Hochschule Bremen

Bachelorarbeit

Thesis

Konzeption und Implementierung einer Makrosprache in C++

Autor:
Roland Jäger
360 956

1. Mai 2016

Inhaltsverzeichnis

Allgemeines

Eidesstattliche Erklärung

Ich, Roland Jäger, Matrikel-Nr. 360 956, versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

Konzeption und Implementierung einer Makrosprache in C++

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bremen, den 1. Mai 2016 _____

Roland Jäger

Danksagung

Ich möchte mich bei meinen Eltern bedanken, die mich immer unterstützt haben und dafür sorgten, dass ich nicht in der Sonderschule endete! Meine Freunde und alle Menschen, die mich zu diesem Studium geführt haben, verdienen auch mehr als meinen Dank – sonst wäre ich wohl Bauer geworden. Ein großes Dankeschön gilt auch meinen Professoren und meinem Betreuer bei der P3-Group, die mich diese Arbeit schreiben ließen, die aus unterschiedlichen Gründen ungewöhnlich ist.

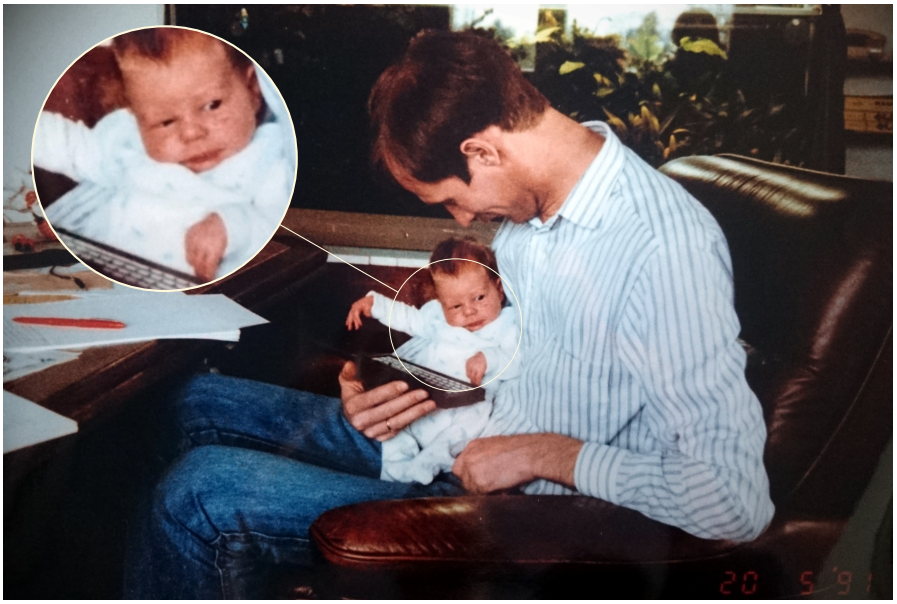

Abbildung 1: *"Das soll meine Zukunft sein?"*

Zuletzt bin ich aber vor allem dankbar dafür, dass ich in einem Land lebe, welches momentan weder von Krieg, Anschlägen, Korruption oder Hungersnot geplagt ist.

1 Einleitung

Die Einführung von Automatisierung in ein Softwaresystem ist vergleichbar mit den Maschinen, die in der industriellen Revolution auftauchten. Anstelle, dass Menschen arbeiten müssen, um ein gewünschtes Ergebnis zu bekommen, drücken sie auf einen Knopf, und ein anderes System nimmt ihnen die aufwändige Arbeit ab. Dies führt dazu, dass Produkte schneller, mit weniger Arbeitsaufwand erstellt werden können. Zudem ist die entstehende Qualität immer auf einem gleichbleibenden Level und hängt nicht von dem Befinden der Arbeiter ab.

Die P3-group arbeitet mit Airbus, um Lösungen für den Flugzeugbau zu entwickeln. Dieser Markt ist hart umkämpft, wodurch minimale Gewinne einen großen Unterschied machen können. Ein Feld, welches seit Jahren immer weiter durch wissenschaftliche und technische Durchbrüche optimiert wird, sind die menschlichen Ressourcen. Automatisierung sorgt dafür, dass sich wiederholende Arbeitsabläufe – aus der Sicht des Nutzers – zu einem einzigen Schritt werden und so Zeit sparen.

Makros sind die Fließbänder der digitalen Welt, und diese Arbeit beschäftigt sich mit der Entwicklung eines Makro-Systems bzw. -Sprache.

Makros werden durch die Verbindung kleinerer Bausteine (Anweisungen) erstellt. Diese können andere Makros oder Anweisungen, die die Anwendungsumgebung bereitstellt sein. Dies ist mit einem Fließband in der Autoindustrie zu vergleichen. Jede Station ist genau für eine Aufgabe zuständig und kümmert sich um nichts anderes.

Die Makrosprache ist ein Baukasten, mit dem Makros erstellt – "Fließbänder" für spezielle Aufgaben erzeugt – werden können.

1.1 Problemfeld

Softwaresysteme haben oft das Problem, dass sie mit einigen zentralen Features anfangen, die fest definiert werden (sollten), bevor ein Vertrag geschlossen wird. Für weitere Funktionalität, die über die Vereinbarungen im Vertrag hinausgehen, muss der Vertrag erweitert werden. Wenn der Vertrag erfüllt ist, und im Anschluss weitere Wünsche aufkommen, muss ein weiterer Vertrag aufgesetzt werden und die vorher gelieferte Software muss angepasst, gegebenenfalls erweitert werden. Dies kann zur Folge haben, dass große Teile der Software umgeschrieben werden müssen oder sogar, dass die Architektur der gesamten Anwendung verändert werden muss.

Wenn frühzeitig ein Makrosystem/-sprache und ein entsprechendes Erweiterungskonzept für Module bzw. Plugins eingeführt wird, ist die Wahrscheinlichkeit, dass der Kern der Applikation für Erweiterungen angefasst werden muss, wesentlich geringer. Durch diese Kombination kann einfach ein weiteres Modul geladen werden, welches die neuen Grundbausteine der Applikation hinzufügt. Diese können dann in einem neuen oder angepassten Makro genutzt werden, um den Wunsch der Kunden zu erfüllen. Im Falle, dass es keiner neuen Grundbausteine bedarf, reicht es sogar, nur ein Makro zu liefern. Die Vorteile dieser Methode sind, dass – wenn man davon ausgeht, dass die benutzen Makros und Grundbausteine fehlerfrei durch ausreichendes Testen der Software sind – keine neuen

Bugs in den Kern der Software eingeführt werden können und somit immens zu der Stabilität der Software beigetragen wird. Ein weiterer Vorteil ist, dass die Makros mit wesentlich weniger Aufwand entwickelt werden können, weil sie sich auf einem höheren Level befinden. Für Kunden ist eine nutzbare Makrosprache auch interessant, weil sie zum Teil durch das hausinterne Personal Anforderungen an die Software realisieren können, ohne den langen Weg über eine Firma zu gehen. Dies bedeutet auch, dass die Software eine bessere Chance hat, die Zeit zu überdauern.

1.2 Vorhandene Lösungen

ChaiScript [6] ist eine der einzigen Scriptsprachen, die es direkt für C++ gibt. Das Projekt ist ungefähr 7 Jahre alt und ermöglicht C++ Programmierern, Funktionen und Daten über Text (das Script) aufzurufen bzw. anzulegen.

ChaiScript bietet von Hause aus viele Funktionalitäten an, die weit über den Bedarf für eine Makrosprache hinausgehen. Man kann in dem Script eigene "Klassen" anlegen, Exceptions werfen und fangen, Funktionen und Variablen anlegen und C++ Funktionen mit den Variablen aus dem Script aufrufen. Die Vielzahl an Funktionen ist auch der Grund, aus dem sich gegen die Nutzung von ChaiScript entschieden wurde. Da beliebige C++ Funktionen[1] aufgerufen werden können, bietet dies eine unangenehm große Fehlerquelle, wenn diese Funktionen den Zustand der Applikation verändern, ohne dass diese rückgängig gemacht werden können.

Zudem ist die Quellcodequalität von ChaiScript fraglich, weil Teile des Kontrollflusses von Exceptions übernommen werden. Das ist nicht nur schwer nachzuvollziehen, sondern auch extrem langsam, wenn man es mit dem Zustand eines Objektes oder Rückgabewerten vergleicht.

1.3 Ziele der Arbeit

Die Ziele der Arbeit sind es ein Makrosystem zu entwickeln, welches ...

- auf keinem festen *Application Programming Interface (API)* aufbaut.

  Ein Feature der bestehenden Software ist es, dass sie durch *Module*[2] erweitert werden kann. Eines dieser Module wird das Makrosystem sein, welches in dieser Arbeit entwickelt wird. Durch diese Modularität, gibt es kein festes Interface1.

- nicht nur Anweisungen abarbeitet.

  Um eine große Bandbreite an Automatisierungsmöglichkeiten anbieten zu können, bedarf es logischer Ausdrücke, die bedingte Anweisungen erlauben. Ebenso ist es wichtig, dass man entscheiden kann, wie oft etwas ausgeführt werden soll, sprich

[1] Funktionen aus der C++ Ebene müssen mit dem Interpreter registriert werden, bevor dieser sie aufrufen kann.

[2] Bibliotheken, die zur Laufzeit – nach den dynamischen Bibliotheken – nachgeladen werden können, um die Funktionalität der Applikation zu erweitern.

Schleifen. Außerdem sollen die Makros sowohl von dem Programm, als auch von anderen Makros Parameter übergeben bekommen können.

- nicht mehr kann, als es können muss.

  Je mächtiger ein System ist, desto komplexer ist es. Zudem sind Features nur etwas wert, wenn sie gewinnbringend verkauft werden können.

- gut wartbar ist.

  Die Implementierung sollte keine komplexen Bestandteile besitzen, über die nicht geschlussfolgert werden kann.

- benutzerfreundlich ist.

  Die Lösung soll benutzerfreundlich sein. Das heißt, dass Fehlermeldungen den Nutzer schnell zu seinem Fehler führen und keine false positives enthalten.

1.4 Hintergründe und Entstehung des Themas

Die P3-group ist daran interessiert, dass sie ihren Kunden Lösungen schnell und in hoher Qualität anbieten kann. Um dies zu erreichen, arbeiten sie daran, dass alle Softwaresysteme, die von ihr angeboten werden, Automatisierung über Makros unterstützen. Wirtschaftlich rentieren sich die Makros dadurch, dass sie von den Firmen gemietet und nicht nur einmal verkauft werden. Zum Beispiel, werden alle Flugzeugteile ausgewählt und dann deren Gewicht ermittelt. Ein anderes Mal sollen nur spezielle Teile aus einem bestimmten Werkstoff zusammengezählt und deren Preis ermittelt werden. Statt eine komplexe Suchfunktion zu entwickeln, würde es genügen, zwei Makros zum Einsatz zu bringen, die jeweils eine Aufgabe erfüllen und somit für den Benutzer sicher zu handhaben sind. Die Komplexität der Software würde durch das zweite Makro nicht sonderlich beeinflusst, weil dieses intern auf Funktionalität zurückgreifen kann, die schon vom ersten genutzt wird.

1.5 Struktur der Arbeit, wesentliche Inhalte der Kapitel

Die Arbeit ist in drei wesentliche Kapitel aufgeteilt, Kapitel 2 Anforderungsanalyse, Kapitel 3 Konzeption und Kapitel 4 Exemplarische Realisierung. Der Fokus dieser Kapitel geht vom Theoretischen zum Praktischen. Innerlich folgen die Kapitel den Arbeitsabläufen, die zur Entwicklung des Makrosystems genutzt wurden. Zudem gibt es dieses Einleitungskapitel, eine Evaluation und einen Ausblick.

In dem Kapitel Anforderungsanalyse werden die Anforderungen, sowie deren Probleme analysiert. Das Kapitel Konzeption beschäftigt sich mit den Lösungen für die Anforderungen sowie der Probleme, die im vorherigen Kapitel gefunden wurden. Unter anderem beinhaltet das Kapitel die Software Architektur, sowie den Syntax für die Makrosprache. In dem Kapitel Exemplarische Realisierung wird auf entscheidende Punkte der exemplarischen Realisierung eingegangen.

2 Anforderungsanalyse

Dieses Kapitel beschäftigt sich mit dem Problemfeld und den Anforderungen an die entstehende Lösung. In der Diskussion des Problemfeldes geht es vor allem darum, das Problemfeld zu analysieren und die Unterprobleme ausfindig zu machen, um abstrakte Lösungsansätze für diese zu entwickeln. Bei den Anforderungen an die angestrebte Lösung ist das Ziel, die Anforderungen, die die Problemlösung erfüllen sollte, zu definieren.

2.1 Diskussion des Problemfeldes

Ein Makrosystem ist eine Komponente eines Softwaresystems, welche es erlaubt, die Software über eine Reihenfolge von Zeichen so zu steuern, als ob ein Mensch die Applikation bedient hätte.

2.1.1 Was ist ein Makrosystem?

Die Funktionalität eines Makrosystems ist vergleichbar mit Programmiersprachen – dort wird durch eine Ansammlung von Zeichen der Computer veranlasst, eine bestimmte Abfolge von Hardwareanweisungen auszuführen. Der Unterschied von einem Makrosystem zum Beispiel zu C ist, dass bei einem Makrosystem der Befehlssatz durch die Anwendung vorgegeben wird, wohingegen der Befehlssatz von C durch die Hardware vorgegeben wird und nicht durch eine weitere Ebene übersetzt werden muss (C ist eine native Programmiersprache). Somit ist ein Makrosystem eher mit einer Sprache zu vergleichen, die sich einer *virtuellen Maschine* (VM) bedient – wie Java – als mit C.

Bei Java gibt die VM den Befehlssatz vor und muss bei der Ausführung des Programms diese Befehle in die entsprechenden Hardwarebefehle, übersetzen (write once, run anywhere). Ähnlich verhält es sich mit dem Makrosystem. Das Makro nutzt die vorhandenen Befehle der Applikation, um einen Arbeitsvorgang zu automatisieren. Da die Befehle jedoch nur als Zeichenketten vorliegen, müssen diese zu den richtigen Funktionen übersetzt werden. Der Befehlssatz dieses Makrosystems kommt aus dem vorhandenen System, weil dort das Command-Pattern [13, S.263] eingesetzt wird. Somit bietet es ein ideales Interface für Automatisierung. Unterabschnitt 2.1.3 geht auf die Architektur des vorhandenen Systems ein, die für diese Arbeit wichtig ist.

Für alle Programmiersprachen ist es vonnöten, die Reihenfolge von Zeichen in sinnvolle Stücke zu zerteilen – dies übernimmt ein Tokenizer, siehe Abbildung 2. Meist wird dieser in den Parser [11, S.46] integriert, der die Stücke der Zeichenkette (String) in ein Format überführt, welches vom Interpreter unterstützt wird. Das Format ist meist ein *abstrakter Syntaxbaum*[3] (AST), welcher den String eindeutig repräsentiert. Ein spezieller Typ aus dem AST nennt sich Literal. Literals sind Daten, die der Programmierer durch Quelltext erstellen kann (zum Beispiel ist `"foo"` ein String und `1.1` ist eine Dezimalzahl). Literals müssen aus dem Format des Quelltextes in die echte Datenstruktur verwandelt werden, was ein Lexer übernimmt. Lexer sind wie Tokenizer meist ein Teil des Parsers. Der

[3] `Abstract syntax tree` ist eine digitale Darstellung einer Programmiersprache.

Interpreter arbeitet dann nur noch mit dem AST, welcher vorgibt, welche Befehle der Interpreter ausführen muss, um das als String angegebene Programm auszuführen. Ein Interpreter arbeitet die AST Elemente sequentiell ab. Das heißt, dass ein Interpreter `a` ausführt, ohne zu 'wissen', dass er im Anschluss `b` ausführen wird.

Abbildung 2: Abstraktes Ziel der resultierenden Architektur

Anstelle eines Interpreters wird bei C ein Compiler genutzt, der vor der Ausführung das gesamte Programm in Maschinencode verwandelt. Dies hat den Vorteil, dass während der Ausführung nur das Programm ausgeführt wird und nicht noch eine weitere Komponente (Interpreter) die CPU in Anspruch nimmt. Ein Kompromiss zwischen beiden Welten ist ein *just in time compiler* (JIT), dieser probiert das Beste aus beiden Welten, die Dynamik vom Interpretieren und die Geschwindigkeit von kompilierten Programmen, zu vereinen.

2.1.2 Parser

Einer der zeitaufwändigsten Schritte ist es, die Quelltext Zeichenkette in einen AST umzuwandeln. Dieser Schritt wird vom Parser übernommen, von dem es drei Hauptgruppen gibt.

