$LL(k)$ languages

How compiling is related to derivation in a context-free grammar (CFG): reminder. The first thing a compiler is supposed to do is to check whether the given program is syntactically correct. As we have mentioned earlier, programming languages are usually described by a context-free grammar:

- whatever can be obtained by the corresponding rules is a syntactically correct program, and
- whatever cannot be derived by these rules is not a syntactically correct program.

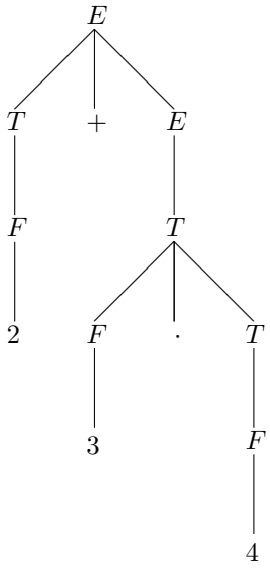
Thus, to check whether a given program is syntactically correct, we need to check whether this program can be generated by the corresponding context-free grammar.

If it turns out that the program is syntactically correct, the compiler will prepare to perform the corresponding instructions – e.g., by forming the adequate quadruples.

How derivation in a CFG corresponds to the sequence of operations. The actual derivation of a program in the corresponding context-free grammar usually directly leads to the appropriate sequence of operations. Let us show this on the example – from the lecture on ambiguous and unambigious grammars – of deriving the expression $2 + 3 \cdot 4$ in the grammar with rules

$$E \to T; \quad E \to T + E; \quad T \to F; \quad F \to (E); \quad T \to F \cdot T; F \to 0; \ldots F \to 9.$$

As we mentioned in that lecture, this derivation takes the form



or, equivalently,

$$\underline{E} \to \underline{T}{+}E \to \underline{F}{+}E \to 2{+}\underline{E} \to 2{+}\underline{T} \to 2{+}\underline{F}{\cdot}T \to 2{+}3{\cdot}\underline{T} \to 2{+}3{\cdot}\underline{F} \to 2{+}3{\cdot}4,$$

where the variable which is replaced in the next step is underlined.

In this derivation, the same variable – e.g., variable E – stands for different expressions:

- from the starting E, we derive the whole expression, while
- from the intermediate variable E, we eventually derive $3 \cdot 4$.

Similarly, we have three different occurrences of the variable F:

- the first occurrence is transformed into 2,
- the second occurrence is transformed into 3, and
- the third occurrence is transformed into 4.

To make the derivation clearer, let us mark different occurrences of the same variables by a number: e.g.,

- the first occurrence of the variable F in this derivation will be denoted by F_1,
- the second occurrence of the variable F in this derivation will be denoted by F_2, etc.

Let us also number of transitions:

- the first transition will be marked as $\to_1$,
- the second transition will be marked as $\to_2$, etc.

As a result, the derivation takes the following form:

$$E_1 \to_1 T_1 + E_2 \to_2 F_1 + E_2 \to_3 2 + E_2 \to_4$$

$$2 + T_2 \to_5 2 + F_2 \cdot T_3 \to_6 2 + 3 \cdot T_3 \to_7 2 + 3 \cdot F_3 \to_8 2 + 3 \cdot 4.$$

Now, we can go backwards, from the very last transition to the very first one, and form the quadruples.

Transforming the last transition into a quadruple. In the above derivation, the last transition is $F_3 \to 4$:

$$E_1 \to_1 T_1 + E_2 \to_2 F_1 + E_2 \to_3 2 + E_2 \to_4$$

$$2 + T_2 \to_5 2 + F_2 \cdot T_2 \to_6 2 + 3 \cdot T_3 \to_7 2 + 3 \cdot \underline{F_3} \to_8 2 + 3 \cdot \underline{4},$$

an application of the general rule $F \to 4$. This transition means to this particular case F_3 of the variable F, we assign the value 4. In Java, this would be described as

```
F3 = 4;
```

and in terms of quadruples, this takes the form

$$= \quad F_3 \quad 4.$$

Transforming the last but one transition into a quadruple. The transition before that is $T_3 \to F_3$:

