Universiteit Leiden
The Netherlands

Opleiding Informatica

Metaprogramming

in Modern Programming Languages

Nouri Khalass

Supervisors:
Harry Wijshoff & Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

26/06/2017

Abstract

Metaprogramming is the practice of *analysing*, *manipulating* and *generating* other programs. It is used in many fields of computer science. One field that uses metaprogramming extensively is the field of compiler theory. Compilers use metaprogramming to analyse the source code that programmers write, possibly manipulate the code to do optimisations and generate an output program. However, metaprogramming is used for more than just that. It enables features that help the programmer to do some kind of code generation, code manipulation and code execution. Some of these features have been in use for a very long time while others are coming just around the corner.

These language features have never really been evaluated and compared with each other. This is needed to see which features still suffice and which need a revision. The goal of this thesis is to give an analysis of the features in programming languages that are enabled by metaprogramming. These features will be compared with each other too see where the flaws lie and what should be improved. By doing so, it is also possible to see what might cause problems in the future and what direction must be followed to avoid those problems.

Contents

Chapter 1

Introduction

1.1 What is metaprogramming

Metaprogramming is something that programmers often deal with unknowingly. Usually the process stays unnamed and concerns things like *analysing*, *manipulating* and *generating* certain things. This is of course very broad and makes giving a comprehensive definition rather difficult because it should encompass many topics. Additionally, because metaprogramming happens without the programmer really knowing it is not often discussed.

Still, in order to analyse metaprogramming with respect to modern programming languages it's important to have a complete definition of what metaprogramming entails. However, no universally agreed upon definition for metaprogramming exists. Most definitions only work from within the context that they are used in. Many of theses definitions thus lack detail and possibly leave out useful information. There are two main points of view from which metaprogramming is looked at. These two points lead to two different definitions which capture some part of metaprogramming.

The first definition is as follows:

> "**Metaprogramming** describes the ability from a program to *manipulate* other programs.
> The program that does these manipulations is called the **metaprogram**."

This definition is from a paper [18] about metaprogramming and the optimisation of programs using metaprogramming. More information about this paper will be given in Section 1.4. The definition contains some very important points with respect to metaprogramming. First, it says that metaprogramming is about the ability of a program to manipulate another program. This is a big part of metaprogramming. Metaprograms usually take an existing program and manipulate it to come up with a new program. However, the part of generating a (new) program is not mentioned in the definition.

Another point that the definition brings up is what the term metaprogram means. Obviously, this term should be closely related to metaprogramming. According to the definition, the program that does manipulations is the metaprogram. This is an acceptable claim which is easy and intuitive to understand. However, things become more difficult when taking into account that the metaprogram is also being controlled by another program or a programmer. The implications of this will discussed after the second definition is introduced.

The second definition gives a different take on what metaprogramming is. It goes as follows:

> "Programs that *generate* code (and thus generate other programs) are **metaprograms**. *Writing* metaprograms is called **metaprogramming**"

This definition is from an article [5] from IBM about what metaprograms are and how they can be written. Now, the second definition approaches metaprogramming from another angle. Instead of saying that metaprogramming is explicitly done by *other* programs, it says that metaprogramming is done by *the programmer* that *writes* the metaprogram. The question if metaprogramming is done by a program or by the programmer is especially important in the context of this thesis. As will be shown, some programming languages allow the programmer to (implicitly) generate code. This is a situation where both sides of the argument are valid. One the one hand, the programmer is writing a program which will generate code, thus the programmer is metaprogramming. But on the other hand, the programming language (or the compiler of the programming language) takes the code that the programmer has written and generates code according to that, meaning that the compiler is metaprogramming. All of this reasoning makes the discussion go in circles with no end in sight.

To clear up the ambiguity about the definition of metaprogramming, a definition will be given which will be used throughout this thesis when speaking about metaprogramming.

> "**Metaprogramming** describes the ability of a program to *analyse*, *manipulate* and *generate* other programs. A program that performs metaprogramming is called a **metaprogram**."

Now, this definition gives enough information so that it is usable within the context of this thesis. It first clearly defines what metaprogramming is, what it can do, and on what type of input. It says that metaprogramming is performed by a program to another program. The term program is defined somewhat loose here, being used to describe both a real program which can be executed and a program which is just lines of code. Typically the programmer writes this program code and the compiler program translates that into something that can be executed. Thus in this situation the compiler is the metaprogram and the source code written by the programmer is the input to the process of metaprogramming. This also means that a programmer does not really perform metaprogramming as it is in fact the compiler that does the metaprogramming. Programmers only make use of features that are possible *because* the compiler applies metaprogramming. What these features are and why they are interesting with respect to this thesis will be explained after a brief explanation about what metaprogramming is used for in practice.

1.2 Metaprogramming in practice

The final definition from Section 1.1 clearly states what metaprogramming does in an abstract form. It says that a program can *analyse*, *manipulate* and / or *generate* another program. Note that these three parts often happen in conjunction with each other. A path of execution might be that a program first analysis another program, then performs manipulations and finally generates a new program. These steps are of course similar to what a compiler does. It takes the programmer written source code and checks if it contains any errors (analysis), then it applies optimisations (manipulation) and finally translates it to an executable program (generation). This is of course an oversimplification but that does not weaken the claim. So knowing this, it is easy to see that a compiler uses all three parts of metaprogramming equally. However, there are other programs which also perform metaprogramming that are not compilers. Good examples are code generators like Lex and Yacc. Code generators take their input program, which is the case of Lex and Yacc is a set of language rules defined by the programmer, and transform it into boilerplate (C) code. This boilerplate code can in turn be used as the code parser in a compiler. Also note the fact that regular C code is emitted and not something like an executable. Code generators are often used to save the programmer from having to write boilerplate code.

To dive a little deeper into how a compiler uses metaprogramming, and also to show how this benefits the programmer, an example will be given in Listing 1.1.

Listing 1.1: Code using templates

```
1  template <typename T>
2  T pow(T x, int n) {
3    if (n == 0) return 1;
4    return x * pow(x, n - 1);
5  }
6
7  int main() {
8    int   x = pow(4, 2);
9    float y = pow(0.5, 3);
10   return 0;
11 }
```

$\Longrightarrow$

Listing 1.2: Implicit code after template deduction

```
1  int pow(int x, int n) {
2    if (n == 0) return 1;
3    return x * pow(x, n - 1);
4  }
5
6  float pow(float x, int n) {
7    if (n == 0) return 1;
8    return x * pow(x, n - 1);
9  }
10
11 int main() {
12   int   x = pow(4, 2);
13   float y = pow(0.5, 3);
14   return 0;
15 }
```

Listing 1.1 shows what generic programming looks like in C++. To support generic programming C++ introduced "templates", further described in 2.2.1. Generic programming is often used to allow the programmer to write a generic implementation for a solution where things like type information are not specified at the time of declaration. Using templates, the programmer does not have to maintain multiple versions of something with different types. This is a big win because it can save the programmer time, but also prevent nasty copy-paste errors.

Templates are also a very good example of how metaprogramming allows for functionality in a programming language. The compiler first has to perform an *analysis* on what types are being used. After that the types are deduced the compiler must create new function with the correct types. In a way, the compiler is *generating* a new programming. Now this new (sub)program has to be merged with the old program and be compiled further. The old program is *mutated* in such a way that the generic implementation is swapped with the new and correct implementations. Every step of the process used some aspect of metaprogramming to come to the end result!

1.3 Analysing metaprogramming

By now it is probably clear that metaprogramming is used to achieve many things, especially in programming languages. Giving an analysis about how metaprogramming is used in all (modern) programming languages would be a huge undertaking. Above all, many of the topics that would be discussed have a lot of overlap. That is why it is important to state what will and what will not be analysed with respect to modern programming languages and metaprogramming.

First, only modern programming languages will be analysed. Modern will be defined not necessarily by how old it is, but also by how relevant it is today (with respect to the discussed topic) and if it is simply a language that many people use. This reduces the scope of the programming languages somewhat, but not to an extend that would lead to leaving out relevant information. If a language is not used by many people but still has very interesting properties than it still will be discussed.

Second, not all aspects of metaprogramming with respect to programming languages will be discussed. Only things that saves the programmer time and doing work. This is of course still very broad but should be kept in mind. To be even more precise, only language properties that enable some kind of code generation, code manipulation and code execution will be looked at. Topics like **conditional compilation**, **macros**, **generic programming** and **compile-time programming** all fit within these categories. To briefly explain what all of these topics mean some small use cases will be given.

Conditional compilation is often used to support code that can compile to different platforms. As the name would imply, conditional compilation allows for the inclusion or exclusion of certain parts of the code based on certain conditions. One part of a code base might be written to work on a Linux platform while the other part is written for the Windows platform. The programmer is able to write multiple versions of for example a

function and have the compiler figure out at compile-time if that function should be further compiled and end up in the executable.

