2 Context-free Grammars and Pushdown Automata

As we have seen in Exercise 15, regular languages are not even sufficient to cover well-balanced brackets. In order even to treat the problem of parsing a program we have to consider more powerful languages. Our aim in this section is to generalize the concept of a regular language to cover this example (and others like it), and to find ways of describing such languages. The analogue of a regular expression is a *context-free grammar* while finite state machines are generalized to *pushdown automata*. We establish the correspondence between the two and then finish by describing the kinds of languages that are not captured by these more general methods.

2.1 Context-free grammars

The question of recognizing well-balanced brackets is the same as that of dealing with nesting to arbitrary depths, which certainly occurs in all programming languages. To cover these examples we introduce the following definition.

Definition 11 *A* **context-free grammar** *is given by the following:*

- *an alphabet* Σ *of* terminal symbols*, also called the* object alphabet*;*
- *an alphabet* N *of* non-terminal symbols*, the elements of which are also referred to as* auxiliary characters*,* placeholders *or, in some books,* variables*, where* $N \cap \Sigma = \emptyset$*;*[7]
- *a special non-terminal symbol* $S \in N$ *called the* start symbol*;*
- *a finite set of* production rules*, that is strings of the form* $R \to \Gamma$ *where* $R \in N$ *is a non-terminal symbol and* $\Gamma \in (\Sigma \cup N)^*$ *is an arbitrary string of terminal and non-terminal symbols, which can be read as 'R can be replaced by* Γ*'.*[8]

You will meet grammars in more detail in the other part of the course, when examining parsers. Here is a simple grammar for arithmetic expressions built from numbers out of a given range. Let Σ contain the symbols (,), $+$, $-$, $\times$, / and the numbers in the desired range, and let $N = \{S, B\}$. The production rules are as follows.

$$\begin{array}{ll} B \to 0 & S \to B \\ B \to 1 & S \to (S) \\ B \to 2 & S \to S + S \\ \ldots & S \to S - S \\ & S \to S \times S \\ & S \to S/S \end{array}$$

We sometimes use the following shorter way of expressing the same rules.

$$\begin{array}{rcl} B & \to & 0 \,|\, 1 \,|\, 2 \,|\, \ldots \\ S & \to & B \,|\, (S) \,|\, S + S \,|\, S - S \,|\, S \times S \,|\, S/S \end{array}$$

[7] There are ways of avoiding that stipulation, but they complicate matters without giving additional insight.

[8] Once again we have a special symbol, namely $\to$ which should not be contained in either Σ or N to avoid confusion.

It may help to view the elements of N as auxiliary letters, or placeholders. The only strings we are really interested in consist entirely of characters from Σ. The non-terminal symbols in N serve to give us rules for generating such strings. We always begin with the start symbol S. The production rules then tell us how we may rewrite that expression. We stop when no non-terminal symbols remains. Here is an example for such a rewriting process for the language or arithmetic expressions given above.

$$S \Rightarrow S \times S \Rightarrow S \times (S) \Rightarrow B \times (S) \Rightarrow B \times (S + S) \Rightarrow B \times (B + S)$$
$$\Rightarrow 2 \times (B + S) \Rightarrow 2 \times (3 + S) \Rightarrow 2 \times (3 + B) \Rightarrow 2 \times (3 + 4)$$

The following definition says how we may generate strings in general.

Definition 12 *A string $\Gamma \in (\Sigma \cup N)^*$* **is generated by a grammar** *G if there is a sequence of strings $S = \Gamma_0 \Rightarrow \Gamma_1 \Rightarrow \cdots \Rightarrow \Gamma_n = \Gamma$ such that each step $\Gamma_i \Rightarrow \Gamma_{i+1}$ is obtained by the application of one of G's production rules to a non-terminal occurring in Γ_i as follows. Let $R \in N$ occur in Γ_i and assume that there is a production rule $R \to \Delta$. Then Γ_{i+1} is obtained from Γ_i by replacing one occurrence of R in Γ_i by the string Δ.*

The **language generated by a grammar** *G is the set of all strings α over Σ which are generated by G.*[9] *Languages generated by context-free grammars are called* **context-free**.

Hence the language generated by the example given above is that of arithmetic expressions, and the symbols S and B have disappeared from those. Each word that is generated by the language has a derivation which shows how it can be arrived at when using the production rules. Rather than generating a string in a linear way as in the above example, it is clearer to construct a *parse tree* instead. Figure 20 gives a parse tree for the expression $2 \times (3+4)$[10]. Note that there are many ways of turning this into a rewriting process, since the order in which the rules are applied is not fixed.

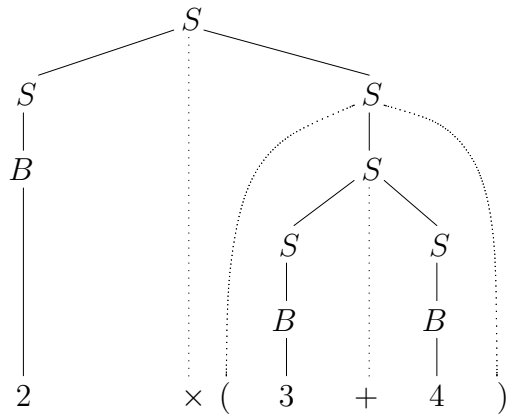

Figure 20: A parse tree

To have a go at designing a grammar, try Exercise 16 and Assessed Exercise 2.1.

[9] Note that such a string may not contain any non-terminal symbols.

[10] Note that there is nothing which forces us to put in the brackets around $3 + 4$, and if we omit these there is more than one parse tree for this expression!