$$E_1 \to_1 T_1 + E_2 \to_2 F_1 + E_2 \to_3 2 + E_2 \to_4$$

$$2 + T_2 \to_5 2 + F_2 \cdot T_3 \to_6 2 + 3 \cdot \underline{T_3} \to_7 2 + 3 \cdot \underline{F_3} \to_8 2 + 3 \cdot 4,$$

an application of the general rule $T \to F$. This transition means that to the variable T_3, we assign the value F_3. In Java, this would be described as

```
T3 = F3;
```

and in terms of quadruples, this takes the form

$$= \quad T_3 \quad F_3.$$

Transforming the 6th transition into a quadruple. The transition before that is $F_2 \to 3$:

$$E_1 \to_1 T_1 + E_2 \to_2 F_1 + E_2 \to_3 2 + E_2 \to_4$$

$$2 + T_2 \to_5 2 + \underline{F_2} \cdot T_3 \to_6 2 + \underline{3} \cdot T_3 \to_7 2 + 3 \cdot F_3 \to_8 2 + 3 \cdot 4,$$

an application of the general rule $F \to 3$. This transition means that to the variable F_2, we assign the value 3. In Java, this would be described as

```
F2 = 3;
```

and in terms of quadruples, this takes the form

$$= \quad F_2 \quad 3.$$

Transforming the 5th transition into a quadruple. The transition before that is $T_2 \to F_2 \cdot T_3$:

$$E_1 \to_1 T_1 + E_2 \to_2 F_1 + E_2 \to_3 2 + E_2 \to_4$$

$$2 + \underline{T_2} \to_5 2 + \underline{F_2 \cdot T_3} \to_6 2 + 3 \cdot T_3 \to_7 2 + 3 \cdot F_3 \to_8 2 + 3 \cdot 4,$$

an application of the general rule $T \to F \cdot T$. This transition means that to the variable T_2, we assign the value $F_2 \cdot T_3$. In Java, this would be described as

```
T2 = F2 * T3;
```

and in terms of quadruples, this takes the form

$$* \quad F_2 \quad T_3 \quad T_2.$$

Transforming the 4th transition into a quadruple. The transition before that is $E_2 \to T_2$:

$$E_1 \to_1 T_1 + E_2 \to_2 F_1 + E_2 \to_3 2 + \underline{E_2} \to_4$$

$$2 + \underline{T_2} \to_5 2 + F_2 \cdot T_3 \to_6 2 + 3 \cdot T_3 \to_7 2 + 3 \cdot F_3 \to_8 2 + 3 \cdot 4,$$

an application of the general rule $E \to T$. This transition means that to the variable E_2, we assign the value T_2. In Java, this would be described as

```
E2 = T2;
```

and in terms of quadruples, this takes the form

$$= \quad E_2 \quad T_2.$$

Transforming the 3rd transition into a quadruple. The transition before that is $F_1 \to 2$:

$$E_1 \to_1 T_1 + E_2 \to_2 \underline{F_1} + E_2 \to_3 \underline{2} + E_2 \to_4$$

$$2 + T_2 \to_5 2 + F_2 \cdot T_3 \to_6 2 + 3 \cdot T_3 \to_7 2 + 3 \cdot F_3 \to_8 2 + 3 \cdot 4,$$

an application of the general rule $F \to 2$. This transition means that to the variable F_1, we assign the value 2. In Java, this would be described as

```
F1 = 2;
```

and in terms of quadruples, this takes the form

$$= \quad F_1 \quad 2.$$

Transforming the 2nd transition into a quadruple. The transition before that is $T_1 \to F_1$:

$$E_1 \to_1 \underline{T_1} + E_2 \to_2 \underline{F_1} + E_2 \to_3 2 + E_2 \to_4$$

$$2 + T_2 \to_5 2 + F_2 \cdot T_3 \to_6 2 + 3 \cdot T_3 \to_7 2 + 3 \cdot F_3 \to_8 2 + 3 \cdot 4,$$

an application of the general rule $T \to F$. This transition means that to the variable T_1, we assign the value F_1. In Java, this would be described as

```
T1 = F1;
```

and in terms of quadruples, this takes the form

$$= \quad T_1 \quad F_1.$$

Transforming the first transition into a quadruple. Finally, the very first transition is $E_1 \to T_1 + E_2$:

$$\underline{E_1} \to_1 \underline{T_1 + E_2} \to_2 F_1 + E_2 \to_3 2 + E_2 \to_4$$

$$2 + T_2 \to_5 2 + F_2 \cdot T_3 \to_5 2 + 3 \cdot T_3 \to_7 2 + 3 \cdot F_3 \to_8 2 + 3 \cdot 4,$$

an application of the general rule $E \to T + E$. This transition means that to the variable E_1, we assign the value $T_1 + E_2$. In Java, this would be described as

```
E1 = T1 + E2;
```

and in terms of quadruples, this takes the form

$$+ \quad T_1 \quad E_2 \quad E_1.$$

Overall, we get the following sequence of quadruples for computing the expression $2 + 3 \cdot 4$:

$$\begin{aligned} = \quad & F_3 \quad 4 \\ = \quad & T_3 \quad F_3 \\ = \quad & F_2 \quad 3 \\ * \quad & F_2 \quad T_3 \quad T_2 \end{aligned}$$

$$
\begin{aligned}
= \;\; & E_2 \;\; T_2 \\
= \;\; & F_1 \;\; 2 \\
= \;\; & T_1 \;\; F_1 \\
+ \;\; & T_1 \;\; E_2 \;\; E_1.
\end{aligned}
$$

So, if we know how a program was derived in the corresponding grammar, we can indeed compile it – i.e., represent it as a sequence of elementary computer operations.

Comment. Of course, as usual, the resulting code can be optimized. For example, in the first quadruple, we assign to F_3 the value 4, but then the only thing we do with this variable F_3 is assign its value to the variable T_3. We could as well directly assign the value 4 to the variable T_3, i.e., replace the first two quadruples with an equivalent single one

$$= \;\; T_3 \;\; 4.$$

The only thing we do with the variable T_3 is use it for multiplication. Since we know that the value of T_3 is 4, why not directly use 4 in this multiplication, i.e., replace this quadruple and the multiplication quadruple with a single one

$$* \;\; F_2 \;\; 4 \;\; T_2.$$

We can continue simplifying the code. As a result, we will have only two quadruples left:

$$
\begin{aligned}
* \;\; & 3 \;\; 4 \;\; E_2 \\
+ \;\; & 2 \;\; E_2 \;\; E_1.
\end{aligned}
$$

How we described compiling so far. So far, we have described the priority-based approach to compiling. In some simple programming langauges and their fragments, we can apply a different algorithm for compiling, known as $LL(k)$.

How some languages are actually compiled. To compile, the computer reads the program symbol-by-symbol. Eventually, after we read a sufficient number of symbols, we understand which rules needs to be applied.

For example, if a line starts with letter i:

- this could means if, the beginning of a conditional statement;
- this could mean int, the definition of an integer variable;
- this could mean the letter in the long name of a variable, e.g., the beginning of an assignment statement,

and in all these cases, the meaning of this line is different. It becomes clear when we reach space or assignment sign or opening parenthesis, and this can take a lot of symbols.

In some languages, however, there is a limit on variable length, a limit on many other constructions. In many such languages, there is an integer k so that when we read k symbols, we know what rule to apply. Such languages are known as $LL(k)$ languages, where L stands for Left – since we read the program from left to right.

How do we compile an $LL(k)$ language. For each such language, for each sequence of k symbols, we know which rule of the grammar to apply. In the computer, this information is stored in a table.

When we see a symbol, we read $k-1$ next symbols, and then look into the table to see which rule should be applied for the resulting sequence of k symbols.

An example of an $LL(1)$ language. Let us give a simple example of such a language in which $k = 1$, i.e., in which just reading one symbol enables us to select the corresponding rule.

This language has the starting variable S, an additional variable F, four terminal symbols (,), $+$, a, and the following three rules:

1. $S \to F$
2. $S \to (S + F)$
3. $F \to a$

What words can be derived in this language? We start with the starting variable S. To this variable, we can apply either Rule 1 or Rule 2.

