
Liste der noch zu erledigenden Punkte

■ Danke an Mutter, Vater und alle Profs	5
■ Ref Section in implementation / architecture	13
■ For, For-each Schleife wenn implementiert	17
■ Ich hätte hier vor, an Hand eines Beispieles, die “Knackpunkte” der einzelnen Klassen zu erklären, bzw. den Ablauf wieder zu spiegeln (String->Tokens->Parser/AST->Interpreter). Hört sich das gut an? (Code wäre getrimmt oder Pseudo-Code)	29
■ Ich vermute das ich in diesem Kapitel 6-8 Seiten zu schreiben habe, wenn es Vorschläge gibt, was noch niedergeschrieben werden sollte, bzw. ausführlicher erklärt werden muss, nur zu. Ich habe wie immer eher das Problem das ich zu wenig Text produziere.	30
■ Ich habe http://chaiscript.com/ gefunden. Quasi das was ich hier mache, bloß das die 9 Jahre Zeit hatten – ich habe also einen Referenzpunkt! :)	31



HOCHSCHULE BREMEN

BACHELORARBEIT

THESIS

**Konzeption und Implementierung einer
Makrosprache in C++**

Autor:
Roland JÄGER
360 956

19. April 2016

Inhaltsverzeichnis

Allgemeines	4
Eidesstattliche Erklärung	4
Danksagung	5
1 Einleitung	6
1.1 Problemfeld	6
1.2 Ziele der Arbeit	7
1.3 Hintergründe und Entstehung des Themas	8
1.4 Struktur der Arbeit, wesentliche Inhalte der Kapitel	8
2 Anforderungsanalyse	9
2.1 Diskussion des Problemfeldes	9
2.1.1 Was ist ein Makrosystem?	9
2.1.2 Parser	10
2.1.3 Das vorhandene System	10
2.2 Anforderungen an die angestrebte Lösung	12
3 Konzeption	14
3.1 Syntax	14
3.1.1 Syntax Grundlagen	14
3.1.2 Definitionen	16
3.1.3 Kontrollstrukturen	17
3.1.4 Befehle	17
3.2 Grundarchitektur	18
3.2.1 Token und Parser Paket	19
3.2.2 Abstrakter Syntaxbaum Paket	20
3.2.3 Interpreter Paket	22
3.3 Detaillierte Teilarchitekturen	23
3.3.1 Die Programmiersprache	23
3.3.2 Parser Architektur	24
3.3.3 OperatorProvider Architektur	25
3.3.4 Stack Architektur	25
3.3.5 Interpreter Architektur	26
3.3.6 Komplexe Rückgabewerte	27
4 Exemplarische Realisierung	29
4.1 Tokenizer	29
4.2 Parser	29
4.3 Interpreter	30
4.4 Fehlermeldungen	30
5 Evaluation	31
6 Zusammenfassung und Ausblick	32
6.1 Ausblick	32

Anhänge	36
----------------	-----------

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 19. April 2016

Roland Jäger

Danksagung

Danke an Mutter, Vater und alle
Profs

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 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*¹ erweitert werden kann. Eines dieser Module wird das Makrosystem sein, welchen in dieser Arbeit entwickelt wird. Durch diese Modularität, gibt es kein festes Interface.

- 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 Schleifen. Und die Makros sollen 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.

- 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 dem Nutzer schnell zu seinem Fehler führen und keine false positives enthalte.

¹ Bibliotheken, die zur Laufzeit – nach den dynamischen Bibliotheken – nachgeladen werden können, um die Funktionalität der Applikation zu erweitern.

1.3 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 ihnen 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 zusammen gezählt und deren Preis ermittelt werden. Anstelle, dass hier eine sehr komplexe Suchfunktion entwickelt wurde, können hier zwei Makros zum Einsatz kommen, die jeweils eine Aufgabe erfüllen und somit für den Benutzer sicher zu handhaben sind. Die Komplexität der Software wurde durch das zweite Makro nicht sonderlich beeinflusst, da dieses intern auf Funktionalität zurückgreifen kann, die schon vom ersten genutzt wird.

1.4 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 dies [Einleitungs](#) Kapitel, 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 zu zum Beispiel C ist, dass bei einem Makrosystem der Befehlssatz durch die Anwendung vorgegeben wird, wohingegen der Befehlssatz von C durch die Hardware vorgegeben ist 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 Funktion übersetzt werden. Der Befehlssatz dieses Makrosystems kommt aus dem vorhandenen System, da dort das Command-Pattern [6, S.263] eingesetzt wird und somit ein ideales Interface für Automatisierung bietet. [Unterunterkapitel 2.1.3](#) geht auf die Architektur des vorhandenen Systems ein, die für diese Arbeit wichtig ist.

Für alle Programmiersprachen ist es von Nöten, die Reihenfolge von Zeichen, in sinnvolle Stücke zu zerteilen – dies übernimmt ein Tokenizer, siehe [Abbildung 1](#). Meist wird dieser in den Parser [4, S.46] integriert, welcher die Stücken des der Zeichenkette (String) in ein Format bringt, welches der Interpreter verstehen kann. Das Format ist meist ein *abstrakter Syntaxbaum*² (AST), welcher den String eindeutig repräsentiert. Einer Typen 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

² *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 1: 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 compilierten Programmen, zu vereinen.

2.1.2 Parser

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

- Links nach rechts, links Auflösung (LL) [4, 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, am Ende des Baumes an.
- Links nach rechts, rechts Auflösung (LR) [4, 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 am Anfang des Baumes an. [8]
- 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 geparsed 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 Implementation des Command-Patterns, siehe [Abbildung 2](#).

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. **Clients** sind über den **CommandProvider** in der Lage einen **Invoker** zu bekommen, der ein konkretes **Command** wrapped. Der **Invoker** bekommt bei der Erzeugung über den Konstruktor den momentanen **HistoryStack**. Wenn der **Invoker** durch einen Aufruf aus einem **Command** (auf den **CommandProvider**) entstanden ist, ist es der **HistoryStack** des **Commands**. Ansonsten ist es der **HistoryStack** des **CommandProviders**. Durch den **HistoryStack** ist es möglich die Veränderungen von ausgeführten **Commands** 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 **Commands** registrieren, die in anderen Teilen der Anwendung genutzt werden können.

