

Script Language Enhancements

Bachelor Thesis

Department of Computer Science
University of Applied Science Rapperswil

Spring Term 2018

Authors:	Michael Gerber & Victor Ruch
Advisor:	Prof. Dr. Luc Bläser
Project Partner:	Patrik Dudler <i>Eaton Automation GmbH</i>
External Co-Examiner:	Prof. Dr. Felix Friedrich <i>ETH Zürich</i>
Internal Co-Examiner:	Prof. Stefan Keller

1 TABLE OF CONTENTS

1	Table of Contents	2
2	Assignment	4
3	Abstract	8
4	Management Summary	9
4.1	Initial Position	9
4.2	Method	9
4.3	Results	9
4.4	Outlook	9
5	Introduction	10
5.1	Lexical Analysis	10
5.2	Syntactical Analysis	10
5.3	Code Generation	11
6	Methods	12
6.1	Initial Steps	12
6.2	Risk Assessment	12
6.3	Feature Analysis and Recommendations	14
6.4	Script function enhancement	15
6.5	Foreach	17
6.6	Range Assignment	17
6.7	Array and UDT Assignments	18
7	Implementation	21
7.1	General Implementation Details	21
7.2	elseif	21
7.3	Script Function Enhancements	22
7.4	Foreach	24
7.5	Range Assignment	26
7.6	+= and -=	28
7.7	Array assignment	29
7.8	User-Defined Type Instance Assignment	30
7.9	switch case	31
7.10	Ternary Operator	32
8	Results	33
8.1	Adjusted Galileo Script Language Definition	36
8.2	Code Metrics	38
9	Conclusion	39

10	Glossary	40
11	Bibliography.....	41
12	Appendix.....	42
12.1	EBNF Notation	42
12.2	Galileo Script language definition before our thesis.....	42
12.3	Tag System.....	43

Aufgabenstellung Bachelorarbeit für Michael Gerber und Victor Ruch

Script Language Erweiterungen für den Automationsbereich

1. Auftraggeber und Betreuer

Diese Bachelorarbeit findet in Zusammenarbeit mit *Eaton Automation GmbH* statt.

Ansprechpartner Auftraggeber:

- Patrik Dudler, Dipl. Ing. FH, Eaton Automation GmbH, Software Engineer,
PatrikDudler@eaton.com
- Michael Greminger, Dipl. Inf.-Ing. ETH, Eaton Automation GmbH, Leiter R&D,
MichaelGreminger@Eaton.com

Betreuer HSR:

- Prof. Dr. Luc Bläser, Institut für vernetzte Systeme, lblaeser@hsr.ch

2. Ausgangslage

(Die nachfolgende Beschreibung stammt grösstenteils von Eaton Automation.)

Eaton Automation bietet ein grafisches Programmiersystem namens Galileo, welches in der Automatisierungsbranche dazu eingesetzt wird, Benutzeroberflächen visuell zu entwickeln. Diese Benutzeroberfläche beinhaltet verschiedene grafische Komponenten wie Labels, Buttons, Listen und weitere. Mittels eines Compilers wird eine solche Benutzeroberfläche in ein binäres Format umgewandelt, welches schliesslich von einem Laufzeitsystem interpretiert wird, um das entsprechende User Interface auf dem Zielsystem anzuzeigen.

Galileo beinhaltet auch eine einfache Scripting Language, welche dazu dient, gewisse Abläufe zu programmieren. Der Umfang der Sprach-Features ist allerdings sehr limitiert. Es unterstützt beispielsweise „if-else“-Statements, mathematische Operationen, Wertzuweisungen und Systemfunktions-Aufrufe. Es fehlt jedoch an einigen gängigen Sprach-Features wie z.B. Return-Werten oder Parameter-Übergaben für Funktionen. Die Scripting Language soll daher um gängige Sprach-Features erweitert werden.

Die Grundlagen für den aktuellen grafischen Script-Editor mit Auto-Completion Funktionalität wurde als Teil einer Diplomarbeit im Jahre 2013 an der Fachhochschule Rapperswil erarbeitet und implementiert.

3. Ziele und Aufgabenstellung

(Die nachfolgende Beschreibung stammt grösstenteils von Eaton Automation.)

Das Ziel dieser Bachelorarbeit ist es, die Script Language von Galileo geeignet zu erweitern, um sie mächtiger und praktischer zu machen.

Die Arbeit besteht aus zwei Teilen:

Teil 1: Analyse

Der generelle Funktionsumfang der aktuellen Scripting Language soll analysiert werden und mögliche Erweiterungen bzw. Defizite sollen eruiert werden. Neben den folgenden offensichtlichen und als wichtig erachteten Sprach-Features gibt es bestimmt noch weitere:

- If-Statement mit Else-if Blöcke
- Return-Werte
- Parameter-Übergabe
- Lokale Funktions-Variablen

Teil 2: Implementation

Basierend auf Teil 1 werden eines oder mehrere Sprach-Features identifiziert, welche in Absprache mit dem Industriepartner konkret implementiert werden sollen. Die Umsetzung soll primär im grafischen Programmiersystem geschehen, bei Eignung und Interesse aber auch auf dem Laufzeitsystem.»

Zusammenfassung

Die Arbeit hat demnach folgende spezifische Ziele:

- Analyse der Galileo Script Language auf Erweiterungs- bzw. Verbesserungsbedarf.
- Entwurf konkreter Spracherweiterungen mit Berücksichtigung möglicher Folgen hinsichtlich Sprachkonsistenz, Nutzen und Implementation.
- Spezifikation der Syntax und Semantik der Spracherweiterungen, z.B. Syntax in EBNF.
- Design, Implementation und automatisierte Tests der Erweiterungen in der integrierten Entwicklungsumgebung von Galileo sowie bei Bedarf auch an Compiler und bei Interesse bzw. Bedarf auch am Laufzeitsystem
- Dokumentation der Spezifikation und Implementation der Spracherweiterung.
- Schlusspräsentation der Resultate beim Auftraggeber.

Die Implementation für das grafische Programmiersystem soll in .NET C# erfolgen. Die Implementation auf dem Zielsystem in C++.

4. Zur Durchführung

Mit dem HSR-Betreuer finden wöchentliche Besprechungen statt. Zusätzliche Besprechungen sind nach Bedarf durch die Studierenden zu veranlassen. Besprechungen mit dem Auftraggeber werden nach Bedarf durchgeführt.

Alle Besprechungen sind von den Studenten mit einer Traktandenliste vorzubereiten und die Ergebnisse in einem Protokoll zu dokumentieren, das dem Betreuer und dem Auftraggeber per E-Mail zugestellt wird.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. An Meilensteinen gemäss Projektplan sind einzelne Arbeitsergebnisse in vorläufigen Versionen abzugeben. Über die abgegebenen Arbeitsergebnisse erhalten die Studierenden ein vorläufiges Feedback. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Dokumentation.

5. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen. Die zu erstellenden Dokumente sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren. Die Dokumentation ist vollständig digital in 3 Exemplaren abzugeben. Auf Wunsch ist für den Auftraggeber und den Betreuer eine gedruckte Version zu erstellen.

6. Termine

Siehe auch Terminplan auf Skripteserver Informatik > Fachbereich > Bachelor-Arbeit_Informatik:

- | | |
|---------|---|
| 19.2.18 | Beginn der Bachelorarbeit, Ausgabe der Aufgabenstellung durch die Betreuer. |
| 8.6.18 | Die Studierenden geben den Abstract für die Diplomarbeitsschöpfung zur Kontrolle an ihren Betreuer/Examinator frei. Die Studierenden erhalten vorgängig vom Studiengangsekretariat die Aufforderung mit den Zugangsdaten zur Online-Erfassung des Abstracts im DAB-Tool. Die Studierenden senden per Email das A0-Poster zur Prüfung an ihren Examinator/Betreuer.
Vorlagen sowie eine ausführliche Anleitung betreffend Dokumentation stehen auf dem Skripteserver zur Verfügung. |
| 13.6.18 | Der Betreuer/Examinator gibt das Dokument mit dem korrekten und vollständigen Abstract für die Schöpfung zur Weiterverarbeitung an das Studiengangsekretariat frei.
Für die Ausstellung der Bachelorarbeiten das A0 Posters per Email bis 10.00 Uhr an das Studiengangsekretariat senden. |
| 15.6.18 | Hochladen aller verlangten Dokumente auf archiv-i.hsr.ch |
| 15.6.18 | Abgabe des Berichts an den Betreuer bis 12.00 Uhr. |
| 15.6.18 | Präsentation und Ausstellung der Bachelorarbeiten, 16 bis 20 Uhr |

7. Beurteilung

Eine erfolgreiche Bachelorarbeit zählt 12 ECTS-Punkte pro Studierenden. Für 1 ECTS Punkt ist eine Arbeitsleistung von ca. 25 bis 30 Stunden budgetiert.

Für die Beurteilung sind die HSR-Betreuer verantwortlich.

Gesichtspunkt	Gewicht
1. Organisation, Durchführung	1/6
2. Berichte (Abstract, Mgmt Summary, technischer u. persönliche Berichte) sowie Gliederung, Darstellung, Sprache der gesamten Dokumentation	1/6
3. Inhalt *)	(1/2)
3.1 Problemanalyse (Vorstudie, Literaturstudium, Anforderungsspezifikation, Anforderungsanalyse, Domainanalyse)	1/6
3.2 Lösungsentwurf (Lösungsvarianten und deren Beurteilung, Variantenentscheid, Konzept, Entwurf)	1/6
3.3 Realisierung und Test	1/6
4. Mündliche Prüfung zur Bachelorarbeit	1/6

*) Die Gliederung des Gesichtspunktes "3. Inhalt" in einzelne Unterpunkte kann den Gegebenheiten der Arbeit angepasst werden. Das Gesamtgewicht des Gesichtspunktes bleibt hingegen bei 50%.

Im Übrigen gelten die Bestimmungen der Abt. Informatik zur Durchführung von Bachelorarbeiten.

Rapperswil, den 8. Februar 2018

Der verantwortliche Dozent

Prof. Dr. Luc Bläser
Institut für vernetzte Systeme

Hochschule für Technik Rapperswil

3 ABSTRACT

Eaton, our business partner, manufactures automation control solutions that can be programmed with a custom scripting language called Galileo Script. Our task was to extend Galileo Script with modern language features to enable Galileo users to write more concise, well structured, intuitive and reusable code. This would require us to extend all the components within Galileo that turn a script into runnable byte code on the devices.

To this end, we analyzed the existing code base and gathered a list of language features that we considered to be major improvements for Galileo Script. We took into consideration time estimates for the implementation, compatibility with existing concepts and impacts on the runtime. We then compiled a list of recommendations including the necessary changes to the language definition and asked Eaton to prioritize them.

Once we knew their priorities, we proceeded to implement all the features that were important to our business partner. Examples of this would be parametrized functions, foreach loops and range assignments. To prove the functionality of our extensions we refactored the Galileo Demo Application using the new language features and wrote a small game within Galileo to showcase its new capabilities.

We are very happy with the improvements we were able to bring to Galileo Script and hope the hundreds of people using it will be too.

4 MANAGEMENT SUMMARY

4.1 INITIAL POSITION

Eaton, an international power management company, manufactures automation control solutions. Some of those automation solutions have displays with a customizable user interface. For customizing those panels, Eaton provides an in-house solution called Galileo. Galileo is a simple-to-use visual integrated development environment, that can be programmed using their own language Galileo Script. This language is designed to be very simple, such that it can be optimized to work on real-time and performance critical embedded systems.

4.2 METHOD

The aim of this thesis is the addition of new programming language features to the Galileo development system. It is split into two major parts: The first of which is analyzing the current state of the Galileo Script and its integration into the Galileo IDE. Based on our findings, we made recommendations for potential new features that are implementable in reasonable time and bring significant improvements to the users of Galileo. In the second part we implemented the features according to Eaton's priorities and the changes they requested.