For this definition to fit into our programme we have to ensure first of all that it is indeed more general than that of regular expressions, so we begin by studying the relation between the two.

Proposition 2.1 *Every regular language is generated by a context-free grammar.*

Proof. Let us assume we are given a regular language. Then we can design a DFA $(Q, q_\bullet, F, \delta)$ which recognizes that language, and we have to produce a grammar such that the strings recognized by this DFA form the language generated by the grammar. We leave Σ as it is and set $N = Q$, so every state becomes a non-terminal symbol, and set $S = q_\bullet$. It remains to define the production rules. For every transition

$$q \xrightarrow{x} q'$$

we add a rule

$$q \to xq',$$

and for every accepting state q we add a rule $q \to \epsilon$.

Strings are generated by this grammar as follows. The only way of starting is with $q_\bullet$. This can be replaced by xq for every transition from $q_\bullet$ to q which is labelled by the character x, and xq can then be replaced by xyq' whenever $q \xrightarrow{y} q'$. So every string thus generated consists of a string over Σ followed by one non-terminal symbol, which is a state in Q. The only way of getting rid of that non-terminal symbol is by reaching an accepting state, in which case this symbol vanishes and we are left with a string that has reached an accepting state. Hence the language generated by this grammar is precisely the set of strings accepted by the automaton, which is the given regular language. □

Assessed Exercise 2.2 asks you to apply this algorithm to a concrete example.

Proposition 2.2 *A language is regular if and only if it is generated by a context-free grammar which is subject to the following restriction: All production rules are of the form $R \to xR'$ or $R \to x$ or $R \to \epsilon$, where $R, R' \in N$ and $x \in \Sigma$.*

Proof. The 'if' direction is shown in Proposition 2.1. For the 'only if' direction assume that we have a grammar whose production rules comply with the given restriction. We define an NFA as follows. Take as states the set N of all non-terminals of the grammar plus one extra state, say Z. S is the initial state. A state $R \in N$ is accepting if and only if there is a rule $R \to \epsilon$, and we define the state Z to be accepting too. For R and R' in N there is an arrow labelled with x from R to R' if there is a production rule of the form $R \to xR'$. There is an arrow labelled x from R to Z if and only if there is a rule $R \to x$.

There is a path through the automaton labelled by a word $x_1 \dots x_n \in \Sigma^*$ if and only if there is a sequence of applications of production rules $S \Rightarrow x_1R_1 \Rightarrow \cdots \Rightarrow x_1 \cdots x_nR_n$. Clearly only the last symbol in the strings generated 'along the way' can be a non-terminal symbol, and the word ends once this non-terminal has been replaced by either a terminal symbol or ϵ. Clearly that occurs if and only if the corresponding path through the NFA ends in an accepting state. □

We can also show that context-free grammars are more powerful than regular expressions because we can use them to describe non-regular languages. Here is an example. Recall that

the language $\{a^n b^n \mid n \in \mathbb{N}\}$ is not regular. Consider the grammar where $\Sigma = \{a, b\}$ as before, $N = \{S\}$ and where we have the two production rules

$$S \to aSb \qquad \text{and} \qquad S \to \epsilon.$$

Then every production of a string takes the form

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \ldots \Rightarrow \underbrace{a \cdots a}_{n} S \underbrace{b \cdots b}_{n} \Rightarrow \underbrace{a \cdots a}_{n} \underbrace{b \cdots b}_{n}.$$

Hence every string ever produced consists of n many as (which might be 0 many) followed by as many bs. Clearly every such string can be created in this way. Therefore the given grammar does indeed generate the language specified.

For more examples of the kinds of languages for which one can give a context-free grammar, see Exercise 17.

2.2 Pushdown automata

As an automaton moves along a word it may return to the same state several times, but it has no way of detecting this. There is no memory in the machine which gives information about its past behaviour. For example, in order to recognize languages like $\{a^n b^n \mid n \in \mathbb{N}\}$ an automaton must be capable of 'remembering' how many as it has seen to ensure that the number of bs will be the same. But a DFA or NFA only has finitely many states, so if a word is long enough then when processing it, some state will be entered twice. The automaton has no way of detecting whether this happens for the first or the second time. One way of making this intuitive idea precise is the Pumping Lemma, and we have seen that this puts a limitation on the family of languages that can be recognized be a finite automaton. In this section we increase the power of such an automaton by adding a gadget called a *stack*. This gives some memory (but still has its limitations).

As before we have an underlying alphabet Σ, and it is our aim to recognize words over that alphabet. However, a stack will be able to hold symbols from an extended alphabet, namely $\Sigma \cup N$, so it will refer to non-terminal symbols, or placeholders, as well. The stack will allow the automaton to keep track of what it has seen so far by pushing these symbols onto the stack. As is usual with stacks, only the symbol added *last* to the stack is accessible to either look at or to remove by popping it from the stack. We assume that popping a symbol from the empty stack results in the empty stack again. We can think of the stack as another word (over the alphabet $\Sigma \cup N$), which should not be confused with the word currently under investigation (over the alphabet Σ). To make the distinction clear, we will usually refer to the stack as 'the stack' (and nothing else). We sometimes think of the 'word currently under investigation' as being written into a file (think of it as the input file to a program), and will refer to it that way, once again to avoid confusion.

We also need a way of recognizing that we have reached the end of that input file, and that the stack is empty. To do this we introduce two new symbols,

$$\mathsf{EOF} \qquad \text{and} \qquad \mathsf{EOS}.$$

The first is short for 'end of file', the last for 'end of stack'.

