



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

---



# Objective TinyTimber : OTTO

Creating an object-oriented alternative to real-time C.

Bachelor's thesis in Computer Science and Engineering

LARS ANDERSSON

OSKAR LENSCHOW

---

Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY  
UNIVERSITY OF GOTHENBURG  
Gothenburg, Sweden 2019



# DEGREE PROJECT REPORT

## Objective TinyTimber : OTTO

Creating an object-oriented alternative to real-time C.

Lars Andersson

Oskar Lenschow



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY  
UNIVERSITY OF GOTHENBURG  
Gothenburg, Sweden 2019

## **Objective TinyTimber : OTTO**

Creating an object-oriented alternative to real-time C.

Lars Andersson, Oskar Lenschow

© Lars Andersson, Oskar Lenschow, 2019.

Supervisor: Jan Jonsson, Department of Computer Science and Engineering

Examiner: Peter Lundin, Department of Computer Science and Engineering

Department of Computer Science and Engineering

Chalmers University of Technology

University of Gothenburg

SE-412 96 Gothenburg

Telephone +46 31 772 1000

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Cover: Image of the programming language OTTO.

Department of Computer Science and Engineering

Gothenburg, Sweden 2019

## Sammanfattning

När vi arbetat som assistenter i kurser inom realtidssystem på Chalmers, märktes det att en signifikant mängd studenter har svårt för maskinorienterad programmering. På grund av detta, så har huvudsakliga målet med detta projekt varit att skapa ett alternativ för studenterna, så att de enklare kan förstå processen bakom realtidssystem. Detta gjordes då genom att skapa ett nytt, objektorienterat språk. Språket är strukturerat med klasser och objekt, som exempelvis Java och C++. Det går att se det som en förenklad och objektorienterad version av programspråket C. Under arbetets gång så följdes kompilatorkedjan, med verktyg som Flex och Bison. Det resulterande programmet, en så kallad transpiler, översätter från det nya språket till språket C. Transpilern integrerades i CodeLite, den integrerade utvecklingsmiljön som används i labbarna för kurserna. Detta gjordes för att göra det så enkelt som möjligt för studenterna att använda sig av den. Den slutgiltiga produkten är förmånlig för de studenter som är mer vana vid objektorientering, då de inte behöver tänka på de maskinorienterade aspekterna, utan kan fokusera på realtidsaspekterna istället.

Keywords: C, Java, Objektorientering, Programmering, Realtidssystem, Transpiler, Flex, Bison.



## Abstract

When working as teaching assistants in the Real-time Systems courses on Chalmers, it was noticed that a significant amount of students find machine-oriented programming difficult to grasp. Therefore, the main purpose of this project has been to create an alternative way for students to understand the process of Real-time systems, by developing a new object-oriented language. This language is structured with classes and objects, much like Java and C++. It can be seen as a simplified, object-oriented version of the programming language C. In the development of the product, the language processing chain was followed, by utilizing tools like Flex and Bison. The resulting program, a transpiler, translates from the new language into the language C. This transpiler was integrated into CodeLite, which is the Integrated Development Environment (IDE) used in the labs. This integration was made to ensure easy usage. The finished product is beneficial for students who are more familiar with object-oriented programming, as they would not have to consider the machine-oriented aspects of the labs. Hence, this would increase the number of students understanding real-time systems, since they would face less obstacles during the course.

Keywords: C, Java, Object-Orientation, Programming, Real-time Systems, Transpiler, Flex, Bison.





## Acknowledgements

Thank you, Jan Jonsson, for letting us work as teaching assistants in your courses throughout these years, and for being our mentor and supervisor in this project. It has been a pleasure working with you. We would also like to thank Linus Aronsson for his support and ideas during the birth of this project. Thank you, Hanna Bergh, for all your help regarding English grammar correctness and proofreading.

Lars Andersson and Oskar Lenschow, Gothenburg, June 2019



# Glossary

- Bison** GNU Bison, free and open source tool used to generate a parser. 1
- C** An imperative, general-purpose programming language. 1
- Cross-Compiler** A compiler whose purpose is to compile an executable for another platform or system. 1
- Flex** Fast Lexical Analyser Generator, free and open source tool used to generate a scanner. 1
- GCC** GNU Compiler Collection, tool used to compile from source to target code. 1
- IDE** Integrated Development Environment, software with extensive support for programming. 1
- Java** An object-oriented programming language. 1
- LLDB** Low Level Debugger, tool used for debugging. 1
- Machine-oriented Programming** Programming close to hardware, commonly accessing raw memory addresses. 1
- MinGW** A cross compiler, enabling compilation for a platform while using a different one. 1
- Parser** A tool used to group tokens into a syntax tree. 1
- Pointer** A programming object that stores an address to another value inside of a computer's memory. 1
- Scanner** A tool used to tokenize a stream of characters into tokens. 1
- TinyTimber** A real-time kernel for C, used during real-time labs on Chalmers University of Technology. 1
- Transpiler** A Source-to-source compiler, translates from one programming language directly to another. 1
- Yacc** Yet Another Compiler-Compiler, tool used to generate a parser. 1



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background and Motivation . . . . .	1
1.2	Aim . . . . .	3
1.3	Objectives . . . . .	4
1.4	Report Structure . . . . .	4
<b>2</b>	<b>Theory</b>	<b>5</b>
2.1	Object-orientation . . . . .	5
2.1.1	Encapsulation . . . . .	5
2.1.2	Abstraction . . . . .	6
2.1.3	Inheritance . . . . .	6
2.1.4	Polymorphism . . . . .	6
2.2	Preprocessing . . . . .	7
2.3	Language Processing Chain . . . . .	8
2.3.1	Lexical Analysis . . . . .	8
2.3.2	Syntax Analysis . . . . .	9
2.3.3	Semantic Analysis and Code Generation . . . . .	10
<b>3</b>	<b>Method</b>	<b>11</b>
3.1	Software and Procedure . . . . .	11
3.1.1	Flex . . . . .	11
3.1.2	Bison . . . . .	12
3.1.3	C11-Standard . . . . .	13
3.1.4	GCC, LLDB and MinGW . . . . .	13
3.2	Inspiration . . . . .	13
3.3	Testing . . . . .	14
<b>4</b>	<b>Results</b>	<b>15</b>
4.1	Software . . . . .	15
4.1.1	Lab Environment . . . . .	15
4.2	The Language . . . . .	16
4.2.1	Syntax . . . . .	16
4.3	Validation . . . . .	18
4.4	Remaining Issues . . . . .	19
4.5	The Manual . . . . .	20

<b>5</b>	<b>Discussion</b>	<b>21</b>
5.1	Alternative Solutions . . . . .	21
5.1.1	TinyTimber for C++ . . . . .	21
5.1.2	Java . . . . .	21
5.1.3	Timber . . . . .	22
5.2	Software . . . . .	22
5.3	Object-oriented Concepts . . . . .	22
5.4	Conclusion . . . . .	23
5.5	Further Development . . . . .	23
	<b>Bibliography</b>	<b>25</b>
<b>A</b>	<b>Appendix</b>	<b>I</b>

# 1

## Introduction

### 1.1 Background and Motivation

Object-orientation solves a great number of issues with regular programming languages. It ensures that the code is where it is supposed to be, and that it is only accessible in the ways that are intended [1]. This, together with the fact that we worked as teaching assistants in a course that does not use object-orientation, is the primary reason this project came to be. When working as teaching assistants, you get valuable insight into how the courses are run behind the scenes, but you also get to experience first-hand, the issues that students face.

