Liste der noch zu erledigenden Punkte

Hochschule Bremen

Bachelorarbeit

Thesis

Konzeption und Implementierung einer Makrosprache in C++

Autor:
Roland Jäger
360 956

9. April 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 9. April 2016 _____

Roland Jäger

Danksagung

Danke an Mutter, Vater und alle Profs

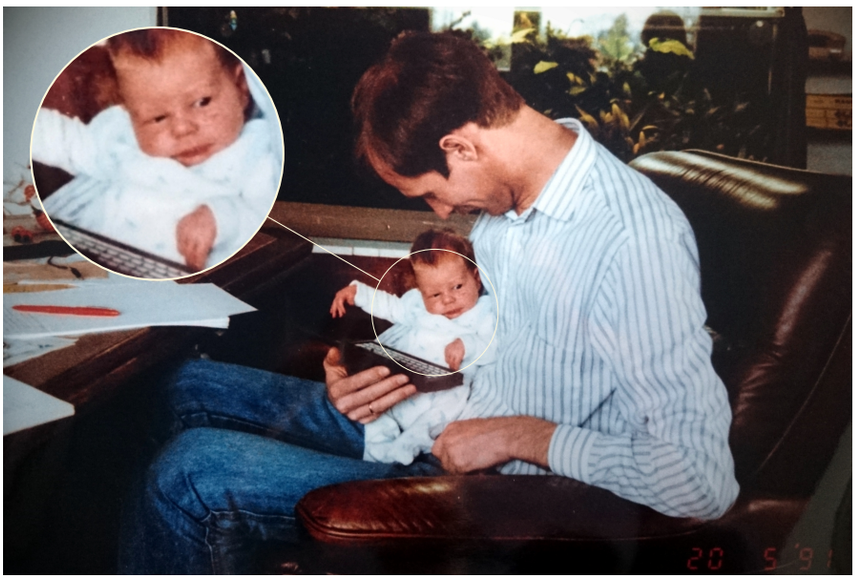

Abbildung 1: *“Das soll meine Zukunft sein?”*

1 Einleitung

Die Einführung von Automatisierung in ein Softwaresystem ist vergleichbar mit den Maschinen, die in der industrielle 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.

Text Größe, Zeilenabstand?

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 festest 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.

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. Somit ist ein Makrosystem eher mit einer Sprache zu vergleichen, die sich einer virtuellen Maschine bedient – wie Java – als mit C.
Der Befehlssatz dieses Makrosystems kommt aus dem vorhandenen System, da dort das *Command-Pattern*[4, S.263] eingesetzt wird und somit ein ideales Interface für Automatisierung bietet.

Für alle Programmiersprachen ist es von Nöten, die Reihenfolge von Zeichen (`string`), in sinnvolle Stücke zu zerteilen – dies übernimmt ein Tokenizer. Meist wird dieser in den Parser[2, S.46] integriert, welcher die Stücken des Strings in ein Format bringt, welches der *Interpreter*[4, S.274] verstehen kann. Das Format ist meist ein *abstrakter Syntaxbaum*[2] (AST), welcher den String eindeutig repräsentiert. Der Interpreter arbeitet dann nur noch mit dem AST, welcher vorgibt welche Befehle der Interpreter ausführen muss, um das als String angegebene Programm auszuführen.

Das folgende ist ziemlich komprimiert – muss das noch ausführlicher durchgekaut werden (Kapitel Grundlagen?), oder ist das 'Allgemeinwissen'?

Hierfür gib's doch sicherlich Bonuspunkte ;)

Das vorhandene System ist in C++ geschrieben, weswegen es sich größten Teils erübrigt über andere Programmiersprachen nachzudenken. Durch das *name mangling*[3] ist es schwierig eine API von C++ zu anderen Programmiersprachen anzubieten – meistens geschieht dies über eine C API. 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 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.

Die vorhandene Software ist Corss-Plattform (Windows und Linux) entwickelt. Im Rahmen dieser Bachelorarbeit wird die Software nur auf Linux entwickelt, da die *Standard* Implementation von Microsoft unberechenbar ist und bei der Fehlersuche meist viel Zeit in Anspruch nimmt. 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.

