M. T. Fisun, Dr. Sc. (Eng.), Professor; I. O. Kandyba;
G. V. Horban, Cand. Sc. (Eng.), Assistant Professor; M. V. Falenkova

USING HIERARCHY ANALYSIS TO SELECT PARSER DEVELOPMENT TOOLS WHEN CREATING DSL

The paper presents the analysis of software tools for generating parsers, based on the Python language. Two main categories are distinguished: general-purpose languages not tied to a particular programming language and tools, developed exclusively for the general-purpose Python language. Detailed analysis of a number of the most common parser generation tools was performed. Based on the ISO/IEC 2510 standard, main characteristics of parser generators were identified. The selected characteristics include performance, context coverage, satisfaction, functionality, transferability, and usability. A hierarchy of criteria is constructed to implement the method of hierarchy analysis, which includes both quality models of ISO/IEC 2510 standard and a set of criteria, described by these models. Expert assessments were used to calculate the vector of local priorities, which formed the basis for calculating the vector of global priorities. In the vector of global priorities, the alternative with the highest score was determined. The selected alternative indicates the most effective tool for generating parsers.

Key words: *relational algebra, Python, DSL, Pyparsing, Parglare, PLY, ANTLR, Unicc.*

Introduction

Domain-specific languages (DSLs) is a class of programming languages, used exclusively in specific domain areas (industries) [1]. This class includes data manipulation languages [2, 3], knowledge representation languages [4, 5], etc.

DSLs are divided into two main types: external and internal ones. Internal DSLs are based on a call chain of method and do not require additional components but have a limited scope. External DSLs are implemented in separate programming languages and have no limitations of the previous type but require lexical and syntactic parsers. Means of automatic generation of lexical and syntactic parsers have different possibilities [6, 7].

In the works, reflecting the studies of the parser's performance quality, the speed indicators of processing the input language [7] or the peculiarities of constructing rules and tools for generating parsers [6, 8, and 9] form the basis. However, the creation of domain-specific language parsers has several peculiarities: support for Cyrillic and adding code in a general-purpose language. In order to select the best alternative among generation tools of syntactic parsers, it is necessary to take into account all listed characteristics and peculiarities. Analysis of the sources shows the lack of a comprehensive approach to solving such a problem, so the topic of the research seems relevant.

Relevance

In recent years, there was a growing interest in domain-specific languages [1]. Most often this class of programming languages is called DSLs in English-language sources. The popularity of DSL is due to two factors:

Increased productivity of software developers due to the smaller code amount in DSL than in general-purpose languages;

Improved interaction with subject matter experts due to the peculiarities of the syntax that consists of the domain area terms.

The best-known examples are the language of relational algebra (RA) [2], the language of data manipulation in the Cypher graph database [3], the language for describing OWL ontologies [4]. Separately, it is worth highlighting modeling languages, most often presented exactly as DSLs [5].

One of the difficult tasks in DSL development is to implement input language syntax handlers (parsers). The very process of DSL creation includes several different stages: formation and formal description of the input language structure, development of the lexical analyzer, development of the syntax parser, generation of the output code [6].

Objective

Syntax parser generators have a wide range of characteristics [6], which influence the results of their work, which, in turn, indicates the need to define a set of criteria for selecting the most efficient tool for code generation. Selecting the most efficient method for parser generation is a difficult task that requires the solution of a multi-objective decision-making problem. This paper aims to create a criteria hierarchy and choose the most efficient tool for parser generation. This research concerns both universal generators not tied up to a specific programming language and tools designed exclusively for one programming language.

Problem solving

In order to reduce the programming time of parsers, specialized software tools – parser generators are most often used. These tools are capable to automatically generate part of the parser code (in some cases completely), which not only reduces the time required for software development but also prevents errors during code writing.

An important element of language parser generators is the description of syntax rules of the input language. Creating rules for the lexer in most cases is done using regular expressions, but developing a parser is a more complex problem that requires programming an algorithm for parsing the input set of terminal symbols and tokens.

