ASSIGNMENT OF BACHELOR'S THESIS

Title:	DET language IDE
Student:	Toghrul Sultanzade
Supervisor:	Ing. Ondřej Guth, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2020/21

Instructions

The aim is to implement an integrated development environment (IDE) for DET scripting language (it is a proprietary language based on Java). The IDE should be capable of syntax highlighting, autocompletion, and error recognition.
1. Study existing parser of DET language [1] and modify it for the needs of the IDE.
2. Research existing open-source IDEs and techniques for syntax highlighting, autocompletion and error recognition.
3. Based on the research, use existing libraries and algorithms to implement the IDE and its required features as a prototype.
4. Use appropriate tools and methods to test the results.

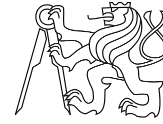
References

[1] GRANKIN, Daniil. A translator of DET scripting language into Java. Bachelor's thesis. Czech technical university in Prague, 2019. Available from: http://hdl.handle.net/10467/83386.

doc. Ing. Jan Janoušek, Ph.D. Head of Department	doc. RNDr. Ing. Marcel Jiřina, Ph.D. Dean

Prague October 17, 2019

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Science



Bachelor's thesis

DET language IDE

Toghrul Sultanzade

Supervisor: Ing. Ondrej Guth, Ph.D.

21st February 2020

Acknowledgements

Firstly, I would like to express my appreciation and thanks to my thesis supervisor, Ing. Ondrej Guth, for his professional attitude and dedication to help me. The door to his office was always open, whenever I had troubles and obstacles in the process of writing the thesis. Moreover, I would like to thank my family and friends for support during writing this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 21st February 2020 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Toghrul Sultanzade. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Sultanzade, Toghrul. *DET language IDE.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Tato diplomová práce si dává za úkol navrhnout a vyvinout vylepšené integrované vývojové prostředí, které vyřeší problémy a zreviduje schopnosti stávajícího Scriptovacího Editoru. Tento Scriptovací Editor je užíván k vývoji a implementaci skriptu, používající DET skriptovací jazyk popsán v diplomové práci [1]. DET jazyk má syntaxi založenou na Java jazyce s vlastními rozšířeními reprezentující Java proměnné a Java metody. Vady předchozí implementace jsou řešeny v prototypu projektu, který je předmětem této práce. K rozvoji podpory pro DET skriptovací jazyk jsou použity dnešní moderní postupy. Současný stav je prezentován společně s preferovanou metodologií zpracování uživatelských vstupů.

Klíčová slova doplňování kódu, našeptávání, zvýrazňování sytanxe, vývojové prostředí, skriptovací jazyk, ANTLR, gramatika, AST

Abstract

This thesis aims to design and develop an improved integrated development environment solution, which resolves problems and revises the capabilities of the existing proprietary Scripting Editor implementation. The Scripting Editor is used to develop and implement scripts, written in DET scripting

language that is described in the thesis [1]. DET language has Java-like syntax with custom language extension into the variables and methods of java classes. The flaws of the previous implementation are approached in the prototype project, which is the subject of this thesis. The modern and contemporary approaches are used to develop support for the DET scripting language. The state of the art is presented alongside the favored methodology of the user input processing.

Keywords code completion, autocomplete, syntax highlighting, development environments, scripting language, ANTLR, grammar, AST

Contents

Introduction **1**
Thesis Overview 1

1 Theoretical background **3**
1.1 Algorithmic trading 3
1.2 DET scripting language 3
1.3 Programming Language 4
1.4 Domain-specific language 5
1.5 Formal Grammar 5
1.6 Graph theory 9
1.7 Parsing 11
1.8 Integrated development environment 12
1.9 Research of modern solutions 16
1.10 Available tools to implement IDE features 22

2 Current State **25**
2.1 Text editor 25
2.2 Features of the text editor 25
2.3 Syntax Highlighting 26
2.4 Code completion 28
2.5 Key disadvantages of the current state 29

3 Analysis and Design **33**
3.1 Web IDE 33
3.2 Features of IDE 34
3.3 Editor framework 36

4 Realisation **37**
4.1 Environment 37
4.2 Syntax highlighting 39

4.3 Code completion 40
4.4 Error recognition and indication 42

5 Testing **45**

Conclusion **49**
Future work 49

Bibliography **51**

A Acronyms **57**

B Contents of enclosed CD **59**

List of Figures

1.1 An example of the syntax diagram for an expression rule, from [2]. 5
1.2 Example of two different parse trees for a single sentence from a language. 8
1.3 An example of a radix tree. 11
1.4 Structure of the code completion plugin in Code::Blocks IDE, from [3] 17
1.5 The main flow of a parser in Code::Blocks 18
1.6 An example of automatic code completion in Code::Blocks. 19

2.1 An example of using primitive types from the DET language in the AvalonEdit text editor. 27
2.2 Representation of the regular expression used to match a declared variable from source code. 29
2.3 An example of the code completion list for the word GET_C. . . . 30
2.4 An example of the code completion list for the word variable. . . . 31

4.1 ANTLR grammar rules representing a digit or letter in a DET script. 39

5.1 Mean features score. Blue bars represent original editor, while red bars represent prototype evaluations 46

Introduction

Low latency trading has been made possible by introducing informational technologies into the capital markets. Algorithmic trading became widely spread during the past few decades, thus raising demand for platforms supporting scripting and real-time usage of the broker's business logic. Dynamic Electronic Trading (DET) company provides one of such platforms, granting fast and discrete tools for implementing the custom routing logic.

In the world of informational technologies, the development of any script or application in a plain text editor brings various complications along the way. One such complication is the time consumed by the compilation of the program only to reveal minor errors in the code. As developers are facing more challenging tasks each day, the significance of time spent on looking up the identifier or locating a particular error has increased. There has been developed an integrated development environment concept, aimed at providing development assistance in the form of various facilities. IDE has become a demanded tool ever since, saving lots of coding time.

This thesis takes as a goal to design an integrated development environment that supports business-driven DET scripting language. An existing parser and grammar for this language can be found in this [1].

DET language IDE is one of the various environments, that is capable of syntax highlighting, code completion and error recognition. The IDE is a lightweight web-based application, providing swift utilization support to most of the popular platforms. It helps clients to avoid searching for particular documentation entry, highlights errors on the go while the developer is writing code and displays source code in different colors and fonts to make it easier to read.

Thesis Overview

Chapter 1 introduces the reader to the terms, tools and concepts used in the project, as well as to the state of art.

Chapter 2 shows and describes an analysis of implementation problems and features of the existing environment that supports the DET language.

Chapter 3 describes approaches to the problems of the old IDE and composes a solution.

Chapter 4 discusses the implementation details of solution.

Chapter 5 describes the necessary tests for the application.

Chapter 1

Theoretical background

The project aims to implement an integrated development environment for DET scripting language, which should be capable of syntax highlighting, code completion, and error recognition. Therefore, in this chapter, I provide an overview of the most common solutions and existing frameworks available to implement theses features. In addition, I present a detailed review of the following concepts: FIX, programming language, parsing, formal grammar, terminology of graph theory, AST and IDE.

1.1 Algorithmic trading

Algorithmic trading is a method of executing a large order (parent order) using special algorithmic instructions. These pre-programmed trading instructions divide the order into several sub-orders (child orders) with their characteristics of price and volume, and each of the sub-orders is sent at a specific time to the market for execution [4]. Such algorithms were invented so that traders do not have to monitor quotes continually and divide a large order into small ones manually.

The main goal of algorithmic trading is to reduce the cost of executing a large order (transaction cost), minimize its impact on the market (market impact), and reduce the risk of its non-execution. It is widely used by investment banks, pension, hedge, and mutual funds, as these institutional investors in their activities operate with large orders and therefore, cannot place such orders on the market with the risk of losses [5].

1.2 DET scripting language

DET scripting language is a proprietary language designed to implement scripts in the DET platform. The language has Java-like syntax with custom language extensions enriching the list of the variables and methods. The

existing grammar that describes DET language is provided in the thesis DET scripting engine work [1].

1.3 Programming Language

Any language, including a programming language, obeys a number of rules. They are usually divided into rules that determine the syntax of a language and rules that determine its semantics.

1.3.1 Syntax and Semantics

The language **syntax** is a set of rules that describes combinations of alphabet characters that are considered to be a properly structured program or its fragment in some language [6]. Each programming language has a syntactic description as part of the grammar, a formal description of the grammar is presented in Section 1.5.

The syntax are checked in the early stages of translation. Program **translation** is converting a program presented in one of the programming languages into a program in another language. For example, the validity of the source code of programs can be checked when editing code using the IDE, for more details about ad IDE see Section 1.8. If any aspects of the source code do not match the syntax of the specified programming language, a syntax error occurs [7].

The **semantics** of a language is a set of rules that determine the meaning of syntactically correct language constructions and their content.

Programming languages belong to the group of formal languages(see Section 1.5.1) for which, unlike natural languages, syntax and semantics are uniquely defined. The Backus-Naur (BNF) form or syntax diagrams are usually used to construct a description of the syntax of a language, a detailed description of the BNF is presented in Section 1.5.4. The syntax diagram allows you to depict the structure of the syntactic unit graphically, see an example of such a diagram in Figure 1.1.

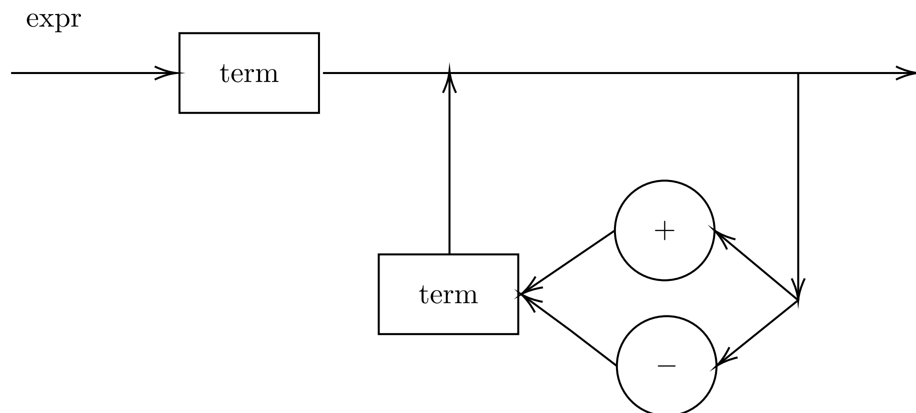


Figure 1.1: An example of the syntax diagram for an expression rule, from [2].