2.2 Anforderungen an die angestrebte Lösung

Die Probleme, ein Makrosystem/-sprache in C++ zu implementieren, fangen dann an, wenn man von den Makros will, dass diese nicht nur hintereinander abgearbeitet werden,

[2] *Abstract syntax tree* ist eine digitale Darstellung einer Programmiersprache.
[3] Beim 'name mangling' fügt der Compiler den Funktionsnamen weite Informationen hinzu, um eine eindeutige Funktionssignatur zu erhalten.

ohne dass sie wissen, dass andere Makros vor ihnen bzw. nach ihnen ausgeführt werden – wie in Abbildung 2 zu sehen ist.

Abbildung 2: Sequenzielles Abarbeiten von Prozessschritten.

Abbildung 3 zeigt den ersten Schritt zu einer nützlichen Implementation – Logik. Hierbei bietet man an, dass der Makro-Entwickler durch Rückgabewerte aus Makros entscheiden kann, welche weiteren Makros er ausführen möchte.

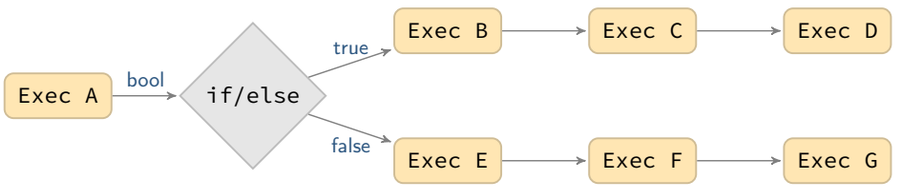

Abbildung 3: Logische Ausdrücke um bedingte Anweisungen zuzulassen.

Obwohl man mit solchen Makros schon einige Probleme lösen kann, ist es nicht das, was man zur Verfügung haben will, wenn man mit objektorientierten Sprachen arbeitet bzw. mehr als ein *'ja'* oder nein *'nein'* braucht.

Was ein Makrosystem/-sprache anbieten muss ist, dass Instanzen von verschiedenen Klassen/Typen zurückgegeben und beliebig viele Parameter (unterschiedlicher Klassen/Typen) dem Makro mitgegeben werden können. Leider sind gerade diese Punkte ein Problem in C++, weil C++ keine Reflexion[9, 3] unterstützt. Dies ist durch eine Implementation[4] des Command-Patterns gelöst worden. Somit kommt in diesem Schritt 'nur' noch hinzu, dass es Schleifen, sowie komplexe Parameter und Rückgabewerte geben kann, siehe Abbildung 4.

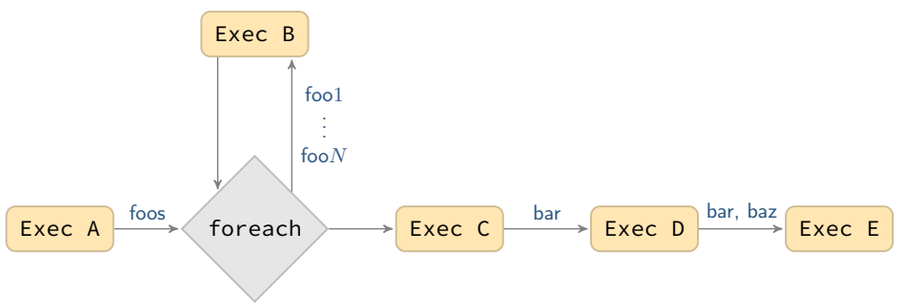

Abbildung 4: Schleife, die Anweisungen für ein Element aus der Liste aufrufen.

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

[4] Das Command-Pattern wurde mit Hilfe des `any`[5] sowie `optional`[6] Typs implementiert.
[5] `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3804.html`
[6] `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3793.html`

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

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

Code 1: Clang Fehlermeldung.

3 Konzeption

Dieses Kapitel beschäftigt sich mit der theoretischen Problemlösung.

Unterkapitel 3.1 beschreibt die Syntax, welcher von dem Tokenizer und Parser umgewandelt werden soll. Im Unterkapitel 3.2 wird hauptsächlich die Architektur des Makrosystems beschrieben. Diese Grundarchitektur umfasst die grobe Architektur des gesamten Makrosystems. In dem Kapitel Detaillierte Teilarchitekturen wird genauer auf die komplexeren Teile der Architektur eingegangen.