Hence we think of the input file to an automaton as being of the form

$$x_1 x_2 \cdots x_n \mathsf{EOF}$$

where $x_1, x_2, \ldots, x_n \in \Sigma$ and of the stack as being of the form

$$X_1 X_2 \cdots X_m \mathsf{EOS}$$

where $X_1, X_2, \ldots, X_m \in \Sigma \cup N$. It is an option to write the stack from the top down rather than from left to right, and that would leave no doubt as to where new symbols are added (and old ones removed), but that is inconvenient in a written medium. If Γ is a stack and X a terminal or non-terminal symbol then we can describe the behaviour of the stack operations by

$$\mathsf{push}(X)\Gamma = X\Gamma, \qquad \mathsf{pop}X\Gamma = \Gamma, \qquad \mathsf{pop}\mathsf{EOS} = \mathsf{EOS}.$$

With the deterministic automata we considered so far, there was only one possible next state given the current state and the current input symbol. Now that there is the stack in play as well, a configuration of the automata is described by three ingredients: The current state, the current input symbol and the current symbol on top of the stack.

For finite automata, all that could happen given a state and a character was for the machine to move to another state. But now, the machine can also take actions: it can apply any of the stack actions,

$$\mathsf{push}(X), \qquad \mathsf{pop},$$

and it can decide to advance (by one symbol) (advance) in the input file. Note that the machine can never go *back* and have another look at previous symbols of the input. In this respect it is the same as a DFA or NFA. The two differences are

- the presence of the stack which serves as a memory device;
- the automaton does not have to advance on the input string at each step.

Transitions are now of the form: In state q, if the current input symbol is x (which might be EOF) and there is an X at the top of the stack (which might be EOS), carry out the following actions and go to state q'. In pictures, we will write:

$$q \xrightarrow{(x,X)\mapsto\langle\text{list of actions}\rangle} q'.$$

In other words, the transition function or relation is between elements of

$$\begin{array}{cccccccc} (q, & & x, & & X) & & (\text{list of actions}, & q') \\ Q & \times & (\Sigma \cup \{\mathsf{EOF}\}) & \times & (\Sigma \cup N \cup \{\mathsf{EOS}\}) & \text{and} & A^* & \times \; Q. \end{array}$$

If there are no conditions as to what x is, but X matters, we write $(, X)$, and similarly if X is irrelevant. $(,)$ thus means that the action is carried out no matter what the current input symbol is or what is on top of the stack. Note that the list of action is supposed to be carried out from left to right. Here is the formal definition of such an automaton.

Definition 13 *A* **deterministic pushdown automaton** *or* **DPDA** *over some alphabet $\Sigma \cup N$ is given by the following:*

- *a finite set Q of* states*;*
- *a* start state *$q_\bullet \in Q$;*

- *a set* $F \subseteq Q$ *of* accepting states*;*
- *a* transition function[11] $\delta\colon Q \times (\Sigma \cup \{\mathsf{EOF}\}) \times (\Sigma \cup N \cup \{\mathsf{EOS}\}) \longrightarrow A^* \times Q$ *where*
  - $\mathsf{EOS} \notin \Sigma \cup N$ *('End Of Stack') and* $\mathsf{EOF} \notin \Sigma$ *('End of File') are symbols not occurring in* $\Sigma \cup N$ *and* A *is the set of all actions, where*
  - *the* actions *are* pop*,* $\mathsf{push}(X)$ *(for* $X \in \Sigma \cup N$*) and* $\mathsf{advance}$*.*

A **non-deterministic pushdown automaton***,* **NPDA** *or* **PDA** *differs in this only by the fact that* δ *is a relation rather than a function.*

Such an automaton is deterministic if at each step there is at most one action it can take, otherwise it is non-deterministic. We are using a different convention here in our abbreviations, when compared to finite automata! The reason for making 'non-deterministic pushdown automaton' the default will become obvious. Of course every DPDA can be viewed as an NPDA.

Definition 14 *A string is* **accepted by a pushdown automaton** *if, starting with an empty stack, there is a path through the automaton such that the automaton stops in an accepting state after the entire string has been read. The content of the stack at that time is irrelevant. The language* **recognized by a (D)PDA** *is the set of all words accepted by it.*

Note that there is a considerable difference between DPDAs and DFAs (beyond the use of a dedicated memory device): Whenever we follow a string through a DFA, we know that the process will end since at each step we look at the next symbol of the input string, and the input string is finite. A DPDA, on the other hand, may have transitions which do not further the examination of the input string, and thus a DPDA might go on forever without telling us anything about the input string in question. Figure 21 shows an example of a DPDA which loops forever for all input strings.

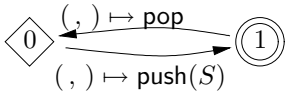

Figure 21: A DPDA that continues forever

Consider the example in Figure 22. The underlying alphabet is $\Sigma = \{a, b\}$ and $N = \{S\}$, and this automaton is non-deterministic: In state 1 there are two actions which can be taken when S is found on the top of the stack: Either that S can just be popped from the stack, or the automaton may carry out (from left to right) the sequence of instructions

$$\mathsf{pop}; \mathsf{push}(b); \mathsf{push}(S); \mathsf{push}(a).$$

In either case the automaton will once again be in state 1.

We can use a table to give the same information. A blank entry in the 'input symbol' or 'top of stack' column means that the action can take place no matter what letter is found there.

[11] Here you can see why we need the new symbols EOF and EOS—it would be difficult otherwise to express what transition to take when the end of the input string has been reached, or when the stack is empty.

