Lecture Notes

Computability

Spring 2021

Rudy van Vliet

Universiteit Leiden

28 January, 2021

Contents

Chapter 1

Preface

These lecture notes are meant to accompany the lectures of the course Computability, Spring 2021. They are based on the lecture slides of this course, which are in turn based on the book

> John C. Martin, Introduction to Languages and the Theory of Computation, 4th edition, McGraw Hill, 2010/2011.

Because of this two-step relation with the book, many definitions, results and exercises in these lecture notes are literal copies from the book. We even use the same numbering of definitions, results and exercises, to allow the reader to easily look up the original text for further or alternative explanation.

The course deals with selected topics from Chapters 7, 8 and 9 from the book:

7 Turing machines

8 Recursive(ly enumerable) languages / general grammars

9 Undecidable problems

Due to the limited time available to write these lecture notes, they do not contain much more than the prior knowledge that students following this course are assumed to have.

Chapter 2

Prior Knowledge

From Automata Theory

1.4 Languages

An *alphabet* Σ is a finite set of symbols, e.g., $\Sigma = \{a, b\}$, $\Sigma = \{a, b, c\}$ or $\Sigma = \{0, 1\}$. The set of all finite strings over an alphabet Σ is denoted by Σ^*. For example,

$$\{a, b\}^* = \{\Lambda, a, b, aa, ab, ba, ba, aaa, \ldots\}$$

Here, the symbol Λ denotes the empty string, i.e., the string consisting of zero symbols. Note that, although the elements of Σ^* have finite length, there are infinitely many of them. A *language* L over an alphabet Σ is a subset of Σ^*. Nothing more, nothing less. For example, Σ^* itself is a language over Σ, and so is the empty set $\emptyset$. Also, the set $\{\Lambda\}$ is a language (over any alphabet Σ).

Note the difference between the empty string Λ, the empty set $\emptyset$, and the set $\{\Lambda\}$ consisting of the empty string only. The empty string Λ is a string, whereas $\emptyset$ and $\{\Lambda\}$ are sets of strings, and thus languages. The set $\emptyset$ contains zero strings, whereas $\{\Lambda\}$ contains one string, the empty string.

The canonical order is an ordering of strings, where shorter strings come before longer strings, and where strings of the same length are 'alphabetically' ordered. For example,

$$\Lambda, a, b, aa, ab, ba, ba, aaa, \ldots$$

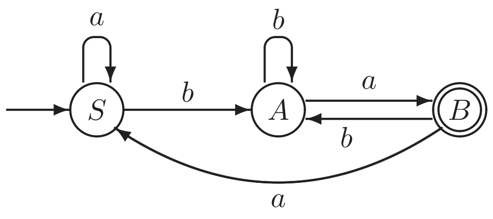
is a list of the ('first') elements of Σ^* in canonical order. The canonical order is defined for any language. For example, the (first) elements of $Pal = \{x \in$

$\{a,b\}^* \mid x = x^r\}$ (the set of palindromes over $\{a,b\}$) in canonical order are:

$$\Lambda, a, b, aa, bb, aaa, aba, bab, bbb, aaaa, \ldots$$

2.1 Finite Automata

The simplest model of computation we studied is the finite automaton (FA). It has a finite number of states. These states are the only type of memory the automaton has. That is, the only way for the automaton to remember anything relevant about the input read so far, is by the state it is in.
Example.
An FA accepting $\{a,b\}^*\{ba\}$



This finite automaton accepts all strings in $\{a,b\}^*$ that end with ba. The tree states keep track of how much of the desired suffix ba we have read so far. In state S, we have not read anything of this suffix. In state A, we have just read b. In state B, we have just read ba. That is, the string we have read so far, ends with ba, and thus we can accept this string. That is why B is the (only) accepting state. ■

2.4 The Pumping Lemma

Regular languages are the languages that can be accepted by a finite automaton. The pumping lemma for regular languages intuitively states that any regular language can be ‘pumped up’. In particular, if a language $L \subseteq \Sigma^*$ is accepted by a finite automaton M having n states, then each string $x \in L$ with $|x| \geq n$ can be pumped up in the first n letters. To be precise: there exist $u, v, w \in \Sigma^*$, such that

1. $x = uvw$ (i.e., x can be split into u, v and w)

2. $|uv| \leq n$ (in particular, substring v occurs within the first n letters of x)

3. for $i = 0, 1, 2, \ldots$, $uv^iw \in L$ (i.e., we can pump up (or down) v)

The proof of this pumping lemma is very intuitive:
Since x is accepted by M, x follows a path through M ending in an accepting state. M has only n different states, whereas x consists of at least n letters. Hence, the first n steps of the path (actually, any n steps of the path) through M for x must visit at least one state multiple times. Let p be such a state, and let v be the substring of x traversed between the first and the second visit of state p. Let u and w be the substrings of x, before and after v, respectively. Indeed $x = uvw$, and $|uv| \leq n$.

Then instead of the path through M for $x = u \cdot v \cdot w$, we could as well traverse the path for $u \cdot w$ (skipping the subpath from state p back to itself), for $u \cdot v \cdot v \cdot w$ (traversing this subpath two times), for $u \cdot v \cdot v \cdot v \cdot w$ (traversing this subpath three times), etcetera. Each of these alternative paths ends in the same state as the path for x, which is an accepting state since $x \in L$. We conclude that all strings uv^iw are accepted by M and thus are elements of L. □

Most often, the pumping lemma for regular languages is used to prove that some language L is *not* regular, simply because it does *not* satisfy the pumping lemma. To this end, one shows that pumping up (and/or down) a particular string $x \in L$ yields a string that is *not* an element of L, while it should be according to the pumping lemma.

Example.
The language $L = AnBn = \{a^ib^i \mid i \geq 0\}$ is not regular, because pumping the string $x = a^nb^n$ (with n the size of the finite automaton from the pumping lemma) would yield a string with fewer (if pumped down) or more (if pumped up) than n a's, but still n b's. ■

Example.
The language $SimplePal = \{xcx^r \mid x \in \{a, b\}^*\}$ is not regular, because pumping the string $x = a^nca^n$ (again, with n the size of the finite automaton from the pumping lemma) would yield a string with fewer (if pumped down) or more (if pumped up) than n a's to the left of the c, but still n a's to the right of the c. ■

3.1 Regular Languages and Regular Expressions

A regular language over an alphabet Σ is a language derived from the empty language $\emptyset$ and the languages $\{a\}$ (with $a \in \Sigma$), by means of the operators union, concatenation and Kleene star.

A regular expression over an alphabet Σ is an expression over the empty expression and expressions a (with $a \in \Sigma$), by means of the operators $+$, concatenation and Kleene star. In principle, the order of application of the operators in a regular expression is determined by brackets. Brackets may, however, be left out when they are redundant with respect to the following rules of precedence: Kleene star comes before concatenation, which in turn comes before $+$.

Each regular language can be described by a regular expression. Usually, there are more than one regular expressions describing the same regular language. Conversely, each regular expression describes a unique regular language.

Example.
An example of a regular language is $\{a, b\}^*\{ba\}$. A regular expression describing this language is $(a + b)^*ba$. Another regular expression describing the same language is $(b + a)^*ba$. ■

3.2 Nondeterministic Finite Automata

By default, a finite automaton is deterministic, which means that for each state p and each symbol a from the input alphabet Σ, there is one transition from p with label a (to some state q). There do not exist Λ-transitions. In other, more formal terms, we have a transition *function* $\delta : Q \times \Sigma \to Q$, where Q is the set of states of the finite automaton.

A *nondeterministic finite automaton* (NFA) is an extension of a finite automaton. For each state p and each symbol a from the input alphabet Σ, it has zero or more transitions from p with label a (to different states q). Each state p may also have zero or more Λ-transitions (to different states q). In other, more formal terms, we have a transition function $\delta : Q \times (\Sigma \cup \{\Lambda\}) \to 2^Q$, where 2^Q is the set of all subsets (including $\emptyset$) of Q.

3.3 The Nondeterminism in an NFA Can Be Eliminated

As said, a *nondeterministic finite automaton* (NFA) is an extension of a finite automaton. Each FA is (in fact) also an NFA. Hence, each regular language, i.e., each language that is accepted by an FA, can also be accepted by an NFA. It is less obvious that each language that is accepted by an NFA can also be accepted by an FA. One might imagine that NFAs are 'stronger' than FAs, i.e. that NFAs can accept more languages than just the regular languages. This is, however, not the case.

Using the so-called subset construction, each NFA M can be transformed into an FA M' accepting the same language. The term 'subset construction' refers to the states in the FA M' that is constructed: the states of M' correspond to subsets of states of M, namely all states that M may be in after reading a certain string.

3.4 Kleene's Theorem, Part 1

In the foregoing, we already mixed up the terms 'language accepted by a finite automaton' and 'regular language'. It is not obvious that each regular language as defined in Section 3.1 can be accepted by a finite automaton and vice versa. Kleene's theorem states that this is the case, indeed.

Part 1 of the theorem states that each regular language L over an alphabet Σ can be accepted by a finite automaton. The proof is by induction on the structure of the regular expression describing L. First, we give NFAs (in fact, FAs) accepting the basic regular languages $\emptyset$ and $\{a\}$ (with $a \in \Sigma$). Next, given two NFAs M_1 and M_2 accepting regular languages L_1 and L_2, respectively, we describe how to construct an NFA accepting $L_1 \cup L_2$, an NFA accepting $L_1 \cdot L_2$ and an NFA accepting L_1^* (or L_2^*). As each regular language is the result of a finite number of applications of (some of) these operations on the basic regular language, each regular language can be accepted by an NFA. But then each regular language can also be accepted by an FA (see Section 3.3).

3.5 Kleene's Theorem, Part 2

Part 2 of Kleene's theorem states that each language that is accepted by a finite automaton, can be described by a regular expression, and thus is a regular language. Indeed, the terms 'language accepted by a finite automaton' and 'regular language' are synonyms.

4.2 Context-Free Grammars

A context-free grammar G is a tuple (V, Σ, S, P) where V and Σ are two disjoint sets of symbols, V is the set of variables (or non-terminals), Σ is the set of terminals, $S \in V$ is the start variable and P is the set of productions. Each production is of the form $A \to \beta$, where $A \in V$ and $\beta \in (V \cup \Sigma)^*$.

A production $A \to \beta$ can be applied to a string α, which means that an occurrence of the variable A in α is replaced by β. For example, if $\alpha = \alpha_1 A \alpha_2$, then we may write: $\alpha_1 A \alpha_2 \;\Rightarrow\; \alpha_1 \beta \alpha_2$. The language generated by G is the set of strings $x \in \Sigma^*$ that can be derived from the start variable S by successively applying productions from P. This language is denoted by $L(G)$. A language generated by a context-free grammar is called a context-free language.

Example.
The context-free grammar (V, Σ, S, P), with $V = \{S\}$, $\Sigma = \{a, b, c\}$ and the following productions:

$$S \to aSa \mid bSb \mid c$$

generates the language $\mathit{SimplePal} = \{xcx^r \mid x \in \{a, b\}^*\}$. For example, the string *abbcbba* is derived as follows:

$$S \;\Rightarrow\; aSa \;\Rightarrow\; abSba \;\Rightarrow\; abbSbba \;\Rightarrow\; abbcbba$$

■

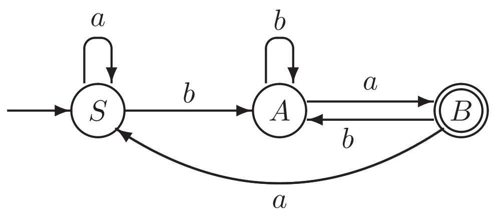
This example illustrates that context-free grammars can generate languages that are non-regular (see Section 2.4).

