Sopronyuk T.M., Sopronyuk A.Yu., Drobot A.V.

PHASES OF CONSTRUCTING A LANGUAGE PROCESSOR FOR THE .NET PLATFORM

The article presents a comprehensive approach to programming language development for the .NET platform. The authors explore the stages of language creation, starting from defining its goals and objectives, designing syntax and semantics, to implementing a language processor with CIL code generation for the .NET virtual machine. The article employs a class hierarchy for operations on regular languages and provides a specific specification for the Vlan language. The research findings underscore the importance of a systematic approach to programming language development and their adaptation to specific tasks and user requirements in the .NET environment.

Key words and phrases: .NET, ANTLR, language processor, formal language theory, formal grammar, finite-state machine.

Yuriy Fedkovych Chernivtsi National University, Chernivtsi, Ukraine
e-mail: *t.sopronyuk@chnu.edu.ua, soproniuk.andrii@chnu.edu.ua, drobot.andrii@chnu.edu.ua*

Introduction

In the context of rapid technological advancements and the constant increase in the complexity of programming tasks, there arises a need to improve programming languages and language processors. This challenge is particularly noticeable in specialized industries, where there is often a demand for the development of custom domain-specific languages. Creating efficient and high-level languages optimized for specific tasks is a crucial stage in the evolution of software.

This article aims to address the problem of developing a language processor for a high-level programming language on the .NET platform. Despite the existence of established programming languages and their language processors, challenges persist, stemming from the necessity to create specialized languages for particular domains and to optimize the efficiency and productivity of software solutions.

Analysis of recent research and publications in this field indicates the need for new methods and tools in the development of programming languages and language processors. Identifying unresolved aspects and gaps in the scientific approach to creating such tools is pivotal for further research.

УДК 004+519.6
2010 *Mathematics Subject Classification:* 6804, 6806.

© Sopronyuk T.M., Sopronyuk A.Yu., Drobot A.V., 2023

1 Work Objective

The main objective of this article is to examine the phases of constructing a language processor for a specific programming language based on the .NET platform, exploring and refining existing methods and tools for the development of languages and language processors.

In this work, tools for programming language development are presented, specifically showcasing the creation of the custom .NET programming language Vlan [1, 2, 3, 4].

First and foremost, it is essential to define the goals and purpose of the new language. This could involve developing a tool for a particular field, such as financial analytics or machine learning, or simplifying the interface for beginners. In our case, it involves working with vectors.

The next step is designing the syntax and semantics of the language. This stage includes creating the language grammar, defining syntactic rules, and specifying data types and operations that can be performed with them. Subsequently, it is necessary to implement a compiler or interpreter for the new language, which will transform the code in the new language into machine code or execute it.

It is worth noting that some languages, such as C# and VB.NET, are compiled into an intermediate representation called Common Intermediate Language (CIL) and then executed by the Common Language Runtime (CLR) runtime environment. This approach is precisely what we have implemented for our custom language processor.

Therefore, the aim of this work is to encompass all phases of constructing a language processor [5] for the high-level programming language Vlan [1] and generate the corresponding CIL code for the .NET virtual machine.

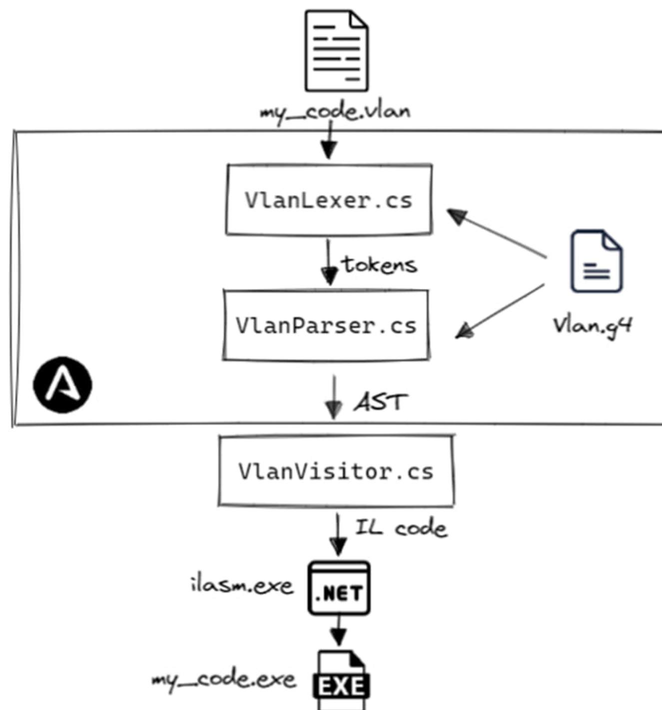

Figure 1: Vlan Language Processor Software Model

The project is divided into separate phases using different approaches, and each stage is visualized. The GraphViz library is employed for visualization. The syntax analysis phase utilizes the Antlr parser generator. For the development of the user interface of the program, the interaction protocol between source code editors and language analysis servers, known as the Language Server Protocol (LSP), is used.