If we apply Rule 1, then we get the word F. We want to have a word that only has terminal symbols, so we must eliminate the variable F. To eliminate F, there is only one rule: Rule 3. If we apply this rule, we get the word

$$a.$$

If we apply the second rule to the starting variable S then we get the word $(S + F)$. We want to have a word that only has terminal symbols, so we must eliminate both variables S and F. To eliminate F, there is only one rule: Rule 3. If we apply this rule, we get the word $(S + a)$. To eliminate S, we can use either Rule 1 or Rule 2.

- If we apply Rule 1, we get the word

$$(a + a).$$

- If we apply Rule 2, we get a word $((S + F) + a)$. By applying Rule 3, we get $((S + a) + a)$.

What can we do with the word $((S + a) + a)$?

- If we apply Rule 1 to eliminate S, we get the word

$$((a + a) + a).$$

- If we apply Rule 2, we get the word $(((S + F) + a) + a)$, etc.

So, the language of all the words derived in this grammar has the following form:

$$\{a, \quad (a + a), \quad ((a + a) + a), \quad (((a + a) + a) + a), \ldots\}.$$

Let us design a table for this language. Let us brainstorm. As you probably remember, when we transformed finite automata into a context-free grammar, variables of a grammar corresponded to states of the corresponding automaton.

In the beginning, we are in the start state corresponding to the starting variable S.

- In the start state, if the first symbol you see is a, then as shown by the above analysis, the first rule to apply is the first rule $S \to F$: indeed, if we apply Rule 2, we will end with a word that starts with the opening parenthesis (. We will denote this by placing 1 in the table's cell corresponding to the state S and the symbol a.
- On the other hand, if you see the opening parenthesis (, this means that we need to apply Rule 2. We will thus place 2 in the corresponding cell.
- If we see any other symbol, this means that the word does not belong to this language. We will denote this by $-$.

If we are in the state F – i.e., if we are ready to apply a rule starting with F – then the only rule that we can apply is Rule 3, and the only symbol that we get this way is the symbol a. Thus, if in this state, we see symbol a, we use Rule 3, if we see any other symbol, we cannot apply any rule.

Resulting table. Here is the resulting table:

	(	)	a	$+$
S	2	$-$	1	$-$
F	$-$	$-$	3	$-$

An example of using the table for compiling: general idea. Let us show how, by using this table, we can find the derivation of the word

$$(a + a)$$

In the beginning, we are in the start state S. So, what we need to do is to find a sequence of rule applications that lead from the starting variable S to the word $(a + a)$.

In other words, we start with two words:

$$(a + a) \text{ and } S$$

and we need to find the sequence of rules applied to the second word after which the two words will match – – i.e., become identical.

Trying to match. The first symbol of the second word is S, the first symbol of the first word is (. According to the table, we have to apply Rule 2, i.e., we perform the transition

$$S \to (S + F)$$

According to Rule 2, the variable S get transformed into the expression

$$(S + F)$$

So now, we need to match our word $(a + a)$ with this expression:

$$(a + a) \text{ and } (S + F)$$

Matching the first symbols. The first symbols of the words $(a + a)$ and $(S + F)$ match, so what we need is to match the remaining parts of the words, i.e., the words

$$a + a) \text{ and } S + F)$$

Trying to match (cont-d). The first symbol of the second word that we need to match is S, and the first symbol in the first word is a. So, according to the table, we need to use Rule 1 ($S \to F$), i.e., to perform a transition

$$S \to (\underline{S} + F) \to (F + F)$$

When we apply Rule 1 to the word $S + F)$, we get $F + F)$. Thus, now, we need to match the words:

$$a + a) \text{ and } F + F)$$

Matching the next symbols. The first symbol of the second word that we need to match is the symbol F (the first of the two symbols F), and the first symbol of the first word is a. Thus, according to the table, we need to use Rule 3, i.e., replace the first occurrence of F with the letter a:

$$S \to (\underline{S} + F) \to (\underline{F} + F) \to (a + F)$$

When we apply Rule 2 to the first symbol F of the word $F + F)$, this word turns into $a + F)$. So now, we need to match the words

$$a + a) \text{ and } a + F)$$

We continue matching. The first two symbols a and $+$ match, so what is left is to match the remaining parts of these two words:

$$a) \text{ and } F)$$

Matching the next symbols. The first symbol of the second word that we need to match is F, and the first symbol of the first word is a. So, according to the table, we need to use Rule 3: replace F with a. In this case, the previous derivation extends to

$$S \to (\underline{S} + F) \to (\underline{F} + F) \to (a + \underline{F}) \to (a + a)$$

When we apply this rule $F \to a$ to the word $F)$, we get $a)$. So now, we need to match the words

$$a) \text{ and } a)$$

These words are identical, so we are done.