3.1 Syntax

Die Syntax ist an C[7], Python[8], JavaScript[9] und Swift[10] angelegt. C liefert den größten Anteil der Syntax, von Python wurde `def` übernommen, von JavaScript `var` und der named parameter Syntax `fun(foo:gun());` von Swift. Die Unterscheidung zwischen `def` und `var` sorgt dafür, dass der Programmierer nach den ersten drei Zeichen weiß, was der Folgende Code machen wird. Das die Makrosprache *named parameter*[11] 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 sie named parameter der bestmögliche Kompromiss, da diese erlauben, die Parameter in beliebiger Reihenfolge anzugeben.

3.1.1 Syntax Grundlagen

Bezeichner müssen dem Regex aus Abbildung 5 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

[7] http://www.open-std.org/jtc1/sc22/wg14/
[8] https://www.python.org/
[9] http://www.ecmascript.org/
[10] https://developer.apple.com/swift/
[11] Named Parameter sind Parameter, die über einen Namen ihren Wert beim Funktionsaufruf zugewiesen bekommen. Die Normale Wertzuweisungsstrategie ist nach der Reihenfolge der Deklaration vor zu gehen.

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 entsprechen (**def**, **var**, **if**, **else**, **for**, **do**, **while**, **break**, **return**), einen Booleanwert entsprechen (`true`, `false`), oder `main` gleichen, verboten.

Update der keywords

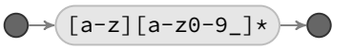

Abbildung 5: Bezeichner

Abbildung 6 zeigt alle Werterzeuger, dass sind Konstrukte, die einen Wert für eine andere Operation bereitstellen.

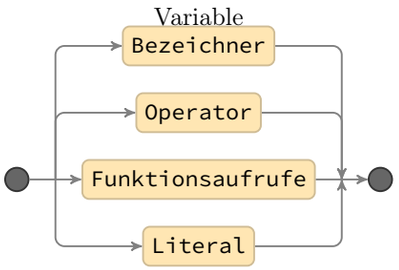

Abbildung 6: Werterzeuger

In Abbildung 7 ist die Syntax von **return** zu sehen.

Abbildung 7: Syntax von return.

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.

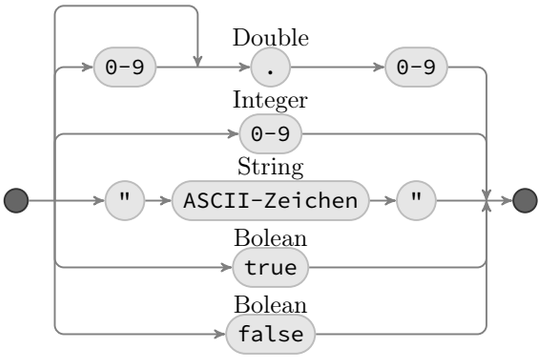

Abbildung 8: Syntax von Literals.

Abbildungen 9, 10 und 11 zeigen den Syntax von einem Scope. Scopes sind Bestandteile von Funktionen und Loops, wobei sich Loop Scopes von normalen Scopes nur darin unterscheiden, dass sie das `break` Keyword unterstützen. Alle Scopes – abgesehen von Funktion Deklarationsscopes – die auf ein Loop Scope folgen, 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.

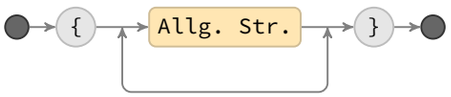

Abbildung 9: Syntax vom Scope.

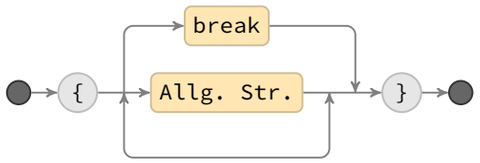

Abbildung 10: Syntax vom Loop Scope.

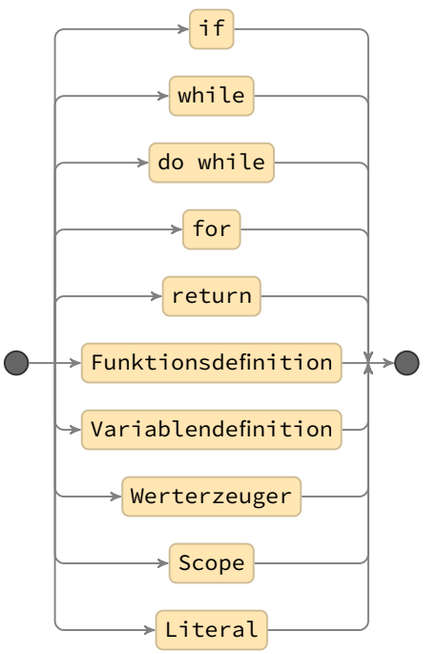

Abbildung 11: Syntax allgemeiner Strukturen.