2 The lexical analysis phase

The first phase in the development of any recognizer is the lexical analysis phase. Regular languages are used to describe lexemes. The application of the closure property of regular languages allows for the construction of recognizers for more complex languages, using simpler recognizers and operations. Therefore, the paper extensively explores the construction of complex recognizers using global regular expressions, where operands are nondeterministic finite automata, right-linear grammars, or regular expressions [5, 6, 7].

To describe more complex structures, a global regular expression is used, where the operands are formalisms of automata languages [7]. The paper defines the rules for writing this expression using LA(1)-grammars [5, 7]. The computation of the global regular expression includes unary and binary regular operations on the formalisms of regular languages.

```
#ifndef LanguageH
#define LanguageH

class Language
{
protected:
    string alphabet;
public:
    Language(string s);
    virtual ~Language() {};
    virtual void operator*() = 0;
    virtual void operator+() = 0;
    virtual void print() = 0;
    virtual void copy(Language*) = 0;
    virtual int is_empty() = 0;
    virtual int check() { return 1; };
};

#endif
```

The implementation of these operations varies for different formalisms, but the invocation interface remains consistent. In the paper, a class hierarchy has been created with an abstract class "Language", which describes the alphabet of a regular language and defines unary operations "iteration*" and "iteration+" [5].

The parent class "Language" code includes the definition of pure virtual unary operators "iteration*" and "iteration+", a constructor, a virtual destructor, and necessary member functions. This class is inherited by the classes "Automata" (non-deterministic finite automata), "RegularExpression" (regular expression), and "Grammar" (right-linear grammar).

All these classes encapsulate data members for representing the form of a regular language, implement overloading of unary operators "iteration*" and "iteration+", as well as binary operators "alternative|" and "concatenation+" [6]. Figure 2 and Figure 3 depict abbreviated and complete UML class diagrams.

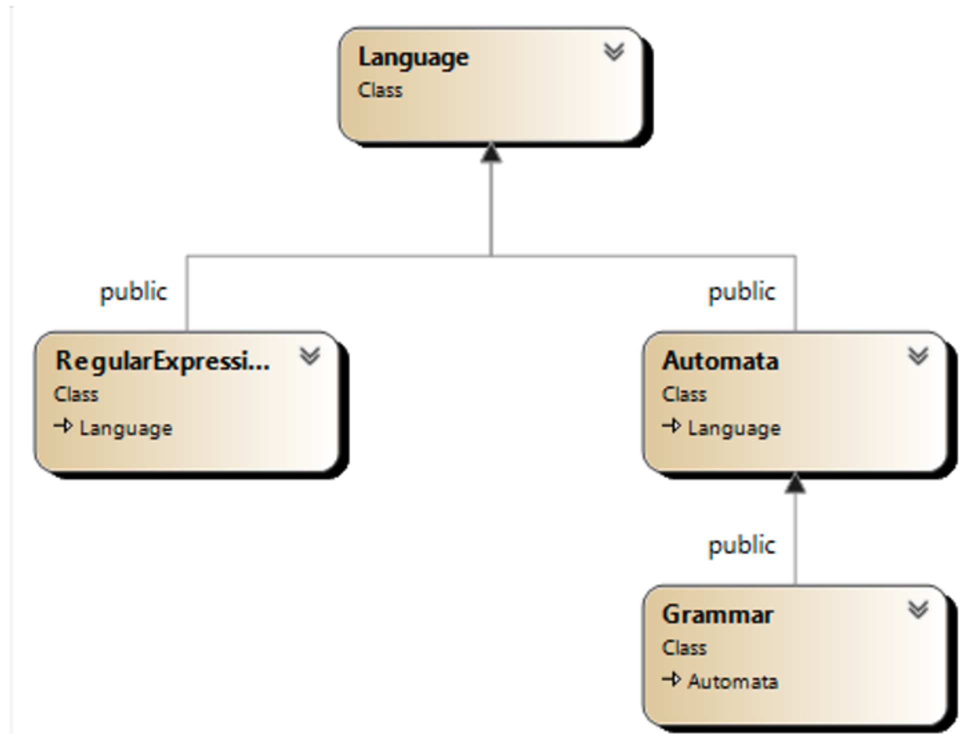

Figure 2: Classes Hierarchy

Therefore, as we can see, the developed class hierarchy allows the construction of complex formalisms, such as transition tables of an automaton, and enables their use in implementing recognition or transformation tasks for strings of a specified automaton structure in various software applications.