- Links nach rechts, links Auflösung (LL) [11, S.77 f.]
  LL Parser arbeiten 'vorwärts', links nach rechts und probieren auf der linken Seite des Gelesenen (teils geparsten Quelltextes), zu reduzieren. Man gelangt zum Schluss an das Ende des Baumes.
- Links nach rechts, rechts Auflösung (LR) [11, S.77 f.]
  LR Parser arbeiten 'rückwärts', links nach rechts und probieren auf der rechten Seite des ungelesenen Quelltextes, zu reduzieren, um Terminals auf der linken Seite zu sammeln. Man gelangt zum Schluss an den Anfang des Baumes. [15]
- Rekursiv Absteigend (Recursive-Descent)
  Recursive-Descent Parser arbeiten wie LR Parser 'rückwärts' und gelangen am Ende auch am Anfang des Baumes an. Im Gegensatz zu LL und LR Parsern arbeiten sie nicht mit Zustandstabellen sondern mit Rekursion. Für jedes Konstrukt, welches geparst werden soll, gibt es eine Methode. In dieser werden all die Methoden aufgerufen, die ein Element produzieren, welches sich in dem der äußeren Methode befinden darf.

2.1.3 Das vorhandene System

Das vorhandene Softwaresystem ist eine C++ Anwendung, die mit Hilfe von Modulen ihre Funktionalität erweitern kann. Ein Teil des zentralen Herzstückes ist eine Implementierung des Command-Patterns, siehe Abbildung 3.

Der `Receiver` stellt alle Objekte dar, die das `ConcreteCommand` braucht, um seine Aufgabe zu erfüllen und ist – so wie der `Client`, der die Objekte darstellt, die ein `Command` ausführen – nicht Teil der Architektur. `Client`s sind über den `CommandProvider` in der Lage einen `Invoker` zu bekommen, der ein konkretes `Command` wrapped. Der `Invoker` bekommt bei der Erzeugung über den Constructor den momentanen `HistoryStack`. Wenn der `Invoker` durch einen Aufruf aus einem `Command` (auf den `CommandProvider`) entstanden ist, ist es der `HistoryStack` des `Command`s. Ansonsten ist es der `HistoryStack` des `CommandProvider`s. Durch den `HistoryStack` ist es möglich die Veränderungen von ausgeführten `Command`s rückgängig zu machen oder wiederherzustellen. Dies wird zum Beispiel von dem `Invoker` genutzt, wenn das aufgerufene `Command` einen Fehler verursacht. An dem `CommandProvider` können sich alle `Command`s registrieren, die in anderen Teilen der Anwendung genutzt werden können.

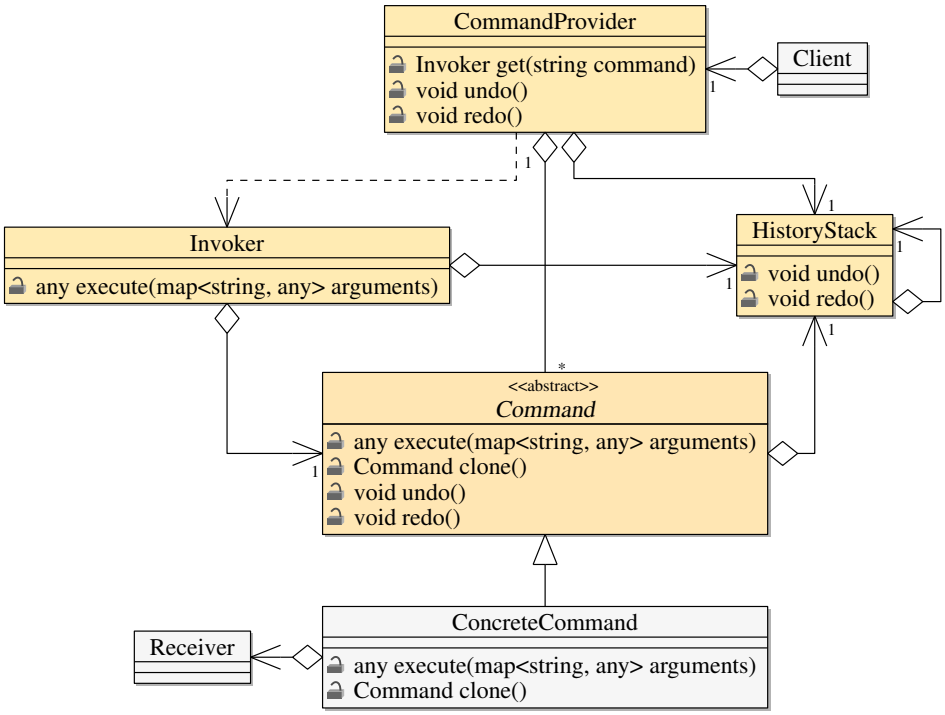

Abbildung 3: Abstrakte Command-Patter Implementierung

Da C++ keine Reflexion [22, 12] unterstützt, ist die Signatur der `any execute(map<string, any> arguments)` Methode ein wichtiger Punkt der Architektur. Durch den `any` [3] Typ, in der Implementierung des Command-Patterns, wurde die fehlende Reflektion umgangen[4]. `any` ist in der Lage, jeglichen Typ aufzunehmen und typsicher[5] aufzubewahren bzw. weiterzugeben. Diese Funktionalität, kombiniert mit der `map<T1, T2>`, erlaubt es, beliebig viele Parameter mit beliebigen Typsignaturen als Parameter für das `Command` zu nutzen, ohne dass die Funktionssignatur angepasst werden muss.

[4] Diese Veränderung ist als Vorbereitung auf die Bachelorarbeit entstanden, weil diese Anpassung den Zeitrahmen der Bachelorarbeit überschritten hätte.

[5] Typsicher bedeutet, dass ein `int` nicht zu einem `char` gecastet werden kann – es bedeutet nicht, dass `any` implizit zu `int` gecastet wird – es ist also mit einem `void *` zu vergleichen.

Der Wert von einer `any` Variable kann entweder invalid (keine Daten) oder valid (Daten) sein. Es ist vergleichbar mit den Variablen von Python. Dies provoziert Fehler, wenn die `ConcreteCommand`s aufgerufen werden und nicht mit dem erwarteten Typ übereinstimmt. Allerdings ist dieses Problem durch eine Prüfung, ob die richtigen Typen in den `any` Parametern stecken, minimiert. Die Prüfung findet in dem `Invoker` statt, bevor das `ConcreteCommand` mit den Parametern aufgerufen wird.

2.2 Anforderungen an die angestrebte Lösung

Die Probleme, ein Makrosystem/-sprache zu implementieren, fangen dann an, wenn man von den Makros will, dass die `Command`s nicht nur hintereinander abgearbeitet werden. Also ohne dass sie wissen, dass andere `Command`s vor bzw. nach ihnen ausgeführt werden. Dieser Ablauf ist in Abbildung 4 zu sehen.

Abbildung 4: Sequenzielles Abarbeiten von Prozessschritten

Abbildung 5 zeigt den ersten Schritt zu einer nützlichen Implementierung – Logik. Hierbei bietet man an, dass der Makro-Entwickler anhand von Rückgabewerten aus `Command`s entscheiden kann, welche weiteren `Command`s er ausführen möchte.

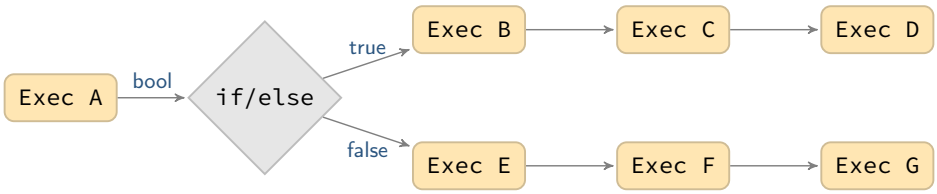

Abbildung 5: Logische Ausdrücke, um bedingte Anweisungen zuzulassen

Obwohl man mit solchen Makros schon einige Probleme lösen kann, ist es nicht das, was man zur Verfügung haben will, wenn man mit Datenstrukturen arbeitet, die normalerweise an Funktionen und Objekte geben werden, um sie zu modifizieren. Somit kommt in diesem Schritt hinzu, dass es Schleifen sowie komplexe Parameter und Rückgabewerte geben kann, siehe Abbildung 6.

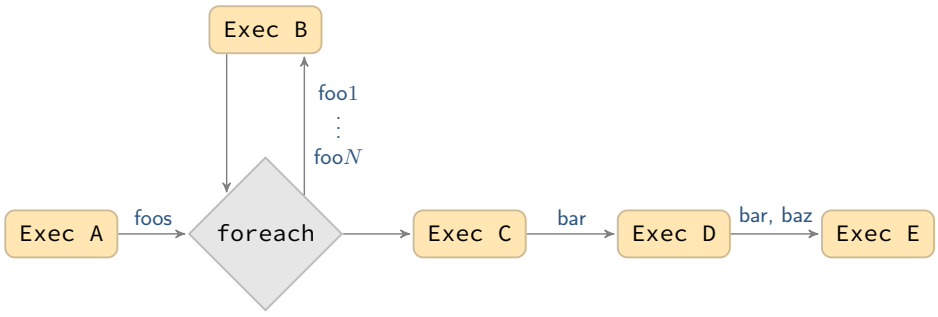

Abbildung 6: Schleife, die Anweisungen für ein Element aus der Liste aufrufen

Letztendlich kann man sagen, dass ein solches Makrosystem/-sprache eine Programmiersprache mit Interpreter [13, S.274] sein sollte, deren Laufzeitumgebung ein anderes Softwaresystem ist.

Fehlermeldungen Die Fehlermeldungen, die bei Syntaxfehlern auftreten, sollten dem Nutzer möglichst viele sinnvolle Informationen liefern – als Vorbild dient hier Clang (siehe Listing 1).

```
main.cpp:4:42: error: expected ';' after expression
  std::cout << "Hallo Welt!" << std::endl
                                         ^
                                         ;
```

Listing 1: Clang Fehlermeldung

3 Konzeption

Dieses Kapitel beschäftigt sich mit der theoretischen Problemlösung.

Unterabschnitt 3.1 Syntax beschreibt die Syntax, welche von dem Tokenizer und Parser umgewandelt werden soll. Im Unterabschnitt 3.2 Grundarchitektur wird die Grundarchitektur des Makrosystems beschrieben. Diese Grundarchitektur umfasst nur die grobe Architektur des Makrosystems, weil in dem Unterabschnitt 3.3 Detaillierte Teilarchitekturen auf die komplexeren Teile der Architektur eingegangen wird.

3.1 Syntax

Die Syntax ist an C [4], Python [17], JavaScript [9] und Swift [20] angelegt. C liefert den größten Anteil der Syntax, von Python wurde `def` übernommen, von JavaScript `var` und die *Named-Parameter*[6] Syntax `fun(foo:gun());` von Swift. Die Unterscheidung zwischen `def` und `var` sorgt dafür, dass die Programmierer nach den ersten drei Zeichen wissen, was der folgende Code ausführen wird. Die Makrosprache unterstützt Named-Parameter, weil das Command-Pattern so implementiert wurde, dass Parameter Aliasbezeichnungen benutzen können, um eine hohe Kompatibilität zwischen unabhängig entwickelten Modulen zu gewährleisten. Dies ist in der Makrosprache nur schwer möglich. Somit sind die Named-Parameter der bestmögliche Kompromiss, weil diese erlauben, die Parameter in beliebiger Reihenfolge anzugeben, was durch die Nutzung von `map<T1, T2>` erforderlich ist.

In den folgenden Abschnitten wird die Syntax der Makrosprache mit Hilfe von Railroad Diagrammen und regulären Ausdrücken erklärt. In den Railroad Diagrammen sind die grauen Terminal- und die orangenen Non-Terminal- Elemente.

3.1.1 Syntax Grundlagen

Bezeichner Die Bezeichner müssen dem regulären Ausdruck aus Abbildung 7 entsprechen.

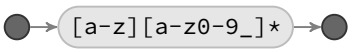

Abbildung 7: Regulärer Ausdruck von Bezeichnern

Das heißt, dass Bezeichner nur aus kleinen Buchstaben, Nummern und Unterstrichen bestehen können und am Anfang einen Buchstaben haben müssen. Grund für diese drastische Einschränkung ist, dass der Code einheitlich aussehen soll (die erste Regel, was Bezeichnungen/Formatierung angeht ist, dass man sich an dem orientiert, was schon existiert). Um dies besser garantieren zu können, wurde die CamelCase Schreibweise von vornherein ausgeschlossen. Außerdem sind Bezeichner, die einem Schlüsselwort (`break`,

[6] Named-Parameter sind Parameter, die über einen Namen ihren Wert beim Funktionsaufruf zugewiesen bekommen. Die normale Wertzuweisungsstrategie ist, nach der Reihenfolge der Deklaration vorzugehen.

continue, **def**, **do**, **else**, **for**, **if**, **print**, **return**, **typeof**, **var**, **while**), einen Booleanwert (`true`, `false`) oder `main` entsprechen – abgesehen von der einen `main` Methode – verboten.

Literals Literals sind entweder Doubles, Integer, Strings oder Boolean Werte, wie Abbildung 8 zeigt. In Strings ist es möglich, besondere Zeichen zu escapen. Zum Beispiel kann ein Zeilenumbruch – wie in anderen Programmiersprachen – mit `"\n"` oder ein Tab mit `"\t"`, erzeugt werden.

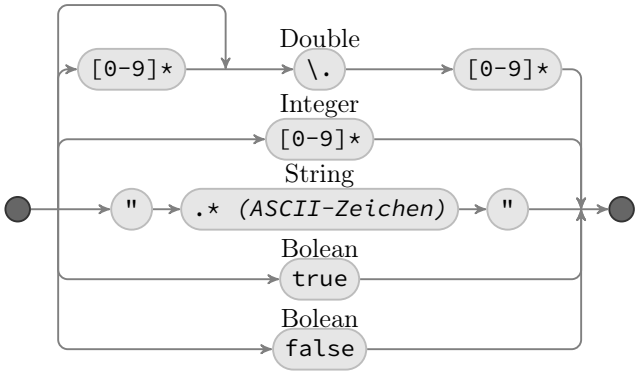

Abbildung 8: Syntax von Literals

Werterzeuger Abbildung 9 zeigt alle Werterzeuger. Das sind Konstrukte, die einen Wert für eine andere Operation bereitstellen.

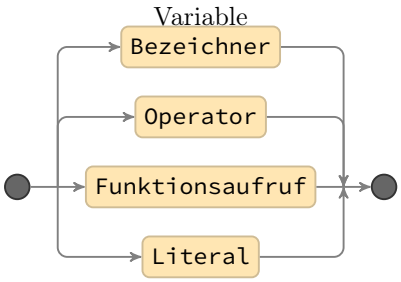

Abbildung 9: Syntax von Werterzeugern

Scopes Abbildungen 10, 11 und 12 zeigen die Syntax von Scopes. Scopes sind Bestandteile von Funktionen und Kontrollstrukturen, wobei sich Loop Scopes von normalen Scopes darin unterscheiden, dass sie die **break** und **continue** Schlüsselwörter unterstützen. Alle Scopes – abgesehen von Funktionsdeklarationsscopes – die sich in einem Loop Scope befinden, sind automatisch Loop Scopes. Scopes verhalten sich wie C Scopes, was bedeutet, dass die Syntax `var foo; {var foo;}` richtig ist – das erste `foo`, wird von dem zweiten verdeckt.

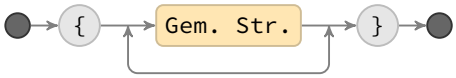

Abbildung 10: Syntax vom Scope

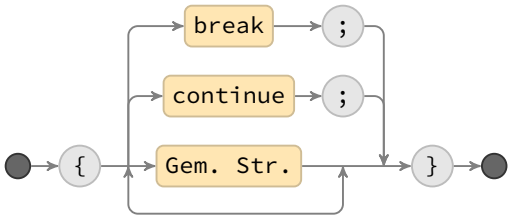

Abbildung 11: Syntax vom Loop Scope

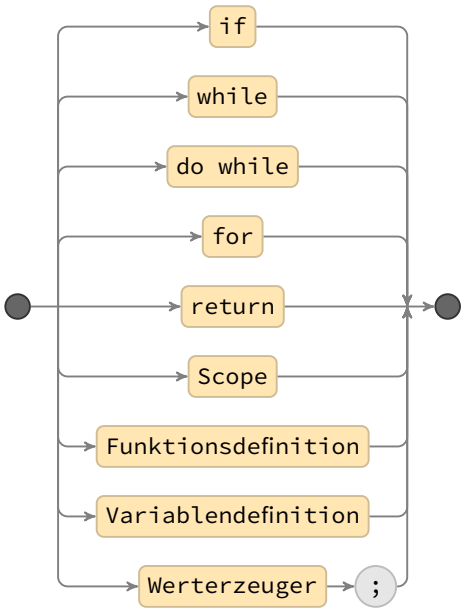

Abbildung 12: Syntax von gemeinsamen Strukturen der Scopes

3.1.2 Kommentar Syntax

Es werden zwei Arten von Kommentaren unterstützt. Die Syntax aus Abbildung 13 sorgt dafür, dass alle folgenden Zeichen bis zu einem Zeilenumbruch als Kommentar angesehen werden. Der Kommentar, der mit Abbildung 14 erstellt wird, sorgt dafür, dass nach dem Kommentar weiterer Code folgen kann, der nicht zu dem Kommentar gehören soll. Diese Art von Kommentar ermöglicht auch, Kommentare über mehrere Zeilen spannen zu können. Kommentare können nach allen Elementen eingefügt werden, die nicht explizit ein anderes Element nach sich benötigen.

