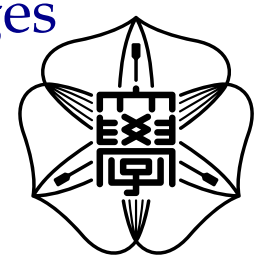
Theory of Computation

Thomas Zeugmann

Hokkaido University
Laboratory for Algorithmics

http://www-alg.ist.hokudai.ac.jp/~thomas/ToC/

Lecture 7: Further Properties of Context-Free Languages



©Thomas Zeugmann

Backus-Naur Form I

As mentioned in the last lecture, context-free grammars are of fundamental importance for programming languages.
However, in the specification of programming languages usually a form different to the one provided in our Definition of *context-free grammars* is used.

Backus-Naur Form I

As mentioned in the last lecture, context-free grammars are of fundamental importance for programming languages.
However, in the specification of programming languages usually a form different to the one provided in our Definition of *context-free grammars* is used.
This form is the so-called ***Backus normal form*** or ***Backus-Naur form***. It was created by John Backus to specify the grammar of ALGOL. Later it has been simplified by Peter Naur to reduce the character set used and Donald Knuth proposed to call the new form Backus-Naur form. Fortunately, whether or not one is following Knuth's suggestion, the form is commonly abbreviated BNF.

Backus-Naur Form II

The form uses four meta characters that are not allowed to appear in the working vocabulary, i.e., in $T \cup N$ in our definition. These meta characters are

$$\langle \quad \rangle \quad ::= \quad |$$

and the idea to use them is as follows. Strings (*not* containing the meta characters) are enclosed by $\langle$ and $\rangle$ denote nonterminals. The symbol $::=$ serves as a replacement operator (in the same way as $\rightarrow$) and $|$ is read as "or."

Backus-Naur Form III

Example 1

Consider a context-free grammar for unsigned integers in a programming language. Here, D stands for the class of digits and U for the class of unsigned integers.

$$
\begin{array}{lcl}
D & \to & 0 \\
D & \to & 1 \\
D & \to & 2 \\
D & \to & 3 \\
D & \to & 4 \\
U & \to & D
\end{array}
\qquad
\begin{array}{lcl}
D & \to & 5 \\
D & \to & 6 \\
D & \to & 7 \\
D & \to & 8 \\
D & \to & 9 \\
U & \to & UD\,.
\end{array}
$$

©Thomas Zeugmann

Backus-Naur Form III

Example 1

Consider a context-free grammar for unsigned integers in a programming language. Here, D stands for the class of digits and U for the class of unsigned integers.

$$
\begin{array}{lcl@{\qquad}lcl}
D & \to & 0 & D & \to & 5 \\
D & \to & 1 & D & \to & 6 \\
D & \to & 2 & D & \to & 7 \\
D & \to & 3 & D & \to & 8 \\
D & \to & 4 & D & \to & 9 \\
U & \to & D & U & \to & UD\,.
\end{array}
$$

Rewriting this example in BNF yields:

$$
\begin{aligned}
\langle \text{digit} \rangle &::= 0|1|2|3|4|5|6|7|8|9 \\
\langle \text{unsigned integer} \rangle &::= \langle \text{digit} \rangle | \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle
\end{aligned}
$$

©Thomas Zeugmann

Backus-Naur Form IV

As this example clearly shows the BNF allows for a very compact representation of the grammar. This is of particular importance when defining the syntax of a programming language, where the set of productions usually contains many elements.

Whenever appropriate, we shall adopt a blend of the notation used in BNFs, i.e., occasionally we shall use | as well as $\langle$ and $\rangle$ but not ::= .

Motivation

Context-free languages play an important role in many applications. As far as regular languages are concerned, we have seen that finite automata are very efficient recognizers.
So, what about context-free languages?

©Thomas Zeugmann

Motivation

Context-free languages play an important role in many applications. As far as regular languages are concerned, we have seen that finite automata are very efficient recognizers.
So, what about context-free languages?
Again, for every context-free language a recognizer can be algorithmically constructed. Formally, these recognizers are *pushdown automata*. There are many software systems around that perform the construction of the relevant pushdown automaton for a given language. These systems are important in that they allow the quick construction of the syntax analysis part of a compiler for a new language and are therefore highly valued.

©Thomas Zeugmann

Motivation

Context-free languages play an important role in many applications. As far as regular languages are concerned, we have seen that finite automata are very efficient recognizers.

So, what about context-free languages?

Again, for every context-free language a recognizer can be algorithmically constructed. Formally, these recognizers are *pushdown automata*. There are many software systems around that perform the construction of the relevant pushdown automaton for a given language. These systems are important in that they allow the quick construction of the syntax analysis part of a compiler for a new language and are therefore highly valued.

We shall study them in Lectures 9 and 10.

Motivation

It is advantageous to treat another important tool for syntax analysis first, i.e., *parsers*.

©Thomas Zeugmann

Motivation

It is advantageous to treat another important tool for syntax analysis first, i.e., *parsers*.
One of the most widely used of these syntax analyzer generators is called `yacc` (yet another compiler-compiler). The generation of a parser, i.e., a function that creates parse trees from source programs has been institutionalized in the `yacc` command that appears in all UNIX systems. The input to `yacc` is a CFG, in a notation that differs only in details from the well-known BNF. Associated with each production is an *action*, which is a fragment of `C` code that is performed whenever a node of the parse tree that (combined with its children) corresponds to this production is created. So, let us continue with parse trees.

©Thomas Zeugmann

Parse Trees I

A nice feature of grammars is that they describe the hierarchical syntactic structure of the sentences of languages they define. These hierarchical structures are described by *parse trees*.

©Thomas Zeugmann

Parse Trees I

A nice feature of grammars is that they describe the hierarchical syntactic structure of the sentences of languages they define. These hierarchical structures are described by *parse trees*.

Parse trees are a representation for derivations. When used in a compiler, it is the data structure of choice to represent the source program. In a compiler, the tree structure of the source program facilitates the translation of the source program into executable code by allowing natural, recursive functions to perform this translation process.

©Thomas Zeugmann

Parse Trees II

Parse trees are trees defined as follows.

Definition 1

Let $\mathcal{G} = [T, N, \sigma, P]$ be a context-free grammar. A ***parse tree*** for $\mathcal{G}$ is a tree satisfying the following conditions.

(1) Each interior node and the root are labeled by a variable from N.

