The MYLL Programming Language: C++ Resyntaxed and Extended

Eine Weiterentwicklung der C Style Syntax

zur Erlangung des Grades

Master of Science – Informatik

an der

Hochschule Niederrhein

Fachbereich Elektrotechnik und Informatik

vorgelegt von

Jan Helge Reitz

Geboren 1978-10-06 in Moers, Deutschland

Matrikelnummer: **756470**

– jan@sordid.de – @JanSordid –

Datum: 2020-07-03

Prüfer: Prof. Dr. Jochen Rethmann

Zweitprüfer: Prof. Dr. Steffen Goebbels

Zusammenfassung

Ich schlage eine neue Syntax für C++ vor, welche den Umgang mit C++ unproblematischer und den Einstieg erleichtern soll.

Die von mir vorgeschlagene neue Syntax für C++ hat den Codenamen **Myll** *(frei von My Language)*. Der Fokus des Sprach-Designs balanciert zwischen zwei grundlegenden Zielen; zum einen soll er alteingesessene C++-Programmierer abholen und dessen Arbeit in Zukunft leichter gestalten und zum anderen soll es Neulingen den Einstieg, in eine immer komplizierter werdende Sprache, erleichtern.

Um dieses Ziel zu erreichen habe ich viele Sprachkonstrukte auf die Probe gestellt und mir überlegt wie man diese besser gestalten kann, teils von anderen Sprachen inspiriert, aber auch komplett selbst erdacht.

Des Weiteren soll Myll einfach ergänzbar sein: Beispielsweise nutzen QT, UE4 und viele andere Bibliotheken *Makros*, welche an den richtigen Stellen in den Code eingesetzt werden und teilweise durch einen zusätzlichen Präprozessor ausgeführt werden. Diese verstreuten Makros können in Myll etwa durch ein Klassen- oder Variablen-Attribut ersetzt werden.

Umgesetzt ist Myll als ein *Language to Language* Compiler, auch Transpiler genannt. Dieser erzeugt als Ausgabe keinen Maschinencode, sondern Quellcode einer anderen Programmiersprache, in diesem Fall C++.

Dies soll ein *hackbarer* Transpiler sein; einfach herunterladen und die eigene Sprache erfinden!

Abstract

I propose a new syntax for C++, which should make the general usage of C++ less problematic and also make it easier to get started.

The new syntax for C++ proposed by me has the code name **Myll** *(free from My Language)*. The focus of the language design balances between two basic goals; on the one hand, it should pick up established C++ programmers and make their work easier in the future, and on the other hand, it should make it easier for newcomers to get started with a language which has become more and more complicated.

In order to achieve this goal, I have put many language constructs to the test and thought about how they could be better designed, partly inspired by other languages, but also completely conceived by myself.

Furthermore Myll should be easy to extend: For example QT, UE4 and many other libraries use macros which are inserted at the right places in the code and are partially executed by an additional preprocessor. These scattered macros can be replaced by an attribute on a type or variable in Myll.

Myll is implemented as a *language to language* compiler, also called a transpiler. This compiler does not produce machine code as output, instead it produces source code of another programming language, in this case C++.

This Transpiler is meant to be *hackable;* download it and invent your own language!

Inhaltsverzeichnis

1 Einleitung

1.1 Definitionen

1.1.1 Formatierung dieses Dokumentes

Exemplarischer Text mit einem *"Eingebetteten Zitat"* und dann weiter mit etwas höchst **Wichtigem**.

> *"Dies ist ein Blockzitat*
> *Von einer im Text genannten Person"*

Exemplarisch eine Codedatei mit dem Namen *myfile.cpp:*

```
// Dies hier ist Code
int a;
class B {…};
```

In Beispielcode werden Keywords **fett** hervorgehoben und eingebaute Typen *kursiv*. Hier wird auf die Variablen *a* und *b* Bezug genommen und die Typen *int* und *float* erwähnt, stets im Singular. Wenn Code im Fließtext geschrieben wird, dann sieht es so aus `int a[] = {1,2,3};`.

1.1.2 Begriffsdefinitionen

In jedem Dokument, welches über mehrere Programmiersprachen schreibt, müssen zunächst Mehrdeutigkeiten und Kollisionen ausgeräumt werden. Besonders erwähnenswert sind Unterschiede zur Namensgebung in C++.

In dieser Arbeit werden oft die englischen Varianten von Begriffen aus der Informatik / Programmierung genutzt, ein Beispiel zur Verdeutlichung: Ein **Feld** ist etwas, wo der Bauer sein Korn anbaut, wo ein Mannschaftsspiel stattfindet oder eine Datenstruktur, welche mehrere gleichartige Daten beherbergt. Im Kontext der Objektorientierung, in Datenbanken und Formularen werden jedoch auch skalare Daten / Variablen Feld (engl. Field) benannt. Ein Ausdruck mit wesentlich mehr Trennschärfe, welcher das Drittgenannte bezeichnet, ist **Array**. Das Wort **Ausdruck**, aus dem letzten Satz hat eine Bedeutung. Ausdruck ist auch das was aus einem Drucker kommt, eine **Expression** hingegen, heißt zwar *Ausdruck* in Englisch, macht aber klarer, dass es sich um einen Ausdruck im Kontext einer Programmiersprache handelt. Aus Gründen der Trennschärfe und weil sie im Alltag auch so verwendet werden, wird in diesem Dokument eben die englische Variante genutzt, aus Gründen der Lesbarkeit und Homogenität mit deutscher Großschreibung.

Namen von Keywords und Sprach-Konstrukten werden nicht pluralisiert.

C++
: Wenn über C++ geschrieben wird, dann ist oft beides gemeint: die *Sprache C++* an sich und die *C++ Standard Library*. Diese sind, insbesondere seit C++11, sehr eng miteinander verbunden; so wurde die Sprache erweitert, um bestimmte Funktionalität in der Bibliothek zu ermöglichen. Ebenso sind Komponenten der Bibliothek, notwendig um die Sprache vollständig zu nutzen.
Beispiele: `std::shared_ptr`, `std::initializer_list`, `std::move`, `std::dynamic_cast`

OOP
: Objektorientierte Programmierung.

Konstrukt

Alles innerhalb einer Programmiersprache: Deklarationen, Definitionen, Statements, Expressions, Attribute, Typen, Variablen. Erklärung dieser Begriffe folgend.

Keyword

Schlüsselwort. Wort mit spezieller Bedeutung innerhalb der Sprache, welches nicht für eigene Namen verwendet werden kann.

Identifier

Bezeichner. Ist der Name, welche etwa eine Variable oder ein eigener Typ haben kann. Ist üblicherweise eine Mischung aus Buchstaben, Ziffern und Unterstrichen.

Funktion & Prozedur

Manche Programmiersprachen unterscheiden hart zwischen diesen zwei Unterprogramm-Typen. (Als *Unterprogramm* versteht sich hier nicht die Definition welche COBOL verwendet)
Zum Beispiel liefern in Pascal *Prozeduren* keine Rückgabewerte. Außerdem wird in vielen Sprachen in *Funktionen* das Resultat nur aus den Eingaben erzeugt und hat keine Nebeneffekte, dieser Spezialfall wird hier *"pure Funktion"* genannt. In C++ werden all diese Unterprogramme *Funktionen* genannt, da es dort aber kein Keyword dafür gibt, konnten Benutzer im Sprachgebrauch natürlich ihre eigene Namensgebung verwenden.
In diesem Dokument wird sich der Gepflogenheit von C++ bedient: Alles sind Funktionen und können, soweit nicht anders annotiert, Nebeneffekte haben und auch keine Rückgabe haben.

Methode

Ist eine Funktion, die mit einem Objekt verknüpft ist.

FhO – Funktion höherer Ordnung

Beschrieben im Kontext einer Prozeduralen Programmiersprache:
Eine FhO ist eine Funktion, welche eine weitere Funktion (oder auch Lambda) und eine Kollektion als Parameter übergeben bekommt. Die übergebene Funktion wird auf jedes Element der Kollektion angewendet. Diese Kollektion kann auch durch einen Iterator repräsentiert werden.

Map & Reduce

Map ist eine FhO, bei der pro Ursprungselement ein Zielelement erzeugt wird, welche wiederum in einer Kollektion gespeichert werden. Sie wird oft zusammen mit der FhO namens *Reduce* genutzt, welche aus vielen Ursprungselementen ein einzelnes Zielelement erzeugt.

Template

Schablone. Art der generischen Programmierung, Programmierung mit offen gehaltenem Typ. Wortkombinationen wie z.B. *Klassen-Template* der C++ Nomenklatur, sollen signalisieren, dass gar keine Klasse existiert, bevor das Template instanziiert wird. Auch wenn dieses Verhalten von C++ geerbt wird, werden hier diese Kombinationen in umgekehrter Reihenfolge geschrieben, also z.B. *Template-Klasse.*

TMP – Template Metaprogramming

Alternative Art der Programmierung bei der Logik zur Kompilationszeit ausgewertet wird.

UB – Undefined Behavior

Undefiniertes Verhalten. Tritt auf, wenn der Programmierer die Regeln der Sprache verletzt. Nach dem Auftritt dieser Verletzung ist es dem Programm erlaubt jedwedes Verhalten zu zeigen. Bei dieser Art der Fehlfunktion muss ein C++-Compiler weder die Kompilation abbrechen, noch eine Warnung ausgeben.

GC – Garbage Collection
: Automatische Speicherverwaltung. Meist durch vollständige Betrachtung des dynamisch allozierten Speichers umgesetzt.

Deklaration & Definition
: Deklaration ist die Bekanntmachung eines Konstrukts, dessen Signatur.
Definition ist die Implementation eines Konstrukts, dessen Innenleben.

RTTI
: Run-Time Type Information, ist die Bereitstellung von Typinformationen zur Laufzeit

Syntactic Sugar
: Einfachere Schreibweise, für die es schon eine Kompliziertere gibt. Beispielsweise ist `int_ptr[5]` das gleiche wie `*(int_ptr+5)` und `shape_ptr->draw()` das gleiche wie `(*shape_ptr).draw()`.

Debug / Release (Build)
: Entwicklungsmodus einer Anwendung, Debug ist meist nicht hoch-performant, dafür mit mehr Fehlerbehandlung, Release ist meist mit allen verfügbaren Optimierungsmöglichkeiten erzeugt.

Pointer
: Zeiger. Adresse des Datums / der Daten.

Expression
: Ausdruck. In diesem Dokument im Englischen verwendet, weil es im Programmiersprach-Kontext viel eindeutiger ist als das Deutsche Wort. Beispiel: `4 + i * obj.factor`.

Statement
: Anweisung / Aussage. In diesem Dokument im Englischen verwendet, weil es im Programmiersprach-Kontext viel eindeutiger ist als das deutsche Wort. Beispiele: `if( a == b ) return true;`.

Deklaration
: Auch: Deklarierung. Bekanntmachung eines Konstrukts (Variable, Konstante, Typ, ...).
Beispiel: `namespace MyLang { class Test { … }; }`.

Exceptions
: Ausnahmen. Ein Mittel zur Fehlerpropagation innerhalb von Programmen, sehr oft in Objektorientierten Sprachen eingesetzt.

Member (Variablen / Funktionen)
: Mitglied (einer Klasse / Struktur). Auch Objektgebundene Variablen / Funktionen (Methoden).

(Operator) Precedence
: Vorrangs Reihenfolge (von Operatoren). Beispiel: Punktrechnung vor Strichrechnung.

RAII – Resource Acquisition Is Initialization
: Ressourcenbelegung ist Initialisierung. Art der deterministischen Speicherverwaltung in C++.

Polyfill
: Füllfunktion. Welche ein Defizit in der Kern-Sprache oder der Standardbibliothek ausgleicht.
Gängige Praxis bei JavaScript Frameworks, da jeder Browser andere Defizite hat.

Cache-Line
: Kleinste Verwaltungseinheit des Cache, siehe auch [CACHE].

1.2 Idee / Motivation

Die Inspiration zur Entwicklung dieser Sprache kam mir ursprünglich durch ein schon etwas älteres Dokument namens *'A Modest Proposal: C++ Resyntaxed'* von Werther und Conway. In ihrem Dokument bemängeln sie zurecht einige Unschönheiten in der Grammatik von C++, welche zum Teil der Kompatibilität zu C geschuldet ist. Ihre syntaktische Reimaginierung von C++ nennen sie **SPECS**, siehe [SPECS].

Auch der Erfinder von C++, Bjarne Stroustrup, glaubt

> *"within C++ there is a much smaller and cleaner language struggling to get out"*
>
> Sinngemäß ins Deutsche: *"in C++ schlummert eine wesentlich kompaktere und sauberere Sprache, welche Mühe hat ans Tageslicht zu kommen"*

was er in *'The Design and Evolution of C++'* schrieb, welches auch SPECS zitierte.

Ich teile diese Meinung. Wenn ich etwas in C++ umsetze, dann ist das Konzept in meinem Kopf meist wesentlich kompakter, als das, was ich später umsetze.

Zwar hat modernes C++ (11/14/17) frischen Wind in die Sprache gebracht, aber trotzdem fast keine Zöpfe abgeschnitten. Auch wurden neue Sprachkonstrukte eingeführt, welche teils syntaktisch unschön und schreibintensiv sind, dem geschuldet, dass alter Code durch diese Ergänzungen nicht aufhören darf zu funktionieren.

Retrospektiv betrachtet hat die in SPECS vorgeschlagene Syntax nicht mehr viel mit C++ zu tun und dessen stark von Pascal angehauchte Syntax liegt fern von meinem Ästhetikempfinden. Mein Lebensweg als Programmierer ist von C und C++ ähnlichen Sprachen geprägt, weshalb die hier vorgestellte Lösung weit von der von SPECS abweicht.

Die zwei Hauptaufgaben von Myll sind folgend beschrieben.

1.2.1 Alteingesessene C++-Programmierer abholen

Alteingesessene C++-Programmierer abholen und dessen Arbeit in Zukunft leichter zu gestalten.

Dies geschieht durch leicht zu merkende Sprachkonstrukte, wenig Wiederholung und Boilerplate (Textbausteine), enge Anlehnung an modernes C++ und Zukunfts-Offenheit für neue Funktionalität. Die Offenheit für zukünftige Sprachfeatures kann zunächst mittels Polyfill erfolgen. Sollte C++ jedoch später eine ähnliche Funktionalität anbieten, wird es dann einfach durchgereicht.

Auch der Ausstieg aus der Nutzung von Myll oder gar der einmalige Einsatz dessen ist möglich. Der erzeugte C++-Code soll leserlich und direkt weiter nutzbar sein. Kein geschriebener Code ist je vergebens.

1.2.2 Neulingen den Einstieg erleichtern

Neulingen den Einstieg in eine immer komplizierter werdende Sprache zu erleichtern.

Weniger Abweichung von anderen modernen Sprachen. Vereinfachung der Positionen, an denen Keywords und Attribute auftauchen. Weniger überraschende und unbemerkte Nebeneffekte. Mehr selbstauferlegte Regeln, mit denen man die Korrektheit seiner Implementierung überprüfen kann.

Durch die Erzeugung von lesbarem C++-Code kann dieser Transpiler auch beim Lernen von C++ helfen, da man sich den erzeugten C++-Code immer anschauen kann.

1.3 Ziel

Diese zwei Hauptaufgaben stehen sich natürlich auch, teils extrem, entgegen. Es wird versucht einen gangbaren Mittelweg zu finden und entschieden wird, im Zweifel, meist für die zukünftige Wartbarkeit und gegen die Vergangenheit.

Ein Ziel soll auch die Betrachtung dessen sein, die Umsetzbarkeit einer Programmiersprache mit ähnlicher Komplexität wie C++ zu handhaben. Welche Aspekte von C++ sind problematisch und lösbar? Sind ähnlich positive Aspekte zu erzielen wie es Transpilation bei Webentwicklungen hervorgebracht hat? Bemerkenswert ist hieran, dass TypeScript den identischen Ansatz wie C++ gewählt hat.

> C++ erlaubte jeglichen C-Code plus seiner eigenen Erweiterungen.
>
> TypeScript erlaubt jeglichen JavaScript-Code plus seiner eigenen Erweiterungen.

Somit ist man sofort *TypeScript-Entwickler*, sobald man seine Dateien umbenannt hat, siehe [IW-TSJS].

Myll soll auch Unterstützung bei der Einhaltung von Code-Richtlinien bieten. Es gibt viele Richtlinien, an die sich ein Team halten kann, aber diese Regeln nicht unabsichtlich zu verletzen ist nicht so einfach. Hier kann Myll dem Nutzer unter die Arme greifen und diese Regeln automatisch überprüfen. Ein paar dieser Regeln sind vordefiniert, es soll jedoch in Zukunft auch möglich sein, eigene Regeln zu definieren.

Das Ziel dieser Arbeit ist es ein stimmiges Gesamtkonzept für das Design der Sprache aufzustellen und den Prototypen des Transpilers umzusetzen, welcher selektierte, nicht-triviale Programme übersetzen kann.

1.4 Grundlagen

1.4.1 Was macht C++ besonders?

Es gibt sehr viele neue und moderne Programmiersprachen, die vielfältige Einsatzgebiete haben, einfacher und sicherer zu benutzen, und in Teilbereichen performanter als C++, sind. Die Popularität von C++ war mal auf einem absteigenden Trend, dieser setzt sich aktuell aber nicht mehr fort, siehe [POP].

Einsatzgebiete, in denen häufig C++ für die Umsetzung verwendet wird:

- Betriebssysteme
- Treiber
- Embedded Computing
- Computerspiele
- Simulationen
- Statistik

In diesen Bereichen wird C++ entweder aufgrund seiner guten Performance oder wegen der geringen Größe der Laufzeitumgebung und des Kompilats eingesetzt. C++ war früher auch als Entwicklungssprache für Desktopanwendungen verbreitet, wurde in diesen Bereichen aber in weiten Teilen von Java, C# aber auch von Webanwendungen abgelöst.

Wieso hält sich C++ in den genannten Bereichen so hartnäckig?

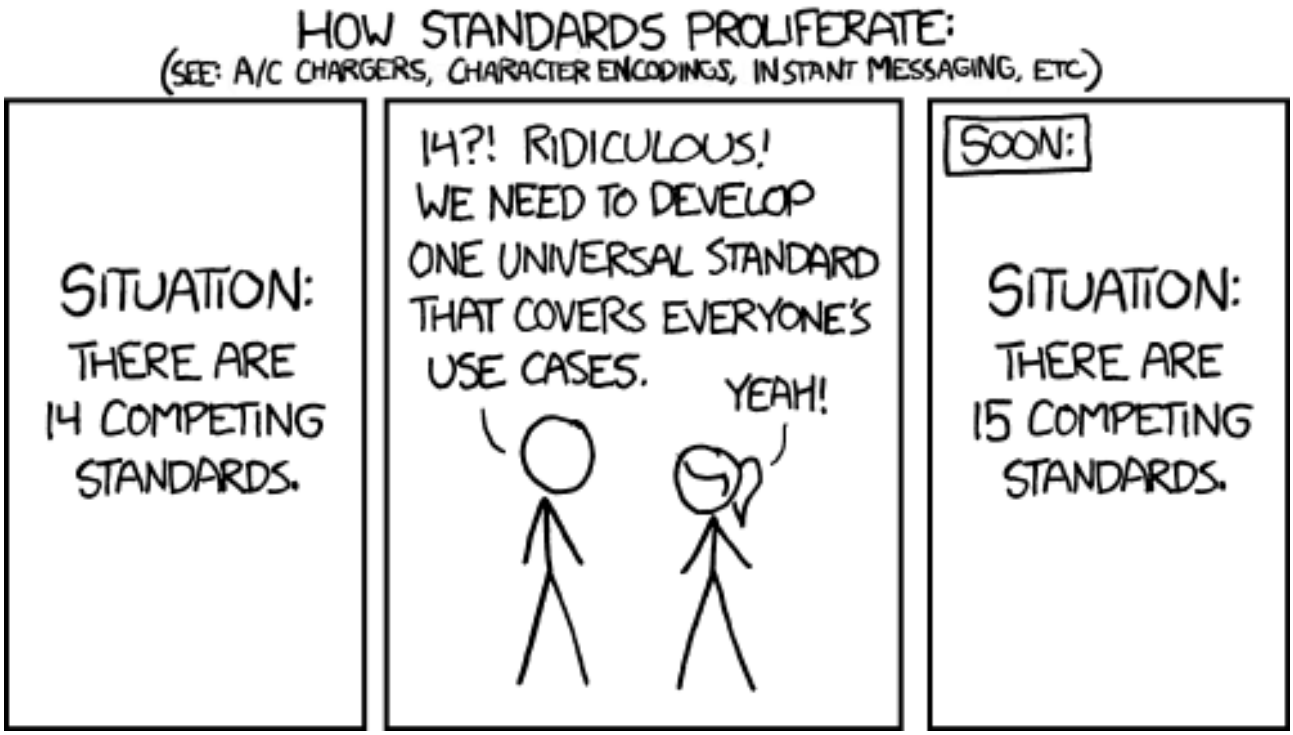
- Maximale Kontrolle über den Speicher
- Keine erzwungene Garbage Collection (GC)
- Keine / schlanke Laufzeitumgebung
- Native Binärdateien
- Nichts bezahlen, was nicht genutzt wird
  (abschaltbar: RTTI, Exceptions, Bounds/Overflow checks, etc)
- Keine erzwungene virtuelle Vererbung
- Mehrere Jahrzehnte alter C- und C++-Code läuft fast immer ohne Anpassung

Mehr hierzu in 2.1.

1.4.2 Wieso noch eine neue Programmiersprache?

Warum eine neue Programmiersprache, wenn C++ doch seine Bereiche so erfolgreich bedient?
Weil es sich mit vielen der anderen, hier betrachteten Sprachen (siehe 2.1) einfacher entwickeln lässt als mit C++.

Trotzdem wären viele Sprachkonstrukte, sobald man sie auf Sprachebene und nicht nur auf Bibliotheksebene angehen kann, eigentlich leicht übertragbar auf C++. Das beste Beispiel hierzu findet sich in 3.3.13.



Quelle: https://xkcd.com/927/

2 Analyse

2.1 Die Besonderheiten von C++

Im vergangenen Kapitel wurde die besondere Stellung von C++ schon erwähnt, hier soll darauf eingegangen werden, was dies im Detail bedeutet.

In C++ hat man maximale Kontrolle über den Speicher und kann auch selektiv, in transparenter Form, diese Kontrolle aufgeben, ohne auf Performance zu verzichten. Man kann einen Garbage Collector (GC) schreiben und diesen nutzen. Man kann sich aber auch dagegen entscheiden und seinen Speicher manuell handhaben oder *Reference Counting* (Mitzählen wie oft ein Pointer verwendet wird) verwenden, welches etwa vom `std::shared_ptr` bereitgestellt wird.

All diese Allokationen kann man entweder vom System-Allokator durchführen lassen oder man kann einen spezifischen Allokator nutzen, zum Beispiel einen Arena-Allokator. Ein Arena-Allokator ist typischerweise für ein Subsystem zuständig. Dieser alloziert beispielsweise einen größeren Speicherblock vom System-Allokator, verwaltet und vergibt dann Stücke von diesem, teils gruppiert nach Größe (Binning), selbst. Dadurch sind beispielsweise gleich große Objekte in einem zusammenhängenden Speicherbereich und im Allgemeinen alle Objekte von ähnlichen Typen nah beieinander. Systemcalls werden dadurch seltener und interne Fragmentierung bekommt eine Obergrenze in Höhe der Bin-Größenabstände.

Frame-Allokatoren können in Programmen genutzt werden, welche in deterministischen Zeitabständen einen *Frame (hier Schritt, Zeitschritt pro Bildaufbau)* abgeschlossen haben. Dies können zum Beispiel Simulationen, graphische Anwendungen oder Computerspiele sein. Ein Frame-Allokator funktioniert so, dass ein relativ großer Speicherbereich einmalig, entweder statisch oder auf dem Heap, alloziert wird. Fordert jemand Speicher von diesem Allokator an, wird dieser einfach vom Beginn des noch verfügbaren Bereiches abgegeben. Ein explizites Freigeben des Speichers gibt es nicht, denn dessen gesamter Speicher wird am Ende des Frames freigegeben, indem dessen Zeiger wieder auf den Anfang des Speichers gesetzt wird. Frame-Allokierter-Speicher hat natürlich eine wichtige Einschränkung; er darf nicht über Frame-Grenzen gehalten werden. Dieser Allokator hat quasi gar keinen Verwaltungs-Overhead.

In einer anderen Sprache, welche auf das Vorhandensein einer *GC* aufbaut, ist all dies nicht mehr, oder nur partiell und mit hohem Aufwand, möglich.

Strukturen besitzen in C++ ein fest vorgegebenes Speicherlayout, welches einem ermöglicht genau zu wissen, ob sie etwa in eine *Cache-Line* passen. Dadurch lässt sich sehr deterministisch die Laufzeit von darauf agierenden Algorithmen abschätzen.

Die aus C++-Code kompilierten Programme sind native Binärdateien, welche keine bzw. schlanke Laufzeitumgebungen mit sich bringen. Dies macht sie sehr schnell im Startup.

In C++ muss nichts bezahlt werden, was nicht genutzt wird, dies beinhaltet sogar Kernfunktionen wie RTTI, Exceptions, Bounds-/Overflow checks, etc. Auch kann man sich aussuchen, ob man virtuelle Methoden und Vererbung verwenden will oder ob man ohne auskommt. Besonders bei kleinen Strukturen kann der Overhead enorm sein, siehe Kasten.

```
struct C { char a; }; // sizeof C: 1
struct V { char a;    // sizeof V: 16
           virtual ~V(){}};
```

Mehrere Jahrzehnte alter C++-Code läuft fast immer und selbst die meisten aktuellen, wie auch ältere C-Programme, laufen ohne Anpassung.

Die Infrastruktur in Form der verfügbaren Compiler greift auf die Reife von jahrzehntelanger Optimierung zurück. Und dies ist nicht nur der Fall bei kommerziellen Produkten, denn es gibt mehrere kostenlose Compiler, viele davon sogar Open-Source: G++ der GNU Compiler Collection, Clang++ das C++ Frontend von LLVM und auch, durch Lizenzbedingungen eingeschränkt, der Microsoft Visual C++ Compiler.

Zur Betrachtung der, in dieser Arbeit aufgeführten Beispiele, nutzte der Autor die Compiler-Parameter: `"-Wall -Wextra -Wpedantic"` mit GCC in Version 10.1 und Clang in Version 10.

2.2 Vergleich von C++ mit anderen Sprachen

Als Kompilationsziel von Myll soll zwar C++ dienen, jedoch mag sich Myll, zum einen syntaktisch und zum anderen von den Features anderer Programmiersprachen, inspirieren lassen. Aus diesem Grund sei hier C++ mit einigen ausgewählten Programmiersprachen verglichen.

2.2.1 Vergleich mit C

Der Vorgänger von C++.

+ Viel Kontrolle, man sieht genau was passiert
+ Viel simpler als C++
+ Sehr viele Freiheiten, aber auch Verantwortung bei der Speicherverwaltung
- Schlechte Datenkapselung / Keine native Unterstützung für OOP
- Unübersichtlich bei großen Projekten, denn ohne Namespaces landet alles im globalen Raum
- Es ist der Ursprung einiger in dieser Arbeit betrachteten syntaktischen Schwächen von C++
- Mehr LoC (Lines of Code, Codezeilen) für die Lösung des gleichen Problems

Hier ist ein Beispiel welches aufzeigt, was ich mit *viel Kontrolle* und *viel simpler* meine. Es ist Code, welcher so in C, als auch in C++ auftreten kann. Die Kommentare beschreiben, was in C++, zusätzlich zum Offensichtlichen, passieren kann.

```
{
    Foo f;                  // kann in C++ einen potentiell teuren Konstruktor von Foo aufrufen
    A * a = f.get();        // kann in C++ eine Exception werfen und den folgenden Code überspringen
    a->update();            // kann in C++ eine virtuelle Methode aufrufen
}                           // kann in C++ einen potentiell teuren Destruktor von Foo aufrufen
```

2.2.2 Vergleich mit Java und C#

Java und C# sind als designierte Nachfolger von C und C++ entstanden, siehe [JAVAHIST].
Sie haben sehr viel untereinander gemein und werden deshalb hier gemeinsam betrachtet.