Before starting to develop parsers, the syntax of the input language should be formally described. Description of the input language structure is usually carried out with the help of Backus-Naur forms (BNF) [6] or their modifications. It is a formal tool for describing the input language when syntactic entities are described using other ones. For example, to implement a programming language for manipulating RA data, it is possible to use the following rules in the BNF:

<union>::=<attribute_name>**UNION**<attribute_name>**;**

<attribute_name>::=<braket> | <table_name>

<table_name>::=<symbol>{<symbol>}|<symbol>{<symbol><number>}

<symbol>::=**a**|**b**|…|**z**

<number>::=**0**|**1**|**2**|…|**9**

<braket>::=**(**<braket_expr>**)**

<braket_expr>::=<minus>|<union>|<join>|<projection>|<extend>|<semiminus>|<semijoin>|<divideby>|<braket>

After a formal description of the input language, it is necessary to start developing the program code of the parsers. At this stage, it is necessary to choose a tool to generate parsers. Most tools of this type are tied to a specific programming language, such as JLanguageTool, JavaCC, PLY, PyParsing, and so on, but there are also universal ones such as ANTLR, Unicc, and others.

According to the "ain.ua" portal, the Python programming language holds one of the leading positions in job ratings in the labor market. Python also supports several different programming paradigms and many additional tools, such as libraries for different DBMS, web frameworks (Flask, Django, and others), libraries for working with hardware, and so on.

Parser generators can be divided into two types: universal – separate tools that allow generating parsers for different programming languages and embedded in a specific programming language. Based on this, the following parser generators exclusively for Python can be distinguished:

a) **Pyparsing**. It is alibrary for generating lexical and syntactic parsers based on specialized rules. The peculiarity of this tool is in the way of rule building, and it requires the use of specialized syntax [7]. Pyparsing uses built-in methods and variables to describe the syntax (Table 1).

Table 1

Basic Pyparsing functions

Method/Variable	Value
DEFAULT_KEYWORD_CHARS	A set of symbols that are not separators: '$_ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'.
Literal()	Search for an exact match with the string.
Keyword()	Similar in principle to Literal(), but a given string must be separated by a symbol not included in the variable DEFAULT_KEYWORD_CHARS.
CaselessKeyword()	Similar to Keyword(), but case-insensitive.
Word()	Used to create a string pattern consisting of characters included in the variable DEFAULT_KEYWORD_CHARS.
CharsNotIn()	Symbols that should not be included in the string.
Regex()	A comparison with a regular expression pattern.
QuotedString()	Defining available separators: disable or enable the use of spaces, tabs, NL, and so on.
SkipTo()	The method allows you to skip mismatched fragments.
White()	Similar to Word(), but includes whitespaces.
Empty()	Checking for the presence of an empty row.
NoMatch()	Opposite of Empty()
Alphas	A string containing only letters.
Nums	A string containing only numbers.
Alphanums	A string containing only letters and numbers.
Printables	A string consisting of any characters that can be printed except for a space.
And() or +	The logical "and.
Or() or ^	The logical "or".
ZeroOrMore()	Zero or more.
OneOrMore()	One or more.
Optional()	String search, may be present or absent

The methods and variables presented in Table 1 allow for a description of the input language for further analysis, but besides those in Table 1, the library contains additional methods designed for specific processing, such as htmlComment() – highlighting comments in the markup HTML (' <!-- ' and ' --> '), cStyleComment() – highlighting comments in the markup CSS (' /* ' and ' */ '). Tools are also available to limit the length of the input string: min – setting the minimum length, max – maximum length, exact – exact string length, and so on..