4.3 RESULTS

We extended Galileo Script with these new features:

- Multiple "if-else" statements can be simplified with "elseif".
- Each script has its own set of local variables.
- Scripts can be used as functions supporting parameters with copy or reference passing semantics.
- Ranges allow assignment of a value to a slice of an array.
- Repetitive operations on arrays can be written with one "foreach" statement. It can also be used on just a slice of the array using the range syntax.
- Arrays or user-defined types can be copied to a tag of the same type using by a single assignment.

These improvements will enable Galileo users to write more concise, well structured, intuitive and reusable code.

4.4 OUTLOOK

A potential future work could address optimizations by altering the bytecode and virtual machine to support the new language features in a more efficient way. For example, the introduction of pointers for parameter references or array copies would be beneficial. We suggest separating the Galileo Script Compiler and the Galileo Integrated Developer Environment completely. This would simplify both the compiler and the IDE. We would also suggest the introduction of a code generation phase, to cope with different target platforms.

5 INTRODUCTION

The first step to extending the language was understanding the details of how Galileo turns a script into bytecode. The component that performs this transformation is called a compiler. Compilers are normally composed of multiple phases, as outlined here.

In this chapter we will explain those phases and mention some Galileo-specific details.

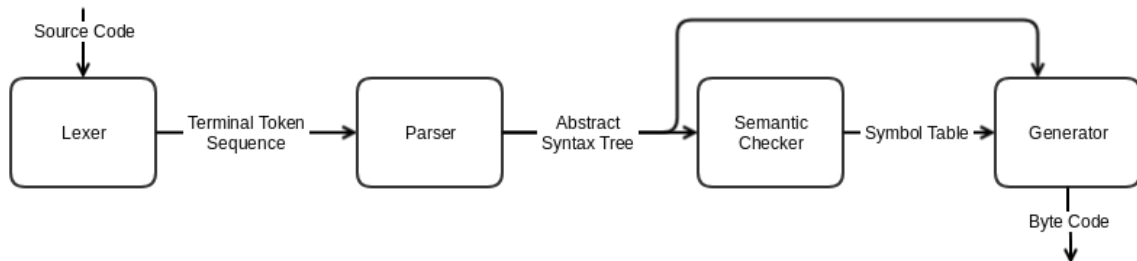


Figure 1: Compiler Phases

5.1 LEXICAL ANALYSIS

The lexer reads the code as a sequence of characters and transforms it to a sequence of terminal symbols¹. The lexer processes regular grammar.

For example:

```
if (num > 5000)
    num := -400;
endif
```

Figure 2: Lexer example of if-statement

This will be transformed into this terminal symbol sequence (token stream):

```
"if" "(" { "num": identifier } ">" {5000: integer-literal} ")" {"num":
identifier} "!=" {-400: integer-literal} ";" "endif"
```

Figure 3: Lexer example symbol tokens

The Galileo lexer does not represent whitespace characters as terminal symbols and only uses those as separators. By grouping the characters into terminal tokens, our parser will only need one lookahead token for building the syntax tree.

5.2 SYNTACTICAL ANALYSIS

In the syntactical analysis, we check if the grammar rules of the programming language are followed. Those rules are formalized in EBNF. In this phase, non-terminal symbols are built from the sequence of terminal symbols in order to generate the syntax tree. The compiler in the Galileo IDE uses a simple recursive descent top-down parser with one lookahead symbol (LL1).

For example, the code in Figure 2 would be parsed into a syntax node:

¹Terminal symbols (and non-terminal symbols) are described in the appendix: EBNF Notation

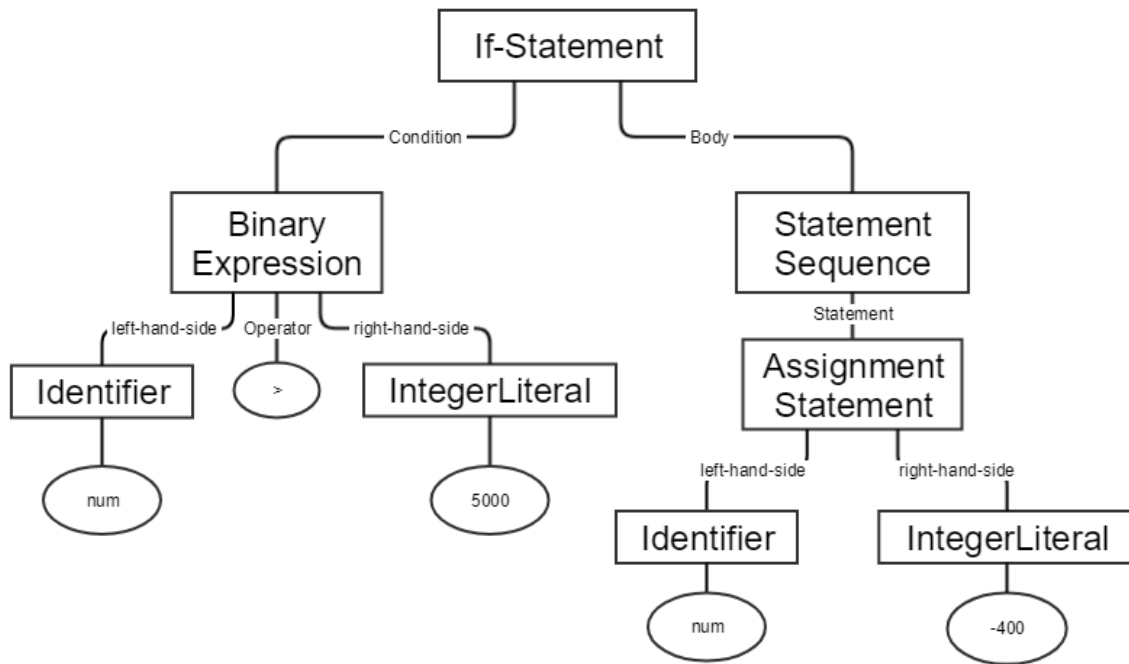


Figure 4: Syntax Tree of Source-Code in Figure 1

Note that this tree has been reduced to the most descriptive node types distinguishable by the parser to improve readability. For example, the right-hand-side of the assignment reduces from an Expression through AndExpression, BinaryExpression, SimpleExpression, Term, Factor and Literal to an Integer Literal. For details see chapter 8 Results.

5.3 CODE GENERATION

This is usually the last step of the compilation process. It takes the analyzed (and often optimized) abstract syntax tree and the symbol table and converts them into instructions that can be executed on the target platform.

For Galileo Script this means generating byte code for the Galileo specific runtime. This step is performed directly by Galileo's semantic checker.

6 METHODS

In this chapter, we describe the approach of this work.

6.1 INITIAL STEPS

We realized that in order to extend the language we were going to need a proper understanding of what was currently possible w.r.t. features and syntax. Hence, we first updated the EBNF² that our predecessors defined as part of their thesis³ in order to match the current syntax. This included, for example, the addition of arrays or the use of “:=” for assignment instead of “=” and the use of “=” for equality comparison instead of “==”.

Next, we split our efforts between making existing features testable and implement the easiest feature extension, that was also almost certain to be approved by Eaton (our business partner), in order to gain a deeper understanding of the status quo.

To gain confidence with the existing code base, we implemented the requested “elseif” feature. This learning-by-doing approach gave us a better understanding of the semantic rules. With this broader picture of the capabilities, we analyzed potential new features given by our business partner and suggested new ones too.

6.2 RISK ASSESSMENT

At the start of the project we collected a list of potential risks that are inherent to working with an existing codebase. The Table 1 shows estimated probabilities of occurrence and the impact on our chances of successfully completing our work. Each letter refers to a risk described in text below.

Table 1: Estimated probabilities of occurrence and the impact on our chances of success for each risk

PROBABILITY	IMPACT		
	Low	Medium	High
	High	A / C	
	Medium	B / E	D
Low			

6.2.1 A: Low codebase quality

Risk that the codebase might contain antipatterns, code duplications or breaks some boundaries of responsibility. This would make implementing new features and new tests difficult or in some cases impossible without major code refactoring.

6.2.1.1 Mitigation

We will do code reviews and automate tests as much as possible.

² 12.1 Appendix EBNF Notation

³ Leimgruber, Stefan and Stojkovic, Slobodan (2013) Guided Script Editor. Student Research Project thesis, HSR Hochschule für Technik Rapperswil.

6.2.1.2 What occurred

The codebase had grown considerably since 2013 and there were some classes that violated separation of concern practices which did impact the time it took to implement certain features.

6.2.2 B: Development environment is not reproducible

It might be impossible to set up a development environment, that is capable of running the Galileo software.

6.2.2.1 B.1: Version incompatibility of components

Visual Studio, Windows, Galileo or other third-party software are compatible with each other.

6.2.2.2 B.2: Missing dependencies

We might not receive all of the necessary dependencies to work on the Galileo software.

6.2.2.3 B.3: Other reasons

6.2.2.4 Mitigation

Request from Business Partner which prerequisites are needed. Ask them for advice on how to setup the environment.

6.2.2.5 What occurred

We received a complete codebase including all dependencies from Eaton and did not have to request additional resources. The project could be imported in Visual Studio 2017 and we were quickly able to build and run the project.

6.2.3 C: There is no or not enough automated tests for existing code

The existing codebase we receive will lack proper test coverage and extending it will therefore either require a lot of work to test existing parts or be very likely to break existing functionality.

6.2.3.1 Mitigation

We found no reasonable steps to take to mitigate this risk.

6.2.3.2 What occurred

There were very few automated tests for the existing code but this problem was partially mitigated by the Galileo project we received to test functionality and the fact that most of our code changes would be contained to the compiler classes.

6.2.4 D: Our extensions of existing code can not be automatically tested

Because the new features we will be implementing require direct integration in the existing codebase, the testability of our extension is dependent on the testability of the existing architecture. If the code we will be building upon is not written with the intent to be testable it will require us to either spend considerable amounts of time refactoring it or to forgo some or all automated testing.

6.2.4.1 Mitigation

We found no reasonable steps to take to mitigate this risk.

6.2.4.2 What occurred

Extending the lexer and parser was straightforward. Unit tests for the lexer and parser were available. The tests were self-contained and covered many edge cases.

The checker phase and code generation phase were by far the most demanding to extend. Both steps were implemented in a single class named Checker with little separation of concerns. Instead of a symbol table the class Tag Manager is used. The Tag Manager is a tree with all global variables, which is tightly integrated into to the user interface.

Initially, we wanted to automatically test the Checker class, but its coupling to other classes, like the Tag Manager, turned out to be too high. We evaluated the work needed to create good test mocks and fakes for the dependencies and concluded that it required too much effort and time to be done reasonably within the scope of this thesis.

6.2.5 E: Missing Language Definition

The current language definition is not well documented. We might break functionality without knowing it.

6.2.5.1 Mitigation

We requested a Galileo project from Eaton that is used by them internally to manually verify functionality and contains a lot of script examples. This helped us to recognize breaking changes early.

6.2.5.2 What occurred

The combination of the testing project and the existing (yet outdated) EBNF-Definition give us a reasonable degree of certainty that we did not miss any major language features in our implementation.

6.3 FEATURE ANALYSIS AND RECOMMENDATIONS

Once we had gained familiarity with the existing code base and capabilities of the current Galileo Script language, we started looking at the feasibility of potential extensions of the language. Eaton provided suggestions, of what they would like to see implemented: Adding an “else-if” construct and extending the function-like calling of one script from another by adding the ability to pass parameters, define local variables in it and return values to the calling script. In addition to analyzing if and how these requests could be implemented, we also explored other extensions of the language, that could be implemented in reasonable time, in order to improve Galileo-users’ experience.