+ Es ist leichter, fehlerfreien C#- und Java-Code zu schreiben
  (Keine Pointer-Arithmetik, kein UB, kein *use after free*, weniger Speicherlecks)
+ Fehlermeldungen in Generics/Templates beschreiben viel präziser den Fehlerfall
+ Einfach auf vielen Architekturen ausführbar (oft ohne Neukompilierung)
+ Weniger LoC für die Lösung des gleichen Problems (keine Header, umfangreiche Standardbib.)
+ Viele äquivalente Features viel früher verfügbar als in C++ (concepts, auto, ranges, override, ...)
- Erzwungene GC, weniger Kontrolle über den Speicher
- Keine multiple Vererbung

- Nicht abschaltbare Bounds/Overflow checks, Reflection, Exceptions
- Ein leichtes Performancedefizit, welches bei Desktopanwendungen verschmerzbar ist [JAVAPERF]

2.2.3 Vergleich mit D

D hatte sich vorgenommen, die Reihe der Sprachen BCPL → B → C → C++ → D fortzusetzen. [DLANG]

Leider konnte D nicht annähernd den Erfolg von C oder C++ erreichen, obwohl vieles von C++ gelernt und womöglich besser gemacht wurde (TIOBE Index: D beste Platzierung #12 in 2009, aktuell Platz #23, vgl. mit C++ beste Platzierung #3 in 2019, aktuell #4, [TIOBE]).

+ Sehr nah an C++ angelehnt
+ Weniger LoC für die Lösung des gleichen Problems (keine Header, mehr Beispiele siehe [DLANG])
- GC für die Nutzung der Standardbibliothek nötig, weniger Kontrolle über den Speicher
- Keine multiple Vererbung

2.2.4 Vergleich mit Rust

Eine sehr neue Programmiersprache, welche ihren Fokus auf die Sicherheit dessen, was maschinell überprüfbar ist, legt. Die statischen Überprüfungen, welche der Rust Compiler vornimmt, sind in der Ausprägung noch nicht verbreitet gewesen, siehe [TR-R5Y].

+ Stärkere statische Checks verhindern einige Fehlerklassen
+ In Zukunft ist Rust ein Wettbewerber um einige Bereiche, welche C++ heute bedient
- (Anfänglich) schwerer zu schreiben durch Paradigmenwechsel des Object-Lifetime-Managements
- Exorbitante Kompilationszeiten wegen nicht deaktivierbarer statischer Überprüfung
  (für Release-Builds vollkommen Okay, für Debug-Builds nicht)

2.2.5 Vergleich mit Ruby

Dieser Vergleich ist ein wenig weiter hergeholt als die anderen betrachteten Sprachen. Ich wollte hier dennoch gern über den nahen Tellerrand hinausblicken, um mich inspirieren zu lassen.

+ Unglaublich wenig Code ist nötig um simple Probleme zu lösen, siehe Beispiel in 3.3.8.2
+ Lambdas sind schon sehr lange ein gut integrierter und fast unsichtbarer Teil der Sprache
- Interpretierte Sprache (langsam in der Ausführung)
- Erzwungene GC, weniger Kontrolle über den Speicher
- Keine Typsicherheit

2.2.6 Vergleich von JavaScript mit TypeScript

Dieser Vergleich findet nicht mit C++ statt, sondern mit der Sprache, auf der TypeScript basiert. Er dient als das Beispiel eines Dreisatzes:

> *Was TypeScript für JavaScript ist, soll Myll für C++ sein!*

Das ist *noch* kein Zitat, weil ich es selbst gesagt habe. Ich finde es nur sehr einfach auf den Punkt gebracht, was das Ziel dieser Sprache sein soll.

JavaScript leidet unter dem gleichen Problem unter dem auch C++ leidet, nämlich das alter Code weiterhin so funktionieren muss wie bisher. Demnach können sich beide Sprachen nur in den Dimensionen aus-

dehnen, welche zuvor kein kompilierendes/lauffähiges Programm ergeben hätte. Dadurch sind sie verdammt, ihre schlechten Entscheidungen für immer mit sich herumzutragen.

Hier zunächst der gut lesbare **TypeScript**-Code welchen wir folgend in JavaScript übersetzen.

```
class Human {
    _name: string;
    constructor(name: string) {
        _name = name;
    }
}

class Student extends Human {
    _mtknr: number;
    constructor(name:string, mtknr:number) {
        super(name);
        _mtknr = mtknr;
    }
    getGrade() : string {
        return "A+";
    }
}
```

Daraus macht der TypeScript Transpiler diesen ES5 **JavaScript**-Code:

```
var __extends = /* gekürzt: 12 Zeilen lange Polyfill Funktion */

var Human = /** @class */ (function () {
    function Human(name) {
        this.name = name;
    }
    return Human;
}());

var Student = /** @class */ (function (_super) {
    __extends(Student, _super);
    function Student(name, matrikelnummer) {
        var _this = _super.call(this, name) || this;
        _this.matrikelnummer = matrikelnummer;
        return _this;
    }
    Student.prototype.getGrade = function () {
        return "A+";
    };
    return Student;
}(Human));
```

Man müsste ähnlichen Code schreiben, wenn man will, dass er auf einem ES5 tauglichen Browser läuft.

TypeScript bietet viele Vorteile gegenüber purem JavaScript.

- Es ist leichter les- und schreibbar
- Es bietet optional Typsicherheit
- Es ermöglicht, Funktionalität von moderneren JavaScript Versionen, in älteren Versionen zu nutzen
- Es erlaubt, direkt bei der Übersetzung, eine Aggregation, von mehreren Dateien, in eine einzelne

2.3 Problematische Aspekte von C++

Einige der problematischen Konstrukte in C++ sind der direkten Kompatibilität mit C geschuldet, andere sind von C++ so eingeführt worden. Die Betrachtung dieser Probleme findet sich, der besseren Übersicht halber, direkt gepaart mit deren Lösung in Kapitel 3.3.

3 Konzept

3.1 Einleitung

Um bei der Konzeption der Sprache das Ziel nicht aus den Augen zu verlieren, wurden einige simple Grundsätze definiert, welche auch auf Englisch verfügbar sind. Die Einhaltung der anfangs aufgestellten Grundsätze soll später bewertet werden.

Der Autor vertritt die Auffassung, dass viel Inspiration von anderen Sprachen eine gute Sache ist und mehr Wiedererkennungswert den Einstieg einfacher macht.

3.1.1 Grundsätze der Sprache Myll

1. Erwarte keine Wiederholung vom Nutzer
2. Außergewöhnliches Verhalten muss explizit sein
3. Das, was man naiv erwartet und keine Nebeneffekte hat, kann implizit sein
4. Breche nicht mit der allgemeinen Semantik von C++
5. Breche mit C sofern es einen Nutzen bringt
6. Entwickle die Syntax so weiter, dass sie eindeutig bleibt
7. Sei auch dann nützlich, wenn die Entwicklung am einmalig erzeugten C++-Code weitergeht
8. Spare nicht an neuen Keywords, wenn die Lesbarkeit profitieren kann

3.1.2 Fundamental Principles of Myll

1. Don't ask the user to repeat themselves
2. Exceptional behavior needs to be explicit
3. What you naively expect to happen and has no side-effects can be implicit
4. Don't break with the general semantic of C++
5. Do break with C if there is a benefit
6. Evolve the syntax that it contains no ambiguity
7. Be useful even if a one-time translation to C++ is all that's wanted
8. Don't be frugal with new keywords, if it benefits readability

3.1.3 Inspiration

Ich habe mich schon eine lange Zeit vor dem Beginn dieser Arbeit, mit dem Design und der Erstellung von Programmiersprachen beschäftigt. Das Video »**Ideas about a new programming language for games**« von **Jonathan Blow**, aus dem Jahre 2014, brachte mich dazu mein Vorhaben, eine eigene Programmiersprache zu erstellen, in die Tat umzusetzen, siehe [JB-JAI]. Dieses und viele seiner weiteren Videos diente auch als Inspirationsquelle vieler Aspekte der hier vorgestellten Sprache. Jonathan Blow ist als Game-Designer und Chef-Entwickler der Spiele *Braid* und *The Witness* bekannt. Er entwickelt aktuell parallel die Programmiersprache **Jai**, zusammen mit einem weiteren Spiele-Projekt, einem komplexen Sokoban Puzzlespiels, welches in eben dieser Programmiersprache implementiert ist.

Auch ein zweites Projekt begleitete den Pfad, der zur Erstellung dieser Arbeit führte, über eine längere Zeit: »**Bitwise**« von **Per Vognsen**. In Bitwise ging es darum einen kompletten Computer, also den gesamten Hard- und Software-Stack, als Open-Source selbst zu erzeugen. Von besonderem Interesse war für mich die Erzeugung des **Ion** getauften Compilers dieses Projektes, siehe [PV-ION]. Herausstechend daran war, dass die Entwicklung live übertragen wurde und der Code dieser Entwicklung jederzeit einsehbar war. Zu sehen, dass ein Compiler innerhalb kurzer Zeit, von seiner Konzeption, zur Funktion gebracht werden konnte, war höchst ermutigend. Auch beim Sprachdesign wurde ich Inspiriert, eine einfach lesbare Syntax, sowohl für Parser, als auch für Menschen zu erzeugen. Per Vognsen war lange Zeit als Spieleentwickler und Systemprogrammierer tätig, welcher aktuell seine Zeit in informative und lehrreiche Projekte investiert.

Konträr zu diesen beiden Projekten liegt mein Fokus nicht auf extrem schneller Verarbeitung / Kompilation, sondern auf schneller Einarbeitung, weniger Flüchtigkeitsfehler und weniger Wiederholung.

Ein Artikel ist mir positiv und inspirierend aufgefallen »**Ideas for a Programming Language Part 3: No Shadow Worlds**« von **Malte Skarupke**, welcher mein Bewusstsein auf die unterschiedlichen Programmebenen hinwies, welche alle gleichzeitig in einer einzelnen Programmiersprache vorhanden sind, siehe [MS-NSW]. Da ist das erwartete Prozedurale, als auch das Objektorientierte in C++. Dahinter verborgen, quasi im Schatten dieser Offensichtlichen, liegen noch Generisch via Templates (und dessen Spezialfall TMP), Funktional und vom Präprozessor vorverarbeitet. Das Schlimme daran ist das sich teils mehrere dieser schattenhaften Ebenen überlagern und dadurch das Verständnis dessen, was an einem Codeblock passiert, verkomplizieren. Auch aufgrund dieses Artikels habe ich mich strikt gegen die Aufnahme eines Präprozessors entschieden und versuche TMP so gut es geht durch einfachere Lösungen zu ersetzen.

Außerdem stieß ich bei der Recherche dieser Arbeit auf ein Proposal (offizielles Vorschlagsdokument), welches auch viele problematische Aspekte von C++ auflistet und für deren Behandlung den offiziellen Weg durch den Standardisierungsprozess geht. Es handelt sich dabei um »**Epochs: a backward-compatible language evolution mechanism**« von **Vittorio Romeo**, siehe [VR-EPO]. Das Proposal zeigt, wie die Probleme von C++ durch einen "Epochen-Schalter" gelöst werden können. Dieser Schalter könnte selektiv, pro Datei / Translation-Unit, sogar "Zöpfe abschneiden". Sollte dieses oder ein ähnliches Proposal die Aufnahme in C++ finden, werden bestimmt nicht alle der hier betrachteten Probleme gelöst und der frühestmögliche Zeitpunkt wäre 2023 oder gar 2026. Leider sehe ich auch einige der Richtlinien unter 8.12 als absolut nicht *unumstritten* an und ich kenne einen ganzen Schlag Leute, die dies ähnlich sehen (Spieleentwickler). Dennoch denke ich, dass es viel gutes Bewirken kann und hoffe auf dessen Aufnahme.

3.1.4 Lerneffekte der verglichenen Sprachen

Welche Lerneffekte kann Myll aus den mit C++ verglichenen Programmiersprachen ziehen?

Von **C** kann man Simplizität lernen: einfache Syntax für Casts, an Raw-Pointer angelehnte Smartpointer Syntax. Syntaktisch schwer durchschaubare und problematische Konstrukte hingegen sollen vermieden werden. Von **C#** und **Java** kann man sich einen Feature-Vorsprung zu C++ und eine entschlackte Syntax abschauen. Das Ziel von **D** war ein recht Ähnliches wie meines, nämlich ein besseres C++ zu sein, Myll wird keine GC verwenden und auch nicht (absichtlich) auf Kernfeatures von C++ verzichten. Von **Rust** direkt schneidet sich Myll keine Scheibe ab, einen Paradigmenwechsel in der Schreibweise von C++ soll es explizit nicht geben, es wird lediglich versucht den halben Weg in Richtung sicherer Software, durch die Behandlung einfach zu lösender Probleme, wie der Korrektur schlechtem Standardverhaltens und der Vermeidung von Flüchtigkeitsfehlern, zu gehen. Von **Ruby** soll sich ein "Das könnte doch viel einfacher geschrieben werden!" abgeschaut werden, welches C# sich schon über die Jahre von Ruby oder ähnlichen Sprachen abgeschaut hat. Von **TypeScript** kommt die ursprüngliche Umsetzungsidee als Transpiler.

3.2 Design der Sprache Myll

Im nächsten Abschnitt werden problematische Aspekte von C++ analysiert und behandelt, aber eine schlichte Behandlung einzelner Probleme macht noch keine Sprache aus, sie bietet lediglich ein besseres Fundament. Eine Programmiersprache sollte auch ein einheitliches Gesamtbild, wenig widersprüchliche Konstrukte und eine gute Lesbarkeit aufweisen.

Zur Entscheidungsfindung, welche Syntax und welche Funktionalität von C++ nach Myll übernommen und welche nicht, informiert sich der Autor stets, was Spezialisten, im Umfeld von C++, für Meinungen, Ansichten und Best-Practices vertreten. Vielen Design-Entscheidungen liegen keine harten Fakten zu Grunde, sondern sind eher ein Stimmungsgefühl, ein Abwägen, in das natürlich die Erfahrung des Autors, als **Meinung**, mit einfließt.

3.2.1 Neue Keywords

Myll nutzt einige neue Keywords, denn Syntax ohne Keywords hat sehr schnell alle *schönen* Konstellationen verbraucht und Monster, wie diese hier, entstehen:

```
[](){}();                           // valides C++11, leeres Lambda welches direkt aufgerufen wird
void(*ffp)(int(*)(int,int));        // valide seit dem Anbeginn von C++
```

Die Liste der neuen Keywords ist:

```
func, proc, method, ctor, dtor      // alle Arten von Funktionen
var, field                          // alle Arten von Variablen
get, set, refget                    // zur Erstellung von Accessoren
import, module                      // Bekanntmachung und Inkludieren von Modulen
loop, times                         // neue Schleifen
requires, alias, …                  // etc
```

Es wurden Abkürzungen mittlerer Länge gewählt, welches sich an die ursprünglichen Keywords aus C++ anlehnt: struct(ure), enum(eration), float(ing point number), double (precision float), int(eger). Für Funktionen wäre *fn* zu wenig Information und kollidiert zu leicht.

Außerdem konnten viele Keywords aus C++ im allgemeinen Programm freigegeben werden, diese haben in Myll nur zum Teil noch als Auszeichnung innerhalb von Attributen eine Bedeutung:

```
static, virtual, override, final, include, export, …
```

Da Myll jedoch im aktuellen Stand nach C++ übersetzt, sollten diese natürlich noch nicht für Identifier verwendet werden.

3.2.2 Übersicht der Syntax von Myll

Jede Deklaration beginnt mit einem Keyword, welches es direkt erkennbar macht, was die Zeile enthält. Was darauf folgt ist meist ein Name oder auch ein Typ, gefolgt von weiteren Informationen, welche entweder mit Semikolon oder einem Block (geschweifte Klammern und deren Inhalt) beendet werden. Dies gilt hier auch für *import* und *module*.

Die Syntax der Deklarationen von Variablen und Funktionen sei vorab kurz erklärt, um ein Grundverständnis aufzubauen. Variablen werden mit dem neuen Keyword *var, const* oder *field* eingeleitet und verhalten sich danach weitestgehend wie C++, mehr dazu in 3.3.2.1. Funktionen werden mit dem neuen Keyword *func, proc* oder *method* eingeleitet, die Rückgabetypen werden nach der Parameterliste angegeben, mehr dazu in 3.3.5.1.

Die übrigen Aspekte der Syntax von Myll werden im Laufe des Kapitels 3.3 erklärt.

3.2.3 Gesamtbild

Dieses Beispiel braucht noch nicht vollständig verstanden zu werden, es soll nur einen ersten Grundeindruck vermitteln.

```
module MyContainers;                // Zielmodul angeben

import std_iostream;                // Benötigte Module einbinden

func main( int argc, char*[] argv ) -> int {
    using std, JanSordid::Container;

    var MyStack<i8> stack;          // i8 ist ein 8-Bit-Integer, ähnlich char
    stack.push( 42 ).push( 23 ).push( 16 ).push( 15 ).push( 8 ).push( 4 );

    // Ausgabe aller Zahlen in umgekehrter Reihenfolge des Hinzufügens, last-in first-out
    if( !stack.empty() ) {
        cout << stack.top() << endl;
        stack.pop();
    }
    return 0;
}

// Namensbereich definieren für folgende Konstrukte
namespace JanSordid::Container;

class MyStack<T> {                  // Einfache Template-Syntax
    const usize _default_reserved = 8;
    field usize _reserved;
    field usize _size = 0;          // usize ähnelt size_t
    field T[*]! _data;              // Unique-Array-Pointer

  public:
    ctor( usize reserved = _default_reserved ) {
        _reserved = reserved;
        _data     = new T[_reserved];
    }

    [const]                         // Attribute in »[]« beziehen sich auf die nächste Deklaration
    method empty() -> bool          // … const verbietet, wie bei C++, die Änderung des Objekts
        => _size == 0;

    [const, precond(!empty())]
    method top() -> T               // Von C# übernommen: => X; ist die Kurzform von { return X; }
        => _data[_size-1];

    method push( T val ) -> MyStack& {
        if( _size >= _reserved )
            grow();
        _data[_size] = val;
        ++_size;
        return self;
    }

    [precond(!empty())]             // Je nach Parametrierung des Kompilers wird ein assert erzeugt
    method pop() {
        --_size;
    }

  private:
    method grow() {
        const usize new_reserved = _reserved * 2;
        var   T[]*! new_data     = new T[new_reserved];
        do _size times i {
            new_data[i] = _data[i];
        }
        _reserved = new_reserved;
        _data     = new_data;
    }
}
```

3.3 Lösung der problematischen Aspekte von C++

Einige der problematischen Konstrukte in C++ sind der direkten Kompatibilität mit C geschuldet, andere sind von C++ so eingeführt worden.

Die Lösung der Probleme findet sich jeweils direkt im gleichen Absatz. Wenn stattdessen eine *Mögliche Lösung* vorliegt, hat es die Lösung nicht (vollständig) in den Prototypen geschafft.

3.3.1 Deklarationen, Definitionen und Initialisierung

C++ verhält sich selbst in sehr simplen Beispielen, wie eine *One-Pass parsende* Programmiersprache (was sie nicht ist) und erfordert dadurch vom Programmierer ein hohes Maß an Aufmerksamkeit bei der Reihenfolge der Programmierung oder fordert die Wiederholung von bereits Niedergeschriebenem.

3.3.1.1 Reihenfolge und Prototypen

Dieser Code kompiliert so nicht, er wirft den Fehler, dass `a()` nicht deklariert wurde. Die Regel die hier nicht erfüllt wird ist C++ Std 3.4.1/4 "A name used in global scope, outside of any function

```
int main() {
    a();
}
void a() {…}
```

Um diesen Fehler zu beseitigen, müssen wir entweder den Code so umstellen, dass `a()` vor `main()` definiert wird oder wir fügen eine Prototypen-Deklaration für `a()` vor `main()` ein. Im Falle eines alternierend-rekursiven Funktionsaufrufs bleibt nur die Möglichkeit des Prototypen, siehe anhand dieses Beispiels:

```
void partition(…);
void qsort(…)     { … partition(…); … }
void partition(…) { … qsort(…);     … }
```

Computer sind schnell genug diese Probleme für den Nutzer zu lösen. Lediglich der C++-Standard, steht der Unterstützung solch einfacher Inferenz im Wege. Die Regel StdC++:3.4.1/8 und 8.2 erlauben die Invarianz der Reihenfolge innerhalb von Klassen. Das heißt, dass die oben genannten Beispiele, innerhalb von Klassen, funktionieren würden, also keine *Deklaration vor Benutzung* benötigen. Eine Regel, dass dies auch im globalen Scope funktioniert, könnte C++ einführen, aber es gibt bestimmt berechtigte Argumente die gegen diese Einführung sprechen.

Lösung

In Myll gibt es keine separate Deklaration und Definition, beides erfolgt in einem Schritt und die Reihenfolge ihrer ist irrelevant. Sollte im erzeugten C++-Code dadurch ein Reihenfolgeproblem entstehen, wird automatisch eine Prototyp-Deklaration vor der Nutzungsstelle erzeugt.

3.3.1.2 Aufteilung auf verschiedene Orte

Deklarationen, Definitionen und Initialisierung können entweder innerhalb einer Datei an verschiedenen Stellen und verschiedenen Schreibweisen vorkommen oder oft sogar über Dateigrenzen hinweg verteilt sein.

Dies macht den Code schwerer durchschaubar. Jegliche andere moderne Programmiersprachen erlauben es, oder erzwingen es gar, alles, was z.B. zu einer Klasse gehört, in einer einzelnen Datei abzulegen.

Dieses Beispiel soll einige der Probleme aufzeigen.

Datei *myclass.h*:

```
class MyClass {
    int myField1;
    int myField2;
    int myField3 = 3;              // Funktioniert seit C++11
    static int myStatic;           // Funktioniert nicht wie myField3
public:
    MyClass();
    int myMethod1();
    int myMethod2() {…}            // Kann inlined werden, implizit weil inline implementiert
    inline  int myMethod3() const;
    virtual int myVirtual();
}
int MyClass::myMethod3() const {…} // Kann inlined werden, Keyword const wird wiederholt
```

Datei *myclass.cpp*:

```
#include <myclass.h>
int MyClass::myStatic = 1;         // Keine Wiederholung des Keywords static, Typ int wiederholt
MyClass::MyClass() : myField(1) {
    myField = 2;
}
int MyClass::myMethod1() {…}       // Kann nicht ohne Weiteres inlined werden
int MyClass::myVirtual() {…}       // Keine Wiederholung des Keywords virtual
```

Würde man in diesem Beispiel *myMethod1* mit `inline` auszeichnen, würde weder inline-fähiger Code entstehen, noch würde davor gewarnt, dass diese Auszeichnung keinen Nutzen hat.

Bewertung

- Manche Keywords müssen in Deklaration und Definition geschrieben werden, andere nicht
- Der Abschnitt "`MyClass::`" wird sehr oft wiederholt geschrieben
- Oft gibt es mehre unterschiedliche Möglichkeiten, das Gleiche zu schreiben

Lösung

In Myll wird solch zusammenhängender Code nicht in mehrere Dateien aufgespalten. Es werden natürlich für den erzeugten C++-Code weiterhin .h und .cpp Dateien erzeugt.

Das folgende Beispiel enthält die gleiche Menge an Information, wie die Dateien *myclass.h* und *myclass.cpp*. Es lassen sich aus diesem Beispiel die originalen Dateien (fast identisch) wiederherstellen. Das Attribut `inline` entscheidet hier, ob die Implementierung in der .cpp oder der .h Datei landet, es hat immer eine Auswirkung.

Datei *myclass.myll:*

```
class MyClass {
    field int myField1;
    field int myField2;
    field int myField3 = 3;
    [static] field int myStatic = 1;
  public:
    ctor() {
        myField1 = 1;              // Generiert, wenn möglich, C++-Code wie myField1 im Original
        myField2 = 2;              // Generiert, wenn möglich, C++-Code wie myField1 im Original
    }
    method myMethod1() -> int {…}
    [inline]       method myMethod2() -> int {…}
    [inline,const] method myMethod3() -> int {…}
    [virtual]      method myVirtual() -> int {…}
}
```

Analyse des Ursprungscodes und der bereinigten Syntax, ohne Leerzeilen, Leerraum und Kommentare:

Ursprung, C++: 18 Zeilen, 34 Wörter, 351 Zeichen
Bereinigung: 13 Zeilen, 26 Wörter, 264 Zeichen

Diese Transformation brachte in diesem sehr simplen, von Implementierung befreiten, Beispiel eine Reduktion der Zeilen um 28%, der Wörter um 24% und der Zeichen um 21%. Es geht in diesem betrachteten Punkt des Konzepts auch explizit nicht um die Implementierung. Größere und realistischere Beispiele werden im späteren Verlauf der Thesis betrachtet und ausgewertet. **??? neu berechnen**

Die Fähigkeit etwas zu *inlinen* ist im Allgemeinen sehr wichtig und wird besonders in großen Projekten, bei denen (übermäßig) viel Wert auf Abstraktion, klar separierte Schichten und Indirektionen gelegt wird, benötigt. Als Beispiel kann schon mal etliche Ebenen tief immer nur ein und derselbe Wert durchgegeben werden. Sollte dies nicht *inlined* werden springt man viele, möglicherweise weit von einander entfernte Funktionen an (was zu Cache-Misses und Cache-Pollution führt, siehe [CACHE]). Inlining macht, in diesem Fall, aus vielen verketteten Call- und Ret-Instruktionen, eine oder wenige Load-Instruktionen.

3.3.1.3 Aufteilung im Kontext sich stetig verändernder Codebase

Die Verteilung auf mehrere Dateien ist besonders problematisch, wenn ein C++-Projekt ständig signifikante Änderungen / Restrukturierungen erlebt. So muss im Beispiel, das eine Methode *inline*-fähig gemacht werden soll (selbst wenn die Logik unverändert bleibt), ihre komplette Implementierung aus der .cpp Datei ausgeschnitten und in die .h Datei eingefügt werden. Dadurch verliert man in seiner Versionsverwaltung den Bezug und kann nicht mehr einfach die Veränderungshistorie der Methode nachschlagen.

Das gleiche Problem tritt auf, wenn eine Klasse parametrisiert werden soll (mittels Templates), nur muss in diesem Fall die **komplette** Implementierung in die Header Datei transferiert werden.

Lösung

Sich ständig wandelnder Code profitiert von der irrelevant gewordenen Reihenfolge von Deklarationen und der Vereinigung auf eine Datei. Ein hinzugefügtes oder entferntes Attribut, erzeugt eine geänderte Zeile im *diff* der Versionen, der C++-Code kann sich dadurch signifikant ändern.

Wenn in Myll eine Klasse generisch gemacht wird, fügt man einfach den Template-Parameter hinzu (und ändert natürlich die Vorkommen der zu ersetzenden Typen durch z.B. `T`), sonst nichts.

3.3.2 Fehleranfällige Deklarationen

Wie findet die Deklaration von Variablen, Funktionen, Pointern und Typen in C und C++ statt und welche problematischen Situationen können dabei entstehen?

3.3.2.1 Ambiguität einer Deklaration – Kenntlich machen von Variablen

Sind die folgenden Statements Deklarationen oder nicht?

```
a * b;
c d;
c e();
```

Die Antwort darauf lautet: *Es kommt drauf an.*

Sollte *a* ein Typ sein, wird *b* als ein Pointer auf ein *a* deklariert, ist *a* aber eine Variable wird die Multiplikation (`a.operator*(b)`) aufgerufen. Obwohl die zweite und dritte Zeile sehr ähnlich aussehen, sind sie doch

recht unterschiedlich; *d* ist eine Variable vom Typen *c*, *e* ist aber keine Variable vom Typen *c*, welche den Standardkonstruktor aufruft, sondern ein Prototyp einer Funktion, welche den Typ *c* zurückgibt.

Diese Probleme bieten sich nicht nur dem Leser dieser Statements, sondern auch der Compiler musste aufwändiger erstellt werden, um diese Konstruktionen richtig interpretieren zu können [LEX-HACK].

Lösung

Um der Ambiguität, ob etwas beispielsweise eine Variable, eine Multiplikation oder ein Funktionsprototyp ist, entgegen zu wirken, führt Myll Keywords zur einfachen Identifikation von Variablen- und Funktionsdeklarationen ein.

```
var a * b;          // Variablendeklaration, b ist Pointer auf Typ a
const a * b;        // Konstantendeklaration (mit fehlender Initialisierung)
a * b;              // Multiplikation
var c d;            // Variablendeklaration, d ist Objekt von Typ c
var c e();          // Variablendekl., e ist Objekt von Typ e, mit Aufruf des Standard Konstruktors
func e() -> c;      // Funktionsdeklaration (unnötig in Myll, mehr dazu im dedizierten Kapitel)
```

In Myll wird das Keyword *var* für Variablen, *const* für Konstanten (was in C++ bereits so aussieht) und *func* für Funktionen eingeführt (mehr zu Funktionen in 3.3.5.1).