Let's demonstrate the lexical analysis phase with the construction of a deterministic finite automaton for recognizing lexical types such as identifiers, numerical constants, assignment operations, addition and multiplication operations, left and right parentheses. The result of such analysis is visualized (Figure 4) through the transition diagram of the deterministic finite automaton with different types of final states [4].

Below (Figure 5) is an explanation of the operation of this automaton, namely: the groups of symbols assigned to the edges of the automaton and the final states where a specific type of lexeme is recognized.

To proceed to the syntactic analysis phase, let's briefly describe the specification of our own programming language, Vlan [3].

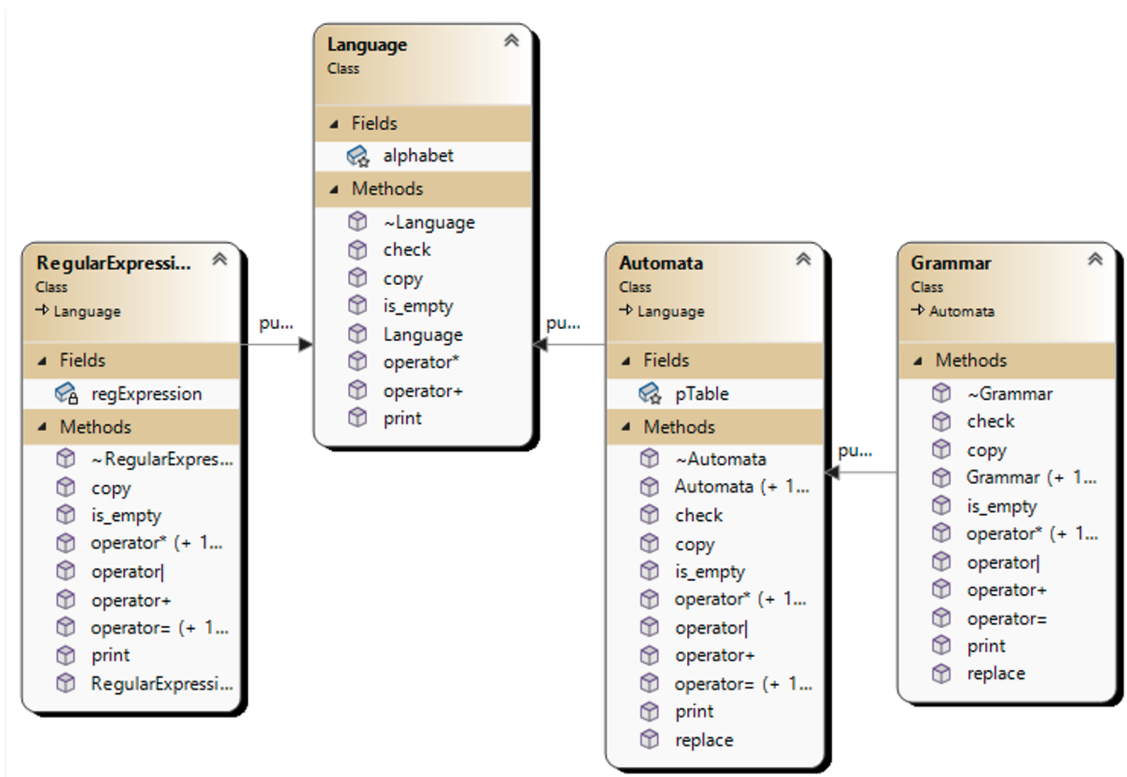

Figure 3: Classes UML diagrams

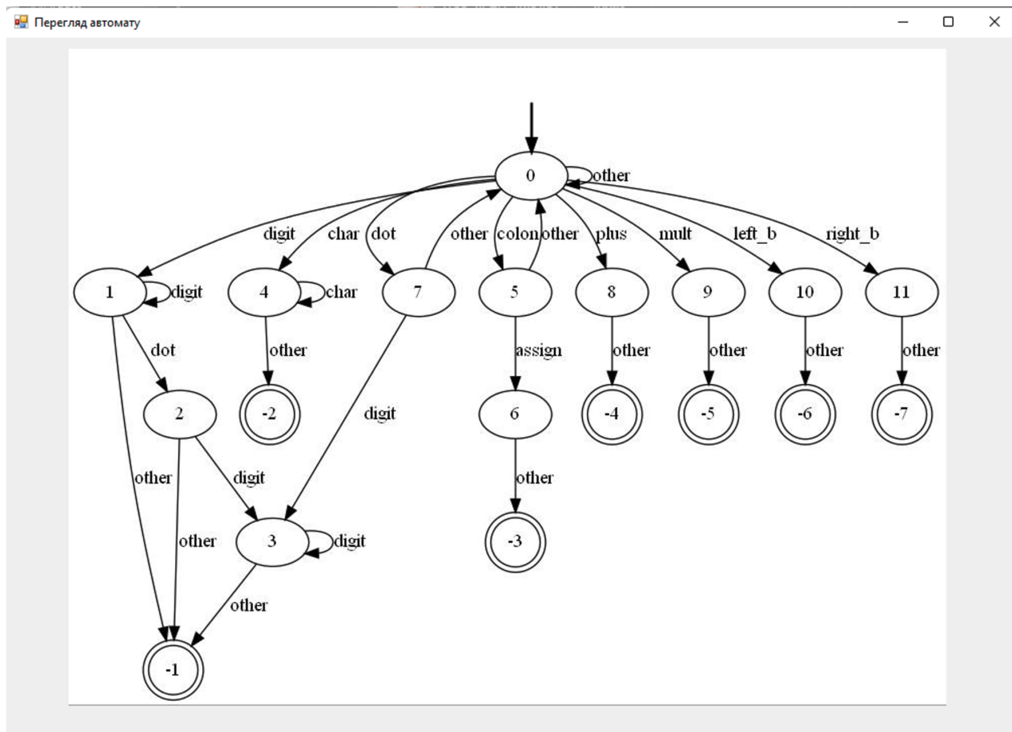

Figure 4: Transition diagram of a deterministic finite automaton with various types of final states.

Automaton details

Symbol groups

Group name	Symbols
CHAR	A-Z a-z
DIGIT	0-9
DOT	.
COLON	:
ASSIGN	=
MULT	*
PLUS	+
LEFT_B	(
RIGHT_B	)
OTHER	Any other symbol

Automaton final states

State	Meaning
-1	Number recognized
-2	Identificator recognized
-3	Found any other symbol after assignment
-4	Found any other symbol after +
-5	Found any other symbol after *
-6	Found any other symbol after (
-7	Found any other symbol after)

Figure 5: Details of the deterministic finite automaton

3 Specification of the Vlan language

The main functionality of the Vlan language revolves around working with vectors and numbers. Therefore, the language supports two data types:

- scl - integer number
- vec - vector (array of integers)

Each Vlan language instruction must end with a semicolon ";". To declare a variable, it is necessary to specify its type and name. In the case of creating a vector, its dimensionality should be defined. If the compiler detects two variables with the same names, it will return an error. Here is an example of variable declaration:

```
scl num;
vec i{4};
```

Available operators:

- <vec> : <scl> - obtain the vector element with the corresponding index. Vector element indices start from zero.
- <vec1> : <vec2> - obtain a vector composed of elements from vector vec1 with indices corresponding to the values of the elements in vector vec2. For example: [2,4,8,6]:[1,1,3,2] -> [4,4,8,6].
- <vec1> . <vec2> - dot product of two vectors.
- <vec1 / scl1> +, -, *, / <vec2 / scl2> - perform arithmetic operation on two values. When the arguments are two vectors, the corresponding operation is performed on their elements, and a third resulting vector is returned. If the operation is performed