Macros are often used as a time saver for the programmer. Instead of having to write long repetitive strings of text a macro can be used as a shorthand for it. A macro can be used to represent a piece of text and when the macro gets resolved the compiler replaces all instances of the macro with its corresponding text. Macro resolving usually happens before everything else in the compiler pipeline. Often entities further up the compiler pipeline are not aware that macro substitution has taken place. This is because usually purely work with text and have no notion of the abstract syntax tree of the program.

Section 1.2 already explains what *generic programming* is. It is related to macros in that some form of substitution takes place. However, implementations of generic programming are often more ingrained in the compiler pipeline than macros. This is because in order to support generic programming it is important to have a deep understanding of the program.

Finally *compile-time programming* is the act of performing computation at compile-time. Usually programs are written with the idea that they will be executed somewhere in the future. The notion of performing computation at compile-time makes the programmer also think about what is possible at compile-time. There are major benefits to this. The programmer can for instance perform some expensive calculations at compile-time and use the result of that at run-time. This gives an obvious performance boost. Other reasons might be that the programmer wants to validate its program in a special way at compile-time. The compiler might allow for this functionality by exposing a hook that will be called at compile-time. In this way, the programmer can inspect its program at compile-time and have more control over the compilation process.

Now, why were these four topics mentioned specifically, besides the fact that they fall within the aspect of metaprogramming that will be analysed? The reason for that is that all those topics share a history with each other. Most solutions that were introduced to those problems did not just solve one single problem, but a whole group of seemingly similar problems. At the time this may have been a good thing but some of those choices have consequences that programmers have to deal with till this day. A good example of this is the how the `C` and `C++` programming languages enable conditional compilation and the use of macros. The choice was made to use a preprocessor which would include parts of the code depending on certain conditions and resolve macros. The preprocessor would be a separate entity and not be part of the compiler pipeline. Many problems with this solution have become apparent over time and those will be discussed further in Section 2.1.3.

Nevertheless, it is obvious to see that the use of a preprocessor tried to solve multiple problems. Common sense dictates that a solution that tries to solve multiple problems usually solves those problems poorly. Although, common sense also dictates that it is easy to see flaws in something when looking at it in hindsight. However, now that programmers have used those solutions for a very long time it is possible to summarise there shortcomings and come up with better solutions. Now more than ever it is possible to see what is wrong, reevaluate the solution and try to come up with a better one. And that is precisely what this thesis aims to do.

That isn't to say that the industry has not come up with solutions already. Quite the contrary, many new programming languages are being developed and they all introduce new solutions. Going back to the example of the preprocessor, most new programming languages explicitly choose not to use one. Languages like `D` and `Rust` have come up with their own solution. However, around the topic of generic programming history is repeating itself somewhat. As shown in Listing 1.1, templates can be used to write generic implementations. But that is not the only thing templates can be used for. Templates can also be used to perform computation at compile-time. Again a solution that tries to solve multiple problems.

But the problem of how to enable compile-time programming "correctly" is fairly new. Because of this, it is important to have a clear description of the problem that must be solved. An evaluation can be made about what might be possible problems and that information can be used to come up with a solution. That is also what this thesis wants to do.

1.4 Relevant work

Although there is much written about metaprogramming topics, especially in compiler theory, it usually is not mentioned directly. This is even more true for the topics that will be discussed in this thesis. No real comparison of programming languages and their features with respect to the four topics has been made. There exists the work of Tim Sheard [13] about what possible challenges are in the domain of metaprogramming. The paper gives insight into what metaprogramming means with respect to programming languages and their ability to analyse themselves. It brings up some interesting points about what theoretical problems can emerge with respect to metaprogramming.

On the topic of compile-time programming there exists a paper by Lucian Radu Teodorescu et al [18]. The paper proposes a programming language with heavy emphasis on compile-time programming. It uses its advanced analysis system to hyper optimise the resulting output code. The programming language proposed will be discussed in Section 2.5.4.

1.5 Thesis Overview

The form of this thesis is defined in the following way. Chapter 2 will provide an analysis of multiple modern programming languages. As stated before, the topics that will be looked are *conditional compilation*, *macros*, *generic programming* and *compile-time programming*. The analysis will give insight and enable a discussion about the way programming languages implement the before mentioned topics. Some languages have made improvements on certain topics and others just copied the same implementation. All examples and information will be given in such a way that it is not necessary to know how a certain programming language specifically works. Chapter 3 will give a summary of the analysis to highlight all the differences. Finally, Chapter 4 will give a conclusion on what the future might be for the discussed topics.

Chapter 2

Analysis

2.1 C

The programming language C was originally developed by Dennis Richie at Bell Labs in 1969 [9]. Its strives to be a low-level system language whereby most responsibilities lie with the programmer. The language has influenced countless other languages. While it is fairly dated it still has a very large and dedicated user base. However the use of C in the modern day is something that could be debated. When handled incorrectly, C can cause major safety issues for the programmer, so caution is advised.

2.1.1 Preprocessor

The first part of compiling a C program is taking the source code trough a **preprocessor**. The preprocessor is used to, among other things, resolve **macros** and perform **conditional compilation**. It can be seen as a separate entity in the compilation pipeline, which means that the actual compiler is not aware of the fact that a preprocessor was used. Interfacing with the preprocessor usually happens with *directives* which are denoted in source code with a `#`.

2.1.2 Conditional Compilation

Conditional compilation in C allows for the inclusion or exclusion of certain parts of source code. In order for the preprocessor to know what parts should be included and what parts should be ignored, the preprocessor looks for the `#ifdef` and `#endif` directives. The `#ifdef` directive checks if a value has been defined to the preprocessor. It can be used as follows:

```
#define <name>
```

$\Longleftrightarrow$

```
#ifdef <name>
.. CODE TO INCLUDE ..
#endif
```

The preprocessor will include the code between the `#ifdef` and `#endif` if `<name>` is defined. Additionally, it is possible to have an `#else` statement which can be used as one would expect, and an `#ifndef` if the code should be included if `<name>` is not defined. The code between the clauses is taken verbatim and thus with no regard if it is syntactically or semantically correct.

Listing 2.1 shows what this looks like and Listing 2.2 shows the result of doing conditional compilation.

Listing 2.1: Conditional compilation in C

```
1  #include <stdio.h>
2  #define DEBUG
3
4  int main() {
5    #ifdef DEBUG && __linux__
6      printf("Debug and Linux");
7    #else
8      printf("Other systems");
9    #endif
10   return 0;
11 }
```

Listing 2.2: Resulting code

```
1  #include <stdio.h>
2  #define DEBUG
3
4  int main() {
5    printf("Debug and Linux");
6    return 0;
7  }
```

Notice the `__linux__` on line 5 of Listing 2.1. Preprocessors implicitly expose some defined values depending on things like the compilation platform and architecture. These can be used by the program to control what parts of the source code are compiled depending on the compilation platform.

Something to note is the fact `DEBUG` does not have a value. It is possible to assign it a value, as will be explained in Section 2.1.3, and using that value to then perform conditional compilation. This is done with the `#if` directive which behaves similar to a regular if-statement. Instead of having to remove or insert the `DEBUG` define it could also have the value `0` for false and `1` for true. Having to only type a number makes it easier to toggle between modes.

2.1.3 Macros

Macros in `C` are used to substitute a word with an arbitrary list of other words called tokens. The preprocessor recognises a declaration of a macro by looking for the `#define` directive.
A macro rule can have the following forms:

```
#define <name> <replacement tokens>                 // Variable
#define <name>(<parameters>) <replacement tokens> // Function
```

The arguments of a function macro are taken as if they were variable macros and every use of a parameter is replaced with the corresponding argument. The replacement tokens of a macro can contain almost anything and do not have adhere certain form or sequence.

Listing 2.3 shows a variable macro named `PI`, which serves as an alias for π, and function macro, which is a wrapper for +.

Listing 2.3: Code with macros

```
1  #define PI 3.14
2  #define ADD(A, B) A + B
3
4  int main() {
5    float r1 = 1;
6    float c1 = 2 * PI * r1;
7
8    float r2 = 2;
9    float c2 = 2 * PI * r2;
10
11   return ADD(r1, r2) * ADD(r2, r1);
12 }
```

$\Longrightarrow$

Listing 2.4: Resulting code after macro substitution

```
1  int main() {
2    float r1 = 1;
3    float c1 = 2 * 3.14 * r1;
4
5    float r2 = 2;
6    float c2 = 2 * 3.14 * r2;
7
8    return r1 + r2 * r2 + r1;
9  }
```

As can be seen in Listing 2.4, all instances of `PI` are replaced with `3.14`, and `ADD(A, B)` is substituted to `A + B`. The examples show how simple and straightforward macros work. However the resulting code in Listing 2.4 contains a subtle bug. Macros `ADD(r1, r2)` and `ADD(r2, r1)` are substituted as expected but the resulting expression `r1 + r2 * r2 + r1` will probably not compute the value that was intended. Instead the output should be `(r1 + r2) * (r2 + r1)`. To ensure that this happens the replacement tokens of the macro use parenthesis like `(A + B)`. This unfortunate situation is caused by the fact that macros are replaced without taking precedence into account. It is not difficult to imagine the amount bugs this problem causes. As previously mentioned, the preprocessor in `C` can be seen as a separate entity from the rest of the compiler pipeline. This makes the problem even worse. When a bug occurs it will go unnoticed because technically it isn't invalid code. Above all, debugging the problem is a nightmare because setting breakpoints on implicitly generated code is pretty much impossible.