Außerdem ist es gleichförmig zu anderen Konstrukten in C++, welche mit einem vorangestellten Keyword dem Leser direkt wissen lassen, was folgen wird: *class*, *struct*, *union*, *enum*, *namespace*, *using*, *static*; selbst *template*, *typename*, *typedef*, *#define* erfüllen dieses Kriterium.

Inspiriert war diese Syntax von einigen anderen Programmiersprachen, aber die Entscheidung diese Keywords zu verwenden, kam durch das Projekt Bitwise von Per Vognsen, welcher in seiner Programmiersprache **Ion** die gleichen Keywords verwendet [PV-ION].

3.3.2.2 Pointer, Referenz & Array

Pointer

Naiv hätte ich bei der Variablendeklaration von

```
int * ip, jp;
```

erwartet, dass *ip* und *jp* Pointer auf *int* sind. Damit lag ich (so wie viele meiner Kommilitonen und Kollegen) falsch und als ich zum ersten Mal über dieses Problem stolperte, stellte sich nach etwas Recherche heraus, dass *ip* ein Pointer und *jp* nur ein Objekt vom Typ *int* waren. Sollte man zwei Pointer haben wollen wäre

```
int *ip, *jp;
```

richtig gewesen. Aus diesem Grund empfehlen viele Bücher, dass man nur eine Variable pro Zeile deklariert, aber gerade jene Einsteiger, welche über dieses Problem stolpern, lesen (wenn überhaupt) nicht zuerst Best-Practice-Bücher.

Der Ursprung dieser Syntax ist in Kernighan & Ritchie: *The C Programming Language* [KR-CPL] auf Seite ??? so beschrieben:

> *"The declaration of the pointer ip,*
>
> ```
> int *ip;
> ```
>
> *is intended as a mnemonic; it says that the expression *ip is an int."*

Das heißt, dass man später durch Aufruf von **ip* an das bezeigte *int* gelangt. Kommt noch ein zweiter Variablenname hinzu, muss auch er mit der Eselsbrücke versehen werden, wie man an sein *int* gelangt.

Obwohl ich diese Gedächtnisstütze jetzt schon lange kenne, arbeitet mein Hirn mit einem anderen mentalen Modell. In diesem Modell besteht eine Variablendeklaration aus einem Typ (der Indirektionsebenen beinhaltet) und einem oder mehrerer Namen. Nicht aus einem Basistyp (ohne Indirektionsebenen) und Namen vermischt mit Eselsbrücken, wie ich an den Basistyp komme.

Als eines von vielen möglichen Beispielen von Template-Klassen, verhalten sich die in C++11 eingeführten Smartpointer getreu meinem mentalen Modell, sodass

```
unique_ptr<int>  ip, jp;
```

jeweils *ip* und *jp* als Unique-Pointer auf ein *int* definiert werden.

Referenzen

Das gleiche Problem der Pointer gilt auch bei der Deklaration von Referenzen. Sie haben nur den wichtigen Unterschied, dass die Mnemonik bei ihnen nicht mehr gilt, denn um aus einer mit

```
int &ir;
```

definierten Referenz das *int* abzurufen, darf man nicht `&ir` schreiben, denn dadurch würde man einen Pointer auf das *int* erhalten. Dafür kann C nichts, weil Referenzen erst in C++ Einzug hielten.

Arrays

Ein ähnliches Bild, wie bei den Pointern, findet sich bei der Syntax zur Definition von Arrays [KR-CPL]:

> *"The declaration*
> `int a[10];`
> *defines an array a of size 10, that is, a block of 10 consecutive objects named a[0], a[1], ... , a[9]."*

Auch hier kommt man, im Nachhinein durch Aufruf von `a[zahl]` an die jeweiligen *int.* Interessanterweise ist aber genau der Aufruf von `a[10]` ein Fehler, welcher hinter das Ende des Array adressiert und dadurch UB hervorruft. Um zwei oder mehr Arrays zu erzeugen, muss man dies hier schreiben: `int a[10], b[10];`

Das C++11-Äquivalent dazu ist `array<int,10> a, b;`, bei dem wieder *a* und *b* jeweils Arrays der Größe 10 vom Typ *int* sind.

Lösung

In Myll ändert sich die Pointer-, Referenz- und Array-Syntax so, dass der Stern, das Kaufmanns-Und sowie die eckigen Klammern zum Typ gehören und der Typ für alle Variablennamen gleich ist:

```
var int*     a, b; // Variablen a und b sind beide Pointer auf int
var int&     c, d; // Variablen c und d sind beide Referenzen auf int
var int[10]  e, f; // Variablen e und f sind beide int Arrays mit 10 Elementen
```

Verloren geht dadurch die Möglichkeit total unterschiedliche Initialisierungen, wie diese hier in einer Zeile schreiben:

```
int i = 1, *p = NULL, f(), (*pf)(double), a[10];
```

Diese Änderungen beseitigen nicht nur die potentiell fehleranfällige Syntax von K&R C, sondern harmonisieren außerdem mit der Template-Schreibweise in C++ und der allgemeinen Schreibweise in anderen Sprachen wie C#, Java, D und vielen weiteren.

Dies ist ein Trade-off, welcher die eine Art der Deklaration vereinfacht und die andere erschwert. Ich bin der Meinung, dass sich diese Veränderung insgesamt positiv auswirkt.

3.3.2.3 Die Kombination von Pointern und Arrays

Sofern man Variablen immer jeweils einzeln erzeugt oder man sich mit dem mentalem Modell aus [KR-CPL] anfreunden konnte, aber man dennoch nicht auf etwas Problematik verzichten will, dann kann man die beiden zuvor genannten Fälle kombinieren:

```
int *c[4];
```

Man weiß hier zwar das `*c[4]` ein *int* ist, aber der Anteil mit dem *int* ist auch das leicht Verständlichste.

Ob dies ein Pointer auf ein Array mit 4 *int* ist oder ob es ein Array aus 4 Pointern auf *int* ist, wird nicht direkt ersichtlich.[1] Dafür muss man wissen, welche Operation zuerst ausgeführt wird, die Dereferenzierung oder das Subskript. Und es stellt sich noch eine weitere Frage: Wenn es das Eine ist, wie bekommt man es dazu das Andere zu sein?[2]

Alternativ dazu sind die C++11 Äquivalente zwar schreibintensiv, aber eindeutig und direkt verständlich:

```
unique_ptr<array<int,4>>  uptr_to_ary;     // Ein Unique-Pointer auf ein Array aus 4 int
array<unique_ptr<int>,4>  ary_of_uptr;     // Ein Array aus 4 Unique-Pointern auf int
```

Lösung

Die Kombination aus Pointern und Arrays ist durch die bereits in 3.3.2.2 genannten Änderungen schon abgedeckt und die Unklarheit der Reihenfolge, welche in C und C++ vorhanden war, ist nicht mehr vorhanden. Es gibt:

```
var int[4]*  ptr_to_ary_of_4;      // Ein Pointer auf ein Array aus 4 int
var int*[4]  ary_of_4_ptrs;        // Ein Array aus 4 Pointern auf int
```

Dadurch besteht keine Notwendigkeit, Klammern zu setzen und es ist auch nicht so schreibintensiv wie die Template-Syntax.

3.3.2.4 Funktionspointer

Ein besonders schwer zu lesender und schreibender Teil der C und C++ Syntax ist die der Funktionspointer. Hier zwei Beispiele:

```
int f(int a,int b) {…}       // Definition der Funktion f mit zwei int Parametern und int Rückgabe
int ff(int(*f)(int,int)) {…} // Definition der Funktion ff die einen Parameter vom Typ von f hat
int (*fp)(int,int) = f;      // Pointer auf eine Funktion fp vom gleichen Typ wie f, die auf f zeigt
void (*ffp)(int(*)(int,int));// Pointer auf eine Funktion ffp vom gleichen Typ wie ff
```

Das Problematische sind hier die Funktionspointer, deren Namen inmitten der Deklaration *versteckt* sind. Dies wird noch undurchsichtiger bei verschachtelten Funktionspointern, da diese namenlos nur durch ein "`(*)`" identifiziert werden.

Die dritte Zeile kann auf den ersten Blick für eine *Functional-Cast-Expression* oder eine Deklaration eines Pointers auf ein *int* gehalten werden; vergleiche mit der Auflösung des vorangegangenen Punktes:

```
int(*fp)(int,int);
int(*c)[4];
```

1 Es ist ein Array mit 4 Pointern auf *int*.

2 Die Umkehr wird durch Klammersetzung im Typen erreicht: *int* (*c)[4];

Lösung

In Myll erkennt man am linken Rand der Deklaration sehr schnell, ob etwas eine Funktion, ein Funktionspointer oder etwas anderes ist, die Namen sind außerdem leicht auffindbar; bei Funktionen direkt am Anfang und bei Variablen an der rechten Seite der Deklaration. Auch verschachtelte Funktionspointer sind nicht schwerer lesbar als Funktionspointer im Allgemeinen.

Das identische Beispiel in Myll:

```
func f(int a,int b)->int {…}
func ff(func(int,int)->int f)->int {…}
var func(int,int)->int fp = f;
var func(func(int,int)->int)->void ffp;
```

3.3.3 Schlechtes Standardverhalten

3.3.3.1 Konstruktor mit einem Parameter

Ein Konstruktor, welcher einen einzelnen Parameter übernimmt, erfüllt nicht nur die Rolle eines Konstruktors mit einem Parameter, sondern er exponiert damit auch die Funktionalität der impliziten Konvertierung des Parametertypen zum Klassentypen. Dieses Standardverhalten bringt den nicht-Profi in die Situation, Funktionalität bereitzustellen, ohne davon zu wissen. Die Kennzeichnung als *explicit* verhindert diese Bereitstellung.

```
class C {
    C(int) {…}

    explicit C(float) {…}
};
C c1 = C(1);       // OK
C c2 = 2;          // Was soll das? Verwirrende Bereitstellung von Funktionalität
C c3 = C(3.0f);    // OK
C c4 = 4.0f;       // Error, musste explizit Verboten werden
```

Der Entwickler wollte *c1* und bekommt *c2* ohne jede Warnung hinzu. Gleichfalls funktioniert *c3*, nur *c4* wird mit einem Kompilationsfehler verhindert.

Lösung

Jonathan Müller, C++ Library Developer und Konferenzsprecher, betrachtete dieses Problem in einem Blogpost und behauptet [JM-EA]:

> *"As with most defaults, this default is wrong. Constructors should be explicit by default and have an implicit keyword for the opposite."*
>
> *Ins Deutsche: "So wie die meisten Voreinstellungen ist diese falsch. Konstruktoren sollten standardmäßig explicit sein und ein implicit Keyword für das Gegenteil haben."*

Ich teile seine Meinung, deswegen setzt Myll genau diese Umkehrung der Bereitstellung um. Sollte die implizite Konvertierung gewünscht sein, muss sie explizit erwähnt werden. Besonders als Anfänger der Sprache erlebt man so keine Überraschungen, durch nicht angeforderte Verhaltensweisen.

Dieses Beispiel in Myll weist die gleiche Bereitstellung wie im C++-Beispiel auf:

```
class C
{
    [implicit] ctor(int) {…}

    ctor(float) {…}
}
```

```
var C c1 = C(1);    // OK
var C c2 = 2;       // OK, man wollte explizit die implizierte Konvertierung von int erlauben
var C c3 = C(3.0f); // OK
var C c4 = 4.0f;    // Error, man bekommt standardmäßig nur den float Konstruktor
```

Der Entwickler bekommt ein funktionierendes *c1* und, wie angefordert, auch *c2*. Gleichfalls funktioniert *c3*, jedoch wird *c4* mit einem Kompilationsfehler verhindert.

3.3.3.2 Private Ableitung

Die private Ableitung von Klassen ist in C++ das Standard-Verhalten einer Ableitung. Im Buch "The C++ Programming Language" wird die Nutzung einer privaten Ableitung als *hat-ein-Beziehung* zur Basisklasse präsentiert. Solch eine Beziehung hat durchaus einen Nutzen, sie lässt sich aber auch mittels eines privaten Members umsetzen. Der Vorteil sie als Member so definieren ist, dass man problemlos mehrere solcher Beziehungen modellieren kann und sich diese Modellierung auch auf andere Sprachen übertragen lässt.

```
class D : C {…};                    // D leitet private von C ab, was keinen Sinn ergibt…
class D { C base; … };              // …sollte man dies gewollt haben, kann man es auch so schreiben
class D : public C {…};             // Dies ist der Fall den man üblicherweise will
```

Lösung

In Myll sind Ableitungen standardmäßig *public*, wie auch in vielen anderen OOP-Sprachen.

```
class D : C {…}                     // Dies ist der Fall, den man üblicherweise will
class D : private C {…}             // Wenn man will, kann D private von C ableiten
```

3.3.3.3 Alles kann Exceptions werfen

Funktionen sind, wieder aufgrund von Kompatibilitätsgründen, auch ohne explizite Auszeichnung so definiert, dass sie Exceptions werfen können. Das führt dazu, dass man in Projekten mit wenigen Exceptions jede Funktion mit *noexcept* auszeichnen muss (alternativ erzeugt der Compiler eine suboptimale Ausgabe).

In C# gibt es die Klasse Dictionary (äquivalent zu *std::map*), welches Key/Value Paare speichert. Die Abfrage ob ein Key vorhanden ist wurde anfangs (~2005) von den mir bekannten Tutorials so empfohlen:

```
try {
    var val = myDict[keyThatMightNotBeThere];      // wirft eine Exception wenn Key nicht vorhanden
} catch (KeyNotFoundException) {
    myDict.Add(keyThatMightNotBeThere, something); // was passiert wenn der Key nicht vorhanden war
}
```

Kein Wunder, denn Exceptions sind ja schließlich dass was man in jeder Objektorientierten Sprache als die Nonplusultra-Alternative zu Error-Codes o.ä. angepriesen bekommt. Wenn *myDict* aber erst über die Zeit wächst, ist das kein *Ausnahmefall*, es ist etwas, was ständig passiert, für jeden neuen Key, welcher dem Dictionary hinzugefügt wird.

Alternativ zur Lösung mit Exceptions, kann man auch mittels `TryGetValue(value, out value)` prüfen, ob ein Wert im Dictionary existiert und es gleichzeitig abfragen.

```
if(myDict.TryGetValue(keyThatMightNotBeThere, out val)) {
    val;                                           // irgendwas mit val machen
} else {
    myDict.Add(keyThatMightNotBeThere, something); // was passiert wenn der Key nicht vorhanden war
}
```

Die Performance der zweiten Variante ist auf meinem PC etwa 400-mal schneller als die erste Variante, die genaue Zahl ist nicht relevant, es geht hier lediglich darum eine Größenordnung aufzuzeigen. Die zweite Variante setzte sich allgemein durch, um Werte, welche fehlen könnten, aus einem Dictionary abzufragen.

Ein Zitat von Tim Sweeney von Epic Games im Januar 2020 auf Twitter [TS-EX]:

> *"Just eliminated C++ exceptions in a complicated multithreaded library. It's 15% faster despite all performance sensitive functions previously being declared noexcept, with no obvious explanation."*
>
> *Sinngemäß ins Deutsche: "Habe gerade in einer komplexen Mutithreaded Bibliothek C++ Exceptions eliminiert. Sie ist nun 15% schneller obwohl zuvor alle Funktionen auf dem kritischen Pfad noexcept markiert waren. Ich habe keine Erklärung dafür"*

Für uns als Spieleentwickler sind 15% weniger Performance in einem Code, welcher lediglich Exceptions bereitstellt und nicht einmal nutzt, ein extremer Verlust. Diese Menge will kein aktuelles Spiel opfern.

Lösung

Exceptions sollen Ausnahmen sein, nicht die Regel. Nur wenn etwas *Außergewöhnliches (engl.: exceptional)* passiert, sollen sie auftreten. Etwa wenn jemand von Hand die Datei löscht welches das Programm mitbrachte und die eigentlich immer da sein sollte oder wenn die Netzwerkverbindung mitten beim Senden des Paketes abbricht. Es ist nicht *außergewöhnlich*, dass ein Dictionary anfangs keine Keys enthält.

Ich denke, es hat sich erst nach und nach herausgestellt welcher Grad der Benutzung eines Sprachfeatures sinnvoll ist oder auch nicht. Ein Problem welches C++ zusätzlich hat, ist das dessen Exceptions aktuell dynamisch Alloziert sind, siehe diese Präsentation von Herb Sutter [HS-EX] (40:37). In der gesamten Präsentation bringt er auch Kritik an der Überbenutzung von Exceptions zur Sprache. Außerdem regt er an das in Zukunft in weniger Funktionalität der Standardbibliothek Exceptions genutzt werden und zeigt ein neues, zu *noexcept* invertiertes, Keyword namens *throws* auf, welches neuartige Exceptions, ohne dynamische Allokation, werfen kann (welche eher so etwas wie Error-Codes sind). Dies sind alles nur Vorschläge.

Das Attribut *except* ist in Myll nötig um in einer Funktion Exceptions nutzen zu können. Dieses Keyword kann in Myll auch einmalig einer gesamtem Klasse zugewiesen werden und gilt dann für alle Member. Das gegensätzliche Attribut *noexcept* oder *except(false)*, ist weiterhin möglich um in einer allgemein *except* Klasse einzelne Methoden zu markieren.

3.3.3.4 Switch fällt von selbst

Als *Labels* und *goto* noch ein oft genutzter Teil von Programmiersprachen war, konnte man noch einfach verstehen, was an dem folgenden Beispiel nicht wie erwartet läuft. Da aber heutzutage, bis auf das *switch/case*-Statement, alle Kontrollstrukturen durch geschweifte Klammern begrenzt sind, tritt der hier gezeigte Fehler nicht selten auf.

```
switch( a ) {
    case 1:
        do_this();
    case 2:
        or_do_that();
        break;
    case 3:
    case 4:
        two_cases_want_the_same();
        break;
}
```

Hier wird im Falle, dass a = 1 ist *do_this* ausgeführt und nach dessen ordnungsgemäßer Beendigung auch noch *or_do_that*. In den Fällen a = 3 oder a = 4 erwarten wir jedoch den *fallthrough (das durchfallen)* von *case 3* nach *case 4*.

C++ bietet schon die Möglichkeit der Auszeichnung, dass ein *fallthrough* erwünscht war, dieser wird aber nicht erzwungen, siehe [FALL].

Geplante Lösung

Entweder *break* oder *fall* sind die letzten Statements in jedem *case*. Dadurch ist weder der eine noch der andere Fall des Vergessens möglich. Diese Sprache möchte einem nicht mühsam angewöhnte, positive Verhaltensweisen abgewöhnen.

```
switch( a ) {
    case 1:
        do_this();
        fall;                       // Fällt in den darunterliegenden case, Error ohne fall
    case 2:
        or_do_that();
        break;
    case 3, 4:                      // Menn mehrere case genau das gleiche wollen, geht es auch so
        two_cases_want_the_same();
}
```

Oft sieht man auch, dass ein *case* seinen kompletten Inhalt in einen Block einbettet, um beispielsweise lokale Variablen zu unterstützen (diese machen Probleme, wenn sie nicht in einem Block eingebettet sind). Dieser Spezialfall könnte auf den Doppelpunkt verzichten.

Man könnte auch, pro *switch* dessen Standardverhalten entweder auf *break* oder *fall* stellen, damit reduziert man Schreibarbeit, wenn stets das Gleiche gewünscht ist (und kann mit `[default=fall]` den C++-Zustand wieder-herstellen). Auch eine Auszeichnung, dass ein *switch* erschöpfend sein muss, also alle Fälle abdeckt, wäre insbesondere für Enums denkbar. Dies könnte Programmierfehler, durch unbehandelte Fälle verhindern.

Auch die Angabe einer Spannweite im *case* wäre denkbar und könnte Schreibarbeit bei aufeinanderfolgenden Fällen vereinfachen. Das folgende Beispiel existiert bereits als GNU Erweiterung und könnte durch Myll selbst in C++-Compilern genutzt werden, die dies nicht nativ unterstützen.

```
switch( age ) {
    case  0 ... 12: return "child";
    case 13 ... 17: return "teenager";
    case 18 ... 66: return "work-work";
    default:        return "pensioner";
}
```

3.3.4 Implizite Konvertierung

In C++ sind viele unsichere implizite Konvertierungen möglich. Dies betrifft insbesondere numerische Datentypen. Alles was hier steht trifft auch auf 3.3.7 zu.

3.3.4.1 Integrales Heraufstufen – Integer Promotion

```
char a = 17,  b = 4;
char c = a + b;
int  d = 707, e = 1337;
char f = d + e;            // Konvertierungen zu schmaleren Typen erzeugen keine Warnung
static_assert( -1 < 1u );  // Error: -1 ist nicht kleiner als 1
```

Dass die Addition zweier *int* sich sehr wahrscheinlich nicht vollständig in einem *char* speichern lässt (`char f = d + e`), ist kein Zustand über den sich C++ beschwert. Manche Compiler bieten die Option mittels `-Wconversion` die ausreichende Größe des Zieltyps einer solchen Zuweisung zu überprüfen, diese Option warnt jedoch (meines Erachtens *leider*) auch im Falle von `c = a + b`. Noch absurder wird es wenn `-1` nicht mehr kleiner als `1u` sein soll.

Die Erklärung für all dies ist, dass in C++ bei arithmetischen Operationen für alle Ganzzahlen, welche einen geringeren Wertebereich als *int* haben, eine Typenerweiterung durchgeführt wird. Das heißt, dass

sie auf die Größe eines *int* erweitert werden. Sollte jedoch der Wertebereich von *int* nicht ausreichen, wird alternativ zu *unsigned int* konvertiert. Dies passiert auch dann, wenn der Wertebereich von *unsigned int* ebenfalls nicht ausreichend ist. Im Falle der `-1`, welche ein *int* ist, und der `1u`, welche ein *unsigned int* ist, findet die Konvertierung der `-1` so statt, dass sie einfach als *unsigned int* re-interpretiert wird und dann den Wert `4294967295u` hat, welcher natürlich nicht kleiner als `1u` ist.

Geplante Lösung

Das man für die Berechnung von `c = a + b` keine Warnung erhält ist nachvollziehbar. Da C++ aber auch in diesem Fall die *Integer Promotion* vornimmt ist das Resultat aus `a + b` vom Typen *int.* Eine Warnung in diesem Fall wäre ein *false-positive* (eine Warnung, welche keine hätte sein dürfen). Wenn ich also hierfür keine Warnung erhalten will, dann kann ich auch für `f = d + e` keine Warnung erwarten, weil das Ergebnis von `d + e` ebenso *int* ist.
Eine mögliche Lösung des Problems ist *anzunehmen* das keine Integer Promotion stattfindet und dann generell eine strengere Typenüberprüfung anzusetzen. Hier würde dies bedeuten, dass `b + c` ein *char (i8 in Myll)* ergeben würden, welches mit dem empfangenden Typen *char* übereinstimmt. Im Falle der Zuweisung von `f = d + e` würde nun ein *int* in ein *char* gespeichert werden, welches dank der Elimination des *false positive* nun eine Warnung oder gar einen Fehler erzeugen darf.

Unberührt dessen findet effektiv natürlich die Integer Promotion weiterhin statt, denn diese ist hilfreich um etwa dieses Beispiel vom arithmetischen Überlauf abzuhalten.

```
var i8 a = 28, b = 16;      // In Myll ist i8 eine vorzeichenbehaftete 8-bit Ganzzahl
var i8 c = (a * a) / b;     // Mehr zu den Typen von Myll in 3.3.8.1
```

3.3.4.2 Einschränkende Konvertierung – Narrowing Conversion

C++ leidet unter dem Problem, dass es *implizit einschränkende Konvertierungen* durchführt, also beispielsweise ein *int* einfach in ein *bool* konvertiert. Um dieses Problem zu lösen, wurden Initialisierungslisten (mittels `{}`) eingeführt, welche diese Konvertierung, bei einer Initialisierung, verbieten.

```
takeBool(bool b) {…}        // Funktion
struct TakeBool {
    TakeBool(bool b) {…}    // Konstruktor
};
// Ohne Initialisierungsliste
takeBool( 42 );             // Ohne Initialisierungsliste, ohne Warnung wird int in bool konvertiert
TakeBool tbc( 42 );         // Ohne Initialisierungsliste, ohne Warnung wird int in bool konvertiert
bool bool1 = 42;            // Ohne Initialisierungsliste, ohne Warnung wird int in bool konvertiert
// Mit Initialisierungsliste, führt wie gewünscht zu Kompilationsfehlern
takeBool({ 42 });           // Error: no implicit narrowing conversion, sonderbare Syntax
takeBool({ true });         // Okay, aber leider mit Warnung: braces around scalar initializer
TakeBool tbc2{ 42 };        // Error: no implicit narrowing conversion
bool bool2{ 42 };           // Error: wie die Anderen, irritierende Syntax für Wertetyp
```

Das Problem hieran ist das geschweifte Klammern auch für viele andere Konstrukte verwendet werden. Verwandt mit den Initialisierungslisten ist die Initialisierung von Arrays und Strukturen, dessen sie leider sehr ähnlich sieht, um nicht beim Lesen mit ihnen verwechselt zu werden. Außerdem stellt diese Syntax einen Bruch zu eigentlich allen anderen aktuellen objektorientierten Sprachen dar, nur um ein Problem zu lösen, welches eigentlich gar nichts direkt mit Konstruktoren oder mit Initialisierung zu tun hat.

Geplante Lösung

Einschränkende Konvertierung finden in Myll nicht statt, sollte man eine einschränkende Konvertierung wollen, kann man explizit konvertieren. Dies kann natürlich nur unter der Annahme des letzten Abschnitts erfolgen, also, dass es theoretisch kein *integrales Heraufstufen* gibt. Konstruktoren und Initialisierung haben wieder eine gewohnte Optik und geschweifte Klammern sind Strukturen und Arrays vorbehalten:

```
takeBool( 42 );             // Error: Keine implicit narrowing conversion
var TakeBool tbc( 42 );     // Error: Keine implicit narrowing conversion
var bool bool1 = 42;        // Error: Keine implicit narrowing conversion

var i8 a = 28, b = 16;      // Funktioniert weiterhin, denn laut Lösung von 3.3.4.1 ist i8*i8/i8->i8
var i8 c = (a * a) / b;     // … dadurch ist die Zuweisung hier keine implicit narrowing conversion
```

Die Lösung von C, die Funktionalität der Initialisierung von Arrays und Strukturen mittels *designierter Initialisierung* zu erweitern, griffen auch andere Sprachen wie C#, Go und Rust auf und wären eine alternative, nützliche Erweiterung der Syntax gewesen, dies ist hier aber nicht das betrachtete Problem.

3.3.4.3 Konvertierung von Pointern

Pointer sind eine spezielle Art von vorzeichenlosen Ganzzahlen, welche Adressen im Speicher darstellen.

```
int * ip = 0;               // 0 ist implizit zu jeglichem Pointer konvertierbar…
int * jp = NULL;            // …das ist so, weil das NULL-Makro oft einfach als 0 definiert ist
if( a  ) {…}                // Jegliche numerische Datentypen sind implizit zu bool konvertierbar,…
if( ip ) {…}                // …was auch für Pointer gilt
int * sp = new int( 8472 ); // sp ist ein Pointer auf ein einzelnes int …
sp++;                       // … dennoch erlaubt es Pointer-Arithmetik …
sp[23] = 9;                 // … und den Aufruf des Subscript-Operators
void * vp = sp;             // Alle Pointer sind implizit zu void Pointern konvertierbar
```

Um einem Pointer, mit dem besonderen leeren Wert zu initialisieren wird ihm entweder *0*, *NULL* oder *nullptr* zugewiesen, wobei aber nur der letztgenannte vom Typ *Pointer* ist.

Wird ein einzelnes Element als Pointer gespeichert, lässt sich auf diesen Pointer dennoch Arithmetik und der Subscript-Operator anwenden, man könnte hier sagen ein *nicht-arithmetischer Pointer* wird implizit in einen *arithmetischen Pointer* konvertiert.

Geplante Lösung

Einem Pointer kann keine Ganzzahl mehr zugewiesen werden, es sei denn sie wird explizit konvertiert.

Das Keyword *null* wird eingeführt und ist die präferierte Schreibweise für Pointer, welche nirgendwohin zeigen, *nullptr* ist weiterhin möglich, *NULL* ist nicht mehr möglich (man kann natürlich eine Konstante mit dem Namen erzeugen und diese nutzen).