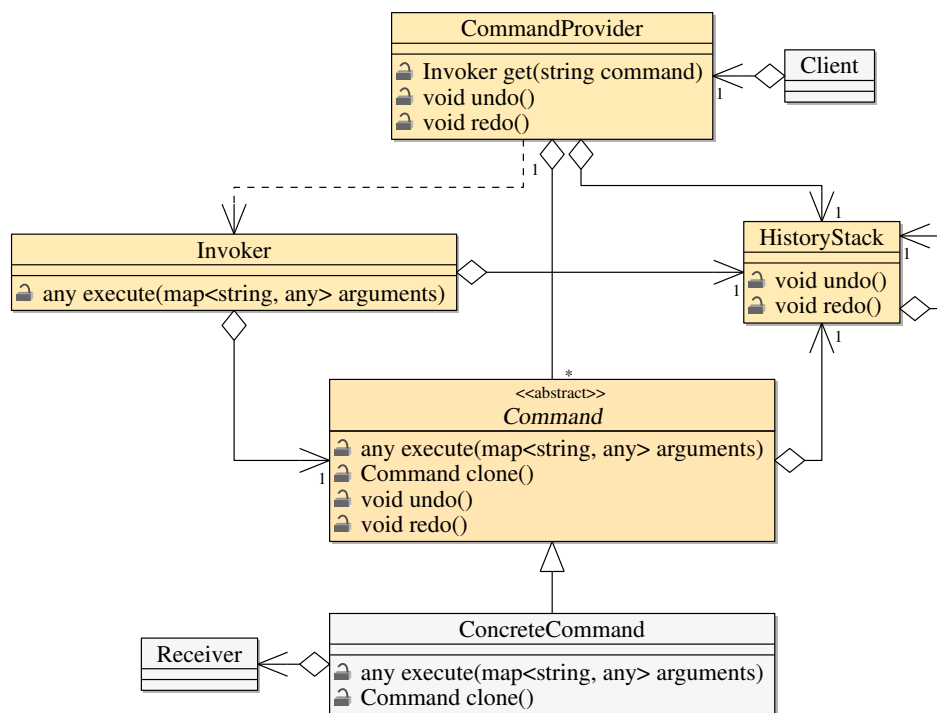


Abbildung 2: Abstrakte Command-PatterImplementation.

Da C++ keine Reflexion [14, 5] unterstützt, ist die Signatur, der `any execute(map<string, any> arguments)` Methode, ein wichtiger Punkt der Architektur. Durch den `any` [1] Typ, in der Implementation des Command-Patterns, wurde die fehlende Reflexion umgangen³. `any` ist in der Lage, jeglichen Typ aufzunehmen und typsicher⁴ 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.

³ Diese Veränderung ist als Vorbereitung auf die Bachelorarbeit entstanden, da diese Anpassung den Zeitrahmen der Bachelorarbeit überschritten hätte.

⁴ Typsicher bedeutet das 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 – dies ist zu vergleichen mit den Variablen von Python. Dies provoziert Fehler, wenn die `ConcreteCommands` aufgerufen werden und der erwartete Typ nicht 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 `Commands` nicht nur hintereinander abgearbeitet werden, ohne dass sie wissen, dass andere `Commands` vor ihnen bzw. nach ihnen ausgeführt werden – wie in [Abbildung 3](#) zu sehen ist.

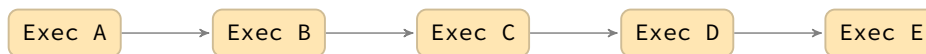


Abbildung 3: Sequenzielles Abarbeiten von Prozessschritten.

[Abbildung 4](#) zeigt den ersten Schritt zu einer nützlichen Implementation – Logik. Hierbei bietet man an, dass der Makro-Entwickler durch Rückgabewerte aus `Commands` entscheiden kann, welche weiteren `Commands` er ausführen möchte.

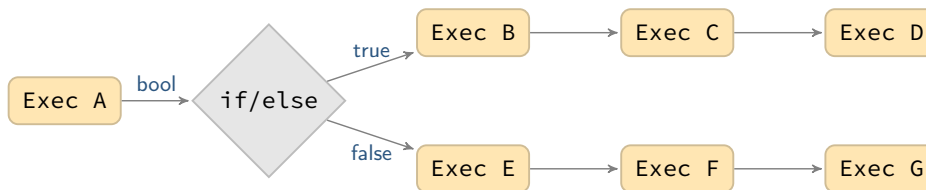


Abbildung 4: Logische Ausdrücke um bedingte Anweisungen zuzulassen.

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 5](#).

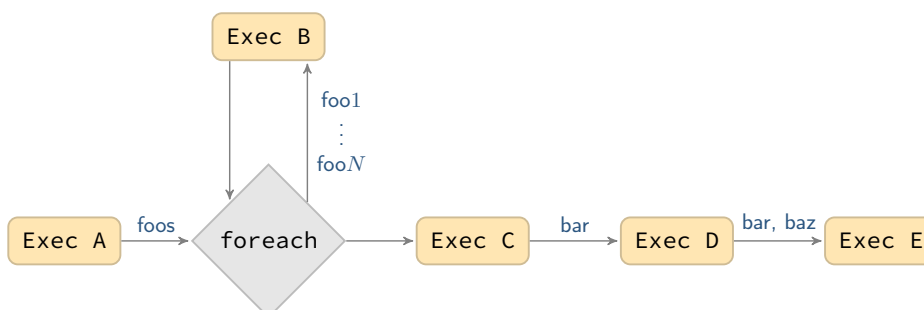


Abbildung 5: Schleife, die Anweisungen für ein Element aus der Liste aufrufen.

Letztendlich kann man sagen, dass ein solches Makrosystem/-sprache eine Programmiersprache mit Interpreter [6, S.274] sein sollte, deren Laufzeitumgebung ein anderes Softwaresystem ist.

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.

Ref Section in implementation /
architecture

3 Konzeption

Dieses Kapitel beschäftigt sich mit der theoretischen Problemlösung.

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

3.1 Syntax

Die Syntax ist an C [2], Python [10], JavaScript [3] und Swift [13] angelegt. C liefert den größten Anteil der Syntax, von Python wurde **def** übernommen, von JavaScript **var** und die *named parameter*⁵ 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 machen wird. Das die Makrosprache *named parameter* unterstützt, liegt daran, dass das Command-Pattern so implementiert wurde, dass Parameter alias Bezeichnungen benutzen können, um eine hohe Kompatibilität zwischen unabhängig entwickelten Modulen zu gewährleisten. Dies ist in der Makrosprache nicht so einfach möglich, somit sind die *named parameter* der bestmögliche Kompromiss, da diese erlauben, die Parameter in beliebiger Reihenfolge anzugeben, was durch die Nutzung von `map<T1, T2>` von Nöten ist.