A major constraint for our recommendations was that they needed to be backwards-compatible. This means that the new version of the Galileo Script compiler would allow new syntax for the additional features but would generate the same bytecode for existing scripts. Moreover we had to be careful not to break existing features like syntax highlighting or auto-completion in the IDE.

The following list summarizes our feature extension recommendations. The actual implementation of the approved features is discussed in chapter 7 Implementation.

6.3.1 elseif

The existing Galileo script only supported a simple if-else statement. This meant that implementing multiple mutually exclusive conditionals would require deep nesting by adding more if-else statements inside the else statements of the previous conditional. Most modern languages support some form of the “elseif” statement that allows the writer of the script to keep such conditionals to one level of nesting and consequently indentation.

This is the first feature we looked at, since it seemed simple enough to serve as a starting point to exploring the code. The only question left to answer was which keyword they preferred:

- elif
- elsif
- elseif
- else if

Syntax Example:

```

if (v = 0)
    countEqual := countEqual + 1;
elseif (v > 0)
    countGreater := countGreater + 1;
else
    countSmaller := countSmaller + 1;
endif

```

Figure 5: Else-If Example

6.4 SCRIPT FUNCTION ENHANCEMENT

Eaton requested the introduction of local tags, return values and passing parameters, which build upon each other. We discuss them separately, but we will introduce the concept of a script in Galileo here.

6.4.1 Background information

Galileo Script is similar to early procedural programming languages like Pascal. This influenced our decision when designing these features.

The Galileo Script language has two concepts of functions. There are predefined native functions called “special functions” and there are scripts the user has written, which we will refer to as “script” or “function”. There are three types of scripts: “event script”, “loop script” and “loop function script”.

The Galileo runtime is optimized for hard real time support on embedded systems. The runtime does not allow dynamic memory allocation and only uses the preallocated memory the compiler declared in advance. For the introduction of recursion and local variables in combination a stack would be required, but this would defy the concept of the Galileo runtime.

6.4.2 Overview of all three features working together

Syntax example of the final proposal:

```

param dividend: DWord,
    divisor: DWord,
    remain: ref DWord;
var times: DWord;
times := dividend / divisor;
remain := dividend - times * divisor;

```

Figure 6: Remain.scrp

```

cars := passengers / car_seats;
System.Script(Remain, passengers, car_seats, empty_seats);

```

Figure 7: FillAllSeats.scrp

A script file is always a single function. It requires all its parameter definitions as the first statement, followed by a second statement with all the local tag definitions. Parameters are passed by value or by reference when the tag is marked with the “ref” keyword.

“System.Script” executes a script. With backwards compatibility in mind, we did not want to introduce a new syntax of calling scripts, because the user would have two ways of executing scripts with and without parameters, which in our opinion does not seem intuitive. We recommended to extend the “System.Script” function to take additional arguments, which would be passed to the called script.

6.4.3 Local script variables

The first major restriction of Galileo Script is that there is no recursion. Instead of functions, there is the feature of calling one script from another script. Yet scripts may never be called recursively, and this was to stay this way.

The other design decision that we made in agreement with Eaton was that there would be no variable scoping inside of the existing block constructs (if, else) or the new ones we would be adding. This meant a variable would always either be scoped globally or inside an entire script (=function).

With these restrictions in mind and the fact that the language was statically typed, we recommended adding the keyword “var” which could be optionally used at the beginning of a script to specify a list of variables and their types. This meant forcing the separation of variable declaration and variable initialization.

Syntax Example:

```
var f: Float,  
    a: DWord;  
  
f := 3.0;  
a := 2*4;
```

Figure 8: Local tags

6.4.4 Script parameters

Similar to local variables there was a request to allow parameter passing from one script to another when calling it. Since scripts are defined inside the Galileo GUI and there is no traditional function keyword that could be extended with a signature, we saw two possible ways of defining the parameter names and types:

- Defining them in the Galileo GUI like global variables
- Defining them in one statement at the start of the function with a ‘param’ keyword.

For the second option we provided this syntax example:

```
param x: DWord,  
      y: DWord;  
var   dist2: DWord;  
  
dist2 := (x * x) + (y * y);
```

Figure 9: Parameter Example

6.4.5 Return value

If there is an option to pass parameters to a script it is reasonable to assume that there is also an option to pass results of that script back to the caller.

Therefore, the signature definition, we introduced for the script parameters feature, would have to be extended by an option to define what the script returns and what type(s) it has.

We found two options to achieve the feature of passing back values. One would be to have a return keyword as it is common in most modern languages. When we looked at the existing system functions, we realized that there was an alternative, that was more in line with how things were done in Galileo: Defining the behavior of script parameters to always be pass-by-reference instead of pass-by-value

which meant that values could be written back to the tags used as parameters. This would also force all parameters to be tags.

Eaton ultimately preferred a third option, that we initially considered to be too complicated for Galileo users: Adding a keyword to optionally define a script parameter as pass-by-reference (instead of the default pass-by-value).

6.5 FOREACH

Adding foreach loops is the first example of a feature that was not originally requested but we considered it to be a useful extension after looking through some of the Galileo Demo Scripts.

The language supports fixed-size arrays and it seemed obvious that there would be a desire to repeat a set of instructions on each element of those arrays. None of the existing features could significantly simplify this so we looked at some options that did. Ultimately, we recommend the addition of the foreach construct to perform all statements inside the block on each element of an array.

Syntax Example:

```
AlarmList.bNoAlarm := 0;
foreach (error in AlarmList.Errors)
  if (error = 1)
    Math.Inc(AlarmList.bNoAlarm);
  endif
endforeach
```

Figure 10: Foreach Example

Eaton agreed with our recommendation but requested to change the keywords from “for” and “endfor”, as we initially suggested, to “foreach” and “endforeach”.

6.6 RANGE ASSIGNMENT

Another feature that would help to reduce a lot of duplicate code was range assignment. It provides the ability to assign a subset of array elements denoted by a start and end index in one statement.

We presented four different syntax options for this feature. Some of them are more flexible, while others are simpler to understand.

Option 1 allows easy assignment of a sequence of numbers to an equally long range inside an array but is restricted to use with continuous sequences.

```
ary1[3..7] := 6..10;
```

Figure 11: Range Assignment Proposal 1

is translated to

```
ary1[3] := 6;
ary1[4] := 7;
ary1[5] := 8;
ary1[6] := 9;
ary1[7] := 10;
```

Figure 12: Unrolled Range Assignment Proposal 1

Option 2 allows complete control of what index gets assigned what number but at the same time it requires all values to be written out specifically.

```
ary1[3..7] := [4,8,3,5,4];
```

Figure 13: Range Assignment Proposal 2

is translated to

```
ary1[3] := 4;  
ary1[4] := 8;  
ary1[5] := 3;  
ary1[6] := 5;  
ary1[7] := 4;
```

Figure 14: Unrolled Range Assignment Proposal 2

Option 3.a allows initializing or resetting all values to one specific value but provides no improvements for cases where different values are required.

```
ary1[3..7] := 6;
```

Figure 15: Range Assignment Proposal 3.a

is translated to

```
ary1[3] := 6;  
ary1[4] := 6;  
ary1[5] := 6;  
ary1[6] := 6;  
ary1[7] := 6;
```

Figure 16: Unrolled Range Assignment Proposal 3.a

Option 3.b is inspired by languages like ruby and would make it clearer than 3.a that the value on the right of the assignment is duplicated for each element in the range.

```
ary1[3..7] := [6]*5;
```

Figure 17: Range Assignment Proposal 3.b

is translated to

```
ary1[3] := 6;  
ary1[4] := 6;  
ary1[5] := 6;  
ary1[6] := 6;  
ary1[7] := 6;
```

Figure 18: Unrolled Range Assignment Proposal 3.b

Eaton preferred solution 3.a, as this seems to be how their customers would like to use it.

6.7 ARRAY AND UDT ASSIGNMENTS

An additional request by Eaton was a feature that allows direct assignment of one array to another of the same type and size. Syntactically, it would be a simple assignment, but currently Galileo disallows

the use with arrays inside the semantic checker. They specifically wanted it to mean an assignment of each individual element in one array to the corresponding element in the other as opposed to pointing the variable left to the assignment to the same memory location. The result would behave more like array copy and would not result in multiple variables holding a reference to the same array.

Syntax Example:

```
ary2 := ary1;
```

Figure 19: Array and UDT Assignment

is translated to (assuming both arrays are of size 3 and have a start index of 0)

```
ary2[0] := ary1[0];  
ary2[1] := ary1[1];  
ary2[2] := ary1[2];
```

Figure 20: Unrolled Array and UDT Assignment

They also requested the ability to assign user-defined type (12.3.4.2 User-Defined Type) instances to other instances of the same UDT.

Syntax Example:

```
MotorInstance2 := MotorInstance1;
```

Figure 21: UDT Assignment Proposal

is translated to (assuming the UDT Motor definition from the Galileo Demo)

```
MotorInstance2.StateOnOff      := MotorInstance1.StateOnOff;  
MotorInstance2.Rpm             := MotorInstance1.Rpm;  
MotorInstance2.Temperature     := MotorInstance1.Temperature;  
MotorInstance2.Parameters[0]  := MotorInstance1.Parameters[0];  
MotorInstance2.Parameters[1]  := MotorInstance1.Parameters[1];  
MotorInstance2.Parameters[2]  := MotorInstance1.Parameters[2];  
MotorInstance2.Parameters[3]  := MotorInstance1.Parameters[3];
```

Figure 22: Unrolled UDT Assignment Proposal

6.7.1 Switch Case

While a lot of nesting issues could already be solved with the addition of the “elseif” keyword, we thought it might still be useful to have the switch case statement available.

Syntax Example:

```
case myInt:  
  0:   a := 1;  
  1:   a := 2;  
  2:   a := 3;  
  else: a := 4;  
endcase
```

Figure 23: Switch Case Proposal

Eaton agreed but gave the feature very low priority. Because of this it was ultimately not implemented in favor of other features.

6.7.2 Ternary Operator

The demo application we analyzed showed a lot of if-else statements that simply performed an assignment on the same variable with a different value. These statements could easily be reduced to one line with the introduction of the ternary operator.

Syntax Example:

```
AlarmList.Errors[0] := temp > 200 ? 1: 0;
```

Figure 24: Ternary Operator Proposal

Eaton regarded this feature as unimportant, since most of their customers would never use it.

7 IMPLEMENTATION

This chapter explains all the features we added to the Galileo scripting language with a brief description, the extension of the EBNF notation, interesting design decisions and a before and after example of a representative script inside the Galileo demo application.

7.1 GENERAL IMPLEMENTATION DETAILS

The preceding bachelor thesis⁴ introduced tests for the compiler, which were not migrated to the newest code base. Importing the tests for the lexer and parser required only minor code changes. The checker class originally generated an intermediate language but was adapted by Eaton since 2013 to directly generate bytecode and gather information for the project by interacting with the view components of the IDE. We tried in the first couple of weeks to adjust both checker and tests in a way that would enable us to verify existing functionality and add tests for our new features. While we did manage to extract a builder that would load lexer, parser and checker at once, we could not properly decouple the logic that used view components in order to make it testable. One example of this would be the TagManager class, that was required to define variables and was only initializable by the main view and had a two-way relationship with it. Another example is view panels that stored setting information that could not be easily extracted or faked in a testing environment. Because of this we were unable to define variables in scripts we wanted to test which severely restricted our ability to write tests for this layer. Ultimately, we decided not to include checker testing and instead opted to verify that the bytecode generated by the original version of Galileo and our version was the same.

7.2 ELSEIF

To simplify nested if-else statements, the elseif statement was requested.

