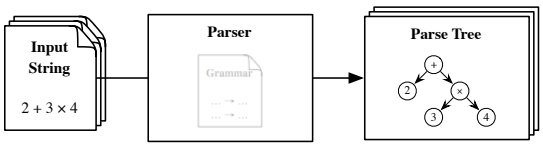


Figure 3.7: A parser transforms input strings into parse trees according to a given grammar.

From an abstract point of view, a parser implements a push-down automaton corresponding to the given grammar. An interesting, and for computer scientists very convenient consequence, which follows from the low complexity of push-down automata, is that it is possible to *automatically* generate a parser by only providing the grammar!

This is in essence what a *parser generator* does:

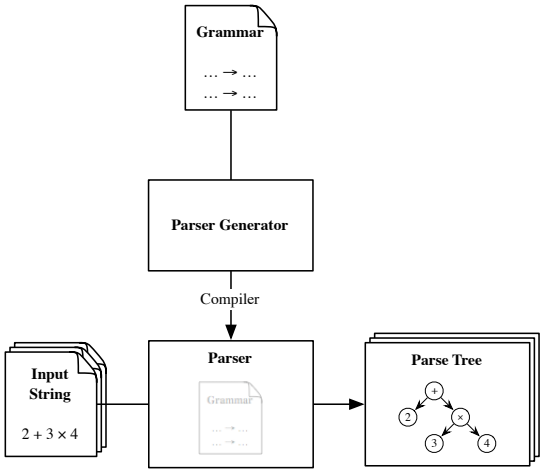


Figure 3.8: A parser generator creates a parser from a given grammar.

Note that the output of a parser generator usually needs to be compiled to create an executable parser.

Example 3.7. `yacc` is a parser generator that uses a syntax close to the EBNF. The following (partial) listing illustrates an input grammar that can be transformed into a parser for simple mathematical expressions using integers, the operators "+", "−" and "×" as well as parenthesis "(" and ")":

```
%token DIGIT

  line : expr
       ;

  expr : term
       | expr '+' term
       | expr '-' term
       ;

  term : factor
       | term 'x' factor
       ;

factor : number
       | '(' expr ')'
       ;

number : DIGIT
       | number DIGIT
```

3.4.2 Compiler Compilers

Even though parsers are more exciting than push-down automata, they nevertheless offer only limited interest, since they can only verify whether input strings comply to a particular syntax.

In fact, the only piece of missing information needed to process a parsing tree is the *semantics* of the production rules. For instance, the semantics of the following production rule

$$T \to T \times F$$

is to multiply the numerical value of the term T with the numerical value of the factor F.

A *compiler compiler* is a program that

- takes as input a grammar complemented with atomic *actions* for each of its production rule, and
- generates a compiler (or to be more exact, a interpreter) that not only checks for any input string the correctness of its syntax, but also evaluates it.

Figure 3.9 illustrates what a compiler compiler does. The very interesting advantage of a compiler compiler is that you only have to provide the specifications of the language you want to implement. In other words, you only provide the "what" – i.e. the grammar rules – and the compiler compiler automatically produces the "how" – i.e. takes care of all the implementation details to implement the language.

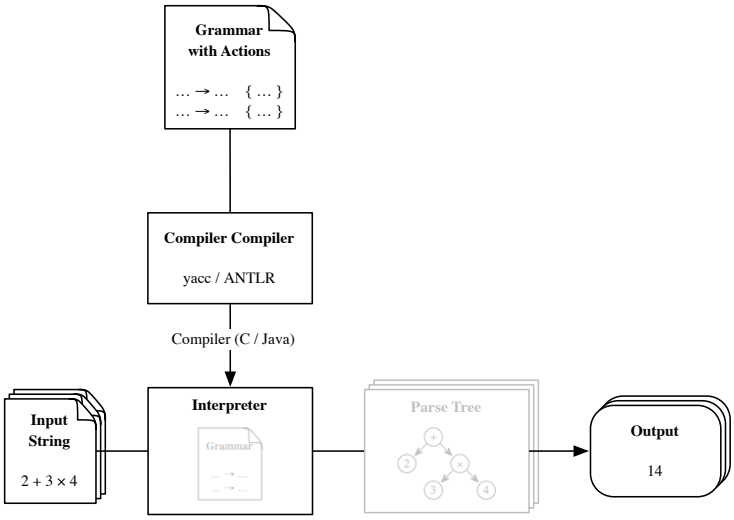


Figure 3.9: Overview of what a compiler compiler does. From a given grammar complemented with atomic actions (the "what"), it automatically generates a interpreter that checks the syntax and evaluates any number of input string (the "how").