All of can make using macros extremely dangerous. The number of alternatives to macros maybe somewhat limited but in general they should suffice. They all revolve around the fact that in some instances macros are not strictly necessary. Macros just allow for ease of life for the programmer. When dealing with a value that is used in multiple places it is recommended to use a (static) variable. Compilers are smart enough to inline the use of constant values so there will be no overhead. This will also provide way better type safety. When dealing with function macros a possible alternative might be to write a regular function and denote it with the `inline` keyword. While this does not guaranty that the function will be inlined, the compiler will still check if it is possible to do so.

2.2 C++

The development on a successor to C called C++ was started in 1979 by Bjarne Stroustrup [15]. The language wants to solve the same problems as C but with more safety for the programmer. It could be argued that C++ is a superset to C, however this is not strictly true with even the Stroustrup admitting this [16]. However, the vast majority of features supported in C also work in C++. In terms of syntax C++ is almost identical to C with only the addition of a few keywords. Semantically, C++ adds some new constructs, some of which will be discussed shortly. Because of the vast amount of constructs that C and C++ share, porting a program from C to C++ is trivial.

Something to note is that preprocessor used for C is almost exactly the same as the preprocessor used for C++. No improvements have been made to it, so the same critique applies.

2.2.1 Templates

A feature that is new in C++ and that has no equivalent in C are templates. They enable **generic programming** which is a style of programming whereby types and values are not bound at the time of declaring a construct but at the time of using the construct. In the case of C++, templates are allowed on the declaration of functions, structs and classes. Use of templates in C++ is known as template metaprogramming.

In essence, templates allow the programmer to write a generic implementation of something. When the implementation is used the types are deduced and the generic type information is replaced with the actual type information. This means that the programmer is not required to declare multiple versions of the same function whereby only the parameter types and the return type differs. An example of this can be seen in Listing 2.5 where an implementation of `pow` is given which works for multiple types.

Listing 2.5: Code using templates

```
1  template <typename T>
2  T pow(T x, int n) {
3    if (n == 0) return 1;
4    return x * pow(x, n - 1);
5  }
6
7  int main() {
8    int   x = pow(4, 2);
9    float y = pow(0.5, 3);
10   return 0;
11 }
```

Listing 2.6: Implicit code after template deduction

$\Longrightarrow$

```
1  int pow(int x, int n) {
2    if (n == 0) return 1;
3    return x * pow(x, n - 1);
4  }
5
6  float pow(float x, int n) {
7    if (n == 0) return 1;
8    return x * pow(x, n - 1);
9  }
10
11 int main() {
12   int   x = pow(4, 2);
13   float y = pow(0.5, 3);
14   return 0;
15 }
```

It might look as if the compiler is only using substitution, similar to how macros work, to create the correct functions. This is however not the case. While macros are substituted with their corresponding tokens, templates require more analysis by the compiler in order to be resolved correctly. For example the argument `x` in the function call `pow(x, ...)` defines what function to generate and what type `T` should be. The compilers knows that in the case of `pow(4, 2)` the type of `x` is `int` and it generates a function with the correct parameter types. This is the core principle of metaprogramming at work! Because of the fact the the compiler is able to deduce the type of the arguments it is then able to create a function. All of this happens behind the scenes, the programmer does not notice that the compiler generates those functions. After these functions are generated they get merged with the rest of the processed source code and compiled further.

Templates also give more useful error messages when something goes wrong in comparison to macros. This is mainly because of the fact that resolving templates is part of the regular compiler pipeline and not done by a separate program. Templates require the compiler to deeply analyse the code. Calling the function `pow("Hello World", 42)` is nonsensical. The function that the compiler will generate will contain an error because the operator `*` is not defined for the `const char *` type, which is the type of the value `"Hello World"`. Because of the fact that the compiler applies analysis to the generated function and notices the error it is able to give a helpful message with both the location and source of the problem.

Another powerful feature of templates in C++ is that they can be used for compile-time programming. This is made possible by the fact that they also allow constant values as template arguments. These values are known at compile-time and thus allow the compiler to perform optimisations. Unfortunately, there is no universal way to use templates for compile-time programming. A bit of trickery might be required in some situations. Listing 2.7 and 2.8 show two different implementations to calculate x^n. Some examples are explained in depth in [14].

Listing 2.7: Calculate x^n using an `enum`

```
1  template <int x, int n>
2  struct pow {
3    enum { val = x * pow<x, n - 1>::val };
4  };
5
6  template <int x>
7  struct pow<x, 0> {
8    enum { val = 1 };
9  };
10
11 int main() {
12   return pow<4, 2>::val;
13 }
```

$\Longleftrightarrow$

Listing 2.8: Calculate x^n using `operator int()`

```
1  template <int x, int n>
2  struct pow {
3    operator int () {
4      return x * pow<x, n - 1>();
5    }
6  };
7
8  template <int x>
9  struct pow<x, 0> {
10   operator int () { return 1; }
11 };
12
13 int main() {
14   return pow<4, 2>();
15 }
```

Both examples calculate exactly the same value at compile-time using the `g++ v6.3` compiler whereby no run-time overhead is introduced. However Listing 2.8 does require the `-O1` optimisation flag to enforce no overhead during run-time. If the flag is not set then the value will still be calculated at run-time, which is not what was intended.

In Listing 2.7 an `enum` forces the compiler to calculate the value. It makes use of the fact that enums in `C(++)` can have a specified (integer) value. This value must always be known at compile-time in order to have a correct program. That is why Listing 2.7 does not require the optimisation flag. Note that the value in Listing 2.7 is retrieved by accessing the `enum` named `val` with the `::` operator while Listing 2.8 uses an alias operator for `int` in order to calculate the correct value. All in all, the ability to perform computation at compile-time with templates very valuable but does require some trickery to get the desired results. This is unfortunate because it makes using templates for compile-time programming somewhat inaccessible for programmers.

Still the power of templates are not restricted by this, they are in fact even Turing Complete [19]. It is possible to encode a working Turing Machine with the use of templates. This might imply that templates have enormous potential in what they can compute and what they can be used for. However, are still restricted by the way they are defined in the language. Templates are not really usable to do proper IO, which means they cannot load files from disk. They can only use typenames or constant values, so they in essence have no knowledge of a concept like IO. Given that restriction, templates are only really useful if used for mathematical calculations. Yet this has not stopped people from exploring their potential. A good example of this is the implementation of Tetris with templates [1]. While being a very cool example it does suffer from the problem described earlier: the code that makes it all possible is not very easy to understand and looks somewhat convoluted.

The syntax sometimes being convoluted is a problem, but that is not the only problem templates have in that area. Specifying template arguments can also lead to ambiguity issues. This is caused by the fact that template arguments use the < and > symbols. Doing a comparison in a template argument like `pow<4 > 5>()` is therefor not really possible.

Lastly, using templates can have an impact on compilation time. It is obvious that every computation that is performed during compilation increases the compilation time. However making the compiler implicitly generate declarations also has an impact on this. Additionally, templates are also very difficult to debug. Like with macros, placing breakpoints on implicitly generated code is difficult and inspecting a compile-time computation using templates is also virtually impossible.

2.2.2 constexpr

While templates are very helpful to reduce the amount of code a programmer has to write they are not as easy and obvious to use in order to perform computation at compile-time. Ideally a programmer would be able to do that with the same rules as when writing regular code. Not having to resort back to template trickery would be a major win. Compilers already perform optimisations and resolve some computation at compile-time. However, the programmer does not have much control on over this process. Furthermore there

are some instances where the compiler will refuse to optimise something even though this certainly possible. For example, things like recursion, iteration and branching are usually not optimised. This is not what is desirable because the programmer just wants to be able to write regular code and have it be computed at compile-time if there is need for that.

For that reason C++ introduced the `constexpr` specifier. Using `constexpr` tells the compiler that the code is guaranteed to be able to be computed at compile-time and asks the compiler to do so. Variables, functions and to some extend data structures can be denoted with `constexpr`. There are however some restrictions as to what can be done with `constexpr`. When `constexpr` was introduced in C++11 it only supported a very limited subset of functionality [2]. Only functions that did not contain any declarations, conditional statements or looping statements were allowed. As a result of that restriction, only recursion was available to do some basic computation. This was of course very restrictive, and while it was possible to create some simple functions for things like the Fibonacci sequence it still did not give full control to the programmer. Luckily this changed with C++14 which significantly relaxed these restrictions. Functions with `constexpr` were allowed to have declarations, `if`, `for` and `switch` statements.