1.4 Domain-specific language

A **domain-specific language** is a programming language with a higher level of abstraction specialized for a specific class of problems. The construction of such a language and its data structure reflect the specifics of the tasks solved with its help. It is a crucial concept of language-oriented programming.

Typically, a domain-specific language is less complicated than a general programming language such as C ++, Java, or Ruby. Typically, this type of language is not intended for use by developers, but instead by those who are fluent in the domain which is addressed by the domain-specific language.

1.5 Formal Grammar

The basic definitions of the terminology used in this section are given below.

1.5.1 Basic notions

Terminal symbols is minimal grammar elements that do not have their own grammatical structure. Terminal characters are either predefined identifiers or chains — sequences of characters in quotation marks or apostrophes.

Non-terminal symbols – grammar elements that have their own names and structure. Each non-terminal symbol consists of one or more terminal and non-terminal symbols, the combination of which is determined by the rules of grammar. Each non-terminal character has a name, which is a string of characters.

Production rule – A grammar rule specifying which symbols can be substituted with other symbols. Such rules can be used to generate or parse strings. A production is of the form $left\ part \rightarrow right\ part$, where:

- $left\ part$ – non-empty sequence of terminals and non-terminals containing at least one non-terminal
- *right part* – any sequence of terminals and non-terminals

Start symbol – a member of a set of non-terminals from which all strings in a language can be obtained by sequentially applying production rules

Alphabet – finite set of terminal symbols. A sequence of symbols in the alphabet is called a string over an alphabet.

Formal language – a set of finite words (e.g., words, sentences) over a finite alphabet [8].

Empty string (denoted as ϵ) – is the special case of a string over an alphabet where the sequence of characters has length zero, so there are no symbols in the string.

Derivation (denoted as $\Rightarrow$) – a sequence of applications of the grammar rules that produces a string of terminals. A derivation is also called a **parse**.

A **formal grammar** in the theory of formal languages is a way of describing language, that is, constructing correct sentences from a specified language [9]. Grammar is a quadruple $G = (N, \Sigma, P, S)$, where

- N – a finite nonempty set of non-terminal symbols
- Σ – a finite nonempty set of terminal symbols ($\Sigma \cap N = \emptyset$)
- P – a set of production rules
- $S \in N$ – the start symbol of the grammar

Other possibilities to describe grammar: Backus-Naur Form (BNF) and Extended Backus-Naur Form (EBNF) (see Section 1.5.4).

Example: A language of unsigned integers can be described using grammar G, which is specified by BNF [7].

$$G = (\{integer, digit\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, P, integer)$$

, where:

- $\{integer, digit\}$ is a set of non-terminals
- $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is a set of terminals
- P is a set of productions rules:

  $integer \rightarrow digit\ integer \mid digit$

  $digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- *integer* is the starting symbol

1.5.2 Grammar Types

According to the **Chomsky hierarchy**, grammars are divided into 4 types, each subsequent one is a more limited subset of the previous one (but also easier to analyze):

- **Type 0** unrestricted grammars – any rules are possible
- **Type 1** context-sensitive grammars – the left side may contain one non-terminal surrounded by a "context" (sequences of symbols that are in the same form on the right side). The non-terminal itself is replaced by a nonempty sequence of terminals on the right side.
- **Type 2** context-free grammars - the left part consists of one non-terminal.
- **Type 3** regular grammars are simpler, equivalent to finite automata.

Since most programming languages are described using context-free grammars, the more detailed description is provided in the next section.

1.5.3 Context-free grammars

Since the Algol project [10], the formal definition of a language includes some sort of context-free grammar. **Context-free grammar** is a particular case of formal grammar, in which the left parts of all production rules are single non-terminals. The meaning of the term "context-free" is that it is possible to apply products to a non-terminal, regardless of the context of this non-terminal [9]. Context-free grammars are widely used in computer science. They define the syntactic structure of most programming languages, structured data, etc.

A context-free grammar is **ambiguous**, if there is a sentence from a language generated by this grammar that is a result of at least two different parse trees [11].

Trivial example of ambiguous grammar:

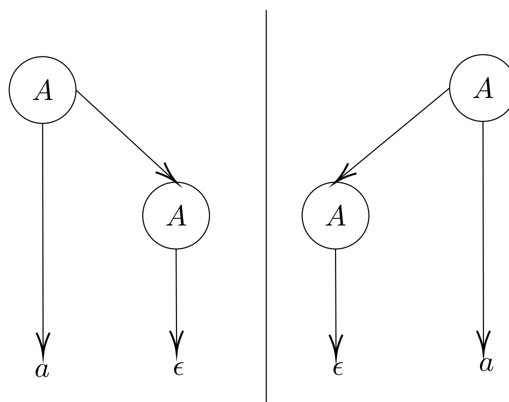


Figure 1.2: Example of two different parse trees for a single sentence from a language.

A grammar that can generate the string "a" with the following production rules has two different parse trees for this string(see Figure 1.2).:

$$A \rightarrow aA \mid Aa \mid \epsilon$$

where:

A is a starting symbol

1.5.4 BNF and EBNF

Backus–Naur form or **Backus normal form (BNF)** is a formal syntax description system for the representation of context-free grammars, the definition of such grammars is given in Section 1.5.3. It is widely used to describe the syntax of programming languages, data, communication protocols (for example, in RFC documents), etc. **Extended backus-naur form (EBNF)** is one of the variations of the Backus-Naur form which differs only in more capacious designs and EBNF is commonly used [12].

As in the BNF, the grammar description in the EBNF is a set of rules that define the relationship between terminal symbols (terminals) and non-terminal symbols (non-terminals).

The rule in the EBNF is:

$$Identifier = Expression.$$

where

- $Identifier$ is the name of a non-terminal symbol
- $Expression$ is a combination of terminal and non-terminal characters and special characters corresponding to the EBNF rules.

- The **dot** . at the end is a special character that indicates the end of the rule.

The semantics of the EBNF rule are a non-terminal symbol specified by the identifier to the left of the equal sign, is a combination of terminal and non-terminal symbols defined by the expression.

A complete grammar description is a set of rules that defines all non-terminal symbols of a grammar so that each non-terminal symbol can be reduced to a combination of terminal symbols by sequential (recursive) application of the rules. There is no special order of rules in the EBNF definition for writing rules, although such prescriptions can be introduced when using the EBNF with software tools that automatically generate parsers.

1.6 Graph theory

In this section, I present a brief introduction to graph theory, since the terminology related to graphs is used in my work.

1.6.1 Graph

In computer science, a graph G is an ordered pair

$$G = (V, E)$$

where

- V – a nonempty set of vertices or nodes
- E – a set of pairs of vertices, where elements of this set are called edges.

Usually, a graph means an undirected graph for distinguishing from a directed graph [13].

1.6.2 Tree

A **tree** is a connected acyclic graph. Connectivity means the presence of paths between any pair of vertices. The acyclic graph indicates the absence of cycles in the graph, which means that there is only one path between pairs of vertices [14].

1.6.2.1 Basic Definitions

Path in a graph – a sequence of vertices that are joined by a sequence of edges, where the edges are distinct.

Rooted tree – a tree in which one vertex is marked as the root of the tree [15].

Parent of some vertex, in a rooted tree – the vertex connected to it on the path to the root. The root has no parent.

Child of a vertex v – vertex u, where v is the parent of vertex u, in a rooted tree [13].

Leaf – a vertex with no children.

Descendant of a vertex v – any vertex which is either the child of v or is the descendant of any of the children of v [15].

Ascendant of some node v – any node which is either the parent of v or is the ascendant of the parent v [15].

1.6.3 Prefix and Radix tree

1.6.3.1 Prefix tree

A **prefix tree**, also called a **trie** or **digital tree** is a data structure that allows you to store a symbol table where the keys are strings [16]. It is a rooted tree, where all edges are marked with a symbol. In addition, for any node, all the edges connecting this node with its children are marked with different symbols. The position of the node in the tree determines the key with which it is associated. Moreover, all descendants of a node have a common string prefix associated with the given node. Not every node has a key, though some of them may correspond to keys. However, all leaves contain strings. Hence, the trie contains a given key string if and only if this string can be read on the way from the root to some selected node [17].

1.6.3.2 Radix tree

A **radix tree** (also called as compact prefix tree, compact Patricia tree or radix trie [18]) is a data structure that is a memory-optimized implementation of the prefix tree. Unlike prefix trees, node of a radix tree can be marked with a sequence of symbols, which makes this tree more efficient for small sets of strings (especially if the strings themselves are long enough), and also for sets that have a small number of lengthy prefixes. An example of a radix tree can be seen in Figure 1.3.

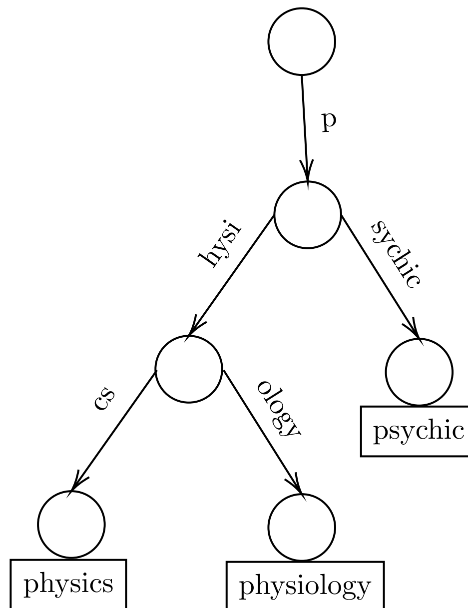


Figure 1.3: An example of a radix tree.

Radix trees find particular applications for parsing languages and in the IP routing [19, 20]. The compact prefix tree is also one of the data structures used in Linux kernels [21].

1.7 Parsing

Parsing or syntactic analysis, in the context of programming languages, is transforming of human-readable representation of source code to a syntactical description which is often some kind of parse tree or abstract syntax tree (AST) [22].

1.7.1 Lexer and Parser

A **lexer** or lexical analyzer is a software program that performs lexical analysis. **Lexical analysis** is the process of breaking a piece of text into different words, which are called **tokens** in computer science. Lexer generates an lexical error if some text cannot be recognized to any type of token [23].

Tokens are sequence of characters defined by grammar rules, by means of patterns using regular expressions.

A **parser** or **syntactical analyzer** is a program that converts input data (usually text) into a structured format. The parser parses the tokens generated from the lexer.

1.7.2 Abstract Syntax Tree

Abstract syntax tree (AST) in computer science is a tree, in which internal vertices are mapped (labeled) with programming language operators, and leafs with corresponding operands. ASTs are used in parsers to provide an intermediate representation of the program between the parse tree (CST) and the data structure [24].