The primary course we worked in is called real-time programming. In that course, you are to create a melody player that can play the tune "Brother Jacob" together with other groups, on multiple micro-controllers, specifically one called MD407. It is written in the programming language C, using the real-time kernel TinyTimber, that handles concepts like deadlines for tasks. The students part of the masters course were sometimes having problems understanding the way the programming language C worked, especially when coded together with hardware. Although this way of thinking does work for many students, some come from different coding backgrounds, and offering them an alternative could benefit both them and the course.

We started discussing ways of developing an alternative to the system currently in place, and found it relevant to incorporate object-oriented programming, as we felt that this way of working is more open to people not used to programming as much, because of how well structured this type of coding is [1]. Not only did we want to change regular C, but it had to include some simplifications to the ways the TinyTimber system calls were written as well. The syntax for this today is not necessarily difficult, but if things were to be made more object-oriented and trivial, those should follow suite. After consulting the professor in the course, whom is also our supervisor in this project, it was decided that this was beneficial for the future of the course.

Below, there are three examples showcasing areas of difficulty in the current system, written with the programming language C together with TinyTimber, highlighting what could be improved with an object-oriented language:

## 1. Introduction

---

```
1 typedef struct {
2     Object super;
3     int count;
4 } Example;
5
6 Example obj = { initObject(), 0 };
7
8 void incCounter(Example *self, int unused) {
9     self->count++;
10    AFTER( SEC(1), self, incCounter, 0 );
11 }
```

**Figure 1.1:** Example of code written in C, together with TinyTimber

The pointers are the first thing to note (Figure 1.1, code in red). These need to be kept in check, because mistakes can easily be made with them. For example, students often put the sign `&` before the pointer, which indicates that you are to use the variable's address. This confuses some students, since they have difficulties differing pointers from addresses. This issue should be eliminated completely.

```
1 typedef struct {
2     Object super;
3     int number;
4 } Right;
5
6 typedef struct {
7     Object super;
8     char notNumber;
9     int number;
10 } Wrong;
11
12 Right obj1 = { initObject(), 0 };
13 Wrong obj2 = { initObject(), 'x', 0 };
14
15 void incNumber(Right *self, int unused) {
16     self->number++;
17     AFTER( SEC(1), &obj2, incNumber, 0 );
18 }
```

**Figure 1.2:** Example showing method called with object of wrong type

Another issue with the current system is that you are able to invoke a method with a different object type than intended. This can cause major errors when said method tries to use that object. This can easily happen because the compiler does not throw any errors when a regular pointer is sent to it. As can be seen in the example above (Figure 1.2), the method `incNumber` is expecting the object type "Right", and increases that objects variable "number" by one. But inside of the method, the same method is called again, but with the object type "Wrong" (obj2). Both objects still have the variable "number" though, which might confuse the programmer, leading



him or her to believe that the correct variable is increased. However, because the method expects the object type "Right", it will still read the object's pointer as that type. This means that when the code "self->number++" executes, it will do so on the variable "notNumber" in the "Wrong" struct, as it shares the same address offset inside the struct with the variable "number", as in the "Right" structure.

All of this might be a little difficult to follow, and is a perfect example of why an object-oriented alternative would help, because this error does not happen in object-oriented languages. They keep track of what methods belong to what objects, so a method can not be invoked from the wrong object by mistake.

Lastly, there are some issues with how the TinyTimber system calls are structured overall. For one, there is a problem with invoking methods with multiple arguments, so the example below (Figure 1.3) does not actually work. Methods invoked by TinyTimber can only accept a single, int-sized argument. This can be solved by creating a pointer to a list of arguments instead, and passing the pointer as an argument. Although the students are taught this solution, it is almost never utilized.

```
1 void multipleArguments(Example *self, int arg1, int arg2) {  
2     /* ... */  
3 }  
4  
5 ASYNC(&obj, multipleArguments, 5, 7);
```

**Figure 1.3:** Example showing a method incorrectly called with multiple arguments

## 1.2 Aim

The scope is limited to only creating a "source-to-source compiler", also known as a "transpiler" [2], to translate the code from the new language directly to the C language used today, and not create a complete new language from the ground up. This is because creating a language that way is too large of a project, and would have to include changes in how the system communicates and works with the hardware. Hence, it would not be compatible with the lab system in place.

The focus is also mainly put on eliminating the mistakes that the student make when working with the labs, and not necessarily to add everything object-orientation has to offer. All important concepts have still been considered and worked on, but only a handful of them were completed. This is discussed further in the following chapters.

### 1.3 Objectives

The first objective is to create an object-oriented styled alternative to the current programming language used, while keeping every aspect of the real-time system mindset intact. The second objective is to create documentation and a manual for the newly developed language. This is because the students should be able to easily learn how to use it, and look for help if issues arise.

The main objectives that need to be completed are the following:

- A syntax that makes sure the language looks properly object-oriented.
- Encapsulation and abstraction present in the language.
- Removal of the need for pointers or addresses.
- A working preprocessor, so that the transpiler can be used in the labs.
- A manual/tutorial describing how the language is used.

Side objectives/wishlist:

- Inheritance and polymorphism so that the language is more object-oriented.
- Error handling and error codes.
- Proper type-checking, with multiple passes.
- Native support of invoking methods with multiple arguments.

### 1.4 Report Structure

This project is divided into three main parts: research about the topics that are necessary to understand, find the tools needed to develop, and lastly, actually develop the product. Therefore the structure of the report is the same.

First, the theory is discussed, where the concepts of object-orientation and the language processing chain is covered. Afterwards, the methods used to develop the product are presented, together with some information regarding the inspiration for the language, and how it was tested. This is followed by the result, which showcases the finished product, what it has solved, and what problems remain. Lastly, the project as a whole, with information about alternative solutions and further development, is discussed.

Throughout the chapters, examples are given, that are mostly based on the examples given in chapter one. They are there to help better understand the improvements needed, and how those improvements look.

# 2

## Theory

### 2.1 Object-orientation

Object-orientation is a way of coding, that enables the program to be divided into parts, called objects. These objects have their own code, and can be used and owned by other objects. It is a way of keeping the code readable and organized, and is mainly built up by the four concepts Encapsulation, Abstraction, Inheritance and Polymorphism [1].

In the examples given in section 1.1, the most common mistakes that were made by students were highlighted. To solve these issues, encapsulation and abstraction are the most important concepts to understand. Inheritance and polymorphism are just as important for a real object-oriented language [1], but do not necessarily solve those kinds of issues. If the language is to be truly object-oriented, these should still be added, and can still improve how students write code. Therefore, all of these concept are explained below.

#### 2.1.1 Encapsulation

Encapsulation is the practise of sorting and placing code together with the class it is used by [4]. In a regular language like C, it needs to be manually ensured that the right code is used by the right method [3]. In an object-oriented language, code is placed within classes, and is only part of that class [4]. So if that code is to be used, an object created from that class needs to be accessed. This gives the programmer a helping hand in keeping track of what belongs to what, and ensures that the code is more protected. In the example below (Figure 2.1), one can see how a variable and a method is part of, and inside, a class.

```
1 class c {  
2     var v;  
3     method m() {  
4         /* ... */  
5     }  
6 }
```

Figure 2.1: Simple encapsulation

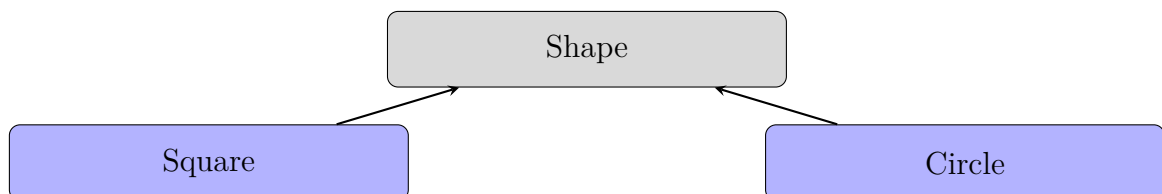
Encapsulation solves one major issue with regular C and pointers, as the code that is relevant to the object used is only accessible through that object. Therefore, a student would no longer be able to accidentally send in the wrong object type to a method, and the idea of pointers would not be seen by the user.