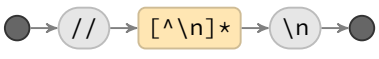

Abbildung 13: Syntax vom Zeilenkommentaren

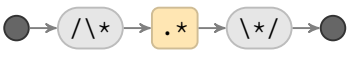

Abbildung 14: Syntax vom Kommentaren in Zeilen

3.1.3 Define Syntax

Variablen Wie in Abbildung 15 zu sehen ist, können Variablen folgendermaßen definiert werden: `var foo;`. Um anschließend der Variablen einen Wert zuzuweisen, ist unter ande-

rem folgendes erlaubt: `var foo = fun();` oder `var foo = true == false;`. Diese Syntax erlaubt es nicht, den Variablen einen Typ zuzuweisen – dieser Teil der Sprache ähnelt deswegen Python und JavaScript sehr.

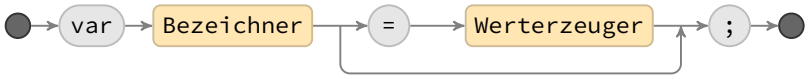

Abbildung 15: Syntax von Variablendeklaration

Funktionen Über `def fun(){...}` können Funktionen definiert werden, siehe Abbildung 16. Um eine parametrisierte Funktion zu definieren, gibt man die Parameternamen, Kommata getrennt, nach dem Funktionsnamen an: `def fun(foo, bar){...}` (`def fun(bar, foo){...}` definiert die gleiche Funktion und würde zu einem Fehler führen, wenn beide Funktionen in demselben Scope definiert werden). Zudem ist es nicht erlaubt, ein Whitespace oder Kommentar zwischen dem Bezeichner und der Klammer zu haben.

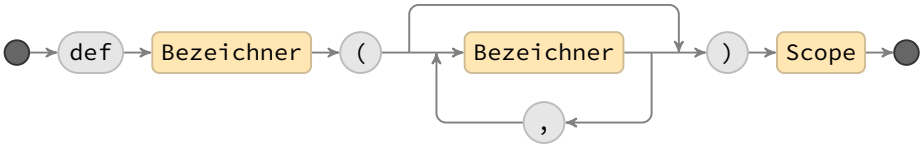

Abbildung 16: Syntax von Funktionsdeklaration

Der Einstiegspunkt eines jeden Makros ist eine `def main(){...}` Funktion. Die Syntax ist dieselbe, wie bei den normalen Funktionen und erlaubt es daher, auch Parameter anzugeben. Somit können Makros auch aus anderen Makros oder aus der C++ Ebene über eine äquivalente Syntax mit Parametern aufgerufen werden.

3.1.4 Return Syntax

In Abbildung 17 ist die Syntax von `return` zu sehen.

Abbildung 17: Syntax von return

3.1.5 Kontrollstrukturen Syntax

Die Syntax von `if`/`else`, `do`-/`while` und `for` aus den Abbildungen 18, 19, 20 und 21 sollten wie erwartet aussehen.

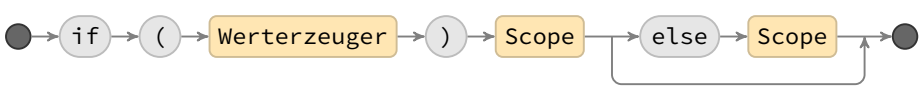

Abbildung 18: Syntax von if

Abbildung 19: Syntax von while

Abbildung 20: Syntax von do-while

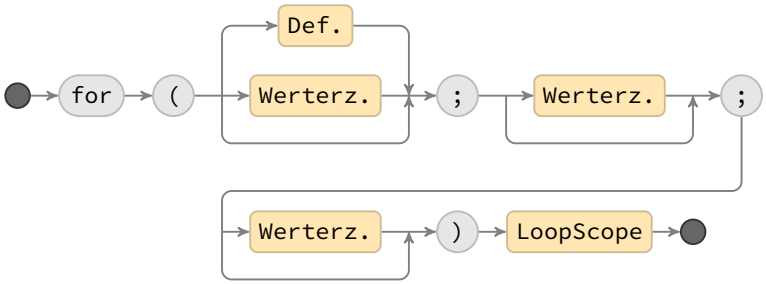

Abbildung 21: Syntax von for

3.1.6 Befehlssyntax

Funkionsaufrufe Abbildung 22 zeigt die Syntax, um eine definierte Funktion aufzurufen. `fun(foo:gun(), bar:foo);` weist dem `foo` Parameter den Wert von `gun()` zu und dem Parameter `bar` wird der Wert von `foo` aus dem Scope zugewiesen. Wie auch bei der Definition der Funktion ist es nicht erlaubt, Whitespace Zeichen oder Kommentare zwischen dem Bezeichner und der Klammer zu haben.

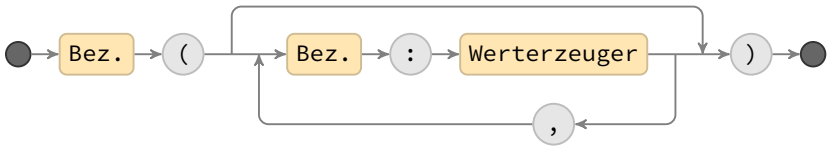

Abbildung 22: Syntax von Funktionsaufrufen

Operatoren Die Operator Syntax aus Abbildung 23 folgt wie die **`return`** Syntax, den Vorbildern dieser Syntax (C, Python, Javascript, Swift).

Geklammerte Ausdrücke werden zuerst vollständig ausgewertet, bevor der Operator angewendet wird. Das heißt, dass `(a || b) && c` in folgender Weise interpretiert wird.: Als erstes wird `a` ausgewertet. Wenn a `false` war, wird `b` ausgewertet, falls einer der beiden `true` ergibt, wird `c` ausgewertet. Entgegen dessen wird bei `a || b && c` zuerst `a`, und wenn `a` `false` war, `b` und `c` ausgewertet. Es wird also von links nach rechts ausgewertet, und es gibt eine Kurzschluss-Semantik [7, S.212]. Die Präzedenz der Operatoren ist – abgesehen von **`print`** (der vor `=` kommt) – aus der Reihenfolge in der Grafik zu entnehmen. Dabei ist zu beachten, das die `+` und `-` Operatoren zweimal vorkommen – unär und binär.

Zu den 'normalen' Operatoren aus C gibt es zudem noch den **`typeof`** Operator aus JavaScript. Dieser Operator wandelt den Typ der Variable in einen `string` um (`1`→`"int"`). Zudem gibt es den **`print`** Operator, der den Wert einer Variablen ausgibt (z.B. auf die Konsole) und als `string` zurückgibt. Ein weiterer Operator, der nur implizit genutzt wird, ist der **`bool`** Operator. Dieser wird angewendet, wenn ein boolean Wert benötigt wird, ein Beispiel sei **`if`**`(2.2){...}`, hierbei wird aus `2.2` `true`, da `2.2` größer ist als `0`.

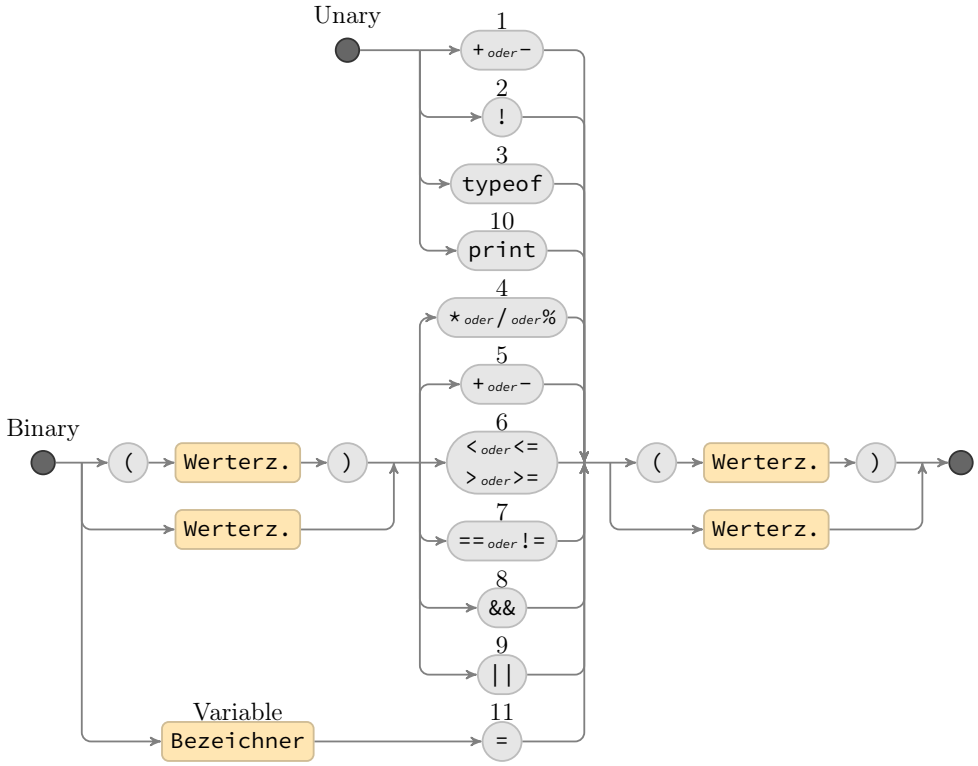

Abbildung 23: Syntax von Operatoren

3.2 Grundarchitektur

Da das komplette UML Diagramm sehr unübersichtlich ist und nicht auf ein A2 Blatt passt, sind die folgenden Diagramme Ausschnitte aus dem kompletten und spiegeln es zusammen wieder.

Abbildung 24 zeigt die Abhängigkeiten der Pakete (namespaces) voneinander in dem Modul, welches die Makrofunktionalität anbieten soll.

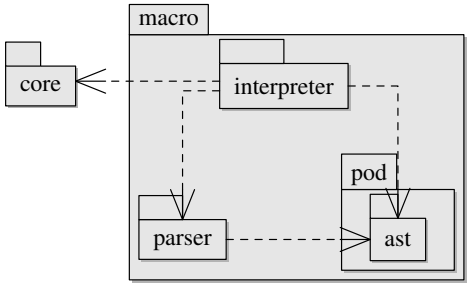

Abbildung 24: Abhängigkeiten von dem Makro Modul

`core` ist das Paket, indem das Command-Pattern aus Unterabschnitt 2.1 implementiert ist. Das `pod` Paket enthält alle Klassen, die nur zur Verwaltung von Daten dienen (separation of concerns). Deswegen werden sie *plain old data* (POD) oder auch *passive data structure* (PDS) genannt. In dem `pod` Paket befindet sich die `Token` Klasse und das `ast` Paket, d.h.

alle Klassen, die den abstrakten Syntaxbaum ausmachen. In dem `parser` Paket befinden sich der Tokenizer und Parser.

3.2.1 Token und Parser Paket

Der `Parser` aus Abbildung 25 bedient sich des `Tokenizers`, um eine `TokenList` zu bekommen. Diese `TokenList` kann der `Parser` dann parsen, bzw. in einen abstrakten Syntaxbaum umwandeln. Die `TokenList` Klasse dient nur zur Erklärung und wird sich nicht in der Implementierung wiederfinden, weil sie nur ein Array symbolisiert.

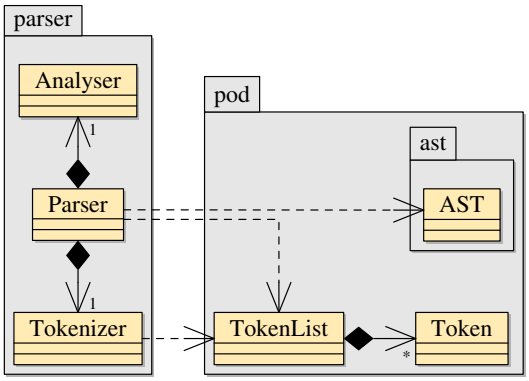

Abbildung 25: Parser Paket UML

Tokenizer Der `Tokenizer` wandelt den String, der das Makro beschreibt, in eine Reihenfolge von `Token`s um. Tokens sind alle Zeichen, die von Whitespace (\s*) getrennt sind, die nicht den Anforderungen als Bezeichner genügen ([^a-zA-Z0-9_], siehe Abbildung 7), die durch einen Punkt eine Dezimalzahl bilden (\d*\.\d+) oder einen String darstellen (".*?"[7]). Wenn der Tokenizer eine der beiden Arten von Kommentaren erkennt, verwirft er die kommentierten Zeichen, weil diese keine Tokens bilden sollen. Der `Tokenizer` ist nicht für die Umwandlung (Lexen) verantwortlich. Beim Lexen wird z.B. aus dem Token `"1.01"` der Double `1.01`. Das Lexen übernimmt der Parser.

Token Jedes `Token` beinhaltet Daten über die Zeile sowie Spalte als Zahl und den gesamten Quelltext aus der Zeile, aus der es stammt. Dies ist notwendig, um später hilfreiche Fehlermeldungen zu erzeugen, mit denen ein Benutzer schnell weiß, wo er nach dem Fehler suchen muss. Zusätzlich enthält die Klasse einen String, der den Teil des Makros enthält, den die Instanz darstellen soll (z.B. `if`).

Parser Der `Parser` ist dafür verantwortlich, dass die `TokenList` in einen AST umgewandelt wird. Wie in Unterabschnitt 2.1.2 beschrieben, gibt es drei Hauptarten von Parsern LL, LR und Recursive-Descent. LL und LR Parser sind meistens schneller als Recursive-Descent, weil sie mit Hilfe von Zustandstabellen arbeiten, die meist aus der

[7] Dieser reguläre Ausdruck funktioniert nur für einfache Varianten (kein escapen) von Strings und dient deswegen nur zur Veranschaulichung.

Backus-Naur-Form heraus entstehen. Der Nachteil der beiden ist, dass LL und umso mehr LR Parser schwer zu warten sind, weswegen meist Parser-Generatoren genutzt werden, um den Quelltext für den Parser zu generieren. Außerdem sind die Fehlermeldungen, die LL und LR Parser erzeugen, meistens schlechter als die, die Recursive-Descent Parser von Natur aus mit sich bringen[19]. Da die Wartbarkeit und Fehlermeldungen wichtige Punkte auf der Anforderungsliste sind, wurde sich für einen Recursive-Descent Parser entschieden. Zudem sind die gelungenen Parser von Clang[14] und GCC[16] ein gutes Beispiel und Vorbild dafür, was mit Recursive-Descent Parsern erreicht werden kann.

Die Operatoren müssen in der richtigen Reihenfolge zusammengestellt werden. Das heißt, dass bei `!a || b` zuerst `!a` ausgewertet werden muss und im Anschluss daran `x || c` (`x` sei das Ergebnis von `!a`). Der Baum muss so aufgebaut sein, dass diese Reihenfolge eindeutig und korrekt ist.

Variablendeklarationen mit anschließender Wertzuweisung müssen in zwei Schritte aufgeteilt werden (erst deklarieren und dann der Variablen den Wert zuweisen[8]).

Analyser Nachdem der `Parser` die `TokenList` in einen AST umgewandelt hat, wird der `Analyser` genutzt, um den AST zu validieren. Dies muss geschehen, weil gewisse Fehler nicht aus der Syntax hervorgehen – z.B., dass die `main()` Methode nicht explizit aufgerufen werden darf. Zwar ist es möglich den `Parser` zu erweitern, so dass er auch solche Fehler finden kann, allerdings ist es nicht die Aufgabe eines Parsers, etwas zu validieren.