(2) Each leaf is either labeled by a non-terminal, a terminal or the empty string. If the leaf is labeled by the empty string, then it must be the only child of its parent.

(3) If an interior node is labeled by h and its children are labeled by $x_1, \ldots, x_k$, respectively, from left to right, then $h \to x_1 \cdots x_k$ is a production from P.

©Thomas Zeugmann

Parse Trees III

Thus, every subtree of a parse tree describes one instance of an abstraction in the statement. Next, we define the yield of a parse tree. If we look at the leaves of any parse tree and concatenate them from left to right, we get a string. This string is called the *yield* of the parse tree.

©Thomas Zeugmann

Parse Trees III

Thus, every subtree of a parse tree describes one instance of an abstraction in the statement. Next, we define the yield of a parse tree. If we look at the leaves of any parse tree and concatenate them from left to right, we get a string. This string is called the *yield* of the parse tree.

Clearly, of special importance is the case that the root is labeled by the start symbol and that the yield is a ***terminal string***, i.e., all leaves are labeled with a symbol from T or the empty string. Thus, the language of a grammar can also be expressed as the set of yields of those parse trees having the start symbol at the root and a terminal string as yield.

©Thomas Zeugmann

Parse Trees IV

Let us assume that we have the following part of a grammar on hand that describes how assignment statements are generated.

Example 2

$$
\begin{aligned}
\langle \texttt{assign} \rangle &\rightarrow \langle \texttt{id} \rangle := \langle \texttt{expr} \rangle \\
\langle \texttt{id} \rangle &\rightarrow \texttt{A} \mid \texttt{B} \mid \texttt{C} \\
\langle \texttt{expr} \rangle &\rightarrow \langle \texttt{id} \rangle + \langle \texttt{expr} \rangle \\
&\mid \langle \texttt{id} \rangle * \langle \texttt{expr} \rangle \\
&\mid (\langle \texttt{expr} \rangle) \\
&\mid \langle \texttt{id} \rangle
\end{aligned}
$$

©Thomas Zeugmann

Parse Trees V

Now, let us look at the assignment statement $\mathtt{A} := \mathtt{B} * (\mathtt{A} + \mathtt{C})$ which can be generated by the following derivation.

$$
\begin{aligned}
\langle \mathtt{assign} \rangle &\Rightarrow \langle \mathtt{id} \rangle := \langle \mathtt{expr} \rangle \\
&\Rightarrow \mathtt{A} := \langle \mathtt{expr} \rangle \\
&\Rightarrow \mathtt{A} := \langle \mathtt{id} \rangle * \langle \mathtt{expr} \rangle \\
&\Rightarrow \mathtt{A} := \mathtt{B} * \langle \mathtt{expr} \rangle \\
&\Rightarrow \mathtt{A} := \mathtt{B} * (\langle \mathtt{expr} \rangle) \\
&\Rightarrow \mathtt{A} := \mathtt{B} * (\langle \mathtt{id} \rangle + \langle \mathtt{expr} \rangle) \\
&\Rightarrow \mathtt{A} := \mathtt{B} * (\mathtt{A} + \langle \mathtt{expr} \rangle) \\
&\Rightarrow \mathtt{A} := \mathtt{B} * (\mathtt{A} + \langle \mathtt{id} \rangle) \\
&\Rightarrow \mathtt{A} := \mathtt{B} * (\mathtt{A} + \mathtt{C})
\end{aligned}
$$

Parse Trees V

The structure of the assignment statement that we have just derived is shown in the parse tree displayed below.

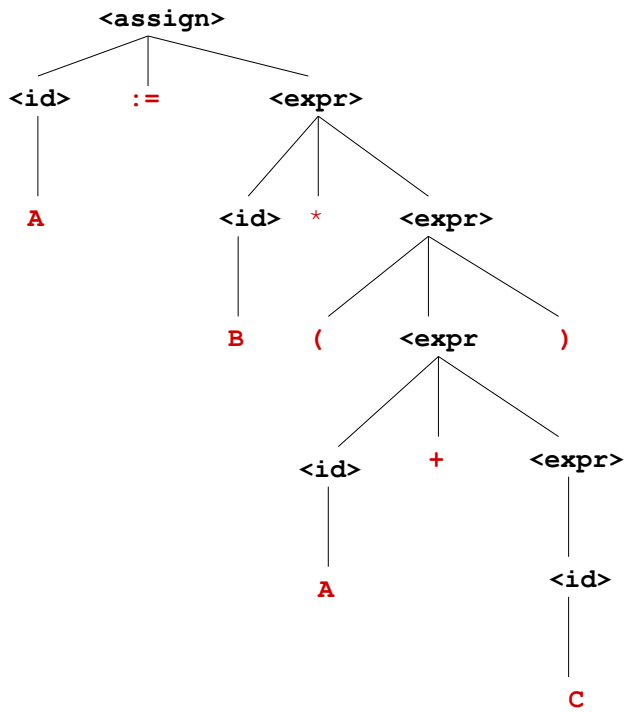


Figure 1: Parse tree for sentence $\mathtt{A} := \mathtt{B} * (\mathtt{A} + \mathtt{C})$

©Thomas Zeugmann

Parse Trees VI

Note that syntax analyzers for programming languages, which are often called *parsers*, construct parse trees for given programs. Some systems construct parse trees only implicitly, but they also use the whole information provided by the parse tree during the parse. There are two major approaches of how to build these parse trees. One is *top-down* and the other one is *bottom-up*. In the top-down approach the parse tree is built from the root to the leaves while in the bottom-up approach the parse tree is built from the leaves upward to the root.

Parse Trees VI

Note that syntax analyzers for programming languages, which are often called *parsers*, construct parse trees for given programs. Some systems construct parse trees only implicitly, but they also use the whole information provided by the parse tree during the parse. There are two major approaches of how to build these parse trees. One is *top-down* and the other one is *bottom-up*. In the top-down approach the parse tree is built from the root to the leaves while in the bottom-up approach the parse tree is built from the leaves upward to the root.

A major problem one has to handle when constructing such parsers is ***ambiguity***. We thus direct our attention to this problem.

©Thomas Zeugmann

Ambiguity I

Definition 2

A grammar that generates a sentence for which there are two or more distinct parse trees is said to be ***ambiguous***.

©Thomas Zeugmann

Ambiguity I

Definition 2

A grammar that generates a sentence for which there are two or more distinct parse trees is said to be ***ambiguous***.