To make use of `constexpr` a programmer should denote both the function and the target variable with the `constexpr` keyword. This will kick off the process of computing the value at compile-time. Values that are `constexpr` can also be used as template arguments. Listing 2.9 what code using `constexpr` looks like (in combination with templates to create a generic solution).

Listing 2.9: Use of constexpr to calculate x^n

```
1  template <typename T>
2  constexpr T pow(T x, int n) {
3      T res = 1;
4      for (int i = 0; i < n; i++) {
5          res *= x;
6      }
7      return res;
8  }
9
10 int main() {
11     constexpr int value = pow(4, 2);
12     return value;
13 }
```

The code in Listing 2.9 uses `constexpr` and calculates x^n in an iterative way at compile-time without using any special compiler flags. Sadly, `constexpr` cannot be used for everything. For example, it is possible to do some very basic string manipulation but actual memory allocations are not allowed. Like with templates, it is impossible to do IO and debugging. This really impacts the usefulness of the feature, and makes it difficult to use to solve non trivial problems.

2.3 D

The name of the programming language D suggests that it is a successor to C(++). Development started by Walter Bright in 2001. Like C(++), D is a system language aimed for low-level programming. While the syntax is not compatible with its main inspirator the goal was to have similar semantics for most situations. However, in some places D specifically chooses not to follow the route of C++ to save it from making the same mistakes. For example, D does not use a preprocessor which eliminates the problem of having a disconnect between two parts of the compilation process. In order to support conditional compilation and macros, D has to come up with other solutions. This also means that those solutions might not map one to one to the solutions used in C(++) which makes comparing them a bit tricky.

2.3.1 Conditional Compilation

To achieve conditional compilation, D exposes certain keywords like `version` and `debug` which can be used to annotate a scope that should be compiled depending on the version of the compiler or platform. Additionally, the `static if` keyword is used to decide if the statements in its scope are included in the code that will be compiled further. It must be possible for the condition to be evaluated at compile-time. An example of what this all looks like can be found in Listing 2.10.

Listing 2.10: Conditional compilation in D

```
1  import std.stdio;
2
3  int main() {
4    int a = 41;
5    const int b = 32;
6
7    // If compiled on Linux then increment a
8    version (linux) {
9       a = a + 1;
10   }
11
12   // If compiling a debug build then print "Hello World"
13   debug {
14     writeln("Hello World");
15   }
16
17   // If b is 16 or greater then set a to 255
18   static if (b > 16) {
19     a = 255;
20   }
21   return a;
22 }
```

2.3.2 Mixins

To some extend, mixins are D take on macros. Mixins allow for the embedding of code using string constants. Listing 2.11 shows what this looks like.

Listing 2.11: Simple example of a mixin

```
1 import std.stdio;
2
3 void main() {
4    mixin(‘writeln("Hello World!");‘);
5 }
```

However, the true power of mixins lies when using them in conjunction with compile-time programming. Since mixins use strings as their input and given the fact that D support compile-time manipulation of strings (explained in Section 2.3.4) it is possible to dynamically generate functions and data structures with them. Unfortunately, the system is not as intuitive to use if the goal is to just create a simple macro.

2.3.3 Templates

Templates in D try to be less complex and cumbersome to used in comparison to templates in C++. The first difference is in the syntax that is used to denote a template. With C++ this requires annotating a declaration with `template <...>`. To reduce verbosity, D does not require annotating a with a keyword. Instead it treats template arguments of a declaration very similar to regular parameters. An example of templates in D can be found in Listing 2.12

Listing 2.12: Example of templates in D

```
1  T exampleFunction(T)(T templateArgument, int normalArgument) {
2    return templateArgument + normalArgument;
3  }
4
5  class ExampleClass(T) {
6    T templateField;
7  }
8
9  int main() {
10   int val = exampleFunction(1, 41);
11   ExampleClass!(int) c;
12
13   c.templateField = val;
14   return c.templateField;
15 }
```

Listing 2.12 also show the difference in syntax for specifying template arguments. In order to not suffer from the same ambiguity problem that arises with C++ template arguments, D uses `!(...)` for template arguments.

A common problem when dealing with templates is that the template annotation is the same for multiple declarations. In C++ there is no easy way to annotate declarations with the same `template<...>` other than using macros. Instead D allows the programmer to define a named template "scope" with arguments where the declarations in the scope have access to the arguments. Additionally, in D it is possible to have variable declarations in a template scope which makes it possible to have actual generic variables which are not bound to a function or data structure.

Like C++, D allows templates to be used for compile-time programming. This system also works with `static if` because the values are known at compile-time. An example showing everything that has been explained thus far can be found in Listing 2.13. It shows a comparison as to how `factorial` can be implemented in D vs. C++.

Listing 2.13: Template to calculate factorial in D

```
1  template fact(int n) {
2    static if (n == 1)
3      const fact = 1;
4    else
5      const fact = n * fact!(n - 1);
6  }
7
8  int main() {
9    return fact!(5);
10 }
```

$\Longleftrightarrow$

Listing 2.14: Template to calculate factorial in C++

```
1  template <int n>
2  struct fact {
3    enum { val = n * fact<n - 1>::val };
4  };
5
6  template <>
7  struct fact<0> {
8    enum { val = 1 };
9  };
10
11 int main() {
12   return fact<5>::val;
13 }
```

Listing 2.13 shows what using a variable in a template scope looks like. Calling `fact!(5)` signals to the compiler that it must calculate the value for `fact`. Using `static if` for conditional compilation makes the compiler recursively call `fact!(...)`. Once the base case of `n == 1` is reached `1` is returned. The compiler folds the expression `n * (n - 1) * (n - 2) * ... * 1` and returns the final value.

Looking at how D implements templates, it has some clear advantages over how C++ implements them. To be able to declare a "scope" where functions within the scope have access to the same arguments greatly reduces typing. Templates in D also work nicely with conditional compilation thanks to `static if`. On top of that, they do not suffer from ambiguity issues in the syntax. Still, debugging templates can be difficult, and using them to perform computation at compile-time does not feel intuitive.

2.3.4 Compile-Time Function Execution

Compile-Time Function Execution or CTFE is a feature in D that allows the programmer to execute functions at compile-time. Instead of requiring special semantics via templates or something similar, CTFE requires no new concepts to perform computation at compile-time. When a function must be evaluated at compile-time, for example when a value must be known in order to do conditional compilation, then the compiler will (try to) execute that function. The `static` keyword before a variable declaration kicks off this process.

Functions that the programmer wants to execute at compile-time must follow some basic rules. For example they may not alter the global state of the program and thus are only allowed to change local variables. They can make use of conditional statements, loops and recursion. In Listing 2.15 CTFE is used to calculate the fifth Fibonacci number at compile-time.

Listing 2.15: Fibonacci using CTFE

```
1  int fibo(int n) {
2    if (n == 0) return 0;
3    if (n == 1) return 1;
4    return fibo(n - 1) + fibo(n - 2);
5  }
6
7  int main() {
8    static int res = fibo(5);
9    return res;
10 }
```

The restrictions that apply to compile-time programming in D are less strict in comparison to C++s `constexpr`. In addition to that, compile-time programming is more embedded in the ecosystem of D. For example, the standard library of D exposes a regular expression system which can specifically be used for compile-time programming. On top of that, D also has a special `import(<file location>)` keyword which can be used to load the contents of a file at compile-time into a string. This string can be further manipulated to do what ever is needed. Compile-time programming in D is even extra powerful because of the addition of mixins (Section 2.3.2) which means code can be generated dynamically at compile-time.

However, compile-time programming in D is restricted by the fact that doing real IO, like printing values, is not possible. In addition to that, debugging compile-time code is pretty much impossible, or at the very least somewhat intuitive. In that regard, it has the same problem as C++s `constexpr`.

2.4 Rust

The most modern programming language that will be analysed is Rust. It has been seeing an increase in popularity over the last few years. One of the major forces behind Rust is the company Mozilla which are mostly known for their web browser Firefox [12]. Like all other languages that have been analysed thus far, Rust is a system language. Its syntax somewhat resembles that of most C like languages. However, Rust introduces a wide range of new constructs which make its semantics vastly different compared to C(++) and D. The greatest focus for Rust is to provide better (memory) safety and better concurrency support.

Because Rust is a new language that is being built from the ground up it purposely chooses not to follow into the footsteps of its predecessors if that is not necessary.

2.4.1 Conditional Compilation

The way Rust allows for conditional compilation is by using both expressions that are evaluated at compile-time and by allowing annotations on functions. Listing 2.16 is taken from the Rust manual and shows both concepts in use. The if on line 17 will be resolved at compile-time resulting in the compiler optimising the branch away.