between a vector and a number, the number is first converted into a vector, where the values of its elements are equal to the input number. The dimension of this vector will be equal to the vector on which the operation is performed, and then the arithmetic operation will be performed on the two vectors.

- (...) - parentheses to increase the priority of operator execution.

In the Vlan language, there is a conditional operator "if". The argument of the "if" block can only be a value of integer type. The content of the "if" block will be executed if the argument is greater than zero. For example:

```
if <scl> {
       <stmnt> -- will execute if scl > 0
}
```

The high-level Vlan language also supports loops with conditions for repetitive execution of instructions. The keyword "loop" is used to define them. Similar to the conditional "if" statement, the argument of the loop with a condition can only be a value of integer type. The content of the loop block will be executed a specified number of times. Example:

```
loop <scl> {
       <stmnt> -- will execute scl times
}
```

The keyword "print" is used in the Vlan language to display values to the user on the screen. To print multiple values at once, they can be listed separated by commas. Additionally, the "print" statement can be used to output strings to the user's screen. Example:

```
print element [,element..];
```

The "read" statement in Vlan is designated for reading values, both vector and integer types, from the console. The argument of this instruction can only be an identifier. To input a vector value, the user needs to list the elements separated by commas. Example of using the instruction:

```
read <iden>;
```

4 Syntax Analysis Phase

To implement the syntax analysis phase of the Vlan programming language, the latest version of ANTLR v4 [8, 9] was utilized. Any created grammar must be stored in a file with the g4 format. The entry point of our grammar is the rule program: codeblock EOF. It can be observed that the main task of this rule is to separate the content of the file from its end, achieved by using the standard ANTLR EOF rule. Next, let's examine the codeblock rule. Its purpose is to distinguish the instructions of the ANTLR language.

```
codeblock: statement*;
```

All available constructs of the language processor are defined by the statement rule. In total, there are 5 different constructs in our language [3]: variable declaration, assignment, if block, loop block, and the print statement.

```
statement: declaration | assignment | if_block | loop_block | print_statement;
```