For having an example, let us look at the following part of a grammar given in Example 3. At first glance this grammar looks quite similar to the one considered above. The only difference is that the production for expressions has been altered by replacing ⟨id⟩ by ⟨expr⟩.

©Thomas Zeugmann

Ambiguity I

Definition 2

A grammar that generates a sentence for which there are two or more distinct parse trees is said to be ***ambiguous***.

For having an example, let us look at the following part of a grammar given in Example 3. At first glance this grammar looks quite similar to the one considered above. The only difference is that the production for expressions has been altered by replacing ⟨`id`⟩ by ⟨`expr`⟩.

However, this small modification leads to serious problems, because now the grammar provides slightly less syntactic structure than the grammar considered in Example 2 does.

Ambiguity II

Example 3

$$
\begin{array}{rcl}
\langle \texttt{assign} \rangle & \rightarrow & \langle \texttt{id} \rangle := \langle \texttt{expr} \rangle \\
\langle \texttt{id} \rangle & \rightarrow & \texttt{A} \mid \texttt{B} \mid \texttt{C} \\
\langle \texttt{expr} \rangle & \rightarrow & \langle \texttt{expr} \rangle + \langle \texttt{expr} \rangle \\
& \mid & \langle \texttt{expr} \rangle * \langle \texttt{expr} \rangle \\
& \mid & (\langle \texttt{expr} \rangle) \\
& \mid & \langle \texttt{id} \rangle
\end{array}
$$

For seeing that this grammar is ambiguous, let us look at the following assignment statement:

$$\texttt{A} := \texttt{B} + \texttt{C} * \texttt{A} \,.$$

We skip the two formal derivations possible for this assignment and look directly at the two parse trees.

©Thomas Zeugmann

Ambiguity III

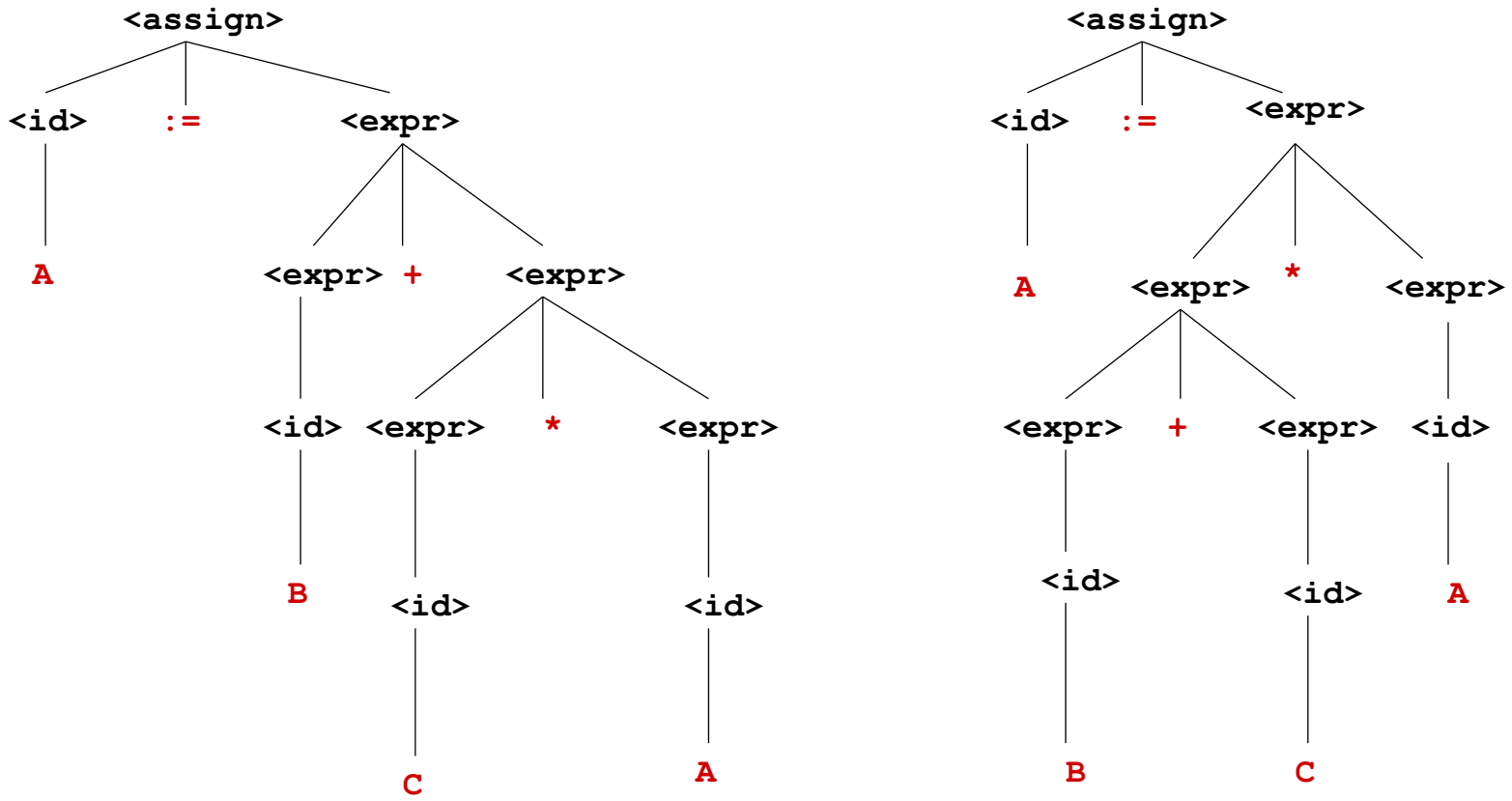


Figure 2: Two parse trees for the same sentence $A := B + C * A$

$A := B + (C * A)$

$A := (B + C) * A.$

©Thomas Zeugmann

Ambiguity IV

These two distinct parse trees cause problems because compilers base the semantics of the sentences on their syntactic structure. In particular, compilers decide what code to generate by examining the parse tree. So, in our example the *semantics* is not clear. Let us examine this problem in some more detail.

Ambiguity IV

These two distinct parse trees cause problems because compilers base the semantics of the sentences on their syntactic structure. In particular, compilers decide what code to generate by examining the parse tree. So, in our example the *semantics* is not clear. Let us examine this problem in some more detail.
In the first parse tree (the left one) of Figure 2 the multiplication operator is generated lower in the tree which would indicate that it has precedence over the addition operator in the expression.