### 2.1.2 Abstraction

Abstraction is a natural consequence of having good encapsulation. It is all about making sure only the relevant code is accessible to other objects [5]. The mechanism that is used should be as high level as possible [10, p. 5], and should hide any code that is not necessary to understand or use, but is still utilized by the program.

### 2.1.3 Inheritance

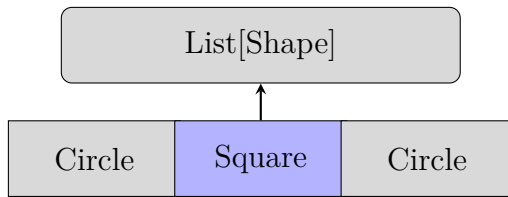
Inheritance is what the name implies. Classes can inherit from other classes. What this means in practice is that you can create an object that gets all the functionality of the parent class, but can keep building on top of that code [6]. This can help the programmer to a great extent, because you might want to create different classes that differ a little, but still have the same basic functionality. An example would be a super class for shapes in a program, although, that is not enough to explain the specifics of a circle. Instead, the circle is it's own class, which inherits from the shape class. This makes it easier to keep track of the program, and minimizes repeating code.



**Figure 2.2:** Inheritance

### 2.1.4 Polymorphism

Using the example of shapes above, a programmer might want to be able to put all created shapes into a list of some sort. This would be a problem, because a circle and a square are not the same type of object, and therefore can not share a list. Polymorphism solves this, by letting all child objects, in this case a circle and a square, be categorized as their parent object [7]. This means that the programmer can put both a circle and a square in a list made for shapes. The circle can have a method for calculating the surface which works in one way, while the square can have it's own. And then the objects inside of the shape-list can be iterated through, and the method can be called for all objects easily, with a small amount of code.



**Figure 2.3:** Polymorphism

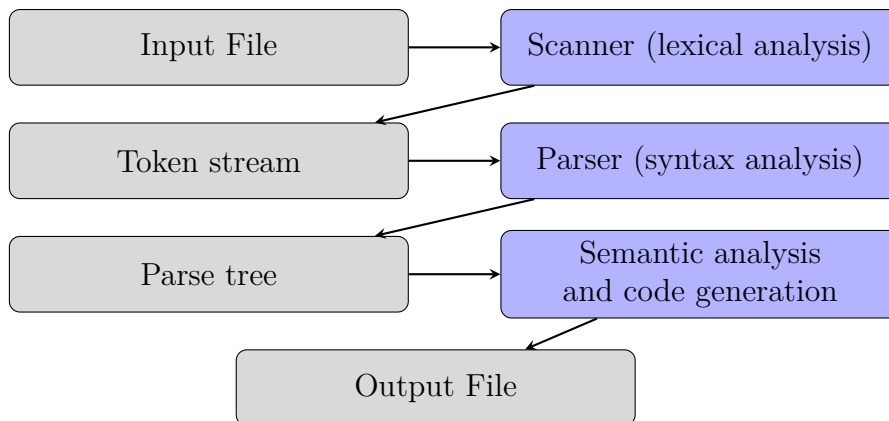
It is important to keep track of your child objects if their methods are named differently. For example, if the circle has a method `getRadius()`, which the square does not have, calling that method for all shapes in the list would result in errors every time the program tried calling `getRadius()` for a square.

## 2.2 Preprocessing

A preprocessor is something that takes input data, modifies it somehow, then generates output data, which is then later used as input data for something else [8]. An example of this could be a coffee grinder. Making coffee with whole beans might be difficult and unsuccessful, so first the beans need to be ground in the grinder, which is our preprocessor. Then that ground coffee can be used in your regular coffee-machine. Regular C has its own preprocessor, which looks for lines beginning with the symbol `#`, and directly replaces those lines with the code that it represents [9]. Because this project is supposed to have its own programming language, which is then translated in to the language C, a preprocessor is needed.

## 2.3 Language Processing Chain

The language processing chain, also known as the compilation phases, describes all the different steps used from converting an input file consisting of a character stream, in this case, a higher level programming language, into an output file [10, pp. 8-10]. The output file should consist of another character stream, usually raw machine code or some other lower level language [10, p. 5]. The output file, in this case, will consist of raw code of the language C.



**Figure 2.4:** The Language Processing Chain

It is important to mention that the the language processing chain described above is greatly simplified, and modern compilers often contain many more phases. A few examples of additional phases are Normalization, Source Code Optimization and Target Code Optimization [10, pp. 12-13].

Normalization as in removal of simplified language constructs, such as  
`int a, b;` turning into `int a; int b;`

Source Code Optimization such simplifying `a = 10 * 3 + 5;` into `a = 35;`

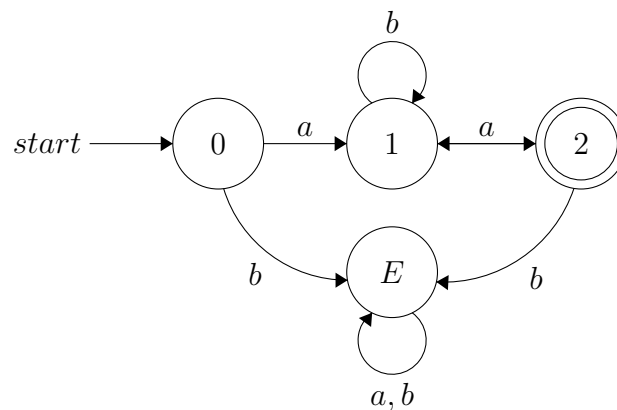
Target Code Optimization, as the name implies, means optimization similar to source code optimization, but is carried out as a final polish on the target code.

### 2.3.1 Lexical Analysis

The first step of the language processing chain is the lexical analysis of an incoming input stream of characters. The main purpose of this step is to analyze and split the character stream into a token stream [10, p. 8]. The way the stream is tokenized is based on syntax rules described as regular expressions of characters. If the incoming stream diverges from the syntax rules, a syntax error has occurred. This behaviour can be described as a finite state-machine. Traversing the states can only be done by following the rules of a regular expression [10, pp. 38-44].

The example below (Figure 2.5) shows the finite state-machine as a deterministic

automaton graph of the regular expression  $a(b+aa)^*a$ . In other words, this expression could be explained as a language with an alphabet only consisting of the letters 'a' and 'b'. In this language, words can only be produced if the word begins and ends with the letter 'a'. Anything between those 'a's can be any sequence of 'aa's and/or 'b's. For example, the word *abaaba* is a valid word within this language, which can be separated into  $\rightarrow a \rightarrow b \rightarrow a \rightarrow a \rightarrow b \rightarrow a$ . This would result in the states  $0 \rightarrow 1 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 1 \rightarrow 2$ , exiting on the state 2. However, the word *aaba* would not classify as a valid word as the states would result in  $0 \rightarrow 1 \rightarrow 2 \rightarrow E \rightarrow E$ , not exiting on the state 2.



**Figure 2.5:** Deterministic automaton of the regular expression  $a(b+aa)^*a$ . Starting at state 0 and exiting on state 2, while the state E indicates an error, and every step represents a letter.