Resulting derivation. The above derivation – sequence of rules that we found by looking at the table – provides the desired derivation of the word $(a + a)$:

$$S \to (S + F) \to (F + F) \to (a + F) \to (a + a).$$

Practice. Use this algorithm to derive other words from the above language.

An example where a word does not belong to the language. Let us show how the use of table will help us find out when a word *does not* belong to the language. Namely, let us show that the word $(a + a$ does not belong to our language.

In the beginning, we are in the start state S. So, what we need to do is to find a sequence of rule applications that lead from the starting variable S to the word $(a + a$.

In other words, we start with two words:

$$(a + a \text{ and } S$$

and we need to find the sequence of rules applied to the second word after which the two words will match – – i.e., become identical.

The first symbol of the second word is S, the first symbol of the first word is (. According to the table, we have to apply Rule 2, i.e., we perform the transition

$$S \to (S + F)$$

According to Rule 2, the variable S get transformed into the expression

$$(S + F)$$

So now, we need to match our word $(a + a)$ with this expression:

$$(a + a \text{ and } (S + F)$$

The first symbols of the words $(a + a$ and $(S + F)$ match, so what we need is to match the remaining parts of the words, i.e., the words

$$a + a \text{ and } S + F)$$

The first symbol of the second word that we need to match is S, and the first symbol in the first word is a. So, according to the table, we need to use Rule 1 ($S \to F$), i.e., to perform a transition

$$S \to (\underline{S} + F) \to (F + F)$$

When we apply Rule 1 to the word $S + F)$, we get $F + F)$. Thus, now, we need to match the words:

$$a + a \text{ and } F + F)$$

The first symbol of the second word that we need to match is the symbol F (the first of the two symbols F), and the first symbol of the first word is a. Thus, according to the table, we need to use Rule 3, i.e., replace the first occurrence of F with the letter a:

$$S \to (\underline{S} + F) \to (\underline{F} + F) \to (a + F)$$

When we apply Rule 2 to the first symbol F of the word $F + F)$, this word turns into $a + F)$. So now, we need to match the words

$$a + a \text{ and } a + F)$$

The first two symbols a and $+$ match, so what is left is to match the remaining parts of these two words:

$$a \text{ and } F)$$

The first symbol of the second word that we need to match is F, and the first symbol of the first word is a. So, according to the table, we need to use Rule 3: replace F with a. In this case, the previous derivation extends to

$$S \to (\underline{S} + F) \to (\underline{F} + F) \to (a + \underline{F}) \to (a + a)$$

When we apply this rule $F \to a$ to the word $F)$, we get $a)$. So now, we need to match the words

$$a \text{ and } a)$$

These words are different, and there are no variables left that we can replace and make the words match. So, we conclude that the word $(a + a$ does not belong to our language.

Another example of an $LL(1)$ language. Another example of an $LL(1)$ language is a language with which we started the study of context-free grammars: the language

$$\{a^n b^n : n = 0, 1, 2, \ldots\} = \{\Lambda, ab, aabb, aaabbb, \ldots\}$$

which is described by a grammar with one variable S, two terminal symbols a and b, and two rules:

1. $S \to \varepsilon$
2. $S \to aSb$

Let us design a table for this language. Let us brainstorm. As you remember, when we transformed finite automata into a context-free grammar, variables of a grammar corresponded to states of the corresponding automaton. In this case, we have only one variable S, so we have only one state S.