7.2.1 EBNF Changes

7.2.1.1 Old

If-Statement = "if" "(" Expression ")" StatementSequence [Else-Statement] "endif"

Else-Statement = "else" StatementSequence

7.2.1.2 New

If-Statement = "if" "(" Expression ")" StatementSequence [Elseif-Statements]
[Else-Statement] "endif"

Elseif-Statements = "elseif" "(" Expression ")" StatementSequence [Elseif-Statements]

Else-Statement = "else" StatementSequence

7.2.2 Design Decisions

We had to decide on what keyword (or keywords) would be used. For this we looked at four of the most popular scripting languages and found four different designs.

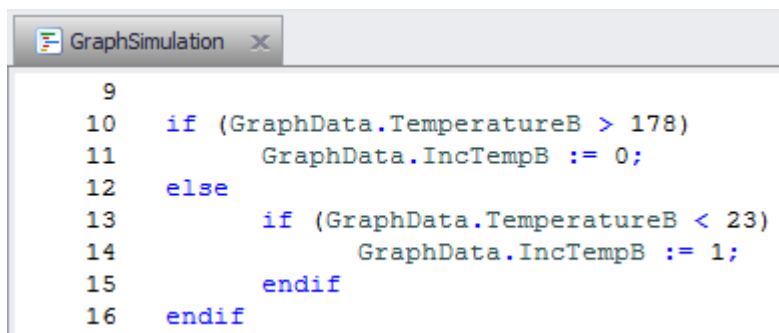
⁴ Leimgruber, Stefan and Stojkovic, Slobodan (2013) Guided Script Editor. Student Research Project thesis, HSR Hochschule für Technik Rapperswil.

Python: elif
PHP: elseif
JS: else if
Ruby: elsif

We relayed our findings to Eaton with our recommendation of “elsif”. They decided on “elseif”. Either option is fine in our opinion. Other languages must solve the “dangling else” problem, in this language there is no ambiguity due to the required “endif” keyword.

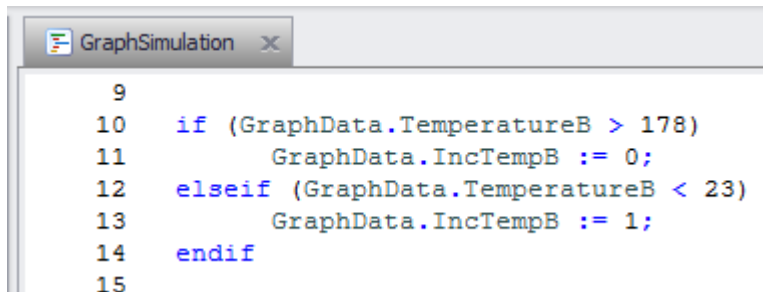
7.2.3 Demo Improvements

For each feature we described in this chapter, we have included snapshots of the official Galileo demo project that show how the Scripts can be improved using the new features. We start with a very simple rewrite using the elseif keyword. Note that the script produces identical bytecode before and after.



```
9
10  if (GraphData.TemperatureB > 178)
11      GraphData.IncTempB := 0;
12  else
13      if (GraphData.TemperatureB < 23)
14          GraphData.IncTempB := 1;
15      endif
16  endif
```

Figure 25: Without “elseif” Sample



```
9
10  if (GraphData.TemperatureB > 178)
11      GraphData.IncTempB := 0;
12  elseif (GraphData.TemperatureB < 23)
13      GraphData.IncTempB := 1;
14  endif
15
```

Figure 26: With “elseif” Sample

7.3 SCRIPT FUNCTION ENHANCEMENTS

Local variables (or local tags as they are called in Galileo) can only be used within the script that defined them. All local tags are default initialized and their lifetime is limited to the end of the script. By forcing all tags to be defined at the top of the script, we simplify the concept of scoping to two possible scopes: global tags and local tags.

All simple types (12.3.1 Simple Types), fixed length arrays of numeric types and user-defined types (12.3.4.2 User-Defined Type) are supported as local tags or as parameter. Numeric types are default initialized with 0. Strings and structs are not allowed as local tags or as parameters.

Similar to local tags all parameters must be defined before any other statement. The function “System.Script”, which is used to call another script, was extended to support calling scripts with arguments. For each parameter the callee defines the caller needs to specify an argument of a compatible type. In other words, there are no optional parameters.

Function parameters can be passed by reference or by value. Pass-by-value parameters is the default behavior and pass-by-reference parameters are marked by the “ref” keyword.

Since local tags and parameters share the same scope, their names must be unique within that scope. That scope is the local scope, in which names may be defined that hide names in the global scope. Local tags and parameters are both assignable, but only reference parameters have any effect outside of their function.

7.3.1 Design Decisions

For the purpose of hard real time support the Galileo runtime has neither a heap nor a stack. Therefore, we decided to implement local tags and parameters by saving them in two dedicated struct with the script name prefixed with “_locals_” or “_params_”. This approach was used by early Fortran languages, which like the Galileo runtime did not support recursions and only allocated memory during compile time.

Because the size of all local tags and parameter must be known at compile time, we decided to allow only fixed length arrays. Structs (12.3.4.1 Struct) are not allowed, since they are inherently global. Strings are not supported, due to the very limited string operations the runtime offers.

We chose working with the Tag Manager instead of introducing a new symbol table, because it would minimize the changes to the existing and untested code base located in the checker class, which would in turn reduce the risk of introducing new bugs. Refactoring the checker class to use a symbol table would have taken up most of the time we had for this project, which would have forced us to implement far fewer features.

The implementation of pass-by-reference turned out to be challenging. The virtual machine has no mechanism to dereference pointers, in order to be backward compatible we decided to copy all references to local variables before calling the script and copying them back to the original location. This method has impact on the performance and should be addressed on future versions of the Galileo runtime.

7.3.2 EBNF Changes

There can be only one “param” statement with all parameter definitions and one “var” statement with all local tag definitions for the script.

7.3.2.1 Old

Program = StatementSequence

7.3.2.2 New

Program = [ParameterDefinition] [LocalVarDefinition] StatementSequence

ParameterDefinition = “param” ParameterList “;”

ParameterList = Parameter { “,” Parameter }

Parameter = [“ref”] Identifier “:” Type

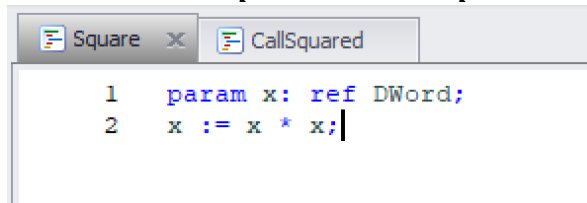
LocalDefinition = “var” LocalList “;”

LocalList = Local { “,” Local }

Local = Identifier “:” Type

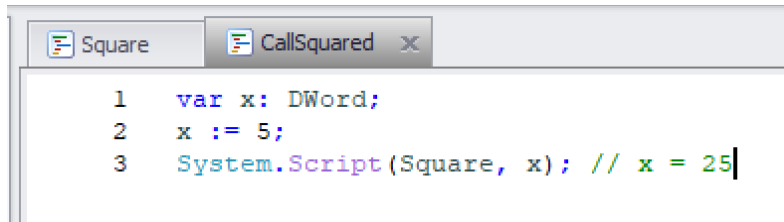
Type = Identifier [“[” IntegerLiteral “]”]

7.3.2.3 Demo Improvements Sample 1:



```
1 param x: ref DWord;
2 x := x * x;
```

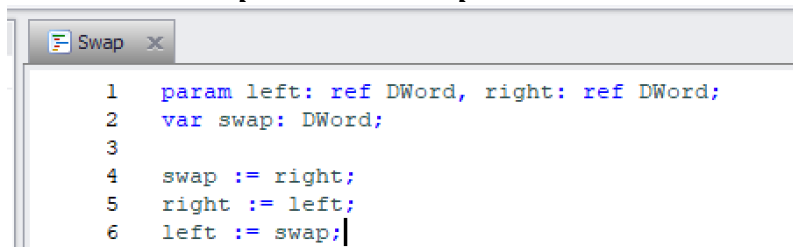
Figure 27: Square.scrp



```
1 var x: DWord;
2 x := 5;
3 System.Script(Square, x); // x = 25
```

Figure 28: CallSquare.scrp

7.3.2.4 Demo Improvements Sample 2:



```
1 param left: ref DWord, right: ref DWord;
2 var swap: DWord;
3
4 swap := right;
5 right := left;
6 left := swap;
```

Figure 29: Swap.scrp

7.4 FOREACH

The purpose of the new foreach statement is to give users the ability to define a statement sequence containing a static number of iteration steps that is performed on each element of an array (which is always of static length in Galileo Script).

7.4.1 EBNF Changes

We extended the existing EBNF by a new Foreach-Statement that can occur anywhere the existing Statements can. It is enclosed in the keywords “foreach” and “endforeach”. It can be used with any designator that denotes an array but the variable that holds the individual array elements during the loop must be an identifier.

7.4.1.1 Old

Statement	=	Assignment
		FunctionCallStatement
		If-Statement

7.4.1.2 New

Statement	=	Assignment
		FunctionCallStatement
		If-Statement
		Foreach-Statement
Foreach-Statement	=	"foreach" "(" Identifier "in" Designator ")" Statement-Sequence
		"endforeach"

7.4.2 Design Decisions

Galileo Script comes with an interesting set of limitations both in what the code base is capable of and in what Eaton wants to enable Galileo users to do. It is not meant for writing complex applications and often simplicity takes precedence over flexibility when it comes to language features. This is why the foreach statement we designed works a little different than one would expect from the way some popular languages implement it.

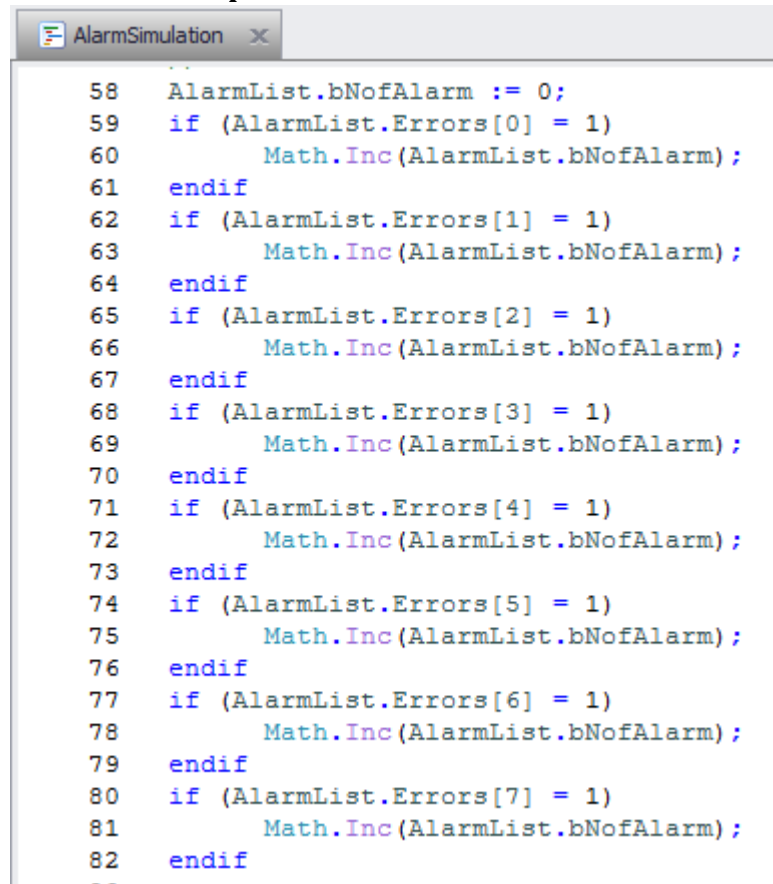
The first major difference is that there is, deliberately, no scoping. A variable remains defined the same before, inside and after an if statement for example. Local variables are always defined at the start of the script (see chapter 6.4.3 Local script variables). This means that defining variables dynamically to hold the array element the loop is currently on would lead to unintuitive behavior because the variable would still be defined after the loop ends. Therefore, we decided that this variable would also have to be declared at the start of the script so no confusion about variable shadowing was possible.