### 2.3.2 Syntax Analysis

The second step of the language processing chain is the syntax analysis of the generated token stream, often called parsing. This analysis makes sure the stream of tokens follows the grammar of the language [10, pp. 8-9]. This parser traverses the given token stream, and generates a syntax tree containing the tokens following this grammar.

The two figures below (2.6 and 2.7) show a simple expression,  $3+8*6$ , parsed into an abstract syntax tree. The grammar applied to this example is expressed in Backus-Naur form [11], and every digit and operator is a token. In this example, the precedence levels of multiplication being carried out before addition is applied, just as the regular algebra operator precedence.

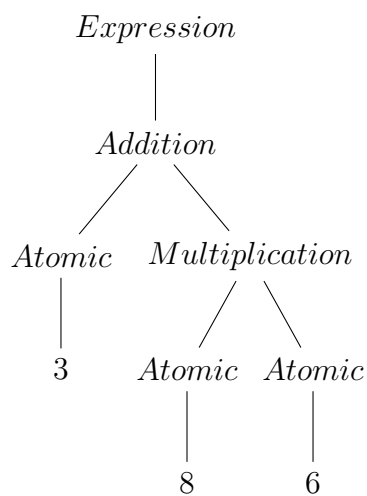
```
<Expression> ::= <Addition>

<Addition> ::= <Addition> + <Multiplication>
<Addition> ::= <Multiplication>

<Multiplication> ::= <Multiplication> * <Atomic>
<Multiplication> ::= <Atomic>

<Atomic> ::= number
<Atomic> ::= ( <Expression> )
```

**Figure 2.6:** Example of a syntax grammar, expressed in Backus–Naur form.



**Figure 2.7:** Abstract syntax tree of the expression 3+8\*6.

### 2.3.3 Semantic Analysis and Code Generation

When the parse tree is generated, the next step is to make sure this tree of tokens actually is semantically correct [10, p. 8-9]. Simply put, this is often carried out by traversing each branch of the tree, making sure the whole branch consists of the correct type, called type-checking. When the parse tree is proven to be semantically correct, the output code can be generated.



# 3

## Method

### 3.1 Software and Procedure

In this chapter, the tools used to create the transpiler are covered. For the language processing chain, Flex and Bison were used. GCC and MinGW were used for compiling the binary executables and LLDB was used for debugging purpose.

#### 3.1.1 Flex

The tool Flex (Fast lexical analyzer generator) [12] was used to generate the lexical analyzer part of our language processing chain. Flex was chosen due to it being free and open-source, in comparison to Lex [13], an other widely known lexical analyzer generator.

In order to generate a useful lexical scanner for the new language, syntax rules were created. Those syntax rules represented keywords in the language, marking places in the input file where to start and end the tokenization. These self-made rules were combined with the regular C-language ANSI-standard rules [16], as a complement for the scanner. The generated scanner was able to find points in the input file where to start tokenizing. After this, the scanner began tokenizing the character stream following the C-standard rules until finally reaching the end point keyword. When this keyword was found, the scanner halted the tokenizing until either the end of file or a new start keyword was reached.

The example below, figure 3.1, shows 5 lines of code tokenized by the syntax rules in figure 3.2, resulting in the token stream in figure 3.3. The ANSI standard C11 was utilized for the C-language syntax rules, which will be explained later.

```
1 char x = 5;  
2 START  
3     int x = 3;  
4 END  
5 x = 'X';
```

**Figure 3.1:** Input character stream (Example)

```

1 "START"      { startLexing(); } // Start point
2 "END"       { endLexing(); } // End point
3
4 "int"       { if (lexing) return TYPE_INT; }
5 [a-z]+     { if (lexing) return IDENTIFIER; }
6 "="        { if (lexing) return EQUALS; }
7 [0-9]+     { if (lexing) return NUMBER; }
8 ";"        { if (lexing) return SEMICOLON; }
9
10 .         { ; } // Skip anything else

```

Figure 3.2: Syntax rules (Example)

```

1 // int x = 3 ;
2 [ TYPE_INT, IDENTIFIER, EQUALS, NUMBER, SEMICOLON ]

```

Figure 3.3: Output token stream (Example)

```

54 "CLASS"      { LEX_CLASS++; return CLASS; }
55 "new"        { return NEW; }
56 "OBJECT"     { LEX_STM++; return OBJECT; }
57 "EXTENDS"    { return EXTENDS; }
58 "SUBR"       { LEX_STM++; return SUBR; }
59 "ASYNC"      { LEX_STM++; return ASYNC; }
60 "SYNC"       { LEX_STM++; return SYNC; }
61 "AFTER"      { LEX_STM++; return AFTER; }
62 "BEFORE"     { LEX_STM++; return BEFORE; }
63 "SEND"       { LEX_STM++; return SEND; }
64 "CALLBACK"   { LEX_STM++; return CALLBACK; }
65 "CONSTRUCTOR" { return CONSTRUCTOR; }
66 "INSTALL"    { LEX_STM++; return INSTALL; }
67 "TINYTIMBER" { LEX_STM++; return TINYTIMBER; }
68

```

Figure 3.4: Some of the syntax rules constructed for the language

### 3.1.2 Bison

As Flex being the free alternative to Lex, Bison [15] was used as a free and open-source alternative to Yacc [14] for generating the parser part of the language processing chain.

A grammar was developed throughout the course of the project, which the parser used to generate the parse tree. This grammar was a combination of the C-language ANSI-standard grammar together with grammar rules defined by ourselves. This way of method allowed for regular C syntax to be used inside the scope of our own language, liberating us from defining a whole grammar ourselves, which is re-

quired for a functioning programming language. As in the lexical analysis utilized ANSI standard C11 for the syntax rules, the parser utilized C11 as the language's grammar.

```

734
735 ▼ tinytimber_systemcall
736   : async_call ':' IDENTIFIER '.' IDENTIFIER argument_expression_list_brackets {$$$ = convertAsync($3,$5,$6);}
737   | async_call ':' IDENTIFIER argument_expression_list_brackets {$$$ = convertAsync(NULL,$3,$4);}
738   | subr_call ':' IDENTIFIER '.' IDENTIFIER argument_expression_list_brackets {$$$ = convertSubroutine($3,$5,$6);}
739   | subr_call ':' IDENTIFIER argument_expression_list_brackets {$$$ = convertSubroutine(NULL,$3,$4);}
740   | sync_call ':' IDENTIFIER '.' IDENTIFIER argument_expression_list_brackets {$$$ = convertSync($3,$5,$6);}
741   | sync_call ':' IDENTIFIER argument_expression_list_brackets {$$$ = convertSync(NULL,$3,$4);}
742   | after_call ':' IDENTIFIER '.' IDENTIFIER argument_expression_list_brackets {$$$ = convertAfter($1,$3,$5,$6);}
743   | after_call ':' IDENTIFIER argument_expression_list_brackets {$$$ = convertAfter($1,NULL,$3,$4);}
744   | before_call ':' IDENTIFIER '.' IDENTIFIER argument_expression_list_brackets {$$$ = convertBefore($1,$3,$5,$6);}
745   | before_call ':' IDENTIFIER argument_expression_list_brackets {$$$ = convertBefore($1,NULL,$3,$4);}
746   | send_call ':' IDENTIFIER '.' IDENTIFIER argument_expression_list_brackets {$$$ = convertSend($1,$3,$5,$6);}
747   | send_call ':' IDENTIFIER argument_expression_list_brackets {$$$ = convertSend($1,NULL,$3,$4);}
748   ;
749
750 ▼ argument_expression_list_brackets
751   : '(' argument_expression_list ')' {$$ = $2;}
752   | '(' ')' {$$ = NULL;}
753   ;
754
755 ▼ async_call
756   : ASYNC '(' ')' {}
757   | ASYNC {}
758   ;
759

```

**Figure 3.5:** A small segment of the grammar constructed for the language

### 3.1.3 C11-Standard

To ease the development of the language processing chain, a complete predefined set of lexical syntax rules [16] and grammar [17] was utilized. C11 [18] was the ANSI standard for the C language developed in 2011, replacing the older standard C99 [19] and has been replaced by the standard C18 [20]. Due to availability of the open-source C11 Lex and Yacc files, C11 was chosen.