3.1.2 Definitionen

Variablen können, wie in Abbildung 12 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;`.

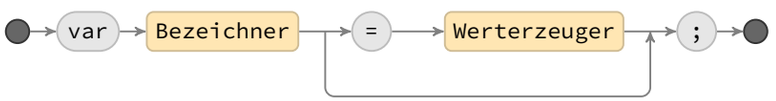

Abbildung 12: Syntax der Variablendeklaration.

Funktionen können über `def fun(){...}` definiert werden, was Abbildung 13 zeigt. Um eine parametrisierte Funktion zu definieren, gibt man die Parameternamen, Komma getrennt, nach dem Funktionsnamen an: `def fun(foo, bar){...}`. Es ist nicht erlaubt, ein Whitespace zwischen dem Bezeichner und der Klammer zu haben. 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.

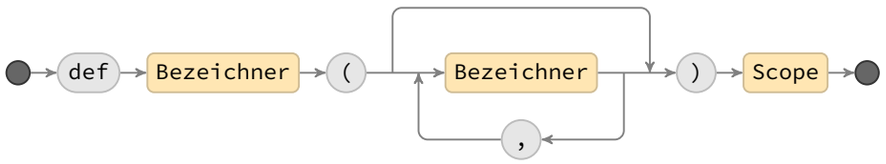

Abbildung 13: Syntax der Funktionsdeklaration.

3.1.3 Kontrollstrukturen

Die Syntax von `if`/`else` und `do`-/`while` aus den Abbildungen 14, 15 und 16 sollten wie erwartet aussehen.

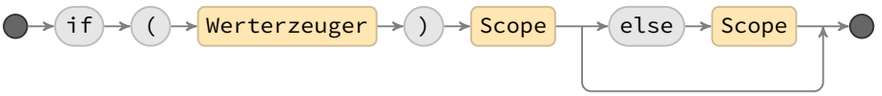

Abbildung 14: Syntax von if.

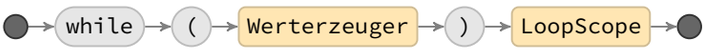

Abbildung 15: Syntax von while.

Abbildung 16: Syntax von do-while.

For, For-each Schleife wenn implementiert

3.1.4 Befehle

Abbildung 17 zeigt die Syntax, um eine definierte Funktion aufzurufen. `fun(foo:gun(), ↪ bar:foo);` weißt dem `foo` Parameter den Wert von `gun()` zu und dem Parameter `bar` wird der Wert von `foo` aus dem Scope zugewiesen.

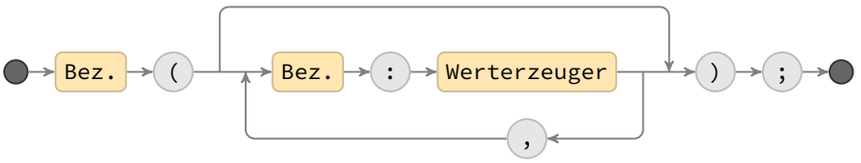

Abbildung 17: Syntax von Funktionsaufrufen.

Die Operator Syntax aus Abbildung 18 sollte, ähnlich wie die `return` Syntax, nicht allzu überraschend sein. Geklammerte Ausdrücke werden zuerst vollständig ausgewertet, bevor der Operator angewendet wird. Das heißt, dass `(a || b) && c` folgender Weise interpretiert wird. `a` oder `b` wird zuerst ausgewertet und deren Ergebnis wird mit `c` genutzt. Entgegen dessen wird bei `a || b && c` zuerst `a`, und dann `b` und `c` ausgewertet.