3.1.1 Syntax Grundlagen

Bezeichner müssen dem regulären Ausdruck (Regex) aus [Abbildung 6](#) entsprechen. 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 vorn herein ausgeschlossen. Außerdem sind Bezeichner, die einem keyword (**break**, **def**, **do**, **else**, **for**, **if**, **print**, **return**, **typeof**, **var**, **while**), einen Booleanwert (**true**, **false**), oder `main` entsprechen – abgesehen von der einen `main` Methode – verboten.

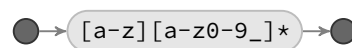


Abbildung 6: Bezeichner

Literals sind entweder Doubles, Integer, Strings oder Boolean Werte, wie [Abbildung 7](#) zeigt. In Strings ist es möglich besondere Zeichen zu escapen, zum Beispiel kann ein

⁵ Named Parameter sind Parameter, die über einen Namen ihren Wert beim Funktionsaufruf zugewiesen bekommen. Die normale Wertzuweisungsstrategie ist, nach der Reihenfolge der Deklaration vorzugehen.

Zeilenumbruch, wie in anderen Programmiersprachen, mit `"\n"` oder ein Tab mit `"\t"`, erzeugt werden.

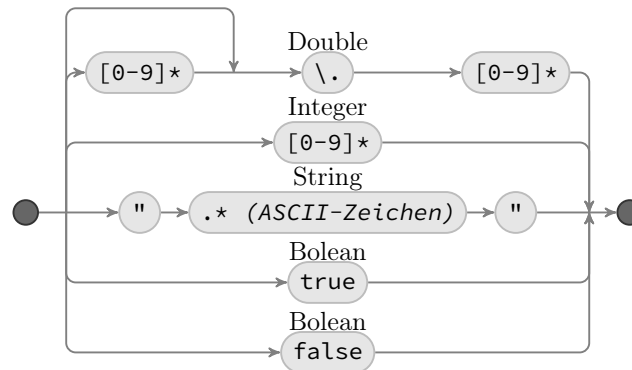


Abbildung 7: Syntax von Literals.

Abbildung 8 zeigt alle Werterzeuger, das sind Konstrukte, die einen Wert für eine andere Operation bereitstellen.

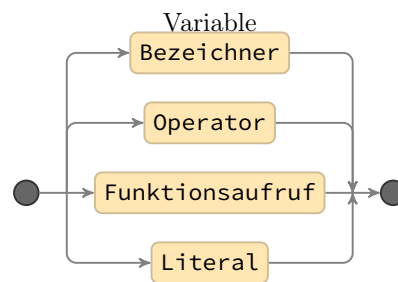


Abbildung 8: Werterzeuger

In Abbildung 9 ist die Syntax von `return` zu sehen.

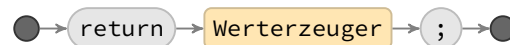


Abbildung 9: Syntax von `return`.

Abbildungen 10, 11 und 12 zeigen den Syntax von einem Scope. Scopes sind Bestandteile von Funktionen und Kontrollstrukturen, wobei sich Loop Scopes von normalen Scopes nur darin unterscheiden, dass sie das `break` Keyword 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 der Syntax `var foo; {var foo;}` richtig ist – das erste `foo`, wird von dem zweiten verdeckt.

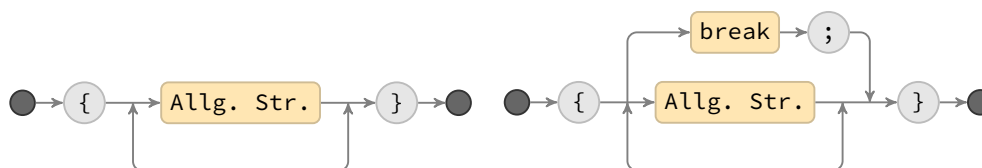


Abbildung 10: Syntax vom Scope. Abbildung 11: Syntax vom Loop Scope.

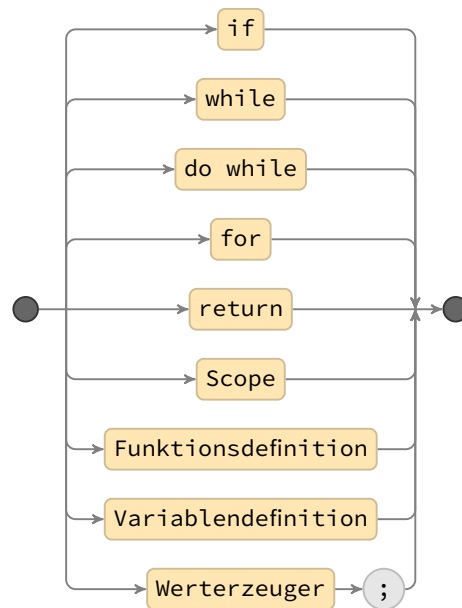


Abbildung 12: Syntax allgemeiner Strukturen.

3.1.2 Definitionen

Variablen können, wie in [Abbildung 13](#) zu sehen ist, definiert werden: **var** foo;. Um anschließend der Variablen einen Wert zuzuweisen, ist es unter anderem erlaubt, dies zu tun: **var** foo = fun(); oder **var** foo = true == false;. Dieser Syntax erlaubt es nicht den Variablen einen Typ zuzuweisen – dieser Teil der Sprache ähnelt deswegen Python und JavaScript sehr.

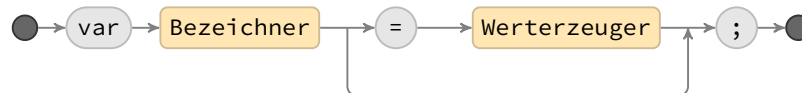


Abbildung 13: Syntax der Variablendeklaration.

Funktionen können über **def** fun(){...} definiert werden, was [Abbildung 14](#) zeigt. Um eine parametrisierte Funktion zu definieren, gibt man die Parameternamen, Komma 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 dem selben Scope definiert werden). Zudem ist es nicht erlaubt, ein Whitespace zwischen dem Bezeichner und der Klammer zu haben.