Example 3.8. `yacc` is in fact also a compiler compiler (it is the acronym of "yet another compiler compiler"). The following lines illustrates how each production rule of the grammar provided in Example 3.7 can be complemented with atomic actions:

```
line : expr               { printf("result: %i\n", $1); }
     ;

expr : term               { $$ = $1; }
     | expr '+' term      { $$ = $1 + $3; }
     | expr '-' term      { $$ = $1 - $3; }
     ;

term : factor             { $$ = $1; }
```

```
        | term 'x' factor    { $$ = $1 * $3; }
        ;

factor : number             { $$ = $1; }
        | '(' expr ')'      { $$ = $2; }
        ;

number : DIGIT              { $$ = $1; }
        | number DIGIT      { $$ = 10 * $1 + $2; }
```

The point of this example is that the above listing provides in essence all what is required to create a calculator program that correctly evaluate any input string corresponding to a mathematical expression, such as "$2 + 3 \times 4$" or "$((12 - 4) \times 7 + 42) \times 9$".

3.4.3 From yacc to ANTLR

yacc offers a very powerful way of automatically generating a fully working program – the "how" – just from the description of the grammar rules that should be implemented – the "what". Yet, a quick glance at the cryptic code generated by yacc with immediately reveal an intrinsic problem: based on the implementation of a finite state machine, the code is completely opaque to any human programmer. As a result, the code cannot be re-used (we shouldn't expect a full-fledged program to be generated automatically, but only some parts of it), extended or even debugged (in the case that there is an error in the grammar). This severe limitation turned out to be fatal for yacc: nowadays, it is basically only used by the creators of this tool, or by some weird lecturers to implement toy problems in computer science classes.

The landscape has dramatically changed since the emergence of new, modern and much more convenient tools, such as ANTLR[3]. This tool not only provides a very powerful framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions, but also generates a very transparent code. In addition, ANTLR has a sophisticated grammar development environment called ANTLRWorks, which we will be demoing during the lecture. Nowadays, ANTLR is used by hundreds of industrial and software companies.

[3] http://www.antlr.org/

3.4.4 Understanding Parsing as "Consuming and Evaluating"

The code generated by ANTLR is based on a general idea that is quite useful to know. In essence, the code is based on the concept of *parsing* as *consuming and evaluating*. It consists of multiple functions, each implementing a particular grammar rule. Each function parses a portion of the input string by consuming as many characters of the input string as it can, and returning the value corresponding to the evaluation of what has been consumed.

Example 3.9. The grammar rule

```
number : DIGIT              { $$ = $1; }
       | number DIGIT       { $$ = 10 * $1 + $2; }
```

can be implemented by the following code:

```
def scanNumber():
    n = 0
    ok, d = scanDigit()
    if not ok:
        return False, 0
    while ok:
        n = 10 * n + d
        ok, d = scanDigit()
    return True, n

def scanDigit()
    if scanCharacter('0'):
        return True, 0
    if scanCharacter('1'):
        return True, 1

    [...]

    if scanCharacter('9'):
        return True, 9
    return False, 0
```

Each function returns two values: 1) a boolean indicating whether anything could be scanned, and 2) a number corresponding to the evaluation of the scanned characters. The function `scanCharacter(c)` returns whether or not the character `c` could be consumed from the input string (in which case the character is consumed).

Similarly, the following grammar rules

```
  term : factor             { $$ = $1; }
       | term 'x' factor    { $$ = $1 * $3; }
       ;

factor : number             { $$ = $1; }
       | '(' expr ')'       { $$ = $2; }
       ;
```

can be implemented by a code with the same structure:

```
def scanTerm():
    ok, x = scanFactor()
    if not ok: error()
    while scanChar('x'):
        ok, y = scanFactor()
        if not ok: error()
        x = x * y
    return True, x

def scanFactor():
    if scanChar('('):
        ok, x = scanExpression()
        if not ok: error()
        if not scanChar(')'): error()
        return True, x
    else:
        ok, x = scanNumber()
        return ok, x
```

3.5 Turing Machines

Let us go back, for the remaining part of this chapter, to the formal aspect of automata theory.