©Thomas Zeugmann

Ambiguity IV

These two distinct parse trees cause problems because compilers base the semantics of the sentences on their syntactic structure. In particular, compilers decide what code to generate by examining the parse tree. So, in our example the *semantics* is not clear. Let us examine this problem in some more detail.
In the first parse tree (the left one) of Figure 2 the multiplication operator is generated lower in the tree which would indicate that it has precedence over the addition operator in the expression.
The second parse tree in Figure 2, however, is just indicating the opposite. Clearly, in dependence on what decision the compiler makes, the result of an actual evaluation of the assignment given will be either the expected one (that is multiplication has precedence over addition) or an erroneous one.

©Thomas Zeugmann

Ambiguity V

Although the grammar in Example 2 is not ambiguous, the precedence order of its operators is not the usual one. Rather, in this grammar, a parse tree of a sentence with multiple operators has the rightmost operator at the lowest point, with the other operators in the tree moving progressively higher as one moves to the left in the expression.

©Thomas Zeugmann

Ambiguity V

Although the grammar in Example 2 is not ambiguous, the precedence order of its operators is not the usual one. Rather, in this grammar, a parse tree of a sentence with multiple operators has the rightmost operator at the lowest point, with the other operators in the tree moving progressively higher as one moves to the left in the expression.

So, one has to think about a way to overcome this difficulty and to clearly define the usual operator precedence between multiplication and addition, or more generally with any desired operator precedence. As a matter of fact, this goal can be achieved for *our example* by using separate nonterminals for the operands of the operators that have different precedence. This not only requires additional nonterminals but also additional productions.

©Thomas Zeugmann

Ambiguity VI

Question

Can we always detect ambiguity?
If ambiguity has been detected, can it always be removed?

©Thomas Zeugmann

Ambiguity VI

Question

Can we always detect ambiguity?
If ambiguity has been detected, can it always be removed?

The answers are given by the following theorems.

©Thomas Zeugmann

Ambiguity VI

Question

Can we always detect ambiguity?
If ambiguity has been detected, can it always be removed?

The answers are given by the following theorems.

Theorem 1

There is no algorithm deciding whether or not any context-free grammar is ambiguous.

©Thomas Zeugmann

Ambiguity VI

Question

Can we always detect ambiguity?
If ambiguity has been detected, can it always be removed?

The answers are given by the following theorems.

Theorem 1

There is no algorithm deciding whether or not any context-free grammar is ambiguous.

Theorem 2

There are context-free languages that have nothing but ambiguous grammars.

©Thomas Zeugmann

Ambiguity VII

But in practice the situation is not as grim as it may seem. Many techniques have been proposed to eliminate ambiguity in the sorts of constructs that typically appear in programming languages.

©Thomas Zeugmann

Ambiguity VII

But in practice the situation is not as grim as it may seem. Many techniques have been proposed to eliminate ambiguity in the sorts of constructs that typically appear in programming languages.

So, let us come back to our example and let us show how to eliminate the ambiguity we have detected. We continue by providing a grammar generating the same language as the grammars of Examples 2 and 3, but which clearly indicates the usual precedence order of multiplication and addition.

©Thomas Zeugmann

Ambiguity VIII

Example 4

$$
\begin{array}{rcl}
\langle \texttt{assign} \rangle & \rightarrow & \langle \texttt{id} \rangle := \langle \texttt{expr} \rangle \\
\langle \texttt{id} \rangle & \rightarrow & \texttt{A} \mid \texttt{B} \mid \texttt{C} \\
\langle \texttt{expr} \rangle & \rightarrow & \langle \texttt{expr} \rangle + \langle \texttt{term} \rangle \\
& \mid & \langle \texttt{term} \rangle \\
\langle \texttt{term} \rangle & \rightarrow & \langle \texttt{term} \rangle * \langle \texttt{factor} \rangle \\
& \mid & \langle \texttt{factor} \rangle \\
\langle \texttt{factor} \rangle & \rightarrow & (\langle \texttt{expr} \rangle) \\
& \mid & \langle \texttt{id} \rangle
\end{array}
$$

©Thomas Zeugmann

Ambiguity VIII

Example 4

$$
\begin{array}{rcl}
\langle \texttt{assign} \rangle & \rightarrow & \langle \texttt{id} \rangle := \langle \texttt{expr} \rangle \\
\langle \texttt{id} \rangle & \rightarrow & \texttt{A} \mid \texttt{B} \mid \texttt{C} \\
\langle \texttt{expr} \rangle & \rightarrow & \langle \texttt{expr} \rangle + \langle \texttt{term} \rangle \\
& \mid & \langle \texttt{term} \rangle \\
\langle \texttt{term} \rangle & \rightarrow & \langle \texttt{term} \rangle * \langle \texttt{factor} \rangle \\
& \mid & \langle \texttt{factor} \rangle \\
\langle \texttt{factor} \rangle & \rightarrow & (\langle \texttt{expr} \rangle) \\
& \mid & \langle \texttt{id} \rangle
\end{array}
$$

Next, let us derive the same statement as above, i.e.,

$$\texttt{A} := \texttt{B} + \texttt{C} * \texttt{A} \,.$$

©Thomas Zeugmann

Ambiguity IX

The derivation is unambiguously obtained as follows:

$$
\begin{aligned}
\langle \texttt{assign} \rangle &\Rightarrow \langle \texttt{id} \rangle := \langle \texttt{expr} \rangle \Rightarrow \texttt{A} := \langle \texttt{expr} \rangle \\
&\Rightarrow \texttt{A} := \langle \texttt{expr} \rangle + \langle \texttt{term} \rangle \\
&\Rightarrow \texttt{A} := \langle \texttt{term} \rangle + \langle \texttt{term} \rangle \\
&\Rightarrow \texttt{A} := \langle \texttt{factor} \rangle + \langle \texttt{term} \rangle \\
&\Rightarrow \texttt{A} := \langle \texttt{id} \rangle + \langle \texttt{term} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \langle \texttt{term} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \langle \texttt{term} \rangle * \langle \texttt{factor} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \langle \texttt{factor} \rangle * \langle \texttt{factor} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \langle \texttt{id} \rangle * \langle \texttt{factor} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \texttt{C} * \langle \texttt{factor} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \texttt{C} * \langle \texttt{id} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \texttt{C} * \texttt{A}
\end{aligned}
$$

©Thomas Zeugmann

Ambiguity X

Furthermore, you should note that we have presented a *leftmost derivation* above, i.e., the leftmost nonterminal has always been handled first until it was replaced by a terminal. Thus, the sentence given above has more than one derivation. If we had always handled the rightmost nonterminal first, then we would have been arrived at a *rightmost derivation*.