When it comes to the actual implementation we had to decide whether to extend the runtime with support of the foreach statement or to do an unrolling inside the code generator. The only reason this was even a question was the fact that arrays in Galileo Script are always of fixed size such that the number of sequence repetitions is known at compile time.

Ultimately, we decided to do loop unrolling to ensure backwards compatibility with the runtime, which would also make the features usable without having to update the runtime on the devices.

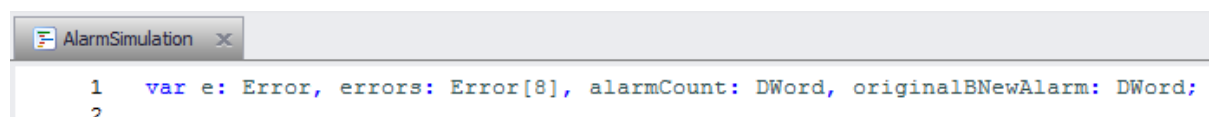
With this decision the implementation became syntactic sugar for repeatedly assigning each element of the array to the loop variable and performing the statement sequence inside the foreach loop.

7.4.3 Demo Improvements



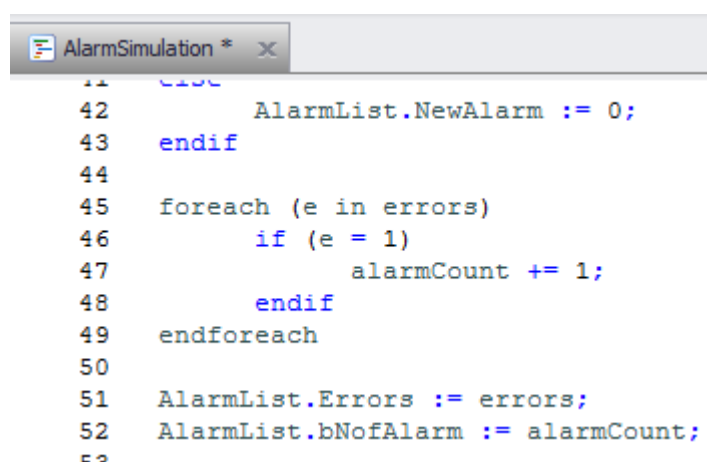
```
AlarmSimulation x
58 AlarmList.bNofAlarm := 0;
59 if (AlarmList.Errors[0] = 1)
60     Math.Inc(AlarmList.bNofAlarm);
61 endif
62 if (AlarmList.Errors[1] = 1)
63     Math.Inc(AlarmList.bNofAlarm);
64 endif
65 if (AlarmList.Errors[2] = 1)
66     Math.Inc(AlarmList.bNofAlarm);
67 endif
68 if (AlarmList.Errors[3] = 1)
69     Math.Inc(AlarmList.bNofAlarm);
70 endif
71 if (AlarmList.Errors[4] = 1)
72     Math.Inc(AlarmList.bNofAlarm);
73 endif
74 if (AlarmList.Errors[5] = 1)
75     Math.Inc(AlarmList.bNofAlarm);
76 endif
77 if (AlarmList.Errors[6] = 1)
78     Math.Inc(AlarmList.bNofAlarm);
79 endif
80 if (AlarmList.Errors[7] = 1)
81     Math.Inc(AlarmList.bNofAlarm);
82 endif
83
```

Figure 30: Before "foreach" Statement



```
AlarmSimulation x
1  var e: Error, errors: Error[8], alarmCount: DWord, originalBNewAlarm: DWord;
2
```

Figure 31: "foreach" Local Tag Definition



```
AlarmSimulation * x
42 AlarmList.NewAlarm := 0;
43 endif
44
45 foreach (e in errors)
46     if (e = 1)
47         alarmCount += 1;
48     endif
49 endforeach
50
51 AlarmList.Errors := errors;
52 AlarmList.bNofAlarm := alarmCount;
53
```

Figure 32: "foreach" Statement

7.5 RANGE ASSIGNMENT

Many modern scripting languages like Python or Ruby have a concept of ranges that allows the user to access a subset of elements inside of an array defined by a start and stop index. There is a lot of

different possible implementations with varying degrees of uses they enable. Eaton had a very specific use case in mind and they wanted to keep it as simple as possible for the users (see the recommendations in chapter 6.6 Range Assignment).

7.5.1 EBNF Changes

7.5.1.1 Old

```
Designator    =    Identifier
                | Designator "." Identifier
                | Designator "[" IntegerLiteral "]"
```

7.5.1.2 New

```
Designator    =    Identifier
                | Designator "." Identifier
                | Designator "[" IntegerLiteral | Range "]"

Range          =    IntegerLiteral ".." IntegerLiteral
```

7.5.2 Design Decisions

The EBNF specification prevents a dynamic range definition. This is on purpose and lets us ensure validity of the indices at compile time. Because of this restriction, we were able to implement this feature as element assignment unrolling inside the parser. The parser checks that the start index of the range is not greater than the stop index and then generates assignment statements for each index in the range. The checker later ensures that those indices do in fact refer to valid locations inside the range of the array.

7.5.3 Demo Improvements

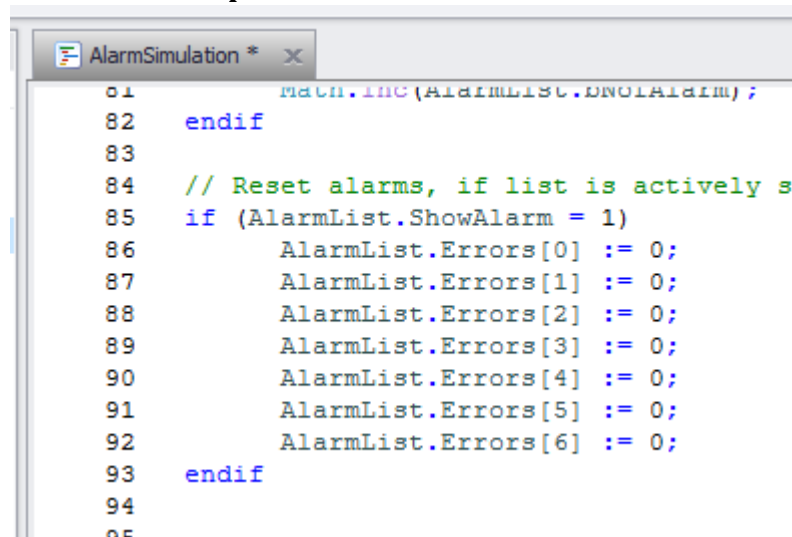


Figure 33: Before Range Assignment

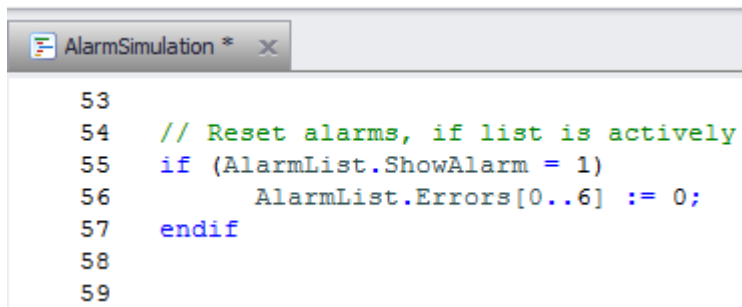


Figure 34: Range Assignment

7.6 += AND -=

This feature is simple syntactic sugar that is expanded into a conventional assignment that adds or subtracts the expression from the designator on the left. For example, “a += 1 * 2;” is transformed to “a := a + (1 * 2);”. We also added support to combine it with the range assignment feature such that values can easily be added or subtracted from a partial array. For example, “arr[0..2] += 1;” is unfolded into the source-code depicted in Figure 35: Unfolded ‘+=’:

```

arr[0] := arr[0] + 1;
arr[1] := arr[1] + 1;
arr[2] := arr[2] + 1;

```

Figure 35: Unfolded ‘+=’

7.6.1 EBNF Changes

7.6.1.1 Old

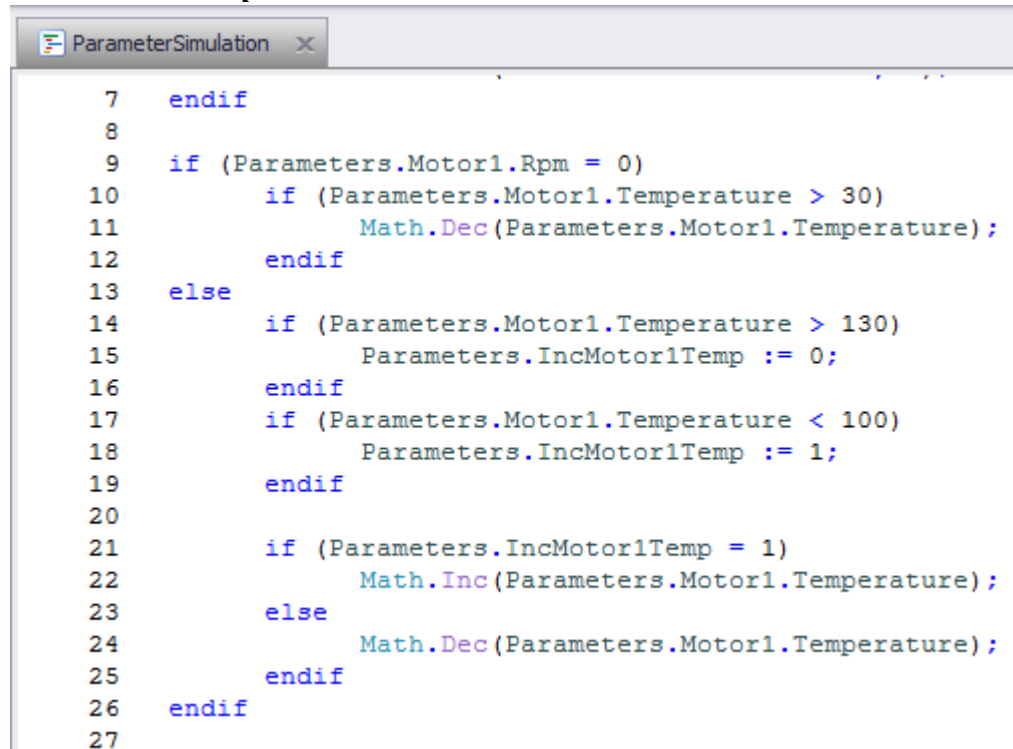
Assignment = Designator “:=” Expression “;”

7.6.1.2 New

Assignment = Designator Assignment-Operator Expression “;”