Der `Analyser` wird den AST ablaufen und vor und nach jedem AST Element ein Signal abschicken. Signale sind mit dem Visitor-Pattern [13, S.366] sehr eng verwand. Ein Visitor wird bei dem entsprechenden Signal registriert und durch das Absenden des Signale aufgerufen – es können pro Signal mehrere Visitoren registriert werden. Wenn der `Analyser` dann zum Beispiel einen Funktionsaufruf, auf die `main()` Methode findet, kann er dies als Fehler melden. Im Gegensatz zu dem Visitor-Pattern werden bei Signalen alle Visitoren gesammelt und in einem Durchlauf abgearbeitet. Der Vorteil hierbei ist, das der AST nur einmal durchlaufen werden muss, was schneller ist und keine Nachteile mit sich bringt.

3.2.2 Abstrakter Syntaxbaum Paket

Wie Abbildung 26 zeigt[9], sind für alle Konstrukte, die das Scope (siehe Abbildung 12) aufnehmen kann, Klassen erforderlich. Alle Klassen erben von der `AST` Klasse, welche ein `Token` als Attribut besitzt. Da die `Token` alle Informationen über den Makro-Quelltext besitzen, und es sich bei der resultierenden Datenstruktur um einen Baum handelt (der Baum ist rekursiv – siehe Abbildung 27), ist auch der Interpreter in der Lage, informative Fehlermeldungen zu generieren.

[8] Dies ist valider C Code: `const int foo = foo + 1;`

[9] Es wurden einige Klassen hinter allgemeinen Begriffen – wie `Loop` – versteckt, um etwas Übersicht zu bewahren.

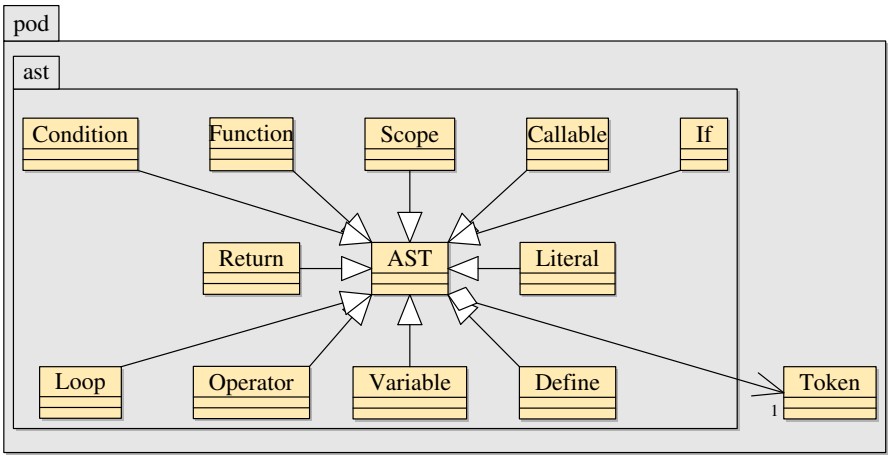

Abbildung 26: Stammbaum der AST Klassen

Abbildung 27 zeigt die `Scope` Klasse. Diese kann beliebig viele andere `AST` Instanzen aufnehmen.

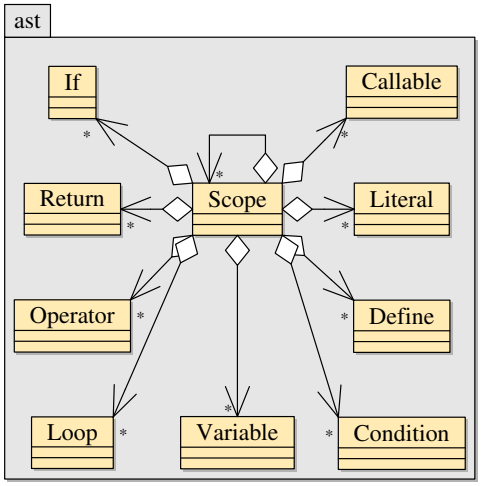

Abbildung 27: Verbindungen vom Scope

In Abbildung 28 sind die restlichen Abhängigkeiten zwischen den `AST` Klassen zu sehen.

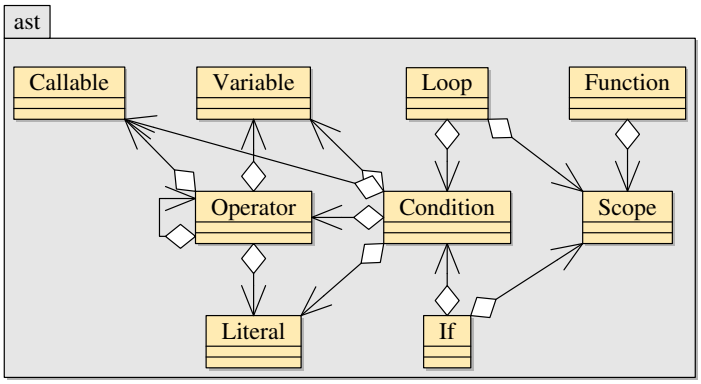

Abbildung 28: Abhängigkeiten der AST Klassen

3.2.3 Interpreter Paket

Der `Interpreter` nutzt den `Parser`, wie in Abbildung 29 zu sehen ist.

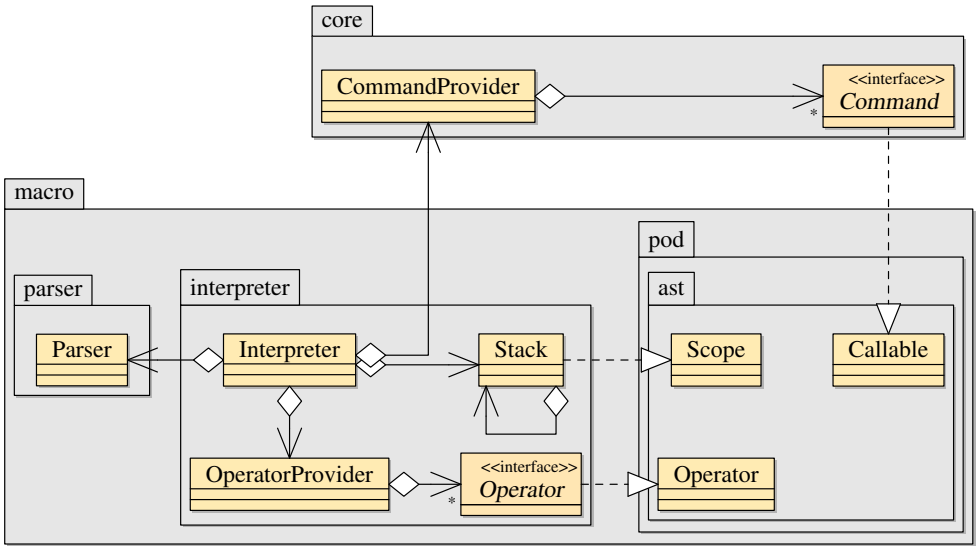

Abbildung 29: Interpreter Beziehungen

OperatorProvider Durch den `OperatorProvider` als Mittelmann ist es möglich, Operatoren auf spezielle Datentypen anzuwenden, die über die der Literals hinausgehen (siehe Unterabschnitt 3.3.4). Die Trennung von `OperatorProvider` und `Interpreter` beruht darauf, dass es nur einen `OperatorProvider` geben muss, aber es viele `Interpreter` geben kann. Zudem müssen deshalb die Operatoren nur ein einziges Mal registriert werden.

Stack Während der Ausführung des Makros durch den `Interpreter` übernimmt der `Stack` die Verwaltung der definierten Funktionen und Variablen sowie deren Werte und repräsentiert demzufolge die `ast::Scope`s.

Wenn ein neues `ast::Scope` in dem `AST` geöffnet wird, wird ein neuer `Stack` erzeugt, der als vorherigen `Stack` den aktiven `Stack` bekommt. Durch diese Verkettung wird eine verkettete Liste aufgebaut, bei der man nur das zuletzt hinzugefügte Objekt entfernen muss, um ein `ast::Scope` zu beenden, siehe Abbildung 30. In Unterabschnitt 3.3.5 wird die Architektur weiter erläutert.

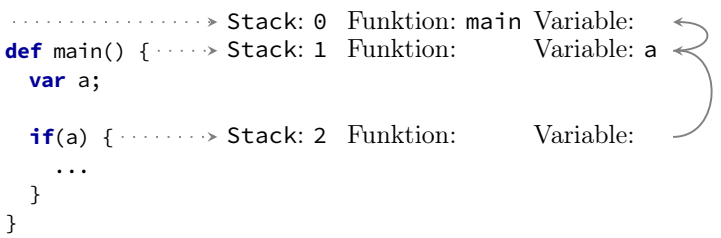

Abbildung 30: `Stack` Beispiel

Sollte der `Stack` jedoch keine passende Funktion haben, wird der `CommandProvider` nach einem passenden `core::Command` gefragt, welches dann in der C++ Ebene ausgeführt wird. Damit realisieren `core::Command`s die `ast::Callable` Funktionen genauso, wie `ast::Function`, die aus dem Makro hervorgehen.

Interpreter Der Interpreter selber läuft den AST ab, erweitert den `Stack` bei `ast::Scope`s, fügt dem `Stack` Variablen und Funktionen bei deren Definition (durch `ast::Define`) zu und entfernt das `Stack` Objekt, wenn er mit dem `ast::Scope` fertig ist. `ast::Condition`s werden mit Hilfe des `OperatorProvider`s ausgewertet. Der Rest der `ast` Klassen, lässt sich durch eine Verknüpfung der vorherigen Vorgehensweisen lösen.

3.3 Detaillierte Teilarchitekturen

Dieser Abschnitt bildet den Übergang vom theoretischen zum praktischen Teil, in dem hier die komplexeren Bestandteile der Architektur beschrieben werden. Um diese Teilarchitekturen beschreiben zu können, wird in Unterabschnitt 3.3.1 die Programmiersprache gewählt, in welcher das Makrosystem entwickelt wird. In den darauf folgenden Kapiteln wird die Architektur des `Parser`s, `OperatorProvider`s, `Stack`s, `Analyser`s und des `Interpreter`s genauer betrachtet.

3.3.1 Die Programmiersprache

Python ist eine der meist genutzten Programmiersprachen, wenn es um Plugin-Systeme oder die allgemeine Erweiterbarkeit für Programme geht. Ein Grund dafür ist, dass Python viele Interface zu unterschiedlichen Programmiersprachen bietet und somit auch als 'Kleber' genutzt werden kann. Ein anderer Vorteil von Python ist, dass es eine sehr große Codebasis hat und dadurch Probleme mit wenig Code lösbar sind. Diese Arbeit würde vor allem von den wesentlich komplexeren und vollständigeren String Manipulationen profitieren, die von Hause aus angeboten werden. Damit würde die Implementierung eines Tokenizers und Parsers einfacher sein.

Egal welche Programmiersprache genutzt wird, müssten die Objekte aus der C++ Ebene in die neue übersetzt werden. Durch das *Name-Mangling*[10] ist es schwierig, eine API von C++ zu anderen Programmiersprachen anzubieten. Meistens wird eine C API aus der C++ Welt von den Entwicklern angeboten, um mit anderen Programmiersprachen zu kommunizieren. Bei dieser verliert man den Vorteil der Objektorientierung und muss meistens auch die Daten zwischen C++$\longleftrightarrow$C und C$\longleftrightarrow$XYZ (z.B Python oder Lua) konvertieren, was langsam ist.

Eine Alternative zum manuellen Konvertieren der Daten sind Frameworks wie z.B. boost.python[11]. Diese können genutzt werden, um objektorientiertes C++ über Python auszuführen – allerdings ist dieses wie auch ChaiScript zu 'mächtig'. Diese Frameworks

[10] Beim 'Name-Mangling' fügt der Compiler dem Funktionsnamen weitere Informationen hinzu, um eine eindeutige Funktionssignatur zu erhalten.

[11] http://www.boost.org/doc/libs/1_60_0/libs/python/doc/html/index.html

sind ideal, um Software mit neuen Funktionalitäten zu erweitern, ohne die Anwendung neu zu kompilieren. Das Ziel dieser Arbeit ist allerdings nicht die Erweiterung, sondern die bessere Ausnutzung bzw. Automatisierung von den vorhandenen Funktionen. Zudem bauen viele Frameworks auf eine feste API auf, was in diesem Fall nicht garantiert ist.

Wenn ein Mittelschritt gewählt wird – also keine Frameworks, welches C++ Funktionen Python direkt zugänglich machen – dann müsste der `Stack` in der nativen Ebene (C++) bleiben, weil die C++ `core::Commands` auf Daten arbeiten müssen und sich so die Konvertierung gespart wird. Das würde die Implementierung in zwei Sprachen aufteilen, was selten eine gute Idee ist.

Letztlich ist der Preis, eine weitere Programmiersprache einem Projekt hinzuzufügen, ein großer Komplexitätsanstieg. Wenn eine weitere Sprache zu einem Projekt hinzukommt, sollte deshalb sicher sein, dass alle Vorteile der Sprache genutzt werden können.

Aufgrund der genannten Bedenken wird das Makrosystem in C++ implementiert. Dabei sind die folgenden Dinge zu berücksichtigen:

- dependency cycle
  Ein dependency cycle entsteht, wenn sich in C oder C++ zwei (oder mehr) Datenstrukturen gegenseitig brauchen.
  Beispiel: `class A` braucht `class B` und `class B` braucht `class A`, um sich zu definieren. Eine Lösung besteht darin, eine Klasse so zu definieren, dass sie nur von dem Pointer der anderen Klasse abhängt. Da der Typ bekannt ist, nämlich *Pointer* von `B`, kann die Klasse `A` vollständig definiert werden und somit auch `B`.

  Da das `Scope` Instanzen von allen `AST` Klassen aufnehmen kann, und einige Klassen wiederum ein `Scope` besitzen, kommt es hier zu einem dependency cycle. Das `Scope` ist eine der meist benutzten Klassen, weshalb die Auflösung des dependency cycles bei den anderen Klassen liegt. Das heißt, dass Klassen wie `ast::Function` einen Pointer[12] auf eine `ast::Scope` Instanz besitzen und nicht direkt eine `ast::Scope` Instanz.

- plain old data
  In C++ sind nur Klassen PODs, die trivial sind und Standard-Layout haben [23, 9 Classes §10]. Da die `Tokens` `std::string` benutzen, sind alle Klassen die von `Token` ein Attribut haben, genauso wenig POD wie `std::string`. Somit ist das `pod` Paket nur ein Hinweis darauf, dass die Klassen keine Logik haben, aber nicht POD Klassen nach dem C++Standard sind.

3.3.2 Parser Architektur

Der `Tokenizer` hat keinen Zustand, weil er nur einen stream von Zeichen aufteilen muss, und sollte daher nicht weiter beschrieben werden müssen. Dies gilt auch für alle `ast` Klassen, weil diese POD Klassen sind und somit nur Daten verwalten.

Da der `Parser` Recursive-Descent implementiert wird, kann der `Parser` ohne einen Zustand – sprich Attribute – auskommen. In diesem Fall kann darauf verzichtet werden, den

[12] In der Implementierung werden nur Smart-Pointer verwendet, um memory leaks vorzubeugen.

`Parser` als Klasse zu implementieren und anstelle dessen eine **`static`** Funktion anbieten. Dies hat den Vorteil, dass der `Parser` garantiert Thread-Safe ist, weil alle Daten von dem nativen Funktionsstack verwaltet werden.

Für nahezu alle `ast` Klassen bzw. Schlüsselwörter sollte der `Parser` eine Methode zum Parsen besitzen. Diese Modularität erlaubt es später, den `Parser` leichter zu erweitern, bzw. Fehler zu lokalisieren können. Da jede Methode genau für einen Teil der Syntax zuständig ist und es keinen Objektzustand gibt, können die Methoden für sich betrachtet und überprüft werden.

Die Einstiegsmethode des `Parser`s muss zusätzlich zu dem zu parsenden String den Namen des Makros bzw. der Datei übergeben bekommen. Dies sorgt dafür, dass die Fehlermeldungen für den Benutzer eindeutig zuzuordnen sind.

Die Fehlermeldungen sollten so viel Informationen wie möglich an den Nutzer liefern, ohne dass es sich um nutzlose Informationen handelt. So ist es für den Nutzer nicht nur wichtig, in welcher Zeile und Spalte der Fehler liegt, sondern auch in welchem Kontext. Das bedeutet, dass es einen Stack gibt (siehe Unterabschnitt 5.2: Punkt 3), der dem Nutzer gezeigt werden kann. Sinnvoll ist es, z.B. alle Scopeanfänge anzugeben (Funktionen/-Kontrollstrukturen). Durch diese Informationen ist der Nutzer sofort mit dem Kontext des Fehlers versorgt und kann über den Grund des Fehlers oder die Lösung nachdenken, während er zu der Zeile und Spalte navigiert.