©Thomas Zeugmann

Ambiguity X

Furthermore, you should note that we have presented a *leftmost derivation* above, i.e., the leftmost nonterminal has always been handled first until it was replaced by a terminal. Thus, the sentence given above has more than one derivation. If we had always handled the rightmost nonterminal first, then we would have been arrived at a *rightmost derivation*.

Clearly, one could also choose arbitrarily the nonterminal which allows the application of a production. Try it out, and construct the resulting parse trees. Usually, one is implicitly assuming that derivations are leftmost. Thus, we can more precisely say that a context-free grammar is *ambiguous* if there is a sentence in the language it generates that possesses at least two different leftmost derivations. Otherwise it is called *unambiguous*. A language is said to be unambiguous if there is an unambiguous grammar for it.

©Thomas Zeugmann

Ambiguity XI

Another important problem in describing programming languages is to express that operators are *associative*. As you have learned in mathematics, addition and multiplication are associative. Is this also true for computer arithmetic? As far as integer addition and multiplication are concerned, they are associative. But floating point computer arithmetic is not always associative. So, in general correct associativity is essential. The book provides more information.

Ambiguity XI

Another important problem in describing programming languages is to express that operators are *associative*. As you have learned in mathematics, addition and multiplication are associative. Is this also true for computer arithmetic? As far as integer addition and multiplication are concerned, they are associative. But floating point computer arithmetic is not always associative. So, in general correct associativity is essential. The book provides more information.

As an exercise you should try to provide an *unambiguous* grammar for the **if-then-else** statement that is present in many programming languages.

©Thomas Zeugmann

Ambiguity XI

Another important problem in describing programming languages is to express that operators are *associative*. As you have learned in mathematics, addition and multiplication are associative. Is this also true for computer arithmetic? As far as integer addition and multiplication are concerned, they are associative. But floating point computer arithmetic is not always associative. So, in general correct associativity is essential. The book provides more information.

As an exercise you should try to provide an *unambiguous* grammar for the **if-then-else** statement that is present in many programming languages.

We continue with further properties of context-free grammars.

©Thomas Zeugmann

Separated Grammars I

In a context-free grammar, there is no *a priori* bound on the size of a right-hand side of a production. This may complicate many proofs. But, there is a *normal form for context-free grammars* bounding the right-hand side to be of length at most 2. Knowing and applying this considerably simplifies many proofs.

First, we define the notion of a separated grammar.

Separated Grammars I

In a context-free grammar, there is no *a priori* bound on the size of a right-hand side of a production. This may complicate many proofs. But, there is a *normal form for context-free grammars* bounding the right-hand side to be of length at most 2. Knowing and applying this considerably simplifies many proofs.

First, we define the notion of a separated grammar.

Definition 3

A grammar $\mathcal{G} = [T, N, \sigma, P]$ is called ***separated*** if for all $(\alpha \rightarrow \beta) \in P$ we either have $\alpha, \beta \in N^*$ or $\alpha \in N$ and $\beta \in T$.

Separated Grammars I

In a context-free grammar, there is no *a priori* bound on the size of a right-hand side of a production. This may complicate many proofs. But, there is a *normal form for context-free grammars* bounding the right-hand side to be of length at most 2. Knowing and applying this considerably simplifies many proofs.

First, we define the notion of a separated grammar.

Definition 3

A grammar $\mathcal{G} = [T, N, \sigma, P]$ is called ***separated*** if for all $(\alpha \rightarrow \beta) \in P$ we either have $\alpha, \beta \in N^*$ or $\alpha \in N$ and $\beta \in T$.

Theorem 3

For every context-free grammar $\mathcal{G} = [T, N, \sigma, P]$ there exists an equivalent separated context-free grammar $\mathcal{G}' = [T, N', \sigma, P']$.

Separated Grammars II

Proof. First, we introduce for every $t \in T$ a new nonterminal symbol h_t, where by new we mean that $h_t \notin N$ for all $t \in T$. Furthermore, we set $N' = \{h_t \mid t \in T\} \cup N$.

Separated Grammars II

Proof. First, we introduce for every $t \in T$ a new nonterminal symbol h_t, where by new we mean that $h_t \notin N$ for all $t \in T$. Furthermore, we set $N' = \{h_t \mid t \in T\} \cup N$.
Next, for $(\alpha \to \beta) \in P$, we denote the production obtained by replacing every terminal symbol t in β by h_t by $(\alpha \to \beta)[t/\!\!/h_t]$. The production set P' is then defined as follows.

$$P' = \{(\alpha \to \beta)[t/\!\!/h_t] \mid (\alpha \to \beta) \in P\} \cup \{h_t \to t \mid t \in T\}.$$

Separated Grammars II

Proof. First, we introduce for every $t \in T$ a new nonterminal symbol h_t, where by new we mean that $h_t \notin N$ for all $t \in T$. Furthermore, we set $N' = \{h_t \mid t \in T\} \cup N$.
Next, for $(\alpha \to \beta) \in P$, we denote the production obtained by replacing every terminal symbol t in β by h_t by $(\alpha \to \beta)[t/\!\!/h_t]$. The production set P' is then defined as follows.

$$P' = \{(\alpha \to \beta)[t/\!\!/h_t] \mid (\alpha \to \beta) \in P\} \cup \{h_t \to t \mid t \in T\}.$$

By construction, we directly see that $\mathcal{G}'$ is separated.
Moreover, the construction ensures that $\mathcal{G}'$ is context-free, too.

Separated Grammars III

It remains to show that $L(\mathcal{G}) = L(\mathcal{G}')$.

Claim 1. $L(\mathcal{G}) \subseteq L(\mathcal{G}')$.

Let $s \in L(\mathcal{G})$. Then there exists a derivation

$$\sigma \overset{1}{\Rightarrow} w_1 \overset{1}{\Rightarrow} w_2 \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w_n \overset{1}{\Rightarrow} s\,,$$

where $w_1, \ldots, w_n \in (N \cup T)^+ \setminus T^*$, and $s \in T^*$. Let P_i be the production used to generate w_i, $i = 1, \ldots, n$.

Separated Grammars III

It remains to show that $L(\mathcal{G}) = L(\mathcal{G}')$.

