5-1-2021

Why Chomsky Normal Form: A Pedagogical Note

Olga Kosheleva
The University of Texas at El Paso, olgak@utep.edu

Vladik Kreinovich
The University of Texas at El Paso, vladik@utep.edu

Comments:
Technical Report: UTEP-CS-21-47

Why Chomsky Normal Form: A Pedagogical Note

Olga Kosheleva and Vladik Kreinovich

Abstract To simplify the design of compilers, Noam Chomsky proposed to first transform a description of a programming language – which is usually given in the form of a context-free grammar – into a simplified "normal" form. A natural question is: why this specific normal form? In this paper, we provide an answer to this question.

1 Formulation of the Problem: Why Chomsky Normal Form?

How programming languages are usually described. The usual way to describe a programming language is by introducing special auxiliary notions. For example:

- The notion of a *digit* can be described as 0, 1, . . . , or 9.
- An *unsigned integer* can be described as either a digit, or a digit followed by an integer.
- An *if-then statement* can be described as the word *if* followed by an opening parenthesis, a condition, a closing parenthesis, and a statement.

One way to describe this in precise terms is by using *context-free grammars*; see, e.g., [1]. In this description, we separate:

- symbols that will appear in the resulting program; such symbols are called *terminal*, and

Olga Kosheleva
Department of Teacher Education, University of Texas at El Paso, 500 W. University
El Paso, TX 79968, USA
e-mail: olgak@utep.edu

Vladik Kreinovich
Department of Computer Science, University of Texas at El Paso, 500 W. University
El Paso, TX 79968, USA
e-mail: vladik@utep.edu

- symbols describing auxiliary notions – like integer or digit – that will *not* appear in the final program; these symbols are called *variables*.

In a formal description, terminal symbols are usually described by small letters, and variables by capital letters. In terms of such symbols, the above informal descriptions are written down as *rules*. For example:

- The fact that 0 is a digit (D) can be written as $D \to 0$.
- Similarly, the fact that $1, \ldots, 9$ are digits can be written as

$$D \to 1, \ldots, D \to 9.$$

In general, a rule $S \to r_1 \ldots r_k$ means that if we have a combination of texts corresponding to $r_1, \ldots, r_k$, then this combination is of type S. For example:

- The above description of an unsigned integer (I) can reformulated as the rules

$$I \to D \text{ and } I \to DI.$$

- The description of an if-then statement (T) can be reformulated into the rule

$$T \to if(C)S,$$

  where C means a condition and S is a statement.

In a description of a programming language, we start with a notion of a program – and in general, we start with some variable which is called *starting variable*. Then, we can repeatedly use the rules to replace each notion with its clarification – until we get to a text that only includes terminal symols.

For example, to show that 2021 is an unsigned integer, we can start with I and then:

- first, we apply the rule $I \to DI$;
- we then apply the rule $D \to 2$ to replace D with 2, and the rule $I \to DI$ to get

$$I \to DI \to 2DI;$$

- we apply the rule $D \to 0$ to replace D with 0, and the rule $I \to DI$ to get

$$I \to DI \to 2DI \to 20DI;$$

- we apply the rule $D \to 2$ to replace D with 2, and the rule $I \to D$ to get

$$I \to DI \to 2DI \to 20DI \to 202D;$$

- finally, we apply the rule $D \to 1$ to get

$$I \to DI \to 2DI \to 20DI \to 202D \to 2021.$$

Why do we need this formal description? But why do we need to translate a clear and understandable natural-language description into a barely understandable formal one?

The answer becomes clear if we take into account that the whole idea of a programming language is that:

- we write a program, and
- the computer will automatically translate it into executable code and implement it.

Unfortunately, computers do not understand natural language well. So, to have an automatic way of designing a compiler based on the description of the programming language, we need to translate the original description into a precise language – a language that a computer can understand.

Need for a normal form. There exist such "compiler compilers" that automatically produce a compiler based on the description of a programming language. Probably the best known is yacc – short of Yet Another Compiler Compiler – which is part of a usual Unix setting.

The problem is that context-free grammars can be very complicated, with long and complex rules. It is therefore desirable to be able to describe the original language in a simplified ("normal") form.

Chomsky normal form. The first such simplified form was produced by Noam Chomsky, the famous linguist and the author of many concepts actively used in programming languages [1]. He showed that every context-free grammar can be transformed into a simplified form, in which only three types of rules are allowed:

- a rule $S \to \varepsilon$, where S is a starting variable, and ε means an empty string;
- rules of the type $V \to a$, where V is a variable and a is a terminal symbol; and
- rules of the type $A \to BC$, where A, B, and C are variables.

Why Chomsky normal form? A natural question is: why these three types of rules?

In this paper, we provide an answer to this question.

2 Analysis of the Problem and the Resulting Explanation

Let us restrict the length of the right-hand sides. The longer the right-hand side of the rule, the more complex this rule. Thus, to make the description simpler, it is desirable to restrict the lengths of the right-hand sides.

The fact that every context-free grammar can be transformed into Chomsky normal form – in which every rule has right-hand side of length at most 2 – shows that it is possible to have a normal form in which all these lengths are bounded by 2.

Can we bound it further, to 1 or 0? Not really: if we only have rules in which the length of the right-hand side is 0 or 1, i.e., in which the right-hand side is either an empty string or a single symbol, then, since we start with a single symbol, we will

only get one-letter words – and many programs have more than one letter. So, we do need rules in which the right-hand side has length 2.

What rules with 2-symbol right-hand sides can we have? Each of the two symbols on the right-hand side of a rule can be either a terminal symbol a, b, ..., or a variable A, B, ... Thus, we can have four possible types of such rules: $A \to bc$, $A \to bC$, $A \to Bc$, and $A \to BC$.

For simplicity, it is desirable to restrict ourselves to rules of only one of these four types. Which type should we choose so that we will still be able to describe any context-free grammar in this form?

Suppose first that we only allow rules of the type $A \to bc$. All other rules – with right-hand sides of length 0 or 1 – do not increase the length of the word. So, using rules $A \to bc$ is the only way to get words longer than one symbol. Thus, we can get some 2-symbol words. However, these words do not contain variables, so we cannot apply any rules to make them longer. Thus, with this type of rules, we will only get 2-letter words, not enough to describe all possible programming languages.

What if we only allow rules of the type $A \to bC$? It is known that such rules correspond to finite automata – every finite automaton can be represented as such a grammar if:

- we assign, to each state of this automaton, a variable, and
- we transform each transition $a \stackrel{b}{\to} c$ into a rule $A \to bC$.

It is known that not all context-free grammars can be described by finite automata; see, e.g., [1]. So this restriction also does not allow us to represent all possible context-free languages.

Similarly, if we only allow rules of the type $A \to Bc$, then the resulting language consists of reverses of all the words obtained by using reversed rules $A \to cB$. The language of reverses is thus obtainable by a finite automaton – and hence, the original language too. So, selection of these rules also does not allow us to represent all possible context-free languages.

The only remaining case is rules of the type $A \to BC$, which is exactly what we have in Chomsky normal form.

What rules with 1-symbol right-hand sides can we have? The symbol in the right-hand side is either a terminal symbol or a variable. So, rules with a single symbol in the right-hand side are either of the form $V \to A$ or of the form $V \to a$.

For simplicity, it is desirable to restrict ourselves to rules of only one of these two types. Which type should we choose so that we will still be able to describe any context-free grammar in this form?

If we only allow rules of the type $V \to A$, then we will never be able to introduce any terminal symbols at all. Thus, if we restrict ourselves to this type of rules, we will never be able to generate any program at all.

The only remaining case is rules of the type $V \to a$, which is exactly what we have in Chomsky normal form. And we need such rules – otherwise, we will not be able to get any programs at all.

What rules with empty right-hand sides can we have? All these rules have the form $A \to \varepsilon$, for some variable A. There is only one fixed variable: the starting variable. So, the only way to limit these rules is:

- ether to allow these rules only for the starting variable,
- or to allow such rules only for all other variables.

In the second case, we may have many such rules, while in the first case, either one such rule or none. So the first restriction – to the starting variable A – is simpler.

This is exactly what Chomsky normal form allows. And we may need such rule – since by only using rules of the type $A \to BC$ and $V \to a$ – none of which decreases the length – we will never get an empty string, while some concepts in programming languages can be empty strings.

Conclusion. So, we explained why Chomsky normal form is used – because it is the simplest possible normal form.

Acknowledgments

This work was supported in part by the National Science Foundation grants 1623190 (A Model of Change for Preparing a New Generation for Professional Practice in Computer Science), and HRD-1834620 and HRD2034030 (CAHSI Includes).

It was also supported by the program of the development of the Scientific-Educational Mathematical Center of Volga Federal District No. 075-02-2020-1478.

References

1. M. Sipser, *Introduction to the Theory of Computation*, Cengage Learning, Boston, Massachusetts, 2012.