Ein Pointer auf ein einzelnes Objekt / einen Skalar **verbietet** die Nutzung von Arithmetik und dem Subscript-Operator. Arithmetik und der Subscript-Operator sind lediglich *Pointern auf Arrays* vorbehalten, welche ein neuer Pointer-Typ sind. Sie haben diese Schreibweise:

```
var int[*] a;                       // int[]* war nicht möglich da dies schon eine Bedeutung hat
```

Damit lassen sich schon in Myll Fehler diagnostizieren, wenn ein Pointer auf ein Skalar so verwendet wird wie ein Pointer auf ein Array z.B.:

```
var int      skl;
var int[10]  ary;
var int*     ptr_to_skalar = &skl;
var int[*]   ptr_to_ary1   = &ary; // Expliziter Decay des Arrays zu einem Array Pointer
var int[*]   ptr_to_ary2   = &skl; // Error: Skalar kann nicht in Array Pointer konvertiert werden
ptr_auf_skalar++;                  // Error: Pointer auf Skalare verbieten Arithmetik
ptr_auf_ary++;                     // Funktioniert wie erwartet
```

Auch wird der implizite *Decay* (Verfall eines Arrays zu einem Pointer) von Arrays zu Pointern verhindert. Will man, dass ein Array zum Pointer wird, nutzt man die identische Syntax um einen Pointer auf ein Skalar zu bekommen.

Die einzige Ausnahme des Arithmetik-Verbots für Skalar-Pointer kann die Differenzbildung (`ptr1 - ptr2`) zweier Pointer sein, etwa zur Distanzermittlung im Speicher.

3.3.5 Inkonsistente Syntax

Aufgrund des Alters der Sprache gibt es oft mehr als eine Möglichkeit oder einen Stil etwas zu schreiben.

3.3.5.1 Funktions- & Methodendeklarationen

In C++ gibt es zahlreiche unterschiedliche Möglichkeiten, Funktionen und Methoden zu definieren.

```
float pow(float b, int e)           {…}    // nicht optimal weil fest vorgegebener Typ
auto pow(auto b, int e) -> float    {…}    // ab C++11, auto vorn überflüssig, aber syntaktisch nötig
auto pow(auto b, int e)             {…}    // ab C++14
template <typename T>
T pow(T b, int e)                   {…}    // Analog auch auto Rückgabetyp, auch anhängend möglich
[](auto b, int e) -> auto           {…}    // Lambda-Syntax, kein Rückgabetyp vorweg
```

Die Spezifikation der Typen von Parameter und Rückgabe ist in jedem aufgeführten Fall unterschiedlich, besonders die Positionierung des Rückgabetypen variiert stark. Die mit C++11 eingeführte Lambda-Syntax besitzt gar keine Möglichkeit eines vorangestellten Rückgabetypen mehr.

Lösung

So sehen die identischen Deklarationen in Myll aus.

```
func pow(float b, int e) -> float  {…}     // nicht optimal weil fest vorgegebener Typ
func pow(auto b, int e) -> auto    {…}     // Rückgabetyp
func pow(auto b, int e)            {…}     // das gleiche wie die Zeile davor
func pow<T>(T b, int e) -> T       {…}     // Auch auto Rückgabetyp möglich
func(auto b, int e) -> auto        {…}     // Lambda-Syntax, fast identisch zur Funktions-Syntax
```

Funktionen und Methoden aller Art werden mit dem neuen Keyword *func* eingeleitet, die Rückgabetypen werden nur folgend angegeben. In der Evolution von C++ wurde der folgende Rückgabetyp mit C++11 eingeführt und gewann seitdem immer mehr an Bedeutung und Funktionalität. Die Template-Syntax wurde für simple Fälle wie diesen hier entschlackt. Lambdas sind nun leichter identifizierbar und, bis auf ihre Namenlosigkeit, identisch in der Schreibweise zu Funktionen.

Die Nutzung der Keywords *proc* und *method* ist auch möglich, *proc* ist momentan komplett Synonym zu *func*, *method* ist hingegen nur im Klassenkontext nutzbar.

```
proc do_something()                {…}     // Momentan Synonym zu func
method draw()                      {…}     // Innerhalb von Klassen Synonym zu func und proc
```

Analog zu *const* Methoden, welche eine Veränderung des assoziierten Objektes verbieten, gibt es die Möglichkeit jede Funktion als *pure* zu markieren, welche die Veränderung von jeglichem globalen Zustand verbietet.

3.3.5.2 Notwendigkeit des Semikolons

Manchmal braucht man ein Semikolon nach Sprachkonstrukten und ein andermal braucht man es bei sehr ähnlichen Konstrukten nicht.

```
class E {…};
do {…} while (…);

namespace A {…}
while (…) {…}
if (…) {…}
```

Lösung

Ohne die Notwendigkeit, wie in C, für alles via *typedef* einen leicht nutzbaren Alias zu erzeugen, fällt auch die Notwendigkeit des Semikolons nach Klassen, Strukturen, Unions und Enumerationen weg. Verloren ist die Möglichkeit der direkten Erzeugung von Variablen des neuen Typs.

Für *do-while Schleifen* hat das Semikolon noch nie wirklich Sinn gemacht, wenn man es dennoch schreibt, ist es valider Code. Wie gehabt will diese Sprache einem maximal schlechte Verhaltensweisen abgewöhnen, keine Guten oder Neutralen.

```
class E {…}
do {…} while (…)
```

3.3.5.3 Verteilung der Attribute

In C++ sind die Attribute einer Funktion / Methode chaotisch um den Kern der Deklaration verteilt.
Gegeben sei dieses Beispiel, zur besseren Lesbarkeit nach logischen Gruppen umgebrochen, formatiert und erklärt:

```
[[noreturn]] virtual const float const foo() const override final noexcept {…}

[[noreturn]]                  // Attribut welches angibt das die Funktion nicht zurückkehren wird
virtual                       // Methode ist virtuell überschreibbar
const float const             // Rückgabetyp
foo()                         // Methodenname und leere Parameterliste
const override final noexcept // Methode verändert this nicht, ist überschrieben, nicht weiter…
{…}                           // …überschreibbar und wirft keine Exceptions
```

Man beachte, dass *const* dreimal vorkommt, wobei sich die ersten zwei Vorkommen (redundant) auf den Rückgabeparameter[3], das Dritte auf die Methode selbst bezieht. Man betrachte das *virtual*, *override* und *final* verwandte Terme sind (die innerhalb einer Definition, wie hier aufgeführt, nicht gemeinsam verwendet werden) aber dennoch an unterschiedlichen Positionen geschrieben werden.

Diese wahllos wirkende Verteilung der Attribute ist der Evolution von C++ geschuldet, *virtual* ist ein Keyword aus der Entstehung von C++, *override* und *final* sind relativ neu und lediglich *identifiers with special meaning*, welche nur in einem speziellen Kontext als Keyword dienen. In anderen Kontexten können sie etwa als Typ oder Variablenname (Identifier) verwendet werden, aus diesem Grund können sie nicht an der gleichen Position wie *virtual* geschrieben werden.

Lösung

Das gleiche Beispiel, auch hier nach logischen Gruppen formatiert:

```
[noreturn, pure, virtual, override, final, noexcept] func foo() -> const float {…}

[noreturn, pure, virtual, override, final, noexcept] // Alles was sich auf die Methode bezieht
func foo() -> const float                            // Keyword, Name und Rückgabetyp mit const links
{…}
```

Myll gruppiert Attribute in eckigen Klammern vor dem jeweiligen Konstrukt, dadurch sind zukünftige Erweiterungen einfach möglich, ohne das es Kollisionen mit bestehendem Code & Syntax gibt. *West const* (Siehe 3.3.10) ist hier die einzig korrekte Schreibweise, um etwas *unveränderbar* zu machen.

3.3.6 Problematische Reihenfolge der Syntax

Expressions (nicht nur Operatoren, so wie der Name vermuten lässt) sind im aktuellen C++ in 17 Ebenen einer Vorrangs-Tabelle eingeteilt. Geringes Precedence-Level bindet stärker als Höheres.

3 Und ja, mir ist klar das es bei einer by-value Rückgabe irrelevant ist ob sie *const* ist oder nicht

Tabelle 3:1: Operator Precedence Table

Precedence	Operator
1	`::`

2	`e++`, `e--`
	`e()`
	`e[]`
	`e.e`
	`e->e`
	`typeid()`
	`const_cast`
	`dynamic_cast`
	`reinterpret_cast`
	`static_cast`

3	`++e`, `--e`
	`+e`, `-e`
	`!e`, `~e`, `*e`, `&e`
	`(type)`
	`sizeof`
	`new`, `new[]`
	`delete`, `delete[]`

4	`e.*e` `e->*e`

5	`e * e`
	`e / e`
	`e % e`

6	`e + e` `e - e`

Precedence	Operator
7	`e << e`, `e >> e`
8	`e <=> e`
9	`e < e`, `e <= e` `e > e`, `e >= e`
10	`e == e`, `e != e`
11	`e & e`
12	`e ^ e`
13	`e | e`
14	`e && e`
15	`e || e`

16	`e ? e : e`
	`e = e`
	`e += e`
	`e -= e`
	`e *= e`
	`e /= e`
	`e %= e`
	`e <<= e`
	`e >>= e`
	`e &= e`
	`e ^= e`
	`e |= e`
	`throw`

17	`e, e`

3.3.6.1 Unerwartete Reihenfolge der Operatoren

Manche dieser Ebenen sind in einer Reihenfolge, in welcher man sie vielleicht nicht vermuten würde. Die Kommentare zeigen mittels Klammerung welcher Teil zuerst evaluiert wird.

```
cout << 1 & 2;              // (cout << 1) & 2;
if( a ^ b == 2 ) …          // if( a ^ (b == 2) ) …
cout << a == b;             // (cout << a) == b;
```

Die Einordnung in der Vorrangs-Tabelle der Bitweisen *Und* und *Oder* Operation geht auf BCPL zurück. Dort erfüllte der Operator `&` kontextsensitiv entweder die Funktionalität der Konjunktion (logischem *Und*) oder des bitweisen *Und*, analog traf dasselbe auf den Operator `|` und *Oder* zu. In der Entwicklung von C wurde die Funktionalität in `&` und `&&` / `|` und `||` aufgespalten [CRIT-OP]. Als Kontext sei auf die Operatorliste von BCPL hingewiesen [BCPL-OP].

Dennis Ritchie schrieb dazu in [DR-CDEV]:

> *Their tardy introduction explains an infelicity of C's precedence rules. In B one writes*
> `if (a == b & c) ...`
> *to check whether* `a` *equals* `b` *and* `c` *is non-zero; in such a conditional expression it is better that* `&` *have lower precedence than* `==`*. In converting from B to C, one wants to replace* `&` *by* `&&` *in such a statement; to make the conversion less painful, we decided to keep the precedence of the* `&` *operator the same relative to* `==`*, and merely split the precedence of* `&&` *slightly from* `&`*. Today, it seems that it would have been preferable to move the relative precedences of* `&` *and* `==`*, and thereby simplify a common C idiom: to test a masked value against another value, one must write*
> `if ((a & mask) == b) ...`
> *where the inner parentheses are required but easily forgotten.*

Die Notwendigkeit B Programme nach C umzusetzen ist nicht mehr gegeben, dennoch trägt C++ immer noch diesen Ballast mit sich herum.

Lösung

In Java und C# sind die Operatoren &, | und ^ in ähnlichem Vorrang wie in C++ und seinen Vorgängern einsortiert und diese verhalten sich je nach booleschen oder ganzzahligem Kontext, analog zu C++.
Die Sprachen Rust und Ruby hingegen sortieren die Operatoren &, | und ^ direkt über die Vergleichsoperatoren ein und eliminieren die angesprochene Problematik.

Myll ordnet die booleschen Operatoren in die Ebenen der Punkt- und Strichrechnung ein: Operator & in die Ebene der Punktrechnung, | und ^ in die Ebene der Strichrechnung.

Die Idee hierzu kam durch das Projekt *Bitwise* von Per Vognsen, welcher in seiner Programmiersprache **Ion** die gleiche Änderung der *Operator Precedence* umgesetzt hat. Vognsen lies sich von der Programmiersprache Go inspirieren.

Der letzte aufgeführte Problemfall wurde nicht behandelt, damit dies hier noch so funktioniert:

```
if( a << 1 == 24 ) // if( (a << 1) == 24 )
```

3.3.6.2 Präfix und Suffix Operationen

Ähnlich zur Variablendeklaration von Pointern, in Kombination mit Arrays, bringt auch das Zusammenspiel von Präfix und Suffix Operationen oft eine Ambiguität mit sich, welche Klammersetzung erfordert.

Verwirrend ist auch die Nutzung der gleichen Symbole für unterschiedliche Arten von Operationen, besonders bei: `&`, `*`, `+`, `-`. Diese werden zum einen für unäre Präfix-, sowie binäre-Operationen verwendet, als auch in doppelter Erwähnung als Präfix, Suffix und binäre Operation.

```
*a[i];              // Ausgeführt wird *(a[i]) denn [] bindet stärker als *
(*a)[i];            // Sollte man a zuerst dereferenzieren wollen so muss man dies hier schreiben
(*a)->draw();       // Auch bei mehrfacher Dereferenzierung muss geklammert werden

&i & & j && &k;     // Keine Verletzung der Sprache! Funktioniert z.B. nicht mit int…
++i++ + ++j++;      // …aber kompiliert wenn i, j und k von einem präpariertem Typ sind
*i**j;              // In diesem Beispiel sind nicht mal Leerzeichen nötig, Ausführung: (*i)*(*j);

if( !(*MySubsystemManager::GetInstance()->GetPtrToVectorOfPtrs(withParams))[5]->IsGood() ) {…}
//  8542222222222222222222222222222222233333333333333333333333311111111113566677777777777
```

Die Ziffern in der untersten Zeile sagen aus, in welcher Reihenfolge die Teile dieser Expression ausgewertet werden. 1 ist zuerst, dann aufsteigend bis 8. Fett gedruckt sind die Stellen markiert, welche einen Unterschied von mehr als Eins zu ihrem Nachbarn haben, hier *bricht* der Lesefluss oder besser, er *sollte*

brechen. Wenn er nicht bricht, dann heißt das, dass man wichtige Informationen übersieht und an der falschen Stelle weiterliest. Im obigem Beispiel sind fünf dieser Stellen.

(Nicht) Lösung

Lösungsmöglichkeiten in Myll:

```
*a[i];              // Gleiches Verhalten wie bisher
a->[i];             // Wäre eine mögliche Alternative zu (*a)[i], aber nicht wirklich schön
(*a)->draw();       // Dies bleibt so: Bei mehrfach Dereferenzierung muss geklammert werden
a->>draw();         // Alternativ wäre auch dies hier möglich, hätte zu Verwirrung führen können
a->->draw();        // Alternativ wäre auch dies hier möglich, hätte zu Verwirrung führen können
```

Wären alle unären Operationen Suffix-Operationen dann könnte das Beispiel so aussehen

```
if( MySubsystemManager::GetInstance()->GetPtrToVectorOfPtrs(withParams)*[5]->IsGood()! ) {…}
//  22222222222222222222222222222222233333333333333333333333311111111134555666666666667
```

In diesem Beispiel sind nur noch zwei Stellen, welche den Lesefluss unterbrechen.

Eine generelle Umstellung, dass alle Präfix-Operationen zu Suffix-Operationen werden, geht aber leider nicht, weil z.B. ein Suffix-`*` mit dem Multiplikations-Operator kollidiert (Okay, das macht das Präfix-`*` auch, aber nicht in Kombination mit anderen Operationen), Suffix-`++` schon belegt ist und `-e` für den Vorzeichenwechsel einfach besser aussieht als `e-`.

Es gibt in C++ schon die Möglichkeit z.B. `e.operator!()` als Suffix aufzurufen, was aber nicht auf primitiven Typen funktioniert und außerdem so schreibintensiv ist, dass es entweder deshalb nicht genutzt wird oder schon durch seine Präsenz allein den Lesefluss beeinträchtigt. In der gleichen Art könnte man zusätzlich zur normalen Schreibweise z.B. `e.not`, `e.neg`, `e.addr`, `e.preinc` als Pseudomember auf die relevanten Typen einführen, was jedoch eine Explosion an (kontextuellen) Keywords mit sich bringen würde.

Die Lösung für Myll ist, *dass es aktuell keine zufriedenstellende Lösung gibt.*

3.3.6.3 Zuweisungen und Throw Expressions

Ein altbekanntes Problem ist die fälschliche Zuweisung in einem *if* Statement. In den meisten Fällen war ein Vergleich gewünscht gewesen. Ein ähnliches Grundproblem existiert bei der *throw* Expression, sie kann sich sehr intransparent in verschachtelten Expressions verhalten. Im folgenden Beispiel wird die Exception nur geworfen wenn isSomething *false* ist.

```
if( a = 1 ) { … }           // Wird immer Aufgerufen und a wird Verändert
cout << a[c = 1];           // Gewünschte Zuweisung
const int i = isSomething ? 42 : throw new exception(); // der throw Fall weist i nichts zu
```

Lösung

Zuweisungen und *throw* sind keine Expressions mehr. Dadurch sind die genannten Beispiele nicht mehr valider Code und müssen in einfacher durchschaubareren Code umgeschrieben werden. In den seltenen Fällen, wo eine Zuweisung eine Expression war, muss sie nun einzeln als Statement geschrieben werden.

```
if( a = 1 ) { … }           // Error, eliminiert viele fälschliche Zuweisungen
c = 1;                      // Separate Zuweisung, macht diese auch sichtbarer
cout << a[c];
if( !isSomething )          // Dieser Code ist äquivalent zur obigen Version & leichter durchschaubar
    throw new exception();
const int i = 42;
```

In C# war bis vor Kurzem *throw* ein Statement, ist aber seit Version 7.0 auch als Expression nutzbar. Meiner Meinung nach war es eine gute Idee *throw*, genau wie *return*, *break* und *continue,* als Statement umgesetzt zu haben.

3.3.7 Schwer durchschaubare automatische Verhaltensweisen

Es gibt einige Stellen, an denen man nur eine kleine Änderung am Code macht und sich dann automatisch noch eine ganze Menge mehr ändert, von dem man direkt nichts mitbekommt.

3.3.7.1 Special Member Functions

Es gibt sechs spezielle, an eine Klasse gebundene, Funktionen (und Operatoren) in C++ und diese werden unter jeweils unterschiedlichen Bedingungen automatisch generiert.

Siehe **[SPEC-MEM]**:

- Default-Constructor
- Destructor
- Copy-Constructor
- Copy-Assignment-Operator
- Move-Constructor
- Move-Assignment-Operator

Problematisch ist dies in vielerlei Hinsicht. Zuallererst muss man diese Funktionen überhaupt erkennen, um sie mit der notwendigen Sorgfalt zu behandeln, denn es kann abgesehen von diesen speziellen Typen, beliebig viele normale Konstruktoren und Zuweisungsoperatoren geben. Drei von den hier aufgelisteten sind Konstruktoren (Constructor), zwei davon erwarten einen einzelnen Parameter mit besonderem Typ. Die speziellen Zuweisungsoperatoren (Assignment-Operator) erwarten die gleichen Parametertypen.

Sobald man eine dieser Funktionen selbst implementiert, fallen automatisch einige der anderen weg. Dies ist schwer durchschaubar und wirft im Normalfall noch zusätzliche Probleme auf, sodass die Verhaltensregeln *Rule-of-Three*, *Rule-of-Five* und *Rule-of-Zero* erdacht wurden. Diese geben vor, das wenn man eine der speziellen Funktionen implementiert (mit Ausnahme des *Default Konstruktors*), dass man in dem Zuge auch einige der anderen Implementieren soll. Anders der Fall des Rule-of-Zero, welcher vorgibt dass man komplett auf die Implementierung der speziellen Funktionen verzichten sollte, wiederum mit Ausnahme des *Default Constructors*. Siehe auch [JM-SM].

Diese Regeln existieren auf dem Papier, aber leider nicht in der Sprache, somit ist die Validierung von deren Befolgung auch komplett an den Nutzer übertragen.

Zusammenfassung

- Es ist schwer zu erkennen, ob und welche Probleme vorhanden sind
- Der Compiler validiert nichts davon

Geplante Lösung

Myll kann helfen, diese selbst auferlegten Regeln zu validieren. Das Attribut kann entweder ohne Parameter angegeben werden und würde so alle drei Regeln erlauben oder man gibt die Regeln an, welche man einhalten will.

```
[rule_of_n=5]              // Error: Es fehlt dtor und [move]operator=
class Test {
    [move] ctor(o) {…}     // Der Leser sieht direkt, dass es sich um einen Move Konstruktor handelt
    [copy] ctor(o) {…}     // Der Leser sieht direkt, dass es sich um einen Copy Konstruktor handelt
    [copy] operator=(o) {…}// Der Leser sieht direkt, dass es sich um ein Copy Assignment handelt
}
```

3.3.7.2 Zweierlei Enums

Es gibt zwei verschiedene Arten von Enums in C++. Das Erste ist dass, welches schon seit C-Zeiten vorhanden ist. Es hat den Nachteil, das seine Member nicht *nur* unter dem Namen des Enums erreichbar sind, aus diesem Grund werden im Normalfall dessen Member mit dem Enum-Namen präfigiert.

Die von C++ eingeführte Variante der Enums, mit dem Zusatz *class,* beseitigt das Problem der ungewollten Veröffentlichung aller Member in den aktuellen Namensraum. Sie heißen *Scoped-Enumeration*. Es wird allerdings auch die automatische Konvertierung in das zugrundeliegende Integer abgeschaltet. Das ist grundsätzlich etwas Gutes, hat aber auch negative Begleiterscheinungen: Zum Beispiel funktionieren bitweise Operatoren erst nach expliziter Implementierung für den speziellen Enum Typ.

```
enum A { X = 1, Y = 2, Z = 4 };
enum B { X, Y, Z };               // die Membernamen X, Y, Z, kollidieren mit denen von A
enum C { C_X, C_Y, C_Z };         // Präfigierung mit dem Namen des Enums
A a = A::X | A::Y;                // Bitweise Operatoren funktionieren auf diesen Enums

enum class D { X = 1, Y = 2, Z = 4 };
enum class E { X, Y, Z };         // kollidiert nicht mit den Membern von D
D d = D::X | D::Y;                // funktioniert nicht so einfach, erst nach Definition von:
D operator|(D lhs, D rhs) {
    return (D)((int)lhs|(int)rhs); // normalerweise mit static_cast und underlying_type
}                                 // dies muss für alle bitweisen Operatoren wiederholt werden
```

Lösung

In Myll sind alle Enums *scoped*. Sollte man auf sie bitweise Operatoren anwenden wollen, zeichnet man einfach das Enum mit `[operators(bitwise)]` aus und man bekommt die Standardimplementierung dieser. Sollte man exponentielle Durchnummerierung der Elemente wünschen, lässt sich dies mittels `[flags]` einschalten. Enums in C# haben ein `[Flags]` Attribut, welches leider nur die textuelle Ausgabe via ToString() verbessert, bitweise Operatoren hat in C# jedes Enum.

```
[operators(bitwise), flags]
enum A { X, Y, Z };               // automatisch X = 1, Y = 2, Z = 4
```

3.3.7.3 Überladung kombiniert mit Vererbung

In C++ findet keine Überladung von Methoden zwischen der abgeleiteten Klasse und der Basisklasse statt, siehe [ISO-OD]. Die Implementierung einer Methode in der abgeleiteten Klasse versteckt (*Shadowing* und *Hiding* genannt) alle Methoden mit gleichem Namen der Basisklassen. Dass dies so geschieht, ist nicht leicht zu durchschauen, besonders für einen Anfänger der Sprache sieht es aus, als wenn *public*-Funktionalität der Basis einfach so verschwindet.

Dieses Beispiel zeigt das Problem:

```
class Base {
  public:
    void f(int) {…};       // virtual würde die Situation nicht verbessern
                           // Base besitzt noch nicht die Mittel um void f(double) zu implementieren
}
class Derived : public Base {
  public:
    // using Base::f;      // diese Zeile würde das Problem beheben
    void f(double) {…};
}
int main() {
    Derived d;
    d.f(1337);             // Ruft Derived::f(double) auf
                           // … und nicht wie vielleicht erwartet Base:f(int)
}
```

Geplante Lösung

Inversion des Verhaltens, erwarte vom Nutzer, dass er explizit erwähnt, dass er *Shadowing* will.

```
class Derived : Base {
  public:
    func f(double) -> void {…};    // using Base::f; wird automatisch im C++-Code hinzugefügt
}
class OtherDerived : Base {
    [shadow]                       // in C# bekannt als hiding mit dem irritierenden Keyword new
    func f(double) -> void {…};    // using Base::f; wird nicht hinzugefügt
}
func main() -> int {
    Derived d;
    d.f(1337);                     // Ruft Base::f(double) auf
    OtherDerived d2;
    d2.f(1338);                    // Ruft wie explizit angefordert OtherDerived::f(double) auf
}
```

3.3.8 Schlechte & Schreibintensive Namensgebungen

C++ hat viele schlechte Schreibweisen und Namensgebungen von C geerbt. Manche hat aber auch C++ selbst erfunden und die eigentlich gut lesbaren von C verdrängt.

3.3.8.1 Datentypen

Es gibt die sehr schreibintensive Variante, um eine *vorzeichenlose sehr sehr große Ganzzahl* zu definieren mittels *unsigned long long int,* das ist fast soviel zu schreiben, wie es im Deutschen hier geschrieben steht.

Und es ist nicht einmal portabel verlässlich, wie viel Bit an Länge man von diesem Typen erwarten kann. Ein *short* hat üblicherweise 16 Bit, dürfte aber auch 32 haben. Ein *int* darf laut C++ Standard auch 16 Bit haben, hat üblicherweise 32 und es ist auch nicht verboten, dass es 48 oder 64 hat. Besonders prekär ist der Typ *long,* denn er verhält sich auf aktuellen 64 Bit Betriebssystemen unterschiedlich, auf Windows nämlich als 32 Bit große Zahl, konträr dazu auf Linux, MacOS und UNIX mit 64 Bit.

Auf diesen Freiheiten möchte man keinen Serialisierer schreiben, welcher Daten in Binärform speichert. Deshalb gibt es im Header *cstdint* die Definition der Typen: `std::int8_t`, `std::int16_t`, `std::int32_t` und `std::int64_t` und die identischen *unsigned* Varianten mit dem Präfix `u`. Diese Datentypen sind optional, es heißt, man kann sich nicht darauf verlassen, dass sie existieren. Außerdem verleiten sie Einsteiger dazu `using namespace std;` in ihren Programmen zu verwenden oder sie nutzen weiter die variablen Typen.

Lösung

Myll führt durchschaubare und aufgeräumte Namen für eingebaute Typen ein. Die meisten besitzen eine feste Bitbreite und die wenigen, welche eine dynamische Größe haben, sind streng limitiert in ihrer möglichen Bitbreite.

Vorzeichenbehaftete Ganzzahlen: `i8`, `i16`, `i32`, `i64`, `int`, `isize`

Vorzeichenlose Ganzzahlen: `u8`, `u16`, `u32`, `u64`, `uint`, `usize`

Bit-Sammlungen ohne Ganzzahl-Arithmetik: `b8`, `b16`, `b32`, `b64`, (`b128`,) (`b256`,) `bool`, `byte`

Gleitkommazahlen: (`f16`,) `f32`, `f64`, `f128`, `float`

Weitere Typen: `void`, `auto`, `char`, `codepoint`(, `string`)

Alle Typen, welche Zahlen enthalten, haben genau die angegebene Bitbreite. Die Typen `int` und `uint` haben mindestens 32 Bit, nutzen aber möglicherweise mehr. Die Typen `isize` und `usize` entsprechen großteils `ptrdiff_t` und `size_t` und haben mindestens 32 Bit, aber immer genug Platz um einen normalen Pointer zu speichern. Der Typ `f16` ist optional (half precision floating point number) und `float` ist mindestens 32 Bit breit, möglicherweise aber auch mehr.

Die Bit-Sammlungen sind nur als Speicher für Bits gedacht und erlauben nur logische / bitweise Operationen, sie gelten nicht als Zahlen und haben demnach auch kein Vorzeichen.

Der Typ `codepoint` hat die Größe, um einen Unicode Codepoint zu enthalten, üblicherweise 32 Bit.

Sollte man Daten serialisieren wollen, etwa um sie abzuspeichern oder über das Netzwerk zu verschicken, wird dazu geraten dies nur mit den Datentypen mit fester Breite umzusetzen. Es wäre sogar möglich, die Nutzung der variablen Typen in Strukturdefinitionen, welche der Serialisierung dienen, zu verbieten.

3.3.8.2 Funktionale Programmierung

Die Sprache C++ zählt funktionale Programmierung zu seinen verfügbaren Programmierparadigmen. Schauen wir uns ein relativ simples Beispiel an und vergleichen dies mit der nicht-funktionalen Implementierung, als auch mit der Lösung, die andere Programmiersprachen bieten.