The syntax of the RA language parser [2], created with Pyparsing, will look as follows:

```
TableName=Word(alphas)
UnionOper=TableName+Keyword('UNION')+TableName
MinusOper=TableName+Keyword('Minus')+TableName
JoinOper=TableName+Keyword('Join')+TableName
SemiJoinOper=TableName+Keyword('SemiJoin')+TableName
SemiMinusOper=TableName+Keyword('SemiMinus')+TableName
Prjection=TableName+Word('[')+TableName+Optional(','+TableName)+Word(']')
OperProcesing=TableName^UnionOper^MinusOper^JoinOper^SemiJoinOper^SemiMinusOper^Prjection
s = 'StudentUnionTradeUnion'
try:
res = OperProcesing.parseString(s)
print(res.asList())
exceptParseException:
```

print('Processingerror of ParseExceptionoperator');

The rules, described above allow for parsing a certain operator and for catching an exception in case of an error. Pyparsing makes it possible to use operator precedence and infix Notation functions to process expressions. Operator precedence divides an expression into separate parts using parentheses as a separator, returns this pyparsingtool forward to an object, which in turn contains an iterator implementation for sequential processing of expressions in parentheses.

The main algorithm for parsing the input string with Pyparsing is the Recursive descent parser method. The essence of this algorithm is the mutual invocation of functions that implement rule checking; the implementation of grammatical rules takes place using variables. The first function is run as follows: OperProcesing.parseString(s), where OperProcesing is the name of the variable containing rule description and s is the input string containing code for parsing. It's worth mentioning that this tool doesn't support the ability to work in Cyrillic.

b) **Parglare.** It is a library for the Python language, which includes an implementation of the GLR algorithm (Generalized Left-to-right Right most derivation parser, extended version of the LR method) [8]. It is intended for parsing non deterministic and ambiguous grammars.

Parglare, when creating parsers, requires the implementation of several components: Grammar block that describes lexemes (nested block) and syntax rules; the block can be represented as a separate file or as a line in the middle of Python code.The next component to be implemented is the actions block, which includes a description of methods or lambda expressions to process the operators defined in the previous block. In this case, the number of actions described in the actions block must match the number of operator writing options in the Grammar block.An object of Parser class is also a necessary part of Parglare, which contains Grammar and actions variables in its constructor. The analysis starts by calling the Parser.parse() method, where the string to be parsed is passed.Syntax of rule description for this parser generation tool is similar to BNF but does not forbid left-hand recursion due to the GLR algorithm. There is no support for Cyrillic, which leads to error output when trying to describe an operator using Cyrillic.

A built-in error handler implemented in the parglare.exceptions.ParseError class is supported. In case of error detection in the code, the user will see a message with options to correct the error, for example, when the union operator makes a misspelling, the following message will be displayed:parglare.exceptions.ParseError: Errorat 1:16:"nt unions **> (TradeUnio" =>Expected:) or STOP or join or minus or semijoin or semi minus or union but found.

The parser grammar for the operators [2] of RA language will look as follows:

```
grammar = r"""
E: E 'minus' E  | E 'join' E | E 'union' E  | E 'semijoin' E | E 'semiminus' E | '(' E ')'|
IDs;
terminals
IDs: /[A-Za-z0-9]+/;
"""
actions = {
   "E": [lambda _, n:  n[0] +" minus "+ n[2],
lambda _, n:  n[0] +" join "+ n[2],
lambda _, n:  n[0] +" union "+ n[2],
lambda _, n:  n[0] +" semijoin "+ n[2],
lambda _, n:  n[0] +" semiminus "+ n[2],
lambda _, n:  n[1],
lambda _, n:  n[0] ],}
g = Grammar.from_string(grammar)
```

```
parser = Parser(g, actions=actions)
result = parser.parse("(Studentunion (TradeUnionminusTable)) minus (TableminusTable)")
```

c) PLY(PythonLex-Yacc)[9]. It isa library for the Python programming language that implements LEX and YACC tools. Lex is a lexical parser generator built into the UNIX system; this tool is a part of the POSIX.Yacc standard and serves to generate syntactic parsers, which is the next step of program code processing

The main idea of PLY is to simplify the creation of parsers, based on a single set of rules. Lex generates a set of rules similar in syntax to the BNF representation form but set using specialized entities. The Lex tool implementation happens using the importply.lex command.