Let's examine the declaration rule. Its purpose is to identify instructions responsible for variable declarations. The Vlan language supports two data types: integer type - scl, and vector type - vec. It's worth noting that when declaring a vector variable, the user must specify its dimensionality. For this reason, the declaration rule has been divided into two sub-rules - one for recognizing integer variable declarations and another for vector declarations.

```
declaration: scalar_dec  SCOL | vector_dec SCOL;
scalar_dec: SCALAR_TYPE ID;
vector_dec: VECTOR_TYPE ID LCBRACKET POSITIVE_DIGIT RCBRACKET;
```

The purpose of the assignment rule is to identify assignment instructions. In the left part of the assignment, we can have either a variable identifier or a reference to a vector element. The right part can contain any expression.

```
assignment: (vector_element | ID) ASSIGN expression SCOL;
```

The expression rule is one of the most complex in our grammar. Its task is to recognize any expressions available in the language, including arithmetic operations on vectors and numbers, dot product of vectors, priority elevation using parentheses, etc. It is essential to emphasize the order of branches in the expression rule. The higher a rule is in the hierarchy, the higher its priority. For further convenience in working with this rule in the program code, aliases were assigned to each branch.

```
expression
: ID                                     # identifierExpression
| digit                                  # digitExpression
| vector                                 # vectorExpression
| STRING                                 # stringExpression
| LBRACE expression RBRACE               # parenExpression
| vector_element                         # vectorElementExpression
| vector_product                         # vectorProductExpression
| expression (MULT | DIV ) expression    # multOrDivExpression
| expression (PLUS | MINUS) expression   # plusOrMinusExpres
```

Let's now consider the rules that govern the recognition of if and loop blocks. Both of them have a similar structure. The arguments of the blocks can be any expression, and the body can be any sequence of instructions. We can see that at this stage, data types are not enforced; this will happen during the compilation phase of the program.

```
if_block: IF expression LCBRACKET statement* RCBRACKET;
loop_block: LOOP expression LCBRACKET statement* RCBRACKET;
```

The "read" rule is responsible for recognizing input instructions. It is defined by the keyword "read" followed by the identifier of the variable to be read.

```
read_statement: READ ID SCOL;
```

The last of the main parser rules is the rule for recognizing output instructions. It is defined by the keyword "print" followed by any expression. Additionally, there is the possibility to print the results of multiple expressions at once by listing them with commas or to print the value of a string type.

```
print_statement: PRINT expression (COMMA expression)* SCOL;
```

Let's illustrate the operation of ANTLR using the example of constructing a parse tree (Figure 7) for a fragment of code written in the Vlan language (Figure 6).