3.3.3 Analyser Architektur

Durch die Entscheidung, dass der `Analyser` mit Hilfe von Signals implementiert wird, die dem Visitor-Pattern ähneln, ist die Architektur in zwei Teile aufteilbar.

Der erste Teil beschäftigt sich mit dem Ablaufen des AST und dem Absenden der Signale (der Ausführung der Visitoren). Dieser Teil ist recht einfach, weil es keiner besonderen Logik bedarf. Vor dem Aufruf eines Signals von einem `ast::Scope` fügt der `Analyser` das Token des `ast::Scope`s als Referenz zu einer Liste hinzu. Diese Liste kann dann genutzt werden, um Fehlermeldungen mit der gleichen Informationsqualität des `Parser`s zu erzeugen[13].

Der zweite Teil sind die Tests, die als Visitoren bei den Signalen angemeldet werden. Die meisten der Visitoren kommen ohne weitere Daten aus, aber Visitoren, die wissen müssen, ob sie sich gerade in einem Loop Scope befinden, brauchen auch einen Zustand, der von dem ersten Teil beim Durchwandern des AST aktuell gehalten wird. Der Zustand ist nötig, weil der AST nicht als doppelt verkettete Liste implementiert ist – also nur die Kinder, die Eltern kennen. Dieser Zustand lässt die Grenze der beiden Teile ein wenig verschwimmen, ist aber die simpelste und schnellste Möglichkeit, den `Analyser` leicht erweiterbar zu halten.

Unter anderem findet der `Analyser` Fehler wie z.B. die doppelte Funktionsdeklaration. `def fun() {...}` und `def fun(foo) {...}` stehen nicht in Konflikt, weil sie andere Parameter haben. Da die Reihenfolge der Parameter nicht Bestandteil der Signatur ist, ist

[13] Der `Analyser` setzt Unterabschnitt 5.2: Punkt 3 um.

`def fun(a, b) {...}` und `def fun(b, a) {...}` – in demselben `ast::Scope` – nicht zulässig.

3.3.4 OperatorProvider Architektur

Der `OperatorProvider` ist eine Klasse, deren Aufgabe es ist, Datentypen Funktionen zuzuordnen und zur Verfügung zu stellen. Da die `Operatoren` zustandslos sind[14], müssen die `Operatoren` nur ein einziges Mal registriert werden. Die `Operatoren` für die `ast::Literal` Klassen werden von dem `OperatorProvider` automatisch registriert.

Die Überladung von `Operatoren` ist ausgeschlossen. Ebenso ist nicht vorgesehen, dass Datentypen implizit konvertiert werden. Das heißt, dass ein **`char`** nicht zu einem **`int`** promoted[15] werden kann, wie es in den meisten typisierten Programmiersprachen der Fall ist. Eine Ausnahme ist der **`bool`** und `!` Operator. Wenn eine Variable als Ausdruck für Logik (**`for`**, **`if`**, **`while`** oder Operatoren) ist, und nicht Bestandteil eines anderen Vergleichsoperators, probiert der `Interpreter` den **`bool`** Operator für die Variable anzuwenden. Der **`bool`** Operator existiert nur für diese Fälle und kann nicht explizit aufgerufen werden. In dem Kontext `i && a == b` wird für `i` der **`bool`** Operator aufgerufen aber, nicht für `a` und `b`.

3.3.5 Stack Architektur

Die `Stack` Klasse ist der komplexeste Bestandteil des `Interpreter`s. Der `Stack` verwaltet alle Variablen und Funktionsdeklarationen, die ein Makro beinhaltet.

Für die Variablen muss der `Stack` einem `std::string` eine `any` Instanz zuweisen. Da keine überflüssigen Kopien bei der Übergabe von Parametern erzeugt werden sollen, muss der `Stack` auch eine Referenz auf eine `any` Instanz aus einem anderen `Stack` erlauben, die einem `std::string` zugeordnet ist.

Die Funktionen brauchen nur als konstante Referenzen auf die Funktionsdefinitionen in dem `ast` gespeichert zu werden. Diese Objekte werden nicht verändert und dienen nur als Vorlage, welche der `Interpreter` interpretieren muss.

Letztlich haben die `Stack` Instanzen einen Pointer auf den `Stack` über ihnen, siehe Abbildung 29 und Abbildung 30. Der `Stack` bildet mit anderen Instanzen von sich einen Stack. Dies Erlaubt es eine Art Baum zu erstellen – der Baum ist nicht navigierbar, weil die Nodes nur ihre Eltern kennen. Jeder Ast dieses Baumes entspricht genau einem Stack, der die `ast::Scope`s bis zu einem Funktionsaufruf abbildet – dies wird in Abbildung 37 aus dem Unterabschnitt 4.3.4 deutlicher.

Der Pointer wird dann genutzt, um nach Variablen und Funktionsdeklarationen zu fragen. Im Fall, dass der `Interpreter` nach einer Funktion fragt, kann der `Stack` durch die verkettete Liste (siehe Unterabschnitt 5.2: Punkt 4) nicht nur die Funktionsdefinition zurückgeben, sondern auch gleich den Pointer auf den `Stack`, in dem die Funktion definiert

[14] Dies kann nicht garantiert werden, es ist allerdings unwahrscheinlich, dass dies ein Problem ist.
[15] Promoted bedeutet, dass ein kleinerer, primitiver Datentyp zu einem größeren implizit konvertiert werden kann. Dies ist immer dann möglich, wenn kein Datenverlust auftritt.

wurde. Es ist also nicht vonnöten, die Funktionen dem `Stack` Pointer zuzuordnen, in dem sie deklariert worden sind, weil dies automatisch geschieht. Wieso das nötig ist, wird in Unterabschnitt 4.3.4 deutlich.

3.3.6 Interpreter Architektur

Die Architektur des `Interpreter`s ist durch den AST und `Parser` relativ simpel. Außerdem lösen `OperatorProvider` und `Stack` die komplexesten Teile des `Interpreter`s.

Ähnlich wie der `Parser`, ist der `Interpreter` Recursive-Descent implementiert. Das heißt, dass für jede `ast` Klasse eine `interpret()` Methode existiert. Durch den Aufbau des ASTs endet der `Interpreter` 'immer' an einem Werterzeuger (Funktionsaufruf, Variable, ...), der von Kontrollstrukturen (`if`, `while`, ...) oder weiteren Werterzeugern konsumiert wird.

Der `Interpreter` interpretiert den AST in mehreren Schritten:

1. Alle Funktionsdefinitionen aus dem aktuellen `ast::Scope` werden interpretiert.

   Das sorgt dafür, dass keine Definitionsreihenfolge eingehalten werden muss – `fun() ; def fun() {...}` ist kein Fehler.

2. Das aktuelle `ast::Scope` wird abgearbeitet.

   Das 'root' `ast::Scope` wird normal behandelt, was bedeutet, dass `if`, `while`, Funktionsaufrufe, ... erlaubt sind. Eine Ausnahme ist die `main()` Methode. Diese darf niemals aufgerufen werden.

3. Die Abarbeitung der Einstiegsfunktion `main()`.

   Hier werden wieder die ersten zwei Schritte durchgeführt, welche sich in jedem folgenden `ast::Scope` sowie Funktionsaufrufen wiederholen.

Wenn eine Funktion interpretiert werden soll, muss der `Interpreter` Variablen, die als Parameter übergeben werden, dem neuen `Stack` als Referenzen hinzufügen. Im Anschluss daran kann er dann das `ast::Scope` der Funktion interpretieren. Das Zuweisen von Parametern verhält sich ähnlich wie bei JavaScript – die Parameter werden als (konstante) Referenz übergeben und nicht kopiert. Wenn dem Parameter ein neuer Wert zugewiesen wird, verändert sich der Wert aus dem aufrufenden Scope nicht. Stattdessen wird die Referenz aus dem `Stack` gelöscht und eine Variable, mit dem neuen Wert, angelegt.

Die Besonderheiten des Interpreters:

- Wenn der `Stack` keine entsprechende Funktion besitzt, wird der `CommandProvider` nach einer Funktion gefragt. Das bedeutet, dass die nativen Funktionen von Funktionsdefinitionen in dem Makro überschrieben und auch unerreichbar werden können.
- Wenn eine Funkion interpretiert werden muss und diese von dem `CommandProvider` kommt, werden die Parameter kopiert. Es ist durch die C++ Ebene vorgegeben und wird normalerweise durch Pointer beschleunigt. In der C++ Ebene ist es auch möglich die Move-Semantics[23, S.268 ff.] aus C++11 anstelle von Pointern anzuwenden. Das ist allerdings nicht aus der Makro Ebene möglich.

- Wenn eine Variable aus einem `ast::Scope` returned wird, also ein Funktionsrückgabewert ist, kann dies ohne eine Kopie passieren. Dies ist dann möglich, wenn die Variable, die returned wird in dem `Stack` angelegt wurde und keine Referenz ist.

Um einen `ast::Operator` zu interpretieren, geht der `Interpreter` durch folgende Schritte:

1. Interpretieren des `ast::Operator`s, bis er einen/zwei Werterzeuger und einen Operatortyp (z.B. `==`) hat.
2. Erzeugen des/der Wert/e des/der Werterzeuger/s.

   Im Falle, dass es sich um den Operatorentyp `&&` oder `||` handelt, kann der Interpreter, durch die Kurzschluss-Semantik, auch früher aufhören.
3. Nutzen des `OperatorProvider`s, um die beiden Werte mit einem registrierten Opteratoren zu vergleichen.

Da die `ast` Instanzen das `Token` besitzen, durch welches sie entstanden sind, ist der `Interpreter` – im Falle, dass kein passender `Operator` oder Funktion gefunden werden kann – in der Lage genau so gute Fehlermeldungen zu produzieren wie der `Parser`.

3.3.7 Komplexe Rückgabewerte

Die komplexen Rückgabewerte, aus der Makro Ebene in die C++ Ebene sind durch den `any` Typ kein Problem, weil sich diese durch den `Stack` schon von Anfang an in der C++ Ebene befinden. An dieser Stelle muss es sich um eine `ast::Variable` oder `ast::Literal` handeln, die angelegt wurde und kann deswegen ohne eine Kopie anzulegen aus, der C++ Funktion des Interpreters returned werden.

Um die Makros auch aus der C++ Ebene aufzurufen, bedarf es eines Wrappers für den Interpreter, der das `core::Command` Interface implementiert. Da die `interpret()` Methoden von dem `Interpreter` alle einen `any` Wert zurückgeben, ist der Rückgabewert kein Problem. Der `Interpreter` selber hat keine Ahnung, was sich in dem `any` Typ befindet – das ist nicht gut, aber dieselbe Situation, mit der die `core::ConcreteCommand`s klarkommen müssen. Darum ist es nur ein kleines Problem.

Als Parameter kann der `Interpreter` eine Liste von `any` Werten – die einem `std::string` zugewiesen sind – annehmen sowie zwei `std::string`s (Makro und Makro Name). `core::Command`s können nur eine Liste von `any` Werten, die einem `std::string` zugewiesen sind, annehmen. Drei `any` Werte mehr in der Liste des Wrapper `core::Command`s – als Makro, Makro Name und Datei – löst das Problem der unterschiedlichen Anzahl von Parametern.

Da Makros `core::Command`s aufrufen können, sind sie auch in der Lage andere Makros aufzurufen (siehe Unterabschnitt 5.2: Punkt 5). Deshalb könnten Makros nicht nur zur Automatisierung genutzt werden, sondern auch zur internen Erweiterung der Applikation. Das ist allerdings eine schlechte Nutzung, weil Makros langsamer sind, als reine C++ Funktionen oder `Command`s. Allerdings ist ein wertvoller Vorteil dieser Kombinierbarkeit, dass Makros durch C++ Test-Frameworks ausgiebig getestet werden können.

4 Exemplarische Realisierung

Um schnell Resultate zu sehen, wird die Implementierung nicht nur inkrementell, sondern auch test-driven [2] vorgenommen. Inkrementell bedeutet, dass nicht komplett vollständige Teile der Software genutzt werden, um abhängige Elemente zu entwickeln. Die inkrementelle Entwicklung von Software kommt aus der Agile Softwareentwicklung [8], einer der Vorteile ist, dass Resultate schneller gesehen werden. Zum Beispiel fallen Architektur Fehler schneller auf, wodurch grundlegende Probleme frühzeitig beseitigt werden können.

Test-driven bedeutet, dass es für 'alle' Funktionen einen Test gibt, den sie bestehen müssen. Das hat zur Folge, dass wenn eine neue Softwarekomponente entwickelt wird, diese schon 'ausprobiert' werden kann, ohne dass die Teile der Software bereits existieren müssen, die diese Komponente benutzten werden. Zudem sind Fehler leichter zu lösen, weil diese erstens früh gefunden werden und zweitens deren Lösungen durch Regressionstests auch auf Korrektheit überprüft werden können. Dadurch werden in den seltensten Fällen weitere Fehler durch Fehlerlösungen eingeführt. Test-driven bedeutet allerdings nicht, dass die Software am Ende komplett ausgetestet ist. Es bedeutet nur, dass Tests früher geschrieben werden, und dass es, im Vergleich zu anderen Entwicklungsmethoden, meistens mehr Tests gibt, die die Software auf Korrektheit überprüfen.

Die vorhandene Software ist Cross-Plattform (Windows und Linux) entwickelt. Im Rahmen dieser Bachelorarbeit wird die Software nur auf Linux entwickelt, weil die Implementierung des C++ Standards von Microsoft zum Teil unvollständig oder auch falsch ist und bei der Fehlersuche meist viel Zeit in Anspruch nimmt [18]. Bei der Entwicklung wird daher darauf geachtet, dass keine Linux spezifischen Bibliotheken in Anspruch genommen werden. Das schließt leider nicht aus, dass die entstehende Software ohne Anpassungen auf Windows ausgeführt werden kann.

In den folgenden Unterabschnitten wird der Ablauf für das Makro aus Listing 2 durchgegangen. Die Reihenfolge ist die, die in Abbildung 2 bereits beschrieben wurde.

```
1  def fun(foo) {
2    do {
3      foo = foo + 1.1;
4    } while(!foo);
5
6    return foo;
7  }
8
9  def main() {
10   var bar = "1 ";
11
12   return fun(foo:bar);
13 }
```

Listing 2: Beispiel für die exemplarische Realisierung

4.1 Tokenizer

Der `Tokenizer` verwandelt den Code aus Listing 2 zu den `Token`s, die in Listing 3 zu sehen sind (abgesehen von dem Pointer, der auf die Quelltextzeile zeigt).

Um die `Token`s zu erstellen, geht der `Tokenizer` Zeichen für Zeichen vor.

Whitespace Zum Beginn vom Tokenizen eines neuen Tokens, liest der `Tokenizer` alle Whitespaces und verwirft diese. Wenn es sich bei dem Whitespace um einen Zeilenumbruch handelt, wird der Zähler für die momentane Zeile hochgezählt und für die Spalte zurückgesetzt. Ansonsten wird nur der Zähler für die Spalte hochgezählt.

Dezimalzahl Wenn das Zeichen eine Zahl ist, probiert der `Tokenizer` den regulären Ausdruck \d*\.\d+ auf das aktuelle und die folgenden Zeichen anzuwenden – siehe Listing 3 Zeile 13. Integer werden als normale Token geparst. (Siehe Unterabschnitt 5.2: Punkt 6 für Verbesserungen.)

String Wenn das momentane Zeichen ein " ist, werden alle Zeichen gelesen, bis ein weiteres " gefunden wird. Um das escapen von " zu unterstützen, wird das jeweils letzte Zeichen gespeichert. Wenn es sich bei dem letzten Zeichen um \\ handelt, und das aktuelle Zeichen ein " ist, ist dieses escaped und wird nicht als String Ende angesehen. Wenn ein \\ escaped wird, wird das letzte Zeichen auf 0 gesetzt. Das Ergebnis dieses Verfahrens ist in Listing 3 Zeile 34 zu sehen. Ohne dieses Verfahren würde es drei `Token`s geben, und der Whitespace wäre "verloren" gegangen.

Besondere Token Besondere Token sind zum Beispiel: == oder !=. Für diese Token ist es nötig, das folge Zeichen zu kennen, um zu entscheiden, was für ein Token die Zeichen darstellen sollen.