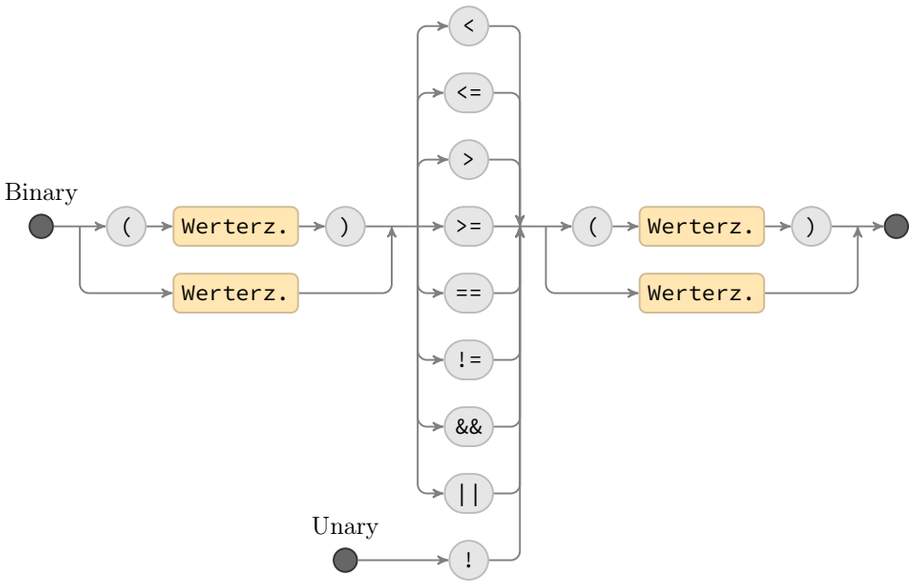

Abbildung 18: Syntax von Operatoren.

Wenn implementiert ===, !==, typeof, print

3.2 Grundarchitektur

Das Ziel der folgenden Architektur ist,die Funktionalität, die in Abbildung 19 zu sehen ist, zu ermöglichen.

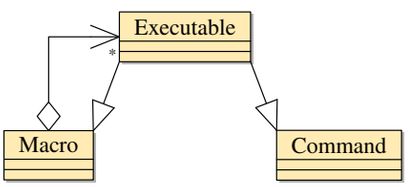

Abbildung 19: Abstraktes Ziel der resultierenden Architektur.

`Executable` ist das Interface, welches alle `Commands` aus dem Command-Pattern implementieren. Eine weitere Implementation dieses Interfaces ist – nicht zu verwechseln mit Ma*k*ros – die `Macro` Klasse. Diese erlaubt es Entwicklern ein Makro auszuführen. Im weiteren Verlauf dieser Arbeit geht es hauptsächlich um die Ma*k*ros, da sich die Implementation der `Macro` Klasse am Ende aus den Teillösungen ergibt.

Am Ende erwähnen /& referenzieren.

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 Makro Funktionalität anbieten soll. Das `ast` Paket beinhaltet alle Klassen, die den abstrakten Syntaxbaum ausmachen und von dem Parser und dem Interpreter verwendet werden. In dem `parser` Paket befinden sich der Tokenizer und Parser. In dem `token` Paket befindet sich die `Token` Klasse, da diese in kein anderes Paket gehört.

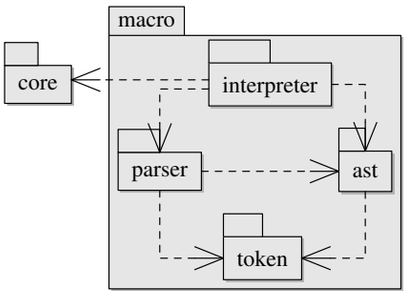

Abbildung 20: Abhängigkeiten von dem Makro Modul.

3.2.1 Token und Parser Paket

Der `Parser` aus Abbildung 21 bedient sich dem `Tokenizer`, um eine `TokenList` von `Token`s zu bekommen. Diese `TokenList` kann er dann parsen, bzw. in einen abstrakten Syntaxbaum umwandeln.

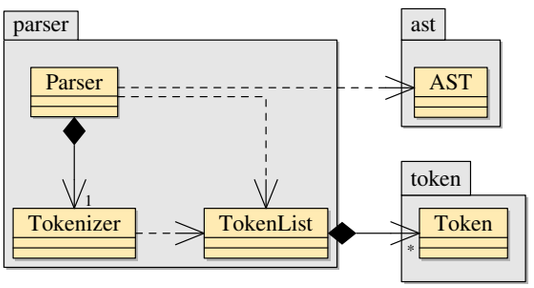

Abbildung 21: Parser Paket UML

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 5), die durch einen Punkt eine Kommazahl bilden (\d*\.\d+) oder einen String darstellen (".*?" [12]). Der `Tokenizer` ist nicht für das Lexen verantwortlich – dies wird von dem Parser übernommen.