Als *Map- & Reduce*-Beispiel soll diese einfache Berechnung dienen. Eine Liste von Zahlen soll zuerst durch zwei dividiert und dann das Produkt gebildet werden. Die drei Zahlen im Container (4, 6, 14) werden zunächst in *Map* jeweils durch 2 dividiert, dann wird in *Reduce* das Produkt gebildet und ergibt die Zahl 42.
Zuerst die **C++** Variante, aus Platzgründen mit *using namespace std*:

```
using namespace std;

// erste Variante, 144 Zeichen ohne signifikanten Leerraum und ohne nicht notwendiges const
vector<int> c = {4, 6, 14};
transform(begin(c), end(c), begin(c), [](auto d){ return d / 2; });
const int fortyTwo = accumulate(begin(c), end(c), 1, multiplies<int>());

// zweite Variante, 114 Zeichen ohne signifikanten Leerraum und ohne nicht notwendiges const
vector<int> c = {4, 6, 14};
const int fortyTwo = transform_reduce(
                          begin(c), end(c), 1, multiplies<int>(), [](auto d){ return d / 2; });
```

Die erste Variante verändert den Ursprungs-Container, was möglicherweise unerwünscht ist.
Die zweite Variante schreibt irreführenderweise die *Reduktion* vor der *Transformation* und verschmelzt die Operationen, welche eine einfache Komposition in komplexeren Fällen erschwert.

In **C#** würde es äquivalent so aussehen, ohne Nebeneffekte:

```
// 93 Zeichen ohne signifikanten Leerraum
List<int> c = new List<int>{4, 6, 14};
int fourtyTwo = c.Select( d => d / 2 ).Aggregate( (accu, e) => accu * e );
```

Das gleiche *Map & Reduce* Beispiel in **Ruby**, auch ohne Nebeneffekte:

```
# erste Variante, 56 Zeichen ohne signifikanten Leerraum
c = [4, 6, 14]
fortyTwo = c.map{ |d| d / 2 }.reduce{ |accu, e| accu * e }

# zweite Variante, 46 Zeichen ohne signifikanten Leerraum
c = [4, 6, 14]
fortyTwo = c.map{ |d| d / 2 }.reduce( :* )
```

Trotz der Nutzung von *using namespace std* im Beispiel, benötigt C++, verglichen mit der Lösung in Ruby, doppelt bis drei mal soviel Zeichen. Der Vergleich, mit einer dynamisch Typisierten Skriptsprache, mag hinken, aber C++ könnte so kompakt sein wie die Lösung in C#, wenn nur die Lambda-Syntax und die Boilerplate für die Iteratoren nicht wären.

Wie sieht es mit der nicht-funktionalen Implementierung in C++ aus?

Die folgenden nicht-funktionalen Implementierungen in C++ sehen viel lesbarer aus, viel eher, wie die funktionalen Implementierungen in C# und Ruby. Da stellt sich natürlich die Frage: Wieso schaffen es die Sprachen, welche schon viel länger Lambdas und funktionale Bibliotheken haben als C++, dennoch eine besser lesbare funktionale Implementierung zu haben?

```
// erste Variante
vector<int> c = {4, 6, 14};
int accu = 1;
for (auto& d : c) {    d /= 2; }
for (auto  e : c) { accu *= e; }
const int fortyTwo = accu;

// zweite Variante (diese Variante cheatet (betrügt) ein wenig, weil sie beide Schritte verschmilzt)
vector<int> c = {4, 6, 14};
int fortyTwo = 1;
for (auto d : c) { fortyTwo *= d / 2; }
```

Anmerkung zu den C++-Beispielen: Die funktionalen Implementierungen haben Vorteile gegenüber den nicht funktionalen Versionen. Zum einen kann zur potentiellen Steigerung der Performance `transform`, `reduce` und `transform_reduce` *parallel* sowie *nicht-sequenziell* ausgeführt werden. Zum anderen wird entweder *accu* nach außen hin sichtbar oder es wird mehrfach in *fortyTwo* geschrieben und es kann deshalb nicht als *const* markiert werden.

Geplante Lösung

Ruby hat die Methoden *map* und *reduce* auf seinen Containern *Array* und *Enumerable* definiert. Das Äquivalent dazu wäre, die C++-Container um diese zu erweitern, was das Ziel von Myll verfehlen würde (keine Neuimplementierung der C++-Klassen).

C# geht einen anderen Weg; es erlaubt statische Methoden durch *Syntactic Sugar* an Objekte zu binden. Diese Funktionalität wird Erweiterungsmethoden (Extension Methods) genannt. In einem Schritt des C# Compilers wird etwa der Aufruf von `c.Select( lambda )` in `Select( c, lambda )` übersetzt, dies passiert aber nur bei speziell markierten statischen Methoden.

Die Erweiterungsmethoden von C# beschreiben einen Subset dessen, was UFCS (Universal Function Call Syntax) erlaubt, nämlich bei jeder Funktion das erste Argument auch vor die Funktion zu ziehen und sie wie eine Methode aufzurufen. In UFCS wäre etwa auch `3.pow( 3 )` möglich, welches in C# nicht möglich ist, weil `pow()` nicht als Erweiterungsmethode markiert ist. In D und in Nim gibt es UFCS.

Myll könnte entweder UFCS einführen, welches sich extrem auf die komplette Sprache und deren Nutzung auswirken könnte, oder es könnte den gleichen Weg wie C# gehen und nur selektiv und vorsichtig einzelne sinnvolle Funktionen an Objekte binden. Der selektive Weg wird der erste Versuch sein und nach diesem Experiment kann besser entschieden werden, wie es weiter geht.

C++20 soll eine *Ranges* genannte Erweiterung in der Standardbibliothek erhalten, welche möglicherweise den aktuellen Missstand mit der funktionalen Programmierung behebt.

3.3.8.3 C++- und C-Style-Casts – Explizite Konvertierungen

C-Style-Casts haben eine leicht les- und schreibbare Form. Leider hat ihre assoziierte Semantik dazu geführt, dass sie immer weniger in C++ genutzt werden. Die originale Semantik von C umsetzend, bieten die C-Style-Casts in C++ die automatische Verhaltensweise, dass sie unterschiedliche `*_cast` Expressions kombinieren, um den Typen umzuwandeln.

Siehe Auszug aus *C++ Reference – Explicit type conversion*, siehe [EXPL-CAST]:

> *When the C-style cast expression is encountered, the compiler attempts to interpret it as the following cast expressions, in this order:*
>
> *a)* `const_cast<new_type>(expression);`
>
> *b)* `static_cast<new_type>(expression)`, *with extensions: pointer or reference to a derived class is additionally allowed to be cast to pointer or reference to unambiguous base class (and vice versa) even if the base class is inaccessible (that is, this cast ignores the private inheritance specifier). Same applies to casting pointer to member to pointer to member of unambiguous non-virtual base;*
>
> *c)* `static_cast` *(with extensions) followed by* `const_cast;`
>
> *d)* `reinterpret_cast<new_type>(expression);`
>
> *e)* `reinterpret_cast` *followed by* `const_cast`.
>
> *The first choice that satisfies the requirements of the respective cast operator is selected, even if it cannot be compiled...*

Aufgrund dieser ausgearteten Verhaltensweise verbannen sehr viele Projekte C-Style-Casts komplett oder erlauben sie lediglich für grundlegende Typen oder POD (Plain Old Data, salopp: alle Strukturen welche genau so auch in C vorkommen können). Man kann sagen, sie sind *quasi* ausgestorben.

Außerdem ist *std::move* im Grunde nur ein Cast in einen »*aus mir kann der Inhalt verschoben werden*« Typ.

Lösung

In Myll fällt die ursprüngliche Verhaltensweise des C-Style-Casts weg und `(Typ) val` ist **nur** noch ein *static_cast*, weil dieser am häufigsten verwendet wird. Der Aufruf mit `(? Typ) val` ist ein *dynamic_cast* und `(! Typ) val` ist je nach gegebenem *Typ* entweder ein *const_cast*, ein *reinterpret_cast* oder beides. Sollte man hier eine Unterscheidung wünschen, ist natürlich die schreibintensive Variante (wie auch für alle Casts) weiterhin verfügbar.

Analog zu den grade beschriebenen Casts, verhält sich auch *std::move.* Man kann mit dem Aufruf `(move) val` ein Objekt zu einem verschiebbaren Objekt *casten*.

3.3.8.4 Raw- & Smartpointer

Auch Rawpointer haben eine leicht les- und schreibbare Form. Leider hat auch ihre assoziierte Semantik dazu geführt, dass sie immer weniger in C++ genutzt werden.

Die Nutzung von **Rawpointern** wurde durch die Einführung von Smartpointern der Standardbibliothek, sowie von Boosts, als auch selbstgeschriebenen Intrusive-Smartpointern und Handles (siehe [AW-HDL]) weit zurückgedrängt. Dies hat die relative Nutzungshäufigkeit von Rawpointern zurückgehen lassen.

Rawpointer haben aber auch einen nicht zu unterschätzenden Vorteil, sie sind meiner Meinung nach wesentlich übersichtlicher zu lesen. Außerdem kommt, selbst in modernsten C++-Projekten, niemand drumherum Rawpointer verstehen zu müssen.

Ausnahmsweise sind hier nicht die Keywords **fett** gedruckt sondern die Zieldatentypen der Parameter, nur *const* ist hier *kursiv*, es wurde auch mehr Energie in eine lesbare Formatierung gelegt als üblicherweise.

```
void composeGUI( Widget *parentElement, unique_ptr<const ScrollableWidget> &scrollElement,
weak_ptr<ButtonWidget> &buttonChild, unique_ptr<Widget> anotherChild ) {…}

void composeGUI(
    const Widget                * parentElement,
    unique_ptr<ScrollableWidget> & scrollElement,
    weak_ptr<const ButtonWidget> & buttonChild,
    unique_ptr<Widget>             anotherChild
) {…}
```

Man kann jetzt meinen, dieses Beispiel ist *konstruiert*, na klar ist es das: und zwar um Probleme aufzuzeigen, welche in echten Projekten mit unübersichtlichen Parameterlisten auftauchen. Ich habe schon viel unübersichtlicheren Code gesehen (welcher produktiv/live/ausgeliefert ist) als dieses Beispiel hier. Mehr zur komplexen Situation der Parameterübernahme von Smartpointern kann in [HS-SPP] gelesen werden.

Verstecktes Verhalten, welches hier leicht zu übersehen ist: Der Parameter *scrollElement* wird als Referenz auf einen Unique-Pointer übergeben, das bedeutet, dass die aufgerufene Funktion den Besitz übernehmen kann, aber nicht muss. Der Besitz des anderen Unique-Pointers *anotherChild* wird auf jeden Fall an die Funktion übergeben. Dieser wichtige Unterschied geht schnell unter, weil der Text *unique_ptr* weit vom (nicht) vorhandenen `&` entfernt steht. Das *const* welches das ButtonWidget vor Veränderung schützt, ist ebenso leichter zu übersehen als beim Raw-Pointer auf Widget.

Unübersichtlich wird auch die sehr weite Entfernung des Zieldatentypen von der linken Seite.

Geplante Lösung

Smartpointer bekommen in Myll *Syntactic Sugar* verpasst. An den Positionen im Code, an denen Rawpointer auftreten können, kann ein *unique_ptr* sich als `*!` schreiben lassen, ein *shared_ptr* als `*+` und der *weak_ptr* als `*?`. Diese Symbole wurde aufgrund ihrer Ähnlichkeit zu *Reguar Explressions* gewählt. Ein *Plus* zeigt *Mehrfachnutzung* an, ein *Fragezeichen Optionalität*. Sollte man nicht die Smartpointer der Standardbibliothek nutzen wollen, sondern etwa die von Boost oder seine eigenen, lässt sich dies leicht umstellen.

Hier das Smartpointer Beispiel in Myll.

```
func composeGUI( Widget * parentElement, const ScrollableWidget *!& scrollElement, ButtonWidget *?&
buttonChild, Widget *! anotherChild ) -> void {…}

func composeGUI(
    const Widget       *   parentElement,
    ScrollableWidget   *!& scrollElement,
    const ButtonWidget *?& buttonChild,
    Widget             *!  anotherChild
) {…}
```

Hier sind die Indirektionsebenen direkt beieinander, nicht an unterschiedlichen Stellen der Deklaration. Feinheiten wie die *erzwungene* oder die *mögliche* Übergabe des Besitzes entgehen einem nicht mehr so leicht. Das Keyword *const,* welches sich auf das bezeigte Element / den Zieldatentyp bezieht, ist stets ganz links abzulesen.

3.3.9 Keyword Wiederverwertung

C++ hat eine inflationäre Nutzung von immer wieder den gleichen Keywords in unterschiedlichen Kontexten. Als Beispiele seien hier static und const betrachtet.

Das Keyword **static** kann entweder die Veröffentlichung eines Identifiers über Translation Unit Grenzen verstecken, eine Variable innerhalb einer Funktion persistent machen oder innerhalb einer Klasse Variablen und Funktionen schaffen die global sind, anstelle sich auf die Instanz zu beziehen. Was mich verwundert ist, dass dieses Keyword nicht genutzt wird, um *statische Bindung,* im Gegensatz zur *dynamischen Bindung (via virtual),* bei Klassen zu signalisieren, wobei ich *statisch* als erstes damit Verknüpfen würde.

Das Keyword **const** kann angeben, ob eine Variable nicht mehr als einmal beschrieben werden darf oder ob eine Methode den Zustand des Objekts nicht ändert. Also ist *const* in keinem der Fälle zu verwechseln mit mathematischen Konstanten, welche immer den gleichen Wert haben; das Keyword *constexpr* erfüllt eher diese Eigenschaft in C++.

Geplante Lösung

Die Attribute in Myll sind keine Keywords und kollidieren demnach nicht mit Identifiern und dem Rest der Sprache. Dadurch können hier viel sprechendere Bezeichnungen gewählt werden. Wer sich nicht an neue Keywords gewöhnen mag, wird trotzdem glücklich, denn die bisherigen funktionieren auch noch.

```
class Test {
    [global]                        // Hier kann auch static stehen, global  finde ich verständlicher
    var i32 notInstance = 1337;

    [pure]                          // Hier kann auch const  stehen, pure    finde ich verständlicher
    method man() {
        [persist]                   // Hier kann auch static stehen, persist finde ich verständlicher
        var f32 expensiveResult = expensiveCalc();
    }
}
```

3.3.10 East Const und West Const

Gangkriminalität zwischen verfeindeten Clans ist hier nicht gemeint. Es geht lediglich um die mögliche Positionierung des Keywords *const*. (Auch wenn es hier analoge Auseinandersetzungen, wie zwischen der *geschweifte Klammer in neuer Zeile* mit der *geschweifte Klammer auf der gleichen Zeile* Fraktionen, gibt)

```
const int * a;              // konstantes Objekt
int const * b;              // konstantes Objekt
int * const c;              // variables Objekt, konstanter Pointer
```

Die Variablendeklaration von *a* und *b* sind identisch, das *const* bezieht sich auf das *int*. Bei der Deklaration von *c* hingegen bezieht sich das *const* auf den Pointer. Das heißt, dass es an manchen Stellen möglich ist das *const* links von der zu attribuierenden Stelle zu schreiben (*west const*) und an anderen rechts (*east const*).

Auch wenn es nach meiner persönlichen Erfahrung wesentlich häufiger der Fall ist, *west const* anstelle von *east const* anzutreffen, gibt es in C++ nur eine Schreibweise die immer funktioniert, welches der seltener vorkommende Fall des *east const ist*.

Lösung

Myll setzt konsequent *west const* ein, *east const* ist nicht valide.

```
const int * a;              // konstantes Objekt
var   int const * b;        // variables Objekt, konstanter Pointer
```

3.3.11 Präprozessor

Der Präprozessor ist eine Schattenwelt [MS-NSW], welche teilweise ähnliche Funktionalität wie C++ selbst bereitstellt, nur leider mit einer komplett anderen Schreibweise, neuen Freiheiten sowie vielen Einschränkungen daher kommt.

Viel der Funktionalität, welche ursprünglich nur von Präprozessor ausgeführt werden konnte, kann C++ schon selbst, teils sogar besser, erfüllen. Hier ist immer zuerst die Variante aufgeführt, welche den Präprozessor benötigt und dann die moderne Alternative.

```
#define MAX(l,r) ((l)<(r)?(r):(l))          // Hat potentiell Nebeneffekte durch Mehrfachausführung

template <typename T>
T max(T l, T r) { return l < r ? r : l; }   // Keine Nebeneffekte und sogar typsicher
```

```
#define ARY_SIZE 707
int ary[ARY_SIZE];

constexpr int ary_size = 707;
int ary[ary_size];                          // Geht dank constexpr
```

```
#ifdef DEBUG
    log.print("only in debug mode");
#endif

if constexpr (DEBUG) {                      // Seit C++17, aber nicht außerhalb von Funktionskörpern
    log.print("only in debug mode");
}
```

Bei diesen Beispielen ist keine Alternative zu Macros vorhanden:

```
#ifdef DEBUG                                // Geht nicht mit if constexpr
#include <debug.h>
#endif
```

```
#if !__cpp_noexcept                         // Funktionalitätsprüfung mit leerem Polyfill
#define noexcept
#endif
```

Geplante Lösung

Zur Lösung des *Präprozessor if* nutzt Myll Attribute. Alles, was Myll nicht löst, kann in .cpp Dateien erledigt werden, denn der von Myll erzeugte Output ist ja kooperativ mit *normalem* C++-Code kompatibel.

```
[Debug]                             // Führt das attribuierte Statement nur im Debug Mode aus
log.print("only in debug mode");

[CT]                                // Erzwingt die Ausführung zur Compiletime, nicht nur bei if
if( someConstant ) …

                                    // Polyfills sollten nicht nötig sein, dafür ist Myll ja da
```

Sollte Myll erkennen, dass in einem *if* nur *constexpr* Anteile stehen, wird automatisch ein *if constexpr* erzeugt.

3.3.12 Goto

Das *goto* Statement und die Labels, zu denen mittels *goto* gesprungen werden kann, sind heutzutage für die meisten Nutzungen verpönt. Eine der wenigen Ausnahmen dessen, ist der Fall, dass man mehr als eine Schleife verlassen will.

Geplante Lösung

Myll bietet die Möglichkeit bei *break* und *continue* Statements eine Zahl mitzugeben, wie viele Schleifen verlassen oder fortgesetzt werden sollen. Hier ein Beispiel mit der neuen *loop* und *times* Schleife:

```
loop {                                  // #1. Endlose Schleife
    do 10 times i {                     // #2. Zählschleife von 0 bis 9
        if( something[i] ) continue;    // macht bei #2 weiter, identisch mit continue 1;
        if( someother[i] ) break;       // macht bei #3 weiter, identisch mit break 1;
        if( somewhat[i] )  continue 2;  // macht bei #1 weiter
        if( someelse[i] )  break 2;     // macht bei #4 weiter
    }
    cout << "10x fertig\n";             // #3
}
cout << "infinite loop fertig\n";       // #4
```

Ein *goto* gibt es in Myll nicht.

3.3.13 Generische Typeneinschränkung

Um in C++ Einschränkungen auf Template-Parametern umzusetzen, muss man zu einer besonderen Technik der Programmierung greifen, welche **SFINAE** genannt wird. Dies ist eine weitere komplexe Schattenwelt (siehe [MS-NSW]), welche man erst beherrschen muss, bevor man sie fehlerfrei nutzen kann. Will man beispielsweise seine Template-Funktion nur mit Parametern nutzbar machen, welche Klassen sind, muss man das Folgende schreiben (in **fett** ist hier der wichtigste Teil markiert)

```
template <typename T, typename enable_if<is_class<T>::value, int>::type = 0>
T somethingWithClasses() {…}
// ODER
#define REQUIRES(TRAIT) typename enable_if<TRAIT::value, int>::type = 0
template <typename T, REQUIRES(is_class<T>)>
T somethingWithClasses() {…}
```

enable_if ist eine Template-Klasse, die, nur im Falle, dass ihr erster Template-Parameter *wahr* ist, einen Member namens *type* erzeugt. Sollte man nun also, durch Instanziierung dieses Templates einen Datentypen übergeben, welcher von einem nicht akzeptierten Typen ist, würde die Anfrage von *type* einen Fehler werfen. Zumindest könnte man dies erwarten. SFINAE besagt aber, dass ein solches Einsetzen (Substitution) welches zu einem Fehler (Failure) führt, jedoch keinen Kompilationsfehler (Error) erzeugt. Es wird lediglich die weitere Betrachtung dieses Templates beendet und die Template-Funktion nicht für den verwendeten Typen in Betracht gezogen. Das `= 0` am Ende der Zeile ist dafür da, dass wir diesen Template-Parameter nicht übergeben müssen.

In C# ist die gleiche Aufgabe so zu lösen:

```
T somethingWithClasses<T>() where T : class {…}
```

Das liest sich als Sprachkonstrukt ganz einfach: »Ich habe eine generische Funktion bei welcher *T* eine *class* sein muss.« Es bedarf keiner Metaprogrammierung oder komplexer Substitutionsregeln. Das hier drückt am besten aus was ich mit "Wenn etwas Teil der Sprache ist und nicht nur der Bibliothek" meine. Es geht natürlich auch via Bibliothek, aber um den C++-Code weiter oben zu schreiben, muss man zusätzlich TMP und SFINAE verstanden haben und den wichtigen Teil (in **fett**) der enthaltenen Information erst in einem Wust von Boilerplate suchen.

Im Falle der Fehlparametrierung des Templateparameters sehen die Fehlermeldungen wie folgt aus:

C++ in Clang 10 (minimal bereinigt, sechs Zeilen):

```
error: no matching function for call to 'somethingWithClasses'
    somethingWithClasses<GIVEN>();
    ^~~~~~~~~~~~~~~~~~~~~~~~~~
note: candidate template ignored: requirement 'is_class<GIVEN>::value' was not satisfied [with T =
                                                                                              … GIVEN]
    T somethingWithClasses() {…}
      ^
```

C++ in GCC 10.1 (minimal bereinigt, zwölf Zeilen):

```
error: no matching function for call to 'somethingWithClasses<GIVEN>()'
    | somethingWithClasses<GIVEN>();
    |                              ^
note: candidate: 'template<class T, typename std::enable_if<std::is_class<_Tp>::value, int>::type
                                                          … <anonymous> > T somethingWithClasses()'
    | T somethingWithClasses() {…}
    |   ^~~~~~~~~~~~~~~~~~~
note:   template argument deduction/substitution failed:
error: no type named 'type' in 'struct std::enable_if<false, int>'
    | #define REQUIRES(TRAIT) typename enable_if<TRAIT::value, int>::type = 0
    |                                                                     ^
note: in expansion of macro 'REQUIRES'
    | template <typename T, REQUIRES(is_class<T>)>
```

C# in .NET Core 3.1 (minimal bereinigt, eine Zeile):

```
The type 'GIVEN' must be a reference type in order to use it as parameter 'T' in the generic type or
                                                                  … method 'somethingWithClasses<T>()'
```

Die Fehlermeldung der C# Variante besteht nur aus einer Zeile und hat dennoch alle nötigen Informationen, die enthaltene Zeilennummer (nicht sichtbar) zeigt auf die Stelle der Instanziierung. Die C++-Varianten haben als Hauptfehler, dass die Funktion nicht existiert und die Zeilennummern zeigen auf die Instanziierungsstelle. Als Notiz erwähnen sie mehr oder weniger geschwätzig, dass es Kandidaten gab, welche jedoch scheiterten. Diese Fehlermeldungen sind dennoch wesentlich kompakter als würde man den Typen nicht einschränken und einfach versuchen ihn zu verwenden. Dies endet üblicherweise mit Bildschirmseiten voll mit Fehlermeldungen, die einem in keiner Zeile eine verständliche Erklärung liefern.

Geplante Lösung

Für Myll ist folgende Lösung geplant:

```
func somethingWithClasses<T>() requires is_class<T> {…}
```

Das C++20 Concepts-Feature wird wohl das Keyword *requires* nutzen, deswegen wurde dies hier gewählt, *where* ist alternativ möglich. Solange wie es noch keine Compiler gibt welche Concepts unterstützen wird identischer C++-Code wie zuvor gezeigt generiert, später wird die Ausgabe auf Concepts umgestellt.

Sollte beides, die Deklaration sowie die Instanziierung, in Myll stattfinden, kann hier auch schon bei der Transpilation zu C++-Code ein Fehler ausgegeben werden.

3.3.14 Verkettete Nullpointer

Hat man Pointer auf Objekte, welche wiederum Pointer enthalten, muss man stets überprüfen ob zuerst der Ursprungs-Pointer *Null* ist und nur dann darf der enthaltene Pointer überprüft werden. Dies führt entweder zu einer *Pyramid-of-Doom* (immer mehr geschachtelte if-Statements) oder einer immer länger werdenden Dereferenzierungs-Schlange, welche in ihrer kompletten Länge sogar noch einmal wiederholt werden muss, für die effektive Operation die man ausführen will. Die Komplexität der Bedingung, bei der Schlange, wächst quadratisch mit der Zahl der dereferenzierten Pointer.

```
auto* ptr1 = …;
if(ptr1 != nullptr) {
    auto* ptr2 = ptr1->member;
    if(ptr2 != nullptr) {
        auto* ptr3 = ptr2->member;
        if(ptr3 != nullptr){
            ptr3->doSomething();
        }
    }
}
…
auto* ptr1 = …;
if(ptr1 != nullptr && ptr1->member != nullptr && ptr1->member->member != nullptr) {
    ptr1->member->member->doSomething();
}
```

Geplante Lösung

Der *Null-Conditional Operator* (`?.`) reduziert dieses Problem gewaltig. Er prüft was links von ihm steht auf *Null* und führt nur im *nicht Null* Fall aus, was rechts von ihm steht. Dies bezieht sich auch auf folgende verkettete Operationen.

```
auto* ptr1 = …;
ptr1?.member?.member?.doSomething();

// die hinzugefügten Klammern () zeigen die Reihenfolge des ersten Schritts der Ausführung
(ptr1)?.(member?.member?.doSomething());
```

Dieser Operator existiert so in C#.

4 Implementierung

4.1 Übersicht

Meine Arbeit mit TypeScript und ähnliche andere Projekte, welche eine *Source-to-Source*-Übersetzung (Transpilation) durchführen, brachten mich dazu, dieses Projekt mit dem gleichen Verfahren zu beginnen.

Durch Transpilation entgeht man vielen problematischen und arbeitsintensiven Aspekten, wie:

- Erzeugung von Assembly / Maschinencode
- Optimierung von Assembly / Maschinencode
- Erzeugung eines ABI (Application Binary Interface, Binäre Repräsentation)
- Linking von Object Files
- Bereitstellung eines Debuggers

Es bietet einem die Möglichkeit, eine Sprache zu bedienen, ohne sie schreiben zu müssen, sowie die gesamte Infrastruktur der Zielsprache nutzen zu können.

Auch andere aktuelle Entwicklungen nutzen Transpilation um schneller einsatzbereit zu werden, wie zum Beispiel Jonathan Blow's **Jai**, welches anfänglich zu C++-Code transpilierte und Per Vognsen's **Ion**, welches immer noch zu C-Code transpiliert. C++ selbst war in seinen Anfängen (**C with Classes**) als Transpiler zu C-Code umgesetzt.

Myll soll direkt mit C++-Code zusammenspielen, also einer Mixtur aus .cpp/.h und .myll Dateien. Es soll die C++-Standardbibliothek, sowie C- und C++-Bibliotheken, nutzen können. Die Umsetzung des Projekts erfolgt in **C#** und nutzt **Antlr4** als Lexer und Parser Generator, siehe [PHF-A4].

Die Entstehung und die Komponenten eines Compilers ist meist in diese Schritte unterteilt:

- Design der Sprache
- Lexikalische Analyse mittels Lexer
- Syntaktische Analyse mittels Parser
- Semantische Analyse
- Ausgabe des Kompilats (hier: Erzeugung der C++-Quellcode-Dateien)

4.1.1 C++ predigen, aber C# und Java trinken

Man sollte immer das beste Werkzeug für seine Arbeit nutzen. Aufgrund der Komplexität eine Sprache wie C++ zu lexen und zu parsen, bediene ich mich eines weitverbreiteten Parser-Generators und einer einfacher zu nutzenden Sprache. Die *Performance,* welche bei dieser Arbeit im Vordergrund steht, ist die Umsetzung von möglichst viel Funktionalität im gegebenen Zeitfenster. Dies lässt sich mit C# einfacher realisieren. ANTLR ist in Java geschrieben und erfordert lediglich für die Übersetzung der Grammatik ein installiertes JRE (Java Runtime Environment).

4.1.2 Alternative Umsetzungsmöglichkeiten

Im Kontext der Erstellung dieser Masterarbeit habe ich auch Alternativen zur Umsetzung betrachtet.