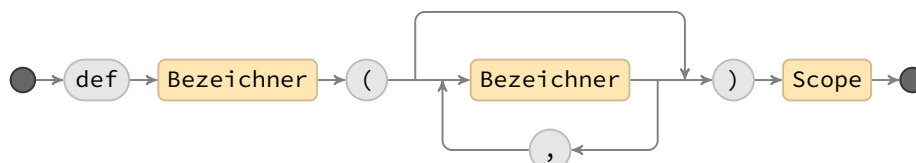


Abbildung 14: Syntax der Funktionsdeklaration.

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

3.1.3 Kontrollstrukturen

Die Syntax von **if/else** und **do-/while** aus den Abbildungen 15, 16 und 17 sollten wie erwartet aussehen.

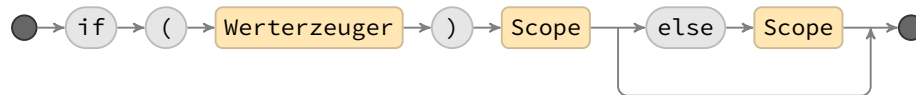


Abbildung 15: Syntax von if.

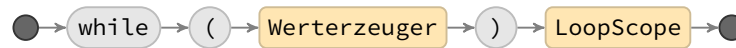


Abbildung 16: Syntax von while.



Abbildung 17: Syntax von do-while.

For, For-each Schleife wenn implementiert

3.1.4 Befehle

Abbildung 18 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 einen Whitespace zwischen dem Bezeichner und der Klammer zu haben.

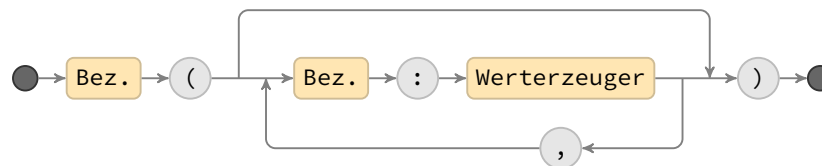


Abbildung 18: Syntax von Funktionsaufrufen.

Die Operator Syntax aus Abbildung 19 folgt, wie die **return** Syntax, den Vorbildern dieser Syntax (C, Python, Javascript, Swift).

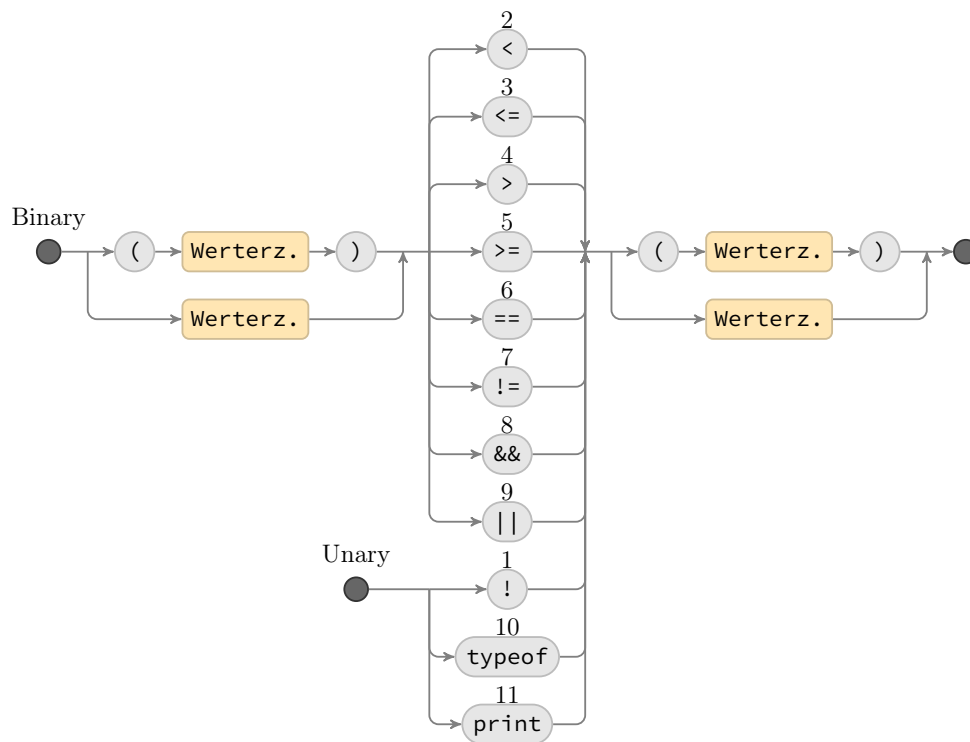


Abbildung 19: Syntax von Operatoren.

Geklammerte Ausdrücke werden zuerst vollständig ausgewertet, bevor der Operator angewendet wird. Das heißt, dass $(a \ || \ b) \ \&\& \ c$ folgender Weise interpretiert wird. Als erstes wird a ausgewertet. Wenn a **false** war, wird b ausgewertet, wenn eins 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 Kurzschlusssemantik. Die Präzedenz der Operatoren ist – abgesehen von $!$ (der die höchste Priorität hat) die Reihenfolge aus der Grafik (siehe Nummerierung).

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 ($1 \rightarrow \text{"int"}$). Zudem gibt es den **print** Operator, der den Wert einer Variable ausgibt (zB. auf die Konsole) und als **string** zurückgibt.

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 20 zeigt die Abhängigkeiten der Pakete (namespaces) in dem Modul, welches die Makrofunktionalität anbieten soll. **core** ist das Paket, indem das Command-Pattern aus dem [Unterkapitel 2.1](#) implementiert ist. Das **pod** Paket enthält alle Klassen, die die nur zur Verwaltung von Daten dienen (separation of concerns) und deswegen *plain old data* (POD), oder auch *passive data structure* (PDS) genannt werden. In dem **pod** Paket

befindet sich die `Token` Klasse und das `ast` Paket, welches alle Klassen, die den abstrakten Syntaxbaum ausmachen, beinhaltet. In dem `parser` Paket befinden sich der Tokenizer und Parser.

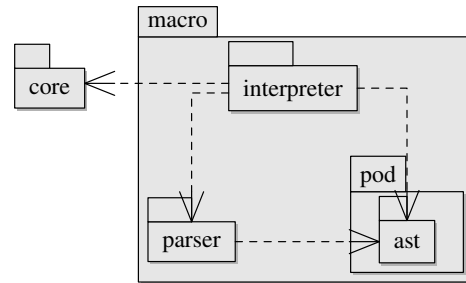


Abbildung 20: Abhängigkeiten von dem Makro Modul.