The task of Lex is to decompose the input string into tokens. Working with ply.lex requires creating a specialized array of tokens, named for tokens and further describing the token as a variable starts with "t":

```
tokens = ('MINUS', ' UNION ', 'JOIN', 'SEMIMINUS ', 'SEMIJOIN ','ident','open','close')
t_UNION = r'union'
t_ident=r'\w+'
t_open=r'\('
t_close=r'\)'
```

The second part of the library is ply.yacc, which serves to implement parsing. The functions of this tool require the connection of this library: importply.yacc.

The creation of ply.yacc rules is done by combining the tokens described in ply.lex. Each implemented language expression must be described with methods that start with "p_". For example, the syntax of the description of the RA UNION operator is:

```
defp_expr(p):
 '"expr : openclose| openexprclose | exprUNIONident | exprUNIONexpr| identUNIONexpr'"
print(p[2])
def p_UNION(p):
   '"union : identUNIONident'"
   p[0]+=" "+p[1:]+" "+p[2]+" "+p[3]
```

The result of one of the parsing methods is a string that is stored in the variable "p [0]", where "p" is an object of class ply.yacc–YaccProduction (the name of variable "p" may be changed by the developer). The variable "p" contains the input string, decomposed by separators, as well as the result of the parsing method that was called.

Like the previous parser generation tool, PLY does not support the ability to operate with Cyrillic. Errors are handled by the "p_error" method.

PLY implements the LALR (1) (Look ahead Left to Right) algorithm [6] based on the construction of the parsing table, and the generated parsing table is saved in a separate file "parser.out."

Wide spread universal generators that support the ability to generate for Python are:

d) ANTLR(ANother Tool for Language Recognition) [10].It is a universal parser generator developed in the java programming language.This software product is implemented by separate applications that receive as input a set of specialized rules in a *.4g file, and generate as output a set of classes for lexical and syntax parsing.The command to generate parsers for ANTRL is as follows: **antlr4 -Dlanguage=Python3 "F:\real\REAL_Lang.g4"**,where the **Dlanguage**key specifies the programming language, for which the parsers are generated, and **"F:\real\REAL_Lang.g4"** is a file containing the description of grammars of the input language.

After generation, files appear in the directory with the grammar description file:REAL_LangLexer.py ,REAL_LangListener.py, REAL_LangParser.py, REAL_Lang.tokens, REAL_LangLexer.tokens, whereREAL_Langis the name ofthe inputlanguage.

Each generated file performs a different role:

a) REAL_LangLexer.py contains lexical parsing rules;

b) REAL_LangParser.py contains the parsing rules;

c) REAL_LangListener.py is an additional class that contains a description of the methods that are executed during traversing the syntax tree (in the target language);

d) REAL_Lang.tokens, REAL_LangLexer.tokens are auxiliary files with the description of tokens.

In the middle of the described files, in REAL_LangParser.py, there are generated classes for each grammar rule; these classes contain the rule processing mechanism and are inherited from ParserRuleContext.

A file of type * .g4 contains a description of the rules in a specialized syntax similar to the BNF:

```
parseoperator: minus_stmt
|where_stm|projection_stmt|union_stmt|join_stm|matching_stm|notmatching_stm;
args_stm:table_name|projection_stmt|bracket;
minus_stmt: args_stm MINUS args_stm;
table_name:IDENTIFIER;
IDENTIFIER: [a-zA-Z_] [a-zA-Z_0-9]*;
```

ANTLR generates lexical and syntax parsers based on the same set of rules written within the *.g4 file. However, for the correct work of the generated code, it is necessary to load the antlr4-Python3-runtime package (pipinstall antlr4-Python3-runtime).

Unlike previous tools, ANTLR supports the ability to work with Cyrillic based on the use of specialized unicode codes. According to the official ANTLR documentation, Cyrillic characters are in the range 0400-04FF.In order to handle syntactic and semantic errors, a class is used that inherits from ANTLRErrorListener, which in turn is a part of ANTLR. The method of parsing is LL (*) (left to right leftmost derivation).

e) **Unicc** is also a universal parser generator. This tool is written in C and supports the ability to work with different operating systems. The generated parser works using the LALR(1) method.