Normale Token Normale Token sind all die Zeichenketten, die dem regulären Ausdruck [a-zA-Z0-9_] genüge tun. Diese werden an dem Zeichen beendet, welche dem regulären Ausdruck nicht gleichen (also [^a-zA-Z0-9_]).

```
1  l:1  c:1  t:def
2  l:1  c:5  t:fun
3  l:1  c:8  t:(
4  l:1  c:9  t:foo
5  l:1  c:12 t:)
6  l:1  c:14 t:{
7  l:2  c:3  t:do
8  l:2  c:6  t:{
9  l:3  c:5  t:foo
10 l:3  c:9  t:=
11 l:3  c:11 t:foo
12 l:3  c:15 t:+
13 l:3  c:17 t:1.1
14 l:3  c:20 t:;
15 l:4  c:3  t:}
16 l:4  c:5  t:while
17 l:4  c:10 t:(
18 l:4  c:11 t:!
19 l:4  c:12 t:foo
20 l:4  c:15 t:)
21 l:4  c:16 t:;
22 l:6  c:3  t:return
23 l:6  c:10 t:foo
24 l:6  c:13 t:;
25 l:7  c:1  t:}
26 l:9  c:1  t:def
27 l:9  c:5  t:main
28 l:9  c:9  t:(
29 l:9  c:10 t:)
30 l:9  c:12 t:{
31 l:10 c:3  t:var
32 l:10 c:7  t:bar
33 l:10 c:11 t:=
34 l:10 c:16 t:"1 "
35 l:10 c:16 t:;
36 l:12 c:3  t:return
37 l:12 c:10 t:fun
38 l:12 c:13 t:(
39 l:12 c:14 t:foo
40 l:12 c:17 t::
41 l:12 c:18 t:bar
42 l:12 c:21 t:)
43 l:12 c:22 t:;
44 l:13 c:1  t:}
```

Listing 3: Tokenized Makro / TokenList

Token Erstellung Ein `Token` wird mit der Zeile und Spalte, in der es beginnt, und dem Token (z.B. `"1 "`) initialisiert. Ebenso wird dem `Token` ein Pointer auf einen (momentan) leeren String mitgegeben. Am Ende einer Zeile wird diesem leeren String dann die gesamte Zeile zugewiesen. Da es sich um einen Pointer handelt, wird die Zeile allen `Token`s aus der Zeile gleichzeitig zugewiesen. Der Pointer, den der `Tokenizer` hat, wird durch einen neuen Pointer auf einen leeren String ersetzt, um diesen den nächsten `Token`s aus der nächsten Zeile, zu geben.

4.2 Parser

Der `Parser` arbeitet intern mit der `TokenList` und gibt den einzelnen Funktionen diese sowie den aktuellen Index(`size_t`/`unsigned int`). Da der Index so wie die `TokenList` als Referenz übergeben wird, erstellen die einzelnen Methoden eine Kopie von dem Index, bevor sie anfangen zu arbeiten. Die Kopie des Indexes sorgt dafür, dass im Falle eines Fehlers in einem Unter-Unterfunktionsaufruf, die aufrufenden Funktionen noch 'wissen', welches `Token` sie am Anfang bekommen haben und so gute Fehlermeldungen produzieren können. Das die Indexe als Referenz übergeben wurden, liegt daran, dass es zwar möglich ist, mehrere Rückgabewerte unterschiedlichen Typs zu haben (`std::pair<T1,T2>` oder `std::tuple<T1, T2, ... Tn>`), dies aber ein schlechteres Interface bieten würde.

Der `Parser` fängt mit einem `ast::Scope` als root Node des Baumes an. Im Anschluss daran werden die `Token`s nacheinander geparst. Die Funktionen, die das Parsen übernehmen, implementieren die Syntax aus Unterabschnitt 3.1.

Die erzeugte AST Struktur – von dem root `ast::Scope` – ist in Listing 4 zu sehen. Diese wie alle folgenden AST Strukturen, sind von dem Parser generiert und den AST Elementen zu Text konvertiert worden. Das Ausgeben der Datenstruktur wurde den `pod` Klassen hinzugefügt, um diese leichter entwickeln und debuggen zu können. Da der AST zu viel eingerückt und zu lang für eine Seite ist, werden nur Ausschnitte gezeigt, die aus dem Beispiel (Listing 2) generiert wurden.

```
@Scope {
  l:0 c:0 t:
  ...
}
```

Listing 4: Root Scope des Beispiels

```
l:1  c:1  t:def
l:1  c:5  t:fun
l:1  c:8  t:(
l:1  c:9  t:foo
```

Aktuelle `TokenList`

4.2.1 Definition parsen

Um ein `ast::Define` Objekt zu erzeugen, erwartet die `Parser` Methode, dass das momentane `Token` entweder def entspricht, oder var.

Um `ast::Funktion` zu parsen, erwartet die Methode, dass das momentane `Token` def entspricht – var um eine `ast::Variable` zu parsen. Wenn keiner der beiden Fälle eintritt, wird `false` zurückgegeben – alle Methoden zum Parsen verhalten sich so. Das

`false` anstelle von einem `ast::Define` Objekt zurückgeben werden kann, wird durch den `optional<T>` Typ erreicht [3].

Funktion Wenn def geparst wurde, wird die `TokenList` an die Funktion für `ast::Funtion` übergeben. Diese erwartet als aktuelles `Token` einen Bezeichner (siehe Abbildung 7) und, dass das nächste Zeichen eine offene Klammer \(ist. Nach der Klammer kann eine beliebige Anzahl von Kommata getrennten Bezeichnern angegeben werden. Diese werden dem `ast::Function` Objekt als Array von `ast::Variable` Objekten übergeben. Nach den Parametern wird eine geschlossene Klammer \) erwartet. Zuletzt wird die Funktion zum Parsen von `ast::Scope` aufgerufen und das Ergebnis dieser in dem `ast::Function` Objekt gesetzt.

Wenn Fehler beim Parsen auftreten – zum Beispiel, dass kein Bezeichner oder das `ast::Scope` fehlt – wirft die Methode eine Exception, die bis zu der **`static`** globalen Methode nach oben wandert. Das Parsen der Funktion ist von einem **`try-catch`** Block umschlossen. Sollte also eine Exception geworfen werden, wird diese gefangen, und mit dem Dateinamen, Zeile, Spalte und Quellcode der Zeile erweitert, um dann weiter geworfen zu werden.

Das Ergebnis, von dem Beispiel (**`def`** `fun(foo) {`), ist in Listing 5 zu sehen, `ast::Define` (Zeile 1) enthält die `ast::Function` (Zeile 3) mit ihren Parametern (Zeile 5 ff.) und dem Funktionsscope (Zeile 9).

```
1  @Define {
2    line: 1 column: 1 token: def
3    @Function {
4      line: 1 column: 5 token: fun
5      parameter:
6        @Variable {
7          line: 1 column: 9 token: foo
8        }
9      @Scope {
10       ...
11     }
12   }
13 }
```

Listing 5: Funktionsdefinition des Beispiels

```
1  l:1  c:1  t:def
2  l:1  c:5  t:fun
3  l:1  c:8  t:(
4  l:1  c:9  t:foo
5  l:1  c:12 t:)
6  l:1  c:14 t:{
7  l:2  c:3  t:do
8  l:2  c:6  t:{
9  l:3  c:5  t:foo
10 l:3  c:9  t:=
11 l:3  c:11 t:foo
12 l:3  c:15 t:+
13 l:3  c:17 t:1.1
```

Aktuelle `TokenList`

Variable Wenn das geparste `Token` var war, wird die `TokenList` an die Funktion für `ast::Variable` übergeben. Diese erwartet als aktuelles `Token` nur einen Bezeichner. Es ist also wichtig, dass zuerst eine `ast::Funktion` probiert wird zu parsen und erst im Anschluss eine `ast::Variable`. Wie bei der Methode zum Parsen von `ast::Function`, befindet sich die Logik dieser Methode in einem **`try-catch`** Block.

Vermutlich entgegen der Erwartungshaltung des Lesers wird hier nicht die Zuweisung behandelt, dies geschieht bei der `ast::Operator` Funktion. Im Verlauf der Erklärung, wie Operatoren geparst werden, sollte es offensichtlich werden, wieso dies so ist.

4.2.2 Scope parsen

Die `ast::Scope` Klasse ist eine der meist verwendeten Klassen aus dem `ast` Paket. Die `ast::Scope` Klasse macht sich `variant<T1, T2, ..., Tn>` [21] zu Nutze – `variant` ist eine typsichere Union. Der `variant` Typ des `ast::Scope`s hat als Templateparameter alle `ast` Typen, die in einem Loop Scope auftreten können (Abbildung 11).

Die `Parser` Methode erwartet, dass das aktuelle `Token` \{ entspricht. Wenn dem so ist, werden so lange alle `ast` Klassen probiert geparst zu werden, bis ein `Token` nicht aufgelöst werden kann. Dies ist entweder der Fall, wenn das Token \} ist – somit dieses `ast::Scope` schließt – oder ein unerwartetes Token ist. Im letzteren Fall wird eine Exception geworfen. Wenn eine Methode einen validen Wert zurück gibt, wird das Objekt dem Array des `ast::Scope`s als Node hinzugefügt.

Wie in der Syntax aus Abbildung 12 müssen einige `ast` Elemente mit einem ; gefolgt werden, weil sie auch in einem Kontext eingesetzt werden können, in dem kein ; benötigt ist. Diese Token werden durch das Scope gelesen und finden sich nicht im AST wieder, weil sie in diesem keinen Wert haben. Sollten sie fehlen, wodurch Statements nicht zuverlässig geparst werden können, wird eine Exception geworfen.

Die parse Methode des `ast::Scope`s ist relativ simpel, weil hier nur die richtige Reihenfolge der Methoden eingehalten und bei bestimmten `ast` Elementen ein ; gelesen werden muss. `ast::Variable` muss zum Beispiel als letztes probiert geparst zu werden, weil diese nur einen Bezeichner brauchen, welcher auch zu einem Funktionsaufruf gehören kann.

Das geparste `ast::Scope` der `fun(foo)` Methode (Zeile 1 `def fun(foo) {`) ist in Listing 6 zu sehen.

```
1 @Scope {
2   line: 1 column: 14 token: {
3   @DoWhile {
4     ...
5   }
6   @Return {
7     ...
8   }
9 }
```

Listing 6: Funktionsscope des Beispiels

```
6  l:1  c:14 t:{
7  l:2  c:3  t:do
8  l:2  c:6  t:{
9  l:3  c:5  t:foo
10 l:3  c:9  t:=
11 l:3  c:11 t:foo
12 l:3  c:15 t:+
13 l:3  c:17 t:1.1
14 l:3  c:20 t:;
```

Aktuelle `TokenList`

4.2.3 do-while parsen

In Zeile 2 des Beispiels, wird eine **`do-while`** Schleife definiert. Der `Parser` guckt bei `ast::DoWhile` nach dem do, erwartet im Anschluss ein Scope, das wiederum von einem while, einer `ast::Condition` und ; gefolgt wird.

Ähnlich wie bei der Funktionsdeklaration befindet sich die Logik dieser Methode in einem **`try-catch`** Block.

Listing 7 zeigt den geparsten AST für den `ast::DoWhile`.

```
@DoWhile {
  line: 2 column: 3 token: do
  Contition:
    @UnaryOperator {
      ...
    }
  Scope:
    @Scope {
      line: 2 column: 6 token: {
      @BinaryOperator {
        ...
      }
    }
}
```

Listing 7: do-while Schleife des Beispiels

```
l:2  c:3  t:do
l:2  c:6  t:{
l:3  c:5  t:foo
l:3  c:9  t:=
l:3  c:11 t:foo
l:3  c:15 t:+
l:3  c:17 t:1.1
l:3  c:20 t:;
l:4  c:3  t:}
l:4  c:5  t:while
l:4  c:10 t:(
l:4  c:11 t:!
l:4  c:12 t:foo
l:4  c:15 t:)
```

Aktuelle `TokenList`

4.2.4 Operator und Condition parsen

Der Unterschied zwischen dem Parsen von einem Operator und einer Condition ist, dass Operatoren geparst werden, wenn der linke Teil des Operators (im Fall, dass es sich um einen binären Operator handelt), schon geparst ist. Die Logik der `ast::Operator` parse Methode muss dies berücksichtigen – die `ast::Condition` parse Methode nutzt intern die `ast::Operator` Methode, um Operatoren zu unterstützen. Abgesehen von diesem Unterschied können die beiden Methoden gleich behandelt werden.

Wenn der `Parser` `ast::Operator`en parsed, passiert dies in zwei Schritten. Im ersten Schritt werden alle `ast` Elemente (Operatoren und Werterzeuger) erstellt, die der `Parser` aus den `Token`s erstellen kann. Diese sind nach dem Parsen allerdings nur als eine Liste angeordnet, weil der `Parser` nicht wissen kann, in welcher Reihenfolge er die `Token`s zusammenzusetzen hat. Die Liste wird dann, in dem zweiten Schritt, in einen Baum verwandelt. Das geschieht, indem der Parser einen `ast::Operator` mit seinen jeweiligen Operanden verbindet. Dabei wird die Präzedenz bzw. Reihenfolge, die in Abbildung 23 zu sehen ist, angewendet bzw. erstellt.

Operatoren richtig zu parsen, ist eine der schwierigsten Aufgaben beim Bau eines Parsers, weil der zweite Schritt nicht nur die Präzedenz der Operatoren berücksichtigen muss, sondern auch die Auflösungsreihenfolge. Mit Ausnahme von dem Assignmentoperator, werden alle Binäroperatoren von links nach rechts zusammengesetzt. Alle Unäroperatoren werden von rechts nach links zusammengesetzt. Alle `+` und `-` Operatoren sind zu Beginn unär, und werden während des zweiten Schrittes zu Binäroperatoren, wenn sich ein Werterzeuger vor ihnen befindet. Operatoren, die keinen Operanden haben, sind in diesem Fall keine Werterzeuger.

Beide Methoden umschließen ihre Logik mit einem **`try-catch`** Block, um Informationen zu dem Kontext des Fehlers zu liefern.

Binäroperator In Zeile 3, des Beispiels (`foo = foo + 1.1;`), sieht man den Fall, in dem der linke Operand von dem `ast::Operator` schon geparst ist. `foo` wurde als Variable geparst, weil dieser Parser ohne Vorausschauen implementiert wurde. Als nächstes Token ist `=` in der Liste, was das Parsen von einem binären Operator indiziert.

Falls es sich um einen binären `ast::Operator` handelt, erwartet die ast::Operator Methode, dass das zuletzt geparste Element ein Werterzeuger ist (Abbildung 9). Dieses wird als linkes Element des `ast::Operator` gesetzt. Als Nächstes erwartet die Methode, dass das nächste `Token` auch ein Werterzeuger ist.

Werterzeuger werden in dem `ast::Operator` sowie in anderen Klassen, die Werterzeuger als Attribute haben, als `variant` gespeichert. Dass die Werterzeuger, in einem `variant` gespeichert werden, spiegelt sich nicht in dem generierten Text des AST wieder.

Die Datenstruktur nach dem ersten Schritt zeigt Listing 8.

```
@Variable {
  line: 3 column: 5 token: foo
}
@BinaryOperator {
  line: 3 column: 9 token: =
  Operation: assignment
}
@Variable {
  line: 3 column: 11 token: foo
}
@UnaryOperator {
  line: 3 column: 15 token: +
  Operation: add
}
@Double {
 ...
}
```

Listing 8: Erster Schritt der Variablen Addition des Beispiels

```
@Variable {
  line: 3 column: 5 token: foo
}
@BinaryOperator {
  line: 3 column: 9 token: =
  Operation: assignment
}
@BinaryOperator {
  line: 3 column: 15 token: +
  Left operand:
    @Variable {
      line: 3 column: 11 token: foo
    }
  Operation: add
  Right operand:
    @Double {
     ...
    }
}
```

Listing 9: Halber zweiter Schritt der Variablen Addition des Beispiels

In dem zweiten Schritt wird die Liste zuerst von rechts nach links durchgegangen und der unäre Operator + wird zu einem binären Operator konvertiert. Im Anschluss wird die Liste von links bis zu dem + durchgegangen. Beim Zusammensetzen des + greift dieses auf `foo` und auf den Double zu. Anschließend wird die Liste von rechts nach links bis zu dem = durchgegangen. Der = Operator greift dann auf `foo` und den + Operator zu, der das andere `foo` und den Double enthält, was in Listing 9 zu sehen ist. Der generierte AST ist in Listing 10 zu sehen.

```
@BinaryOperator {
  line: 3 column: 9 token: =
  Left operand:
```

```
l:3  c:5  t:foo
l:3  c:9  t:=
l:3  c:11 t:foo
```

```
4     @Variable {
5       line: 3 column: 5 token: foo
6     }
7   Operation: assignment
8   Right operand:
9     @BinaryOperator {
10      line: 3 column: 15 token: +
11      Left operand:
12        @Variable {
13          line: 3 column: 11 token: foo
14        }
15      Operation: add
16      Right operand:
17        @Double {
18         ...
19        }
20    }
21 }
```

Listing 10: Variablen Addition des Beispiels

```
12 l:3  c:15 t:+
13 l:3  c:17 t:1.1
14 l:3  c:20 t:;
15 l:4  c:3  t:}
16 l:4  c:5  t:while
17 l:4  c:10 t:(
18 l:4  c:11 t:!
19 l:4  c:12 t:foo
20 l:4  c:15 t:)
21 l:4  c:16 t:;
22 l:6  c:3  t:return
23 l:6  c:10 t:foo
24 l:6  c:13 t:;
25 l:7  c:1  t:}
26 l:9  c:1  t:def
27 l:9  c:5  t:main
28 l:9  c:9  t:(
29 l:9  c:10 t:)
```

Aktuelle `TokenList`

Unäroperator Im Fall, dass es sich um einen Unäroperator handelt, wird der Operator gelesen (\+|\-|!|print|typeof) und im Anschluss ein Werterzeuger erwartet.