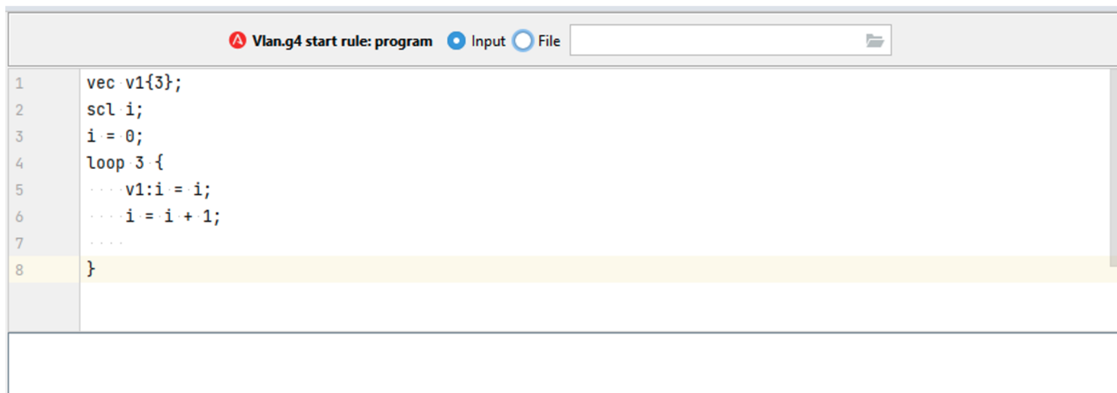

Figure 6: Input window for parsing a textual sequence

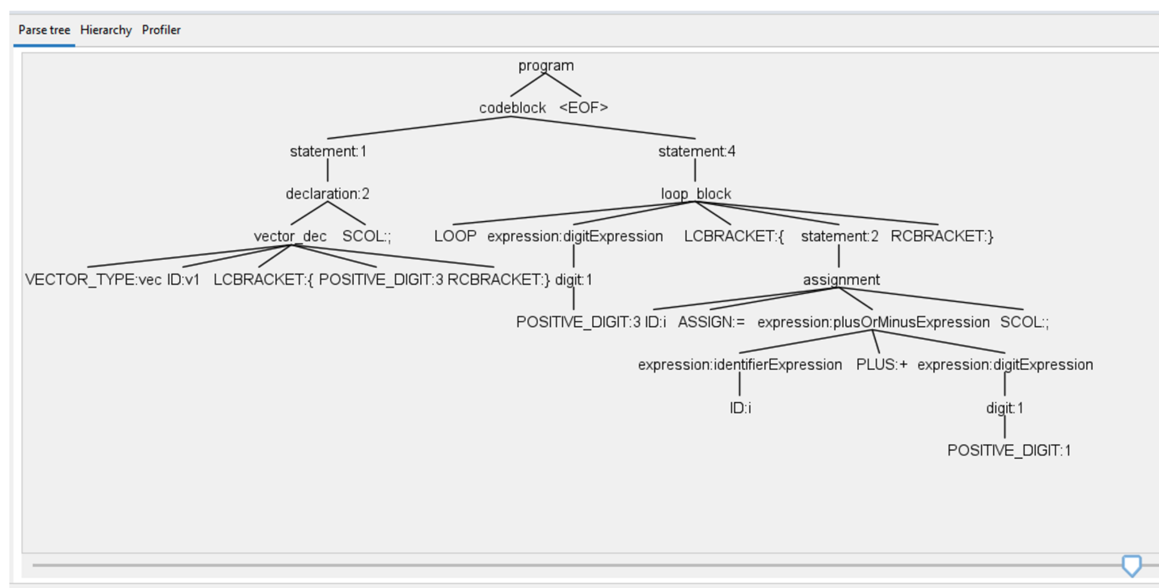

Figure 7: Parse tree visualization window

5 Intermediate code generation phase

Utilizing the behavioral design pattern "Visitor," after traversing the constructed parse tree (Figure 7), we obtained the following code in the intermediate language CIL.

```
.maxstack 3                  // Initializing the stack and variables
  .locals init (
      [0] int32[],           // Initializing the array v1
      [1] int32              // Initializing variable i
  )
  ldc.i4.3                   // Loading the constant value 3 onto the stack
  newarr [System.Runtime]System.Int32 // Creating an integer array with a size
                                      // equal to the value on the stack (3)
  stloc.0                    // Storing the array in variable 0 (v1)
  ldc.i4.0                   // Loading the constant value 0 onto the stack
  stloc.1                   // Storing the value from the stack in variable 1 (i)
  ldc.i4.3                  // Loading the constant value 3 onto the stack
                            // (number of loop iterations)
  start_loop_0:             // Label for the beginning of the loop
  dup                       // Duplicating the current stack value
                            // (remaining loop iterations) to preserve it
  brfalse end_loop_1        // Branching to the label indicating
                            // the end of the loop if the current stack value
                            // (remaining iterations) is equal to 0
  ldloc 0                   // Loading the array from variable v1
                            // onto the stack
  ldloc 1                   // Loading the value of variable i
                            // onto the stack; it serves as the index
                            // for the array element to be written
  ldloc 1                   // Loading the value of variable i
                            // onto the stack; it serves as the value to be
                            // written to the corresponding array element
  stelem.i4 Int32           // Storing the last value from the stack into the
                            // for the stelem instruction)
  ldloc 1                   // Loading the value of variable i onto the stack
  ldc.i4.1                  // Loading the constant value 1 onto the stack
  add                       // Adding the value of variable i to the constant
  stloc 1                   // Storing the new value in variable i
  ldc.i4.1                  // Loading the constant value 1 onto the stack
  sub                       // Subtracting the value on the top of the stack
                            // (remaining loop iterations) from the second to
                            // last stack element (1)
  dup                       // Duplicating the remaining loop iterations
                            // to preserve it after the next comparison
  brtrue start_loop_0       // Branching to the label indicating the start
                            // of the loop if the value on the top of the stack
                            // (remaining iterations) is greater than 0
  end_loop_1:               // Label indicating the end of the loop
  pop                       // Popping the value that stored the remaining
                            // loop iterations from the stack (it will always
                            // be 0 after the loop completes)
  ret                       // Returning from the program
```

6 Integration of an LSP server for interaction between code editors and IDEs with the Vlan language

The obtained list of program tokens and abstract syntax tree at the previous stages is processed by the LSP server, facilitating syntax highlighting, autocompletion, and validation. IDEs that support the LSP protocol, such as Visual Studio, VS Code, IDEA, Sublime, Atom (Figure 8), act as clients to the LSP server.

Through LSP, code editors and IDEs can interact with various programming languages and access different features and services without the need to create separate plugins for each language. This streamlines the work for developers and ensures a unified user interface for working with different programming languages.