1.7.3 Concrete Syntax Tree

A **concrete syntax tree (CST)** or **parse tree** is a tree that records the tokens and grammar rules used to match the text. The starting symbol of a grammar that was used to parse the text is always the root node of CST. The leafs consist of all terminals that were found when the text was parsed and all other nodes of the tree contain encountered non-terminals [25].

1.7.4 Symbol table

A **symbol table** is an important data structure used to store information about the declaration or appearance of variable names, method names, objects, classes, interfaces in the program's source code. The information recorded in the entries of the symbol table may be related to the data type, visibility, region, memory location, and location in the source code[26].

1.8 Integrated development environment

An **Integrated Development Environment (IDE)** is a software application that provides comprehensive services to make it easier to develop software for the developer or programmer. An IDE scans and parses the source program to provide these services. A few examples of the most commonly used IDEs are IntelliJ IDEA [27] and Visual Studio [28].

Usually, an IDE consists of:

- a text editor(e.g., source code editor)
- syntax highlighting
- Code completion (IntelliSense)
- a debugger

Many modern IDEs, such as NetBeans and Eclipse, contain a compiler, an interpreter, a class browser, an object finder, and a class hierarchy diagram, for use with the development of object-oriented software.

The first IDEs were created to work using a console or terminal. Before that, programs were created on paper, entered into the machine using pre-prepared paper media (punched cards, punched tape).

Maestro I is a product from Softlab Munich, was the first integrated software development environment in the world in 1975 [29]. Today, Maestro I belongs to history and can only be found at the Museum of Information Technology in Arlington.

Certain IDE features may use AI to speed up or improve results. In particular, to improve the IDE functionality, it is possible to gather information between developers. For example, intelligent code completion can be achieved using a data-driven approach to code completion [30].

The following section describes the basic functions of the IDE, language server protocol, and online integrated development environment.

1.8.1 Text editor

A text editor is an independent computer program designed to create and modify text data [31]. Text editors are designed to work with text files online. They allow you to view the contents of text files and perform various actions on them - insert, delete and copy, search and replace text, sort lines, view character codes and convert encodings, etc.

1.8.2 Syntax highlighting

Syntax highlighting is a feature of integrated development environments and text editors that displays source code in different colors and fonts according to some criteria. Commonly used to make reading the source code of computer programs easier and emphasise syntax and semantic errors [32]. One of the first text editors to support syntax highlighting was developed by Wilfred Hansen in 1969 [33]. Since then majority of Integrated Development Environments and text editors provide syntax highlighting.

Some code formatting tools perform syntax highlighting using pattern matching instead of implementing a parser for a programming language [34]. This approach leads to the inaccurate syntax highlighting and poor performance. However, in most cases, strict parsers that are used in the compilation process, cannot parse the code displayed in the editor as it is incomplete or incorrect, while pattern matching is successful.

Syntax highlighting also helps programmers identify programming errors. For example, most editors highlight errors using a red font or red wavy lines [35]. Errors are generally classified into three groups:

- lexical errors – when some text cannot be recognized as belonging to any type of token
- syntactical errors – when the structure of the code is not correct
- semantic errors – they depend on the nature of the language. Example of semantic errors are usages of undeclared variables, or operations involving incompatible types.

1.8.3 Code Completion(IntelliSense)

Code completion is a feature in the IDE that predicts and provides the rest of the input the user is typing. Mostly, it is used when querying parameters of functions, available variables and functions in the current scope, query hints related to syntax errors, etc.

IntelliSense is a general term for code completion tool that refers to the basic features of an IDE such as complete word, quick info, parameter info, and member lists. In addition, it is also used to access documentation and to remove ambiguous names using reflection. A brief description of the code completion features is given below, for more details see [36].

- **Complete word** is a feature that completes the rest of the name after the user has entered enough characters to disambiguate the term.
- **Quick info** shows the declaration details for any identifier in the source code.
- **Parameter info** displays information about the number, types, and names of method parameters. It also indicates the next parameter that is required as the user types the function.
- **Member list** is a feature that displays all valid and available members from a particular type or namespace that the user entered before triggering IntelliSense (usually a trigger character used to display the member list).

IntelliSense first appeared in Visual Basic 5.0 Control Creation Edition in 1996, which was a publicly available prototype of Visual Basic 5.0 [37]. It has entered a new development phase with the advent of Visual Studio .NET and is currently supported in Visual Studio for languages such as C++, C#, Visual Basic, XML, HTML, and others. Starting with Visual Studio 2005, IntelliSense, by default, begins to offer code completion options as soon as the user starts typing, without typing trigger characters(e.g., '.' character). Since the available suggestions now include language constructs (such as for or if keywords), they have also been included in the list of options for auto-completion.

The next segment lists and briefly discusses the key benefits of code completion, for more details see [38].

- **Fast typing**: code completion systems assist IDE users by displaying and completing descriptive and large names for methods, variables, or classes that are commonly used nowadays.
- **Error free code**: IntelliSense offers only syntactically correct items in the completion list.

- **Valuable documentation**: when code completion systems display a list of completions, the user can see all available documentation. This additional information helps not to remember the names and descriptions of all public methods and variables from any library or framework.

1.8.4 Refactoring

Refactoring is the process of changing the internal structure of a program that does not affect its external behavior and aims to facilitate understanding of the code [39]. Fundamentally, refactoring is a sequence of transformations. Since each transformation is small, it is easier for the programmer to follow its correctness. At the same time, the entire sequence can lead to a significant restructuring of the program and improve its consistency and clarity [40].

Refactoring can be moving a field from one class to another, taking a piece of code from a method and turning it into an independent method, renaming class property or even moving the code through a class hierarchy. Each step may seem elementary, but the combined effect of such small changes can radically improve the project.

1.8.5 Version control

Version control is a feature to facilitate work with changing information. It allows you to store several versions of the same document, return to earlier versions, determine who made a change, and much more. A letter code or a number identifies each change [41].

1.8.6 Debugger

A **debugger** is a computer program for automating the debugging process: searching for errors in code. Depending on the built-in capabilities, the debugger allows you to trace, set, or change the values of variables during code execution, set and delete breakpoints or stop conditions, for more details see [41].

1.8.7 Language Server Protocol

The **Language Server Protocol** (LSP) is a protocol that is used for communication between language servers (also called services) that provide features to support programming language and integrated development environments. Historically, implementing support for features such as code completion, syntax highlighting, goto definition, or hover documentation for a programming language must be repeated for each source code editor or IDE, as development tool provides different interfaces for implementing the same features. So the fundamental goal of the protocol is to allow the programming language sup-

port to be implemented independently of the given integrated development environment [42].

1.8.8 Online integrated development environment

An **online integrated development environment**, also known as web IDE or Cloud IDE is an interactive development system that is based on browsers [43]. The web IDE can be accessed from a web browser such as Google Chrome or Mozilla Firefox, which provides a portable working environment, and allows you to develop software on low-power devices that are usually not suitable [44].

1.9 Research of modern solutions

An important part of this work is the research and analysis of existing open-source integrated development environments and techniques for syntax highlighting, code completion and error indication. This section describes some IDEs, briefly discusses their history, development, current versions and scope. Besides, I provide an overview of the available libraries and algorithms commonly used to implement the required features of an IDE.

1.9.1 Code::Blocks

Code::Blocks is a free, cross-platform, open-source integrated development environment. It is a desktop-based software application developed in the C++ programming language. The first official version was released on February 28, 2008 [45]. Currently, it supports C, C++, D (with restrictions), Fortran programming languages and can be extended with plugins. The majority of basic features of the code::blocks IDE are implemented through core plugins. These plugins are installed by default and are maintained by the official development team.

Code::Blocks uses Scintilla as a component for editing source code. **Scintilla** is a free, open-source text editing component, which provides features especially useful when editing source code. These include support for: syntax highlighting, line numbering in the margin, code completion, code folding and error indicators. The first published version of Scintilla was released on March 14, 1999, and was developed by Neil Hodgson [46].

I present an overview of the techniques that were used to implement the syntax highlighting and code completion(for more details see [47, 45]).

Syntax Highlighting is implemented in scintilla using a lexer that parses the given document and sets a predefined style for each part of the text. In other words, the lexer determines how the specified range of text

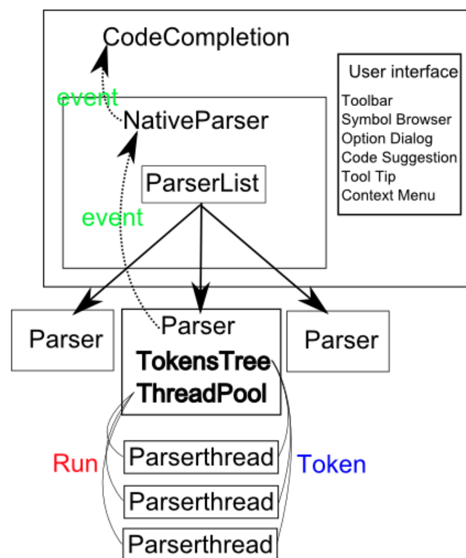


Figure 1.4: Structure of the code completion plugin in Code::Blocks IDE, from [3]

should be colored. The colors per each style are set in a controlling application that displays a scintilla window.

Each programming language has a specific implementation of the lexer and the corresponding styles written in C++. Therefore, to enable syntax highlighting for a particular programming language that is not supported by scintilla, it is necessary to write an implementation of the lexer.

Code Completion feature is implemented as a plugin that comes with the source code of code::blocks and only C/C++ programming languages are supported. The structure of this plugin is shown in Figure 1.4.

Below I present a brief overview of the main classes.

- **CodeCompletion** is the main class of the plugin that handles events sent from code::blocks.
- **NativeParser** is a member of the CodeCompletion class that manages Parser instances; that is, it creates or removes parsers when a project loads or closes, respectively.
- **Parser** – a class that mainly processes tokens. It also contains a thread pool to run the analysis task in a separate thread from this pool.
- **Parserthread** is a class for computing complex tasks for code completion or parsing code.

The main flow of a parser is demonstrated in Figure 1.5. Where the parser consists of low and high level parsers.