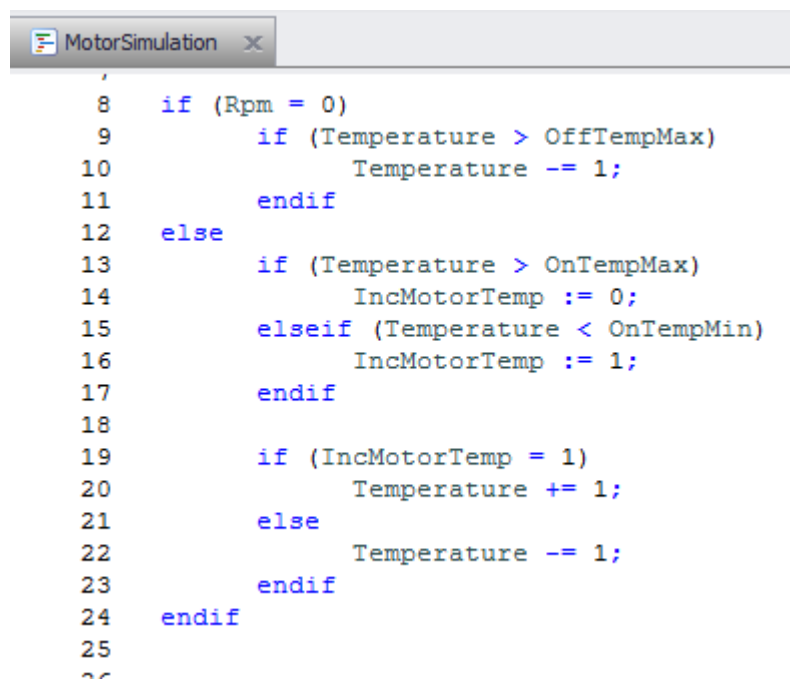
Assignment-Operator = “:=” | “+=” | “-=”

7.6.1.3 Demo Improvements



```
7     endif
8
9     if (Parameters.Motor1.Rpm == 0)
10         if (Parameters.Motor1.Temperature > 30)
11             Math.Dec(Parameters.Motor1.Temperature);
12         endif
13     else
14         if (Parameters.Motor1.Temperature > 130)
15             Parameters.IncMotor1Temp := 0;
16         endif
17         if (Parameters.Motor1.Temperature < 100)
18             Parameters.IncMotor1Temp := 1;
19         endif
20
21         if (Parameters.IncMotor1Temp == 1)
22             Math.Inc(Parameters.Motor1.Temperature);
23         else
24             Math.Dec(Parameters.Motor1.Temperature);
25         endif
26     endif
27
```

Figure 36: Before "+" and "-" Assignment



```
8     if (Rpm == 0)
9         if (Temperature > OffTempMax)
10             Temperature -= 1;
11         endif
12     else
13         if (Temperature > OnTempMax)
14             IncMotorTemp := 0;
15         elseif (Temperature < OnTempMin)
16             IncMotorTemp := 1;
17         endif
18
19         if (IncMotorTemp == 1)
20             Temperature += 1;
21         else
22             Temperature -= 1;
23         endif
24     endif
25
```

Figure 37: "+" and "-" Assignment

Note that the difference in script name and variable names comes from extracting this code into a parameterized script

7.7 ARRAY ASSIGNMENT

Similar to range assignment, we implemented this feature as an unrolling of one assignment statement into many individual statements, one for each element of the two arrays. But unlike the range

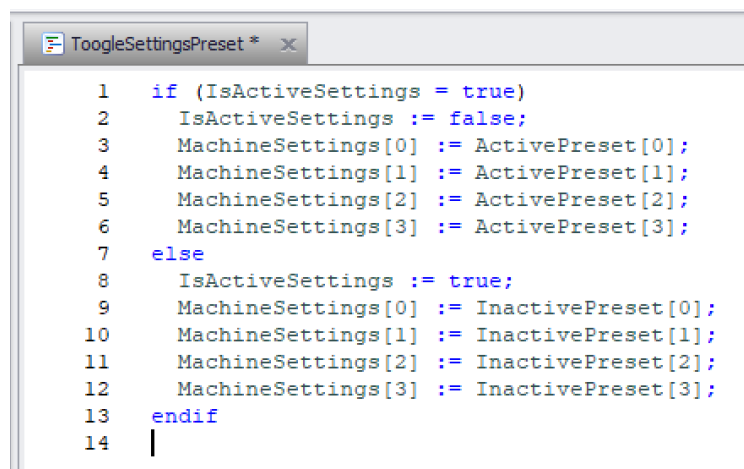
assignment the unrolling could not happen inside the parser because type and size information of the array should only be available inside the checker step. This lead us to a solution in which we create multiple bytecode assignment statements inside the checker after verifying the types and sizes of the two arrays are the same.

7.7.1 EBNF Changes

No changes to lexer or parser were necessary and, therefore, the EBNF definition also remained the same.

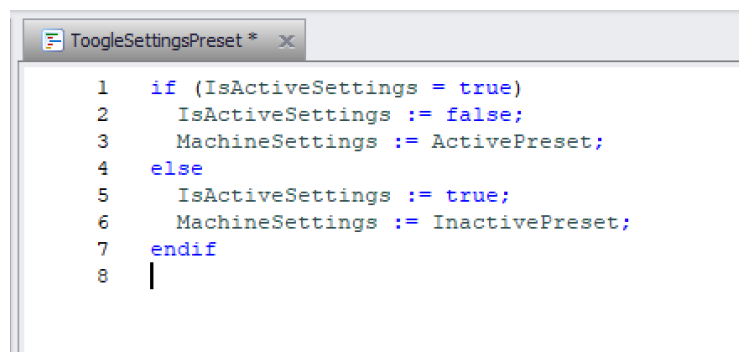
7.7.2 Demo Improvements

Unfortunately, the official demo application contains no script code that can reasonably be refactored to use this feature, instead we provide our own. The addition of array assignment allows a use case, where settings are stored as values of an array, that can be used to apply those settings in one statement.



```
1  if (IsActiveSettings = true)
2    IsActiveSettings := false;
3    MachineSettings[0] := ActivePreset[0];
4    MachineSettings[1] := ActivePreset[1];
5    MachineSettings[2] := ActivePreset[2];
6    MachineSettings[3] := ActivePreset[3];
7  else
8    IsActiveSettings := true;
9    MachineSettings[0] := InactivePreset[0];
10   MachineSettings[1] := InactivePreset[1];
11   MachineSettings[2] := InactivePreset[2];
12   MachineSettings[3] := InactivePreset[3];
13 endif
14 |
```

Figure 38: Before Array Assignments



```
1  if (IsActiveSettings = true)
2    IsActiveSettings := false;
3    MachineSettings := ActivePreset;
4  else
5    IsActiveSettings := true;
6    MachineSettings := InactivePreset;
7  endif
8  |
```

Figure 39: Array assignments

7.8 USER-DEFINED TYPE INSTANCE ASSIGNMENT

Eaton wanted this feature to behave like a deep copy of one UDT instance to the other. Because of this it would behave very similarly to the array assignment in that we would traverse the type definition both instances needed to share and create a simple assignment if the property is of a basic type, unroll an array assignment if the property is an array or recursively process the property if it is itself a UDT instance. This assignment should also behave like copying values from one instance to the other. Since a UDT definition could include a property, that is of a type which is itself a UDTs, it would even be a deep copy from the UDT instance on the right-hand side of the assignment to the one on the left.

7.8.1 EBNF Changes

No changes to lexer or parser were necessary and, therefore, the EBNF definition also remained the same.

7.8.2 Demo Improvements

Unfortunately, the official demo application contains no script code that can reasonably be refactored to use this feature thus we forgo an example here. The chapter 8 Results will include examples using scripts of our own design.

7.9 SWITCH CASE

We did not implement this feature, because Eaton assigned it very low priority. However, we prepared the EBNF extension that we include for future reference.

7.9.1 EBNF Changes

7.9.1.1 Old

Assignment = Designator “:=” Expression “;”

7.9.1.2 New (only one statement per case)

Switch-Case = “case” Expression “:” Case-Statement {Case-Statements}
“else:” Statement “endcase”

Case-Statement = Literal | Designator “:” Statement “;”

7.9.1.3 New (alternative)

Switch-Case = “case” Expression “:” Case-Statement {Case-Statements}
“else:” Statement-Sequence “endcase”

Case-Statement = “when” Literal | Designator “:” Statement-Sequence

7.10 TERNARY OPERATOR

We did not implement this feature, because Eaton assigned it very low priority. However, we prepared the EBNF extension that we include for future reference.

7.10.1 EBNF Changes

7.10.1.1 *Old*

Assignment = Designator “:=” Expression “;”

7.10.1.2 *New (restrictive)*

Assignment = Designator “:=” (Expression | Ternary-Expression) “;”

Ternary-Expression = Expression “?” Expression “:” Expression ;

Expression = AndExpression { “or” AndExpression }

7.10.1.3 *New (unrestricted)*

Assignment = Designator “=” Conditional-Expression “;”

Conditional-Expression = Expression “?” Expression “:” Expression ;

Expression = (Conditional-Expression | Or-Expression);

Or-Expression = AndExpression { “or” AndExpression }

8 RESULTS

The primary result of our work is a set of new and powerful tools for Galileo users that enable them to write more concise, well-structured intuitive and reusable code. In the chapter implementation we showed glimpses of these improvements using the official demo application. Here we would like to give complete examples of what is now possible in Galileo.

To this end we have written a simple 2-D environment puzzle game called RobotGame. It makes extensive use of the new features and would be far more complex and verbose to implement without them. In this game the goal is to maneuver a robot to a trophy on the opposite end of the playing field. The robot can be accelerated in one of four directions which will gradually increase its velocity in that direction. When changing the acceleration direction, the robot will keep its current velocity and it will only slowly decay over time. To make it a bit more challenging there are also three enemy robots that will cause the player to lose if their robot touches any of them.