### 3.1.4 GCC, LLDB and MinGW

The working environment for the project was on a UNIX system, which enabled the use of GCC [21] for compiling into a UNIX binary, and LLDB [22] for debugging. In order to compile for the target operating system Windows, which was mainly used within the real-time labs, MinGW was utilized to cross-compile from UNIX to Windows [23].

## 3.2 Inspiration

To build the syntax of the language, inspiration was taken directly from Java and the way that language handles objects. In the following example (Figure 3.6), one can see how Java declares a class with local states, a constructor and methods [24]. It is based on the previously given example written in language C (Figure 1.1), but without the TinyTimber system calls.

```
1 public class Example {
2
3     private int count;
4
5     public Example() {
6         count = 0;
7     }
8
9     public void incCounter() {
10        count++;
11    }
12 }
```

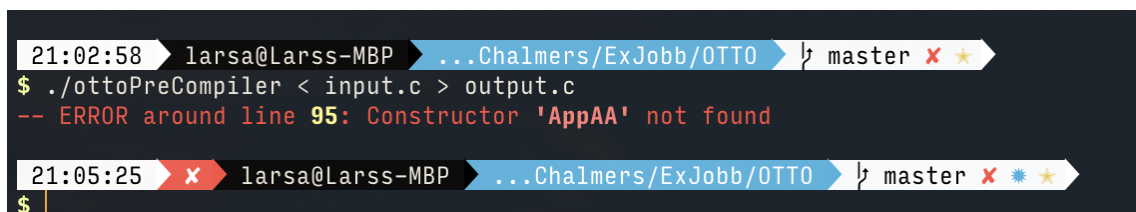
**Figure 3.6:** Class declaration in Java

The keywords `public` and `private` were omitted due to everything being public in the language being created.

## 3.3 Testing

As this project is made to support the Real-time Systems labs, the developed software was tested with those labs. When investigating whether the language could build proper C, the labs were written in the new language and compared to how they were written in C, and then uploaded to the hardware used and tested.

The main way of testing that the transpiler worked overall, and that it produced the output correlating with the equivalent input, was to write code that should not work, and see if it worked. Many times the code would pass through, and the output would be full of code that is not allowed and does not work. Coding with intentional errors proved to be a good way of securing the language structure. Error codes were also added (see figure 3.7), which was on the wishlist for this project, and while developing, the importance of them were made clear. Discovering anomalies within the program was facilitated when regular errors were displayed in a clear way. This is also beneficial for students in the future, so that they can find out what type of error they have made.



```
21:02:58 larsa@Larss-MBP ...Chalmers/ExJobb/OTTO | master ✖ ★
$ ./ottoPreCompiler < input.c > output.c
-- ERROR around line 95: Constructor 'AppAA' not found

21:05:25 ✖ larsa@Larss-MBP ...Chalmers/ExJobb/OTTO | master ✖ ★
$
```

**Figure 3.7:** Example of an error thrown when invoking a misspelled constructor

# 4

## Results

### 4.1 Software

The main focus of this project has been the creation of the transpiler together with an object-oriented language syntax. It was also important to ensure that the finished product could be run in a way that fit the lab environment, and had to be integrated into the program that is used today.

The transpiler works as intended. It is able to read code with the new syntax and convert it to regular C. The transpiler can run both on Windows and on unix-based systems. It is executed through a command prompt, where a source file is sent in and an output file is given (See example below).

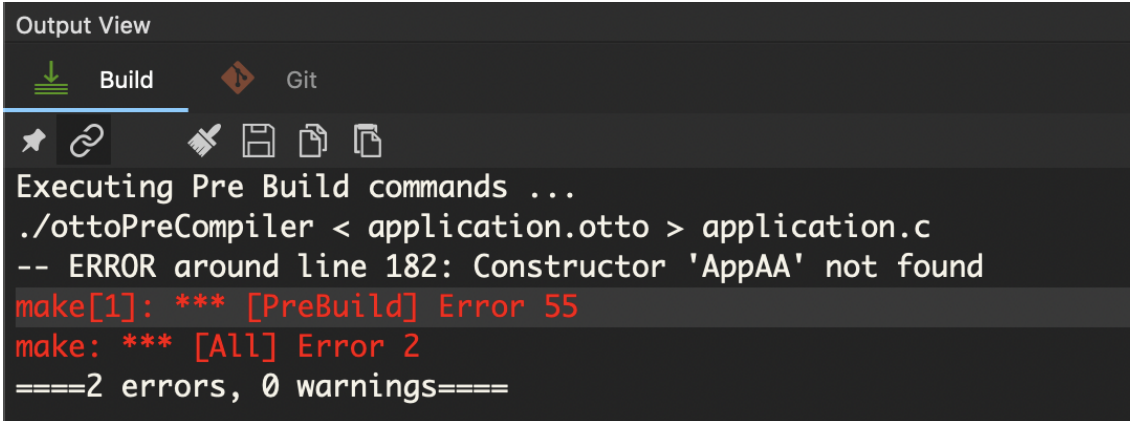
```
$ ./ottoPreCompiler < input.otto > output.c
```

#### 4.1.1 Lab Environment

Because this product is to be used within the Real-Time Systems labs, it should also be fully integrated into the IDE which the students are utilizing. Currently, the students use the open-source freeware CodeLite [25]. The goal was to integrate the transpiler and somehow add it to CodeLite's compiler pipeline, so that it was always available as an option on the workstations. After a bit of searching and reading, no clear way of doing this was found, so another solution had to be used. It turned out that there was a fairly easy way of adding it to a project folder. The transpiler would not be usable in all projects, but it would be in all project folders it was manually added to.

This was not the only problem though, as some issues were encountered because the program runs on windows machines, and the transpiler was compiled to unix systems. After figuring this out, the transpiler was cross-compiled, which is when a program is compiled using one system to work on another system [26]. This proved to be successful. After correcting a few more minor errors, the transpiler was added as a pre-build command, so that it would work as a preprocessor, and is automatically called when the project is built.

Below, there is a figure (4.1) showing the same type of error message as shown in figure 3.7, but this time within CodeLite. Errors thrown during the pre-build stage unfortunately resulted in the loss of the color coding.



```
Output View
Build Git
Executing Pre Build commands ...
./ottoPreCompiler < application.otto > application.c
-- ERROR around line 182: Constructor 'AppAA' not found
make[1]: *** [PreBuild] Error 55
make: *** [All] Error 2
====2 errors, 0 warnings====
```

Figure 4.1: Example of an error thrown within Codelite

## 4.2 The Language

The language created is a very basic version of an object-oriented language. It has basic encapsulation, abstraction and inheritance, but no polymorphism. The reason behind this is that polymorphism was the most challenging to tackle, but also not worth the effort, as the scope of the labs does not require this. It has support for real-time programming, specifically that of TinyTimber, as it is the system used in the labs today. The name for the language is OTTO, which is a recursive acronym of "Objective TinyTimber: OTTO".