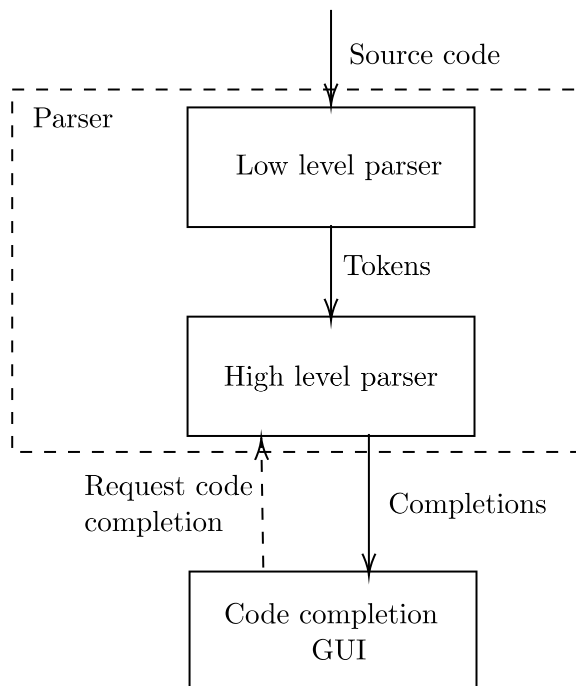


Figure 1.5: The main flow of a parser in Code::Blocks

A **low level parser** is also called a **tokenizer**, which performs lexical analysis of the source code and divides it into tokens. Each token has a unique meaning and cannot be divided into smaller tokens. For example, a token can be a keyword, identifier, number, etc. In fact, the tokenizer moves the internal pointer across the text and returns tokens to feed the high **level parser** also called a **syntax analyzer**. For performance reasons, this analyzer contains a reference to a thread pool and runs syntax analysis in separate threads. A high-level parser collects all tokens and stores them in a tree, and then the graphical user interface can request keywords associated with some tokens to show hints, code completion, or build a class browser tree.

A token class contains all the necessary information to obtain its type, location in the source code, class hierarchy, etc. All tokens are stored in the TokenTree class. Besides, for fast access to tokens, a compact prefix tree (see Section 1.6.3.2) is used to store token names. Additionally, each node contains links to tokens associated with the same name as the node's string key.

When code completion is triggered, the first step of the automatic code completion algorithm is to **find the search scope**. I present an example (see Figure 1.6) used in the documentation of code::blocks IDE [3]:

```
13
14 #include "Scintilla.h"
15 #include "SplitVector.h"
16 #include "Partitioning.h"
17 #include "RunStyles.h"
18
19 #ifdef SCI_NAMESPACE
20 using namespace Scintilla;    A
21 #endif
22                                    C
23 // Find B first run at a position
24 int RunStyles::RunFromPosition(int position) {
25     int run = starts->PartitionFromPosition(position);
26     // Go to D st element with this position
27     while ((  > 0) && (position == starts->PositionFromPartition(run-1))) {
28         run--;
29     }
30     return run;
31     |
32 }   E
33
34 // If there is no run boundary at position, insert one continuing style.
35 int RunStyles::SplitRun(int position) {
36     int run = RunFromPosition(position);
37     int posRun = starts->PositionFromPartition(run);
38     if (posRun < position) {
39         int runStyle = ValueAt(position);
40         run++;
41         starts->InsertPartition(run, position);
42         styles->InsertValue(run, 1, runStyle);
43     }
44     return run;
45 }
46
```

Figure 1.6: An example of automatic code completion in Code::Blocks.

In the example shown above, the caret is located at "**E**" label. The following steps must be taken to get the search scope.

- All child tokens of the namespaces used in the current file should be searched when providing code completion. In this particular case, all tokens from Scintilla namespace are searched, see the label "A".
- If the caret is located in the method body, then method parameters and all local variables are searched as well. In the example above, the caret symbol is located in the "RunFromPosition()" method, therefore the method parameter "position" (indicated by the label "C") and the local variable "run" (marked as "D") also correspond to tokens.
- In addition, the search scope is expanded by tokens from the class in which the caret symbol is located, see Label "B", and tokens from the global namespace are also displayed.

When the search scope is found, the algorithm tries to match the search scope with the components from the statement before the caret symbol and produces the matching result that is code completion items. The following example explains how to create components from the statement and matching algorithm works. For example, if a string of some

statement is equal to $VarA.metho|$ and the caret symbol is located after *metho* part, then the entire line is divided into the sequence of components with names "$VarA$" and "*metho*". The matching algorithm begins by matching the initial search scope with the name of the first component. When matched, a new search scope is obtained from the "$VarA$" token reference. The same procedure is performed with the new scope and the second component. This matching routing runs continuously until the last component is matched. Finally, the code completion list is populated with elements from the search scope retrieved from the last component.

1.9.2 Visual Studio Code

Visual Studio Code is a source code editor developed by Microsoft for Windows, Linux, and macOS operating systems and positioned as a "lightweight" code editor for cross-platform development. It includes a debugger [48], tools for working with Git [49], syntax highlighting, IntelliSense [50], and refactoring tools. Visual studio code is an open source project and released under the MIT License [51]. This source code editor can be used to develop applications using various programming languages, including Java, JavaScript, C ++, and Node.js. Visual Studio code was announced by Microsoft at the Build 2015 conference on April 29, 2015. Later that year, it was released under the MIT license [52].

Visual Studio Code has plugin support available through a central repository, called the Visual Studio Marketplace. The capabilities and features of the IDE can be expanded using these plugins, for example, support additional programming languages, using the Language Server Protocol [53], or static code analyzers.

VSCode Programmatic Language Features are a suite of source code editing features based on vscode.languages.* API [54]. Visual Studio Code offers two common ways to provide these features [55].

- VSCode provides an interface (API) with available methods that register providers of any of the VSCode Programmatic Language Features.
- An alternative way is to implement a language server and client that use the language server protocol. For a better understanding of this approach, an example from the [55] is given. When a user hovers over a certain code in the source code editor, VS Code informs the client about it. Then the language client sends a request to the language server and receives back the hover result, which is propagated to the VS code.

The majority of the supported languages in the Visual Studio Code are implemented through language servers that use language services to provide features such as code completion, syntax highlighting, and error recognition.

Meanwhile, these services utilize the corresponding compilers to analyze the source code and obtain all the necessary information.

Due to poor documentation of the design and implementation details of the syntax analysis process, which is required to provide code completion and error recognition, and lack of examples in such compilers, there is no description of the techniques used to provide these features in VSCode. However, the particular parts of the source code of compilers were examined during the realization of the solution.

The following section describes the design of the syntax highlighting feature and basic details of IntelliSense in the Visual Studio Code. All documentation related to implementation details, techniques, and approaches described below is available at [56, 54].

Syntax highlighting in Visual Studio code establishes the color and style of the code written in the source code editor. Firstly, the text is divided into a list of tokens and scopes. Then a theme is used to map tokens and scopes to predefined colors and styles.

TextMate grammars are used by the VS code to break a source code from the editor into a list of tokens. **TextMate** is one of the most popular text editors for developers using the Mac OS operating system. It allows users to create their custom complex syntax highlighting modes using a modified version of the Apple ASCII property list format to define language grammars, see [57, 58]. TextMate grammars are used to denote elements of the source code, such as keywords, comments, lines, annotations or similar. The primary purpose of this is to enable syntax highlighting and help find the context in which the caret is located. The syntax highlighting colors are defined for each token, in the TextMate themes files [57].

Code completion(IntelliSense) features are provided by the language service based on analysis of the source code and language semantics. IntelliSense is triggered by typing Ctrl+Space or when you enter a trigger character, which depends based on language.

IntelliSense, in the VSCode, offers a wide variety of completion types:

- Methods and Functions
- Variables
- Classes
- Interfaces
- Fields
- Properties
- Enumerations

- Keywords
- References
- Colors
- Snippet Prefixes
- Words

1.9.3 Web-IDEs

This section provides a list of web-based integrated development environments.

Monaco editor is a browser based source code editor that powers Visual Studio code. It is released under the MIT License and supports Chrome, Safari, Classic Edge, Edge and Opera browsers.

Eclipse Theia is an integrated development environment framework for web-based and desktop application. It is open-source and free software project that was released under the Eclipse Foundation and licensed under the Eclipse Public License 2.0 [59].

Codemirror is a JavaScript text editor for the browser. It has a rich programming API and supports customization of advanced code editing functionality. CodeMirror is an open-source project released under an MIT license and is compatible with Chrome, FireFox, and Safari.

1.10 Available tools to implement IDE features

The goal is to support DET scripting language, which is a small subset of the Java programming language with custom extensions. Thus, there is no need to build a complex compiler for this language.

There are many different approaches and tools available to accomplish this task. In the next section, I present tools that are publicly available and widely used to analyze and parse a language.

1.10.1 Parser Generator ANTLR

Tools such as parsers and lexers are used to process user input from a text editor. In this section, I present a detailed description of ANTLR tool.

ANTLR (ANother Tool for Language Recognition) is an efficient parser generator used to parse structured text. It is used commonly for developing programming languages, tools, and frameworks. Initially, ANTLR was created for the Java language, but today, a parser can also be created for C#, Python 2 and 3, JavaScript, Typescript, Go, C++ and Swift programming languages.

ANTLR takes the grammar specified in an EBNF-like syntax as input and performs the following actions:

1. Generates a lexer that breaks the text into tokens(i.e., generates stream of tokens)
2. Generates a parser that creates an AST using token stream as an input.
3. Detects lexical and syntax errors (if present) in the grammar and syntax of the text.
4. Provides two ways of traversing parse tree in its runtime library: visitor and listener.

The following section describes the ANTLR tree walkers:

Listener ANTLR generates a listener interface that responds to events triggered by the built-in tree walker. The generated interface is unique for each grammar with enter and exit methods for each rule. The enter method for some rule X from the grammar is triggered when the walker encounters the node for this rule. As the walker visits all children of the rule X, it triggers exit method for that rule [60].

Visitor Similarly, ANTLR is able to generate a visitor interface from a specified grammar. For each rule in the grammar it generates corresponding method to walk a parse tree. Visitor mechanism allows us to control the walk itself, explicitly calling methods of each rule to visit children.

The biggest difference between listener and visitor strategies is that listener methods are called by the walker object provided by the ANTLR, while visitor methods via explicit visit calls.

1.10.1.1 ANTLR4 Code Completion Core

The **ANLTR4 Code Completion Core** (antlr4-c3) is a project that contains a grammar-independent code completion engine for ANTLR based parsers. This engine is useful for text editors since it is able to provide candidates for code completion regardless of the provided grammar or programming language. Initially, antlr4-c3 was implemented in the TypeScript programming language and released under the MIT License [61].

The following sections provide a brief description of existing tools and structures that generate software instruments to add support for a specific language.