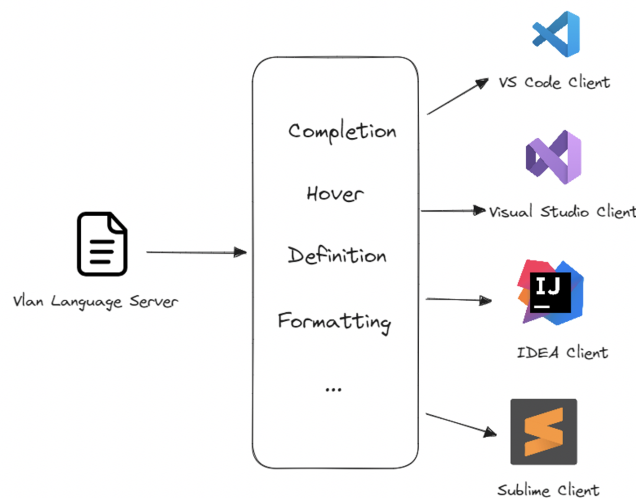

Figure 8: Interaction of LSP Server and Clients

Interaction between the client and the server takes place through a textual protocol based on JSON-RPC (JavaScript Object Notation Remote Procedure Call) [10]. An example of a client's request to the server:

```
"jsonrpc": "2.0",
"id" : 1,
"method": "textDocument/definition",
"params": {
}
"textDocument": {
    "uri": "file:///p/mseng/VSCode/Playgrounds/my_code.vlan"
},
"position": {
    "line": 3,
    "character": 12
}
```

Using LSP, we've implemented syntax highlighting for Vlan language, error and warning analysis (Figure 9), and code autocompletion (Figure 10).

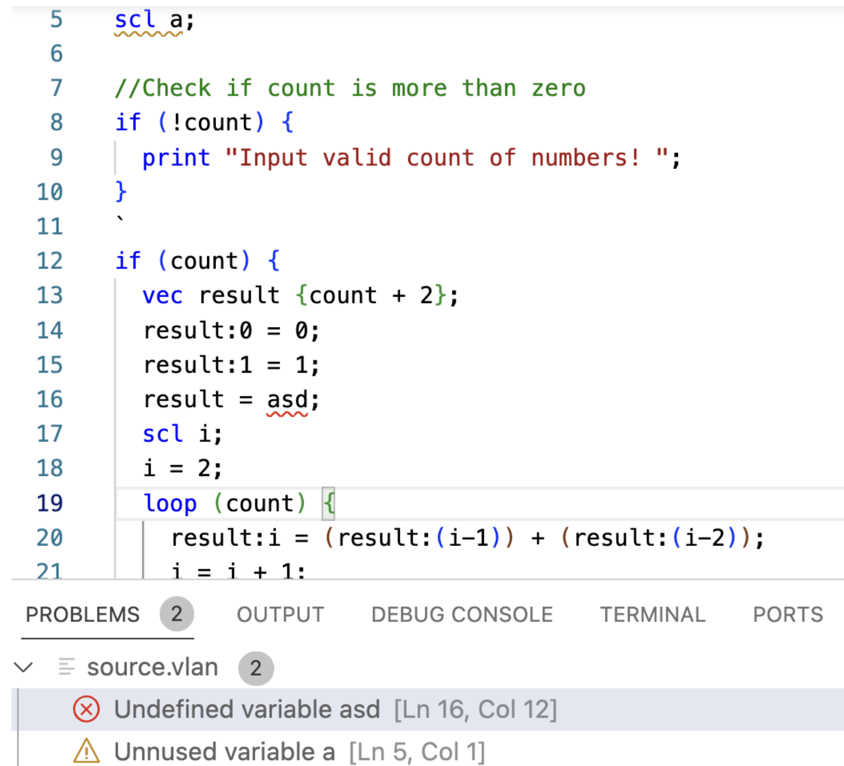

Figure 9: Code analysis

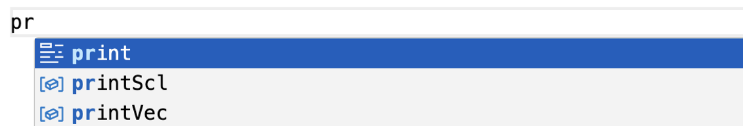

Figure 10: Code autocompletion

Thus, the development includes the creation of an interpreter for a proprietary high-level language Vlan into the Common Intermediate Language (CIL) bytecode. The execution of the generated code is translated onto the .NET platform. A unified interface is provided for developers to enable syntax highlighting and other features in code editors and Integrated Development Environments (IDEs) for various programming languages.