A description of the input language grammar for this tool is in the ".pam" file. Input language grammars are described using syntax similar to the BNF.The grammar description file also contains the "!Language" parameter that specifies the programming language for which the parsers will be generated. This and other parameters start with symbol #; it is obligatory to denote tokens for lexical parsing:

```
#!languagePython;
#lexemetabltableListproj;
RelExpr$ ->expression;
expression: expressionoperexpression  | '(' expression ')'| proj| tabl ;
oper -> "minus" | "join" | "union" | "semiminus" | "semijoin";
tabl -> 'A-Za-z_' | tabl 'A-Za-z_';
tableList ->tabl| tableList ',' tabl;
proj ->tabl '[' tableList ']';
```

The ParseException object handles errors.This object is inherited from Exception and except the error text, it prints possible variants of the operator's spelling. By using the LALR(1) method, the grammar rules allow recursion.

Unicc also generates an output file with the name of the input file, but with the type corresponding to the programming language (*.py).In the middle of the generated file there is a

mechanism for processing the described grammars. Figure 1 shows this mechanism in the form of several individual classes as a diagram of classes.

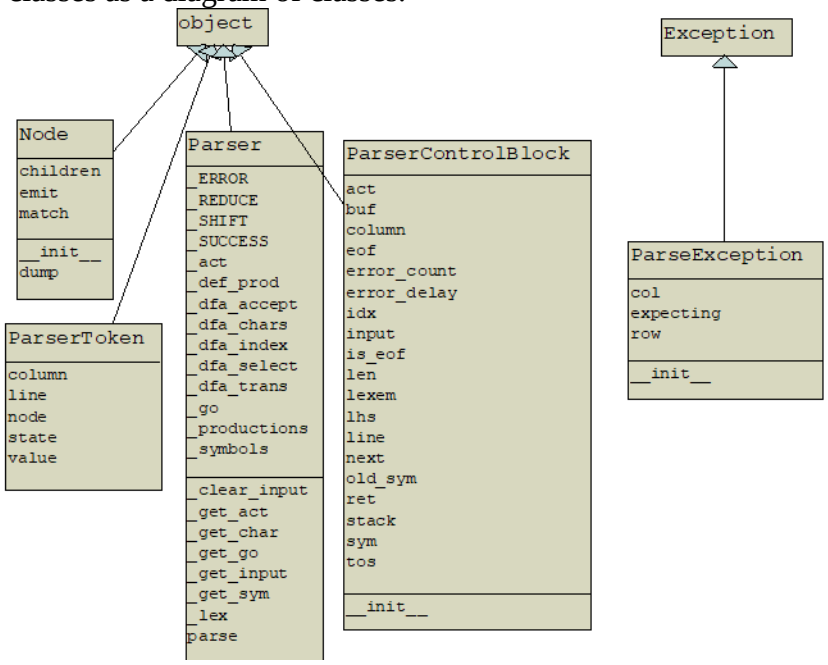

Fig. 1. Diagram of parser classes generated by Unicc

The Unicc error handler is generated by a separate class, inherited from Exception, whose object can be created if necessary.

It is possible to choose the optimal parser generation tool based on the quality models described in the ISO / IEC 2510 standard. This standard offers two models of quality determination: quality in use and product quality. From the proposed quality models the following classes of criteria can be chosen, which are the most important for the parser generator: performance, context coverage, satisfaction, transferability, functionality, usability.

Each presented class of quality characteristics includes subclasses, which, in turn, can be expressed in software characteristics. Parser generators have the following characteristics belonging to the above classes.

Performance of generated parsers is the speed of processing the incoming string.

Context coverage because when analyzing generation tools, the subclass of this class is important for the use of "context completeness" characteristics. A representative of this subclass in parser generators is Cyrillic support, which is important for creating DSL with operator support in Ukrainian.

Satisfaction includes the "comfort" subclass, which reflects the user's satisfaction with physical comfort. In DSL operation, an important factor of physical comfort is ways of finding an error in the entered code or self-descriptiveness of the error message text, variations of correction, and the position of the error in the input string. Based on the above, the built-in error handler of the input programming language can be considered a representative of this subclass.