Listing 2.16: Conditional Compilation in Rust

```
1  // This function only gets compiled if the target OS is linux
2  #[cfg(target_os = "linux")]
3  fn are_you_on_linux() {
4      println!("You are running linux!")
5  }
6
7  // And this function only gets compiled if the target OS is *not* linux
8  #[cfg(not(target_os = "linux"))]
9  fn are_you_on_linux() {
10     println!("You are *not* running linux!")
11 }
12
13 fn main() {
14     are_you_on_linux();
15
16     println!("Are you sure?");
17     if cfg!(target_os = "linux") {
18         println!("Yes. It's definitely linux!");
19     } else {
20         println!("Yes. It's definitely *not* linux!");
21     }
22 }
```

2.4.2 Macros

One area that Rust improved upon over its predecessors is the area of macros. They took lessons from the mistakes that were made by C and C++ and their use of a preprocessor. An enormous amount of issues can arise by using a preprocessor so Rust drops the idea of using a preprocessor entirely. First of all, it is not possible to use macros to replace variables which means only function like macros (Section 2.1.3) can be used. The need for variable macros is very low since they can be replaced with regular global variables. A common source of errors with C(++) style macros is caused by the fact that a macro can be replaced with an invalid string of tokens or a string of tokens which result in ambiguity when resolved. These errors are not noticed until the macros have been substituted and will cause untraceable bugs and errors. Because of this Rust is more strict in the way it implements macros. It has a regular expression like language used to formulate both the type of arguments accepted and the way in which the substitution is resolved.

The example in Listing 2.17 is taken from the Rust manual [11] and shows how the vec! macro works.

Listing 2.17: Declaring the vec! macro

```
1  macro_rules! vec {
2      { $( $x:expr ),* } => {
3          {
4              let mut temp_vec = Vec::new();
5              $( temp_vec.push($x); )*
6              temp_vec
7          }
8      };
9  }
10
11 let x: Vec<u32> = vec!(1, 2, 3);
```

Listing 2.18: Generated code by the vec! macro

```
1 let x: Vec<u32> = {
2     let mut temp_vec = Vec::new();
3     temp_vec.push(1);
4     temp_vec.push(2);
5     temp_vec.push(3);
6     temp_vec
7 };
```

As can be seen in the examples, the syntax of Rust is quite a bit different compared to other languages. While covering all syntactic and semantic differences is beyond the scope of this example, some small things require a bit more explanation. Specifically the way Rust handles scopes. A scope can be denoted with curly braces like in most programming languages. However a scope can also be seen as an expression and thus must have a value. In Listing 2.18 on line 1 the variable x is being assigned the value that the scope after the = returns. On line 6 the value that will be associated with the scope is stated. This is might seem strange but suffice to know that this is possible in Rust and a common way of assigning values.

The vec! macro that is declared in Listing 2.17 on the first line creates a vector filled with its input arguments. Macros in Rust have a name and are invoked by using the name followed by an exclamation mark, seen on line 11 in Listing 2.17. The exclamation mark is there to distinguish between regular function calls and macro invocations. A macro declaration has two parts: a specification about what type of arguments it will accept hand how and a specification about what code it will (implicitly) generate when invoked. Both are separated by a => and use curly braces as delimiters, as can be seen on line 2. The first part has a $(...),*

which represents a matched argument. Within that argument a "metavariable" `$x` is bound to the current argument. The `:expr` part denotes that `$x` should be an expression. Note the use of the Kleene star operator `*` which means that that part should be matched zero or more times. The second part starts with a curly brace to denote a new scope. Then a new vector called `temp_vec` is created which will contain all values passed down by the arguments. After that another `$(...)*` can be seen which is related to the one declared in the first part. Using the `$(...)*` means that the code between the parenthesis must be repeated as many times as the `$(...),*` from the first part is matched. It also means that all arguments declared in the first part, in this case only `$x` can be used in the second part. This is how all arguments of the macro are inserted into `temp_vec` one by one. The result of this can be seen in Listing 2.18 on line 3, 4 and 5. Finally once all arguments are matched and processed `temp_vec` is returned. Then calling `vec!(1, 2, 3)` will generate the code seen in Listing 2.18.

Having the arguments of a macro type checked is a huge benefit over how `C(++)` does its things. It allows for better error messages when things go wrong but also require the programmer to think ahead of time how the macro is going to be used. In addition to that, having more control over the amount of macros that can be captured, and being able to use that amount to influence the way the macro is resolved is extremely powerful.

Additionally, `Rust` also addresses the precedence and hygiene issues present in `C(++)` macros. The issue of precedence is solved by implicitly wrapping the macro in parenthesis. To solve the problem of not having hygienic macros whereby it is whereby it is possible to accidentally have an identifier declared outside the macro clash with one from the inside. Macros in `C(++)` are not designed to be hygienic macros, while macros in `Rust` are. Listing 2.19 and 2.20 show what this looks like.

Listing 2.19: Bug caused by variable `state`

```
1  #define LOG(msg) { \
2      int state = get_log_level(); \
3      if (state > 0) { \
4          printf("log(%d): %s\n", state, msg
               ); \
5      } \
6  }
7
8  int main() {
9      const char *state = "Some Message";
10     LOG(state);
11 }
```

Listing 2.20: No bug thanks to hygienic macros

```
1  macro_rules! log {
2      ($msg:expr) => {{
3          let state: i32 = get_log_level();
4          if state > 0 {
5              println!("log({}): {}", state,
                   $msg);
6          }
7      }};
8  }
9
10 fn main() {
11     let state: &str = "Some Message";
12     log!(state);
13 }
```

Listing 2.19 contains a bug. The macro `LOG(msg)` is called with a variable named `state`. Because of macro substitution all occurrences of the word `msg` are replaced with `state`. Problem is that there is already a variable named `state` declared on line 2. Now the value of the integer `state` from line 2 will be printed instead of the argument `state` as intended on line 10. This problem cannot arise in `Rust` because of its macro system. All parameters of a macro only specifically refer to the correct scope. No clashing of identifiers can occur.

2.4.3 Generics

Like many languages, Rust also has support for generics. Since the concept has been explained already (Section 1.3 and 2.2.1) it will not be explained again. In comparison to C++ and D, Rust does not allow for using generics to do computation at compile-time. Generics in Rust can only be used in conjunction with types, not with values. Other than that, there is only one difference with generics in comparison to other languages. The operations that are allowed on something of a generic type are a bit more restricted than in other languages. In C++ it is valid to call an arbitrary method on a variable of a generic type without knowing ahead of time if that method is present. The compiler will check at compile-time if that is the case, and if not an error will be thrown. Instead of that, Rust forces the generic type to specify that the methods it has. This is done via the concept of *traits*. Listing 2.21 and 2.22 show a comparison between C++ and Rust.

Listing 2.21: Valid use of generics in C++

```
template <typename T>
void doSomething(T value) {
    value.someMethod(42);
}
```

$\Longleftrightarrow$

Listing 2.22: Invalid use of generics in Rust

```
fn doSomething<T>(value: T) {
    value.someMethod(42);
}
```

The code in Listing 2.21 will compile without problems, at least if `someMethod(int)` can be called on something of type `T`. However the code in Listing 2.22 will throw a compilation error because there is no guarantee that something of type `T` can call `someMethod(42)`. To solve this, a trait must be defined which says that the method is present on the type and that trait must be used on `T`. This looks like the code in Listing 2.23.

Listing 2.23: Use of generics in Rust together with a `trait`

```
trait HasSomeMethod {
    fn someMethod(&self, i32);
}

fn doSomething<T : HasSomeMethod>(value: T) {
    value.someMethod(42);
}
```

It could be argued that this choice is very arbitrary and does not give more power to the programmer. While it is true that it guarantees some safety it does not give more safety over how C++ does its things. As said previously, if the function from Listing 2.21 is called with an invalid type the compiler will still complain. Forcing the programmer to declare what the properties of a generic will be defeats the point of generics. If more security is needed to say what can and cannot be done with an object then one should look at things like inheritance and composition.

2.5 Other languages

By now, most modern (system) programming languages have been analysed. There are however many more programming languages which have (or purposely omit) features that are powered by metaprogramming. Giving an in depth analysis for these languages would be extremely boring for both the readers and the writer of this thesis. Yet omitting some of these languages leaves out interesting information in the discussion about metaprogramming in modern programming languages. For this reason from now on only interesting features from programming languages will be looked at in a shorter and more concise form.

2.5.1 Go

Another fairly new programming language is `Go`, which is developed by Google. Development was started by Robert Griesemer, Rob Pike and Ken Thompson in 2007 and became public in 2009 [3]. It was born out of frustration with existing programming languages, especially in the area of system programming. Languages like `Python` and `Javascript` sacrifice safety and performance for ease of use. With `Go` the goal was to eliminate this sacrifice and provide a best of both worlds experience. Nevertheless `Go` is very similar to other `C` like languages in terms of syntax and also being a statically typed, compiled language.