The desired language consists of an empty word and words of the type ab, $aabb$, ... The empty string Λ is obtained by using Rule 1:

$$\underline{S} \to \varepsilon.$$

All other words are obtained if we first apply Rule 2 several times and then, at the end, Rule 1. For example, the word ab can be obtained if we apply Rule 2 once:

$$\underline{S} \to a\underline{S}b \to ab;$$

The word $aabb$ is obtained if we apply the Rule 2 twice:

$$\underline{S} \to a\underline{S}b \to aa\underline{S}bb \to aabb.$$

So:

- If in the start state, we see the end-of-line symbol eol, this means that we have an empty string Λ, so we need to apply Rule 1.
- If we see symbol a, this means that we need to apply Rule 2.
- If we see symbol b, this means that we have went through all a's, so now we need to apply Rule 1.

Here is the resulting table:

	a	b	eol
S	2	1	1

Comment. In the previous example, we did not need to invoke end-of-line symbol: this is only needed when we can march a variable with a remaining

empty part of the given string, i.e., in effect, when have a rule of the type $A \to \varepsilon$ for some variable A.

Examples. Let us show how, by using this table, we can find the derivation of the words Λ, ab, and $aabb$ – and thus show that these words belong to the language, and let us also show that, e.g., the word abb does not belong to this language.

Empty string. We are in the state S, so we need to match the empty string Λ with the expression S:

$$\Lambda \text{ and } S.$$

The string is empty, so the first symbol we see is the end-of-line symbol eol. According to the table, we have to use Rule 1, so the transition takes the form

$$S \to \varepsilon.$$

Replacing S with ε, the matching turns into

$$\Lambda \text{ and } \varepsilon.$$

On both sides, we get empty strings – a perfect match, so we are done. We conclude that the empty string belongs to the language – and $S \to \varepsilon$ is the derivation of this string in our grammar.

String ab. We are in the state S, so we need to match the string ab with the expression S:

$$ab \text{ and } S.$$

The first symbol we see is the symbol a. According to the table, we have to use Rule 2, so the transition takes the form

$$\underline{S} \to aSb.$$

Replacing S with aSb, the matching turns into

$$ab \text{ and } aSb.$$

The first symbols in these two expressions match, so the remaining expression-to-match are as follows:

$$b \text{ and } Sb.$$

Now, we need to match S with the word whose first symbol is b, so, according to the table, we use Rule 1. The derivation thus continues:

$$\underline{S} \to a\underline{S}b \to ab.$$

Replacing S with an empty string in the expression Sb that we try to match, we conclude that we need to match

$$b \text{ and } b.$$

These two expressions match, so we are done. We conclude that the string ab belongs to the language – and $\underline{S} \to a\underline{S}b \to ab$ is the derivation of this string in our grammar.

String $aabb$**.** We are in the state S, so we need to match the string $aabb$ with the expression S:

$$aabb \text{ and } S.$$

The first symbol we see is the symbol a. According to the table, we have to use Rule 2, so the transition takes the form

$$\underline{S} \to aSb.$$

Replacing S with aSb, the matching turns into

$$aabb \text{ and } aSb.$$

The first symbols in these two expressions match, so the remaining expression-to-match are as follows:

$$abb \text{ and } Sb.$$

Now, we need to match S with the word whose first symbol is a, so, according to the table, we use Rule 2. The derivation thus continues:

$$\underline{S} \to a\underline{S}b \to aaSbb.$$

Replacing S with an expression aSb in the expression Sb that we try to match, we conclude that we need to match

$$abb \text{ and } aSbb.$$

The first symbols in these two expressions match, so the remaining expression-to-match are as follows:

$$bb \text{ and } Sbb.$$

Now, we need to match S with the word whose first symbol is b, so, according to the table, we use Rule 1. The derivation thus continues:

$$\underline{S} \to a\underline{S}b \to aa\underline{S}bb \to aabb.$$

Replacing S with an empty string in the expression Sbb that we try to match, we conclude that we need to match

$$bb \text{ and } bb.$$

These two expressions match, so we are done. We conclude that the string $aabb$ belongs to the language – and $\underline{S} \to a\underline{S}b \to aa\underline{S}bb \to aabb$ is the derivation of this string in our grammar.