Listing 11 zeigt die geparste `ast::Condition` aus Zeile 4 des Beispiels (`} while(!foo);`).

```
1 Contition:
2   @UnaryOperator {
3     line: 4 column: 11 token: !
4     Operation: not
5     Operand:
6       @Variable {
7         line: 4 column: 12 token: foo
8       }
9   }
```

Listing 11: do-while Condition des Beispiels

```
17 l:4  c:10 t:(
18 l:4  c:11 t:!
19 l:4  c:12 t:foo
20 l:4  c:15 t:)
21 l:4  c:16 t:;
22 l:6  c:3  t:return
23 l:6  c:10 t:foo
24 l:6  c:13 t:;
25 l:7  c:1  t:}
```

Aktuelle `TokenList`

4.2.5 Literals parsen

Um Literals zu parsen bzw. zu erkennen, wendet der `Parser` für Strings ".*", Integer \d+, Doubles \d*\.\d+ und Booleans true| false als regulären Ausdruck an. Für den String reicht in diesem Fall der reguläre Ausdruck aus, weil der `Tokenizer` den String als ein `Token` produziert hat.

Um die nummerischen Tokens zu Werten zu konvertieren (lexen), nutzt der `Parser` die C++ Funktionen `std::stod(...)` und `std::stoi(...)` – diese Funktionen parsen

aus einem `std::string` einen **`double`** bzw. **`int`** Wert. Für die Booleanwerte reicht die Überprüfung, ob das `Token` ein Boolean darstellt. Danach wird der String kopiert und anschließend werden alle escapeden Zeichen unescaped (`"t\tt"`→`"t  t"`[16]).

Der Double aus Zeile 3 des Beispiels ist in Listing 12 zu sehen.

```
1 @Double {
2   line: 3 column: 17 token: 1.1
3   Data:
4     1.1
5 }
```

Listing 12: Double Literal des Beispiels

```
13 l:3  c:17 t:1.1
14 l:3  c:20 t:;
15 l:4  c:3  t:}
16 l:4  c:5  t:while
17 l:4  c:10 t:(
```

Aktuelle `TokenList`

4.2.6 Return parsen

Return ist leicht zu parsen, weil es nur einen Werterzeuger nach dem Schlüsselwort return erwartet.

Wie einige andere Methoden zuvor, umschließt die `ast::return` Methode die Logik mit einem **`try-catch`** Block.

Listing 13 zeigt Zeile 6 des Beispiels.

```
1 @Return {
2   line: 6 column: 3 token: return
3   @Variable {
4     line: 6 column: 10 token: foo
5   }
6 }
```

Listing 13: return Statement des Beispiels

```
22 l:6  c:3  t:return
23 l:6  c:10 t:foo
24 l:6  c:13 t:;
25 l:7  c:1  t:}
26 l:9  c:1  t:def
27 l:9  c:5  t:main
```

Aktuelle `TokenList`

4.2.7 Funktionsaufruf parsen

Um den Funktionsaufruf in Zeile 12 des Beispiels (**`return`** `fun(foo:bar);`) zu parsen, erwartet die `ast::Callable` Funktion, dass das erste `Token` ein Bezeichner ist und darauf folgend (ohne Abstand) sich ein \(befindet. Anschließend wird eine Kommata getrennte Liste von Parameternamen zu Werterzeugern erwartet. Dabei ist der Parametername von dem Werterzeuger durch ein : getrennt. Auf diese Liste muss eine \) folgen.

Auch diese Methode umschließt die Logik mit einem **`try-catch`** Block.

Listing 14 zeigt das Ergebnis aus der Zeile 12 des Beispiels.

[16] In C++ sieht es so aus: `"t\\tt"`→`"t\tt"`.

```
@Return {
  line: 12 column: 3 token: return
  @Callable {
    line: 12 column: 10 token: fun
    parameter:
      foo: @Variable {
        line: 12 column: 18 token: bar
      }
  }
}
```

Listing 14: Funktionsaufruf des Beispiels

```
l:12 c:3  t:return
l:12 c:10 t:fun
l:12 c:13 t:(
l:12 c:14 t:foo
l:12 c:17 t::
l:12 c:18 t:bar
l:12 c:21 t:)
l:12 c:22 t:;
l:13 c:1  t:}

```

Aktuelle `TokenList`

4.2.8 Analyser

Nachdem der `Parser` den AST geparst hat, nutzt dieser den `Analyser`, um herauszufinden, ob sich in dem Baum Konstrukte befinden, die unerwünscht sind – zum Beispiel, das die `main(...)` Methode manuell aufgerufen wird.

Wie zuvor beschrieben (siehe Unterabschnitt 3.3.3) nutzt der `Analyser` ein Signal/Visitor System[17], um bei jedem beliebigen Element, Funktionen auszuführen. In dem Fall, dass überprüft werden soll, ob die `main()` Methode manuell aufgerufen wird, reicht es aus, sich bei dem Signal anzumelden, welches `ast::Callable` signalisiert. In der Funktion, die das Signal ausführt, wird dann überprüft, ob das `Token` main gleicht. Wenn dem so ist, ist der AST nicht kompatibel, und eine Fehlermeldung wird in dem `Analyser` zu der Fehlerliste hinzugefügt. Wenn der AST durchlaufen ist, und somit alle Prüfungen ausgeführt wurden, wird die Fehlerliste an den `Parser` übergeben. Dieser wirft die Fehlerliste dann als Exception.

Der `Analyser` baut während des Ablaufens des ASTs auch einen Stack auf. Dieser Stack gleicht dem des Interpreters, ist aber nur eine Liste von `Token`s, weil die Variablen und Funktionen nur auf Existenz geprüft werden müssen. Beim Prüfen, ob alle Variablen existieren, wird nachgesehen, ob in der Liste von Variablen ein `Token` dem der `ast::Variable` gleicht.

Ein Fehler den der `Analyser` nicht prüfen kann ist, dass eine Funktion nicht existiert. Da der `Interpreter` auf den `CommandProvider` zugreift, um Funktionen zu finden, und diese Funktionen durchaus durch ein Makro registriert werden können, ist ein solcher Fehler erst zur Laufzeit zu finden.

4.3 Interpreter

Der `Interpreter` durchläuft den AST Recursive-Descent wie der `Analyser`. Anstatt Signale bei den AST Elementen auszuführen, setzt der `Interpreter` die Elemente um.

[17] Das Signal System ist eine vorhandene Firmenbibliothek und wurde somit nicht während der Arbeit entwickelt.

Der `Interpreter` ist durch die Nutzung von `Stack` und `OperatorProvider` eine der kleinsten Klassen – abgesehen von den Klassen aus dem `pod` Paket.

4.3.1 Scope Interpretierung

Bevor ein `ast::Scope` interpretiert werden kann, erweitert der `Interpreter` den `Stack`, mit einem weiteren `Stack` Objekt. Dieses repräsentiert das `ast::Scope`, indem es Variablen und Funktionsdefinitionen verwaltet.

Um ein `ast::Scope` zu interpretieren, führt der `Interpreter` zuerst alle Funktionsdeklarationen durch. Im Anschluss, geht der `Interpreter` die `ast` Elemente in dem `ast ::Scope` durch und interpretiert sie. Im Anschluss an das Interpretieren des root `ast:: Scope`s, wird die `main()` Methode von dem `Interpreter` aufgerufen.

Abbildung 31 zeigt die Startsituation des Interpreter Stacks.

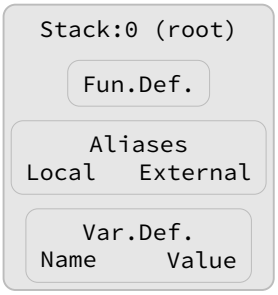

Abbildung 31: Root `Stack`

4.3.2 Funktionsdeklarationen

Bei Funktionsdeklarationen fügt der `Interpreter` dem `Stack` eine Referenz des `ast:: Function` Objektes aus dem AST hinzu. Da es sich um eine Referenz handelt, wird hier nicht kopiert. Abbildung 32 und Abbildung 33 zeigen den `Stack`, nach den jeweiligen Funktionsdeklarationen.

Abbildung 32: `Stack` nach der `fun(foo)` Deklaration

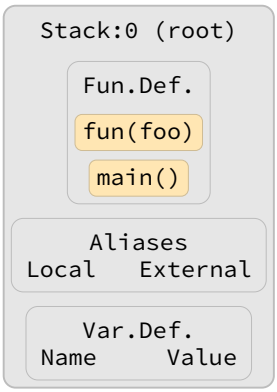

Abbildung 33: `Stack` nach der `main()` Deklaration

4.3.3 Variablendeklarationen

Bei einer Variablendeklarationen wird die `std::map` des `Stacks` um einen String und ein `any` Objekt erweitert (siehe Punkt Unterabschnitt 5.2: Punkt 8).

In Abbildung 34 ist die Variablendeklaration und in Abbildung 35 die anschließende Initialisierung der Variable aus der `main()` Methode zu sehen.

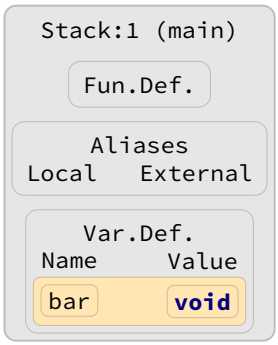

Abbildung 34: `Stack` nach Variablendeklaration

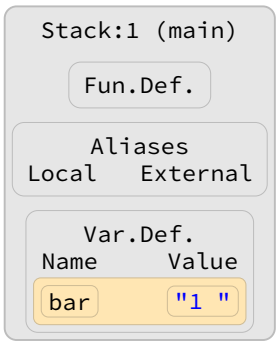

Abbildung 35: `Stack` nach Variableninitialisierung

4.3.4 Funktionsaufruf

Bei dem Funktionsaufruf (`return fun(foo:bar);`), fragt der `Interpreter` als erstes den `Stack` nach einer Referenz eines `ast::Function` Objektes, welches dem Funktionsaufruf gleicht. Es muss also eine Funktion definiert worden sein, die dem Bezeichner fun gleicht und einen Parameter namens foo besitzt. Im Anschluss erstellt der `Interpreter` einen neuen `Stack`(2), der den root `Stack`(0) als Pointer hat – siehe Abbildung 37. Es gibt nun also zwei Stacks, die jeweils zwei `Stack` Elemente besitzen. Dem neuen `Stack` wird dann ein Alias mit dem Namen `foo` hinzugefügt, der auf die `bar` Variable aus dem `main()` `Stack` zeigt. Dies ist in Abbildung 36 zu sehen. Nachdem der `Stack` vorbereitet ist, wird das `ast::Scope` der Funktionsdeklaration interpretiert.

Abbildung 36: `Stack` am Anfang der `fun` Methode

Wenn der neue `Stack`(2) den `main()` `Stack`(1) als Pointer hätte, könnte der neue `Stack`(2) direkt auf alle Variablen aus der `main()` Methode zugreifen und damit auch `bar` verändern.

Das ist durch die zwei Stacks nicht möglich, weil `Stack`(2) `bar` nur als Alias bzw. Referenz hat und diese nicht verändern kann – siehe Abbildung 38.

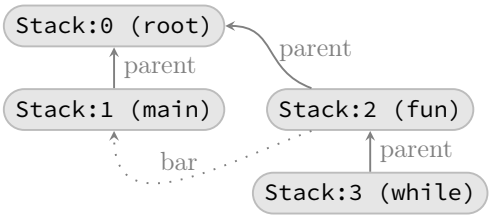

Abbildung 37: `Stack` Verbindungen

Sollte der `Stack` keine passende Funktion haben, greift der `Interpreter` auf den `core::CommandProvider` zurück. Wenn dieser ein `core::Command` zurück gibt, werden diesem die Argumente übergeben. Wenn der `core::CommandProvider` auch kein passendes `core::Command` für den Funktionsaufruf hat, wird eine Exception geworfen, was zum Ende des Interpretierens führt. Da der Interpreter Recursive-Descent implementiert ist, wird durch den Funktionsstack wieder ein Stack von Fehlermeldungen aufgebaut, der dem Nutzer zur Problemquelle führt.

4.3.5 Do-While

Ein `ast::DoWhile` wird interpretiert, indem das `ast::Scope` ausgeführt wird und im Anschluss die `ast::Condition` ausgewertet wird. Sollte das Ergebnis `true` sein, wird das `ast::Scope` wieder ausgeführt und am Ende wieder die `ast::Condition` geprüft.

Während der Ausführung ist es möglich, dass **`return`** oder z.B. **`break`** aufgerufen wird. Wenn ein solches Elemente interpretiert wird, wird ein Flag gesetzt, das der `DoWhile` Methode mitteilt, ob der Loop unterbrochen wurde. Diese Logik findet sich in allen `ast::Loop` und der `ast::Scope` Methode wieder.

4.3.6 Operator

Um `ast::Operator` Objekte zu interpretieren, löst der `Interpreter` die Werterzeuger so weit auf, dass zwei `any` Werte überbleiben. Diese Werte werden, mit dem Operator als **`enum`**, an den `OperatorProvider` zum Evaluieren gegeben.

Der `OperatorProvider` nutzt die Typinformation des `any` Typs (**`typeid`**`(T)`), um die richtige Funktion für die beiden Typen zu finden. Wenn keine passende Funktion gefunden wird, wird eine Exception geworfen. Ansonsten werden die Werte an die Funktion gegeben und das Ergebnis an den `Interpreter` returned. In dem Fall des Beispiels (`foo = foo + 1.1;`) wird die Funktion für `std::string` und **`double`** gesucht, die diese Typen addieren kann.

Der Assignmentoperator wird nicht von dem `OperatorProvider` angeboten, weil dieser abhängig von dem `Stack` Zustand ist. Wenn die Variable ein Alias ist (wie in dem Fall

des Beispiels – siehe Abbildung 36), muss dieser gelöscht werden und durch eine 'echte' Variable ersetzt werden, bevor der Wert verändert werden kann.

Die Abbildung 38 zeigt das Ergebnis der zwei Operatoren – der `Stack` von dem `DoWhile` wurde ausgelassen, weil dieser leer bleibt und deshalb als Proxy zu `Stack 2` dient.

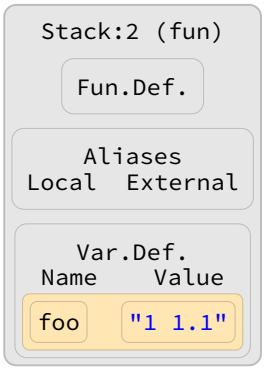

Abbildung 38: `Stack` nach dem Assignmentoperator der `fun` Methode

Im Fall von `} while(!foo);` wird der **`bool`** Operator implizit für `std::string` aufgerufen. Dieser gibt `true` zurück, wenn der String nicht leer ist. Durch den `!` Operator wird dann aus `true` `false`.

4.3.7 Return

Um ein `ast::Return` zu interpretieren, interpretiert der `Interpreter` den Werterzeuger soweit aus, dass er ein `any` Objekt erhält. In diesem Fall kann die Variable aus der Funktion extrahiert werden (Move-Semantics), was gut für die Performance des `Interpreters` ist.

Abbildung 39 zeigt den Zustand nach der Rückgabe aus `fun(foo)`.