3.2.1 Token und Parser Paket

Der Parser aus [Abbildung 21](#) bedient sich des Tokenizer, um eine `TokenList` von Tokens zu bekommen. Diese `TokenList` kann der Parser dann parsen, bzw. in einen abstrakten Syntaxbaum umwandeln. Die `TokenList` Klasse dient nur der Erklärung und wird sich nicht in der Implementation wiederfinden, da sie nur ein Array beschreibt.

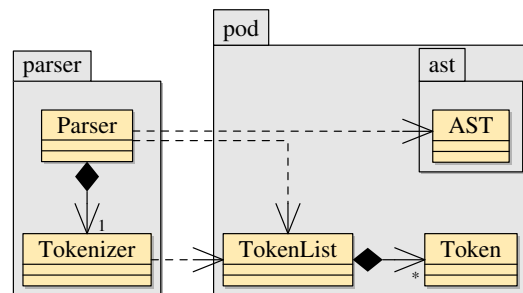


Abbildung 21: Parser Paket UML

Der `Tokenizer` wandelt den String, der das Makro beschreibt, in eine Reihenfolge von Tokens um. Tokens sind alle Zeichen, die von whitespace (`\s*`) getrennt sind, die nicht den Anforderungen als Bezeichner genügen (`[^a-zA-Z0-9_]`, siehe [Abbildung 6](#)), die durch einen Punkt eine Dezimalzahl bilden (`\d*\.\d+`) oder einen String darstellen (`".*?"`⁶). Der `Tokenizer` ist nicht für das Lexen verantwortlich – dies wird von dem Parser übernommen.

Tokens beinhalten die Zeile sowie Spalte als Zahl, und den gesamten Quelltext aus der Zeile, aus der das Token entstanden ist. Dies ist von Nöten, um später gute Fehlermeldungen zu erzeugen, mit denen ein Benutzer schnell weiß, wo er nach dem Fehler suchen muss. Des weiteren enthält die Klasse einen String, der den Teil des Makros enthält, den die Instanz darstellen soll (zB. `if`).

⁶ Dieser Regex funktioniert nur für einfache Varianten (kein escapen) von Strings und dient deswegen nur der Veranschaulichung.

Der `Parser` ist dafür verantwortlich, dass die `TokenList` in einen AST umgewandelt wird. Wie in [Unterunterkapitel 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*, da sie mit Hilfe von Zustandstabellen arbeiten, die meist aus der Backus-Naur-Form heraus entstehen. Der Nachteil bei den 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[\[12\]](#). 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[\[7\]](#) und GCC[\[9\]](#) ein gutes Beispiel und Vorbild, was mit *recursive descent* Parsern erreicht werden kann.

Beim Parsen müssen die Literals umgewandelt werden (Lexen) – aus dem Token `"1.01"` muss der Double `1.01` werden, escape Symbole in einem String müssen umgewandelt werden (z.B.: `"t\tt" → "t t"`).

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 zuweisen⁷).

3.2.2 Abstrakter Syntaxbaum Paket

Wie [Abbildung 22](#) zeigt⁸, sind für alle Konstrukte, die das Scope (siehe [Abbildung 12](#)) aufnehmen kann, Klassen von Nöten. Alle Klassen erben von der AST Klasse, welche ein `Token` als Attribut besitzt. Da die `Token` alle Informationen über den Makro Quelltext haben, und es sich bei der resultierenden Datenstruktur um einen Baum handelt (der Baum ist rekursiv – siehe [Abbildung 23](#)), ist auch der Interpreter in der Lage informative Fehlermeldungen zu generieren.

⁷ Dies ist valider C Code: `const int foo = foo + 1;`

⁸ Es wurden einige Klassen hinter allgemeinen Begriffen – wie `Loop` – versteckt, um etwas Übersicht zu bewahren.

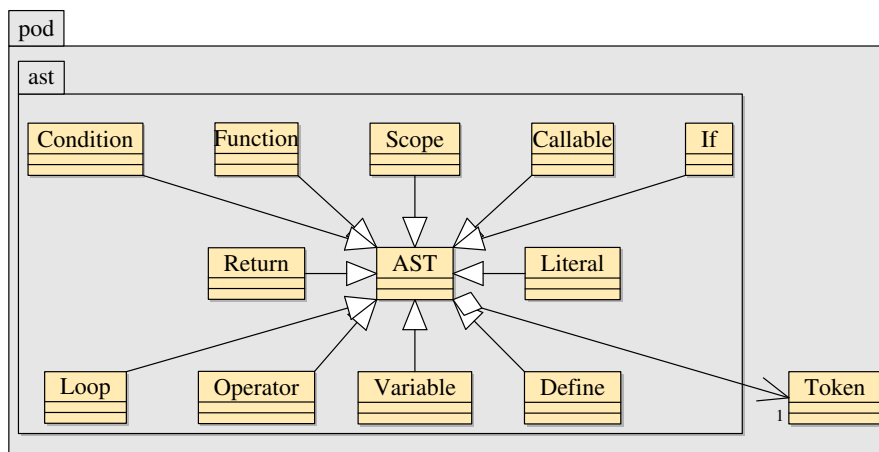


Abbildung 22: Stammbaum der AST Klassen.

Abbildung 23 zeigt die **Scope** Klasse, diese Klasse kann beliebig viele andere AST Instanzen aufnehmen.

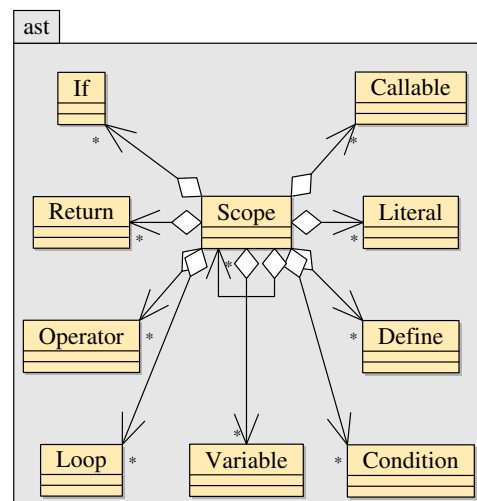


Abbildung 23: Verbindungen vom Scope.

In [Abbildung 24](#) sind die restlichen Abhängigkeiten zwischen den AST Klassen zu sehen.