1.10.2 Xtext

Xtext is a tool for development of textual programming languages [62]. It is an open-source project released under Eclipse Public License [59]. Based on grammar defined in EBNF-like notation, Xtext generates the following artifacts:

- a parser that reads the text and return an Eclipse Modeling Framework based AST.
- a compiler, type checker and linker
- an editing support for Eclipse and any editor that supports the Language Server Protocol.

Eclipse Modeling Framework (EMF) is a free modeling framework and code generation application for building applications based on a structured data model described.

1.10.3 textX

textX is a metalanguage framework for building Domain-Specific Languages in Python programming language. A metalanguage is a language used to define another language [63].

Based on a grammar description, textX automatically creates a metamodel and a parser for the described language. The parser analyzes the expressions of the language and automatically builds a graph of Python objects corresponding to the metamodel. textX is inspired by Xtext, so it follows the syntax and semantics of Xtext

1.10.4 JetBrains MPS

JetBrains MetaProgrammingSystem (**MPS**) is a tool to design languages. It uses projection editor, which allows users to create language editors, with the support of non-textual elements such as math notations, diagrams, and forms [64]. MetaProgrammingSystem is an environment for language definition and integrated development environment (IDE) for such languages [65].

Chapter 2

Current State

The current source code editor used for the DET scripting language is implemented as a component of the desktop user interface application, which is developed in the C# programming language. Supported features of this editor are: syntax highlighting and autocompletion. Error recognition and indication are **not supported** in the current state. This chapter presents the design and approaches used in the current state and points out main the disadvantages.

2.1 Text editor

The current state is based on the AvalonEdit. It is a Windows Presentation Foundation based text editor [66]. WPF is a graphical system, developed by Microsoft, for building Windows-based applications with user interface capabilities [67]. AvalonEdit offers many options for displaying a text document, which is suitable for a code editor, where the style and color of the text depends on the syntax of the language.

2.2 Features of the text editor

The IntelliSense service (IntellisenseService.cs) component provides data to support syntax highlighting and code completion features in the text editor. It is implemented as part of the editor application. The service is created when the text editor is launched.

The DET language supports custom language extensions that are provided from the backend part. Whenever the IntelliSense service is instantiated, it requests the definitions of custom language extensions to provide additional support for them. The structure of language extension definition would be the following:

- Name

- Type
- Description
- Attributes

The following list represents all possible types:

- Function
- Procedure
- Field
- Notation
- Namespace
- Constant
- RuntimeFunction
- Session
- Template

The attributes of language extensions are defined only for functions, procedures, and RuntimeFunctions.

2.3 Syntax Highlighting

Syntax highlighting data for the DET language are defined via the definitions in the file, which would be loaded by the editor in runtime. In addition, an extra highlighting information is provided by the IntelliSense service to support custom extensions.

The AvalonEdit editor uses regular expressions to process text and colorize words. It requests the data in the following format:

- text
- color info

where the **text** is used to generate a regular expression to find all the words from the source code of the editor that match the specified text, and **color info** defines the color and font size of the matched text.

The file lists all language metadata constructs such as a font size and color name. For example, the data to support syntax highlighting for all available primitive types from the DET language is defined in the format shown in Listing 2.1.

```
1 boolean boolVar = false;
2 double doubleVar = 1.0;
3 int intVar = 1;
4 short shortVar = 0;
5 long longVar = 2;
6 float floatVar = 0.1;
7 byte byteVar = 0;
8 char charVar = '0';
9 String stringVar = "string";
10
11
12
```

Figure 2.1: An example of using primitive types from the DET language in the AvalonEdit text editor.

Listing 2.1: The part of the text that defines the color and style of primitive data types.

```
<Color name="ValueTypes" foreground="Red" fontWeight="bold" />

<Keywords color="ValueTypes">
  <Word>boolean</Word>
  <Word>double</Word>
  <Word>int</Word>
  <Word>short</Word>
  <Word>long</Word>
  <Word>float</Word>
  <Word>byte</Word>
  <Word>char</Word>
  <Word>String</Word>
</Keywords>
```

The file format is determined by the syntax highlighting loader provided by the AvalonEdit editor. When a text editor is created, the loader retrieves the color and style definitions for the DET language. An example of syntax highlighting for primitive types is shown in Figure 2.1.

IntelliSense provides additional data to support syntax highlighting for custom language extensions. The data structure is the same as described above, that is, it has information about the name and color. It is collected from language extension definitions provided by the backend system.

The name field within the syntax highlighting info corresponds to the name of the language extension.

The color depends on the type of language extension. The mapping is visible from the following code snippet:

```
private static Color GetColor(LanguageExtensionType type)
{
    switch (type)
    {
        case LanguageExtensionType.RuntimeFunction: return
            Color.Orange;
        case LanguageExtensionType.Function: return Color.Green;
        case LanguageExtensionType.Procedure: return Color.Green;
        case LanguageExtensionType.Field: return Color.DarkMagenta;
        case LanguageExtensionType.Notation: return Color.DarkMagenta;
        case LanguageExtensionType.Namespace: return Color.DarkMagenta;
        case LanguageExtensionType.Constant: return Color.DarkMagenta;
        default: return Color.Black;
    }
}
```

2.4 Code completion

The code completion feature automatically gets triggered when the user starts typing in the text editor or press Ctrl+Space.

The code completion list is shown in a dedicated window provided by the AvalonEdit. In the following cases, the code completion list is empty, and therefore the completion window is not displayed:

- The caret position is inside the **comment** section.
- The last entered character is the **whitespace** character.

The structure of the completion data is provided by the AvalonEdit and has the following format:

- **text** is an entry in the completion window
- **type** defines an icon of the completion entry
- **description** is additional information that can be seen by pressing Ctrl+Space when any completion element is selected
- **priority** determines the order in which the elements are displayed in the window.

Initially, when the code completion gets invoked, the list is populated with completion items provided by the IntelliSense service. The structure of such items is listed below:

- Keywords

- Declared variables
- Custom language extensions

All available DET language **keywords** with corresponding descriptions are collected from a hard-coded list defined in an XML file.

IntelliSense service utilizes the VariableExtractionService.cs component to provide **declared variables**. This component parses the text from the editor with an algorithm that uses the following regular expression. This regular expression only detects declared variables with primitive data types and stores variable names.

```
(?:double|float|int|boolean|String
|short|long|byte|char)\s+(?<var_name>\w+)
```

Figure 2.2: Representation of the regular expression used to match a declared variable from source code.

When the processing of the entire source code of the text editor is completed, the collected names are propagated to the IntelliSense service.

Custom language extensions are handled internally by the IntelliSense service. It bypasses all definitions and generates a list of completion items as follows:

- the **text** and **type** of the completion item correspond to the name and type of the language extension, respectively.
- the **priority** depends on the type of custom extension. The dependency is hard-coded in the IntelliSense service.
- the combination of attributes and the description of language extensions defines the **description** of the code completion item.

When all completion items are assembled, they are displayed in the completion window. Besides, if the window is open and the user continues to enter text, items in the completion list are sorted alphabetically by the word that the cursor points to. Moreover, the most suitable suggestion for the current word is highlighted, see Figure 2.3.

2.5 Key disadvantages of the current state

The design and approaches used in the existing script editor to implement the core IDE features that were discussed in this chapter have many drawbacks. This section lists critical flaws in the current state.

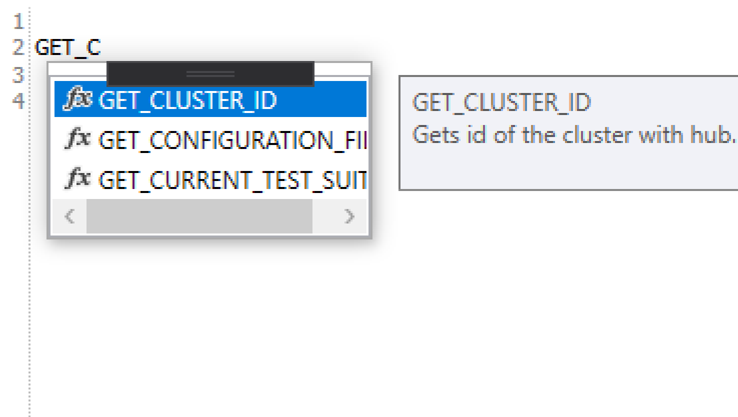


Figure 2.3: An example of the code completion list for the word GET_C.

- The text editor has no ability to recognize and display errors when the user enters a code.
- Hard to maintain and analyze the source code of the current state due to the lack of project documentation and poor source code style.
- The code completion list contains elements that when used generate errors in the script, that is, the search scopes according to the caret position are not supported. See the following example for a better understanding.

  In the example (see Figure 2.4), the completion list contains variable names that are not available at the current position of the caret symbol. variable_B is inaccessible after line 4, and variable_C is not declared and initialized.
- Only declared variables with primitive data types are supported in the code completion function. Thus, the completion list is not populated with the names of declared variables that have a custom class type.
- The code completion feature does not display the type of declared variables.

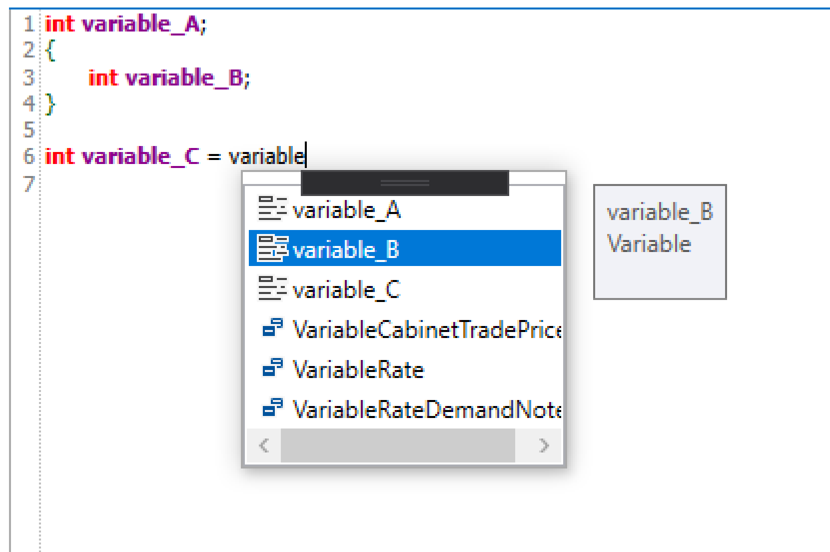


Figure 2.4: An example of the code completion list for the word variable.