Functionality reflects the degree to which the software performs functions according to the user's needs.This class contains the "functional usability" subclass, which shows the degree of functional simplicity of performing certain tasks and achieving objectives. In the context of the study of parsers generators, the method of parsing should be included here. It affects many factors: the possibility to use recursion in input rules, speed of processing incoming string, and so on.

Transferability reflects the ease of transferring developed grammars to other versions or other parser generators. The ease of transferring grammars directly depends on the way of describing the syntax of the input language.

Usability contains the "manageability" subclass, which reflects the presence of attributes in the system for simple control and monitoring.Parser generators use general-purpose programming language code to implement control, so, this feature subclass includes the dependency on the way the general-purpose language code is added to the input language's grammar.

At the initial development stages, choosing the most optimal parser generator is a difficult task. It is possible to solve this problem by using the analytic hierarchy process (AHP) [11].

The AHP requires a well-defined goal, criteria, and alternatives. In this case, the goal is to choose the optimal parser generation tool for the domain-specific programming language and criteria. Two software quality models, described in the ISO / IEC 2510 standard, perform the role of criteria: quality in use and product quality. Each model has a set of classes and subclasses of software quality characteristics. The parsers' quality characteristics described above can be represented as a hierarchy (Figure 2).

Alternatives are the parser generation tools discussed above: Parglare, ANTLR, PLY, Pyparsing, Unicc.

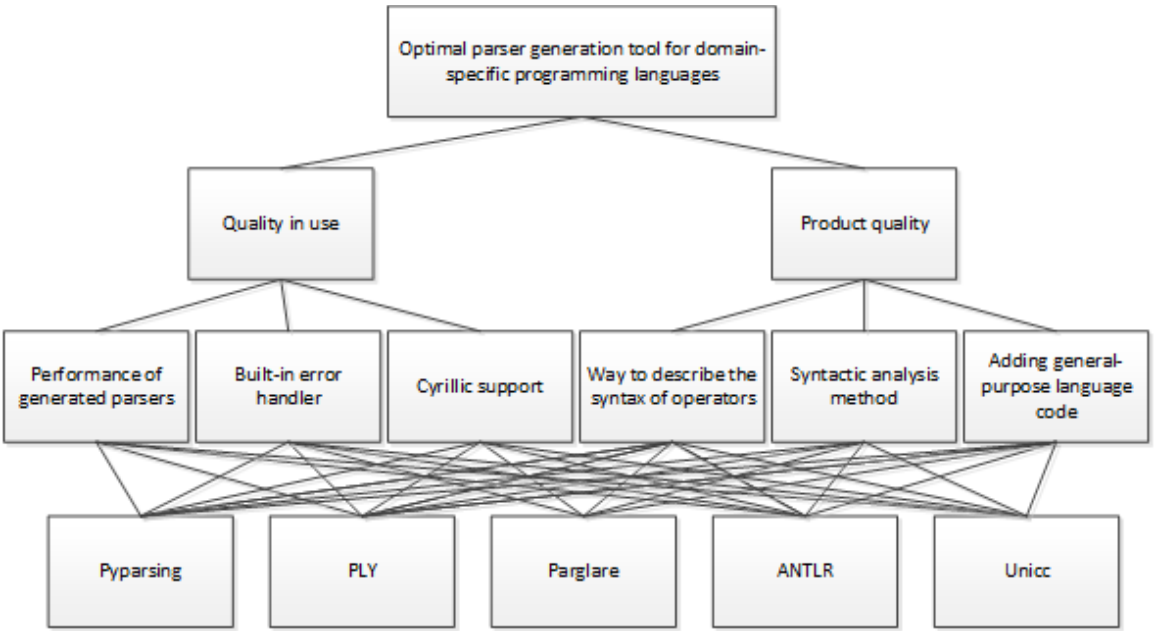

Fig. 2. Displaying a hierarchical decision-making model for selecting the optimal parser generator for DSL

The performance of the generated parsers is an important part of the evaluation of the considered tools. The hardware used to test the parsers was a Mobile Dual-Core Intel Pentium 2020M, 2400 MHz with 4 GB of RAM. Several relational algebra expressions were used as the text language: Student [IdStudent], Student union coworkers, Student union (Trade Union minus coworkers), (Student union (Trade Union minus coworkers)) minus (Student minus Trade Union). During the execution of each operator, the processing time of the input query was measured and the average execution time of the query was calculated (Table 2).