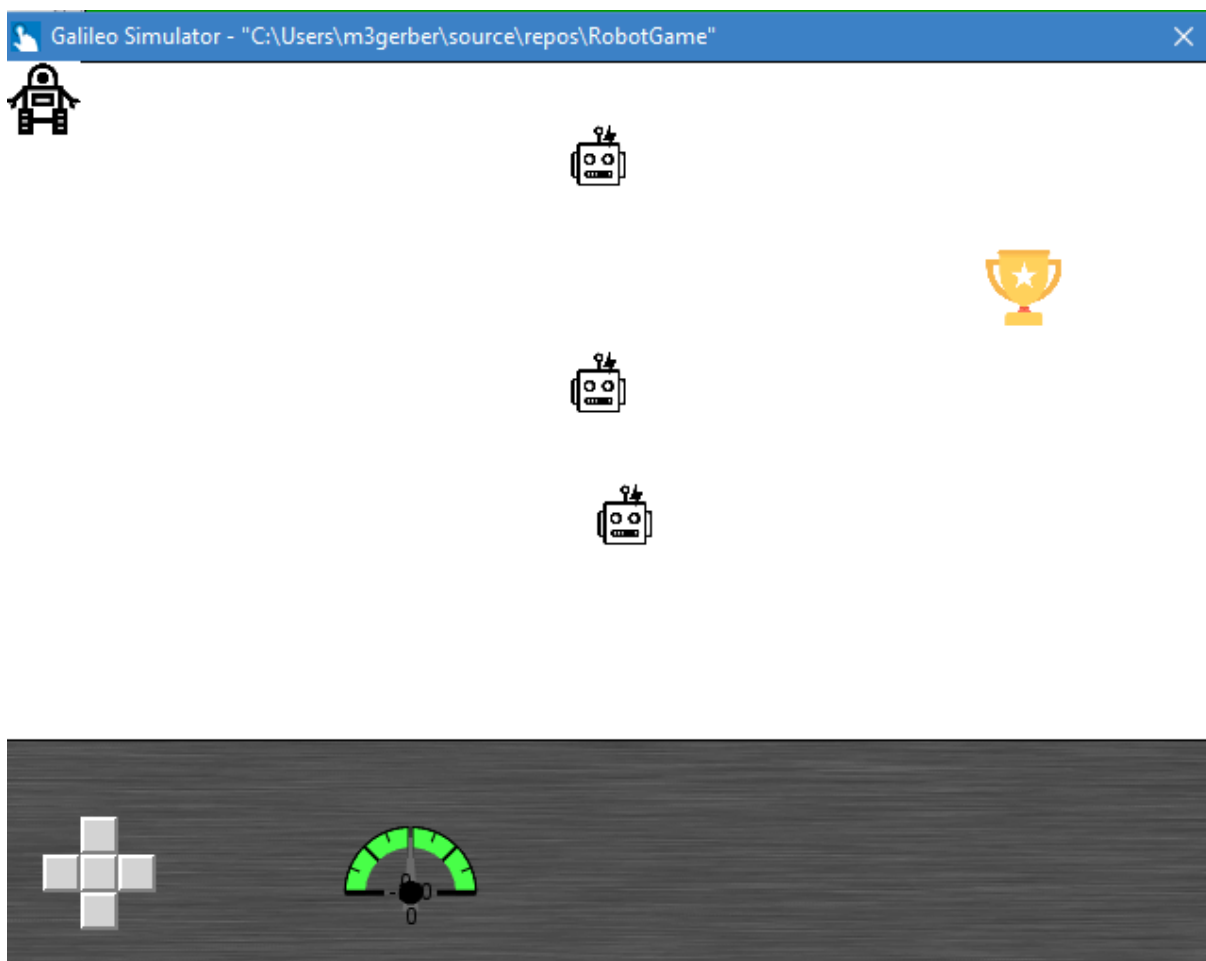


Figure 40: RobotGame Gameplay

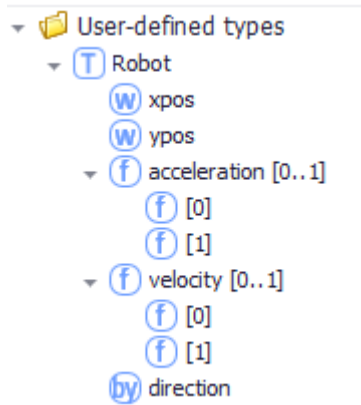


Figure 41: UDT Definition of Robot in TagTree

To represent the robots, we used a user defined type called Robot to store position, acceleration, velocity and direction of each robot. Then there are four loop scripts to monitor game state and move the robots. The first one is called MovePlayer and simply calls the shared movement function MoveRobot.

```
1 System.Script(MoveRobot, RobotInstancePlayer);
```

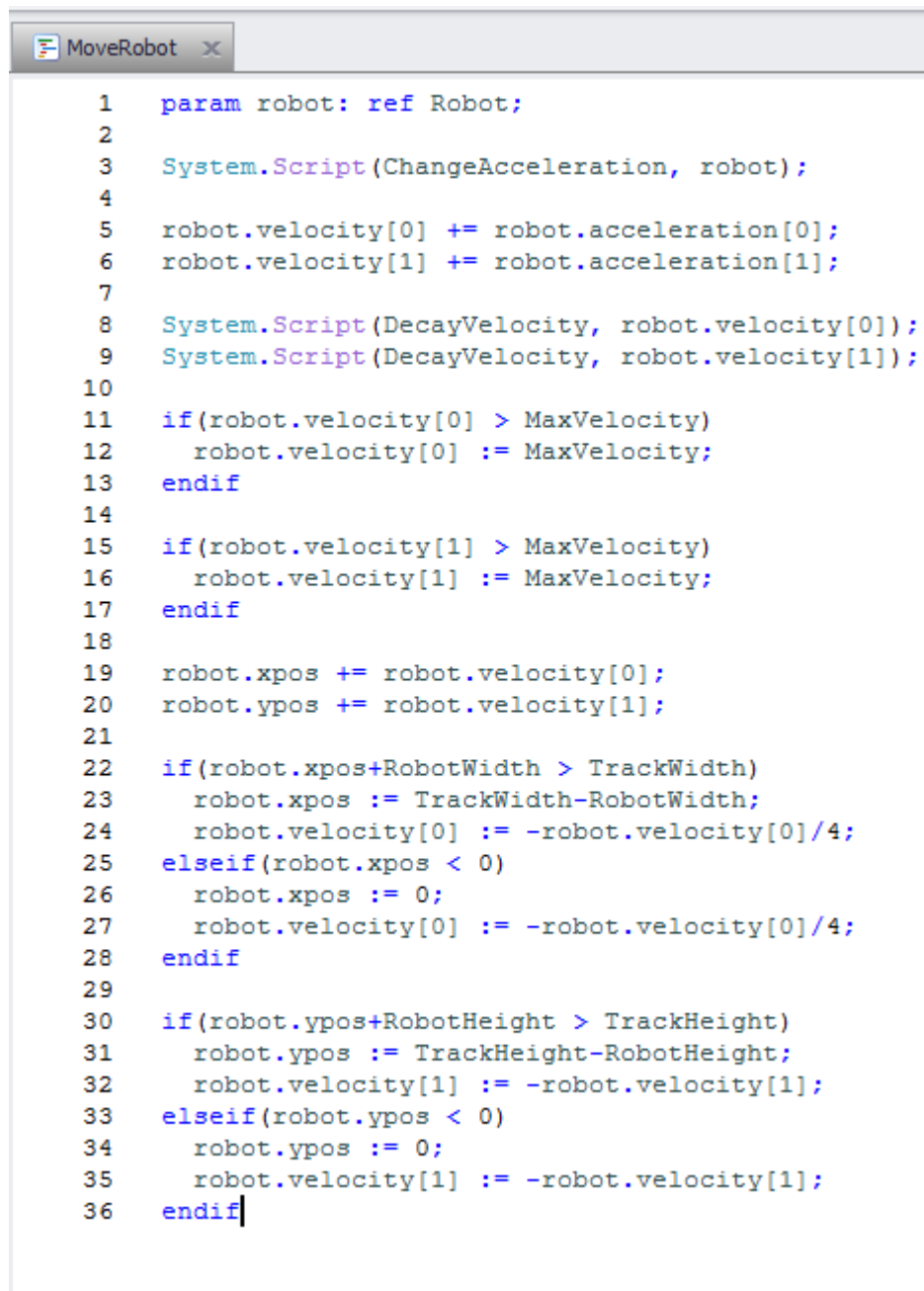
Figure 42: MovePlayer Loopscrip

The second loop script MoveEnemies uses the same movement function but uses a loop counter to decide which enemies get moved in what loop iteration. This way they move at different speeds.

```
1 var rand:Word, rest:Word;
2 AnimationCounter += 1;
3
4 rand := AnimationCounter;
5 rest := rand / 3;
6 rand -= rest*3;
7
8 if(rand > 1)
9   System.Script(ChooseBadRobotMove, BadRobot3);
10 endif
11 if(rand > 0)
12   System.Script(ChooseBadRobotMove, BadRobot2);
13 endif
14 System.Script(ChooseBadRobotMove, BadRobot1);
```

Figure 43: MoveEnemies Loopscrip

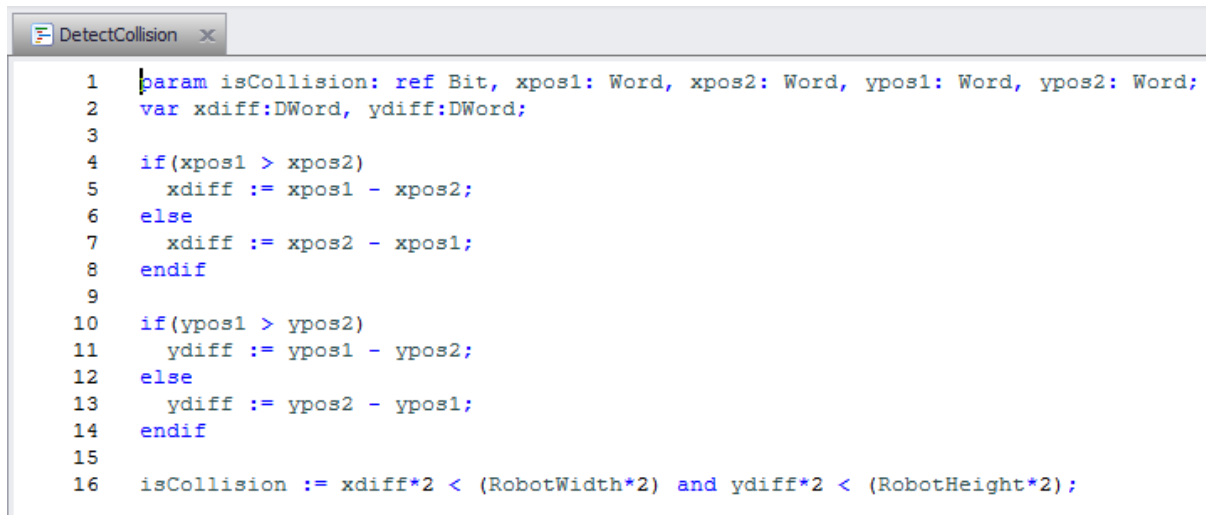
The movement script itself then does most of the work by changing the acceleration based on the direction the player has chosen, increasing the velocity based on the acceleration, decaying velocity to simulate friction and of course moving the robots position based on its velocity. In addition, it also checks if the robot has collided with the bounds of the track and stops it from moving beyond them by bouncing them off the wall (by reversing velocity).



```
1  param robot: ref Robot;
2
3  System.Script(ChangeAcceleration, robot);
4
5  robot.velocity[0] += robot.acceleration[0];
6  robot.velocity[1] += robot.acceleration[1];
7
8  System.Script(DecayVelocity, robot.velocity[0]);
9  System.Script(DecayVelocity, robot.velocity[1]);
10
11 if(robot.velocity[0] > MaxVelocity)
12     robot.velocity[0] := MaxVelocity;
13 endif
14
15 if(robot.velocity[1] > MaxVelocity)
16     robot.velocity[1] := MaxVelocity;
17 endif
18
19 robot.xpos += robot.velocity[0];
20 robot.ypos += robot.velocity[1];
21
22 if(robot.xpos+RobotWidth > TrackWidth)
23     robot.xpos := TrackWidth-RobotWidth;
24     robot.velocity[0] := -robot.velocity[0]/4;
25 elseif(robot.xpos < 0)
26     robot.xpos := 0;
27     robot.velocity[0] := -robot.velocity[0]/4;
28 endif
29
30 if(robot.ypos+RobotHeight > TrackHeight)
31     robot.ypos := TrackHeight-RobotHeight;
32     robot.velocity[1] := -robot.velocity[1];
33 elseif(robot.ypos < 0)
34     robot.ypos := 0;
35     robot.velocity[1] := -robot.velocity[1];
36 endif
```

Figure 44: MoveRobot Function for Loopscript

We also do very rudimentary collision detection between objects to check if the player has won or lost the game. This happens inside the DetectCollision function which makes use of both ref parameters and local variables for its calculations.