Chapter **3**

Analysis and Design

The following section would analyze the main requirements of the project and suggest the solution design.

3.1 Web IDE

The existing editor is a part of an application developed in C#. Therefore it could not be used on any other platform rather than Windows. The backend part of the DET platform is Java based and ordinarily deployed onto Linux distributive systems. To prune the complexity of the deployment to end-user, the new IDE should platform-independent. Moreover, developing multiple applications for required systems would be inefficient in terms of maintainability. Web-based IDE would be supported by most Linux distributives out of the box, focusing the development on one application.

The web-based integrated development environment has many advantages over the desktop-based IDE which is used in the previous approach. In the next section, I introduce the key benefits of the web IDE. For a more detailed comparison of web-based and desktop-based editors see [44].

- **Installability** is a capability of the software product to be installed in a specified environment. It is perhaps the most notable advantage for web-based IDE. The idea is that the user would be able to log in through the browser and directly start writing code without installing anything (e.g., IDE itself, tools, extensions). Besides, the user would no longer need to install any updates, as they are applied in a central way.

- **Software mobility and portability (or accessibility)** is a software features that describes how easily this software can accessed in different environments. It is the second prominent advantage of a web-based solution over a desktop tool. The goal is that the IDE could be accessed from any device that has a browser.

3.2 Features of IDE

The primary IDE usecase is projecting business logic onto DET language scripts. The goals of the IDE to ease the process of writing the scripts and to provide useful interactive context information on the go.

As the user interferes with the central platform, custom-defined functions and constants are used in the script to reflect the idea of the message routing. The code completion feature should be implemented to give an opportunity of not remembering the whole identifier name. That would give the user the ability to enter only part of the identifier he remembers and then select the appropriate one from the list of suggested identifiers. The completion should not be performed for every possible token in the text, at least often encountered static constraints, and the identifiers should be supported. Some state of code completion feature was implemented in the initial version of IDE, which forces the feature to be present in the prototype, as this work considered to be an improvement.

A dedicated color could provide the context regarding the particular entry of the code. Highlighted code is much easier to perceive when compared with plain black letters text. Syntax highlighting feature should be implemented in the prototype and support at least all the highlighting which was present in the original editor. The highlighting should be configurable so that any color scheme could be used. Users should be able to define their own color schemes or use the existing ones if it satisfies their needs. Instant error feedback should be provided, meaning the error context should be presented to the user as the code is being written. On the other hand, there should not be context overloading; thus, the error description should be provided only when hovered on with the cursor. While the error is not in focus, it could be indicated with the underline. The error recognition and indication feature should be introduced in the prototype.

The following sections describe the design of code completion, syntax highlighting, and error recognition features of DET language integrated development environment.

3.2.1 Code completion

The approach chosen to implement the code completion feature is based on traversing an abstract syntax tree. ANTLR parses the text entered in the script editor and creates an abstract syntax tree that is processed to provide the necessary data for automatic code completion. The main points of processing an AST are listed below:

- Discover the context in which code completion was triggered, and collect all possible words that could follow the caret symbol.

- Collect the necessary information about the declared variables from the part of the text in the editor located before the caret symbol.. This information includes:
  - name of variable, to populate the completion list, if necessary
  - type of declared variable for being able to list all public fields and method of this variable
  - the search scope of each variable to find out if it can be accessed from the location indicated by the caret symbol

3.2.2 Syntax highlighting

The previous approach used to support syntax highlighting was implemented for the AvalonEdit editor, so it needs to be rewritten in order to be compatible with the Monaco editor. In addition, the generated ANTLR lexer can be utilized to implement syntax highlighting feature, so there is no need to process text using regular expressions.

The idea of implementing syntax highlighting feature is to use the ANTLR lexer, which performs lexical analysis of user input and divides the source code into tokens. Thus, style and color can be defined for each type of token and used in the Monaco editor to highlight text.

3.2.3 Error recognition and indication

The old script editor does not offer error recognition and indication while the user is typing the source code. Therefore, the approach to supporting this feature was developed from scratch.

The plan is to support lexical, syntactical and intuitive semantic errors. Error recognition design of the mentioned types of errors is provided in the section below.

- lexical and syntax errors in the source code can be detected using the listener interface, which is generated by ANTLR in accordance with the grammar of the DET language. The listener traverses through the abstract syntax tree and collects the information necessary to provide errors.
- to provide supported semantic errors, such as usage of undeclared variables, methods, or classes, and duplicate variable declarations, the AST produced by the ANTLR parser needs to be traversed. A tree walker that traverses the abstract syntax tree collects the information necessary to detect these types of errors.

When all errors are detected, the position of the error in the code and the error message are sent to the editor of Monaco to be able to highlight them.

3.3 Editor framework

The end-users should not be dealing with significant latencies over the input. Thus the solution should be dramatically improved in terms of processing speed and overall response time. The technologies of the web-based application also allow the solution to be light-weight and to require as few resources as possible.

Comparing the web-based IDEs such as Monaco editor, Eclipse Theia, and Codemirror, the differences were not significant for the current use case.

The chosen web-based integrated development environment is Monaco editor. The list of Monaco editor benefits is presented below:

- it provides an extensive list of features that are integrated into Visual Studio code
- Microsoft thoroughly supports it
- it has well described API documentation and many available examples, which simplifies the coding for this editor
- it provides a framework that allows implementing light-weight solutions

To support the required features of the DET language IDE, such as syntax highlighting, code completion and error recognition, proper lexical and syntax analysis needs to be performed.

Specially designed context-free grammar should be used to support Java-like languages, as was presented in [1]. The grammar would be relatively easy to maintain, unlike the regular expressions which make the base of the text processing for the previous implementation of the DET language editor.

The DET language has already been defined in DET Scripting Engine work [1] using ANTLR domain-specific language. Maintenance of several frameworks dealing with the definition and parsing of DET language would be cost-inefficient. Although most of the tools mentioned in section 1 can also provide custom language support for most IDEs, lexer, and parsers generated by defined ANTLR grammar would be utilized in this project.

ANTLR grammar provides tools to generate parser and lexer, that could be used to parse and analyze user input, to a wide variety of languages, including one used in this project. Moreover, the API of the provided tools is extensive and suits the goals of this project.

Chapter 4

Realisation

4.1 Environment

Like any program, this prototype project behavior depends on the input, which is composed of two main parts:

- User input, which is dynamic and can be accessed only during runtime
- Various static data, which is available at the compilation time. This data builds the environment of the prototype

The environment could also be divided into smaller parts. The following subsections present the main components of the environment and describe some of the implementation aspects.

4.1.1 Metadata of custom methods and variables

This project would not include communication with the backend systems to obtain the metadata for the DET language. For prototype purposes, the metadata would be provided as part of the source code. When the prototype would be integrated with the platform, a proper provider could implement the interface in order to receive data from the backend. The language extensions metadata for the DET language contains specific information:

- Name of the class where the language extension is implemented. It can be used in the code to access methods or fields of this class.
- Alias for the method or variable, which could be used as part of the DET language script
- The replacement for the alias, which would take place right before the compilation
- The return type of the extension method or type of the extension variable

- Number of required arguments, valid only if the extension is a method
- Method parameter types and other method information

The provided hardcoded metadata is generated only for testing purposes and does not reference any real extension methods nor variables.

4.1.2 Grammar

The grammar defined in ANTLR domain-specific language is available from the DET Scripting Engine project [1]. ANTLR framework is capable of generating TypeScript code based on the provided grammar. However, some of the rules in the provided grammar contain Java code inside their actions. TypeScript equivalent code should have been used before the lexer and parser could be generated. Some of the actions which were modified could be seen in Figure 4.1 Furthermore, the majority of the rules were transformed to be non-ambiguous. The reason for this is described in one of the following sections

The generated lexer and parser provide extensive API for parsing the DET language scripts into an abstract syntax tree for later traversal using the listener pattern.

4.1.3 Prototype web server

The prototype would be accessible as an HTML page, containing only the editor. The web server for the page is implemented as a standalone Kotlin application. When the solution would be integrated with the platform, a standalone web server for the editor would be obsolete, as it would be using the central platform web server. When the user opens the HTML page, the instance of the Monaco editor is being created. As Monaco requires language to be registered on the startup, DET language is being defined via Monaco API. This process could be divided into the following parts:

- Setting language configuration which defines the brackets and comments which are supported in DET language
- Setting styles and colors for language constructs
- Registration of the following IDE features providers that are implemented as editor components:
  - RaTokensProvider for dividing the text to tokens. These tokens used by the Monaco editor for syntax highlighting.
  - SuggestAdapter for providing code completions.
  - DiagnosticsAdapter is used for static code analysis. Each time the user edits text in the editor, this adapter checks the text for errors.

```
fragment JavaLetter:
   [a-zA-Z_] // these are the "java letters" below 0x7F
   | // covers all characters above 0x7F which are not a surrogate
   ~[\u0000-\u007F\uD800-\uDBFF]
       {Character.isJavaIdentifierStart(_input.LA(-1))}?
   | // covers UTF-16 surrogate pairs encodings for U+10000 to
       U+10FFFF
   [\uD800-\uDBFF] [\uDC00-\uDFFF]
       {Character.isJavaIdentifierStart(Character.toCodePoint((char)_input.LA(-2),
       (char)_input.LA(-1)))
      }?;

fragment JavaLetterOrDigit:
   [a-zA-Z0-9$_] // these are the "java letters or digits" below 0x7F
   | // covers all characters above 0x7F which are not a surrogate
   ~[\u0000-\u007F\uD800-\uDBFF]
       {Character.isJavaIdentifierPart(_input.LA(-1))}?
   | // covers UTF-16 surrogate pairs encodings for U+10000 to
       U+10FFFF
   [\uD800-\uDBFF] [\uDC00-\uDFFF]
       {Character.isJavaIdentifierPart(Character.toCodePoint((char)_input.LA(-2),
       (char)_input.LA(-1)))
      }?;
```

Figure 4.1: ANTLR grammar rules representing a digit or letter in a DET script.

Once the language and all components are registered, the editor would be displayed on the HTML page. An initial script would be presented for the instant demonstration.

4.2 Syntax highlighting

Being a feature-rich framework, Monaco editor provides syntax highlighting API. It requires two components: tokens provider and theme for each token type. As the code would be presented to a token provider, tokens would be requested as the output. Each token has the name and the starting position. The token provider for the prototype project is implemented in RaTokens-Provider class and internally uses the functionality of the lexer, which was generated by the ANTLR framework based on the DET language grammar. The lexer is parsing the input into a sequence of tokens that have all the necessary information for the Monaco API. The supported tokens for the syntax highlighting contain the following types:

- Digit
- Keyword
- String

The token type has its own theme, which consists of the font, style, and color. The themes should be registered via Monaco API for each token type.