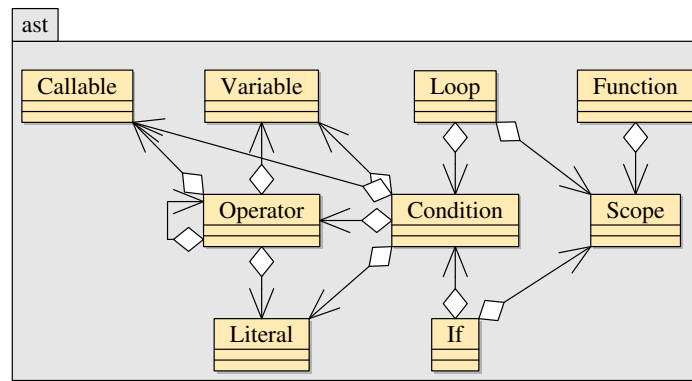


Abbildung 24: Abhängigkeiten der AST Klassen

3.2.3 Interpreter Paket

Der Interpreter nutzt den `parser::Parser`, um einen `ast::Scope` zu erzeugen, wie in [Abbildung 25](#) zu sehen ist.

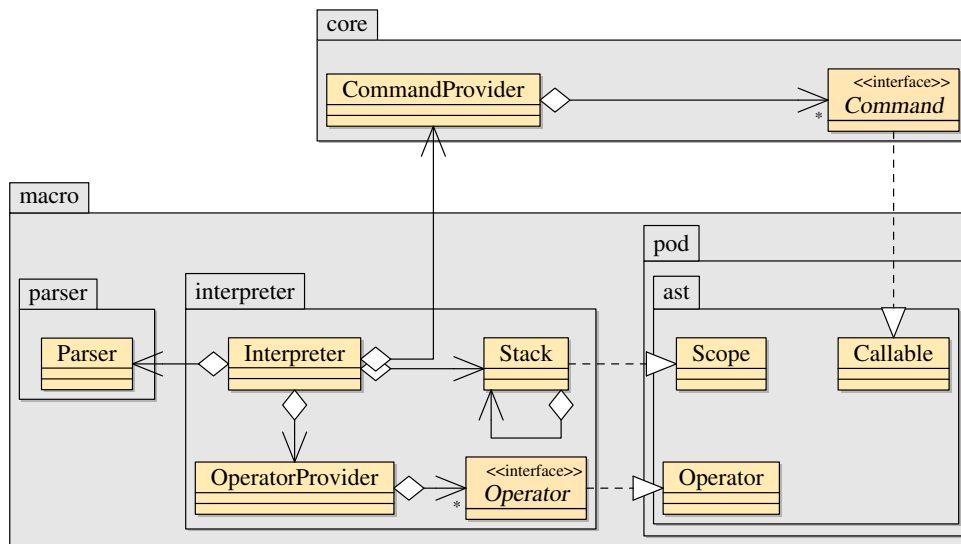


Abbildung 25: Interpreter Beziehungen.

Durch den `OperatorProvider` als Mittelsmann, ist es möglich Operatoren auf spezielle Datentypen anzuwenden, die über die der Literals hinausgehen (siehe [Unterunterkapitel 3.3.3](#)). Die Trennung von `OperatorProvider` und `Interpreter` beruht darauf, dass es nur einen `OperatorProvider` geben muss, aber es viele `Interpreter` geben kann. Zudem müssen so die Operatoren nur ein einziges mal registriert werden.

Während der Ausführung des Makros, durch den `Interpreter`, übernimmt der `Stack` die Verwaltung der definierten Funktionen und Variablen sowie deren Werte. Dadurch repräsentiert der `Stack` die `ast::Scopes`. Wenn ein neues `Scope` in dem AST geöffnet wird, wird ein neuer `Stack` erzeugt, der als vorherigen `Stack` den aktiven `Stack` bekommt. Durch diese Verkettung wird eine Verketteliste aufgebaut, bei der man nur den Kopf

entfernen muss, um ein `ast::Scope` zu beenden, siehe [Abbildung 26](#). In [Unterunterkapitel 3.3.4](#) wird die Architektur weiter ausgeführt.

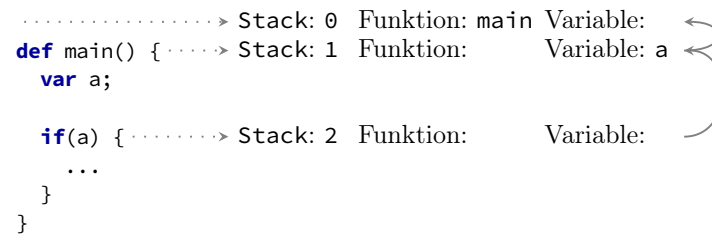


Abbildung 26: Stack Beispiel.

Sollte der `Stack` keine passende Funktion haben, wird der `CommandProvider` nach einem passendem `core::Command` gefragt, wonach dieses in der C++ Ebene ausgeführt wird. Damit realisieren `core::Commands` die `ast::Callable` Funktionen. Funktionen, die in der Makrosprache definiert sind, realisieren ebenso die `ast::Callable`.

Der Interpreter selber läuft den AST ab, erweitert den `Stack` bei `ast::Scopes`, fügt dem `Stack` Variablen und Funktionen bei deren Definition (durch `ast::Define`) zu und popped den `Stack` wenn er fertig ist, ein `ast::Scope` zu interpretieren. `ast::Conditions` werden mit Hilfe des `OperatorProviders` ausgewertet und 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 [Unterunterkapitel 3.3.1](#) die Programmiersprache gewählt, in welcher das Makrosystem entwickelt wird. In den darauffolgenden Kapiteln wird die Architektur des `Parsers`, `OperatorProviders`, `Stack` und des `Interpreters` genauer betrachtet.

3.3.1 Die Programmiersprache

Da das vorhandene System in C++ geschrieben ist, ist es größten Teils überflüssig über andere Programmiersprachen nachzudenken.

Durch das *name mangling*⁹ 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. Die Makros müssen aber auf Daten arbeiten, welche als Objekte von C++ vorliegen. Deswegen müsste der **Stack**, mit dem der **Interpreter** arbeitet, in der C++ Ebene bleiben. Damit

⁹ Beim ‘name mangling’ fügt der Compiler den Funktionsnamen weite Informationen hinzu, um eine eindeutige Funktionssignatur zu erhalten.

würde eine Implementierung von `Tokenizer`, `Parser`, `ast` und `Interpreter` die Wartbarkeit, durch die Teilung und die weitere Programmiersprache deutlich verschlechtern. Zusätzlich ist das Ziel eine Applikation zu automatisieren, und nicht mit komplett neuen Funktionalitäten zu erweitern, was die Vorteile von z.B Python größtenteils zunichte macht. Aufgrund dessen wurde C++ als Programmiersprache gewählt.