- Selbstgeschriebener Lexer/Parser

    - Nach einigen kleinen Prototypen war klar, dass eine Programmiersprache, welche annähernd die Komplexität von C++ aufweist, sich nicht in absehbarer Zeit manuell lexen & parsen lässt
    - Dieses Unterfangen wäre sicher interessant gewesen, ist hier aber nicht der gewünschte Fokus
- C++ mit Boost Spirit X3
    - Sehr schwierig, eine komplexe und rekursive Grammatik zu definieren
    - Lange Kompilationszeiten
    - Spirit X3 hätte angeblich eine gute Runtime Performance
      (keine verlässlichen Quellen, nur der Grundtenor vieler Diskussionen)
- C++ mit PEGTL
    - Sehr ähnliche Probleme wie mit Spirit X3
- C++ mit ANTLR
    - Grammatik relativ einfach zu definieren, nah an der Erweiterten Backus-Naur-Form
    - Unterstützung von ANTLR in C++ ist nicht so ausgereift wie die von C#
    - Siehe die aufgeführten Nachteile von C++ in diesem Dokument
    - Implementierungs-Geschwindigkeit wichtiger als Runtime-Performance
- C# mit ANTLR
    - Grammatik relativ einfach zu definieren, nah an der Erweiterten Backus-Naur-Form
    - ANTLR hat die beste Interoperabilität mit C# von den unterstützten Sprachen, welche ich beherrsche
    - Die Arbeit mit C# erlaubt einfache Nutzung einer graphischen Oberfläche und hat auch sonst eine schnelle Implementierungsgeschwindigkeit

4.2 Lexer / lexikalischer Scanner

Lexer werden verwendet um einen Eingabetext in logisch zusammenhängende Stücke zu zerteilen. Diese Stücke werden Tokens genannt. Dieser Schritt, welcher *lexikalische Analyse* genannt wird, dient der Vorverarbeitung der Eingabe und wird von nahezu jedem Compiler durchgeführt. Leerraum wird im Normalfall in diesem Schritt verworfen (außer bei Sprachen, in denen Leerraum Bedeutung hat, wie etwa Python). Oft werden Lexer durch ein Tool erzeugt. Diese nehmen ein Vokabular und erzeugen daraus Programmcode. Alternativ zur Generierung können sie aber durchaus auch von Hand erstellt werden.

Ein Beispiel:

```
var int test = 99;
```

Diese Eingabe wird in folgende mit `|` getrennte Tokens aufgeteilt, Klammern zeigen optionalen Inhalt:

```
VAR | INT | ID( test ) | ASSIGN | INTEGER_LIT( 99 ) | SEMI
```

Der für Myll implementierte Lexer hat nur wenige Besonderheiten. Zu seiner Erstellung wurde der Lexer-Generator von ANTLR 4 zu Hilfe genommen wurde, hierzu musste ein Vokabular definiert werden.

Die üblichen Spezialfälle, wie *String* und *Character-Literale* sind so definiert, dass sie an ihrem einleitenden Zeichen anfangen (»`"`« und »`'`«). Von da an werden alle folgenden Zeichen und die meisten Escape-Sequenzen zum Literal gezählt, bis die Verarbeitung auf ein schließendes Zeichen stößt, welches identisch mit dem respektiven einleitenden Zeichen ist. Das Character-Literal muss genau ein Zeichen enthalten.

Die Escape-Sequenzen für oktale-, hexadezimale- und Unicode-Zeichen werden bislang nicht unterstützt. Alternativ dazu ist aber die direkte Eingabe von UTF-8 Zeichen im Texteditor möglich.

Auch *Kommentare* sind in Myll möglich und erlauben die aus C++ bekannten Varianten, *bis zum Ende der Zeile* via `// …` und *über mehrere Zeilen hinweg* via `/* … */`. Sie schreiben alle Eingaben in ihrem Geltungsbereich in einen speziellen Kanal, welcher abseits der zu verarbeitenden Code-Tokens liegt. Aktuell wird dieser Kanal noch nicht weiter genutzt, d.h. die Kommentare werden nicht in den erzeugten übernommen. Es ist aber durchaus möglich, dies zu einem späteren Zeitpunkt nachzurüsten.

Die Sprachen C sowie C++ leiden unter einem Entscheidungsproblem, welches ich in 3.3.2.1 schon aufgeführt habe. Die Problematik ist, dass hier Identifier (Bezeichner) schon während der lexikalischen Analyse in *Typen Identifier* und *Variablen Identifier* klassifiziert werden müssen. Zur Lösung dieser Problematik wird ein sogenannter *Lexer-Hack* verwendet, um die Tokens richtig zu klassifizieren, siehe [LEX-HACK]. Darin wird dem Lexer vom Parser ein Rückwärtskanal, in Form einer Symboltabelle, bereitgestellt, anhand derer eingesehen werden kann, ob ein Identifier nun eine Variable oder einen Typen darstellt.
Clang verzichtet auf diesen *Hack* und klassifiziert zunächst alle Identifier in eine gemeinsame Kategorie und entscheidet weit später, während der semantischen Analyse, die richtige Klassifikation.
Myll braucht diesen *Hack* ebenfalls nicht und unterscheidet Identifier auch nicht. Es ist durch die eindeutigere Syntax aber schon im Parser bekannt, ob es sich um einen Typ oder eine Variable handelt.

Der für Myll implementierte Lexer betrachtet aber auch weiterhin die Tokens `>>` nicht gemeinsam, sondern als jeweils zwei separate `>` Tokens, so wie C++. Dies ist notwendig da sie je nach Auftrittsort entweder die logische bitweise Verschiebung nach rechts oder das Schließen zweier Templates gemeint sind. Dies ist außerdem nun auch für den neuen Potenzierungs-Operator `**` nötig, da es in einem anderen Kontext auch zwei Dereferenzierungs-Aufrufe sein können.

```
vector< vector< int >>     // Zwei separate Templates werden geschlossen
                84 >> 1    // Logische Rechts-Verschiebung
8 ** 2                     // 8 hoch 2
  **a                      // a wird zwei mal dereferenziert
```

Nach dem Schritt der lexikalischen Analyse ist bekannt, ob der eingegebene Text Anomalien aufweist. Keine Anomalien heißt aber noch lange nicht, dass ein Text Sinn macht, also ein syntaktisch korrektes Programm zeigt.

```
if if int ()));    // OK vom Lexer: Keine Anomalie
€                  // Lexer meldet Fehler: € ist weder Keyword, Operator oder Identifier
"                  // Lexer meldet Fehler: Das Anführungszeichen wird nicht geschlossen
```

Die komplette Auflistung des Vokabulars des Lexers ist im Anhang 7.7.1 einzusehen.

4.3 Parser / syntaktische Analyse

Parser nehmen den vom Lexer erzeugten Token-Strom und erzeugen daraus entweder einen AST (Abstract-Syntax-Tree, einen Syntax Baum) oder einen Parse-Tree (auch CST, Concrete-Syntax-Tree).
Mit dem linearen Strom an Tokens, welcher der Lexer erzeugt hat, durchläuft der Parser in der syntaktischen Analyse meist einen Entscheidungsbaum, in jenem er versucht, alle Tokens zu passenden Regeln zu zuordnen. Die definierten Regeln sind mittels *Und-* und *Oder-*Gliedern verkettet und der gesamte Regelsatz wird *Grammatik* genannt. *Und*-Verknüpfungen deuten an, dass mehrere Tokens, in Reihe, übereinstimmen müssen. *Oder* Verknüpfungen bieten alternative Pfade an, trifft eine Regel nicht zu, wird die nächste versucht. Jede Regel, die mit dem aktuell betrachteten Token übereinstimmt, wird in die Tiefe des Baums gefolgt. Sollte eine Regel in keinem Pfad zu einem Token passen, wird im Baum wieder nach oben

gesprungen und von dort aus Alternativen durchlaufen. Dabei werden alle erfolgreich übereinstimmenden Regeln aufgezeichnet. Sollten bis zum Ende der syntaktischen Analyse alle Tokens verbraucht worden und der Regel-Baum wieder am Wurzelknoten sein, so bilden diese aufgezeichneten Knoten den AST oder CST des eingegebenen Textes und bestätigen, dass der eingegebene Text gültig ist. Wie Lexer, werden auch Parser, oft von einem Tool erzeugt. Deren händische Erstellung ist ebenso möglich aber, je nach Komplexität der gewünschten Grammatik, ein sehr ambitioniertes Unterfangen.

Der für Myll implementierte Parser wurde mittels des ANTLR 4 Parser Generators erstellt. In jenem kann eine kontextfreie Grammatik in Erweiterter Backus-Naur-Form (EBNF) formuliert werden. Der Parser ist ein LL(*)-Parser welcher den Token-Strom Top-Down parsed, der Stern deutet an das der Lookahead (das Vorausschauen) nicht begrenzt ist. ANTLR erzeugt seit Version 4 keinen AST mehr, sondern einen CST.

Anhand der folgenden, stark reduzierten Grammatik werden die beispielhaft erzeugten Tokens aus dem Lexer-Beispiel syntaktisch analysiert. Tokens werden hier komplett groß geschrieben, Regeln starten mit einem Kleinbuchstaben.

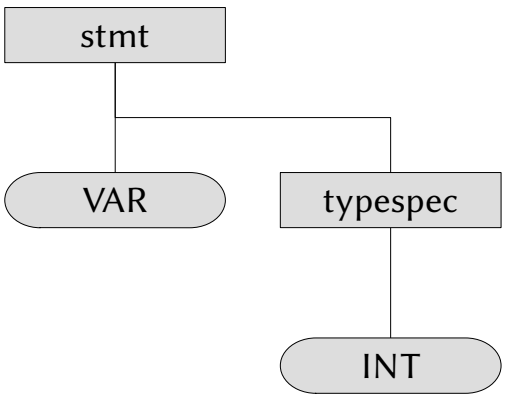
```
stmt        :  (    RETURN  expr            SEMI
               |    VAR     typespec idVars SEMI    );
typespec    :  (    BOOL
               |    FLOAT
               |    INT     );
idVars      :       idVar   (COMMA  idVar)*;
idVar       :       ID      (ASSIGN expr )?;
expr        :  (    expr    PLUS    expr
               |    ID
               |    INTEGER_LIT     );
```

Der Token-Strom des Lexer-Beispiels sieht so aus:

```
VAR | INT | ID( test ) | ASSIGN | INTEGER_LIT( 99 ) | SEMI
```

Der Einstiegspunkt in der Grammatik sei für dieses Beispiel die Regel *stmt* und das aktuelle Token ist VAR. Die Regel *stmt* wird betrachtet und enthält als erste Alternative RETURN (es ist eine Alternative, weil die beiden Regeln mit einem *Oder* getrennt sind). Dies stimmt nicht mit dem aktuellen Token überein, also wird jetzt die Alternative betrachtet. Diese enthält VAR als erstes Element, was mit dem aktuellen Token übereinstimmt, das heißt, der Token-Strom springt zum nächsten Token. Zwischen VAR und *typespec* steht kein Operator, deswegen ist es eine implizite *Und*-Verknüpfung. *typespec* ist eine weitere Regel und kein Token, deswegen wird sie zur weiteren Überprüfung betreten. In *typespec* wird jetzt geschaut, ob ein Token mit INT übereinstimmt (natürlich der Reihe nach), und ja, ist es: Strom einen weiter. Da *typespec* keine weiteren Regeln mehr enthält gilt es als erfolgreich erfüllt.

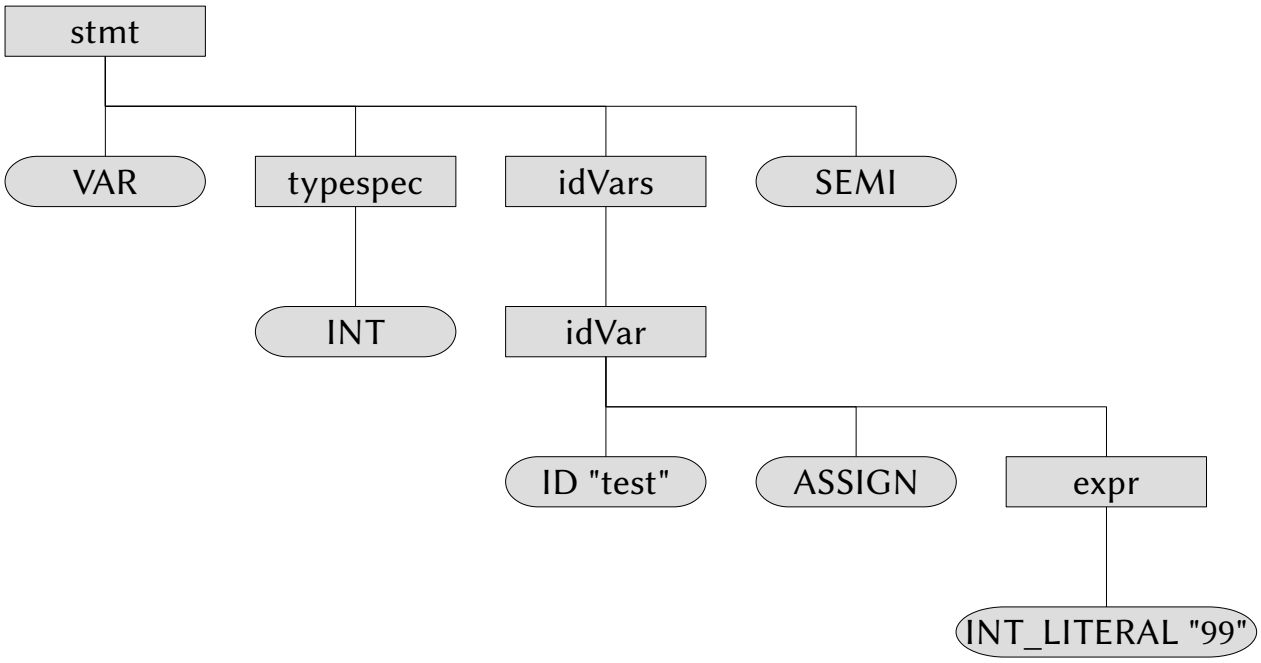
Illustration 4.1: Der Parse Tree / CST sieht aktuell so aus



Zurück im Ursprung in *stmt* steht *idVars* als nächste Regel an, diese wird betreten und es findet sich eine weitere Regel *idVar*, auch dort wird hineingesprungen. ID stimmt mit der ID aus dem Strom überein, der textuelle Inhalt des Token ist hier egal, die Regel ist erfüllt, Strom++. Dann kommt ein optionaler Anteil (zu erkennen am Fragezeichen als Suffix) welcher erfüllt sein kann, aber nicht muss. Es wird überprüft, ob der nächste Token mit ASSIGN übereinstimmt, was er macht, Strom++. Der optionale Bereich ist aber noch nicht vorbei, d.h. die Regel *expr* wird betreten, welche wiederum Alternativen enthält. Die erste Regel enthält wieder *expr* als erste Regel. Aktuell sieht es anscheinend so aus, als würde dies in einem ewigen Abstieg durch immer wieder der ersten (linken) Regel machen. Glücklicherweise hat ANTLR 4 genau für diese Problematik eine Sonderbehandlung, welche die direkte Rekursion der Regel ganz links erlaubt [PHF-A4], welche alternativ bei vielen anderen Parser-Generatoren jetzt dazu geführt hätte, dass man seine Grammatik auf komplett andere Weise neu implementieren müsste, möglicherweise mittels *Precedence Climbing* [P-CLIMB].

Die erwähnte Sonderbehandlung führte dazu, dass die erste Regel temporär übersprungen wurde und nun INTEGER_LIT mit ID verglichen wird, was nicht übereinstimmt. Die nächste Alternative ist INTEGER_LIT, ein Treffer, also Strom++ und damit ist die optionale `(ASSIGN expr)?` Regel erfüllt, als auch idVar beendet. Jetzt steht eine optionale, als auch wiederholbare, Regel an (Erkennbar am Asterisk als Suffix), welche mit einem COMMA beginnen soll; hier könnten weitere Variablen vom gleichen Typ definiert werden. Diese Regel ist nicht erfüllt und *idVars* ist damit abgeschlossen. In *stmt* wird verlangt, dass noch ein SEMI folgt. Wäre dies nicht der Fall, wäre alle Arbeit verworfen worden und der Text als ungültig bewertet worden.

Illustration 4.2: Finaler Parse Tree / CST



Nach dem Schritt der syntaktischen Analyse ist bekannt, ob der eingegebene Text grammatisch valide ist, was noch nicht heißt, dass dessen Code so auch ausgeführt werden kann.

Die vollständige Auflistung der Grammatik des Parsers ist im Anhang 7.7.2 einzusehen.

4.4 Sema / semantische Analyse

Die semantischen Analyse betrachtet die Bedeutung des Textes. Clang nennt die für die semantische Analyse genutzte Softwarekomponente Sema, dieser Namensgebung möchte ich mich hier anschließen.

Für die semantische Analyse gibt es keine Generatoren, zumindest nicht von ANTLR.

Bevor die semantische Analyse durchgeführt wird, seien der *Details zum AST* und *Module* erklärt.

4.4.1 Details zum AST

Da ANTLR anstelle eines AST, einen CST erzeugt, wird dieser Schritt nun nachgeholt. Der Unterschied eines AST, zu einem CST, besteht darin, dass ein AST meist nur Blätter und Verzweigungen enthält, während der Baum eines CST viele unnötige Zwischenschritte beinhaltet. Außerdem ist die Struktur des CST vorgegeben durch die Grammatik und wird von ANTLR erzeugt. Diese Strukturen können deswegen nicht zur weiteren Verarbeitung angepasst oder erweitert werden.

Die Knoten des AST von Myll enthalten einige Zusatz-Informationen, welche der CST nicht hatte. Der gesamte AST hat an manchen Stellen sogar eine andere Hierarchische Struktur als der CST. Als Beispiel:

```
namespace JanSordid;       // { … } wird nicht zwingend gebraucht in Myll, alles danach ist "darin"
class MyClass { var int i; … }
```

Im CST liegt *namespace* parallel zur *class*, im AST wird jedoch, *class* in das *namespace* eingehängt, so wie es später auch erzeugt werden soll.

```
// CST:
NamespaceContext {
    NAMESPACE,                  // hier steht der String "namespace"
    id: "JanSordid",
    children: { /* leer */ },
    SEMI,                       // hier ist das Semikolon als String
}
StructuralDeclContext {
    // hier ist noch viel mehr wie das NAMESPACE, SEMI enthalten
    id: "MyClass",
    children: {
        TypedIdAcorContext { VAR, id: "i", … /* siehe Illustration 4.2: Finaler Parse Tree / CST  }
    }
}

// AST:
Namespace {
    name: "JanSordid",
    children: {
        Structural {
            name: "MyClass",
            children: {
                   Var { name: "i", TypespecBasic { Integer, 4 }, parent: /* siehe unten */ }
            }
            parent: /* Zeigt auf Namespace "JanSordid */
        }
    }
}
```

In diesem Schritt fällt viel überflüssige Zusatzinformation weg, die Verweise auf *parent* werden im AST hinzugefügt, um später bei der Namensauflösung auch in diese Richtung traversieren zu können.

Diese Struktur und Methoden dieser Klassen unterstützen die semantische Analyse bei ihrer Vervollständigung und sie bieten danach auch Funktionalität zur Erzeugung der Ausgabe.

4.4.1.1 AST-Deklarationen

In Myll wird eine Deklaration nicht von der Definition getrennt, es passiert immer beides zugleich. Diese Deklarationen hier umfassen nur globale-, statische- und Instanz-Deklarationen.

Die folgenden im Compiler implementierten Klassen machen alle Deklarationen aus, die Vererbungshierarchie ist durch Einrückung dargestellt (Hierarchical ist von Decl abgeleitet, Structural von Hierarchical):

- Decl – Basisklasse aller Arten von Deklarationen, *abstrakt*
  - Hierarchical – Alle Decls, welche weitere Decls enthalten können, *abstrakt*
    - Structural – Allgemeine strukturelle Deklarationen: `struct`, `class` und `union`
    - Enumeration – Enum Deklaration
    - Namespace – Namespace Deklaration
      - GlobalNamespace – Ausgangspunkt eines jeden Moduls
  - Var – Globale Variablen, globale Konstanten, Klassen-Variablen, automatische Accessoren
  - Using – Deklaration von Typ-Aliasen und Namespaces
  - Func – Deklaration von Funktionen, Methoden, Operatoren und manuelle Accessoren
  - ConDestructor – Deklarationen von Kon- und Destruktoren
  - EnumEntry – Eintrag in einem Enum
  - MultiDecl – Container, welcher mehrere Decl, in einer einzelnen Decl speichert

Der GlobalNamespace ist der Ausgangspunkt jedes Moduls und somit jedes AST. Er entspricht damit quasi dem, was in C++ *Translation Unit* genannt wird. GlobalNamespace enthält im Vergleich zu Namespace, noch den aktuellen Modulnamen, sowie die importierten Module.

MultiDecl war nötig, um aus einem ANTLR `Visit()` mehr als ein Element zurückzugeben. Es gibt keine Notwendigkeit von Myll, dass diese Klasse existiert.

EnumEntry ist nur in Enumeration gültig. ConDestructor nur innerhalb von Structural (class und struct). Namespace darf nur innerhalb von Namespace, damit auch im GlobalNamespace, erscheinen.

4.4.1.2 AST-Statements

Statements können in Funktionskörpern und als Teil von anderen Statements auftreten. Sie enthalten auch lokale-Deklarationen.

Diese Klassen machen alle Statements aus, die Vererbungshierarchie ist durch Einrückung dargestellt:

- Stmt – Alle Arten von Statements
  - VarStmt – Deklaration lokaler Variablen und lokaler Konstanten
  - UsingStmt - Deklaration lokaler Typ-Aliasen und Namespaces
  - IfStmt – Kontrollstruktur zur Verzweigung
  - SwitchStmt - Kontrollstruktur zur Mehrfachverzweigung
  - CaseStmt – Sprungziel einer Mehrfachverzweigung
  - LoopStmt – Basisklasse aller Schleifen, ist selbst eine endlose Schleife

        - ForStmt – Übliche drei Komponenten For-Schleife: Initialisierung, Bedingung, Fortschritt
        - WhileStmt – Kopfgesteuerte Schleife
        - DoWhileStmt – Fußgesteuerte Schleife
        - TimesStmt – Zählschleife
    - ContinueStmt – Bricht aktuelle Iteration ab und fährt mit der Nächsten fort
    - BreakStmt – Abbruchsstatement für Schleifen und Mehrfachverzweigungen
    - FallStmt – Designiert ein gewünschten Fallthrough bei Mehrfachverzweigungen
    - ReturnStmt – Rücksprung aus einer Funktion
    - TryCatchStmt – Ausnahmebehandlung
    - ThrowStmt – Ausnahmeauslösung
    - MultiAssign – Ein oder mehrere Zuweisungen, z.B. `a = b = true;`
    - AggrAssign – Eine aggregierende Zuweisung, z.B. `a += 1.41f;`
    - Block – Ein neuer Scope, welcher mehrere Statements enthalten kann
    - MultiStmt – Mehrere Statements ohne neuen Scope
    - EmptyStmt – Leeres Statement
    - ExprStmt – Ein Statement, welches nur Expressions enthält

MultiStmt ist analog zum MultiDecl. EmptyStmt existiert als *Lückenfüller* für Stellen an denen ein Statement optional wäre.

4.4.2 AST-Expressions

Expressions bestehen aus der größten Menge an unterschiedlichen Arten im gesamten AST. Um diese extreme Diversität nicht ausufern zu lassen, bleiben einige von ihnen ohne spezifische Subklasse. Sie sind von den Typen UnOp und BinOp und ihr effektiver Typ ist in einem Enum codiert. Dieses Enum erfüllt auch einen zusätzlichen wichtigen Zweck; die Änderung der Operator-Precedence bei der Erzeugung der Ausgabe zu steuern.

Diese Klassen machen alle Expressions aus, die Vererbungshierarchie ist durch Einrückung dargestellt:

- Expr – Alle Arten von Expressions, *abstrakt*
    - UnOp – Unäre-Operationen, welche eine weitere Expression enthalten, z.B. `-e`, `e++`, `sizeof(e)`
        - CastExpr – Typenumwandlung der enthaltenen Expression, z.B. `(? MyType) e`, `(move) e`
        - FuncCallExpr – Aufruf der Expression als Funktion/Methode, z.B. `e(arg, arg)`
    - BinOp – Binäre-Operation, welche zwei weitere Expressions enthalten, z.B. `e * e`, `e.e`, `e == e`
    - TernOp – Ternäre-Operation, welche drei weitere Expressions enthalten, z.B. `e ? e : e`
    - Literal – Jegliche Literale / Werte, z.B. `42`, `true`, `"test"`, `0xACAB`, `this`
    - IdExpr – Identifier mit optionalem Template Bestandteil, z.B. `myVar`, `pow<float>`
    - ScopedExpr – Verschachtelte IdExpr, z.B. `MyClass::staticField`, `std::is_pod<Typ>`
    - NewExpr – Speicheranforderungs-Operation, z.B. `new Balloon<0xFF0000>[99]`

4.4.3 Module

Myll nutzt Module als Alternative zu *export* und *include*, welche sich größtenteils analog zu Modulen aus C++20 verhalten sollen. Der Unterschied zu Includes aus C++ ist, dass Module ihre eigenen Imports nicht nach außen eskalieren. Man könnte sagen, Includes sind transitiv, Module jedoch nicht. Hier ein Beispiel:

- Modul *a* importiert Modul *b*
- Modul *b* importiert Modul *c* (Modul C enthält z.B. Klasse X)
- In Modul *a* sind Inhalte aus Modul *c* nicht direkt verfügbar (Modul a sieht Klasse X nicht)
- Sollte Modul *a* Klasse X verwenden wollen, muss es zusätlich Modul *c* importieren

Eine Myll-Datei gehört immer zu einem Modul, sollte keins angegeben sein, entspricht das Modul dem Dateinamen ohne Endung. Mehrere Myll-Dateien können zu ein und demselben Modul gehören. Myll-Dateien können beliebig viele Module inkludieren. Jedes Modul erzeugt jeweils eine .h- und .cpp-Datei mit dem Modulnamen (die .cpp-Datei kann wegfallen, sollte sie, wie etwa bei Template-Klassen, leer sein).

Die Zusammenfassung von mehreren Myll-Quellen zu einem Ziel-Modul kann bei der endgültigen Übersetzungsgeschwindigkeit der C++-Dateien helfen. Wenn man zum Beispiel sehr viele Klassen (und dementsprechend Dateien) für sein User-Interface hat, nutzen all diese, sehr wahrscheinlich, die selben Module/Includes/Bibliotheken. Ordnet man diese, sehr verwandten, Dateien dem gleichen Modul zu, wird eine kombinierte C++-Datei erzeugt, siehe dieses Beispiel:

```
// Datei ui_a.myll
module ui;
import Widgets;
class A {…}

// Datei ui_b.myll
module ui;
import Widgets;
class B {…}

// Datei ui_c.myll
module ui;
import Widgets;
class C {…}
```

Erzeugt wird eine .cpp und eine .h Datei:

```
// Datei ui.h
#include "Widgets.h"
class A {…};
class B {…};
class C {…};

// Datei ui.cpp
#incude "ui.h"
A::impl…
B::impl…
C::impl…
```

Diese enthält nun zwar die Kombination aus all diesen Dateien und braucht im Einzelnen auch länger zu kompilieren, als eine Einzelne, reduziert aber die gesamte Kompilationszeit enorm. Diese Art der Optimierung wird *Unity-Build* genannt, siehe [UNITY-BUILD]. Diese Optimierung ist hier nicht nur "drübergestülpt", sondern ein selektiv nutzbarer Teil des Sprach-Funktionsumfangs. Zur Erklärung sei erwähnt, dass C++ all seine *includes* einfach Textuell einbindet und dann eine *sehr sehr große* Datei kompiliert. Wenn man etwa nur *iostream* benutzt, bearbeitet der Compiler allein 25k – 50k Code-Zeilen für dieses Include, plus der eigenen Zeilen. Kombiniert man beispielsweise 25 C++-Dateien mit je 1k LoC zu einer einzelnen, muss man eine 50k LoC Datei kompilieren, verglichen mit 25-mal Dateien mit 26k LoC.