Fachwort?

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

`Token`s 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.

Der `Parser` ist dafür verantwortlich, dass die Liste von Tokens in einen AST umgewandelt wird. Es gibt drei Hauptarten von Parsern[2, S.77 f.] *LL*[13], *LR*[14] und *recursive descent*[15]. 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[7]. 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[16] und GCC[17] ein gutes Beispiel und Vorbild, was mit recursive descent Parsern erreicht werden kann.

Ist es ok, dass ich die Parser nur kurz in den Fußnoten erklärt habe (Grundlagen Kapitel?)

Beim Parsen müssen die Literals umgewandelt werden – aus dem Token `"1.01"` muss der Double `1.01` werden, escapete Symbole in einm String müssen umgewandelt werden (z.B.: `"t\tt"`→`"t  t"`). Die Operatoren müssen in der richtigen Reihenfolge zusammengestellt werden (bei `a || b && c` muss `a` zuerst ausgewertet werden und dann `b && c`, was das Baum wiedergeben muss). Und Variablen Deklarationen mit Wertzuweisung müssen in zwei Schritte aufgeteilt werden (erst deklarieren und dann der Variablen den Wert zuweisen[18]).

Gibt es ein besseres Beispiel?

3.2.2 Abstrakter Syntaxbaum Paket

Wie Abbildung 22 zeigt, sind für alle Keywords[19] Klassen von Nöten. Alle Klassen erben von der `AST` Klasse, welche ein `Token` besitzt, welches die spezielle Klasse beschreibt. Da die Token alle Informationen über den Makro Quelltext haben und es sich bei der resultierenden Datenstruktur um einen Baum handelt, ist auch der Interpreter in der Lage informative Fehlermeldungen zu generieren.

[13] LL Parser arbeiten 'vorwärts', links nach rechts und probieren auf der linken Seite, des gelesenen bzw. teils geparsten Quelltextes zu reduzieren. Man gelangt am Ende, am Ende des Baumes an.
[14] 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 am Ende zu dem Anfang des Baumes.[5]
[15] Recursive descent Parser bauen durch Funktionsaufrufe einen Stack auf, der den Zweigen des Baumes gleicht.
[16] `http://clang.llvm.org/features.html`
[17] `https://gcc.gnu.org/wiki/New_C_Parser`
[18] Dies ist valider C Code: `const int foo = foo + 1;`
[19] Es wurden einigele Klassen hinter allgemeinen Begriffen – wie `Loop` – versteckt, um etwas Übersicht zu bewahren.

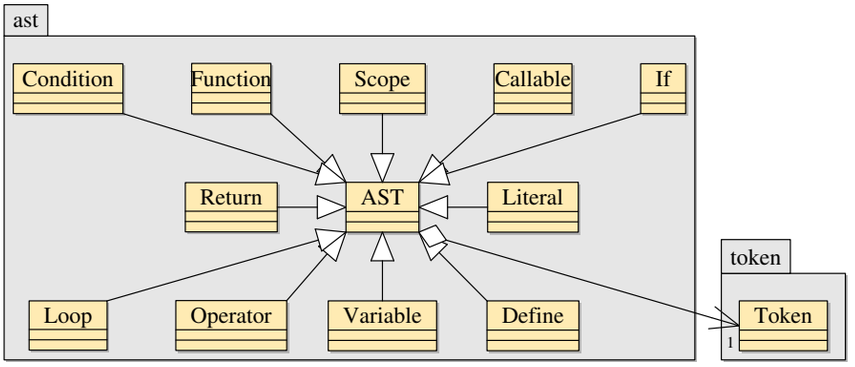

Abbildung 22: Stammbaum der AST Klassen.

Abbildung 23 zeigt die `Scope` Klasse, diese Klasse kann beliebig viele andere `AST` Instanzen aufnehmen. Da das `Scope` Instanzen von allen `AST` Klassen aufnehmen kann, und einige Klassen wiederum ein `Scope` besitzen, kommt es hier zu einem dependency circle. Da das `Scope` eine der meist benutzten Klassen ist, liegt die Auflösung des dependency circles bei den anderen Klassen.

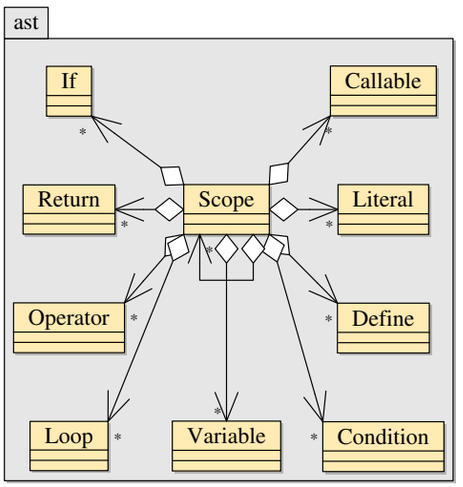

Abbildung 23: Verbindungen vom Scope.

`Loop`, `Function` und `If` müssen den dependency circle mit `Scope` auflösen. In Abbildung 24 sind die restlichen Abhängigkeiten zwischen den `AST` Klassen zu sehen.

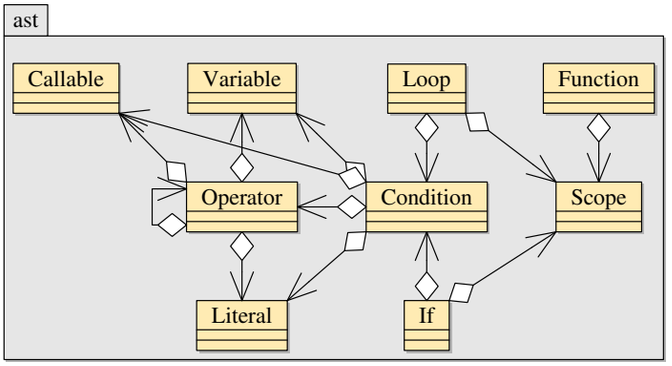

Abbildung 24: Abhängigkeiten der AST Klassen

3.2.3 Interpreter Paket

Der `Interpreter` nutzt den `parser::Parser`, um einen `ast::Scope` zu erzeugen. Diesen Baum interpretiert der `Interpreter` in mehreren Schritten. Als erstes werden alle Funktionsdefinitionen interpretiert – dies sorgt dafür, dass keine Definitionsreihenfolge eingehalten werden muss. Im zweiten Schritt werden die globalen Variablen definiert und initialisiert. Der Letzte Schritt ist die Abarbeitung der Einstiegsfunktion mit allen Deklarationen und Funktionsaufrufen. Das 'root' `ast::Scope` wird normal behandelt – das heißt, dass Funktionsaufrufe vor der `main()` Methode erlaubt sind.

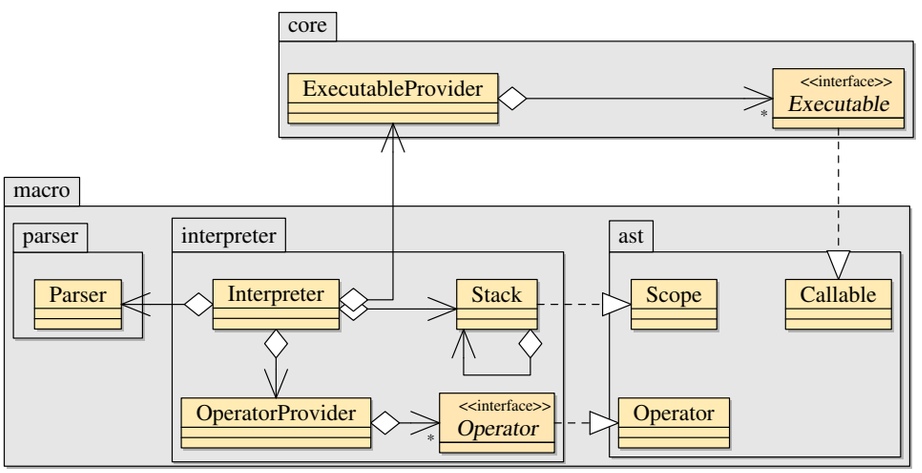

Abbildung 25: Interpreter Beziehungen.