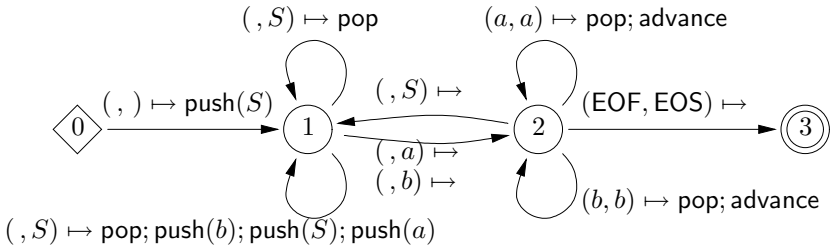

Figure 22: A (non-deterministic) pushdown automaton

configuration				
state	input symbol	top of stack	actions	new state
0			push(S)	1
1		a		2
1		b		2
1		S	pop pop; push(b); push(S); push(a)	1
2		S		1
2	a	a	pop; advance	2
2	b	b	pop; advance	2
2	EOF	EOS		3

In Figures 23 and 24 we show that the word *aaabbb* is accepted by this automaton. What is left of the input string is found in the top left hand corner and the stack can be found in the bottom left hand corner. The current state is black, and the arrow leading to the next state is thicker. It is also the only arrow to have its corresponding action described in full (where there is space), we use abbreviations for the others. Note that we only have to find one path which ends in an accepting state; because of the non-determinism this is not the only path that could be followed for this input string.

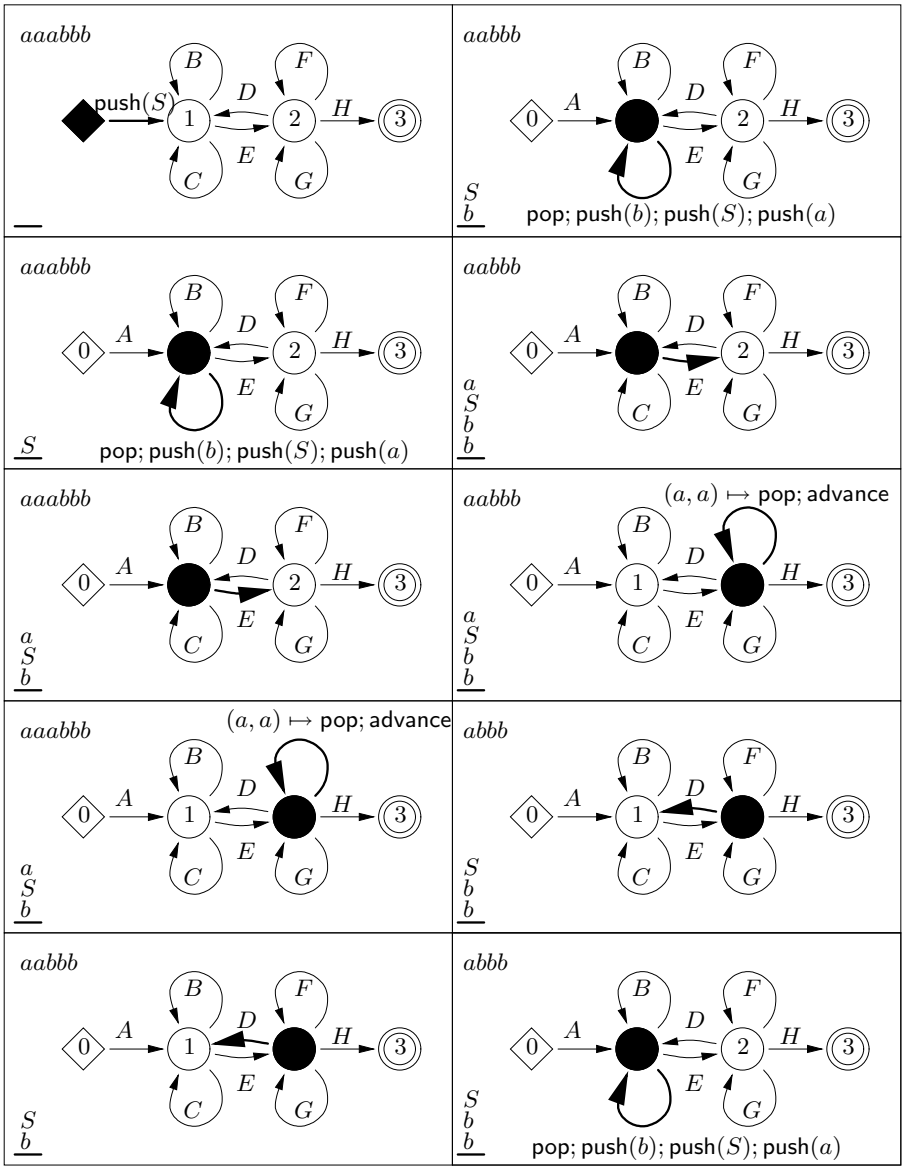

Figure 23: Accepting *aaabbb*
The following abbreviations have been used:

A: push(S)
B: $(, S) \mapsto$ pop,
C: $(, S) \mapsto$ pop; push(b); push(S); push(a)
D: $(, S) \mapsto$,
E: $(, a) \mapsto$, $(, b) \mapsto$,
F: $(a, a) \mapsto$ pop; advance,
G: $(b, b) \mapsto$ pop; advance,
H: (EOF, EOS) $\mapsto$.

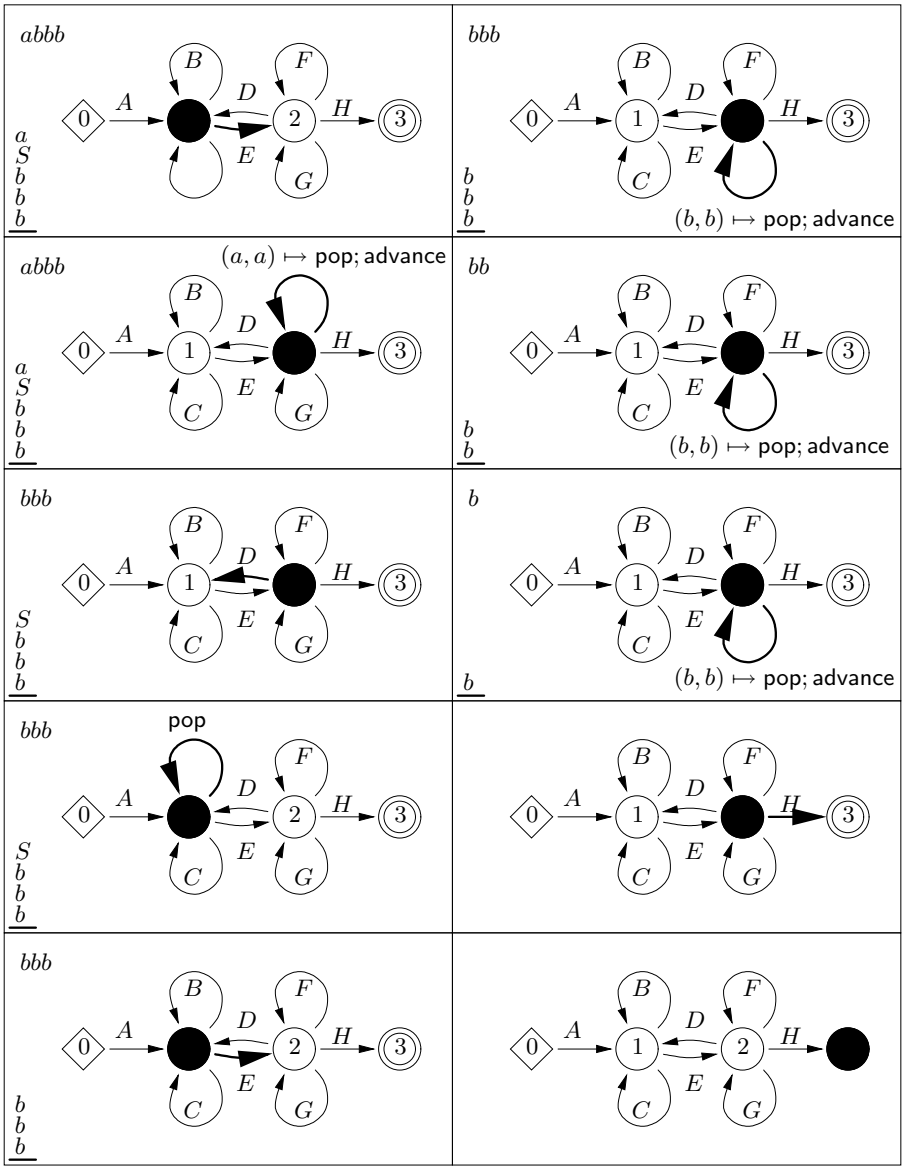

Figure 24: Accepting *aaabbb*, part 2.
The following abbreviations have been used:

A: $\mathsf{push}(S)$
B: $(, S) \mapsto \mathsf{pop}$,
C: $(, S) \mapsto \mathsf{pop}; \mathsf{push}(b); \mathsf{push}(S); \mathsf{push}(a)$
D: $(, S) \mapsto$,
E: $(, a) \mapsto$, $(, b) \mapsto$,
F: $(a, a) \mapsto \mathsf{pop}; \mathsf{advance}$,
G: $(b, b) \mapsto \mathsf{pop}; \mathsf{advance}$,
H: $(\mathsf{EOF}, \mathsf{EOS}) \mapsto$.

An alternative way of presenting the same information would be once again in a table.

old state	remaining input	stack contents	actions	new state
0	$aaabbb$EOF	EOS	push(S)	1
1	$aaabbb$EOF	SEOS	pop; push(b); push(S); push(a)	1
1	$aaabbb$EOF	aSbEOS		2
2	$aaabbb$EOF	aSbEOS	pop; advance	2
2	$aabbb$EOF	SbEOS		1
1	$aabbb$EOF	SbEOS	pop; push(b); push(S); push(a)	1
1	$aabbb$EOF	$aSbb$EOS		2
2	$aabbb$EOF	$aSbb$EOS	pop; advance	2
2	$abbb$EOF	SbbEOS		1
1	$abbb$EOF	SbbEOS	pop; push(b); push(S); push(a)	1
1	$abbb$EOF	$aSbbb$EOS		2
2	$abbb$EOF	$aSbbb$EOS	pop; advance	2
2	bbbEOF	$Sbbb$EOS		1
1	bbbEOF	$Sbbb$EOS	pop	1
1	bbbEOF	bbbEOS		2
2	bbbEOF	bbbEOS	pop; advance	2
2	bbEOF	bbEOS	pop; advance	2
2	bEOF	bEOS	pop; advance	2
2	EOF	EOS		3

We have demonstrated that there is (at least) one path through the automaton which ends in an accepting state when the end of the input is reached. To do this we have shown which state the automaton goes through and which actions it performs, what the stack looks like at a given point, and where we have got in reading the input string.

2.3 From context-free grammars to PDAs

Let G be a context-free grammar. We can quite easily build a pushdown automaton which accepts precisely the strings which are generated by the grammar. The idea is that the processing of a string mimics generating a derivation. As part of a derivation we typically carry out a replacement of a non-terminal by a string of symbols according to some production rule. The stack allows us to remember which terminal symbols we have to read 'further down the string' for some specific production rule to have been applied to generate the given string.

Here is an example. Assume the grammar is that for the language $\{a^n b^n \mid n \in \mathbb{N}\}$ given on page 30. It has the production rules

$$S \to aSb \,|\, \epsilon.$$

Assume the input string is $aabb$EOF. We first find an a. This might have been generated by an application of the rule $S \to aSb$.[12] But for this rule to have taken part in the generation of the given string, it *must* be the case that there is a b at the end of the string to match this a.

[12] In fact, it can only have been generated by this rule because this is a small example with few production rules, but in principle there may be more candidates.

So we have to *remember* somehow that we expect to see that b. We do this by pushing onto the stack the symbols that result from applying this production rule, but from right to left, so that the stack becomes

$$aSb\mathsf{EOS}.$$

We find that the a on the stack matches the a at the start of the input string. So we $\mathsf{advance}$, leaving us with input $abb\mathsf{EOF}$ and we pop a symbol off the stack, giving $Sb\mathsf{EOS}$. Now we have another a at the beginning of (what's left of) the input string, and an S on top of the stack. We decide that the same production rule may have been applied again, so we perform the corresponding actions $\mathsf{pop}; \mathsf{push}(b); \mathsf{push}(S); \mathsf{push}(a)$ to get a stack $aSbb\mathsf{EOS}$. Since the current input symbol matches the top of the stack, we again $\mathsf{advance}$ and pop, giving input $bb\mathsf{EOF}$ and stack $Sbb\mathsf{EOS}$. The production rule $S \to \epsilon$ allows us to pop S off the stack, and then by comparing the remainder of the input and the stack, we reach EOF and an empty stack. This tells us that we have read precisely the symbols that we expected to find, and that the string in question is indeed generated by the grammar.[13]

Using this observation, we try to generalize this by creating a PDA for a given grammar. This automaton has three states: A start state 0, from which the only transition is to state 1 with action $\mathsf{push}(S)$. Clearly, every production will begin with the non-terminal S. This gets us started. We add the following transitions from state 1 to itself:

- If the top symbol on the stack is a terminal symbol and if the current input symbol is that same symbol we pop the symbol off the stack and $\mathsf{advance}$. In other words, for all $x \in \Sigma$, add a transition

  $$(x, x) \mapsto \mathsf{pop}; \mathsf{advance}.$$

  This transition is justified by the idea that the stack tells us which terminal symbols we expect to see in the input string, based on which production rules may have been followed.

- If the top symbol on the stack is a non-terminal symbol R, we do the following for every production rule for R. Add a transition where we pop R off the stack and push the symbols occurring in a production rule for R onto the stack (from right to left). In other words, for each production rule $R \to X_1 \dots X_n$, add a transition

  $$(, R) \mapsto \mathsf{pop}; \mathsf{push}(X_n); \dots; \mathsf{push}(X_1).$$

  Note that X_1 appears *on top of the stack* because it is the first symbol we expect to see in the input string. This is where the automaton becomes non-deterministic, since there may be more than one production rule for R. We can think of the automaton as having to *guess* which production rules it should apply.

Finally we add a third state named 2 which is the only accepting state. There is only one transition involving this state: From 1 we can go to 2 provided that the input string has been consumed and the stack is empty. No actions are being taken with this transition. In other words, there is a transition from 1 to 2

$$(\mathsf{EOF}, \mathsf{EOS}) \mapsto .$$

Figure 25 gives an example turning the grammar for the language $\{a^n b^n \mid n \in N\}$ on page 30 into a PDA.

[13] Note that it is the *last* b that matches the first a, which is as it should be.

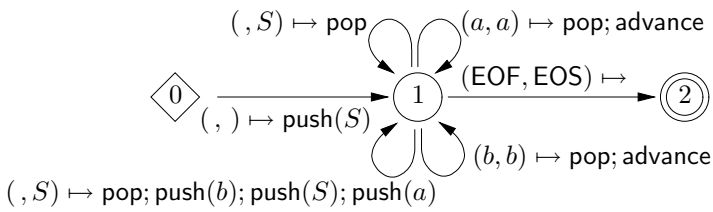

Figure 25: A non-deterministic pushdown automaton

You may find this picture confusing because there are so many arrows from state 1 to itself. The PDA given in Figure 22 accepts almost the same language (it fails to accept the empty word) and is slightly easier to read.[14] In general, it is possible to use similar means to add states to the automaton and make it 'less crowded'.

Note that the automaton in Figure 22 is still non-deterministic in state 1. The reason for this is that there are several production rules for S, and we have allowed the automaton to non-deterministically choose one to apply. That ensures that every word generated by the grammar corresponds to a valid path from the start state to the only accepting state.