We have seen that a push-down automata is an abstract machine capable of recognizing context-free languages, such as $L = \left\{a^n b^n \;\middle|\; n > 0\right\}$ – for instance by pushing (i.e. adding) a token on the stack each time an a is read, popping (i.e. removing) a token from the stack eacheach time a b is read, and verifying at the end that no token is left on the stack.

However, push-down automata also have limits. For instance, it can be shown that they cannot recognize the elements of the context-sensitive language $L = \left\{a^n b^n c^n \;\middle|\; n > 0\right\}$ – intuitively, we see that the stack only allows the automaton to verify that there are as many a's and b's, but the machine cannot "remember" what their number was when counting the c's.

Surprisingly, the next type of automaton we're about to (re)discover looks in some respects even simpler than a push-down automaton. The "trick" consists basically in allowing the machine (a) to move in both directions and (b) read *and* write on the tape (see Figure 3.10).

Definition 3.5. A *Turing machine* is an automaton with the following properties:

- A **tape** with the input string initially written on it. Note that the tape can be potentially infinite on both sides, but the number of symbols written at any time on the tape is always finite.

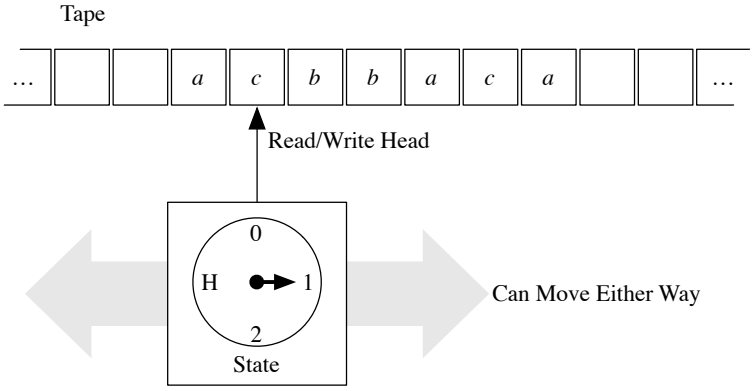


Figure 3.10: Illustration of a Turing machine.

- A **read-write head**. After reading the symbol on the tape and overwriting it with another symbol (which can be the same), the head moves to the next character, either on the left or on the right.

- A **finite controller** that specifies the behavior of the machine (for each state of the automaton and each symbol read from the tape, what symbol to write on the tape and which direction to move next).

- A **halting state**. In addition to moving left or right, the machine may also *halt*. In this case, the Turing machine is usually said to *accept* the input. (A Turing machine is thus an automaton with only one accepting state H.)

The initial position of the Turing machine is usually explicitly stated (otherwise, the machine can for instance start by moving first to the left-most symbol).

Example 3.10. The reader is left with the exercise of finding out what the following Turing machine does on a tape containing symbols from the alphabet $\{a, b, c\}$.

The rows correspond to the state of the machine. Each cell indicates the symbol to write on the tape, the direction in which to move (L or R) and the next state of the machine.

	a	b	c	$\sqcup$
0	a L 0	b L 0	c L 0	$\sqcup$ R 1
1	b R 1	a R 1	c R 1	Halt

3.5.1 Recursively Enumerable Languages

Interestingly, a Turing machine already provides us with a formidable computational power. In fact, it can be shown that Turing machines are capable of recognizing the elements generated by any *unrestricted grammar*. Note however that the corresponding languages aren't called "unrestricted languages" – there still are many languages that just cannot be generated by any grammar! – but *recursively enumerable languages*.

Definition 3.6. A *recursively enumerable set* is a set whose members can simply be numbered. More formally, a recursively enumerable set is a set for which there exist a mapping between every of its elements and the integer numbers.

It is in fact surprising that the three definitions of recursively enumerable languages – namely (a) the languages whose elements can be enumerated, (b) the languages generated by any unrestricted grammar, and (c) the languages accepted by Turing machines – are in fact equivalent!

3.5.2 Linear Bounded Turing Machines

We have seen so far that languages defined by regular, context-free and unrestricted grammars each have a corresponding automaton that can be used to recognize their elements. What about the in-between class of context-sensitive grammars?