There are many things that `Go` does different in comparison to its competitors. It uses a garbage collector, to make the programmer not have to worry about memory management, but at the cost of performance. It does not support classical object-oriented programming and favours composition over inheritance. Most notably `Go` **does not support** ***generic types***. This is a very unpopular choice and one of the main criticisms against `Go`. They are omitted because they would introduce complexity in the type system and at run-time [4]. On top of that similar functionality can be achieved with their current type system. However using their type system for generic programming is very cumbersome and far from a pleasant experience. The result is that the programmer has to maintain multiple versions of functions and has to do a lot of code duplication. Finally `Go` does not feature proper conditional compilation or compile-time function execution. There is no specific reason why these features are not in the language, probably because they are no top priority.

2.5.2 Python

A language that is closer to `Go` than to `C` is `Python`. It was developed by Guido van Rossum in 1991, meaning that is has been around for a long time. Unlike all other languages it is not a compiled language. Instead the source code is translated to byte code which is then executed by an interpreter which uses just-in-time compilation. It is not really suited for system programming and is more used for web development or for scientific purposes. The consensus among programmers is that `Python` is more easy to use than other programming languages because the programmer does not have to worry about so many things. Things like memory management are taken care of by the garbage collector. In addition, the validity of types in the program is checked at run-time. Programmers do not have to specify the type of a variable ahead of time.

For this reason generic types have no place in `Python`. And because the language is not compiled things like conditional compilation and compile-time function execution are also not available.

However because everything happens at run-time, `Python` has more information available when executing the program. This information can be used by the programmer to create things that are not possible in other languages. A good example of this are *decorators* which are shown in Listing 2.24.

Listing 2.24: Decorating a function

```
1  def Log(func):
2    def wrappedFunction(*args):
3      print("Calling function '" + func.
           __name__ + "'")
4      return func(*args)
5    return wrappedFunction;
6
7  @Log
8  def addition(a, b):
9    return a + b
10
11 @Log
12 def double(a):
13   return 2 * a
14
15 def main():
16   a = 41; b = 1
17   c = addition(a, b)
18   d = double(c)
19   print("C is", c, "D is", d)
20
21 main()
```

Listing 2.25: Resulting output

$\Longrightarrow$

```
1  Calling function 'addition'
2  Calling function 'double'
3  C is 42 D is 84
```

In Listing 2.24 a function is decorated with a log decorator. A decorator is nothing more than another function which in this case is declared on the first line. The decorator has a parameter which is the function that it decorates, in the example the parameter is named `func`. It returns a replacement function and what that function does is up to the programmer. The example in Listing 2.24 prints the name of the original function before calling the original function. This can be very valuable when needing to debug something. Instead of having to manually write multiple print statements in different places it is enough to decorate a function with the log decorator to know when the function is being called. The decorator has access to the name of the original function and can use this information to print.

This process is not really code generation, because it all happens at run-time. Yet it does closely resemble code generation and it bring up the question why this kind of functionality is not available in other programming languages.

2.5.3 Jai

The newest language that deserves attention is Jai. It is in fact so new that at the time of writing this thesis, it is not available for the general public. The creator of Jai is Jonathan Blow. He is a well-respected game developer with years of experience and created *Braid* and *The Witness*. Its main goal is to be a better system programming language compared to C and C++, especially in the area of game development. As said, Jai is not available for use just yet, however Jonathan Blow has been sharing the process of creating Jai on the internet, also proving that it is more than just an idea. Most of this can be found on his YouTube channel [8]. A good summary of the language is given in a talk [7] by Blow about how Jai will make programming less terrible.

With respect to metaprogramming, Jai introduces some very interesting features not found elsewhere. To start, Jai features **proper *compile-time function execution***. Instead of having to mangle code that should execute at compile-time with code that executes at run-time, it is possible to specify a "main" entry point that will be called at compile-time. But that is not all, normally compile-time function execution is extremely limited in terms of what it can do. All previous languages have restrictions that prevents them from doing IO. Instead Jai drops this restriction and goes even further by ensuring that **everything that can be done at run-time can also be done at compile-time**. This puts Jai miles ahead of all other languages in terms of what can be done at compile-time. It is even possible to play a video game at compile-time as Blow demonstrated with space invaders.

Another area where Jai improves over its counterparts is with generic programming. While most implementation of generic programming in programming languages suffice, they do suffer from some syntactic overhead. Albeit in C++ needing to declare a function function with a template specifier or in Rust with traits. The solution that Jai proposes is very simple: when a type is not known at declaration it should be prefixed with a $.

```
1 some_function :: (param1: $T, param2: T) -> T {
2     return param1 + param2;
3 }
```

Here a function some_function is declared. The type of the first parameter is $T which means it is unknown. Further uses of T refer back to the original $T. No special syntax or semantics is required to perform generic programming. In Jai functions using this technique are called polymorphic procedures [6].

Lastly, Jai also makes it possible to introspect the compiler when it is compiling the source code. It exposes hooks that allows programmers to receive information about the code that it is processing. With that information it is then possible to generate more code or do some type of validation. This is especially interesting because it enables the implementation of decorators found in Python at compile-time.

2.5.4 Sparrow

The last languages that will be looked at is Sparrow. It is different compared to other programming languages because of the fact that it has a strong academic backing. Programmers can use Sparrow right now but it still has not seen widespread adoption. Still, it is a complete programming language and compares itself with languages like C++. The language was presented in a paper from Lucian Radu Teodorescu et al [10].

The authors of the language claim that Sparrow allows for "hyper-metaprogramming" [18]. In languages that support hyper-metaprogramming it is possible to move a computation from run-time to compile-time regardless of its complexity. Additionally, the syntax that is used to do compile-time programming is the same as is used with run-time programming. The programmer is able to make a distinction between what code should be executed at compile-time and what code should be executed at run-time by using annotations of the form `[rt]`, `[ct]`, `[rtct]` and `[autoct]`. These annotations are placed on a declaration (function, variable etc.) and denote if the declaration can only be used at run-time, only at compile-time, both at run-time and at compile-time, or if the compiler should infer if the declaration can be used at compile-time. By default declarations are annotated with `[rt]` and literals are annotated with `[ct]`.

An example, given by the language authors in [17], of the ability to determine what code should be executed at run-time and what at compile-time can be seen in Listing 2.26.

Listing 2.26: Factorial in Sparrow

```
1  fun [autoCt] fact (n: Int): Int {
2      if ( n == 0 ) return 1;
3      else return n * fact (n - 1);
4  }
5
6  fun testFact () {
7      var n = 5;
8      writeLn(fact(n)); // Runs at run-time
9      writeLn(fact(5)); // Runs at compile-time
10 }
```

The `[autoct]` on line 1 results in the fact that the call to `fib` on line 8 is performed at run-time because `n` is (implicitly) annotated with `[rt]` meaning that it cannot be used at compile-time, and that the call on line 9 is performed at compile-time because in this case the value of the argument is known at compile-time.

Now, this system whereby the programmer can decide what is executed at run-time and what is executed at compile-time is also found in other languages. It is almost identical to `constexpr` used in C++, which is explained in Section 2.9. To an extend, the system is also similar to that used in D.

Now to further elaborate on the differences. Both C++ and Ds implementations are, as discussed, somewhat restrictive. All three languages support the use of if-statements, recursion and iteration at compile-time. The difference lies within the fact that those operations are not always supported on all datatypes. In C++ many

(if not all) datastructures and algorithms of the standard library cannot be used at compile-time. Most of them perform memory allocations which is not allowed. This means that most code will not automatically work when executed at compile-time. The system `D` is less restrictive because it does support more operations on datatypes. String manipulation is much easier, and is even required in order to make full use of mixins (Section 2.3.2). Still, the claim that `Sparrow` makes that it supports hyper-metaprogramming should make its compile-time programming facilities more powerful than others. It should allow for any type of computation to be done at compile-time. This means that virtually anything can be performed at compile-time.

Like all other languages, `Sparrow` supports conditional compilation. Its implementation is similar to that of `D` and its use of `static if`. In the case of `Sparrow` the `if` is annotated with a `[ct]` to indicate that it should be evaluated at compile-time. This annotation can also be placed on `for`-loops and `while`-loops. What this looks like can be found in 2.27.

Listing 2.27: Conditional compilation in Sparrow

```
1 fun main () {
2   if[ct] (<compile-time condition>) {
3     ...
4   }
5
6   for[ct] (<iterator>) {
7     ...
8   }
9 }
```

Finally, the macro system used in `Sparrow` works on AST nodes, just like in `Rust`. Also like `Rust`, it has a trait-like generic system but it is possible to make complete generic implementations by using the special `AnyType` which accepts anything. It also supports some introspection in the compilation process and into the compiler, but not at the scale of `Jai`.

Chapter 3

Feature Summary

A short summary of all features that were analysed will be given. First, a complete table of all topics will be presented after which each topic will be discussed separately. Note that `Go` and `Python` will be omitted (from the tables) because they do not have any interesting properties useful for this summary.