4.4.4 Ausführung der semantischen Analyse

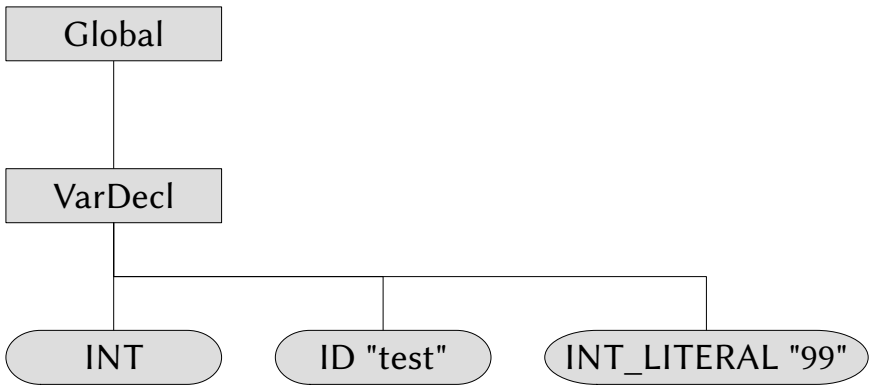
Bei der Frage, wie der CST durchlaufen werden soll, bot ANTLR zwei Möglichkeiten: *Visitor* (Besucher) und *Listener* (Lauscher). Die Entscheidung fiel auf den *Visitor*, da dort der Durchlauf des Syntax Baums selbst gesteuert werden kann; man kann Visit() auf die Kindesknoten in einer selbst gewählten Reihenfolge aufrufen oder auch manche Teile gar nicht besuchen. Diese Aufrufe gehen rekursiv bis in die Spitzen des Baumes. Alternativ dazu hätte der *Listener* alle Knoten von selbst, in vorgegebener Reihenfolge durchlaufen und lediglich die Möglichkeit gebote dabei *zuzuhören*.

Zum Beispiel der Visit Aufruf zur Erzeugung der vorher schon betrachteten lokalen Variable:

```
// Diese im Beispiel aufgeführte Zeile der Grammatik heißt im echten CST typedIdAcor
//        VAR      typespec idVars SEMI
public Stmt VisitStmtVar( TypedIdAcorContext c ) // Dieser Parameter ist ein Knoten des CST
{
    // Hier wird der Datentyp von seiner String Repräsentation in einen AST-Knoten übersetzt
    Typespec type = VisitTypespec( c.typespec() );
    Stmt ret = new VarStmt {
        srcPos   = c.ToSrcPos(),     // speichert Quellcode Positionen für mögliche Fehlermeldungen
        name     = q.id().GetText(), // der Name der Variable wird aus id in Textform abgerufen
                                     // im betrachteten Beispiel: "test" (hier simplifiziert)
        type     = type,             // der Datentyp wird der Variablen zugewiesen
                                     // … hier: BasicTypespec { kind = Integer, size = 4 }
        init     = q.expr().Visit()  // Visit der enthaltenen Expression
    };                               // … hier: Literal { text = "99 }
    return ret;                      // Stmt-Knoten wird zurückgegeben und dort in den restlichen
}                                    // … Baum eingehängt
```

So sieht der AST des zuvor erzeugten Beispiels aus, siehe Illustration 4.3: Finaler AST.

Illustration 4.3: Finaler AST



VarDecl ist eine von vielen Klassen welche einen Knoten des AST darstellt. Sie hat zwei verpflichtende und einen optionalen Kindsknoten: Typ, Name und opt. Initialisierung. VAR, ASSIGN und SEMI sind implizit in VarDecl enthalten und fallen weg. Der Informationsgehalt des AST entspricht dem des CST, die Informationsdichte des AST ist höher.

Die semantische Analyse dient der Auflösung der, bisher nur textuell erfassten, Typen und Namen. *Auflösung* (oder Zuordnung) heißt hier, dass herausgefunden wird, was mit einem Text wirklich gemeint ist. Im vorangegangenen Beispiel wurde die globale Variable *test* erfasst, sollte nun eine andere Expression irgendwo *test* enthalten, muss der Compiler diese beiden erst einmal miteinander verknüpfen. Die semantische Analyse ist in Myll in drei Schritte unterteilt: der Erste dieser Schritte findet schon beim Durchlauf des CST statt. Der Gesamtablauf ist wie folgt:

Schritt 1:

Durchlauf des CST und Erzeugung eines vorläufigen AST des kompletten Programms.

Die Namen, von global sichtbaren Deklarationen, werden in einer separaten Datenstruktur erfasst. Diese Deklarationen sind:

- Namespaces
- Typen Deklarationen (struct, class, enum, union, alias)
- globale-, statische- und Instanz-Variablen
- Funktionen und Methoden

Alle importierten Module müssen diesen Schritt durchlaufen haben, bevor der nächste Schritt beginnen darf, weil sonst Typen womöglich falsch aufgelöst werden. Alle Module können diesen Schritt aber schon parallel zueinander durchführen.

Schritt 2:

Da alle globalen Namen nun bekannt sind, können alle Referenzen auf diese Globalen nun zugeordnet werden. Diese Referenzen sind (enthalten in):

- Using Namespace
- Basisklassen
- Typen der globalen-, statischen- und Instanz-Variablen
- Parameter- und Rückgabetypen von Funktionen und Methoden

Die Zuordnung geschieht durch die Ersetzung der textuellen Repräsentation durch einen Zeiger auf das aufgelöste Element. So wird z.B. `myUsingDecl { namespace = "std" }` nach Auflösung `myUsingDec { namespace = &myStdDecl }`.

Für die Zukunft wäre es sinnvoll diese Daten in einer einfach einzulesenden Form abzuspeichern. Für partielle Rekompilationen können diese eingelesen werden, ohne die abhängigen Module neu kompilieren zu müssen. Auch Code außerhalb von Myll, kann seine Signaturen über diesen Weg bereitstellen, analog zu TypeScripts .d.ts Dateien (welche genutzt werden, um typenlose JavaScript-Signaturen, mit Typen zu ergänzen).

Hinweis: Dieser Schritt ist noch nicht vollständig im aktuellen Prototypen implementiert.

Schritt 3:

Funktionen, Methoden und Expressions können nun aufgelöst werden. Da diese keine globalen Deklarationen mehr verändern, kann der Schritt massiv parallel ausgeführt werden.

Innerhalb von Funktionen werden lokale Variablen in einem Scope-Stack erfasst, jeder neue lokale Scope fügt eine Ebene oben auf diesen Stack hinzu. Wenn nun ein Identifier (Variablen- oder Typenname) aufgelöst werden soll, wird alles was in diesem Stack ist, von oben nach unten, in Betracht gezogen. Sollte der Scope-Stack den Identifier nicht beherbergen, so wird in der globalen Datenstruktur aus Schritt 1 weitergesucht.

Hinweis: Dieser Schritt ist ebenso noch nicht vollständig implementiert im aktuellen Prototypen.

Um die bisher theoretisch betrachteten Schritte noch einmal zu Verdeutlichen, seien sie an einem Beispiel wiederholt. Wenn hier im Beispiel von fertig markiert gesprochen wird, heißt das, dass alles, was ein Konstrukt enthält, fertig aufgelöst ist.

Beispiel-Code

```
var int g_i;                                // (1) g_i
func myfunc() -> BC::Sub {                  // (2) myfunc
    var BC      a;
    var BC::Sub b;
    a.c( b, g_i );
    return b;
}
class BC : SomeBase {                       // (3) BC
    class Sub {}                            // (4) BC::Sub
}
class SomeBase {                            // (5) SomeBase
    func c<T>( BC::Sub& b, T i ) {…}        // (6) SomeBase::c<T>
}
```

In diesem Beispiel sind sechs globale Deklarationen enthalten

Schritt 1:

- Alle globalen Deklarationen werden hierarchisch durchlaufen.
- Bei (1): Die globale Variable namens *g_i* wird mit dem eingebauten Typen *int* erfasst, diese Deklaration ist damit schon vollständig aufgelöst und wird als fertig markiert.
- Bei (2): *myfunc* wird global als Funktion erfasst mit *BC::Sub* als Rückgabetyp, da dieser bisher unbekannt ist, wird er aktuell nur textuell erfasst und die Implementierung von *myfunc* wird vorerst ignoriert.
- Bei (3): *BC* wird global als Klasse erfasst, die Basisklasse *SomeBase* ist noch unbekannt und wird deswegen auch nur textuell erfasst.
- Bei (4): *Sub* wird unter *BC* als Klasse erfasst, ist vollständig aufgelöst und wird als fertig markiert.
- Bei (5): *SomeBase* wird global als Klasse erfasst.
- Bei (6): Die Funktion *c* wird unter *SomeBase* mit Parametertyp *BC::Sub* textuell erfasst, T bleibt offen
- Die Implementierung von *SomeBase::c* wird vorerst ignoriert.

Schritt 2:

- Alle unfertigen globalen Deklarationen werden hierarchisch durchlaufen
- Bei (2): Der Rückgabetyp von *myfunc* wird jetzt durch einen Verweis auf die in Schritt 1 erfasste *BC::Sub* Klasse ersetzt und da ihre Signatur vollständig aufgelöst ist, wird sie als fertig markiert
- Bei (3): Die Basisklasse von *BC* wird durch einen Verweis auf die Klasse *SomeBase* ersetzt, sie wird als fertig markiert, da hier nichts direkt auf die (bisher unfertige) Basisklasse *SomeBase* verweist.
- Bei (5): Klasse *SomeBase* ist noch nicht fertig, weil nicht alle Kinder fertig sind.
- Bei (6): Die Parametertyp *BC::Sub* von *SomeBase::c* wird aufgelöst, wird als fertig markiert, hier bleibt der Template Parameter T jedoch offen.
- Bei (5): Rückkehr zur Klasse *SomeBase*, diese wird als fertig markiert, da alle Kinder fertig sind.
- Der gesamte Schritt 2 wird ein erneut durchlaufen, sollte etwas nicht als fertig markiert sein.
- Sollte die Anzahl, der noch nicht fertigen Deklarationen, zwischen den erneuten Aufrufen nicht schrumpfen, liegt ein fehlerhaftes Programm vor und es wird mit einem Fehler abgebrochen.

Schritt 3:

- Alle nicht-Template Funktionskörper werden aufgelöst

- In (2): Lokaler Scope von *myfunc* wird erzeugt
- Die Variable *a* vom Typ *BC* wird in lokalem Scope erzeugt, BC ist global bekannt und wird referenziert.
- Die Variable *b* vom Typ *BC::Sub* wird in lokalem Scope erzeugt, BC::Sub ist global bekannt und wird referenziert.
- Auf die Variable *a* wird die Methode *c* aufgerufen. In dieser wird der Template-Parameter *T* mit dem Typen *int* durch die Übergabe des zweiten Arguments bestimmt (der Typ der globalen Variable *g_i)*. Die Template-Funktion wird instanziiert (hier nicht ausgeführt) und der Typ des ersten Arguments (der Variable *b)* wird als kompatibel mit dem ersten Parameter validiert.
- Die Variable *b* wird zurückgegeben, dessen Typ ist kompatibel mit der Rückgabe der Funktion.
- Der lokale Scope wird geschlossen.
- Template-Funktionen, wie *SomeBase::c,* werden (wie in C++) nur im Falle der Nutzung aufgelöst. Im Falle ihrer Nutzung sind sie natürlich vollständig typisiert und werden wie normale Funktionen erzeugt.
- Da alle nicht-Template Funktionskörper erzeugt werden konnten, ist dies als ein gültiges Programm validiert und die semantische Analyse ist abgeschlossen.

4.5 Ausgabe des Kompilats – Generator

Da sich Myll semantisch nur in Details von C++ unterscheidet, muss für die Ausgabe keine grundlegende Transformation erfolgen. Die kleinen strukturellen Änderungen die nötig waren sind in der semantischen Analyse passiert.

Da einem Myll erlaubt, Konstrukte in beliebiger Reihenfolge zu schreiben, wohingegen C++ oft erst nach Deklaration die Nutzung erlaubt, musste das berücksichtigt werden. Um dieses, schon in 3.3.1.1 betrachtete, Problem zu lösen, braucht man Prototypen in der Ausgabe. Die Lösung, welche wenige Prototypen erzeugt hätte, wäre die Analyse des gesamten Programms mittels Abhängigkeitsbaums gewesen. Diese Lösung konnte aufgrund der unvollständigen Schritte 2 und 3 der semantischen Analyse nicht zuverlässig durchgeführt werden.

Alternativ könnte natürlich einfach für alles Prototypen erzeugt werden, aber selbst dabei hätten Reihenfolgeprobleme auftreten könnten. Durch eine geschickte Sortierung ist es dennoch gelungen, dieses Problem statisch zu lösen, also mit einer festen Reihenfolge der Ausgabe.
Diese Liste zeigt die Ausgabereihenfolge des erzeugten C++-Codes welche für alle von *Hierarchical* abgeleiteten Klassen gilt (siehe 4.4.1.1). Einige der Einträge sind nicht in jeder abgeleiteten Klasse notwendig und dadurch verschwenderisch, es simplifizierte jedoch die Implementierung dieses Prototypen.

- Frühe Prototypen (class, struct, union und enum)
- Alle hierin enthaltenen *Hierarchical* (class, struct, union, enum und namespace)
- Statische (Klassen-)Variablen
- Statische Funktionen und Accessoren (get, set, refget)
- (Klassen-)Variablen
- Späte Prototypen (nur Funktionen außerhalb von Klassen)
- Konstruktoren / Destruktoren
- (Klassen-)Funktionen, Operatoren und Accessoren (get, set, refget)

Dadurch, dass sich nun die Reihenfolge einiger Konstrukte, von Myll, nach C++ ändert, musste Aufmerksamkeit darauf gelegt werden, die Reihenfolge von (Klassen-)Variablen beizubehalten. Diese Reihenfolge bestimmt nämlich das effektive Speicherlayout in C++ und sollte weiterhin vom versierten Benutzer durch optimale Positionierung eingesetzt werden können.

Jeglicher Code ist in einem *Hierarchical* enthalten, da schon der *GlobalScope* ein solches ist. Die Erzeugung startet beim *GlobalScope* und findet mittels Durchlauf des AST statt. Die *Hierarchical* kümmern sich dabei *high-level* um die Erzeugung jeglicher Deklarationen, Statements und Expressions, die Details machen die spezifischen Klassen.

Für die Ausgabe nach C++ werden je Modul zwei Dateien erzeugt, eine .cpp-Datei und eine .h-Datei. Die Liste der Erzeugungsreihenfolge der .cpp-Datei ist fast identisch mit der vorher Aufgeführten (welche sich auf die .h-Datei bezieht), sie enthält nur die beiden Prototypen Sektionen nicht.

Da die Erzeugung von sehr viel Quellcode problematisch in einer Programmiersprache wie C# sein kann, da hier Strings von hause aus unveränderlich sind, denn das Bedeutet wenn man an einen sehr langen String ein Zeichen anhängt, das dessen Inhalt komplett kopiert werden muss. Dieses Problem wurde umgangen indem jede fertige Zeile in einem String gespeichert wurde und mehrere Zeilen in einer `List<String>`, dem Äquivalent zu `vector<string>`. Diese Listen sind veränderbar, also eine einzelne Zeile die an einer langen Liste angehängt wird führt nur sehr selten eine re-allokation ihres Speicherplatzes durch.

5 Auswertung & Zusammenfassung

5.1 Bewertung der Grundsatzeinhaltung

Ich habe mir Grundsätze mir Grundsätze auferlegt um den Fokus nicht aus den Augen zu verlieren.
Deren Einhaltung sei hier von mir selbst bewertet.

5.1.1 Erwarte keine Wiederholung vom Nutzer

Myll bietet leichtere Lesbarkeit, dank weniger Einrückung durch die Fähigkeit, verschachtelte Namespaces in einem einzelnen Block zu schreiben oder den Block gar ganz wegzulassen und den Rest der Datei in letztgenanntem Namespace zu haben.

Es erwartet weniger Boilerplate dadurch, dass nicht bei jeder Definition `template <typename …>`, `Class Name::…` und eine zweifache Aufzählung der Funktions- und Methodensignaturen gefordert wird, welche durch die Auftrennung in .cpp- und .h-Dateien gefordert war.

Die Notwendigkeit von Prototypen wurde eliminiert.

Einige Teile erfordern zwar ein wenig mehr zu schreiben als bisher, allen vorweg `func` und `var`. Sie bieten aber auch leichtere Lesbarkeit, weil sie einem die kognitive Last nehmen den kompletten Inhalt einer Codezeile zu lesen. um zu verstehen, was dort generell passiert.

5.1.2 Außergewöhnliches Verhalten muss Explizit sein

Eine Sprache welche UB bekannt gemacht hat, kann man nicht mehr von *Außergewöhnlichem Verhalten* heilen… ;-)

Ich hoffe jedoch, dass zumindest durch die Änderungen an den nun standardmäßigen *implicit Konstruktoren*, dem nicht mehr automatisch *fallenden Switch*, einem *nicht-arithmetischen Pointer* auf Skalare und Warnungen die, durch das integrale Heraufstufen, sich bisher nicht umsetzen ließen, die Gesamtsituation ein wenig sicherer gemacht habe, ohne der Sprache *Stützräder* anzubauen.

Einschränkende Konvertierungen sind ja schon von C++ durch die Einführung der Initialisierungslisten bekämpft worden, ich bringe die gleiche Funktionalität zurück in herkömmliche Schreibweisen.

Außerdem können einem selektiv *selbst auferlegte Einschränkungen* davor warnen, sollte man sie verletzen.

5.1.3 Das was man häufig will kann Implizit sein

Dieses Ziel wurde erreicht dadurch dass, *Bitweise Operationen vor Gleichheit* einsortiert, jetzt von Hause aus *kein Shadowing* vorliegt, das *Ableitungen standardmäßig public* und Funktionen standardmäßig noexcept sind.

5.1.4 Breche nicht mit der Semantik von C++

Größte Sorgfalt wurde darauf gelegt, die Semantik beizubehalten. Es kommt jetzt drauf an ob man implizite Konvertierungen und das Vorhandensein eines Präprozessors als Bruch mit der Semantik ansehen mag. Aber auch in diesem Fall denke ich das es verschmerzbar ist.

5.1.5 Breche mit C sofern es einen Nutzen bringt

Es gab überraschend viel Gutes an mancher Syntax und der Einfachheit von C, welches bewahrenswert oder in leicht modifizierter Form wiederauflebenswert war.

Jeder Bruch mit der Syntax von C ist in dieser Arbeit dokumentiert und dient einem Nutzen.

- Keine undurchsichtige Funktionspointer und Array Syntax
- Keine Unscoped-Enumerationen
- Kein archaischer Präprozessor
- Keine Typen mit Leerzeichen getrennten Namen z.B. unsigned long long int
- Kein `#define NULL 0` und `myPtr = 0`

Es gab aber auch Revivals von C-Style-Syntax

- Wieder nutzbarer gemachte C-Style-Cast Syntax
- An Raw-Pointer angeglichene Smartpointer Syntax
- Initialisierungslisten wurden unnötig gemacht, dadurch ist die zukünftige Unterstützung von designated Initializers möglich geworden.

5.1.6 Entwickle die Syntax so weiter, dass sie eindeutig bleibt

Viele syntaktische Makel welche in C und C++ auftraten wurden beachtet und vermieden.

Das Problem unter dem der Lexer in C++ leidet, Variablendefinition und Funktionsdeklaration zu unterscheiden wurde durch die Einführung der *var* und *func* Keywords gelöst. Dies macht es auch fürs Menschenauge leichter entscheidbar was man gerade ließt. Siehe 3.3.2.1

- Keywords für Variablen und Funktionen zur schnelleren Erkennbarkeit
- Viele Schreibweisen wurden vereinheitlicht, z.B. Funktions und Lambda Decl
- Schreibweisen wurden erleichtert: Funktionspointer, Mix aus * und []
- Most Vexing Parse (Erklären und Quelle, vector<array<100>>4>>)

5.1.7 Sei auch einmalig nützlich

Kommentare werden leider noch nicht von Myll nach C++ übergeben und es gibt mehr Prototypen als nötig wären, ansonsten denke ich, dass der Code gut lesbar ist und in etwa dem entspricht, was ich selbst von Hand schreiben würde.

5.1.8 Spare nicht mit neuen Keywords

Es wurden einige Keywords eingeführt, um Ambiguitäten zu unterbinden, den Code lesbarer zu gestalten und neue Konstrukte mit angenehmer Syntax einzuführen.

Die Einführung der Attribut-Syntax half dabei nicht zu viele neue Keywords einführen zu müssen und sogar noch bisherige wieder freizugeben. Auch kann sich Myll so leichter fortentwickeln da diese Attribute weder mit Identifern noch mit Keywords kollidieren.

5.2 Fazit

Dies ist **nicht** der erste ambitionierte Compiler den ich angefangen habe zu schreiben. Ich denke mir: *Was soll ich mit einem Taschenrechner welcher umgekehrte polnische Notation ausführt?* **Nichts!** Und deshalb muss es dann natürlich direkt etwas wie C++ werden.

Dies ist der erste Compiler den ich geschrieben habe, welcher nicht trivial ist, der **funktioniert**.

Die Problematik eine so komplexe Sprache wie C++ zu lesen, zu verstehen und dann wieder auszugeben, ist ein hartes aber machbares Unterfangen. Ohne mir schon vor dieser Arbeit viel über C++ und allgemeinem Sprachdesign angeeignet zu haben, wäre es aber nicht in dem Zeitrahmen dieser Arbeit machbar gewesen.

Jemand anderem der ausgehend von dieser Arbeit einen Transpiler nach C++ aber auch C#, Java, Rust oder Ähnlichem schreiben will, wird, so denke ich, keine großartigen Probleme haben, solange die Grundstruktur der Sprache übereinstimmt.

TypeScript Antwortssatz???

Hinweis: Dies ist aktuell nur ein Prototyp, welcher manche der genannten Funktionen gar nicht oder etwa nur bis zur Grammatik umsetzt.

5.3 Rückblick auf die Implementierung

5.3.1 Was lief gut

Das was gut lief, ist dass ich dieses mal nicht irgendwann in einer Sackgasse aufgewacht bin und wusste "um dieses neue Problem zu lösen muss ich alles neu schreiben".

Das erste *high-productivity*-Feature mittels Attribut [flags, operators(bitwise)], auf einem Enum, erzeugt 27 Zeilen nützlichen Code und es hat keinen Tag gedauert es einzubauen.

Parallelisierung lies sich für ein Großteil des Compilers relativ einfach umsetzen. Ein Teil des Compilers benutzt statische Member, weshalb hier aktuell nicht parallelisiert werden konnte.

5.3.2 Was war problematisch

Die schlechteste Entscheidung, die ich bei dieser Modellierung getroffen habe, ist die wenigen Überschneidungen der Deklarationen und Statements vom gleichen Code / den gleichen Klassen behandeln zu lassen. Dies betraf insbesondere die Deklaration der Lokalen-, Globalen- und Instanz-Variablen.

Dies kam wahrscheinlich zustande, weil Decl von Stmt abgeleitet ist. Decl muss zwar alles können, was Stmt auch kann, erfüllt aber nicht die Anforderung *Decl ist ein Stmt*, zumindest nicht in Mylls Definition.

Problematisch waren einige *convenience* Schreibweisen, welche etwa mit einer Decl mehrere Decl erzeugten. Dies bot Probleme mit der Funktionsweise von ANTLR, welches als Rückgabe nur eine einzelne Decl erlaubte. Umschifft wurde dieses Problem durch die Einführung einer Multi-Decl Klasse, welche lediglich ein Container für mehrere Decl ist.

In Voraussicht wurde der Right-Shift-Operator in der Grammatik von Myll als zwei separate `>` definiert, welches die Probleme beim Schließen zweier Templates löste (das auch bei sehr alten C++-Versionen auftrat). Ein Problem mit Template-Argumenten in Myll trat dennoch auf, dass etwa diese Expression hier falsch *verstanden* wurde:

```
static_cast< Type >( lhs ) | static_cast< Type >( rhs );
static_cast<(Type >( lhs ) | static_cast< Type)>( rhs ); // eine große Expression
```

Der Ursprung war aber ein komplett anderer als die alte Template-Problematik. Hier war nämlich definiert, dass ein Template-Argument eine beliebige Expression sein konnte und diese verhält sich *greedy* und konsumierte hier zu viele Tokens. Die Template-Regel war so lange zufrieden, wie noch ein schließendes `>` vor dem nächsten `;` auftaucht. Das generelle Problem war, aber auch hier, dass spitze Klammern zwei unterschiedliche Bedeutungen haben. Dieser Fehler konnte behoben werden, es war nur ein kleiner Schock.

5.4 Das sieht ja gar nicht mehr wie C++ aus!

Das stimmt schon... Aber sieht denn C++11 noch so aus wie C++98?
Nein, siehe folgendes Beispiel:

```
// C++98
int * pi = new int(42);                    // Hier der Standard, verpönt in der Zukunft
delete pi;                                 // Freigeben nicht vergessen!
typedef a b;                               // Typalias
void genVec(vector<int> outVec) {…}        // Gute Performance nur mit pass-by-reference

// C++11
unique_ptr<int> pi = make_unique<int>(42); // So soll es in C++11 aussehen, delete unnötig
using b = a;                               // Moderner Typalias
vector<int> genVec() {…}                   // OK in C++11, perfekt in C++17
```

Und wird C++20 noch so aussehen wie C++11?
Auch nein: Der Standard ist schon abgesegnet und einiges von diesem neuen Standard hat sich, im Vergleich zum letzten großen Standard neun Jahre vorher, verändert. Es gibt schon Compiler, welche viele Funktion des neuen Standards unterstützen.

Das Feature constexpr hat schon viel TMP unnötig gemacht und wird dies, dank der Einführung von consteval fortsetzen. Man kann damit beispielsweise Funktionen schreiben, welche zur Kompilationszeit ausgeführt werden, die selben aber auch zur Laufzeit verwenden.

Concepts werden die am schwersten zu durchschauenden Parametrierungsfehler von Templates, durch sinnvolle Fehlermeldungen ersetzen und einige schwer zu schreibende SFINAE/TMP Konstruktionen unnötig machen.

Der Spaceship Operator `<=>` wird einem die Arbeit abnehmen, die meisten Vergleichsoperatoren zu implementieren. Er verhält sich relativ analog zu *strcmp.*

Es ist auch gut, dass man von einen C++-Standard zum anderen eine Veränderung sehen kann, sonst hätte sich ja auch nicht viel verbessern können.

Genau das versuche ich auch zu bezwecken, Myll soll *vertraut* aber *anders* sein. Vielleicht in eine andere Himmelsrichtung anders als die kommenden C++-Standards, denn sonst wäre Myll ja auch sinnlos.

5.5 Was konnte nicht behandelt werden

Einige Punkte welche nicht umgesetzt wurden, sind schon in Kapitel 3.3 beschrieben.

5.5.1 Alles was der C++-Semantik widerspricht

Das Grundkonzept, welches ein hohes Maß an Machbarkeit dieses Projektes ausgemacht hat, ist, dass die Semantik von C++ bewahrt wird. Wäre ein stärkerer Bruch mit C++ beabsichtigt worden, wäre die Implementation dieses Compilers wesentlich schwerer gefallen.

Ein Bruch mit der Semantik von C++ hätte sicher an einigen Stellen positive Auswirkungen haben können, aber das Grundkonzept wollte auch C++ Programmierer mit geringem kognitivem Umstellungsaufwand abholen. Eine Verletzung dieser Grundlage hätte sich negativ auf dieses Ziel ausgewirkt.

5.5.2 Adresse-Von- und Indirektions-Operatoren

Naive Schreibweise beabsichtigt: Ein Problem, welches meinem ursprünglichen *Verständnis* von C++ im Wege stand, war, dass, um aus einem Objekt einen Pointer zu machen, muss man `&` verwenden und nicht `*`. Umgekehrt, also wenn man schon einen Pointer (also irgendwas mit einem `*` dran) hat, muss man noch einen weiteren Pointer (`*`) dazuschreiben, damit der Pointer *weggeht*.

```
int i;
int *ip = *i;      // Wieso nicht so? Dann hätte ich C++ schon Jahre früher verstanden
                   // … oder ein gutes Buch, ich hatte nur ein schlechtes von Data Becker
cout << &ip;       // Das hier, nicht zu etwas was eh schon *ip ist, noch ein * davor schreiben
```

Eine mögliche Entwicklung wäre die Operationen `&` und `*` zu vertauschen um eine, meinem alten Ich, naiv logische Situation zu erzeugen. Es würde allerdings auch harsch nicht nur mit der Homogenität zu C++ brechen, denn auch D, Rust, Go, C# (ja *auch* C# kann Pointer), etc verwenden die gleiche Syntax.

5.5.3 Nicht weiter zu C++ kompilieren

Die Semantik von C++ kann weiter verfolgt werden, es könnte nur z.B. entweder direkt mit Clang++ (eingebunden als Bibliothek) kommuniziert werden oder LLVM-IR-Code (der Zwischensprache von LLVM) erzeugt werden. Dies würde ermöglichen, dass man sich nicht mehr übermäßig mit der Syntax von C++ herumschlagen muss und auch Funktionalität umsetzen kann, welche in C++ nicht funktioniert.