Table 2

Processing time of the input string generated by parsers

	Student [IdStudent]	Student union coworkers	Student union (TradeUnionminus coworkers)	(Studentunion (TradeUnionminuscoworkers)) minus (StudentminusTradeUnion)	Average reaction time
ANTLR	0.013	0.017	0.044	0.145	0.055
Unicc	0.009	0.009	0.02	0.025	0.016
PLY	0.05	0.116	0.068	0.088	0.081
Parglare	0.15	0.157	0.161	0.158	0.157
Pyparsing	0.028	0.028	0.033	0.1	0.047

The first step in the AHP implementation to select the parser generator is to construct a matrix of pair-wise comparisons of alternatives (A) for each criterion (K). The matrix contains comparative assessments of experts (m) for existing alternatives, and the matrix is always square of size n x n (1). A similar matrix of pairwise comparisons is constructed also for criteria.

$$\begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ \cdots \\ A_i \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_3 \ldots & A_j \end{array} \\ \begin{vmatrix} m_{11} & m_{12} & m_{13} \ldots & m_{1j} \\ m_{21} & m_{22} & m_{23} \ldots & m_{2j} \\ m_{31} & m_{32} & m_{33} \ldots & m_{3j} \\ \multicolumn{4}{c}{\cdots} \\ m_{i1} & m_{i2} & m_{i3} \ldots & m_{ij} \end{vmatrix} \end{array} \quad (1)$$

After scoring, it is necessary to find the vector of local matrix priorities, based on the method of geometric mean, according to expression (2).

$$w_i = \sqrt[n]{\prod_{j=1}^{n} m_{ij}} \quad (2)$$

The resulting vector must be normalized. When implementing the AHP, the formula (3) is usually used.

$$v_i = \frac{w_i}{\sum_{i=1}^{n} w_i} \quad (3)$$

Table 3

Local priorities of criteria

Parser generators	Criteria					
	Adding code in a general-purpose language	Built-in error handler	The method of syntactic analysis	Cyrillic support	Operator syntax description method	Performance of the generated parsers
	Vectors of local priorities					
Parglare	0.069	0.091	0.433	0.059	0.298	0.053
ANTLR	0.545	0.232	0.097	0.261	0.298	0.151
PLY	0.253	0.162	0.184	0.114	0.068	0.102
Pyparsing	0.073	0.075	0.074	0.054	0.036	0.176
Unicc	0.0599	0.439	0.211	0.513	0.298	0.517

The AHP requires performing actions (2) and (3) for each level of the hierarchy, which makes it possible to reflect local priorities (μ). Starting from the second level of the hierarchy, formula (4) must be applied to assess (λ) the alternative.

$$\lambda = v_{i1}\mu_1 + v_{i2}\mu_2 + v_{i3}\mu_3 + \cdots + v_{ij}\mu_j \quad (4)$$

The selection of an optimal parser generation tool depends on two criteria: the functionality of generated parsers, which reflects some characteristics of input programming language, and the functionality of the parsers generator, which reflects its features. Table 3 reflects local criteria priorities and global ones (λ).

Table 4

Global priorities for the criteria

	Generated parsers functionality	Generator functionality	λ
μ	0.667	0.333	
Parglare	0.062	0.251	0.125
ANTLR	0.234	0.281	0.250
PLY	0.117	0.152	0.129
Pyparsing	0.082	0.055	0.073
Unicc	0.504	0.178	0.396

The result of the AHP use is a matrix containing a set of global priorities. Based on the resulting matrix, it is possible to determine the alternative that is the most optimal. A hierarchy with five alternatives and six criteria has been formed to select the best tool for generating parsers of domain-specific programming languages; according to the results of the AHP use, the most optimal alternative is Unicc.