Below, a short description of the syntax can be seen, but a more thorough guide can be found in the appendix.

### 4.2.1 Syntax

Given that the language is based on object-orientation, the syntax is as well. As mentioned in chapter 3, Java was used for inspiration. This is why the language produced has the same structure. Classes have code wrapped inside of them, including a constructor that is called to instantiate an object of that class. Local states and methods are also encapsulated within the class. The TinyTimber system calls still needed to be there, so they were included as well, but reworked. Previously, the calls were only macros in C [27], and had this structure:

```
ASYNC(&object, method, variable)
```

This was reworked to look like this:

```
ASYNC: object.method(variable)
```

This structure was used since, in the former way of writing, it looked more like a function call to the function ASYNC, with the object, method and an argument being parameters. Here, the students also have to keep track of when to include the

& which refers to the address of the given object. The latter looks more like proper object orientation, where the method belongs to, and is accessed through, the object, with its parameter being only the argument (SOURCES). ASYNC in this case looks more like a modifier or attribute, which is what it is supposed to be. The & for the address is also ignored, and dealt with under the hood.

The example shown in the previous chapter, that highlighted the issues of the current system, can be seen again below (Figure 4.2), together with Figure 4.3 as a comparison, showing the improvements made with the new language. The methods are now encapsulated by the class, instead of just expecting the address of the right type of object to be sent in.

```

1 typedef struct {
2     Object super;
3     int count;
4 } Example;
5
6 Example obj = { initObject(), 0 };
7
8 void incCounter(Example *self, int unused) {
9     self->count++;
10    AFTER( SEC(1), self, incCounter, 0 );
11 }

```

**Figure 4.2:** Example of code written in C, together with TinyTimber

```

1 CLASS Example {
2
3     int count;
4
5     Example() { }
6
7     void incCounter() {
8         this.count++;
9         AFTER(SEC(1)): this.incCounter();
10    }
11 }
12 }

```

**Figure 4.3:** The same example written in OTT:O

### 4.3 Validation

If the new language is to be used in a course on Chalmers, it needs to work at least as well, and preferably better, than the current system in place. To make sure that the product meets the standards intended, or even works at all, multiple tests were established to make sure the language could do everything it should be able to do.

To test the custom feature of sending multiple arguments, which was not possible with the old system, without adding extra code, a program was written with a method that expected multiple arguments. These were used inside of the method to modify the local states of the object that owned the method. Afterwards, the local states were accessed and printed. This method was then invoked with all Tiny-Timber modifiers, so that these could be tested to work simultaneously. Below, an example is given (Figure 4.4) with the modifier `ASYNC`.

```
1 CLASS App {
2     char a,b,c;
3
4     App() {}
5
6     void method(){
7         ASYNC: this.multiArg('a', 'b', 'c');
8         SYNC:   s.sci_writechar(this.a);
9         SYNC:   s.sci_writechar(this.b);
10        SYNC:   s.sci_writechar(this.c);
11    }
12
13    void multiArg(char a, char b, char c) {
14        this.a = a;
15        this.b = b;
16        this.c = c;
17    }
18 }
```

**Figure 4.4:** A program to test the feature of multiple arguments

To ensure that all of the real-time aspects worked as intended, all of the labs written in the current language were written in our language and tested on the hardware. This was also done continuously throughout the project, but is of course critical to do as a last test. The testing was successful, and the code worked very well. Because the precompiler generates a file with C-code, this file can be compared to how the labs would be written originally. This proved to be a useful way to see if the transpiler worked, because comparing them demonstrated how well it was able to translate correctly.

To test inheritance, programs were written with two or more classes. Below (Figure 4.5), an example of inheritance is shown. The child class inherits the local state `i` and the method `incValue` from the parent class. When the method `Print` is invoked



the local states `j` and `i` are added together and printed out in the terminal. When the method `incValue` is invoked on the child, the local state `i` is incremented one integer.

```

1 CLASS Parent {
2
3     int i = 5;
4
5     void incValue() {
6         this.i++;
7     }
8 }
9
10 CLASS Child EXTENDS Parent {
11
12     int j;
13
14     Child(int arg){
15         this.j = arg;
16     }
17
18     void Print() {
19         int result = this.i + this.j;
20         char buff[20];
21         snprintf(buff, 20, "Value: %d\n", result);
22         SYNC:    s.sci_write(buff);
23     }
24 }

```

**Figure 4.5:** A program to test inheritance

All of these tests proved that most objectives, set up before the project started, are finished. The syntax is object-oriented. Pointers and addresses are not allowed, which makes them impossible to use by mistake. The preprocessor works and translates the new language into C, and the output file is then compiled automatically in Codelite. The side objectives error handling, inheritance and invoking methods with multiple arguments were also added and are working properly.

## 4.4 Remaining Issues

One major objective that is not fully completed is true encapsulation and abstraction. The code is partly encapsulated. Methods and variables need to belong to classes, but only variables are truly protected in the end. This is because when the code is translated into C, the variables are part of a specific struct, but the methods are only expecting to receive that type of struct, not parts of it.

In practice, this means that you can still invoke the wrong method from the wrong object. But one notable factor to take into consideration is the high degree of which the structure of the new syntax should help with keeping track of what is correct.

In C, methods are just written in a file, but are not part of any object. In the new language, methods need to be wrapped by the class they belong to. This already creates a clear indication of what objects can invoke that method.

The syntax for invoking said method is also greatly simplified, as seen in the examples above. The removal of addresses, together with the fact that objects and methods are not arguments of another method anymore, should remove the majority of confusion about how methods are called.

All of this being said, ensuring that methods can only be called using the correct object is important, and should definitely be considered if the language is to be developed further, which is discussed more in chapter 5.

### 4.5 The Manual

A manual for how the new language is used was absolutely necessary in order for the students to be able to code with it. It consists of short examples showcasing how the different parts of the language are constructed, and what they can be used for. It does not contain thorough information on how to program with the TinyTimber kernel, as a guide for that already exists [27]. The only remaining issue regarding this is that the guide currently only supports the language C, which would have to be modified in the future. The manual as a whole can be found in the Appendix.

# 5

## Discussion

### 5.1 Alternative Solutions

At the start of this project, we had four different ideas of ways we wanted to do it. One of the ideas was used and is what this report is about. The other three were quite interesting as well, but would have been too difficult to conduct, especially within the time-frame we had. We did however start researching these alternatives and spent some time on them, so they are worth discussing.

#### 5.1.1 TinyTimber for C++

Given that the labs are written in C, with the real-time kernel TinyTimber, an interesting idea would have been to rewrite TinyTimber to work with C++. This would have enabled the labs to be written in an object-oriented language, that had everything it needed from the start [3]. The reason to why this idea was scrapped was because it felt too complex. There is not that much documentation on how to actually structure the code for TinyTimber, and it is tailored to specifically C. We also discussed this option with our supervisor and he agreed that it was interesting, but not the best alternative.

#### 5.1.2 Java

This whole project came to be partly because we wanted to be able to write the labs in Java. The problem with this is that Java does not work like most languages. It runs on something called a Virtual Machine, which is a program that runs on the operating system used, and that program in turn runs the Java program [28]. Because the MD407 micro-controller does not really have a proper operating system, the regular Java would not work.

We looked into other options. There is embedded Java, that is used more like its own operating system that can run directly on hardware [29]. The problem there was that there are no embedded Java alternatives for the processor used. Another issue is that Java is not made for real-time handling, and has no proper way of handling concepts like deadlines. There is real-time Java [30], but then that would have to be combined with some type of home-made embedded Java. This idea was obviously way too grand for us to complete, but it was still very interesting to read up on and would have been a fitting project.