4.3 Code completion

Code completion provider in the DET scripting language IDE utilizes a language service. The service is implemented as part of the editor and responsible for parsing the input of the user.

When code completion is triggered, the Monaco editor requests the provider for code completion information. The caret symbol location and the test are passed as part of the API request. The request is then propagated to the language service for the code completion list based on the text preceding the caret position. The language service utilizes the ANTLR4 code completion core to collect candidates that could follow the caret position. It requires the text and caret position and produces a collection of the candidates.

The ANTLR4-C3 is not capable of providing the code completion in case of syntax error before the caret position. Therefore, the language service sends the text of the last statement to isolate it from the rest of the source code written in the editor. In this case, errors that are present in the code before the last statement would not prevent the engine from providing all possible candidates. An AST processing provides the last statement from the source code. The AST is traversed by using the listener API. The listener API is part of the framework generated based on DET language grammar.

All the rules which are defined in the ANTLR grammar would be accessible via listener API using the dedicated methods. When the text is being parsed, there are two methods which are accessed at different times:

- On entering the rule
- On exiting the rule

If the enterStatement method is being invoked during text parsing, that would indicate that Statement rule is being used, and the rule's context can be accessed. In that particular case, the source code of the statement is saved. When the tree traversal ends, the code of the last statement is passed to the engine.

The ANTLR4 code completion core requires the following data:

- tokens that are ignored and not provided as completion items

- grammar rules that indicate the following types of completion items: variables, methods, and classes

The code completion core returns a candidate collection that contains fields for lexer tokens and parser rule indexes. Provided lexer token list consists of token ids which directly follow the given token in the grammar. The actual names of tokens are taken from the parser's vocabulary providing the token id. The names correspond to DET language keywords that could follow the current position of the caret symbol. An obtained parser rule indexes list is used to collect completions of variables, methods, and classes. For showing possible symbols in source code, symbol tables are used. They contain all available symbols at the caret position. The following section describes the realization of symbol tables.

4.3.1 Symbol table

The ANTLR4 code completion core provides symbol table implementation out of the box. The following symbol tables are utilized to provide the symbols:

- Global symbol table – stores information about custom extensions. They are loaded when the code completion feature gets invoked

- Editor symbol table – stores the information about all declared variables available at the current caret symbol position

As was described earlier, processing of each rule during text parsing could be monitored via listener API. That allows to collect all declared variables and save names and type into the symbol table. In addition, search scopes are supported, that is, when a declared variable becomes inaccessible to the current position of a caret symbol, it is removed from the symbol table.

4.3.2 Processing candidates

The parser rules provided by the engine are iterated. Each rule is processed depending on the rule index. The rule indexes represent the specific type of completion items, such as variable, method, and class. Depending on the type, the corresponding symbols are looked up in the symbol tables that were described earlier. Completion items are constructed from the found symbols and provided to the Monaco editor.

4.3.3 ANLTR error recovery mechanism

ANTLR Framework provides a default error handler that is able to recover from the errors. It would ignore the syntax errors and provide a valid AST tree. Moreover, there would be generated error reports, providing some context in

the form of the error position and possible reason. ANTLR also supports cascading errors; thus, a sequence of errors could be handled, not impacting the parsing of the rest of the text. There is a requirement for the ANTLR to support error processing in the given grammar. It would be able to successfully recover only in case the error occurred during the parsing of unambiguous rules. Thus the grammar should have been modified to avoid the possibility of the multiple next token choices from the moment where error can occur. The following techniques were used to alter the grammar:

- Left factorization. The definition follows:

$$A \rightarrow aa_1 \mid aa_2 \mid \ldots \mid aa_n$$

  is changed to

$$\begin{aligned} A &\rightarrow aA' \\ A' &\rightarrow a_1 \mid a_2 \mid \ldots \mid a_n \end{aligned}$$

- Corner substitution. The definition follows:

$$\begin{aligned} A &\rightarrow Ba \\ B &\rightarrow b_1 \mid b_2 \mid \ldots \mid b_m \end{aligned}$$

  is changed to

$$A \rightarrow b_1a \mid b_2a \mid \ldots \mid b_ma$$

Some of the least used rules were left unchanged and can be subject to the future error handling recovery improvement.

4.4 Error recognition and indication

Static code analysis was introduced in the DET language IDE to detect and indicate errors as a user types the code. Each time the user enters text into the editor, the diagnostic provider validates it and returns errors to the Monaco editor so that it can display them.

The ANTLR parser API allows the user to attach an error listener to handle syntactical and lexical errors. If the parser detects an error during text processing, the error listener would be notified. An error listener that collects these errors and propagates them to the user is implemented in the prototype. Other from that, the DET language IDE supports some minor semantic mistakes, such as usage of undeclared variables, methods, or classes and declaring duplicate variables. As was described earlier, the processing of

the AST is able to provide the context of all declared variables. This list is used to detect duplicate variable declaration or usage of an undeclared variable.

Whenever the listener is notified about a declaration of a variable(enterVariableDeclaration is called) during a traversal of the AST, it checks the editor symbol table for the symbol of type variable with the specified name. If the symbol table already contains this symbol and error is generated with the corresponding error message and stored to the list of semantic errors.

The listener's appropriate methods get called when the usage of the name of variable, method, or class is detected during a traversal of the AST. The symbol tables are searched for the symbol of the specified name and of the corresponding type(variable, method, or class). In case the symbol was not found, an error is stored in the list accompanied by the relevant message.

When all errors are collected, they are propagated to the Monaco editor that displays them as an underlined red line. Moreover, if the user hovers over the displayed error, the message of the error is shown.

Chapter 5

Testing

The primary purpose of the experiment was to test the overall usability of the prototype and the previous editor. The test results are composed of the feedback over both editors, enabling evaluation comparison of separate features. The test scenarios were designed to replicate the usual routine use cases of the DET platform, where internal functions and features of the language are commonly used alongside with standard programming tasks such as:

- Conditionals usage
- Accessing custom language extensions such as custom variables and methods
- Variables declarations inside and outside the scopes

The testing process consists of the following steps:

1. Implementation of the given task using the original IDE
2. Fulfilling the same task using new IDE prototype
3. Filling out a questionnaire regarding the advantages and disadvantages of the prototype and documenting the overall user experience with feedback on what might be improved in the future
4. Filling the assessment table of the DET language IDE capabilities. Evaluation is based on points from 0 to 10. See Figure 5.1 for the results. The result shows the overall superiority of the new IDE

All the testing was performed by the system specialists subdivision of the DET company. All the sixteen testers have roughly the same experience required for the experiment:

- Above 2000 lines of code written in DET language

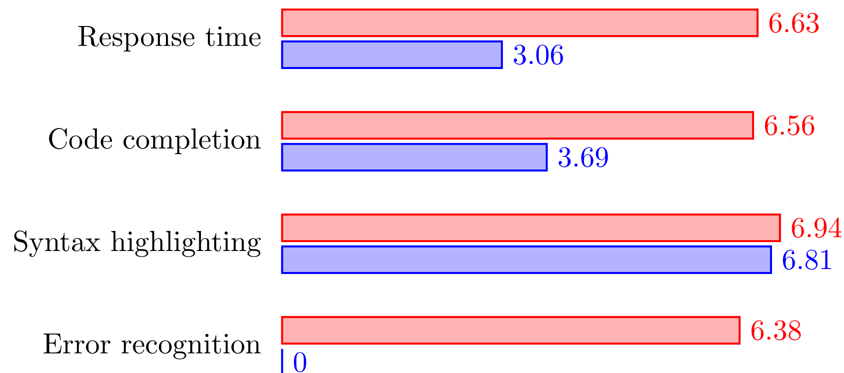


Figure 5.1: Mean features score. Blue bars represent original editor, while red bars represent prototype evaluations

- Long time experience with the original editor

These qualities are distributed evenly in the end-user groups as well, thus making the experiment close to a deployment environment.

The results of the experiment could be seen in Figure 5.1. The chart represents the mean subsequent evaluation among all of the testers. Both assessments, for the original editor and the prototype, are included in the table. The main observations of the experiment are the following:

- The error recognition and indication function in the prototype has reduced the overall script implementation time and was anticipated by the testers
- Syntax highlighting remained virtually on the same level
- Suggestions have become more natural and helpful in the context of the proposed task
- The average input response time has dramatically improved for the short scripts. However, in cases where the actual code length is more than approximately 60 lines, the latency starts to decrease. This flaw is insignificant since the vast majority of the scripts do not exceed 30 lines.

On the other hand, users have encountered a few problems introduced in the prototype. The following list describes the challenges and offers possible solutions which could be revisited in future iterations:

- Autocompletion introduces errors in the code in particular infrequent cases. Semantic error recognition should be improved to prevent the editor from suggesting incorrect code.

- Latency increases as the script take more than 140 lines of actual code. The implementation could be parallelized using threads in the form of web workers, which would decrease the load of the main thread.

- JAVA block code is out of support. More advanced support for Java constraints should be implemented. Moreover, it should be revisited with each new release of Java.

- Multiple errors in the code prevent the editor from suggesting any auto-completion. Error detection and isolation algorithm should be improved.

The tests were performed on the Windows platform as the previous implementation of DET IDE is not supported on any other platform. However, several participants insisted on the testing prototype on one of Linux distributives. The overall prototype user experience has been mostly evaluated as satisfying and pleasant under the condition of addressing confronted problems. The test showed the improvements and minor flaws of the prototype implementation, assuring the validity of the performed work.

Conclusion

The goal of this thesis was to implement an integrated development environment (IDE) for DET scripting language, which is a proprietary language based on Java. The IDE should be capable of syntax highlighting, code completion, and error recognition. Based on the research of modern solutions, use existing libraries and techniques to implement the IDE and its required features as a prototype.

The DET integrated development environment preserved features of previous IDE iteration alongside with modern design concepts. Analysis of design flaws of the original IDE revealed the main problem of processing performance and was addressed using contemporary solutions, resulting in much pleasant user experience. Many features were revisited and improved by switching to the proper lexer and parser, which provides more robust parsing, unlike the regular expressions approach. Along with revised features, new ones have been introduced that improve the user experience furthermore. One of those features is error recognition, which gives an ability to review errors before actual compilation. The web-based approach has introduced much better accessibility than the standalone Windows application. The Monaco editor, as the base component of the IDE, gave an opportunity to integrate new features seamlessly.