3.1 Overview

		C	C++	D	Rust	Jai	Sparrow
Conditional Compilation	Implemented using	Prep	Prep	CL	CL	CL	CL
	Usable with CTP	No	No	Yes	N/A	Yes	Yes
Macros	Implemented using	Prep	Prep	CL	CL	-	CL
	Possible hygiene issues	Yes	Yes	N/A	No	-	No
	Token substitution based on	Char	Char	Str	AST	-	AST
	Arguments are pattern matched	No	No	N/A	Yes	-	Yes
	Possible ambiguity issues	Yes	Yes	N/A	No	-	No
Generic Programming	Possible ambiguity issues in syntax	-	Yes	No	No	No	No
	Support a ”generic” scope	-	No	Yes	No	No	No
	Trait-based generic arguments	-	No	No	Yes	No	Optional
	Require keyword for generics	-	Yes	No	No	No	No
Compile-Time Programming	Requires special keyword to start CTP	-	Yes	Yes	-	No	Yes
	Exposes compile-time entry point	-	No	No	-	Yes	No
	Can do compile-time programming with templates	-	Yes	Yes	-	No	No
	Compile-time programming is debugable	-	No	No	-	Yes	No
	Compile-time programming as powerful as run-time	-	No	No	-	Yes	No
	Able to inspect the program during compilation	-	No	No	-	Yes	Limited

Abbreviations	
Prep	Preprocessor
CL	Core Language
N/A	Not Applicable

Some languages do not support certain features. When that is the case the entry in the table is marked with -. It is also possible for a language to support a certain feature, but only to a certain extend. This means that some aspects of a topic might not be applicable to the language. When that is the case the entry in the table is marked with N/A.

3.2 Conditional Compilation

		C	C++	D	Rust	Jai	Sparrow
Conditional Compilation	Implemented using	Prep	Prep	CL	CL	CL	CL
	Usable with CTP	No	No	Yes	N/A	Yes	Yes

Conditional compilation is possible in all languages in some shape or form. Both `C` and `C++` use a preprocessor to enable conditional compilation. All other languages use semantics that are defined within the language. The way `C(++)` implements conditional compilations makes it impossible to interface with other parts of the language. In `D`, `Jai` and `Sparrow` it is possible to do conditional compilation entirely using compile-time programming. Additionally, `Rust` does not support compile-time programming and thus the system cannot be used in that way.

3.3 Macros

		C	C++	D	Rust	Jai	Sparrow
Macros	Implemented using	Prep	Prep	CL	CL	-	CL
	Possible hygiene issues	Yes	Yes	N/A	No	-	No
	Token substitution based on	Char	Char	Str	AST	-	AST
	Arguments are pattern matched	No	No	N/A	Yes	-	Yes
	Possible ambiguity issues	Yes	Yes	N/A	No	-	No

All languages support macros except for `Jai`. This is likely caused by the fact that it has been no priority to implement it yet. Presumably, it will be added in the future. Like with conditional compilation, the way in which macros are implement differ per language. Again, `C` and `C++` use a preprocessor and all other languages use language semantics. However, unlike conditional compilation, the way macros work in a language differs quite a bit.

The way `C(++)` does its things is by doing straight substitution of characters. This can lead to ambiguity and hygiene problems as can be seen Listing 2.3. One of the most interesting takes on macros comes from `D`, explained in Section 2.3.2. The system is not really comparable to that of the rest, but it does allow for macro-like functionality. It allows for code generation via strings, which can be mutated with compile-time programming. Because of that, some of the feature comparisons do not apply to `D`. The way `Rust` implements macros, detailed in Section 2.4.2, is arguably one of the best. Arguments of macros can be a specific type of node from the AST and can be pattern matched using a regular expression like language. Like `Rust`, `Sparrow` also works by taking AST nodes as arguments. However, its system for resolving macros is not as complex.

3.4 Generic Programming

		C	C++	D	Rust	Jai	Sparrow
Generic Programming	Possible ambiguity issues in syntax	-	Yes	No	No	No	No
	Support a ”generic” scope	-	No	Yes	No	No	No
	Trait-based generic arguments	-	No	No	Yes	No	Optional
	Require keyword for generics	-	Yes	No	No	No	No

Generic programming, named template programming in `C++` and `D`, is often available for languages not that are not dynamically typed. As explained in Section 2.2.1, the way `C++` implements generic programming is somewhat of a mess. It is very difficult to debug implicitly generated code. Templates in `C++` also suffer from ambiguity problems caused by the symbols that are used. The language `D` improves on the system that was introduced by `C++`. It removes any possible ambiguity that can arise by using a different syntax. On top of that, `D` introduces ”generic” scopes. This is a nice facility to be able to reuse generic arguments and as a result reduces the amount of typing that has to be done. The system that `Rust` and also `Sparrow` uses for generic programming differs quite a bit from the others. They enforce that generic arguments adhere to a trait which defines what properties the argument has. This is system somewhat defeats the purpose of generic programming and requires the programmer to do a lot more work for not really much benefit. This is explained further in Section 2.23. Finally, `Jai` uses a nice system for generic programming which is both very low on syntactic overhead and very flexible in use because it does not use a trait system.

3.5 Compile-Time Programming

		C	C++	D	Rust	Jai	Sparrow
Compile-Time Programming	Requires special keyword to start CTP	-	Yes	Yes	-	No	Yes
	Exposes compile-time entry point	-	No	No	-	Yes	No
	Can do compile-time programming with templates	-	Yes	Yes	-	No	No
	Compile-time programming is debugable	-	No	No	-	Yes	No
	Compile-time programming as powerful as run-time	-	No	No	-	Yes	No
	Able to inspect the program during compilation	-	No	No	-	Yes	Limited

Both `C++` and `D` support compile-time programming using templates. However, doing so is convoluted and means a special syntax must be used which is not intuitive. The languages `C++`, `D` and `Sparrow` all have similar compile-time programming system which are accessible with conventional syntax. To start the compile-time computation it is necessary to annotate a declaration with a special keyword. The place of this keyword differs per language. More details on this can be found in Section 2.2.2, 2.3.4 and 2.5.4 respectively. All tree languages have some restrictions over what they can do. The language `Sparrow` claims that is the least restrictive because it supports ”hyper-metaprogramming”. Still, doing things like IO or debugging is impossible for all of these languages. It is here where `Jai` is head and shoulders above the rest, as shown in Section 2.5.3. It allows for everything that can be done at run-time also be done at compile-time. Furthermore, it exposes a special entry point which is called at compile-time meaning that compile-time code and run-time code do not have to be mixed. Finally it also allows for the inspection of the program at compile-time. This enables features like decorators also seen in `Python` (Section 2.5.2).

Chapter 4

Conclusions

After a thorough analysis of all modern programming languages it is now time to look at how features enabled by metaprogramming will develop in the future.

Preprocessor

To start, it is safe to assume that (new) programming languages will and should not depend on a separate **preprocessor**. Of all programming languages that were analysed only `C` and `C++` make use of such a tool. And it could be argued that `C++` has this dependency because of the history that it shares with `C`. The languages heavily depend on the preprocessor. Without one things like `#include` but also macros and conditional compilation would not be possible. However, it is very clear that for other things the preprocessor is not required. The end of Section 2.1.3 gives more explanation as to why the preprocessor causes so many problems and gives some alternatives. Nevertheless, it is fairly obvious to see what the short comings of such a tool are. The preprocessor causes a lot of problems because of the fact that it is not neatly integrated in the compiler pipeline. It is a separate entity and very little communication is done with later stages of pipeline, leading to valuable information being lost. This introduces nasty and hard to trace bugs in the code with no easy way to determine the cause of the problems. The preprocessor also does not enable special features that otherwise would not be possible. All functionality that a (separate) preprocessor enables can be implemented in a better way by other languages, as is demonstrated.

Conditional Compilation

On the topic of **conditional compilation** no new ideas have been introduced. This is not necessarily a bad thing. The way `C` and `C++` is not very flexible because it does not interface with other parts of the language, specifically compile-time programming. Some improvements can be found in `D`. It uses a syntax that is part of the language so using it does not feel out of place. In addition to that, the system interfaces nicely with the system used for compile-time programming. A good example of this can be found in Listing 2.10.

No other big improvements can be found. For the most part, Rust sticks to a very simple system which nicely fits within the language. Conditional compilation in Jai and Sparrow also works well with the compile-time programming system, solidifying the claim that these two systems must work together in a sensible manner.

Macros

Because of the fact that **macros** in C and C++ are implemented using a preprocessor the system has many problems. The substitution system that is used is also very poor. As is explained in Section 2.1.3, macros get resolved without any regards to the abstract syntax tree. Any errors that are introduced because a macro cannot be detected because the compiler is not aware that the code has been altered by a preprocessor. In addition to that, macros in C(++) are also very sensitive to ambiguity issues, as can be seen in Listing 2.3.