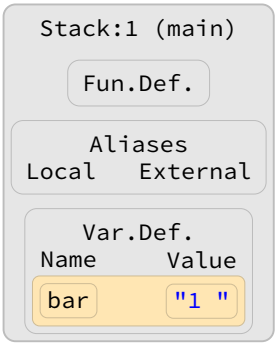

Abbildung 39: `Stack` nach dem **`return`** der `fun` Methode

4.4 Fehlermeldungen

Wenn in Zeile 3 (`foo = foo + 1.1;`) das `;` vergessen wird, erhält man die Fehlermeldung aus Listing 15 – diese Fehlermeldung wird von dem `Parser` ausgegeben und führt dazu,

dass das Parsen abgebrochen wird.

```
Anonymous:1:5: In the 'fun' function defined here:
def fun(foo) {
    ^
Anonymous:2:3: In the do-while defined here:
  do {
  ^
Anonymous:3:17: Expected a ';'
    foo = foo + 1.1
                  ^
```

Listing 15: Fehler bei fehlendem Semikolon

Eine Fehlermeldung von dem `Analyser` ist in Listing 16 zu sehen – weder in dem root `ast::Scope`, noch in der `main()` Funktion, ist eine Variable `ba` definiert.

```
Anonymous:9:5: In the 'main' function defined here:
def main() {
    ^
Anonymous:12:3: At return defined here:
  return fun(foo:ba);
  ^
Anonymous:12:18: Undefined variable 'ba'
  return fun(foo:ba);
                 ^
```

Listing 16: Fehler bei unbekannter Variable

4.5 Ergebnis der Implementierung

Die Implementierung setzt die Architektur und Syntax vollständig um. Es ist also möglich, Funktionen zu definieren, um `core::Command`s zu bündeln und nicht unnötig viel Code zu schreiben. `core::Command`s können aufgerufen werden – gegebenenfalls mit Parametern, die aus dem Makro kommen oder das Ergebnis eines anderen `core::Command`s sind. Über Kontrollstrukturen ist es möglich, zu steuern welche `core::Command`s ausgeführt werden sollen. Zudem bieten Schleifen die Möglichkeit, Code mehrmals ausführen zu lassen.

5 Zusammenfassung und Ausblick

Die meisten Ziele der Arbeit wurden in der Architektur und der exemplarischen Realisierung umgesetzt. Da das Makrosystem nur von dem `core::CommandProvider` und dem `core::Command` abhängig ist, ist es kein Problem, die Software mit Modulen zu erweitern. Funktionen können parametrisiert werden und erlauben Rückgabewerte, womit die Makros nicht nur Anweisungen hintereinander abarbeiten. Mit dem Lösen dieser Punkte ist das Ergebnis dieser Arbeit für die Automatisierung von der bestehenden Software geeignet.

Des Weiteren sind die Fehlermeldungen benutzerfreundlich genug, um auch Kunden eine Schnittstelle zur Verfügung zu stellen, wenn der Bedarf besteht. Außerdem sorgt die schlanke Ausstattung dafür, dass Benutzer wenige Möglichkeiten haben, schwerwiegende Fehler zu begehen, die die Anwendung gefährden.

Die Wartbarkeit des Makrosystems ist durch die Dokumentation und die klare Trennung der Bestandteile des Systems gegeben. Insgesamt umfasst die exemplarische Realisierung 12.000 Zeilen Code. Davon sind 2.500 POD Klassen als `ast`, 6.000 für den `parser` und 3.500 für den `interpreter` – dabei sind $\frac{1}{4}$ der Zeilen die Header mit den Deklarationen und Dokumentation. Zudem bieten die Tests einen Einstiegspunkt zum Verstehen der Software, weil sie als Beispiele genutzt werden können.

5.1 Evaluation

Dieser Abschnitt bewertet das erreichte Ergebnis, in Unterabschnitt 5.1.1 werden entscheidende Punkte der Architektur mit Alternativen verglichen und in Unterabschnitt 5.1.2 werden Vor- und Nachteile von der exemplarischen Realisierung genannt.

5.1.1 Architektur

POD vs OOP Die Architektur ist gut erweiterbar, weil es sich bei den `ast` Klassen um PODs handelt. Damit ist die Logik von den Daten getrennt, was dafür sorgt, dass durch Veränderungen an z.B. dem `Parser` der `Analyser` nicht betroffen ist, und so keine Fehler bei diesem eingeführt werden können.

Eine Alternative wäre es gewesen, rein nach OOP zu programmieren, womit die `ast` Elemente ‘sich selber’ hätten parsen, analysieren und interpretieren können. Das sorgt allerdings auch dafür, dass eine Klasse für drei unterschiedliche Kernelemente des Systems zuständig wäre, was dazu führen könnte, dass die großen Zusammenhänge nicht gesehen würden. Bei der OOP Alternative ist es auch möglich, dass die Klassen virtuelle Methoden für das Analysieren und Interpretieren nutzen.

Durch OOP und virtuelle Methoden würde das interne Interface ‘schöner’ sein – es müsste nicht `variant` genutzt werden. Darum könnten / müssten alle Elemente als `ast::AST *` gespeichert und einheitlich angesprochen werden. Die virtuellen Methoden kümmern sich in diesem Fall um die Typunterscheidung. Der Nachteil bei virtuellen Methoden ist, dass

sie langsamer sind als native Methoden. Zusätzlich bilden die Pointer einen Baum von Pointer, die auch langsamer sind, als wenn man Objekte direkt ansprechen kann.

c++14 und c++17 Ein weiterer positiver Punkt bei der Entwicklung ist, dass C++14 und Teile wie `any` [1], `optional` und `variant` [10] genutzt wurden. Die moderne Sprache und die Klassen aus dem aufkommenden C++17 Standard sorgen dafür, dass die Implementierung sauber und sicher zur selben Zeit ist.

Als Beispiel dient `variant`. Durch die Nutzung von `variant` sind Fehler, die mit **`union`**s entstehen können, minimiert worden, und der Speicherverbrauch von den Objekten hat nicht zu sehr darunter gelitten. Eine Alternative zu **`union`**s und `variant` wären **`virtual`** Methoden – diese sind allerdings langsamer, wie oben genannt.

Der Nachteil bei einer so modernen Ausstattung ist, dass es ebenso moderne Compiler braucht, um den Code zu kompilieren. Außerdem können Fehler auftreten, die von den neuen, wenig getesteten Typen verursacht werden. Ein Beispiel ist dabei die `std::experimental::any` Implementierung von gcc (5.3.0). Diese ist nicht standardkonform und sorgt dafür, dass der Code nur unter gcc kompilieren würde[18].

C++ Standard Ein nicht so gelungener Punkt der Architektur ist, dass sich in dem `pod` Paket keine POD Klassen nach dem C++ Standard befinden. Dies kann durchaus verwirren, wenn man POD Klassen in dem Paket erwartet.

5.1.2 Implementierung

Die Implementierung deckt die Anforderungen, die am Anfang der Arbeit gestellt wurden, ab und ermöglicht Nutzern, die vorhandene Software zu automatisieren. Sie ist allerdings nicht 'fertig' in dem Sinn, dass alle nützlichen Funktionen implementiert sind.

Arrays Das Abhandensein von Arrays ist eine klare Einschränkung, was die Kompatibilität angeht. Dass es keine Arrays gibt, beruht jedoch nur darauf, dass trotz eines straffen Zeitplans nicht genug Zeit vorhanden war, diese auch noch zu implementieren. Da dies von vornherein klar war, wurden die Arrays auch in dem Unterabschnitt 3.1 Syntax ausgelassen.

Tests Ein problematischer Punkt der Implementierung ist die Testabdeckung. Die Implementierung wurde Test-Driven vorgenommen, allerdings sagt dies nichts über die Testabdeckung der Software aus. Aufgrund des jungen Alters der Software ist anzunehmen, dass viele Fehler wegen weniger Nutzung und nicht bedachter Fehlerursachen noch vorhanden sind. Als Bibliothek wurde Catch [5] verwendet.

[18] Aus diesem Grunde wurde eine Bibliothek mit einer standardkonformen Implementierung genutzt.

Templates C++ bietet `template`s an, diese sind (sehr) grob mit "generics" aus Java zu vergleichen. Templates wurden an Stellen genutzt, wo durch sie Codeduplikation vermieden werden konnte. Templates machen den Code für Entwickler, die sich wenig mit dieser Art von Programmierung auskennen, wesentlich unverständlicher oder gar nicht nachvollziehbar. Der Vorteil liegt allerdings nicht nur in der geringeren Codeduplizierung und damit weniger Fehlern im Code, sondern auch darin, dass der Code wesentlich verständlicher werden kann. Dies ist der Fall, wenn man entweder Templates versteht oder sie 'ignoriert', weil man durch Templates weniger Code lesen muss. Ein Beispiel ist der `OperatorProvider`, in dem 352 Zeilen durch 24 ersetzt werden konnten, die zudem selbsterklärend sind.

5.2 Ausblick

Während der Implementierung sind die folgenden Punkte aufgefallen, die verbessert werden können oder nützliche Erweiterungen bieten. Die Liste bezieht sich vor allem auf die Implementierung, weil die Architektur mehr als ausreichend war.

1. Debugger / Stepping
   Es könnte ein Interface angeboten werden, mit dem man durch die Ausführung eines Macros Schritt für Schritt gehen kann.

2. C++17 `std::string_view`
   Um weniger Speicher zu verbrauchen und durch weniger Speicher Allokationen schneller beim Tokenizen zu sein, könnte `std::string_view` genutzt werden.

3. Mehr Fehler von dem Parser
   Anstelle, dass der Parser Exceptions nutzt und nach dem ersten Fehler aufhört, zu parsen, ist es mögliche, einen `Stack` von Fehlermeldungen zu produzieren – wie es in dem `Analyser` gemacht wird. Nach einem Fehler müsste nur bis zum nächsten Scopeanfang (`{`) , Scopeende(`}`) oder Semikolon(`;`) – je nachdem, wo der Fehler aufgetreten ist, die Tokens verworfen und dann weiter geparst werden.
   Eine weitere Variante ist es, AST Elemente als 'vergiftet' (poisoned) zu kennzeichnen, sodass alle Fehler, die in Verbindung mit dem Element auftreten, verworfen werden, weil es sehr wahrscheinlich ein Folgefehler ist.

   Dies hätte zur Folge, dass der Nutzer mehr Fehler auf einmal beseitigen könnte.

4. Verkettete `Stack` Liste in ein Array umwandeln
   Wenn sich herausstellt, dass die verkettete Liste von dem `Stack` zu langsam ist, kann der Interpreter ein Array nutzen, um die `Stack`s zu speichern und die `Stack`s nutzen, anstelle eines Pointers einen Offset, von dem sie aus die anderen `Stack` fragen können. Der Vorteil von Arrays gegen verkettete Listen ist die wesentlich höhere Cache-Locality. Der Nachteil wäre, dass die `Stack` Instanzen rechnen müssen, um zu wissen, welches Element ihr Ansprechpartner ist.

5. Einen Makro AST Buffer bereitstellen.
   Da das Parsen und Validieren der Makros ziemlich langsam ist, ist es ratsam, einen Provider zu implementieren, der den Namen des Makros mit dem geparsten AST

assoziiert. Der AST wird von dem `Interpreter` nicht verändert, weswegen das Makro nur geparst werden muss, wenn es sich verändert hat.

Ein weiterer Vorteil wäre, dass das Makro nicht als String einem anderen Makro bekannt sein muss oder als `core::ConcreteCommand` implementiert sein muss, weil der `Interpreter` im dem AST des anderen Makros weiter parsen würde. Dafür müsste der `Interpreter` natürlich ein wenig angepasst werden, weil er nun den '`MakroProvider`' vor dem `CommandProvider` nach einem passenden Makro fragen müsste.

6. Besseres Tokenizen und Lexen von Zahlen.
   Momentan werden Zahlen sehr primitiv getokenized und gelext. C++ unterstützt Zahlen mit wissenschaftlicher Notation zu lexen – dies wird nicht vom Tokenizer wie Parser unterstützt.

7. AST Optimierungen.
   Ein AST kann genutzt werden, um den Code, den er repräsentiert, zu optimieren. Eine Optimierung wäre es zum Beispiel, wenn Literals, die in arithmetischen Ausdrücken stehen, ausgerechnet würden.

8. Bessere Laufzeitfehler.
   Um Bessere Laufzeitfehler in dem `Interpreter` erzeugen zu können, sollte das gesamte `Token` in der `std::map` gespeichert werden anstelle eines einfachen Strings. Ein Beispiel wäre, wenn eine Variable für ein `core::Command` genutzt würde und der Typ des `any` Objektes nicht stimmte.

9. Objekt Zugriff
   Wenn mit ein Objekt angesprochen werden soll, muss dies momentan über ein `core ::Command` laufen, da Objekte nicht direkt angesprochen werden können. Dies ist sicherlich ein Schritt in die Richtung von Scriptsprachen, ermöglicht aber einen einfacheren Umgang mit Kontrollstrukturen, da die Objekte nach ihrem Zustand gefragt werden könnten.

10. Arrays
    Die momentane Sprache unterstützt weder die Erstellung von, noch den Zugriff auf Arrays. Array können für die Automatisierung wichtig sein und müssen mit dem Ergebnis dieser Arbeit wie Objekte über ein extra `core::Command` angesprochen werden.

11. **`try-catch`**
    Wenn ein `core::Command` nicht vorhanden ist, oder einen Fehler produziert, wird eine Exception geworfen. Es wäre sicherlich nützlich dies auch in dem Makro behandeln zu können.

Die meisten dieser Punkte sind auf die Optimierung der Performance ausgelegt. Wie immer muss für solche 'Verbesserungen' viel getestet und gemessen werden, um zu wissen, ob sich die Veränderungen gelohnt haben. Die Punkte, die weitere Funktionalitäten vorschlagen, sollten geprüft werden, ob diese Funktionalität der Automatisierung dienen, oder doch in die Richtung von Scriptsprachen gehen.

Literaturverzeichnis

[1] *Any*. 2016. URL: https://github.com/thelink2012/any (besucht am 28. 04. 2016).

[2] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[3] *C++ Library Fundamentals V1 TS Components for C++17*. 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0220r0.html (besucht am 09. 04. 2016).

[4] *C Workgroup*. 2016. URL: http://www.open-std.org/jtc1/sc22/wg14/ (besucht am 09. 04. 2016).

[5] *Catch*. 2016. URL: https://github.com/philsquared/Catch (besucht am 28. 04. 2016).

[6] *ChaiScript*. 2016. URL: https://github.com/ChaiScript/ChaiScript (besucht am 25. 04. 2016).

[7] Achim Clausing. *Programmiersprachen–Konzepte, Strukturen und Implementierung in Java*. Springer-Verlag, 2011.

[8] David Cohen, Mikael Lindvall und Patricia Costa. „Agile software development“. In: *DACS SOAR Report* 11 (2003).

[9] *ECMA SStandard*. 2015. URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm (besucht am 11. 04. 2016).

[10] *Eggs.Variant*. 2016. URL: https://github.com/eggs-cpp/variant (besucht am 28. 04. 2016).

[11] Helmut Eirund, Bernd Müller und Gerlinde Schreiber. *Formale Beschreibungsverfahren der Informatik: ein Arbeitsbuch für die Praxis*. Springer-Verlag, 2013.

[12] Jacques Ferber. „Computational reflection in class based object-oriented languages“. In: *ACM Sigplan Notices*. Bd. 24. 10. ACM. 1989, S. 317–326.

[13] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[14] LLVM. *Clang Features*. LLVM. 2016. URL: http://clang.llvm.org/features.html (besucht am 09. 04. 2016).

[15] Julie Zelenski Maggie Johnson. *CS 143. Lectures 4B-6*. Stanford. 2016. URL: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/ (besucht am 01. 03. 2016).

[16] Joseph Myers. *GCC New C Parser*. GCC Wiki. 2016. URL: https://gcc.gnu.org/wiki/New_C_Parser (besucht am 09. 04. 2016).

[17] *Python*. 2016. URL: https://www.python.org/ (besucht am 09. 04. 2016).

[18] *Rejuvenating the Microsoft C/C++ Compiler*. Microsoft. 2015. URL: https://blogs.msdn.microsoft.com/vcblog/2015/09/25/rejuvenating-the-microsoft-cc-compiler (besucht am 13. 04. 2016).

[19] Elizabeth Scott und Adrian Johnstone. „GLL parsing“. In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010), S. 177–189.

[20] *Swift*. 2016. URL: https://developer.apple.com/swift/ (besucht am 09. 04. 2016).

[21] *Variant: a type-safe union that is rarely invalid*. 2015. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0088r0.pdf (besucht am 20. 04. 2016).

[22] Steve Vinoski. „A time for reflection“. In: *Internet Computing, IEEE* 9.1 (2005), S. 86–89.

[23] *Working Draft, Standard for Programming Language C++*. 2013. URL: https://github.com/cplusplus/draft/blob/master/papers/n3691.pdf (besucht am 16. 04. 2016).

Abbildungsverzeichnis

Listingverzeichnis

Anhänge