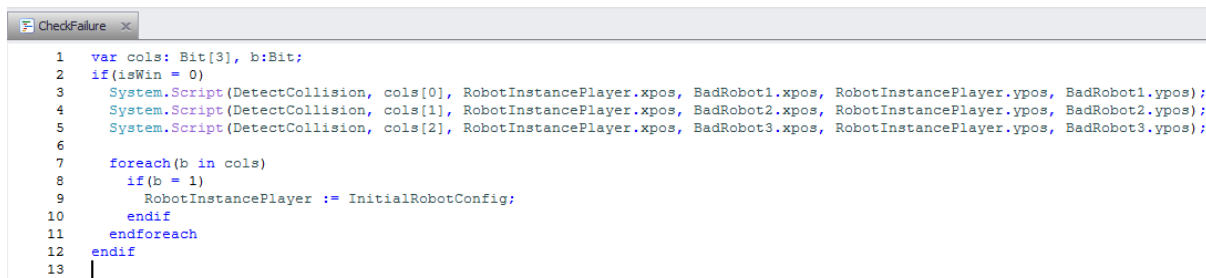
```

1  param isCollision: ref Bit, xpos1: Word, xpos2: Word, ypos1: Word, ypos2: Word;
2  var xdiff:DWord, ydiff:DWord;
3
4  if(xpos1 > xpos2)
5      xdiff := xpos1 - xpos2;
6  else
7      xdiff := xpos2 - xpos1;
8  endif
9
10 if(ypos1 > ypos2)
11     ydiff := ypos1 - ypos2;
12 else
13     ydiff := ypos2 - ypos1;
14 endif
15
16 isCollision := xdiff*2 < (RobotWidth*2) and ydiff*2 < (RobotHeight*2);

```

Figure 45: DetectCollision Function for Loopscrip

In the loop script CheckFailure we make use of the new foreach statement and UDT-assignment to reset the player state back to the original position, velocity, etc. if the players robot collided with any enemy robot.



```

1  var cols: Bit[3], b:Bit;
2  if(isWin = 0)
3      System.Script(DetectCollision, cols[0], RobotInstancePlayer.xpos, BadRobot1.xpos, RobotInstancePlayer.ypos, BadRobot1.ypos);
4      System.Script(DetectCollision, cols[1], RobotInstancePlayer.xpos, BadRobot2.xpos, RobotInstancePlayer.ypos, BadRobot2.ypos);
5      System.Script(DetectCollision, cols[2], RobotInstancePlayer.xpos, BadRobot3.xpos, RobotInstancePlayer.ypos, BadRobot3.ypos);
6
7      foreach(b in cols)
8          if(b = 1)
9              RobotInstancePlayer := InitialRobotConfig;
10         endif
11     endforeach
12 endif
13 |

```

Figure 46: CheckFailure Loopscrip

8.1 ADJUSTED GALILEO SCRIPT LANGUAGE DEFINITION

Another result of our work was the updated EBNF definition for Galileo Script. We include the complete version here. The proposed changes for a switch case statement and the ternary operator are not included since we did not implement them. We also include the initial version from when we started in the appendix for comparison.

Program	=	[ParameterDefinition] [LocalVarDefinition] StatementSequence
ParameterDefinition	=	"param" ParameterList ";"
ParameterList	=	Parameter { "," Parameter }
Parameter	=	["ref"] Identifier ":" Type
LocalDefinition	=	"var" LocalList ";"
LocalList	=	Local { "," Local }
Local	=	Identifier ":" Type
Type	=	Identifier ["[" IntegerLiteral "]"]

StatementSequence	=	{Statement}
Statement	=	Assignment FunctionCallStatement If-Statement Foreach-Statement
Foreach-Statement	=	“foreach” “(“ Identifier “in” Designator “)” Statement-Sequence “endforeach”
FunctionCallStatement	=	FunctionCall “;”
Assignment	=	Designator Assignment-Operator Expression “;”
Assignment-Operator	=	“:=” “+=” “-=”
Designator	=	Identifier Designator “.” Identifier Designator “[“ IntegerLiteral Range “]”
Range	=	IntegerLiteral “..” IntegerLiteral
Identifier	=	Character { Character Number }
Expression	=	AndExpression { “or” AndExpression }
AndExpression	=	BinaryExpression { “and” BinaryExpression }
BinaryExpression	=	SimpleExpression { ComparisonOperator SimpleExpression }
SimpleExpression	=	Term { (“+” “-”) Term }
Term	=	Factor { (“*” “/”) Factor }
Factor	=	Literal Designator (“not” “-”) Factor (“(“ Expression “)”) FunctionCall
Expressionlist	=	Expression { “,” Expression }
FunctionCall	=	Designator “(“ Expressionlist “)”
If-Statement	=	“if” “(“ Expression “)” StatementSequence [Elseif-Statements] [Else-Statement] “endif”
Elseif-Statements	=	“elseif” “(“ Expression “)” StatementSequence [Elseif-Statements]
Else-Statement	=	“else” StatementSequence
ComparisonOperator	=	“=” “>” “<” “>=” “<=” “<>”
Literal	=	IntegerLiteral FloatLiteral BoolLiteral

IntegerLiteral	=	Number
FloatLiteral	=	Number "." Number
BoolLiteral	=	"true" "false"
Number	=	Digit {Digit}
Digit	=	"0" ... "9"
Character	=	"a" ... "z" "A" ... "Z"

8.2 CODE METRICS

Table 2: Code Metrics concerning the final codebase

Number of files changed:	172
Number of lines added:	7026
Number of lines deleted:	1844
Number of automated tests restored:	72
Number of automated tests added:	17
Number of semi-automated tests added:	25

9 CONCLUSION

Our objective was improving the Galileo Scripting language in ways that enable Galileo users to write better code more easily. Our business partner communicated very clearly what improvements were important to them and we were able to implement all of them. We had to make some minor concessions in terms of syntactical aesthetics and performance but ultimately, we were able to deliver a version of Galileo Script that is a major improvement over the original. We consider this a clear success and are very happy with the result.

As with all programming languages, one can always argue for further improvements of the language. We would like to suggest some possibilities that we encountered during the implementation phase but did not deem more important than what we implemented. In terms of performance we would recommend adding runtime support for loops and references. The concept of ranges could be extended beyond the use in assignments and foreach loops to dynamically partition arrays for example.

From a software engineering standpoint, we would also recommend refactoring the checker so that code generation is done in a separate class and neither has direct references to UI elements. This would drastically reduce coupling and allow for automated testing.

10 GLOSSARY

Definition of terms in the context of the Galileo software solution.

Term	Definition
Tag	A tag refers to storage location in static global memory that may be paired with a symbol name
Internal Tag	Specific to a Galileo Project and defined by the user. Behave like global variables.
System Tag	Tags that are predefined by the system and cannot be changed by the user.
Tag Manager	A single tree that contains all tags in the system.
Lexer	Takes the raw character sequence of a Galileo Script and turns it into a sequence of symbols.
Parser	Takes the symbol sequence the lexer provides and turns it into a statement sequence.
Semantic Checker	Checks the statement sequence the parser provides for errors such as type violations or invalid array indices.
Generator	Turns the checked statement sequence into bytecode runnable by the Galileo runtime. Is directly integrated into the checker.
Galileo	Integrated development environment for user interfaces on embedded systems designed for ease of use. The runtime is optimized for hard real time performance on resource-limited embedded systems.
Screen	A graphical user interface container that occupies the whole screen.
Loop Script	A type of script definable in Galileo that is called in a set interval by the runtime.
Event Script	A type of script definable in Galileo that is called when the conditions the user defined are met.
UDT	Stands for User-Defined-Type. Detailed description in the Appendix (12.3.4.2 User-Defined-Types).
Struct	A type for a tag that allows the user to freely define property names and types in the scope of the struct tag. Similar to a JavaScript Object. Can only be defined in the Galileo GUI.

11 BIBLIOGRAPHY

Leimgruber, S., & Stojkovic, S. (2013). *Guided Script Editor. Student Research Project thesis*. Rapperswil: Hochschule für Technik Rapperswil.

Wirth, N. (2017). *Compiler Construction*. Zürich: ETH Zürich.

12 APPENDIX

12.1 EBNF NOTATION

EBNF extends BNF by:

	EBNF	Equivalent BNF
Repetition	$A = \{a\}$	$A = A'$ $A' = \varepsilon \mid a A'$
Optional	$A = [a]$	$A = \varepsilon \mid a$
Groupings	$A = (a b)c$	$A = ac \mid bc$

An EBNF expression without any non-terminal symbol can be used to express all regular grammars. BNF cannot define regular grammar in one expression, since it must use recursion for repetition.

For example, in EBNF, integer numbers can be defined in one expression:

Integer = {"0"..."9"}

In BNF one must rely on non-terminals to define some regular grammars:

Integer = "0" ... "9" | "0" ... "9" Integer

12.2 GALILEO SCRIPT LANGUAGE DEFINITION BEFORE OUR THESIS

This is the language definition we reconstructed by reverse engineering the code. It is based on the language definition of the SA from 2013⁵.

Program	=	StatementSequence
StatementSequence	=	{Statement}
Statement	=	Assignment FunctionCallStatement If-Statement
FunctionCallStatement	=	FunctionCall ";;"
Assignment	=	Designator "：=" Expression ";;"
Designator	=	Identifier Designator "." Identifier Designator "[" IntegerLiteral "]"

⁵ Leimgruber, Stefan and Stojkovic, Slobodan (2013) Guided Script Editor. Student Research Project thesis, HSR Hochschule für Technik Rapperswil.

Identifier	=	Character { Character Number }
Expression	=	AndExpression { "or" AndExpression }
AndExpression	=	BinaryExpression { "and" BinaryExpression }
BinaryExpression	=	SimpleExpression { ComparisonOperator SimpleExpression }
SimpleExpression	=	Term { ("+" "-") Term }
Term	=	Factor { ("*" "/") Factor }
Factor	=	Literal Designator ("not" "-") Factor) ("(" Expression ")") FunctionCall
Expressionlist	=	Expression { "," Expression }
FunctionCall	=	Designator "(" Expressionlist ")"
If-Statement	=	"if" "(" Expression ")" StatementSequence [Else-Statement] "endif"
Else-Statement	=	"else" StatementSequence
ComparisonOperator	=	"=" ">" "<" ">=" "<=" "<>"
Literal	=	IntegerLiteral FloatLiteral BoolLiteral
IntegerLiteral	=	Number
FloatLiteral	=	Number "." Number
BoolLiteral	=	"true" "false"
Number	=	Digit {Digit}
Digit	=	"0" ... "9"
Character	=	"a" ... "z" "A" ... "Z"

12.3 TAG SYSTEM

A tag refers to storage location in static global memory that may be paired with a symbol name.

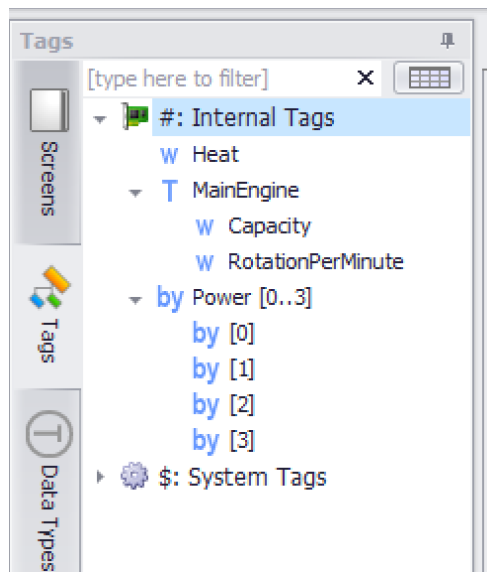


Figure 47: Tag Manager Example

12.3.1 Simple Types

These types are the most basic building blocks. All the simple types are numeric.

Type	Description
Bit	Represents a bit. Can be 0 or 1.
Byte	Represents a byte can be signed or unsigned.
Word	Represents a 16-bit integer can be signed or unsigned.
DWord	Represents a 32-bit integer can be signed or unsigned.
Float	Represents a signed single-precision floating-point.
Error	Represent an error state. Can be 0 or 1.

12.3.2 Strings

Strings in Galileo are fixed length character sequences. The Galileo Script language does not support string literals or character literals. Only predefined special functions can use strings.

12.3.3 Arrays

Arrays in Galileo have always a fixed length and their base type is of a numeric type.

12.3.4 Complex Types

There are two types in Galileo that act as a container for other tags.

12.3.4.1 Struct

A struct is a singleton type composed of multiple named tags. It may contain combinations of types described in this document, except for structs.

12.3.4.2 User-Defined Type

User-defined types behave like structs in other languages but are not to be confused with structs in Galileo script. User-defined types have a specifically defined structure of tag names and types, of which many instances can be created. User-defined types can contain tags of other user-defined types, but not of itself. Cyclic user-defined types are not allowed.

12.3.5 Tag Hierarchy

Tags may belong to three distinct categories: Internal Tags, System Tags and Comm Tags. Internal Tags are tags that are defined by the user for a project. System Tags are predefined tags that given by the environment runtime. Comm Tags are predefined by the hardware or system that the project targets.