Future work

There are a couple of milestones to be achieved before the project could be integrated into the platform. Further work main points are derived from the user feedback during manual testing of the prototype:

- Error correction suggestions as an extension of the error context.
- Type errors which would indicate if an incorrect type is being used

- Integrated version control support, including annotations and file's change history

Besides the tester requests, there are certain ambitions to improve the user experience furthermore:

- Debug mode to improve the developing process and decrease time spent on finding the error
- Warnings support can help developers adhere to specific requirements in script implementation, preventing unpredictable behavior of the script or security issues.
- A powerful code completion engine that uses AI to significantly improve the quality of sentences.

Bibliography

[1] Grankin, D. *A translator of DET scripting language into Java.* Dissertation thesis, 2019. Available from: http://hdl.handle.net/10467/83386

[2] Hartley, R. Syntax diagrams. https://www.cs.nmsu.edu/~rth/cs/cs471/Syntax%20Module/diagrams.html, [Online; accessed 11-January-2020].

[3] Code Completion Design. [Online; accessed 01-February-2020]. Available from: http://wiki.codeblocks.org/index.php/Code_Completion_Design

[4] Lin, T. C. The new investor. *UCLA L. Rev.*, volume 60, 2012: p. 678.

[5] Hendershott, T.; Riordan, R.; et al. Algorithmic trading and information. *Manuscript, University of California, Berkeley*, 2009.

[6] Hope, C. What is Syntax? www.computerhope.com, [Online; accessed 17-January-2020].

[7] Melichar, B.; Holub, J.; Mužátko, P. *Languages and translations.* ČVUT, 1997.

[8] Barron, D. W.; et al. *Introduction to Programming Languages*, volume 7. CUP Archive, 1977.

[9] Hopcroft, J. E.; Motwani, R.; Ullman, J. D. Introduction to automata theory, languages, and computation. *Acm Sigact News*, volume 32, no. 1, 2001: pp. 60–65.

[10] Backus, J. W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. *Proceedings of the International Comference on Information Processing, 1959*, 1959.

[11] Peretti, O. COP 4555. https://www.coursehero.com/sitemap/schools/502-Florida-International-University/, [Online; accessed 10-January-2020].

[12] Scowen, R. S. Generic base standards. In *Proceedings 1993 Software Engineering Standards Symposium*, IEEE, 1993, pp. 25–34.

[13] Bender, E. A.; Williamson, S. G. *Lists, Decisions and Graphs*. S. Gill Williamson, 2010.

[14] Read, R. C. *Graph theory and computing*. Academic Press, 2014.

[15] Gross, J. L.; Yellen, J. *Handbook of graph theory*. CRC press, 2003.

[16] Brass, P. *Advanced data structures*, volume 193. Cambridge University Press Cambridge, 2008.

[17] Black, P. E. Dictionary of algorithms and data structures. Technical report, 1998.

[18] Morin, P. Data Structures for Strings. *PDF). Retrieved*, volume 15, 2012.

[19] Tzeng, H.-Y. Method for IP routing table look-up. May 9 2000, uS Patent 6,061,712.

[20] Korobov, M. Morphological analyzer and generator for Russian and Ukrainian languages. In *International Conference on Analysis of Images, Social Networks and Texts*, Springer, 2015, pp. 320–332.

[21] Love, R. *Linux kernel development*. Pearson Education, 2010.

[22] Tomassetti, G. A Guide to Parsing: Algorithms and Technology. https://dzone.com/articles/a-guide-to-parsing-algorithms-and-technology-part, [Online; accessed 10-January-2020].

[23] Nordquist, R. What Is Parsing? https://www.thoughtco.com/parsing-grammar-term-1691583, [Online; accessed 13-January-2020].

[24] Jones, J. Abstract syntax tree implementation idioms. In *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, 2003, p. 26.

[25] Chiswell, I.; Hodges, W. *Mathematical logic*, volume 3. OUP Oxford, 2007.

[26] Nguyen, B. *Linux Dictionary*. Binh Nguyen, 2003.

[27] IntelliJ, I. The Java IDE for Professional Developers by JetBrains. *URL: https://www. jetbrains. com/idea/(besucht am 19. 02. 2019)*, 2019.

[28] Randolph, N.; Gardner, D. *Professional visual studio 2008*. John Wiley & Sons, 2008.

[29] Computerwoche. Interaktives Programmieren als Systems-Schlager. https://www.computerwoche.de/a/interaktives-programmieren-als-systems-schlager,1205421, [Online; accessed 5-January-2020].

[30] Bruch, M.; Bodden, E.; Monperrus, M.; et al. IDE 2.0: collective intelligence in software development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 53–58.

[31] Kaufman, L. The Best Free Text Editors for Windows, Linux, and Mac. http://www.howtogeek.com/112385/the-best-free-text-editors-for-windows-and-linux/.

[32] d'Anjou, J.; Fairbrother, S.; Kehn, D.; et al. *The Java developer's guide to Eclipse*. Addison-Wesley Professional, 2005.

[33] Hansen, W. J. User engineering principles for interactive systems. In *Proceedings of the November 16-18, 1971, fall joint computer conference*, 1972, pp. 523–532.

[34] Group, M. S. KEDIT Language Definition Files. http://www.kedit.com/wwkld.html, 2012, [Online; accessed 20-January-2020].

[35] Klock, A. H.; Chodak, J. B. Syntax error correction method and apparatus. Oct. 14 1986, uS Patent 4,617,643.

[36] Jillre. Visual Studio IDE documentation. [Online; accessed 04-February-2020]. Available from: https://docs.microsoft.com/en-us/visualstudio/ide/?view=vs-2019

[37] Microsoft. Visual Basic 5.0 Control Creation Edition. https://archive.org/details/VB5CCE, [Online; accessed 7-January-2020].

[38] Asaduzzaman, M. *Context-Sensitive Code Completion*. Dissertation thesis, University of Saskatchewan, 2018.

[39] Kerievsky, J. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[40] Fowler, M.; et al. Refactoring: Improving the Design of Existing Code. 2000. *DOI= http://www. martinfowler. com/books. html/refactoring*, 2003.

[41] Zeil, S. J. Course CS350 Integrated Development Environments. [Online; accessed 15-January-2020]. Available from: https://www.cs.odu.edu/~zeil/cs350/f17/Public/IDEs/index.html

[42] LSP / LSIF. [Online; accessed 09-February-2020]. Available from: https://microsoft.github.io/language-server-protocol/

[43] Tunc, H.; Taddese, A.; Volgyesi, P.; et al. Web-based integrated development environment for event-driven applications. In *SoutheastCon 2016*, IEEE, 2016, pp. 1–8.

[44] Jonas Helming, M. K. Web-based vs. desktop-based Tools – EclipseSource. [Online; accessed 12-January-2020]. Available from: http://eclipsesource.com/blogs/2018/06/19/web-based-vs-desktop-based-tools/

[45] CodeBlocks. [Online; accessed 14-January-2020]. Available from: http://www.codeblocks.org/

[46] Scintilla and SciTE. [Online; accessed 16-January-2020]. Available from: https://www.scintilla.org/ScintillaHistory.html

[47] Scintilla Documentation. [Online; accessed 18-January-2020]. Available from: https://www.scintilla.org/ScintillaDoc.html

[48] Debugging in Visual Studio Code. Apr 2016, [Online; accessed 10-February-2020]. Available from: https://code.visualstudio.com/docs/editor/debugging

[49] Version Control in Visual Studio Code. Apr 2016, [Online; accessed 07-February-2020]. Available from: https://code.visualstudio.com/docs/editor/versioncontrol#_git-support

[50] IntelliSense in Visual Studio Code. Apr 2016, [Online; accessed 02-February-2020]. Available from: https://code.visualstudio.com/docs/editor/intellisense

[51] Stallman, R. M. Various licenses and comments about them. 2006, [Online; accessed 04-February-2020]. Available from: http://www.gnu.org/licenses/license-list

[52] Archived MSDN and TechNet Blogs. BUILD 2015: Visual Studio Code. [Online; accessed 10-February-2020]. Available from: https://docs.microsoft.com/en-us/archive/blogs/

[53] Language Server Extension Guide. Apr 2016, [Online; accessed 04-February-2020]. Available from: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide

[54] Extension API. Apr 2016, [Online; accessed 04-February-2020]. Available from: https://code.visualstudio.com/api

[55] Programmatic Language Features. Apr 2016, [Online; accessed 04-February-2020]. Available from: https://code.visualstudio.com/api/language-extensions/programmatic-language-features

[56] Documentation for Visual Studio Code. Apr 2016, [Online; accessed 11-February-2020]. Available from: https://code.visualstudio.com/docs

[57] Gray, J. E. *Textmate: Power Editing for Everyone*. Pragmatic Bookshelf, 2007.

[58] Anderson, F. *Step into Xcode: Mac OS X development*. Addison-Wesley Professional, 2006.

[59] Beaton, W.; d Rivieres, J. Eclipse platform technical overview. *Retrieved on November*, volume 2, 2006: p. 2009.

[60] Parr, T. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[61] Lischke, M. antlr4-c3 The ANTLR4 Code Completion Core. https://github.com/mike-lischke/antlr4-c3, 2019, [Online; accessed 04-January-2020].

[62] Efftinge, S.; Spoenemann, M. Xtext-language engineering made easy! 2016, [Online; accessed 15-February-2020].

[63] Dejanović, I.; Vaderna, R.; Milosavljević, G.; et al. TextX: a python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems*, volume 115, 2017: pp. 1–4.

[64] Pech, V.; Shatalin, A.; Voelter, M. JetBrains MPS as a tool for extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 2013, pp. 165–168.

[65] Campagne, F. *The MPS language workbench: volume I*, volume 1. Fabien Campagne, 2014.

[66] Table of Content. [Online; accessed 11-February-2020]. Available from: http://avalonedit.net/documentation/

[67] Nathan, A. *Windows presentation foundation unleashed*. Pearson Education, 2006.

Appendix A

Acronyms

GUI Graphical user interface

XML Extensible markup language

Appendix B

Contents of enclosed CD

```
readme.txt ............ the file with CD contents description
exe ................... the directory with executables
src ................... the directory of source codes
  wbdcm ............... implementation sources
  thesis .............. the directory of LaTeX source codes of the thesis
text .................. the thesis text directory
  thesis.pdf .......... the thesis text in PDF format
  thesis.ps ........... the thesis text in PS format
```