Die vorhandene Software ist Cross-Plattform (Windows und Linux) entwickelt. Im Rahmen dieser Bachelorarbeit wird die Software nur auf Linux entwickelt, da die Implementation des C++ Standards von Microsoft zum Teil unvollständig oder auch falsch ist und bei der Fehlersuche meist viel Zeit in Anspruch nimmt [11]. 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.

Dinge die zu berücksichtigen sind:

- dependency circle
Ein dependency circle entsteht, wenn in C oder C++ zwei (oder mehr) Datenstrukturen sich 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 das `Scope` Instanzen von allen AST Klassen aufnehmen kann, und einige Klassen wiederum ein `Scope` besitzen, kommt es hier zu einem dependency circle. Weil das `Scope` eine der meist benutzten Klassen ist, liegt die Auflösung des dependency circles bei den anderen Klassen.
- plain old data
In C++ sind nur Klassen PODs, die trivial sind und standard-layout haben [15, 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, da er nur einen stream von Zeichen aufteilen muss, und sollte daher nicht weiter beschrieben werden müssen. Dies gilt auch für alle `ast` Klassen, da diese POD Klassen sind und somit nur Daten verwalten.

Da der `Parser` recursive decent implementiert wird, kann der `Parser` ohne einen Zustand – sprich Attribute – auskommen. In diesem Fall kann darauf verzichtet werden, den `Parser` als Klasse zu implementieren und anstelle dessen eine `static` Funktion anzubieten. Dies hat den Vorteil, dass der `Parser` garantiert Thread-Safe ist, da alle Daten von dem nativen Funktionsstack verwaltet werden.

Für nahezu alle `ast` Klassen bzw. keywords sollte der `Parser` eine Methode zum parsen haben. Diese Modularität erlaubt es später den `Parser` leichter zu erweitern, bzw. Fehler

lokalisieren zu können. Da jede Methode genau für einen Teil der Syntax zuständig ist und es keinen Objekt Zustand gibt, können die Methoden für sich betrachtet und überprüft werden.

Die Einstiegsmethode des `Parsers` 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 [Unterkapitel 6.1: Punkt 4](#)), der dem Nutzer gezeigt werden kann. Sinnvoll ist es zB. 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 OperatorProvider Architektur

Der `OperatorProvider` ist eine Klasse, deren Aufgabe es ist, Datentypen Funktionen zuzuordnen und zur Verfügung zu stellen. Da die `Operatoren` zustandslos sind¹⁰, 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` promoted¹¹ 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 ist, 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`;

3.3.4 Stack Architektur

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

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

¹⁰ Dies kann nicht garantiert werden, es ist allerdings unwahrscheinlich, dass dies ein Problem ist.

¹¹ Promoted bedeutet, dass ein kleinerer, primitiver Datentyp zu einem größeren implizit konvertiert werden kann. Dies ist immer dann möglich, wenn kein Datenverlust auftritt.

Die Funktionen brauchen nur als konstante Referenzen auf die Funktionsdefinitionen in dem `ast` gespeichert 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 26](#) und [Abbildung 25](#). Diese Pointer werden 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 [Unterkapitel 6.1: Punkt 5](#)) nicht nur die Funktionsdefinition zurückgeben, sondern auch gleich den Pointer auf den `Stack`, in dem die Funktion definiert wurde. Es ist also nicht nötig, die Funktionen zu dem `Stack` Pointer zuzuordnen, in dem sie deklariert worden sind, da dies automatisch geschieht.

3.3.5 Interpreter Architektur

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

Ähnlich wie der `Parser` ist der `Interpreter` recursive decent 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`, ...) 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 heißt, 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. Und 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 aufrufendem 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 **ExecutableProvider** nach einer Funktion gefragt.
Das heißt, dass die nativen Funktionen von Funktionsdefinitionen in dem Makro überschrieben und auch unerreichbar werden können.
- Wenn eine Funktion interpretiert werden muss und diese von dem **ExecutableProvider** 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 aus C++11 anstelle von Pointern anzuwenden, das ist allerdings nicht aus der Makro Ebene möglich.
- Da die Parameternamen zu der Funktionssignatur gehören, ist es möglich mehrere Funktionen mit dem selben Namen anzulegen.
`def fun() {...}` und `def fun(foo) {...}` stehen also nicht in Konflikt. Die Reihenfolge der Parameter ist allerdings nicht Bestandteil der Signatur und somit ist `def fun(a, b) {...}` und `def fun(b, a) {...}` nicht im selben **ast::Scope** zulässig.
- 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. Interpretiert er den **ast::Operator** soweit, dass er einen/zwei Werterzeuger und einen Operortyp (zB. `==`) hat.
2. Erzeugt der **Interpreter** den/die Wert/e des/der Werterzeuger/s.
Im Falle, dass es sich um den Operortyp `&&` oder `||` handelt, kann der Interpreter auch schon durch die Kurzschlusssemantik früher aufhören.
3. Nutzt er den **OperatorProvider** um die beiden Werte mit einem registrierten Operatoren 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 können – in der Lage genau so gute Fehlermeldungen zu produzieren, wie der **Parser**.

3.3.6 Komplexe Rückgabewerte

Die komplexen Rückgabewerte, aus der Makro Ebene in die C++ Ebene sind durch den **any** Typ kein Problem, da sich diese durch den Stack schon von Anfang an in der C++ Ebene befinden. An dieser Stelle muss es sich um eine **ast::Variable** 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 einen Wrapper 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 die selbe Situation, mit der die `core::ConcreteCommands` klarkommen müssen, womit es nur ein kleines Problem ist.

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

Da Makros `core::Commands` aufrufen können, sind sie auch in der Lage andere Makros aufzurufen (siehe [Punkt Unterkapitel 6.1: Punkt 6](#)). Dadurch könnten Makros nicht nur zur Automatisierung genutzt werden, sondern auch zur internen Erweiterung der Applikation, dies ist allerdings eine schlechte Nutzung, da Makros langsamer sind, als reine C++ Funktionen oder `Commands`. Allerdings ist ein wertvoller Vorteil dieser Kombinierbarkeit, dass Makros durch C++ Test-Frameworks ausgiebig getestet werden können.

4 Exemplarische Realisierung

```

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

```

Listing 2: Interpreter Beziehungen.