On page 36 we showed that the string *aaabbb* is accepted by this automaton's brother—the proof that our new automaton accepts this word as well is much the same. From the table given there (or the stack actions taken along that path, to be precise) we can give a derivation for that string as follows. Every sequence of actions which push something onto the stack results from an application of a production rule: $\mathsf{push}(S)$ gets us started, and $\mathsf{push}(b); \mathsf{push}(S); \mathsf{push}(a)$ is an application of the only non-trivial production rule. The resulting derivation of *aaabbb* is

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb.$$

Given the way the above automaton was constructed it is (at least intuitively) clear that whenever the automaton accepts a string then 'along the way' the stack will have given a derivation for this string. Hence every string accepted by the automaton is generated by the grammar. But the automaton has been built precisely so that all derivations result in a path through the automaton that ends in an accepting state. Hence the language accepted by the PDA is that generated by the grammar.

You have now all you need to tackle Assessed Exercise 2.3.

2.4 Context-free grammars and PDAs

Theorem 2.3 *A language is context-free if and only if there is a non-deterministic pushdown automaton that accepts it.*

Proof sketch. We have argued that if a language is context-free then we can construct a non-deterministic pushdown automaton which accepts it. The converse is even more technical—it amounts to showing that the way the stack is produced can be used to define valid production rules. We will not reproduce it here. It can be found in [Sip97] pp. 110ff., [HMU01] pp. 241ff. or [Koz97], pp. 172ff. □

Exercise 18 asks you to construct a PDA for a given grammar.

[14] Can you see how to make the PDA in Figure 22 accept the empty word as well?

2.5 Deterministic and non-deterministic PDAs

Unlike finite state machines, deterministic and non-deterministic pushdown automata do *not* recognize the same languages. In other words there are context-free grammars for which no deterministic pushdown automaton exists. If you want to see an example for this, look at Exercise 19 (which is not part of the examinable material).

Hence the languages recognized by deterministic PDAs form a subclass of the class of all context-free languages. These languages include all the regular ones—one can turn a deterministic finite automaton into a deterministic PDA which never makes use of its stack. There are ways of saying precisely which context-free languages can be described via deterministic PDAs, but that characterization is not very interesting and shall not concern us here.

2.6 Limitations of context-free languages

As a category, context-free languages are not particularly well-behaved. For example the intersection of two context-free languages need not be context-free, nor does the complement of a context-free language have to be context-free. Union, concatenation and Kleene star, however, when applied to context-free languages, will result in context-free languages. Programming languages typically are not context-free. A very informal argument for that can be made by appealing to the fact that a compiler needs random access to a *symbol table* where it keeps track of names, types, and variables declared so far. This kind of memory cannot be modelled using a stack (in order to get to the name/variable needed at a given time, the stack would have to be emptied until that name or variable is found, but everything that is popped from the stack is lost). We can make a formal argument that certainly not all languages are context-free.

We do this by exhibiting another *Pumping Lemma*. Unsurprisingly this is somewhat more complicated than that for regular languages, but the basic idea is the same: A word of a context-free language that is beyond a certain length contains certain portions that can be repeated arbitrarily many times to give another word of that language.

Lemma 2.4 (The Pumping Lemma II) *If L is a context-free language then there is a number n such that any word α of L that has length at least n can be divided into five pieces $\alpha = \kappa\lambda\mu\nu\xi$ such that*

- *the length of $\lambda\nu$ is greater than 0;*
- *the length of $\lambda\mu\nu$ is at most n;*
- *all words of the form $\kappa\lambda^k\mu\nu^k\xi$, where $k \in \mathbb{N}$, belong to L.*

Proof sketch. We will only give an informal argument. Let G be the grammar generating L. Let α be a sufficiently long word. (What 'sufficiently long' means, that is what n is, is too complicated for an informal proof sketch.) Consider the derivation of α, which gives a (fairly large) parse tree. Now consider a path from the root to a leaf in that tree. If our word is long enough then the longest such path will be very long, long enough for at least one of the non-terminal symbols, say R, to occur more than once. (If the path is longer than the number of non-terminal symbols then one of them *has* to occur more than once.) The situation is pictured in Figure 26.

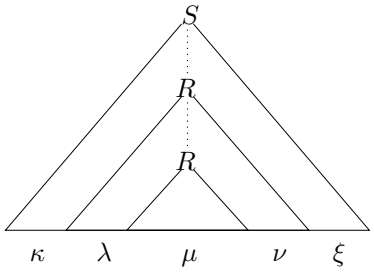

Figure 26: A 'big enough' parse tree

So μ, for example, is the string of terminal symbols that results from rewriting the 'second' R, and $\lambda\mu\nu$ is the string of terminal symbols that results from rewriting the first R. We define the strings κ, λ, μ, ν and ξ as in that figure. We can then take the larger subtree rooted at the first R and use it to replace the smaller subtree rooted at the second R. The resulting parse tree is shown in Figure 27 on the left.

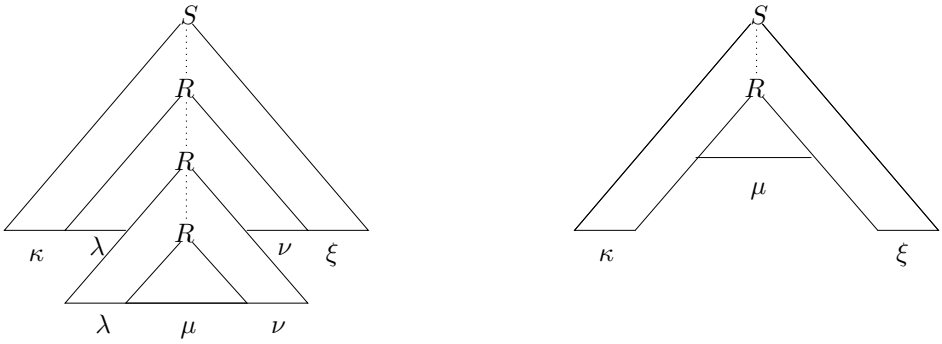

Figure 27: Derived parse trees

The resulting string $\kappa\lambda\lambda\mu\nu\xi$ is therefore also a word of the language, and we can clearly repeat this procedure for the lower two of the three R in that picture, leading to all the strings of the form $\kappa\lambda^i\mu\nu^i\xi$ for which i is greater than 0. Clearly we can also replace the large subtree rooted at the upper R by the smaller one rooted at the lower R, resulting in the tree pictured in Figure 27 on the right which shows a derivation for that string $\kappa\mu\xi = \kappa\lambda^0\mu\nu^0\xi$. □

Note that if L is finite then it may not have any words of length at least n. Hence we can only apply the technique described in the proof of the Pumping Lemma to infinite context-free languages.