The automaton that corresponds to context-sensitive grammars (i.e. that can recognize elements of context-sensitive languages) are so-called *linear bounded Turing machines*. Such a machine is like a Turing machine, but with one restriction: the length of the tape is only $k \cdot n$ cells, where n is the length of the input, and k a constant associated with the machine.

3.5.3 Universal Turing Machines

One of the most astonishing contribution of Alan Turing is the existence of what he called "universal Turing machines".

Definition 3.7. A *universal Turing machine* U is a Turing machine which, when supplied with a tape on which the code of some Turing machine M is written (together with some input string), will produce the same output as the machine M.

In other words, a universal Turing machine is a machine which can be *programmed* to simulate any other possible Turing machine. We now take this remarkable finding for granted. But at the time (1936), it was so astonishing that it is considered by some to have been the fundamental theoretical breakthrough that led to modern computers.

3.5.4 Multitape Turing Machines

There exists many variations of the standard Turing machine model, which appear to increase the capability of the machine, but have the same language-recognizing power as the basic model of a Turing machine.

One of these, the multitape Turing machine (see Figure 3.11), is important because it is much easier to see how a multitape Turing machine can simulate real computers, compared with the single-tape model we have been studying. Yet the extra tapes add no power to the model, as far as the ability to accept languages is concerned.

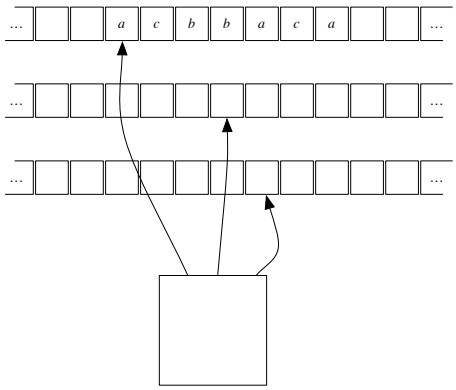


Figure 3.11: Illustration of a multitape Turing machine. At the beginning, the input is written on one of the tape. Each head can move independently.

3.5.5 Nondeterministic Turing Machines

Nondeterministic Turing machines (NTM) are to standard Turing machines (TM) what nondeterministic finite automata (NFA) are to deterministic finite automata (DFA).

A nondeterministic Turing machine may specify any finite number of transitions for a given configuration. The components of a nondeterministic Turing machine, with the exception of the transition function, are identical to those of the standard Turing machine. An input string is accepted by a nondeterministic Turing machine if there is at least one computation that halts.

As for finite state automata, nondeterminism does not increase the capabilities of Turing computation; the languages accepted by nondeterministic machines are precisely those accepted by deterministic machines:

Theorem 3.2. *If* M_N *is a nondeterministic Turing machine, then there is a deterministic Turing machine* M_D *such that* $L(\mathrm{M}_N) = L(\mathrm{M}_D)$.

Notice that the constructed deterministic Turing machine may take exponentially more time than the nondeterministic Turing machine.

The difference between deterministic and nondeterministic machines is thus a matter of time complexity (i.e. the number of steps required), which leads us to the next subsection.

3.5.6 The $\mathrm{P} = \mathrm{NP}$ Problem

An important question is theoretical computer science is how the number of steps required to perform a computation grows with input of increasing length (see Section 3.2.2). While some problems require polynomial time to compute, there seems to be some really "hard" problems that seem to require exponential time to compute.

Another way of expressing "hard" problems is to distinguish between *computing* and *verifying*. Indeed, it can be shown that decision problems solvable in polynomial time on a nondeterministic Turing machine can be "verified" by a deterministic Turing machine in polynomial time.

Example 3.11. The *subset-sum problem* is an example of a problem which is easy to verify, but whose answer is believed (but not proven) to be difficult to compute.

Given a set of integers, does some nonempty subset of them sum to 0? For instance, does a subset of the set $\{-2, -3, 15, 14, 7, -10\}$ add up to 0? The answer is yes, though it may take a while to find a subset that does, depending on its size. On the other hand, if someone claims that the answer is "yes, because $\{-2, -3, -10, 15\}$ add up to zero", then we can quickly check that with a few additions.

Definition 3.8. $\mathbf{P}$ is the complexity class containing decision problems which can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time.

Definition 3.9. $\mathbf{NP}$ – nondeterministic, polynomial time – is the set of decision problems solvable in polynomial time on a nondeterministic Turing machine. Equivalently, it is the set of problems whose solutions can be "verified" by a deterministic Turing machine in polynomial time.

The relationship between the complexity classes $\mathbf{P}$ and $\mathbf{NP}$ is a famous unsolved question in theoretical computer science. It is generally agreed to be the most important such unsolved problem; the Clay Mathematics Institute has offered a US$1 million prize for the first correct proof.

In essence, the $\mathbf{P} = \mathbf{NP}$ question asks: if positive solutions to a "yes or no" problem can be verified quickly (where "quickly" means "in polynomial time"), can the answers also be computed quickly?

3.5.7 The Church–Turing Thesis

The computational power of Turing machine seems so large that it lead many mathematicians to conjecture the following thesis, known now as the *Church–Turing thesis*, which lies at the basis of the theory of computation:

> Every function which would naturally be regarded as computable can be computed by a Turing machine.

Due to the vagueness of the concept of effective calculability, the Church–Turing thesis cannot formally be proven. There exist several variations of this thesis, such as the following *Physical Church–Turing thesis*:

> Every function that can be physically computed can be computed by a Turing machine.

3.5.8 The Halting Problem

It is nevertheless possible to formally define functions that are not computable. One of the best known example is the halting problem: given the code of a Turing machine as well as its input, decide whether the machine will halt at some point or loop forever.

It can be easily shown – but constructing a machine that solves the Halting problem on itself – that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

It is interesting to note that adjectives such as "universal" have often lead to the misconception that "Turing machines can compute everything". This is a truncated statement, missing "...that is computable" – which does actually nothing else than defining the term "computation"!

3.6 The Chomsky Hierarchy Revisited

We have seen throughout Chapters 2 and 3 that there is a close relation between languages, grammars and automata (see Figure 3.12). Grammars can be used to describe languages and generate their elements. Automata can be used to recognize elements of languages and to implement the corresponding grammars.

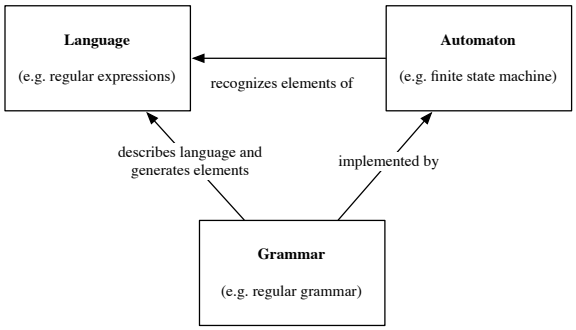


Figure 3.12: Relation between languages, grammars and automata.

The Chomsky hierarchy we have came across in Section 2.7 can now be completed, for each type of formal grammar, with the corresponding abstract machine that recognizes the elements generated by the grammar. It is summarized in Table 3.1.

Type	Grammar	Language	Automaton
0	Unrestricted $\alpha \rightarrow \beta$	Recursively Enumerable e.g. $\{a^n \mid n \in \mathbb{N}, n \text{ perfect}\}$	Turing Machine
1	Context-Sensitive $\alpha A \beta \rightarrow \alpha \gamma \beta$	Context-Sensitive e.g. $\{a^n b^n c^n \mid n \geq 0\}$	Linear Bounded Turing Machine
2	Context-Free $A \rightarrow \gamma$	Context-Free e.g. $\{a^n b^n \mid n \geq 0\}$	Push-Down Automaton
3	Regular $A \rightarrow \epsilon \mid a \mid aB$	Regular e.g. $\{a^m b^n \mid m, n \geq 0\}$	Finite State Automaton

Table 3.1: The Chomsky hierarchy.

3.7 Chapter Summary

- *Computation* is formally described as a "yes/no" decision problem.
- *Finite State Automata, Push-Down Automata and Turing Machines* are abstract machines that can recognize elements of regular, context-free and recursively enumerable languages, respectively.
- A *State Diagram* is a convenient way of graphically representing a finite state automaton.
- *Parsers* are concrete implementations of push-down automata, which not only recognize, but also evaluate elements of a language (such as mathematical expressions, or the code of a program in C).
- *Compiler Compilers* are powerful tools that can be used to automatically generate parts of your program. They only require a definition of the "what" (a grammar with actions) and take care of all details necessary to generate the "how" (some functional code that can correctly parse input strings).
- *Computation* can also be defined as all what a Turing machine can compute. There are however some well-known "yes/no" decision problems that cannot be computed by any Turing machine.