### 5.1.3 Timber

Before TinyTimber was used as a runtime kernel for the labs, it was a part of the programming environment of a programming language called Timber. Timber is a programming language structured mostly like a functional programming language, but also includes concepts from object-orientation [31]. The TinyTimber part of it was later separated from the language, and instead used with C.

One idea was to revitalize Timber into something that would work on the current system. The main reason for this was many of the students in the master course having a background in functional programming. We started reading up on the language to see what could be done, but the idea was sadly killed quite quickly. This because our supervisor contacted the creator of Timber, found out what needed to be done, and explained that the project would be quite massive and difficult. This idea could definitely be expanded on in the future, and would certainly be useful.

## 5.2 Software

In the beginning of the project, a number of issues arose. We had just finished a course on programming languages and how they are structured, so we presumed that the libraries and software from that course would be beneficial to use. This turned out to be partly false, because of the restrictions of the tools used. In that course, a "tool" called BNFC was used [32], which combined the lexical analysis and syntax analysis in an easy way. We started building our transpiler using this, but got stuck after a while because it was too restricting. It was not ideal for being able to allow regular C code outside of the language being built, and was also not very popular or well known, so finding information online was harder. This set us back a couple of weeks, and we had to start learning Flex and Bison instead. But the general knowledge of how compilers are built and how the language processing chain works still helped along the way, as much of it was applied in the project in the end.

## 5.3 Object-oriented Concepts

The goal was always to implement all major concepts from object-oriented programming, because having a language restricted with half of them felt incomplete. For the real-time systems labs, it would have been enough with only encapsulation and abstraction, since the main goal of the project was to minimize the errors made by the students today, caused by C code being a bit too open and error friendly. However, as we wanted more object-orientation, we decided to add inheritance as well. Still, the inheritance in our language is not necessarily true to how it works in regular languages, given that the only thing happening behind the scenes is that the local states of the parents class are copied into the child. Polymorphism is something that we wished we could have added, but was proved to be too difficult. This would also have brought the least improvements for the students.

## 5.4 Conclusion

One major purpose of this project was, as mentioned earlier, to ensure that less issues for the students would arise. The new language can be seen as an ergonomic improvement for the courses, as it would allow students to focus on learning the important real-time concepts covered, instead of them getting frustrated over losing precious lab time, in turn increasing their risk of failing. Many of the master students come with little to no background in C programming, and they should not have to learn a whole new language in order to understand real-time systems, which this new language aims to solve this by offering an alternative. Other environmental and ethical aspects were considered and discussed, but no clear connection to this project could be made.

## 5.5 Further Development

Since this language is not completely finished, but still works for the labs, it is open to a lot of further work. As of now, the whole transpiler only does one pass through the input code, and then produces the output. In the future, more passes could be made, to make it possible to do a lot more typechecking and error handling. The encapsulation and abstraction could be made significantly stronger, with different access modifiers (i.e making local states and methods private, protected or the like, to control how they can be accessed). Inheritance could be improved to support polymorphism, and methods could be allowed to have the same names in different classes, which is currently prohibited because of the restrictions of C, unless complex code is added [33].



# Bibliography

- [1] Stackify (2017) 'Advantages of OOP'. Available: <https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/> [Online; accessed 11-March-2019].
- [2] Gurumoorthy, P. and Padmanabhan, A. (2019) 'Transpiler'. Available: <https://devopedia.org/transpiler> [Online; accessed 13-March-2019]
- [3] Safyan, M. (2016) 'Object-Oriented Programming (OOP) in C'. Available: <https://www.codementor.io/michaelsafyan/object-oriented-programming-in-c-du1081gw2> [Online; accessed 29-March-2019]
- [4] Janssen, T. (2017) 'OOP Concept for Beginners: What is Encapsulation'. Available: <https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/> [Online; accessed 29-March-2019]
- [5] Janssen, T. (2017) 'OOP Concept for Beginners: What is Abstraction?'. Available: <https://stackify.com/oop-concept-abstraction/> [Online; accessed 29-March-2019]
- [6] Janssen, T. (2017) 'OOP Concept for Beginners: What is Inheritance?'. Available: <https://stackify.com/oop-concept-inheritance/> [Online; accessed 29-March-2019]
- [7] Janssen, T. (2017) 'OOP Concepts for Beginners: What is Polymorphism?'. Available: <https://stackify.com/oop-concept-polymorphism/> [Online; accessed 29-March-2019]
- [8] Tutorials Point (Unknown) 'Compiler Design - Overview'. Available: [https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_overview.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_overview.htm) [Online; accessed 07-April-2019]
- [9] cppreference.com (2018) 'C - Preprocessor'. Available: <https://en.cppreference.com/w/c/preprocessor> [Online; accessed 07-April-2019]
- [10] Ranta, A. and Forsberg, M. 'Implementing Programming Languages - An Introduction to Compilers and Interpreters' (Texts in Computing, volume 16). London: College Publications, 2012.
- [11] Estier, Th. (Unknown) 'What is BNF notation?'. Available: <http://cui.unige.ch/isi/bnf/AboutBNF.html> [Online; accessed 03-May-2019]
- [12] Estes, W. (2019) 'Flex GitHub Repository'. Available: <https://github.com/westes/flex> [Online; accessed 06-May-2019].
- [13] Lesk, M. E. and Schmidt, E. (unknown) 'Lex - A Lexical Analyzer Generator'. Available: <http://dinosaur.compilertools.net/lex/index.html> [Online; accessed 06-May-2019].

- [14] Computerworld. (2008) 'The A-Z of Programming Languages: YACC'. Available: [https://www.techworld.com.au/article/252319/a-z\\_programming\\_languages\\_yacc/](https://www.techworld.com.au/article/252319/a-z_programming_languages_yacc/) [Online; accessed 06-May-2019].
- [15] Free Software Foundation. (2014) 'GNU Bison'. Available: <https://www.gnu.org/software/bison/> [Online; accessed 06-May-2019].
- [16] Degener, J. (2012) 'ANSI C grammar, Lex specification'. Available: <http://www.quut.com/c/ANSI-C-grammar-l-2011.html> [Online; accessed 27-March-2019].
- [17] Degener, J. (2012) 'ANSI C Yacc grammar'. Available: <http://www.quut.com/c/ANSI-C-grammar-y-2011.html> [Online; accessed 27-March-2019].
- [18] International Organization for Standardization. (2011) 'ISO/IEC 9899:2011'. Available: <https://www.iso.org/standard/57853.html> [Online; accessed 06-May-2019].
- [19] International Organization for Standardization. (1999) 'ISO/IEC 9899:1999'. Available: <https://www.iso.org/standard/29237.html> [Online; accessed 06-May-2019].
- [20] International Organization for Standardization. (2018) 'ISO/IEC 9899:2018'. Available: <https://www.iso.org/standard/74528.html> [Online; accessed 06-May-2019].
- [21] The GCC Team. (2019) 'GCC, the GNU Compiler Collection'. Available: <https://gcc.gnu.org/> [Online; accessed 06-May-2019].
- [22] The LLDB Team. (2019) 'LLDB 8 Documentation'. Available: <https://lldb.lldb.org/> [Online; accessed 06-May-2019].
- [23] The MinGW Team. (2019) 'MinGW, Minimalist GNU for Windows'. Available: [http://www.mingw.org/Welcome\\_to\\_MinGW\\_org](http://www.mingw.org/Welcome_to_MinGW_org) [Online; accessed 06-May-2019].
- [24] Oracle. (2017) 'Java documentation Classes'. Available: <https://docs.oracle.com/javase/tutorial/java/javaOO/classes.html> [Online; accessed 06-May-2019].
- [25] Ifrah, E. (2019) 'CodeLite IDE'. Available: <https://codelite.org/> [Online; accessed 07-May-2019].
- [26] The GCC Team. (Unknown) 'Cross-Compilation'. Available: [https://www.gnu.org/software/automake/manual/html\\_node/Cross\\_002dCompilation.html](https://www.gnu.org/software/automake/manual/html_node/Cross_002dCompilation.html) [Online; accessed 07-May-2019].
- [27] Programming with the TinyTimber kernel, Johan Norlander, Sweden, 2012.
- [28] Tyson, M. (2018) 'What is the JVM? Introducing the Java Virtual Machine'. Available: <https://www.javaworld.com/article/3272244/what-is-the-jvm-introducing-the-java-virtual-machine.html> [Online; accessed 22-May-2019].
- [29] Oracle. (2014) 'Introducing Oracle Java SE Embedded'. Available: <https://docs.oracle.com/javase/8/embedded/develop-apps-platforms/overview.htm> [Online; accessed 22-May-2019].
- [30] Oracle. (2014) 'An Introduction to Real-Time Java Technology: Part 1, The Real-Time Specification for Java (JSR 1)'. Available: <https://www.oracle.com/technetwork/articles/javase/index-137216.html> [Online; accessed 22-May-2019].