Claim 1. $L(\mathcal{G}) \subseteq L(\mathcal{G}')$.

Let $s \in L(\mathcal{G})$. Then there exists a derivation

$$\sigma \overset{1}{\Rightarrow} w_1 \overset{1}{\Rightarrow} w_2 \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w_n \overset{1}{\Rightarrow} s\,,$$

where $w_1, \ldots, w_n \in (N \cup T)^+ \setminus T^*$, and $s \in T^*$. Let P_i be the production used to generate w_i, $i = 1, \ldots, n$.

Then we can generate s by using productions from P' as follows. Let $s = s_1 \cdots s_m$, where $s_j \in T$ for all $j = 1, \ldots, m$. Instead of applying P_i we use $P_i[t/\!/h_t]$ from P' and obtain

$$\sigma \overset{1}{\Rightarrow} w_1' \overset{1}{\Rightarrow} w_2' \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w_n' \overset{1}{\Rightarrow} h_{s_1} \cdots h_{s_m}\,,$$

where now $w_i' \in (N')^+$ for all $i = 1, \ldots, n$.

Separated Grammars III

It remains to show that $L(\mathcal{G}) = L(\mathcal{G}')$.

Claim 1. $L(\mathcal{G}) \subseteq L(\mathcal{G}')$.

Let $s \in L(\mathcal{G})$. Then there exists a derivation

$$\sigma \overset{1}{\Rightarrow} w_1 \overset{1}{\Rightarrow} w_2 \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w_n \overset{1}{\Rightarrow} s\,,$$

where $w_1, \ldots, w_n \in (N \cup T)^+ \setminus T^*$, and $s \in T^*$. Let P_i be the production used to generate w_i, $i = 1, \ldots, n$.

Then we can generate s by using productions from P' as follows. Let $s = s_1 \cdots s_m$, where $s_j \in T$ for all $j = 1, \ldots, m$. Instead of applying P_i we use $P_i[t/\!/h_t]$ from P' and obtain

$$\sigma \overset{1}{\Rightarrow} w_1' \overset{1}{\Rightarrow} w_2' \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w_n' \overset{1}{\Rightarrow} h_{s_1} \cdots h_{s_m}\,,$$

where now $w_i' \in (N')^+$ for all $i = 1, \ldots, n$. Thus, in order to obtain s it now suffices to apply the productions $h_{s_j} \to s_j$ for $j = 1, \ldots m$. This proves Claim 1.

Separated Grammars IV

Claim 2. $L(\mathcal{G}') \subseteq L(\mathcal{G})$.

This claim can be proved analogously by inverting the construction used in showing Claim 1. ∎

Chomsky Normal Form I

Now, we are ready to define the normal form for context-free grammars announced above.

Definition 4

A grammar $\mathcal{G} = [T, N, \sigma, P]$ is said to be in ***Chomsky normal form*** if all productions of P have the form $h \rightarrow h_1 h_2$, where $h, h_1, h_2 \in N$, or $h \rightarrow x$, where $h \in N$ and $x \in T$.

Chomsky Normal Form I

Now, we are ready to define the normal form for context-free grammars announced above.

Definition 4

A grammar $\mathcal{G} = [T, N, \sigma, P]$ is said to be in ***Chomsky normal form*** if all productions of P have the form $h \to h_1 h_2$, where $h, h_1, h_2 \in N$, or $h \to x$, where $h \in N$ and $x \in T$.

The latter definition directly allows the following corollary.

Corollary 4

Let $\mathcal{G} = [T, N, \sigma, P]$ be a grammar in Chomsky normal form. Then

(1) *$\mathcal{G}$ is context-free,*

(2) *$\mathcal{G}$ is λ-free,*

(3) *$\mathcal{G}$ is separated.*

Chomsky Normal Form II

Now, we can show:

Theorem 5

For every context-free grammar $\mathcal{G} = [T, N, \sigma, P]$ such that $\lambda \notin L(\mathcal{G})$ there exists an equivalent grammar $\mathcal{G}'$ that is in Chomsky normal form.

Chomsky Normal Form II

Now, we can show:

Theorem 5

For every context-free grammar $\mathcal{G} = [T, N, \sigma, P]$ such that $\lambda \notin L(\mathcal{G})$ there exists an equivalent grammar $\mathcal{G}'$ that is in Chomsky normal form.

Proof. Let $\mathcal{G} = [T, N, \sigma, P]$ be given. Without loss of generality, we may assume that $\mathcal{G}$ is reduced.

Chomsky Normal Form III

First, we eliminate all productions of the form $h \to h'$. This is done as follows. We set

$$W_0(h) = \{h\} \text{ for every } h \in N$$

and for each $i \geqslant 0$ we define

$$W_{i+1}(h) = W_i(h) \cup \{\tilde{h} \mid \tilde{h} \in N,\ (\hat{h} \to \tilde{h}) \in P \text{ for some } \hat{h} \in W_i(h)\}.$$

Chomsky Normal Form III

First, we eliminate all productions of the form $h \to h'$. This is done as follows. We set

$$W_0(h) = \{h\} \text{ for every } h \in N$$

and for each $i \geqslant 0$ we define

$$W_{i+1}(h) = W_i(h) \cup \{\tilde{h} \mid \tilde{h} \in N,\ (\hat{h} \to \tilde{h}) \in P \text{ for some } \hat{h} \in W_i(h)\}.$$

Then, the following facts are obvious:

(1) $W_i(h) \subseteq W_{i+1}(h)$ for all $i \geqslant 0$,

(2) If $W_i(h) = W_{i+1}(h)$ then $W_i(h) = W_{i+m}(h)$ for all $m \in \mathbb{N}$,

(3) $W_n(h) = W_{n+1}(h)$ for $n = \text{card}(N)$,

(4) $W_n(h) = \{B \mid B \in N \text{ and } h \overset{*}{\Rightarrow} B\}$.

Chomsky Normal Form IV

Now, we define

$$P_1 = \{h \to \gamma \mid h \in N,\ \gamma \notin N,\ (B \to \gamma) \in P \text{ for some } B \in W_n(h)\}.$$

Let $G_1 = [T, N, \sigma, P_1]$,

Chomsky Normal Form IV

Now, we define

$$P_1 = \{h \to \gamma \mid h \in N,\ \gamma \notin N,\ (B \to \gamma) \in P \text{ for some } B \in W_n(h)\}.$$

Let $G_1 = [T, N, \sigma, P_1]$, then by construction P_1 does *not* contain any production of the form $h \to h'$. These productions have been replaced by $h \to \gamma$. That is, if we had $h \overset{*}{\Rightarrow} B$ by using the productions from P and $B \to \gamma$, then we now have the production $h \to \gamma$ in P_1. Also note that P_1 contains all original productions $(h \to \gamma) \in P$, where $\gamma \notin N$ by the definition of W_0.

We leave it as an exercise to formally verify that $L(G_1) = L(G)$.

Chomsky Normal Form V

Next, from $\mathcal{G}_1$ we construct an equivalent separated grammar $\mathcal{G}_2$ by using the algorithm given above. Now, the only productions in P_2 that still need modification are of the form

$$h \rightarrow h_1 h_2 \cdots h_n \text{ , where } n \geqslant 3 \text{ .}$$

Chomsky Normal Form V

Next, from $\mathcal{G}_1$ we construct an equivalent separated grammar $\mathcal{G}_2$ by using the algorithm given above. Now, the only productions in P_2 that still need modification are of the form

$$h \rightarrow h_1 h_2 \cdots h_n \text{ , where } n \geqslant 3 \text{ .}$$

We replace any such production by the following productions

$$\begin{aligned} h &\rightarrow h_1 h_{h_2 \cdots h_n} \\ h_{h_2 \cdots h_n} &\rightarrow h_2 h_{h_3 \cdots h_n} \\ &\vdots \\ h_{h_{n-1} h_n} &\rightarrow h_{n-1} h_n \end{aligned}$$

Chomsky Normal Form V

Next, from $\mathcal{G}_1$ we construct an equivalent separated grammar $\mathcal{G}_2$ by using the algorithm given above. Now, the only productions in P_2 that still need modification are of the form

$$h \rightarrow h_1 h_2 \cdots h_n \text{ , where } n \geqslant 3 \text{ .}$$

We replace any such production by the following productions

$$\begin{aligned} h &\rightarrow h_1 h_{h_2 \cdots h_n} \\ h_{h_2 \cdots h_n} &\rightarrow h_2 h_{h_3 \cdots h_n} \\ &\vdots \\ h_{h_{n-1} h_n} &\rightarrow h_{n-1} h_n \end{aligned}$$

Hence, the resulting grammar $\mathcal{G}'$ is in Chomsky normal form and by construction equivalent to $\mathcal{G}$. We omit the details. ∎

Pumping Lemma I

We finish this lecture by pointing to an important result which can be proved by using the Chomsky normal form. This result is usually referred to as Pumping Lemma for context-free languages or Lemma of Bar-Hillel or $qrsuv$-Theorem.

Pumping Lemma I

We finish this lecture by pointing to an important result which can be proved by using the Chomsky normal form. This result is usually referred to as Pumping Lemma for context-free languages or Lemma of Bar-Hillel or $qrsuv$-Theorem.

Please note that the $qrsuv$-Theorem provides a necessary condition for a language to be context-free. It is not sufficient. So its main importance lies in the fact that one can often use it to show that a language is *not* context-free.

Pumping Lemma II

Theorem 6 ($qrsuv$-Theorem, Pumping Lemma)

For every context-free language L *there is a number* k *such that for every* $w \in L$ *with* $|w| \geqslant k$ *there are strings* q, r, s, u, v *such that*

(1) $w = qrsuv$,

(2) $|rsu| \leqslant k$,

(3) $ru \neq \lambda$, *and*

(4) $qr^i su^i v \in L$ *for all* $i \in \mathbb{N}$.

©Thomas Zeugmann

Pumping Lemma II

Theorem 6 ($qrsuv$-Theorem, Pumping Lemma)

For every context-free language L *there is a number* k *such that for every* $w \in L$ *with* $|w| \geqslant k$ *there are strings* q, r, s, u, v *such that*

(1) $w = qrsuv$,

(2) $|rsu| \leqslant k$,

(3) $ru \neq \lambda$, *and*

(4) $qr^i su^i v \in L$ *for all* $i \in \mathbb{N}$.

Proof. Let $\mathcal{G} = [T, N, \sigma, P]$ be any context-free grammar in CNF for L. Then all parse trees are binary trees except the last derivation step.

Pumping Lemma III

Let $n = \mathrm{card}(N)$ and $k = 2^n$ and consider the parse tree for any string $w \in L(G)$ with $|w| \geqslant k$. So, any of the parse trees for w must have depth at least n. Hence, there must exist a path from the root σ to a leaf having length at least n. Including σ there are $n+1$ nonterminals on this path. Hence, some nonterminal h has to appear at least twice (see Figure 3, Part (a)).

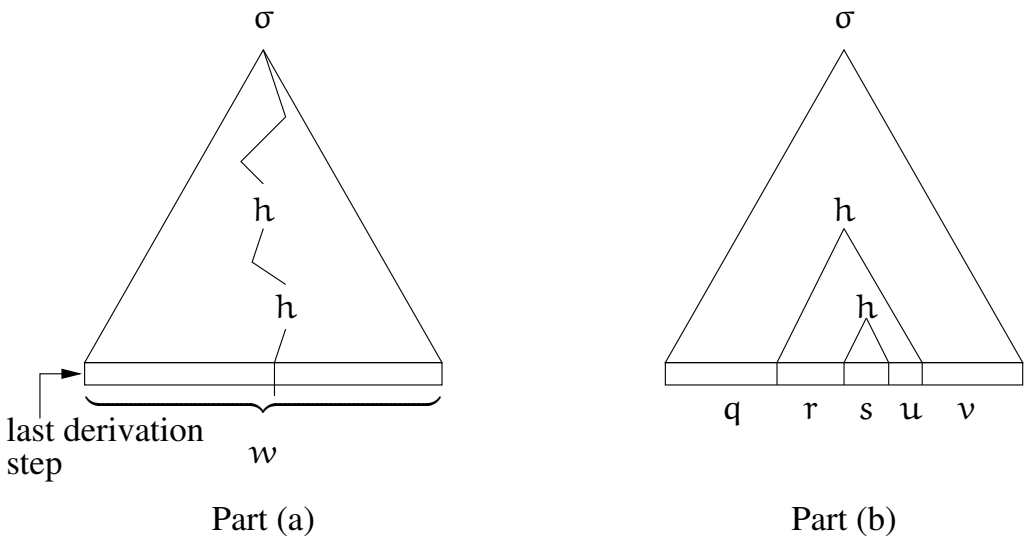


Figure 3: Parse trees for illustrating the proof of the $qrsuv$-Theorem

Pumping Lemma IV

Now, starting from the leaves and going up, we fix a path containing two times h and we fix the first two occurrences of h. In this way, we guarantee that the higher located occurrence of h is at most n steps above the leaves.