The language $L = \{a^i b^i c^i \mid i \in \mathbb{N}\}$ is not context-free. Let n be the length given by the Pumping Lemma and consider the string $a^n b^n c^n$. We show that this language does not satisfy the Pumping Lemma by showing that no matter how we try to divide this string into five substrings, we can never fulfil the conditions. By the first condition, at least one of λ and ν is non-empty. We distinguish two cases.

Case 1 Assume that both λ and ν contain only one kind of character, that is $\lambda = x^i$ for some character $x \in \{a, b, c\}$ and some $i \in \mathbb{N}$, and similarly for ν. But then the string

$\kappa\mu\xi = \kappa\lambda^0\mu\nu^0\xi$ cannot contain an equal number of as, bs and cs because we have changed the number of only two of those from $\kappa\lambda\mu\nu\xi = a^n b^n c^n$ which we know to be in L.

Case 2 Assume that at least one of λ and ν contains more than one kind of character. But then $\kappa\lambda^2\mu\nu^2\xi$ will contain characters in an order not allowed. (If, for example, λ contains both as and bs then λ^2 will contain as following bs which the language does not allow.)

Exercise 20 gives an example for a language which is not context-free. Again, this exercise is not part of the examinable material.

Summary. In this section we have shown that the context-free languages, that is those languages which are generated by a context-free grammar, are precisely those recognized by non-deterministic pushdown automata. (Non-deterministic) PDAs are not equivalent to deterministic PDAs. Finally we have seen that context-free languages also have the property that sufficiently long words have substrings that can be repeated arbitrarily many times. While context-free languages are more powerful than regular languages they still do not suffice to build a compiler. They are useful to treat subproblems (parsing).

2.7 Exercises for Section 2

Exercise 16 Show that the language of all regular expressions is given by a context-free grammar. Give a parse tree for the pattern $a|(b*c)*$.

Exercise 17 (a) Design a context-free grammar which recognizes well-balanced brackets.

(b) Give a context-free grammar which generates the set of *palindromes* (words which read forward the same as backward, such as 'mom') over the alphabet $\{a, b\}$.

(c) Design a context-free grammar for the set of strings over $\Sigma = \{a, b\}$ whose length is odd and whose middle symbol is a.

(d)* *Prove that the following grammar generates precisely those strings which contain twice as many as as bs. Let* $\Sigma = \{a, b\}$, $N = \{S\}$, $S \to SaSaSbS$, $S \to SaSbSaS$, $S \to SbSaSaS$, $S \to \epsilon$.

Exercise 18 (a) Consider the following grammar. Let $\Sigma = \{a, b, c\}$, $N = \{S, T\}$ and

$$S \to aTbS \,|\, \epsilon \qquad \text{and} \qquad T \to c \,|\, S.$$

Show that the string *acbaabb* is generated by this grammar. Can you use this derivation to devise the stack for the corresponding PDA? Draw a PDA for this grammar.

(b) Draw a PDA for the grammar that generates words that contain precisely as many as as bs, see Exercise 17 (d).

Exercise 19* *Show that the language of even-length palindromes which is generated by the context-free grammar* $\Sigma = \{0, 1\}$, $N = \{S\}$, $S \to 0S0 \,|\, 1S1 \,|\, \epsilon$ *is not recognized by a deterministic pushdown automaton.*

Exercise 20* *Use the Pumping Lemma to show that the language* $\{0^i 1^i 0^i 1^i \mid i \in \mathbb{N}\}$ *is not context-free.*

Assessed exercises, due in week 9

Note that some of these exercises require knowledge from Section 3!

Exercise 2.1 *Design a context-free grammar for the set of strings over* $\Sigma = \{a, b\}$ *which contain exactly as many as as bs. Try to argue that every string generated by this grammar satisfies the requirement and that you can generate every such string. Don't worry if you can't do the second argument formally! Give two parse trees for the word abab or argue that in your grammar, only one such parse tree exists.*

Exercise 2.2 Design a grammar for the language given by the pattern $(ab*c*)|(a*b*c)$. (Compare Assessed Exercise 1.3, and make use of the algorithm in Proposition 2.1.)

Exercise 2.3 Give a PDA which recognizes all even-length palindromes over the alphabet $\{a, b, c\}$. (A palindrome is a word that reads forwards as backwards, such as Eve or reviver.)

- Show that the word *acca* is accepted by your automaton by tracking its way along a path ending in an accepting state.
- Show what happens if you try to track the word *aca* along the same path (it should *not* end in an accepting state).

Exercise 2.4 Design a Turing machine which accepts all even-length palindromes over the alphabet $\{a, b\}$. (Hint: Make use of the example on page 46 in Table 3.) Then give the configurations the machine goes through for the input strings *abba* and *aba*.