What is more interesting is the way in which way programming languages after C(++) chose to implement macros. The language D tries to improve by allowing string literals to be used as macro output. This is more robust and, when used with compile-time programming, can create some impressive results. Still the system does not feel as flexible as it could be. This is where Rust comes around and changes the idea of using macros in programming languages completely. Because of the fact that macros in Rust depend on the abstract syntax tree as their input, substituting macros is much more stable. Programmers can clearly state what the arguments to a macro must be and how the macro can be resolved. The example in Listing 2.17 shows this impressive system in use. Above all, macros in Rust are hygienic and not sensitive to ambiguity issues. The industry has clearly learned from the mistakes that were made and it has introduced a solution that is very solid and that programmers actually benefit from.

Generic programming

Almost all languages that have been analysed have the ability to write an implementation of a function or data structure in a generic way whereby the types are unknown at the time of declaration. Only after instantiating the function or structure the types are determined. Benefits of this style of programming are that the programmer only has to maintain one version instead of multiple with different types. This reduces the amount of code the programmer has to write and thus reduces the error potential in the code. The necessity for such a feature is also made clear by the criticism that languages receive when not having support for generic programming. A good example of this is Go whereby the language lacks generics. This is a major mark against Go and undoubtedly impacts the adoption of the language.

How generics should be implemented is another problem. The way C++ implements generics is fairly self explanatory and easy to use. Downside is that error messages can get out of hand for unexpected reasons when dealing with templates. Furthermore using angle brackets feels a little bit awkward and the restriction that template functions cannot be declared and defined separately can give annoying linker issues. More criticism of this system can be found in Section 2.2.1. The improvements that D makes over C++ templates are good, but do not solve all issues. Having the ability to reuse template types within a ”scope” helps with

reducing the amount of code a programmer has to write. Still, in most languages there it feels like there is a disconnect between the part of the language for generics and the rest of the language. This problem continues with Rust. In an attempt to make generics more "safe" by restricting what can be done with something of generic type, Rust makes working with generics more difficult, something which is elaborated on further in Section 2.4.3. Making generics safer is not necessarily a bad thing, but doing so at the cost of usability is probably a bad choice. This leads to programmers having to do more work and gaining very little. In addition to that, while Sparrow does allow declarations to be as "generic" as in C++ it still has a trait-like system which means the same criticism applies.

So there are no major programming languages that implement generics in a satisfying way. However, with the knowledge from Section 2.5.3 it is easy to see that Jai might be the closest to solving this problem. By introducing very little syntactic overhead for using generics and not applying unwanted restrictions, Jai provides a solution that is very easy to use and that is very powerful in the things that can be achieved with it.

Compile-Time Programming

The discussion around compile-time programming is fairly new, especially compared to all previously looked at topics. However, the need for a good solution for compile-time computation is steadily increasing. This can be seen by both the community of programmers asking for these features and languages following this demand.

The discussing around this is also somewhat related to the discussion of *generic programming*. In some way it is very strange that generic programming is related to compile-time programming. They both aim solve entirely different problems. However, in the past these two topics have been intertwined. The primary cause of this is the way generics work in C++. In C++ templates can also be used to perform computation at compile-time, as is shown in Listing 2.7. This is also possible in D.

Yet using templates for compile-time programming comes at a cost. Firstly, using templates for compile-time computation does not feel like an obvious choice. Similar to using templates for generic programming, there is a disconnect between using templates and doing regular programming. And this disconnect is even larger when using templates for compile-time computation. In regular programming things like a while-loop or an if-statement are trivially easy to use because they are part of the language. Those constructs are not really available when using templates and thus programmers have to resolve to different means to get similar behaviour. The price that the programmer must pay for using templates to perform computation at compile-time is that the whole endeavour becomes very cluttered and unpleasant. On top of that, compile-time programming using templates is also very restrictive. There is no possibility for doing IO, no way to debug what is going on and no precise control over how the compiler will resolve the computation. In the end, the programmer is not really in control over the situation which is very problematic. That is why performing computation at compile-time using templates is not on the level that it should be, and that using templates to solve the problem of compile-time computation is probably a bad idea.

Another solution to the problem of compile-time computation is only coming up in the last few years. It should be possible to solve the problem by using regular syntax which is part of the core language. For this reason `D` introduced *Compile-time Function Execution* and `C++` followed sometime after that with `constexpr`, detailed in Section 2.3.4 and 2.2.2 respectively. The goal of these two constructs is to allow compile-time computation using the same principles when writing regular code. This is a good thing and definitely pushes the discussion forward. However, both implementations for compile-time computation from `D` and `C++` are very restrictive. Like with templates, the solutions suffer from the restriction that it is not possible to do IO or debugging. It is still very much a black box whereby it is difficult to asses what is going on. Using compile-time functions is also a bit tedious. Code that is executed at compile-time is mixed with code that will execute at run-time. This mix results in having compiler errors because code for compile-time called code for run-time, without really knowing what and where things went wrong. The language `Sparrow` tries to expand the horizon on what is possible with compile-time programming. However it fails in doing so because it does not really introduce new ideas.

For a good solution on how to implement compile-time programming it is necessary to look into the future. Again, `Jai` is leading in this area. It drops all restrictions concerning what can and cannot be done at compile-time. Programmers should have full control by having access to IO and debugging. Moreover, compile-time and run-time code should not be mixed in a way that leads to spaghetti code. This is solved very cleanly by `Jai` by introducing a compile-time entry point similar to a run-time entry point.

In addition to that, `Jai` allows for advanced introspection into the compilation process. Introspection is common for languages like `Python` but not for languages that are statically compiled. The absence of such a feature might wrongfully imply that it is not possible to support it so it is a good thing to see progress.

Summarising

The features that are enabled by metaprogramming have on the one side made clear steps in solving problems and improving old solutions but on the other side is still figuring out what the best direction is to go into when dealing with other problems. Things like macros and conditional compilation have been used for many years and their implementations have had necessary revisions. Now after all these years it is very clear what the problem is that should be solved and what the best way is to solve it. Generic programming has also been available for a very long time. Its problems however, have been more difficult to solve. When using generic programming for just generic programming, then the current solutions definitely suffice. Once generic programming is mixed with compile-time computation things become muddled. This should be avoided. Luckily this is slowly happening by both making the problems that generic programming tries to solve more strict and introducing better ways to perform computation at compile-time. However, the area of compile-time computation is also one of the newest. Only in the last few years programming languages have come with solutions that suffice the need to do computation at compile-time using both familiar syntax and semantics. Yet there are restrictions to those solutions but with the direction that the industry is heading it is only a matter of time before those hurdles are overcome as well.

Bibliography

[1] Matt Bierner. Super template tetris. http://blog.mattbierner.com/stupid-template-tricks-super-template-tetris/, 2015.

[2] Gabriel Dos Reis, Bjarne Stroustrup, Jens Maurer. Generalized Constant Expressions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2235.pdf, 2007.

[3] Go. History of Go. https://golang.org/doc/faq#history.

[4] Go. Reasoning behind the omission of generics in Go. https://golang.org/doc/faq#generics.

[5] IBM Jonathan Bartlett. The art of metaprogramming. https://www.ibm.com/developerworks/library/l-metaprog1/, 2005.

[6] Jonathan Blow. Demo of generic programming in Jai. https://www.youtube.com/watch?v=BwqeFrlSpuI.

[7] Jonathan Blow. Making Game Programming Less Terrible. https://www.youtube.com/watch?v=gWv_vUgbmug&t=64s.

[8] Jonathan Blow. YouTube page of Jonathan Blow. https://www.youtube.com/user/jblow888/videos.

[9] Bell Labs. The development of the c language. http://www.bell-labs.com/usr/dmr/www/chist.html, 1993.

[10] Lucian Radu Teodorescu et al. Sparrow: Towards a New Multi-Paradigm Language. http://www.lucteo.ro/media/Sparrow%20Towards%20A%20Multiparadigm%20Language.pdf.

[11] Mozilla. Rust Manual about Macros. https://doc.rust-lang.org/book/macros.html.

[12] Rust. Rust FAQ. https://www.rust-lang.org/en-US/faq.html.

[13] Tim Sheard. *Accomplishments and Research Challenges in Meta-programming*, pages 2–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[14] Jeremy Siek and Walid Taha. A semantic analysis of c++ templates. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 304–327, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] Bjarne Stroustrup. Bjarne Stroustrup's personal page. http://www.stroustrup.com/bs_faq.html#invention.

[16] Bjarne Stroustrup. Is C a subset of C++. http://www.stroustrup.com/bs_faq.html#C-is-subset.

[17] L. Teodorescu and R. Potolea. Compiler design for hyper-metaprogramming. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 201–208, Sept 2013.

[18] Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. Moving computations from run-time to compile-time: Hyper-metaprogramming in practice. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 17:1–17:10, New York, NY, USA, 2014. ACM.

[19] Todd L. Veldhuizen. C++ Templates are Turing Complete. https://netvor.sk/~umage/docs/3rdparty/C++%20Templates%20are%20Turing%20Complete%20(Todd%20L.%20Veldhuizen,%202003).pdf, 2003.