Probleme welche hieraus entstehen, sind, dass man nicht mehr so einfach C++- und Myll-Code in Kombination verwenden kann und dass man auf einen Compiler begrenzt ist. Myll erzeugt C++-Code welcher natürlich mit normalem C++-Code, zu einer gemeinsamen ausführbaren Datei gelinkt werden kann.

5.5.4 Einschränkungen in der Implementierung

Es wird Code geben der zwar von Myll nach C++ kompiliert, dann aber beim Kompilieren des C++-Codes scheitert. Der Grund kann sein das der ursprüngliche Code schon Fehlerhaft ist, Myll aber nicht alle Fehler welche ein C++-Compiler erkennen würde erkennt. Wenn man Myll in dieser Form scheitern lassen will, dann schafft man dies auch. Dies wird wohl auch noch lange möglich sein, weil ich keinen C++ Compiler nachbauen kann.

Einige mehr oder minder komplexe Komponenten fehlen schon in der Grammatik:

- range-based-for
- fall
- continue
- try / catch
- Templates auf Variablen
- Template-Specialization
- Variadic-Templates

Dessen Auslassung war für die ersten einfach ein Versehen, für die letzten pure Selbsterhaltung.

Die folgenden Komponenten sind schon in der Grammatik vorhanden, sie werden nur noch nicht in den AST übertragen:

- alias
- throw
- concept
- requires
- aspect
- lambda

Es gibt aber auch das Gegenteil, es existiert Code zu hier nicht beschriebener Funktionalität, welche es entweder aus Zeitgründen nicht in diese Arbeit schaffte oder weil mir keine gute Argumentation eingefallen ist sie hier zu beschreiben.

In Myll kann der Null-Conditional-Operator nicht ohne weiteren Aufwand an Stellen auftreten, wo nur Expressions erscheinen dürfen, weil er zur Umsetzung Statements erzeugen muss.

6 Ausblick

Zuerst müssen die wenigen Kern-Sprachkonstrukte, welche vergessen oder aus Zeitgründen verschoben wurden, umgesetzt werden. Außerdem endet aktuell noch jeder Syntax-Error im Debugger, die Exceptions haben aber zum Teil schon gute Informationen, welche man mit wenig Aufwand in eine brauchbare Ausgabe verwandeln kann. Wenn diese Punkte abgeschlossen sind sollte sich der Compiler in der Alpha Phase befinden und darf dann gerne von Anderen zum Experimentieren genutzt werden.

Es gibt noch sehr viele gute Ideen welche im Kapitel: 3.3 Lösung der problematischen Aspekte von C++ zwar angesprochen, aber noch nicht umgesetzt wurden. Einige davon sind sogar recht leicht lösbar.

Eine andere Herausforderung, sollte dann sein, die Sprache bekannt zu machen und zu schauen ob es eine Nische oder mehr für sie gibt. Feedback ist sehr Willkommen.

Auch will ich weitere *high-productivity-Features* umsetzen, sie könnten sich als ein *Alleinstellungsmerkmal* herausstellen.

7 Appendix

7.1 Acknowledgment

Even though this Thesis seems like a long rant about C and C++ (which it kind of is), it is in no way meant in a demeaning way about Dennis Ritchie's and Bjarne Stroutrup's lifework (and all other people involved in bringing C and C++ forward). Their inventions survived until the current day and will do so in the future, because they are *that good*.

Special thanks to Jonathan Blow and Per Vognsen for all the videos about Jai and Ion, the good explanations and pointing out the simplicity of using something like C again.

7.2 Danksagung

Ein großer Dank geht an die Herren Rethmann, Ueberholz, Goebbels, aber auch die weiteren Professorinnen und Professoren der Hochschule Niederrhein, für die gute Ausbildung und die vielen interessanten Gespräche. Die HSNR wird immer mein zweites Zuhause bleiben.

Ein besonderer Dank geht an meine Anika, für die mitdenkende und auch die mentale Unterstützung in dieser, auch von Außen, anstrengenden Zeit, <3.

Ich danke meinen Eltern, für die Werte die sie mir vermittelt haben, und auch dass sie mir dies niemals aufs Brot geschmiert haben. Ich danke meinem Bruder Arne für die Verbindung, die ich nur mit einem haben kann.

7.3 Literaturverzeichnis

[KR-CPL] Brian Kernighan, Dennis Ritchie: *The C Programming Language,* 19

[WC-SPECS] Ben Werther, Damian Conway: *A Modest Proposal: SPECS*
http://users.monash.edu/~damian/papers/PDF/ModestProposal.pdf
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.2481&rep=rep1&type=pdf

[PHF-A4] Parr, Harwell, Fisher: *Adaptive LL(*) Parsing: The Power of Dynamic Analysis*
https://www.antlr.org/papers/allstar-techreport.pdf

[MA-DOD] Mike Acton: *Data-Oriented Design and C++*
https://www.youtube.com/watch?v=rX0ItVEVjHc
https://github.com/CppCon/CppCon2014/blob/master/Presentations/Data-Oriented%20Design%20and%20C%2B%2B/Data-Oriented%20Design%20and%20C%2B%2B%20-%20Mike%20Acton%20-%20CppCon%202014.pptx

[JB-JAI] Jonathan Blow: *Ideas about a new programming language for games*
https://www.youtube.com/watch?v=TH9VCN6UkyQ&list=PLmV5I2fxaiCKfxMBrNsU1kgKJXD3PkyxO

[PV-ION] Per Vognsen: *Bitwise Ion*
https://github.com/pervognsen/bitwise/blob/master/notes/ion_motivation.md
https://www.youtube.com/watch?v=T6TyvsKo_KI&list=PLU94OURih-CiP4WxKSMt3UcwMSDM3aTtX&index=3&t=3100s

[MS-NSW] Malte Skarupke: *Ideas for a Programming Language Part 3: No Shadow Worlds*
https://probablydance.com/2015/02/16/ideas-for-a-programming-language-part-3-no-shadow-worlds/

[VR-EPO] Vittorio Romeo: *Epochs: a backward-compatible language evolution mechanism*
https://vittorioromeo.info/index/blog/fixing_cpp_with_epochs.html
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1881r0.html

[JM-EA] Jonathan Müller: *Write explicit constructors - but what about assignment?*
https://foonathan.net/2017/10/explicit-assignment

[JM-SM] Jonathan Müller: *Tutorial: When to Write Which Special Member*
https://foonathan.net/2019/02/special-member-functions

[HS-SPP] Herb Sutter: *GotW #91 Solution: Smart Pointer Parameters*
https://herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters

[HS-EX] Herb Sutter: *De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable*
https://www.youtube.com/watch?v=ARYP83yNAWk&t=2680

[AW-1YC] Andre Weissflog: *One year of C – Less Language Feature 'Anxiety'*
https://floooh.github.io/2018/06/02/one-year-of-c.html#less-language-feature-anxiety

[TS-EX] Tim Sweeney: *Tweet about exceptions in C++*
https://twitter.com/timsweeneyepic/status/1223077404660371456

[BCPL-OP] Clive D.W. Feather: *A brief(ish) description of BCPL – Operators*
https://www.lysator.liu.se/c/clive-on-bcpl.html#operators

[DR-CDEV] Dennis M. Ritchie: *The Development of the C Language*
https://www.bell-labs.com/usr/dmr/www/chist.html

[ISO-OD] ISO C++: *Inheritance – Why doesn't overloading work for derived classes?*
https://isocpp.org/wiki/faq/strange-inheritance#overload-derived

[UNITY-BUILD] Viktor Kirilov: *A guide to unity builds*
https://onqtam.com/programming/2018-07-07-unity-builds

[IW-TSJS] InfoWorld, Martin Heller: *TypeScript vs. JavaScript: Understand the differences*
https://www.infoworld.com/article/3526447/typescript-vs-javascript-understand-the-differences.html

[TR-R5Y] The Register, Thomas Claburn: *Rust marks five years since its 1.0 release*
https://www.theregister.co.uk/2020/05/15/rust_marks_five_years_since/

[POP] TechRepublic, Nick Heath: *Most popular programming languages: C++ knocks Python out of top three*
https://www.techrepublic.com/article/most-popular-programming-languages-c-knocks-python-out-of-top-three/

C++ reference:

[FALL] C++ reference: *C++ attribute: fallthrough*
https://en.cppreference.com/w/cpp/language/attributes/fallthrough

[EXPL-CAST] C++ reference: *Explicit type conversion*
https://en.cppreference.com/w/cpp/language/explicit_cast

[JAVAPERF] Stack Exchange – Software Engineering, Michael Borgwardt: *C++ faster than JVM or CLR*
https://softwareengineering.stackexchange.com/questions/159373/what-backs-up-the-claim-that-c-can-be-faster-than-a-jvm-or-clr-with-jit/159380#159380

[DLANG] D Programming Language: *Overview*
https://dlang.org/overview.html

[TIOBE] TIOBE Software BV: *TIOBE Index*
https://www.tiobe.com/tiobe-index

Wikipedia:

[CACHE] Wikipedia: *Cache – Cache-Line/Cache-Zeile*
https://de.wikipedia.org/wiki/Cache#Cache-Line/Cache-Zeile

[CRIT-OP] Wikipedia: *Operators in C and C++ – Criticism of bitwise and equality operators precedence*
https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#Criticism_of_bitwise_and_equality_operators_precedence

[SPEC-MEM] Wikipedia: *Special member functions*
https://en.wikipedia.org/wiki/Special_member_functions

[LEX-HACK] Wikipedia: *Lexer hack*
https://en.wikipedia.org/wiki/Lexer_hack

[JAVAHIST] Wikipedia: *Java – History.* Verweis auf Oak, einer der frühen Version von Java
https://en.wikipedia.org/wiki/Java_(programming_language)#History
https://web.archive.org/web/20070110091619/http://today.java.net/jag/old/green/OakSpec0.2.ps

[P-CLIMB] Wikipedia: *Operator-precedence parser – Precedence climbing method*
https://en.wikipedia.org/wiki/Operator-precedence_parser#Precedence_climbing_method

7.4 Genutzte Software

Xubuntu 19.10, ANTLR 4, JetBrains Rider, Libre Office, Compiler Explorer, C++ Reference, Firefox

Fonts: Libertinus Sans, Roboto Slab und Ubuntu Mono

7.5 Tabellenverzeichnis

7.6 Abbildungsverzeichnis

7.7 Listings

7.7.1 Vokabular des Lexers

```
lexer grammar MyllLexer;

channels { NEWLINES, COMMENTS }

COMMENT      :       (       '#!' ~('\r'|'\n')*       // ignore shebang for now
                     |       '//' ~('\r'|'\n')*
                     |       '/*' .*? '*/'
                     ) -> channel(COMMENTS);

STRING_LIT   : '"' (STR_ESC | ~('\\' | '"'  | '\r' | '\n'))* '"';
CHAR_LIT     : '\'' (CH_ESC | ~('\\' | '\'' | '\r' | '\n')) '\'';
fragment STR_ESC: '\\' ('\\' | '"'  | 'a' | 'b' | 'f' | 'n' | 't' | 'r' | 'v');
fragment CH_ESC:  '\\' ('\\' | '\'' | 'a' | 'b' | 'f' | 'n' | 't' | 'r' | 'v');

MOVE         : '(move)';
ARROW_STAR   : '->*';
POINT_STAR   : '.*';
PTR_TO_ARY   : '[*]';
COMPARE      : '<=>';
TRP_POINT    : '...';
DBL_POINT    : '..';
DBL_AMP      : '&&';
DBL_QM       : '??';
QM_COLON     : '?:';
//DBL_STAR   : '**'; // this is only supported by 2x STAR because of: var int** a
                     // which is a double pointer, not a pow
DBL_PLUS     : '++';
DBL_MINUS    : '--';
RARROW       : '->';
PHATRARROW   : '=>';
LSHIFT       : '<<';
//RSHIFT     : '>>'; // this is only supported by 2x GT because of: var v<v<int>> a;
                     // which is two templates closing, not a right shift
SCOPE        : '::';
AT_BANG      : '@!';
AT_QUEST     : '@?';
AT_PLUS      : '@+';
AT_LBRACK    : '@[';
AUTOINDEX    : '#' DIGIT;
LBRACK       : '[';
RBRACK       : ']';
LCURLY       : '{';
RCURLY       : '}';
QM_LPAREN    : '?(';
LPAREN       : '(';
```

```
RPAREN       : ')';
AT           : '@';
AMP          : '&';
STAR         : '*';
SLASH        : '/';
MOD          : '%';
PLUS         : '+';
MINUS        : '-';
SEMI         : ';';
COLON        : ':';
COMMA        : ',';
QM_POINT_STAR: '?.*';
QM_POINT     : '?.';
QM_LBRACK    : '?[';
DOT          : '.';
CROSS        : '×';
DIV          : '÷';
POINT        : '·';
EM           : '!';
TILDE        : '~';
DBL_PIPE     : '||';
PIPE         : '|';
QM           : '?';
HAT          : '^';
USCORE       : '_';

EQ           : '==';
NEQ          : '!=';
LTEQ         : '<=';
GTEQ         : '>=';
LT           : '<';
GT           : '>';

ASSIGN       : '=';
AS_POW       : '**=';
AS_MUL       : '*=';
AS_SLASH     : '/=';
AS_MOD       : '%=';
AS_DOT       : '·=';
AS_CROSS     : '×=';
AS_DIV       : '÷=';
AS_ADD       : '+=';
AS_SUB       : '-=';
AS_LSH       : '<<=';
AS_RSH       : '>>=';
AS_AND       : '&=';
AS_OR        : '|=';
AS_XOR       : '^=';

AUTO         : 'auto';
VOID         : 'void';
BOOL         : 'bool';
INT          : 'int';
UINT         : 'uint';
ISIZE        : 'isize';
USIZE        : 'usize';
BYTE         : 'byte';
CHAR         : 'char';
CODEPOINT    : 'codepoint';
STRING       : 'string';
FLOAT        : 'float';

I64          : 'i64';
I32          : 'i32';
I16          : 'i16';
I8           : 'i8';
U64          : 'u64';
U32          : 'u32';
U16          : 'u16';
U8           : 'u8';
B64          : 'b64';
B32          : 'b32';
B16          : 'b16';
B8           : 'b8';
F128         : 'f128';       // long double prec. float
F64          : 'f64';        // double prec. float
F32          : 'f32';        // single prec. float
F16          : 'f16';        // half prec. float

NS           : 'namespace';
```

```
MODULE       : 'module';
IMPORT       : 'import';
VOLATILE     : 'volatile';
STABLE       : 'stable';
CONST        : 'const';
MUTABLE      : 'mutable|mut';
PUB          : 'public'|'pub';
PRIV         : 'private'|'priv';
PROT         : 'protected'|'prot';
USING        : 'using';
ALIAS        : 'alias';
UNION        : 'union';
STRUCT       : 'struct';
CLASS        : 'class';
CTOR         : 'ctor';
COPYCTOR     : 'copyctor'|'copy_ctor'|'cctor';
MOVECTOR     : 'movector'|'move_ctor'|'mctor';
DTOR         : 'dtor';
COPYASSIGN   : 'copy=';
MOVEASSIGN   : 'move=';
FUNC         : 'func';
PROC         : 'proc';
METHOD       : 'method';
ENUM         : 'enum';
CONCEPT      : 'concept';
REQUIRES     : 'requires';
PROP         : 'prop';
GET          : 'get';
REFGET       : 'refget';
SET          : 'set';
FIELD        : 'field';
OPERATOR     : 'operator';
VAR          : 'var';
LET          : 'let';
LOOP         : 'loop';
FOR          : 'for';
DO           : 'do';
WHILE        : 'while';
TIMES        : 'times';
IF           : 'if';
ELSE         : 'else';
SWITCH       : 'switch';
DEFAULT      : 'default';
CASE         : 'case';
BREAK        : 'break';
FALL         : 'fall';
RETURN       : 'return';
SIZEOF       : 'sizeof';
NEW          : 'new';
DELETE       : 'delete';
THROW        : 'throw';

ID           : ALPHA_ ALNUM_*;

NUL          : 'null'|'nullptr';
CLASS_LIT    : 'this'|'self'|'base'|'super';
BOOL_LIT     : 'true'|'false';
FLOAT_LIT    :      (       DIGIT* '.' DIGIT+ ( [eE] [+-]? DIGIT+ )?
                    |       DIGIT+ [eE] [+-]? DIGIT+
                    ) [lfLF]?;
HEX_LIT     : '0x' HEXDIGIT HEXDIGIT_*;
OCT_LIT     : '0o' OCTDIGIT OCTDIGIT_*;
BIN_LIT     : '0b' BINDIGIT BINDIGIT_*;
INTEGER_LIT  : (DIGITNZ DIGIT_*|[0]);

fragment DIGITNZ     : [1-9];
fragment DIGIT       : [0-9];
fragment DIGIT_      : [0-9_];
fragment HEXDIGIT    : [0-9A-Fa-f];
fragment HEXDIGIT_   : [0-9A-Fa-f_];
fragment OCTDIGIT    : [0-7];
fragment OCTDIGIT_   : [0-7_];
fragment BINDIGIT    : [0-1];
fragment BINDIGIT_   : [0-1_];
fragment ALPHA_      : [A-Za-z_];
fragment ALNUM_      : [0-9A-Za-z_];

NL           : ('\r'|'\n')+  -> channel(NEWLINES);
WS           : (' '|'\t')+   -> skip;// channel(HIDDEN);
```

7.7.2 Grammatik des Parsers

```
parser grammar MyllParser;

options { tokenVocab = MyllLexer; }

comment      :       COMMENT;

// all operators handled by ToOp except mentioned
postOP       :       v=(     '++' | '--' );

// handled by ToPreOp because of collisions
preOP        :       v=(     '++' | '--' | '+' | '-' | '!' | '~' | '*' | '&' );

powOP        :               '*''*';
multOP       :       v=(     '*' | '/' | '%' | '&' | '·' | '×' | '÷' );
addOP        :       v=(     '+' | '-' | '^' | '|' ); // split xor and or?
shiftOP      :               '<<' | '>''>';

cmpOp        :               '<=>';
orderOP      :       v=(     '<=' | '>=' | '<' | '>' );
equalOP      :       v=(     '==' | '!=' );

andOP        :               '&&';
orOP         :               '||';

nulCoalOP    :               '?:';

memAccOP     :       v=(     '.'  | '?.'  | '->'  );
memAccPtrOP  :       v=(     '.*' | '?.*' | '->*' );

// handled by ToAssignOp because of collisions
assignOP     :               '=';
aggrAssignOP :       v=(     '**=' | '*=' | '/=' | '%=' | '&=' | '·='  | '×='
                       |     '÷='  | '+=' | '-=' | '|=' | '^=' | '<<=' | '>>=' );

lit          :       CLASS_LIT | HEX_LIT | OCT_LIT | BIN_LIT | INTEGER_LIT |
                     FLOAT_LIT | STRING_LIT | CHAR_LIT | BOOL_LIT | NUL;
wildId       :       AUTOINDEX | USCORE;
id           :       ID;
idOrLit      :       id | lit;

specialType  :       v=( AUTO | VOID | BOOL );
charType :           v=( CHAR | CODEPOINT | STRING );
floatingType:        v=( FLOAT | F128 | F64 | F32 | F16 ); // 80 and 96 bit?
binaryType   :       v=( BYTE | B64 | B32 | B16 | B8 );
signedIntType:       v=( INT  | ISIZE | I64 | I32 | I16 | I8 );
unsignIntType:       v=( UINT | USIZE | U64 | U32 | U16 | U8 );

qual         :       v=( CONST | MUTABLE | VOLATILE | STABLE );

typePtr      :       qual*
                     (       ptr=( DBL_AMP | AMP | STAR | PTR_TO_ARY )
                     |       ary=( AT_LBRACK | LBRACK ) expr? RBRACK )
                     suffix=( EM | PLUS | QM )?;

idTplArgs    :       id tplArgs?;

typespec     :       qual*
             (       typespecBasic    typePtr*
             |       FUNC typePtr*    typespecFunc
             |       typespecNested   typePtr*);

typespecBasic:       specialType
             |       charType
             |       floatingType
             |       binaryType
             |       signedIntType
             |       unsignIntType;

typespecFunc :       funcTypeDef (RARROW typespec)?;

typespecNested:      idTplArgs (SCOPE idTplArgs)*;
typespecsNested:     typespecNested (COMMA typespecNested)* COMMA?;

arg          :       (id COLON)? expr;
args         :       arg (COMMA arg)* COMMA?;
funcCall     :       ary=( QM_LPAREN | LPAREN )        args?    RPAREN;
```

```
indexCall    :       ary=( QM_LBRACK | LBRACK )        args     RBRACK;

param        :       typespec id?;
funcTypeDef  :       LPAREN (param (COMMA param)* COMMA?)? RPAREN;

// can't contain expr, will fck up through idTplArgs with multiple templates (e.g. op | from enums)
tplArg       :       lit | typespec;
tplArgs      :       LT tplArg        (COMMA tplArg)*  COMMA? GT;
tplParams    :       LT id            (COMMA id)*              COMMA? GT;

threeWay :   (orderOP|equalOP)        COLON    expr;

// The order here is significant, it determines the operator precedence
expr     :   (idTplArgs      SCOPE)+  idTplArgs        # ScopedExpr
             |       expr
                     (       postOP
                     |       funcCall
                     |       indexCall
                     |       memAccOP idTplArgs )      # PostExpr
             | // can not even be associative without a contained expr
                     NEW     typespec?        funcCall?               # NewExpr
             | // inherent <assoc=right>, so no need to tell ANTLR
                     (       (        MOVE // parens included
                             |        LPAREN (QM|EM)? typespec RPAREN)
                     |       SIZEOF
                     |       DELETE   ( ary='['']' )?
                     |       preOP
                     )                                                 expr     # PreExpr // this visitor
is unused
             |       expr     memAccPtrOP              expr     # MemPtrExpr
             | <assoc=right>
                     expr     powOP                    expr     # PowExpr
             |       expr     multOP                   expr     # MultExpr
             |       expr     addOP                    expr     # AddExpr
             |       expr     shiftOP                  expr     # ShiftExpr
             |       expr     cmpOp                    expr     # ComparisonExpr
             |       expr     orderOP                  expr     # RelationExpr
             |       expr     equalOP                  expr     # EqualityExpr
             |       expr     andOP                    expr     # AndExpr
             |       expr     orOP                     expr     # OrExpr
             |       expr     nulCoalOP                expr     # NullCoalesceExpr
             // TODO: cond and throw were in the same level, test if this still works fine
             | <assoc=right>
                     expr     QM expr COLON            expr     # ConditionalExpr
             | <assoc=right>
                     expr     DBL_QM threeWay+ (COLON expr)?    # ThreeWayConditionalExpr
             | <assoc=right>
                     THROW    expr                             # ThrowExpr
             |       LPAREN   expr     RPAREN                  # ParenExpr
             |       wildId                                    # WildIdExpr
             |       lit                                       # LiteralExpr
             |       idTplArgs                                 # IdTplExpr
             ;

idAccessor   :       id       (LCURLY accessorDef+ RCURLY)?    (ASSIGN expr)?;
idExpr       :       id                                        (ASSIGN expr)?;
idAccessors  :       idAccessor       (COMMA idAccessor)*      COMMA?;
idExprs      :       idExpr           (COMMA idExpr)*          COMMA?;
typedIdAcors:        typespec         idAccessors      SEMI;

attribId :   id | CONST | FALL | THROW;
attrib       :       attribId
                     (       '=' idOrLit
                     |       '(' idOrLit (COMMA idOrLit)* COMMA? ')'
                     )?;
attribBlk    :       LBRACK   attrib (COMMA attrib)* COMMA? RBRACK;

caseStmt :   CASE expr (COMMA expr)*
                     (       COLON            levStmt+ (FALL SEMI)?
                     |       LCURLY           levStmt* (FALL SEMI)? RCURLY
                     |       PHATRARROW       levStmt  (FALL SEMI)?);

initList :   COLON id funcCall (COMMA id funcCall)* COMMA?;

// is just SEMI as well in levStmt->inStmt
funcBody :   PHATRARROW expr SEMI
             |       levStmt;
accessorDef  :       attribBlk? a=( PUB | PROT | PRIV )?
                     qual* v=( GET | REFGET | SET ) funcBody;
funcDef      :       id                       tplParams?       funcTypeDef (RARROW typespec)?
```

```
                    (REQUIRES typespecsNested)?              // TODO
                    funcBody;
opDef       :       STRING_LIT      tplParams?      funcTypeDef (RARROW typespec)?
                    (REQUIRES typespecsNested)?              // TODO
                    funcBody;
// TODO: can be used in more places
condThen :   LPAREN expr RPAREN      levStmt;

// DON'T refer to these in* here, ONLY refer to lev* levels
// ns, class, enum, func, ppp, c/dtor, alias, static
inDecl      :       NS id (SCOPE id)* SEMI                   # Namespace
            |       NS id (SCOPE id)* LCURLY levDecl+ RCURLY  # Namespace
            |       v=( STRUCT | CLASS | UNION ) id tplParams?
                    (COLON          bases=typespecsNested)?
                    (REQUIRES       reqs=typespecsNested)?   // TODO
                                    LCURLY  levDecl* RCURLY  # StructDecl
            |       CONCEPT id tplParams?            // TODO
                    (COLON  typespecsNested)?
                                    LCURLY  levDecl* RCURLY  # ConceptDecl
            // TODO aspect
            |       ENUM    id
                    (COLON  bases=typespecBasic)?
                    LCURLY  idExprs         RCURLY                  # EnumDecl
            |       v=( FUNC | PROC | METHOD )
                    (       LCURLY  funcDef* RCURLY
                    |                       funcDef                         )       # FunctionDecl
            |       OPERATOR
                    (       LCURLY  opDef*          RCURLY
                    |                       opDef                           )       # OpDecl
// class only:
            |       v=( PUB | PROT | PRIV ) COLON            # AccessMod
            |       v=( CTOR | COPYCTOR | MOVECTOR )
                    funcTypeDef     initList?       (SEMI | levStmt) # CtorDecl
            |       DTOR    LPAREN RPAREN   (SEMI | levStmt) # DtorDecl
            ;

// using, var, const: these are both Stmt and Decl
inAnyStmt   :       USING                   typespecsNested  SEMI    # Using
            |       ALIAS id ASSIGN  typespec                SEMI    # AliasDecl
            |       v=( VAR | FIELD | CONST | LET )
                    (       LCURLY  typedIdAcors*   RCURLY
                    |               typedIdAcors             )       # VariableDecl


inStmt      :       SEMI                                             # EmptyStmt
            |       LCURLY  levStmt* RCURLY                          # BlockStmt
            |       RETURN  expr?           SEMI                     # ReturnStmt
            |       THROW   expr            SEMI                     # ThrowStmt
            |       BREAK   INTEGER_LIT?    SEMI                     # BreakStmt
            |       IF                      condThen
                    (ELSE IF condThen)*     // helps with formatting properly and de-nesting the AST
                    (ELSE           levStmt)?                        # IfStmt
            |       SWITCH  LPAREN expr RPAREN       LCURLY
                    caseStmt+       (DEFAULT levStmt+)? RCURLY       # SwitchStmt
            |       LOOP    levStmt                                  # LoopStmt
            |       FOR LPAREN levStmt      // TODO: add the syntax: for( a : b )
                            expr SEMI
                            expr RPAREN     levStmt
                    (ELSE                   levStmt)?                # ForStmt
            |       WHILE           condThen
                    (ELSE           levStmt)?                        # WhileStmt
            |       DO              levStmt
                    WHILE   LPAREN expr RPAREN                       # DoWhileStmt
            |       DO expr TIMES           id?     levStmt          # TimesStmt
            |       expr    (assignOP               expr)+ SEMI      # MultiAssignStmt
            |       expr    aggrAssignOP    expr SEMI                # AggrAssignStmt
            |       expr SEMI                                        # ExpressionStmt
            ;

// ONLY refer to these lev* levels, NOT the in*
levDecl     :       attribBlk       LCURLY  levDecl+ RCURLY  # AttribDeclBlock
            |       attribBlk?      ( inAnyStmt | inDecl )   # AttribDecl;
levStmt     :       attribBlk?      ( inAnyStmt | inStmt )   # AttribStmt;

module      :       MODULE id SEMI;
imports     :       IMPORT id (COMMA id)* COMMA? SEMI;

prog    :   module? imports* levDecl+;
```

8 Eigenständigkeitserklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig angefertigt und mich fremder Hilfe nicht bedient habe.

Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Quellen entnommen sind, habe ich als solche kenntlich gemacht.

Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift