Using Syntax Diagrams for Teaching Programming Language Grammar

Zorica Suvajdzin Rakic*, Srđjan Popov*, Tara Petric*
* Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
tweety@uns.ac.rs , srđjanpopov@uns.ac.rs , tara.petric@gmail.com

Abstract—Understanding the syntax is an important activity in the process of learning the new programming language. One of the main factors, which influence acceptability and clarity of the syntax, is the way in which the syntax is described. There is no consensus about which meta-language is best for describing the syntax. The traditional way of learning syntax involves using Backus-Naur form (BNF), which is a bit confusing and not very popular among the students. Using syntax diagrams and tools for syntax visualization is better way for understanding and learning grammar of the programming language.

I. INTRODUCTION

Most people are taught their first programming language by example. This is probably unavoidable, since learning the language is often carried out in parallel with the more fundamental process of learning to develop algorithms. But the technique has shortcomings because the tuition is incomplete - after being shown only a limited number of examples, one is inevitably left with many unanswered questions.

A programming language is a formal language used to convey the algorithm from developers to developers and the programs to machines. To know and use the programming language one needs to understand it from three perspectives:

• Syntax – set of rules for writing the language constructions
• Semantics – determines the meaning of certain language constructions and the program as a whole
• Pragmatics – use of language constructions.

In other words, the issues of program correctness are a subject of syntax, the question of the meaning of the program is subject of semantics, while pragmatics deals with issues of expressiveness of the programming languages.

Linguistics of the natural language covers phonetics, morphology and syntax, and further, semantics and lexicography. In the area of the programming languages there is no need for phonetics and morphology, and their grammar is reduced only to the syntax [1]. In order to define or describe the syntax of a language, there is a need for a special language to describe it.

Such language is called a meta-language. Typical meta-languages are:

1. Backus-Naur form (BNF)
2. Extended Backus-Naur form (EBNF)

II. TEACHING NEW PROGRAMMING LANGUAGES

Selection of appropriate metalanguage for teaching the syntax of a new programming language is important. Introduction of the syntax in traditional Programming Languages courses starts by learning some of these meta-language, very often Backus-Naur form (BNF).

BNF notation is a meta-language developed in 1960, in the process of defining the programming language ALGOL 60. ALGOL 60 originated from the International Committee, where essential contribution was made by J. W. Backus and P. Naur. Since then, almost every author of a book on a new programming language uses BNF to specify the syntax rules of the language. The difficulty with the BNF is its readability. Understanding of the rules written in BNF can be difficult, especially for beginners. Meta-symbols of BNF are:

:: = means "is defined as"
| means "or"
<> bracket are used around non-terminal symbols.

BNF had a problem that options and repetitions could not be expressed directly. Instead, the use of meta-productions or alternative productions were needed. Alternative productions were defined as nothing, or optional derivation or any other repeated derivation.

The readability issue is partially solved by introducing EBNF notation or Extended Backus-Naur Form. This notation is based on the fact that in most real programming languages terminal symbols are listed in the quotation marks or the apostrophe.

If this is also adopted in meta-language, then there is no need to write symbol names between the brackets < > (to distinguish them). They can be written directly, without brackets. In addition, there is no need to use many-character sign ::= but simple one-character sign =.

This way, some of expressions in meta-language are significantly simplified, and the efficiency of learning programming language syntax is increased.

Meta-symbols of the EBNF notation and their meaning are:

= equals by definition
| exclusive OR
. end mark of production
X non-terminal symbol X
“X” terminal symbol X
{ X } repetition of symbol X, 0 or more times
[ X ] optional symbol X (0 or 1 occurrence)
( A | B ) grouping symbols
To avoid confusion, EBNF is not more expressive than BNF. Actually, EBNF has just introduced a few simple extensions of BNF that make grammar more expressive, while anything that can be expressed by EBNF can also be expressed by BNF, too. BNF and EBNF notation of a number is shown in the following examples (1a and 1b):

\[
S \quad ::= \quad \text{`-'} \ FN \mid FN \\
FN \quad ::= \quad DL \mid DL \ `.' \ DL \\
DL \quad ::= \quad D \mid D \ DL \\
D \quad ::= \quad '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \\
\quad \mid '6' \mid '7' \mid '8' \mid '9' \\
\]

Example 1a. BNF production of a number

\[
S \quad ::= \quad '\text{''}'? \ D^+ \ ('\text{'}.' \ D^+)? \\
D \quad ::= \quad '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \\
\quad \mid '6' \mid '7' \mid '8' \mid '9' \\
\]

Example 1b. EBNF production of a number

As it can be concluded from the previous example, EBNF is not more powerful than BNF in terms of what can be defined, but in terms of convenience. Each EBNF production can be translated into equivalent BNF production.

A. Syntax diagrams

Regardless of the simplification brought by EBNF, especially if it is the first programming language to learn, understanding complex syntax structures can be difficult. Textual form of syntax can be quite unreadable in the case of complex meta-language productions.

Therefore, it is natural to resort to write language syntax in a graphic (visual) form, because visual form can clearly show all alternatives that exist in some language construction.

Unlike BNF, this kind of notation does not seem to have a commonly agreed-on name. Syntax diagrams are also known as Railway Tracks or Railroad Diagrams or Syntax Charts or Syntax Graphs. They do not allow us to write anything that can't be written in BNF, they just make the grammar easier to understand.

Syntax diagram is a directed graph that has one entry point and one exit point, and is used for the description of valid syntax constructions of a programming language. Each syntactically valid language construction is defined as one of the paths through the syntax diagram, from input to output.

Graphical representation of the components of syntax diagram makes syntax of the programming language more readable, even more intuitive. This is shown in the following examples (example 2a shows K&R notation, 2b shows EBNF notation and Fig. 1 shows syntax diagram for C unary-expression):

\[
\text{unary-expression} : \\
\quad \text{postfix-expression} \\
\quad ++ \text{unary-expression} \\
\quad -- \text{unary-expression} \\
\quad \text{unary-operator cast-expression} \\
\quad \text{sizeof} \text{unary-expression} \\
\quad \text{sizeof} \text{ (type-name)} \\
\]

Example 2a. K&R notation for C unary-expression production

\[
\text{unary-exp} : \\
\quad \text{id} \\
\quad | \text{int_const} | \text{char_const} | \text{float_const} \\
\quad | \text{enumeration_const} \\
\quad | \text{string} \\
\quad | '(' \text{exp} ')')' \\
\quad | '[' \text{exp }']'])' \\
\quad | '(' \text{argument-exp_list? }')')' \\
\quad | '.' \text{id} \\
\quad | '->' \text{id} \\
\quad | '++' \\
\quad | '--' \\
\quad | '++' \text{unary-exp} \\
\quad | '--' \text{unary-exp} \\
\quad | \text{unary-operator cast-exp} \\
\quad | \text{sizeof} \text{unary-exp} \\
\quad | \text{sizeof } '(' \text{type-name }')')' \\
\]

Example 2b. EBNF notation for C unary-expression production

![Syntax diagram for C unary-expression](image.png)
of the grammar? What is the structure of that string relative to the grammar?

Not all strings (of tokens) are valid programs and parser must distinguish between valid and invalid programs (strings of tokens), and give error messages for the invalid ones. So, there is a need for a language for describing valid strings of tokens, and an algorithm that distinguishes valid from invalid strings of tokens.

Programming languages have recursive structure, which means that one symbol might be defined by itself. Context-free grammars are a natural notation for this recursive structure. A context-free grammar (CFG) consists of:

- a set of terminals
- a set of non-terminals
- a start symbol (one of non-terminals)
- a set of productions.

Productions have the following form:

\[ X \rightarrow Y_1 \ldots Y_n \]

where \( X \) (on the left hand-side) must be a non-terminal symbol, and every \( Y_i \) (on the right hand-side) can be either terminal or non-terminal symbol or special symbol epsilon that denotes empty string.

The language of the context-free grammar can be defined as the set of string of symbols, that may be constrained by rules that are specific to it.

B. miniC project

Authors of the Compilers course at the Faculty of Technical sciences at the University of Novi Sad, developed the miniC project as a teaching aid for the course. It contains compiler for the small programming language called miniC. This language is a subset of main language characteristics of the C programming language. To describe the language, authors have firstly used the BNF, as the usual notation for describing context-free grammars.

Understanding the recursive structure of the language was quite a challenge for the most of the students. The explanation was given at many different times, at different subjects. This misunderstanding slows down their comprehension of the parsing process. One of the things that are hard for the students to understand, is how to make a difference between two slightly different BNF rules, given in example 3a and 3b.

**Example 3a.** BNF production for function_list

\[
\text{function_list} ::= \text{function} \mid \text{function_list function}
\]

**Example 3b.** BNF production for stmt_list

\[
\text{stmt_list} ::= \mid \text{stmt_list stmt}
\]

The first rule describes the list of functions as one or more functions, and the second rule describes the list of statements as zero or more statements in the statement list. The difference shows what is the minimum number of the items in the list: zero or one.

In collaboration with authors of the Programming Language course, the definition of miniC programming language in the miniC manual [3] was extended. The syntax diagrams were added. So, each production of the miniC grammar was described by its syntax diagram and, separately in the grammar, using BNF notation. For the productions in Example 2 there were two syntax diagrams as in Example 3:

![Figure 2. Syntax diagram for miniC function_list and stmt_list](image)

When the two given productions were presented to students graphically by the syntax diagrams, the difference between these two constructions became obvious. Figure 2. shows function on the upper line, and figure 3. shows statement on the lower line with empty upper line. The empty line shows that stmt_list can follow empty line (and be an empty string). This distinction makes recursion more readable and easier to understand for students.

The result of this modification in the miniC manual, was instantly noticeable improvement in the understanding of the grammar productions and interconnections between the non-terminal symbols. The students have started asking more questions, which indicates their grater engaging in the learning process. Consequently, the understanding of the parsing process was significantly better.

IV. Conclusion

Comprehension and learning the grammar of the new programming language is a complex process. Enabling that it takes place in a controlled, interactive environment, helps a great deal. Allowing students to modify EBNF production and display it as a syntax diagram is a very good didactic tool for teaching grammars.

V. Acknowledgment

Research presented in this paper was supported by Republic of Serbia Ministry of Science and Technology grants: III–44010“Intelligent Systems for Software Product Development and Business Support based on Models”

REFERENCES