String abb**.** We are in the state S, so we need to match the string abb with the expression S:

$$abb \text{ and } S.$$

The first symbol we see is the symbol a. According to the table, we have to use Rule 2, so the transition takes the form

$$\underline{S} \to aSb.$$

Replacing S with aSb, the matching turns into

$$abb \text{ and } aSb.$$

The first symbols in these two expressions match, so the remaining expression-to-match are as follows:

$$bb \text{ and } Sb.$$

Now, we need to match S with the word whose first symbol is b, so, according to the table, we use Rule 1. The derivation thus continues:

$$\underline{S} \to a\underline{S}b \to ab.$$

Replacing S with an empty string in the expression Sb that we try to match, we conclude that we need to match

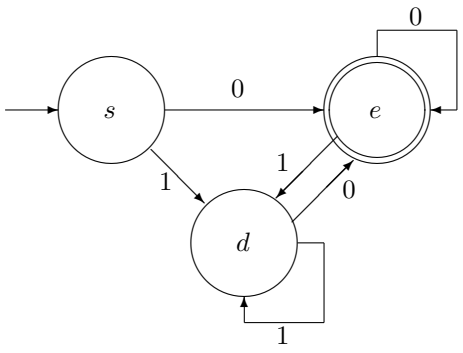
$$bb \text{ and } b.$$

These two expressions do not match, and there are no more variables left – so we cannot replace any variables and make the two expressions match. We therefore conclude that the string abb does not belong to the language.

Other examples of $LL(1)$ languages. One can easily see that any language coming from the standard translation of a deterministic finite automaton into CFG is also $LL(1)$.

Indeed, we start in a starting state s. Then, in any state q, if we see a symbol a, then we use the rule corresponding to automat rule that explains what to do when in this state, we see this symbol. If we are done reading the symbols – i.e., if we see end-of-line – then, if we are in the final state, we use the rule that the corresponding variable get transformed into the empty string.

Let us illustrate it on the example of a finite automaton for recognizing even unsigned integers, an automaton on which we illustrated the general algorithm for transforming finite automata into context-free grammars:



As we mentioned in that lecture, y applying the general algorithm to this finite automaton, we get a CFG with the starting variable S and the following rules:

1. $S \to 0E$
2. $S \to 1D$
3. $E \to 0E$
4. $E \to 1D$
5. $D \to 0E$
6. $D \to 1D$
7. $E \to \varepsilon$

If we are in the state S and we see 0, we use Rule 1 – that corresponds to describing what happens if we are in state s and see 0. If we are in the state S and we see 1, we use Rule 2 – that corresponds to describing what happens if we are in state s and see 1, etc.

Finally, if we are in state E and see end-of-line, we use the last Rule 7. As a result, we get the following table

	0	1	eol
S	1	2	–
E	3	4	7
D	5	6	–

An example of an $LL(2)$-language. Let us recall how binary integers (I) can be defined:

- an integer is either an unsigned integer (U), or a sign ($+$ or $-$) followed by an unsigned integer;
- an unsigned integer is either a digit (D) or a digit followed by an unsigned integer;
- finally, a digit is either 0 or 1.

This leads to the following rules:

1. $I \to +U$;
2. $I \to -U$;
3. $I \to U$;
4. $U \to DU$;
5. $U \to D$;

6. $D \to 0$;
7. $D \to 1$.

In the beginning, we are in state I.

- If in this state, we see $+$ or $-$, we need to use rules 1 or 2, since otherwise, we will not get the sign in the word.
- If in the state I, we see 0 or 1, then we need to use Rule 3.

In both cases, we go to state U. (If in the state I, we see end-of-line, this means that we gave an empty string – not an integer, so no rule can help.)

If we are in state U, and we see 0, this means that:

- either the remaining part of the word is 0, in which case we need to use Rule 5,
- or the remaining part of the word starts with 0 and has other digits, in which case we need to use Rule 4.

So:

- if the next two symbols are 0eol or 1eol, we use Rule 5,
- if the next two symbols are 00, 01, 10, or 11, we use Rule 4.

Thus, it is enough to see the next 2 symbols, and we will know which rule to apply. Here is the corresponding table:

	$+$	$-$	00	01	0eol	10	11	1eol
I	1	2	3	3	3	3	3	3
U	–	–	4	4	5	4	4	5
D	–	–	–	–	6	–	–	7