Pumping Lemma IV

Now, starting from the leaves and going up, we fix a path containing two times h and we fix the first two occurrences of h. In this way, we guarantee that the higher located occurrence of h is at most n steps above the leaves.

Next, we look at the substrings that are generated from these two occurrences of h. This gives us the substrings $qrsuv$ (see Figure 3, Part (b)).

©Thomas Zeugmann

Pumping Lemma IV

Now, starting from the leaves and going up, we fix a path containing two times h and we fix the first two occurrences of h. In this way, we guarantee that the higher located occurrence of h is at most n steps above the leaves.

Next, we look at the substrings that are generated from these two occurrences of h. This gives us the substrings $qrsuv$ (see Figure 3, Part (b)).

Because G is in CNF, at the higher occurrence of h a production of the form $h \rightarrow h_1 h_2$ must have been applied. Consequently, $r \neq \lambda$ or $u \neq \lambda$, and Assertion (3) is shown.

Pumping Lemma V

Assertion (2) follows from the fact that the higher located occurrence of h is at most n steps above the leaves. Thus, the string rsu derived from this h can have length at most $2^n = k$.

Pumping Lemma V

Assertion (2) follows from the fact that the higher located occurrence of h is at most n steps above the leaves. Thus, the string rsu derived from this h can have length at most $2^n = k$.

It remains to show Assertion (4). This is done by modifying the parse tree. First, we may remove the subtree rooted at the higher located occurrence of h and replace it by the subtree rooted at the lower located occurrence of h (see Figure 4, Part (a)). In this way, we get a generation of qsv, i.e., $qr^0su^0v \in L(\mathcal{G})$.

Pumping Lemma VI

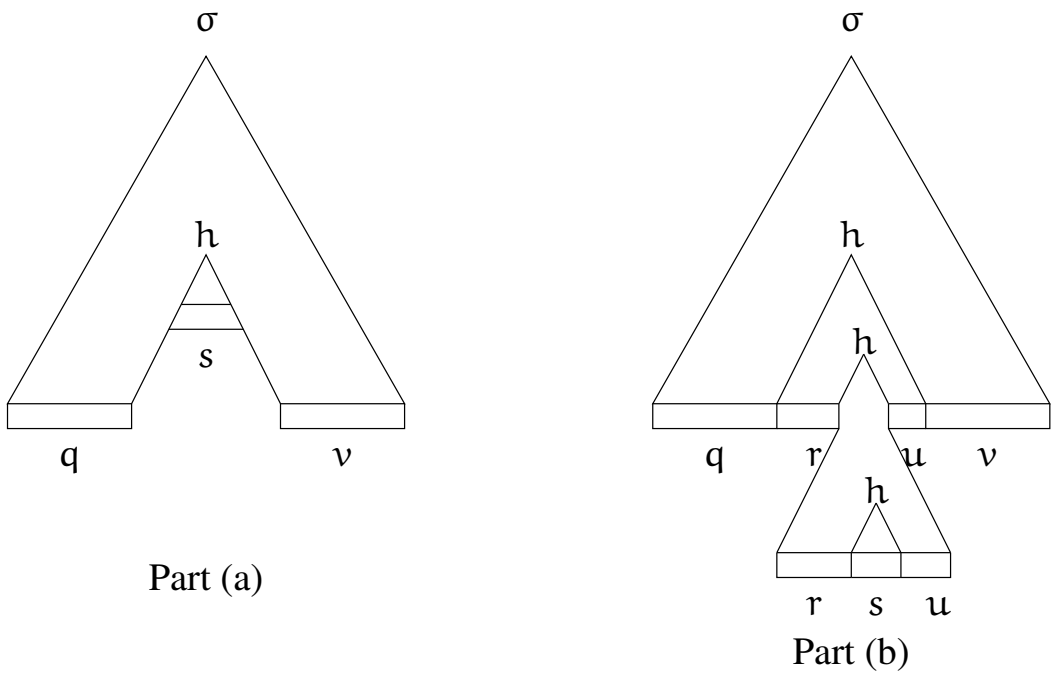


Figure 4: Illustrating Assertion (4) of the qrsuv-Theorem

Pumping Lemma VII

Second, we remove the subtree rooted at the lower located occurrence of h and replace it by the subtree rooted at the higher located occurrence of h (see Figure 4, Part (b)). In this way, we get a derivation of qr^2su^2v and thus, $qr^2su^2v \in L(\mathcal{G})$. Iterating this idea shows $qr^isu^iv \in L(\mathcal{G})$ for every $i \in \mathbb{N}^+$. ∎

Pumping Lemma VII

Second, we remove the subtree rooted at the lower located occurrence of h and replace it by the subtree rooted at the higher located occurrence of h (see Figure 4, Part (b)). In this way, we get a derivation of qr^2su^2v and thus, $qr^2su^2v \in L(\mathcal{G})$. Iterating this idea shows $qr^isu^iv \in L(\mathcal{G})$ for every $i \in \mathbb{N}^+$. ∎

For having an application of Theorem 6, we show the following Theorem, thus completing the proof of Theorem 6.5.

Pumping Lemma VIII

Theorem 7

$L = \{a^n b^n c^n \mid n \in \mathbb{N}\} \notin \mathcal{CF}$.

Proof. Suppose the converse. Then, by Theorem 6 there is a k such that for all $w \in L$ with $|w| \geqslant k$ we have:

There are substrings q, r, s, u, v with $|ru| > 0$ and for all $w = qrsuv \in L$ we must have $qr^i su^i v \in L$, too, for all $i \in \mathbb{N}$. Now, consider $w = a^k b^k c^k$. Consequently, $|w| = 3k > k$. By Assertion (2) of Theorem 6 we have $|rsu| \leqslant k$, and thus ru cannot contain a's, b's *and* c's. Since $|ru| > 0$ and by Assertion (4) (applied for $i = 0$), we must have $qr^0su^0v = qsv \in L$. But, as discussed above, $qsv \neq a^\ell b^\ell c^\ell$ for every $\ell \in \mathbb{N}$, a contradiction. ∎

Thank you!

©Thomas Zeugmann