- [31] timber-lang. (2008) 'Timber Home'. Available: <http://www.timber-lang.org> [Online; accessed 23-May-2019].
- [32] Forsberg, M. and Ranta, A. (2014) 'User Guide'. Available: [https://bnfc.readthedocs.io/en/latest/user\\_guide.html](https://bnfc.readthedocs.io/en/latest/user_guide.html) [Online; accessed 22-May-2019].
- [33] Lockless Inc. (Unknown) 'Overloading Functions in C'. Available: <https://locklessinc.com/articles/overloading/> [Online; accessed 23-May-2019].



# A

## Appendix

# Programming with OTTO

Lars Andersson  
Oskar Lenschow

June 2019

## 1 Classes

Objects are created using classes, which are blueprints for how objects are created and used. To instantiate a class, the class constructor needs to be invoked. The class can contain local states and methods relevant to it.

```
CLASS classname {  
    type localstate0;  
    type localstate1;  
    type localstate2;  
    ...  
    CONSTRUCTOR classname (arguments) {  
        ...  
    }  
    type method (arguments) {  
        ...  
    }  
    ...  
}
```

Tip: The prefix CONSTRUCTOR can be omitted.

### 1.1 Inheritance

A class can inherit local states and methods from another class, using the prefix EXTENDS. A class can only inherit from one single class.

```
CLASS name EXTENDS superclass {  
    ...  
}
```

### 1.2 Instantiating classes

Creating a new instance of an object is done by calling the class constructor and can **only** be carried out inside of the main function.

```
CLASS Example {  
    int value;  
    Example(int arg) { this.value = arg; }  
}  
  
int main() {  
    OBJECT example = new Example(13);  
    return 0;  
}
```

## 2 Invoking methods (TinyTimber attributes)

Synchronously invoke a method with a return value. Returns -1 if deadlock.

```
SYNC: object.method(arguments);
```

Asynchronously invoke a method.

```
ASYNC: object.method(arguments);
```

Asynchronously invoke a method with a given deadline.

```
BEFORE(deadline): object.method(arguments);
```

Asynchronously invoke a method after a given baseline offset.

```
AFTER(offset): object.method(arguments);
```

Asynchronously invoke a method after a given baseline offset, with a given deadline.

```
SEND(offset, deadline): object.method(arguments);
```

Example: Usage of the AFTER attribute.

```
CLASS Obj {  
    int value;  
    Obj() { }  
    setOne() {  
        this.value = 1;  
        AFTER(SEC(1)): this.setZero();  
    }  
    setZero() {  
        this.value = 0;  
        AFTER(SEC(1)): this.setOne();  
    }  
}
```

### 3 Builtin TinyTimber functions

#### Installing interrupt vectors

Interrupts can only be installed inside of the main function.

```
INSTALL(interruptVector): object.method();
```

#### Starting the TinyTimber kernel

TinyTimber can only be started inside of the main function.

```
TINYTIMBER: object.method();
```

#### Installing callbacks

Callbacks can only be created inside of the main function.

```
CALLBACK c = new Example(initMethod, port, object.method);
```

#### Timers

Timers are a custom data type created for TinyTimber and can be used in objects.

```
Timer t;
```

sampleTimer returns the difference between current baseline and timer.

resetTimer resets the timer to current baseline.

```
object.t.sampleTimer();  
object.t.resetTimer();
```

## 4 Examples of usage

```
#include "ottoStructs.h"
#include "ottoPrototypes.h"
#include "ottoObjects.h"

CLASS ClassOne {
    int value;

    ClassOne(int val) {
        this.value = val;
    }

    int getValue() {
        int returnValue = this.value;
        this.value++;
        return returnValue;
    }
}

CLASS ClassTwo {
    int value;

    ClassTwo() { }

    void setValue() {
        this.value = SYNC: objOne.getValue();
        AFTER(SEC(1)):    this.setValue();
    }
}

int main() {
    OBJECT objOne = new ClassOne( 0 );
    OBJECT objTwo = new ClassTwo();
    return TINYTIMBER: objTwo.setValue();
}
```

Simple example showing two objects interacting with each other. Each second, `objTwo` collects the value from `objOne`, which in turn increments the value.



```
#include "sciTinyTimber.h"
#include "ottoStructs.h"
#include "ottoPrototypes.h"
#include "ottoObjects.h"

CLASS App {
    App() { }

    void startApp() {
        // Setup interrupt channels
        SYNC: sci.sci_init();
        // Hello world!
        SYNC: sci.sci_write("OTTO says hello...\n");
    }

    void reader(char c) {
        // Print out what's received
        SYNC: sci.sci_write("\nReceived: \'");
        SYNC: sci.sci_writechar(c);
        SYNC: sci.sci_write("\'\\n");
    }
}

int main() {
    OBJECT app = new App();
    // Setup callbacks on interrupt-vectors
    CALLBACK sci = new Serial(initSerial, SCI_PORT0, app.reader);
    // Install interrupt-vectors
    INSTALL(SCI_IRQ0): sci.sci_interrupt;
    return TINYTIMBER: app.startApp();
}
```

This example showcases how callbacks can be used. In this particular case, each time an interrupt arrives on the `SCI_PORT0`, the `reader` method is invoked.

```
#include "ottoStructs.h"
#include "ottoPrototypes.h"
#include "ottoObjects.h"
#define GENERATOR_PORT = (*(char *) 0x1234)

CLASS Sonar {
    Timer timer;

    Sonar() { }

    int stop() {
        GENERATOR_PORT = SONAR_OFF;
    }

    int echo() {
        Time diff = this.timer.sampleTimer();
        if (diff < MSEC(LIMIT)) /* code */;
    }

    int tick() {
        GENERATOR_PORT = SONAR_ON;
        this.timer.resetTimer();
        AFTER(MSEC(10)): this.stop();
        AFTER(MSEC(500)): this.tick();
    }
}

int main() {
    OBJECT sonar = new Sonar();
    INSTALL(IRQ_ECHO_DETECT): sonar.echo;
    return TINYTIMBER: sonar.tick();
}
```

This example can be seen in 'Programming with the TinyTimber kernel', but in the language C instead.