7 Conclusions

This article presents a comprehensive approach to developing a programming language for the .NET platform. The design of the language's syntax and semantics involves creating grammar and defining operations and data types. Implementing the language processor, taking into account the specific features of the .NET platform, allows for the generation of

corresponding Common Intermediate Language (CIL) code for the .NET virtual machine. The established class hierarchy for operations on regular languages enables the construction of complex formalisms and their utilization in implementing diverse tasks in software applications. In summary, the deployed methodology for programming language development on .NET underscores the importance of a systematic approach to this process and the capability to create languages that meet specific tasks and user requirements.

References

[1] Sopronyuk T.M., Drobot A.V. Development of a GUI for the custom .NET language Vlan. Proc of the Intern Conf. "Mathematics and Information Technologies", Chernivtsi, Ukraine, September 28–30, 2023, Chernivtsi National University, Chernivtsi, 2023, 320-323. (in Ukrainian)

[2] Drobot A.V., Development of a language processor for the .NET platform using ANTLR. Proc of the Intern Conf. "Applied Mathematics and Information Technologies", Chernivtsi, Ukraine, September 22–24, 2022, Chernivtsi National University, Chernivtsi, 2022, 260–262. (in Ukrainian)

[3] Drobot A.V., Development of a language processor for the .NET platform using ANTLR. Qualification work, Chernivtsi, Chernivtsi National University, 2022, 84 p. (in Ukrainian)

[4] Sopronyuk T.M., Drobot A.V. Development tools for programming languages on the .NET platform. Proc of the Intern Conf. "Applied Mathematics and Information Technologies", Chernivtsi, Ukraine, September 22–24, 2022, Chernivtsi National University, Chernivtsi, 2022, 273–276. (in Ukrainian)

[5] Sopronyuk T.M. Systems Programming. Part II. Elements of Compilation Theory: Educational manual in two parts. Chernivtsi National University, Chernivtsi, 2008, 84 p. (in Ukrainian)

[6] Sopronyuk T.M., Sopronyuk A.Yu. Computation of regular expressions over formalisms of automata languages. Proc of the Intern Conf "Analysis, Modeling, Control, Development" of Economic Systems (AMUR-2011)", Sevastopol, Ukraine, September 12-18, 2011, 348-349. (in Ukrainian)

[7] Alfred V. Aho, Jeffrey D. Ullman. The theory of Parsing, Translation and Compiling. Volume 1. Prentice-Hall, Inc., 1972.

[8] ANTLR Reference Manual [Electronic resource] - Access mode: `https://www.antlr3.org/share/1084743321127/ANTLR_Reference_Manual.pdf`

[9] Santosh Singh. ANTLR C# Cookbook [Electronic resource] - Access mode: `https://www.amazon.com/Create-Compiler-Using-ANTLR-Crash-Course-ebook/dp/B09BJ4CRTJ`

[10] Understanding the Language Server Protocol [Electronic resource] - Access mode: `https://medium.com/@malintha1996/understanding-the-language-server-protocol-5c0ba3ac83d2`

Received 20.11.2023

Сопронюк Т. М., Сопронюк А. Ю., Дробот А. В. *Фази побудови мовного процесора для платформи .NET* // Буковинський матем. журнал. — 2023. — Т.11, №2. — С. 71–84.

У статті представлено комплексний підхід до розробки мов програмування для платформи .NET. Не зважаючи на наявність існуючих мов програмування та їхніх мовних процесорів, існують виклики, пов'язані з необхідністю створення спеціалізованих мов для конкретних галузей, а також оптимізації ефективності та продуктивності програмних рішень. Особливо ця проблема помітна на спеціалізованих виробництвах, де дуже часто

виникає потреба в розробці власної вузькоспеціалізованої мови. Створення ефективних та високорівневих мов, оптимізованих для конкретних завдань, є важливим етапом у розвитку програмного забезпечення. Виділення невирішених аспектів та прогалин у науковому підході до створення таких інструментів є ключовим для подальших досліджень. У статті продемонстровано створену ієрархію класів для операцій над регулярними мовами та наводиться конкретна специфікація власної мови програмування Vlan. Розроблена ієрархія класів дозволяє конструювати складні формалізми, такі як таблиці переходів автомату і праволінійні граматики, і використовувати їх для реалізації задач розпізнавання або перетворення ланцюжків визначеної структури в різних програмних додатках. Автори розглядають етапи створення мови, починаючи від проектування синтаксису та семантики, до реалізації мовного процесора з генерацією коду CIL для віртуальної машини .NET. Також досліджено процес інтеграції мов програмування в сучасні середовища програмування. Для такої інтеграції використано протокол взаємодії мовних серверів LSP та генератор синтаксичних аналізаторів ANTLR. Побудовано TextMate граматику для мови програмування Vlan і створено класи LSP клієнта та сервера. Результати дослідження вказують на важливість системного підходу до розробки мов програмування та їхньої адаптації до конкретних завдань та вимог користувачів у середовищі .NET.