Ich hätte hier vor, an Hand eines Beispielen, die “Knackpunkte” der einzelnen Klassen zu erklären, bzw. den Ablauf wieder zu spiegeln (String->Tokens->Parser/AST->Interpreter). Hört sich das gut an? (Code wäre getrimmt oder Pseudo-Code)

4.1 Tokenizer

Erklärung

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

Beispiel

```

1 def
2 fun
3 (
4   foo
5 )
6 {
7   while
8   (
9     !
10    foo
11   )
12 {
13   foo
14 =
15   foo
16 +
17   1
18 ;
19 }
20 return

```

Listing 3: Tokenized Makro

4.2 Parser

Lorem ipsum dolor sit amet ...

```

1 @Scope {

```

```
2  l:0 c:0 t:
3  @Define {
4    l:1 c:1 t:def
5    @Function {
6      l:1 c:5 t:fun
7      parameter:
8        @Variable {
9          l:1 c:9 t:foo
10       }
11    @Scope {
12      l:1 c:13 t:{
13        @While {
14          l:2 c:3 t:while
15          Contition:
16            @UnaryOperator {
17              l:2 c:9 t:!
18              Operation:not
19              Operand:
20                @Variable {
```

Listing 4: Geparsed Makro

4.3 Interpreter

Lorem ipsum dolor sit amet ...

Lorem ipsum dolor sit amet ...

4.4 Fehlermeldungen

Ich vermute das ich in diesem Kapitel 6-8 Seiten zu schreiben habe, wenn es Vorschläge gibt, was noch niedergeschrieben werden sollte, bzw. ausführlicher erklärt werden muss, nur zu. Ich habe wie immer eher das Problem das ich zu wenig Text produziere.

5 Evaluation

Ich habe <http://chaiscript.com/> gefunden. Quasi das was ich hier mache, bloß das die 9 Jahre Zeit hatten – ich habe also einen Referenzpunkt! :)

6 Zusammenfassung und Ausblick

6.1 Ausblick

1. Threading

Sofern alle Funktionen die ausgeführt werden sollen als thread-safe gekennzeichnet sind, kann das gesamte Macro parallel ausgeführt werden.

2. Debugger / Stepping

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

3. C++17 std::string_view

Um weniger Speicher zu verbrauchen und durch weniger Memory Allokationen schneller beim Tokenizen zu sein.

4. Mehr Fehler von dem Parser

Anstelle, dass der Parser Exceptions nutzt und nach dem ersten Fehler aufhört zu parsen, ist es möglich 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 nach dem wo der Fehler aufgetreten ist – die Tokens verworfen werden und dann weiter geparkt werden.

Eine weitere Variante ist es, AST Elemente als poisoned zu kennzeichnen, was alle Fehler, die in Verbindung mit dem Element auftreten, verworfen werden, da es sehr wahrscheinlich eine Folgefehler ist.

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

5. 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 Stacks zu speichern, und die Stacks Nutzen anstelle eines Pointers einen Offset, von dem sie aus die anderen Stack fragen können. Der Vorteil von Arrays gegen LinktLists ist die wesentlich höhere Cache-Locality

6. 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 geparkten AST assoziiert. Der AST wird von dem **Interpreter** nicht verändert, weswegen das Makro nur geparsed 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, da der **Interpreter** in den AST des anderen Makros weiter parsen würde. Dafür müsste der **Interpreter** natürlich ein wenig angepasst werden, da er nun den 'MakroProvider' vor dem **CommandProvider** nach einem passenden Makro fragen müsste.

Literatur

- [1] *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).
- [2] *C Workgroup*. 2016. URL: <http://www.open-std.org/jtc1/sc22/wg14/> (besucht am 09.04.2016).
- [3] *ECMA Standard*. 2015. URL: <http://www.ecma-international.org/publications/standards/Ecma-262.htm> (besucht am 11.04.2016).
- [4] Helmut Eirund, Bernd Müller und Gerlinde Schreiber. *Formale Beschreibungsverfahren der Informatik: ein Arbeitsbuch für die Praxis*. Springer-Verlag, 2013.
- [5] Jacques Ferber. „Computational reflection in class based object-oriented languages“. In: *ACM Sigplan Notices*. Bd. 24. 10. ACM. 1989, S. 317–326.
- [6] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.
- [7] LLVM. *Clang Features*. LLVM. 2016. URL: <http://clang.llvm.org/features.html> (besucht am 09.04.2016).
- [8] 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).
- [9] Joseph Myers. *GCC New C Parser*. GCC Wiki. 2016. URL: https://gcc.gnu.org/wiki/New_C_Parser (besucht am 09.04.2016).
- [10] *Python*. 2016. URL: <https://www.python.org/> (besucht am 09.04.2016).
- [11] *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).
- [12] Elizabeth Scott und Adrian Johnstone. „GLL parsing“. In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010), S. 177–189.
- [13] *Swift*. 2016. URL: <https://developer.apple.com/swift/> (besucht am 09.04.2016).
- [14] Steve Vinoski. „A time for reflection“. In: *Internet Computing, IEEE* 9.1 (2005), S. 86–89.
- [15] *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

1	Abstraktes Ziel der resultierenden Architektur.	10
2	Abstrakte Command-PatterImplementation.	11
3	Sequenzielles Abarbeiten von Prozessschritten.	12
4	Logische Ausdrücke um bedingte Anweisungen zuzulassen.	12
5	Schleife, die Anweisungen für ein Element aus der Liste aufrufen.	12
6	Bezeichner	14
7	Syntax von Literals.	15
8	Werterzeuger	15
9	Syntax von return.	15
10	Syntax vom Scope.	15
11	Syntax vom Loop Scope.	15
12	Syntax allgemeiner Strukturen.	16
13	Syntax der Variablendeklaration.	16
14	Syntax der Funktionsdeklaration.	16
15	Syntax von if.	17
16	Syntax von while.	17
17	Syntax von do-while.	17
18	Syntax von Funktionsaufrufen.	17
19	Syntax von Operatoren.	18
20	Abhängigkeiten von dem Makro Modul.	19
21	Parser Paket UML	19
22	Stammbaum der AST Klassen.	21
23	Verbindungen vom Scope.	21
24	Abhängigkeiten der AST Klassen	22
25	Interpreter Beziehungen.	22
26	Stack Beispiel.	23

Listingverzeichnis

1	Clang Fehlermeldung.	13
2	Interpreter Beziehungen.	29
3	Tokenized Makro	29
4	Geparsed Makro	30

Anhänge