4.3 Regular Languages and Regular Grammars

A regular grammar (V, Σ, S, P) is a context-free grammar whose productions are restricted to two general forms: either $A \to \sigma B$ or $A \to \Lambda$, where $A, B \in V$ and $\sigma \in \Sigma$. For example, a production $S \to aSA$ is not allowed in a regular grammar.

A finite automaton $(Q, \Sigma, q_0, A, \delta)$ accepting a language L can easily be transformed into a regular grammar (V, Σ, S, P) generating the same language. Just take $V = Q$, $S = q_0$,
for each $X, Y \in V$ and $\sigma \in \Sigma$, if $\delta(X, \sigma) = Y$, then add production $X \to \sigma Y$ to P,
and for each $X \in A$ (i.e., each accepting state X), add production $X \to \Lambda$ to P.

Example.
Consider again the following FA accepting $\{a, b\}^*\{ba\}$:



The construction described yields the regular grammar $(\{S, A, B\}, \{a, b\}, S, P)$, with productions

$$S \to aS \mid bA \qquad A \to bA \mid aB \qquad B \to bA \mid aS \mid \Lambda$$

■

With exactly the converse construction, each regular grammar generating a language L, can be transformed into a (nondeterministic!) finite automaton accepting the same language.

This implies that regular grammars can generate precisely the regular languages. Because the class of regular grammars is a subset of the class of context-free grammars, all regular languages can be generated by context-free grammars. However, as we have seen in Section 4.2, context-free grammars can also generate non-regular languages.

4.4 Derivation Trees

A derivation tree represents the structure of a string derived in a context-free grammar. For each production $A \to X_1 \ldots X_n$ applied in the derivation, the corresponding node labelled by A has n ordered children labelled by $X_1, \ldots, X_n$ respectively, read from left to right. If $n = 0$ (i.e., the production is $A \to \Lambda$), the node labelled by A has a child labelled by Λ.

4.5 Simplified Forms and Normal Forms

As said, the class of regular grammars is a subset of the class of context-free grammars. A subset that can generate only regular languages.

There exist other classes of context-free grammars which can generate (nearly) all context-free languages. One such class consists of the context-free grammars in Chomsky normal form. In such a grammar (V, Σ, S, P), the productions are (again) restricted to two general forms: either $A \to BC$ or $A \to \sigma$, where $A, B, C \in V$ and $\sigma \in \Sigma$. It can be proved that for each context-free grammar G, there exists a context-free grammar G' in Chomsky normal form, such that $L(G') = L(G) \setminus \{\Lambda\}$. That is, apart from the empty string, $L(G)$ can be generated by a context-free grammar in Chomsky normal form.

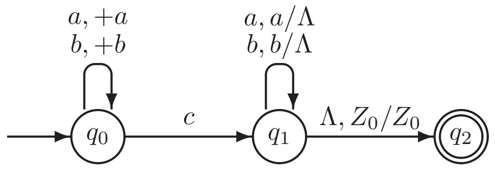
5.1 Definitions and Examples (of Pushdown Automata)

A pushdown automaton (PDA) is a finite automaton, extended with a stack. Initially, the stack contains only the initial stack symbol Z_0. Transitions in the automaton not only depend on the current state and the next input letter to be read, but also on the current top symbol on the stack. As a result of a transition, this top symbol is replaced by a string of symbols. If this string is empty, the top symbol is effectively just popped from the stack.

By default, pushdown automata may have Λ-transitions, and may be nondeterministic: there may be more than one transitions from the same state on the same input letter (or Λ) and the same top stack symbol.

A second source of nondeterminism is the combination of Λ-transitions and 'letter-transitions': for a given state and a given top stack symbol, there may be both Λ-transitions and transitions with an input letter. When, in this case, the next letter to be read is indeed this input letter, we may (or may not) postpone reading this letter by following a Λ-transition.

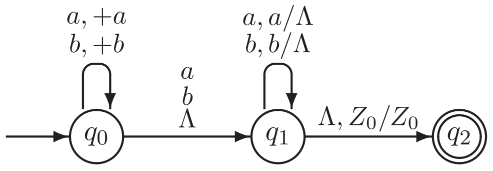
Example 5.3 A Pushdown Automaton Accepting $\mathit{SimplePal} = \{xcx^r \mid x \in \{a,b\}^*\}$.



At the initial state q_0, symbols a and b read from the input are pushed onto the stack. When the special middle symbol c is read, the PDA moves to state q_1 without altering the stack. At state q_1 symbols are popped from the stack, only if they equal the next input symbol. Otherwise, the PDA just crashes. When all symbols a and b have been popped from the stack (the top symbol is Z_0, again), the PDA moves to accepting state q_2 with a Λ-transition. ■

Example 5.7 A Pushdown Automaton Accepting *Pal*

$$Pal = \{x \in \{a,b\}^* \mid x = x^r\}$$



The elements of language *Pal* do not have a special middle symbol c. Hence, there is no way for the PDA to recognize what the middle of the word is. Therefore, while pushing symbols onto the stack in state q_0, it nondeterministically decides what to do with the next input symbol, say a:

- The PDA may assume that a is still part of the first half of the input string, and therefore also push it onto the stack in state q_0.
- The PDA may assume that a is the middle symbol of the input string (which means that the length of the input string is odd), and move to state q_1, while reading a.
- The PDA may assume that a is the first symbol of the second half of the input string (which means that the length of the input string is even), and move to state q_1 without reading a yet (corresponding to the Λ-transition.

The options for input symbol b in state q_0 are the same. Once the PDA has reached state q_1, it continues its operation in exactly the same way as the PDA in Example 5.3. ■

5.2 Deterministic Pushdown Automata

As said before, by default, a pushdown automaton may be nondeterministic. A *deterministic* pushdown automaton is a pushdown automaton in which

- for each combination of state q, stack symbol X and input symbol a, there is at most one transition, and

- for each combination of state q and stack symbol X, if there exists an Λ-transition, then there does not exist any transition for this state q, stack symbol X and any input symbol a.

The PDA from Example 5.3 is deterministic. On the other hand, the PDA from Example 5.7 is not deterministic. There are, e.g., two transitions in state q_0, any stack symbol and input symbol a. Moreover, there is an Λ-transition from q_0 to q_1 for any stack symbol.

There exist languages that can be accepted by a PDA but cannot be accepted by a deterministic PDA. An example of such a language is *Pal*, that was accepted by the PDA in Example 5.7.

Hence, unlike for finite automata, a deterministic pushdown automaton is not equally powerful as a (general) pushdown automaton.

5.3 A PDA from a Given CFG

Every context-free language can be accepted by a pushdown automaton. In Automata Theory, we have seen two constructions from a given CFG G to a PDA accepting $L(G)$:

- a construction yielding the so-called nondeterministic top-down PDA corresponding to G, $NT(G)$.
- a construction yielding the so-called nondeterministic bottom-up PDA corresponding to G, $NB(G)$.

Students following the course Compiler Construction may recognize $NT(G)$ in the top-down parser and $NB(G)$ in the bottom-up parser occurring in that course.

5.4 A CFG from a Given PDA

Every language that can be accepted by a pushdown automaton can be generated by a context-free grammar. This implies that the context-free languages are exactly the languages that can be accepted by PDAs. Recall, however, that there exist languages, like *Pal*, that can be accepted by a PDA, but not by a deterministic PDA. Hence, not all context-free languages can be accepted by deterministic PDAs.

In the course Automata Theory, it is explained how a CFG can be constructed from a given PDA M. You may remember that this was a pretty complicated construction.

6.5 The Pumping Lemma for Context-Free Languages

Although the class of context-free languages is larger than the class of regular languages, there are still many languages that are not context-free. For example, it is plausible that there cannot be a PDA accepting the language $AnBnCn = \{a^i b^i c^i \mid i \geq 0\}$, and hence no CFG generating the language. A PDA is able to count the a's on its stack, by pushing a symbol onto the stack for every a it reads. However, in order to check that the number of b's following the a's is equal, it has to pop all these symbols. By then, it has no way to check that also the number of c's following the b's is equal.

It is also plausible that there cannot be a PDA accepting the language $XX = \{xx \mid x \in \{a,b\}^*\}$. A PDA might push all symbols of the first half of its input string onto the stack. It may even guess the middle of the string. By then, the only way to compare the first letter of the second half of the string to the first letter of the first half of the string (which resides at the bottom of the stack), is by removing all other symbols from the stack. After that the PDA cannot compare the other symbols of the second half to the other symbols of the first half of the string, anymore.

In Section 2.4, we used the pumping lemma for regular languages to prove that certain languages are not regular. We use the pumping lemma for context-free languages to prove that certain languages are not context-free.

The pumping lemma for regular languages was based on the path through a finite automaton for a long sting x. In contrast, the pumping lemma for context-free languages is not based on the operation of a PDA accepting a language, but on the derivation of a string in a CFG.

If a string derived with the grammar is long enough, there should be a nonterminal A in this derivation that generates itself, together with some non-empty substrings. That is, there should be a derivation of our string, containing the following sentential forms:

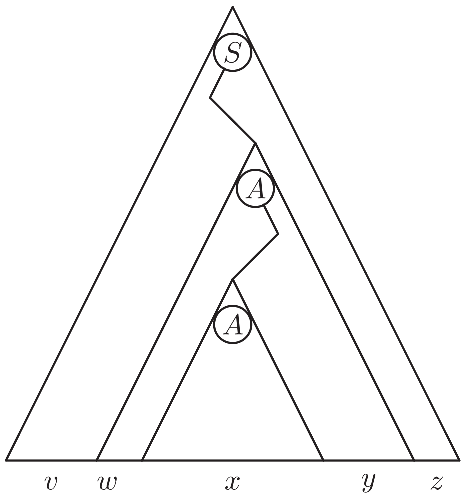
$$S \Rightarrow^* v\underline{A}z \Rightarrow^* v\ w\underline{A}y\ z \Rightarrow^* vw\ x\ yz$$

Apparently the substring wAy can be derived from A. But then this part of the derivation can be repeated, over and over again:

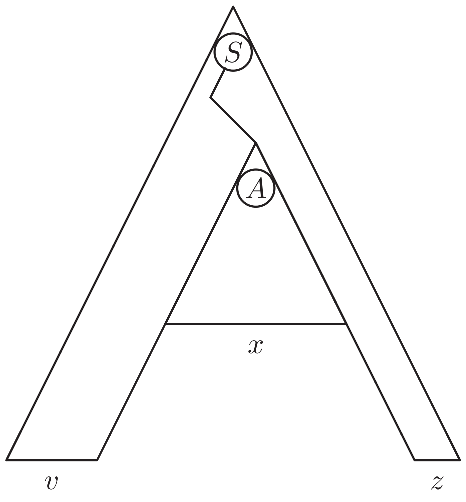
$$S \Rightarrow^* v\underline{A}z \Rightarrow^* v\ w\underline{A}y\ z \Rightarrow^* vw\ wAy\ yz \Rightarrow^* vw^m xy^m z$$

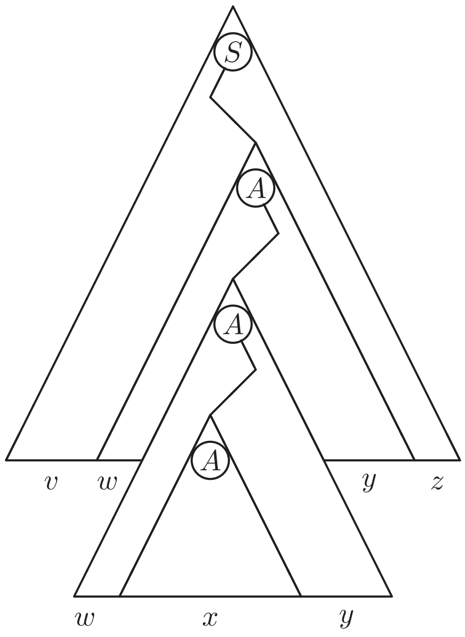
The underlying cause of this is exactly the *context-freeness* of the grammar: no matter what the context is in which the variable A occurs, we can derive the same strings from it. It does not matter if it occurs in vAz or in $vwAyz$.

We can also visualize this argument with derivation tree. If the sting derived is long enough, then there must be a node in the tree labelled by a variable A, which has as a descendent another node labelled by A, as in



But then we could obtain another valid derivation tree by replacing one subtree rooted by A by the other. This may result in the following new derivation trees:





In the left picture, we have replaced the larger subtree by the smaller subtree, yielding a shorter string *vxz* In the right picture, we have replaced the smaller subtree by the larger subtree, yielding a longer string *vwwxyyz*. This way, we can pump up (or down, as in the first case) a string.

A formal description of this result is:

Theorem 6.1 *The Pumping Lemma for Context-Free Languages*

Suppose L is a context-free language. Then there is an integer n so that for every $u \in L$ with $|u| \geq n$, u can be written as $u = vwxyz$, for some strings v, w, x, y and z satisfying

1. $|wy| > 0$
2. $|wxy| \leq n$
3. *for every* $m \geq 0$, $vw^m xy^m z \in L$

In Automata Theory, this pumping lemma was used to prove that, a.o., both the language *AnBnCn* and the language *XX* are not context-free. There are many more non-context-free languages.

Chapter 7

Turing Machines

Till here, we have seen three classes of languages: regular languages, deterministic context-free languages, and (general) context-free languages. Each class of languages corresponds to a certain type of machine that can accept exactly those languages. Regular languages and context-free languages can be generated by regular grammars and context-free grammars, respectively. Finally the regular languages can be specified by regular expressions. We thus have the following situation:

reg. languages	FA	reg. grammar	reg. expression
determ. cf. languages	DPDA		
cf. languages	PDA	cf. grammar	

The regular languages form a proper subset of the deterministic context-free languages, which in turn form a proper subset of the context-free languages. However, as we have seen in Section 6.5, there are many languages which are not even context-free.

In particular, we mentioned $AnBnCn = \{a^ib^ic^i \mid i \geq 0\}$ and $XX = \{xx \mid x \in \{a,b\}^*\}$. A 'simple version' of XX is the language $L = \{xcx \mid x \in \{a,b\}^*\}$ This language is not context-free either.

It should not really come as a surprise that these languages cannot be accepted by a pushdown automaton, i.e., by a finite automaton augmented with a stack. As for $AnBnCn$, in order to check that the number of a's equals the number of b's, we can push an a onto the stack for each a that we read from input, and pop an a from the stack for each b that we read subsequently. By then, the stack is empty (apart from the initial stack symbol), and we have no way to check the number of c's that follow. If, instead of one stack, we had two stacks at our disposal, we could easily accept $AnBnCn$ (how?).

Similarly, suppose we want to use a pushdown automaton to check if a string in $\{a, b, c\}^*$ is an element of $L = \{xcx \mid x \in \{a, b\}^*\}$. We can easily push all symbols we read, in the order of reading, onto the stack, until we read a c. The symbols read before the c then correspond to the first occurrence of x in the definition of L. We now have to check that the subsequent symbols, the symbols after the c, form the same string x. In particular, the first symbol we read after the c should be equal to the first symbol of x. This symbol, however, resides at the bottom of the stack. In order to look up this symbol, in order to know which symbol it is, we would have to remove all other symbols of x from the stack. After having done that, we have no way to verify that the next symbols we read equal the second and later symbols of x. If, instead of a stack, we had a queue at our disposal as auxiliary memory, we could easily accept the language L (how?).