Conclusions and prospects for further research

The analysis of modern tools for generating syntactic parsers when creating domain-specific programming languages (DSLs) has been carried out.Peculiarities of tools of this type for Python programming language have been studied.The AHP method has been applied to choose the most optimal tool for generating parsers. A hierarchy using software quality criteria from the main ISO / IEC 2510 software quality models has been constructed. The advantages of the Unicc tool for DSLs have been demonstrated: Cyrillic support, ability to add general-purpose code, speed of generated parsers, and way to describe the syntax of input operators.

It is planned to use the most optimal Unicc generator for generating the syntax of languages for modeling graph structures, in particular the topological structure of a heating network. The language is built as a superstructure over the universal graph DBMS Neo4j using in the syntax the specific terms of this domain area.Thus, it should include operators to manipulate the topological structure of the heat network described in Cyrillic, which will allow specialists in the domain area to work with it.Similarly, DSL can be built for other domain areas that have a network structure, so the scope of DSL is broader than a purely subject orientation.They can also be problem-oriented, like checking and highlighting syntax, navigating code to determine fragments to build unit tests, tree feature tracking, and the like.

REFERENCES

1. Fowler M. Domain specific programming languages / Martin Fowler. – Kyiv: Dialectics-Williams, 2011. – 576 p. (Rus).

2. Kandyba I. Using the ANTLR software system for the creation of relational algebra languages / I. Kandyba, M. Fisun // The 11th International Scientific and Practical Conference ION-2018: Proceedings of the conference, 22 – 25 may 2018. – Vinnitsa : VNTU, 2018. – 343 p. (Ukr).

3. Eifrem E. Data bases / E. Eifrem, J. Webber, J. Robinson. – Moscow DMK Press, 2016. – 256 p. (Rus).

4. Glibovets A. M. Intelligent Networks / A. M. Glibovets, M. M. Glibovets, M. V. Polyakov. – Dnepropetrovsk : National University "Kyiv-Mohyla Academy, 2014. – 462 p. (Ukr).

5. ANTLR as a Development Platform for the Series DSL for the Learning Process : (2019, 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems) [Electronic resource] // I. Kandyba, Y. Davydenko, V. Panasyuk, A. Shved, M. Fisun // 2019, 10th IEEE International Conference on

Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS). – 2019. – Access mode : https://ieeexplore.ieee.org/document/8924354.

6. Compilers: principles, technologies and tools / [Aho A. V., Lam M. S., Sethi R., Ulman D. D.]. – Kiev : Dialectics, 2019. – 1184 p. (Rus).

7. Dejanović I. Arpeggio: A flexible PEG parser for Python / I. Dejanović, G. Milosavljević, R. Vaderna // Knowledge-Based Syst. –2016. – Vol. 95. – P. 71 – 74.

8. Parglare [Electronic resource] / Official web-portal of the Parglare tool. – Access mode : www.igordejanovic.net/parglare/stable.

9. Behrens S. Prototyping Interpreters Using Python Lex-Yacc. / S. Behrens // Dr Dobb's Journal-Software Tools for the Professional Programme : CMP Technology – 2004. – Vol. 95, №3. – P. 30 – 35.

10. Parr T. The Definitive ANTLR 4 Reference / T. Parr. – United States, Texas : Pragmatic Bookshelf, 2014. – 322 p.

11. Methods of system analysis in the tasks of marine clusters: monograph / [I. I. Kovalenko, S. K. Chernov, A. V. Shved et al.]. – Kharkov : Novoe slovo, 2017. – 268 p. (Rus).

Editorial office received the paper 15.03.2021.
The paper was reviewed 23.03.2021.

Fisun Mykolai – Doctor of Science (Engineering), Professor with the Software Engineering Department.

Horban Glib – Candidate of Technical Sciences, Associate Professor with the Software Engineering Department.

Kandyba Igor – Lecturer with the Software Engineering Department.

Falenkova Marina – Lecturer with the Software Engineering Department.

Petro Mohyla Black Sea State University.