Durch den `OperatorProvider` als Mittelmann, ist es möglich Operatoren auf spezielle Objekte anzuwenden, die über die Datentypen der Literals hinausgehen. Da die Parameternamen zu der Funktionssignatur gehören, ist es möglich mehrere Funktionen mit dem selben Namen anzulegen – `fun()` und `fun(foo)` stehen also nicht in Konflikt. Das zuweisen von Parametern verhält sich ähnlich wie bei JavaScript – die Parameter werden als Referenz übergeben und nicht kopiert. Wenn dem Parameter ein neuer Wert zugewiesen wird, verändert sich der Wert aus dem aufrufendem Scope nicht. Wenn dei aufzurufende Funktion allerdings von dem `ExecutableProvider` kommt, werden die

Parameter kopiert – dies ist durch die C++ Ebene vorgegeben und wird normalerweise durch Pointer beschleunigt.

Während der Ausführung übernimmt der `Stack` die Verwaltung der definierten Funktionen und Variablen sowie deren Werte. Bei jedem `Scope`, wird der `Stack` um eine Instanz von sich selber erweitert.

3.3 Detaillierte Teilarchitekturen

In diesem Abschnitt werden die komplexeren Bestandteile der Architektur beschrieben. 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 *plain old data* (POD) Klassen sind und somit nur Daten verwalten.

3.3.1 Parser Architektur

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 Paket globale 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 `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 Ausblick: 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.

3.3.2 OperatorProvider Architektur

Der `OperatorProvider` ist eine Klasse, deren Aufgabe es ist, Datentypen Funktionen zuzuordnen und zur Verfügung zu stellen. Da die `Operatoren` zustandslos sind[20], 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[21] werden kann, wie es in den meisten typisierten Programmiersprachen der Fall ist.

implizite convert operatoren? oder auch explizite?

3.3.3 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 einen anderen `Stack` erlauben.

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

Anderes Wort?

Letztlich haben die `Stack` Instanzen einen Pointer auf den `Stack` 'über' ihnen. Diesen 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 Ausblick: 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 einen `Stack` Pointer zuzuordnen, da dies automatisch geschieht.

3.3.4 Interpreter Architektur

Die Architektur des `Interpreters` ist durch den `ast` und `Parser` relativ simpel. Außerdem lösen `OperatorProvider` und `Stack` die komplexesten Teile des `Interpreters`. Der `Interpreter` ist ähnlich wie der `Parser` recursive decent implementiert.

Für jede `ast` Klasse braucht der `Interpreter` eine `interpret()` Methode. Durch den Aufbau der `ast` Klassen, endet der `Interpreter` 'immer' an einem Werterzeuger (Funktionsaufruf, Variable, ...), der von Kontrollstrukturen (**`if`**, **`while`**, ...) konsumiert wird.

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

Wenn der `Interpreter` eine Variable braucht, muss er nur den `Stack` fragen. Wenn eine Funktion interpretiert werden soll, muss der `Interpreter`Variablen, die als Parameter übergeben werden, dem `Stack` der Funktion als Referenzen hinzufügen und anschließend das Scope der Funktion interpretieren. Wenn der `Stack` keine entsprechende Funktion besitzt, wird der `ExecutableProvider` nach einer Funktion gefragt.

Wenn der `Interpreter` einen `Operator` zu interpretieren hat, kann dies in drei einfachen Schritten geschehen. Operatoren sind die kompliziertesten Strukturen und dienen daher als Beispiel, wie simpel der `Interpreter` letztendlich geworden ist. Im ersten Schritt interpretiert er den `Operator` soweit, dass er einen/zwei Werterzeuger und einen Operatortyp (zB. `==`) hat. Im zweiten Schritt wird der Werterzeuger zu einem Wert ausgewertet, mit dem im dritten Schritt, dann der `Operator` vom `OperatorProvider` aufgerufen wird.

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`.

3.3.5 Komplexe Rückgabewerte

Der Übergang von der C++ Ebene zu der Makro Ebene schien am Anfang nahe zu unmöglich, doch durch die Nutzung von dem `any` Typ ist es erstaunlich 'einfach'.

Die `interpret()` Methoden von dem `Interpreter` geben alle einen `any` Wert zurück. Dieser kann entweder invalid (keine Daten) oder valid (Daten) sein. Der `Interpreter` selber hat keine Ahnung, was sich in dem `any` Typ befindet – das ist nicht gut, aber der Preis, der gezahlt werden muss (da es keine Reflektion gibt). Dies provoziert Fehler, wenn die `Executable`s aufgerufen werden, und der erwartete Typ nicht übereinstimmt. Allerdings ist dieses Problem nicht 'neu' – die `Executable`s sind von dem Problem ebenso geplagt – was zur folge hat, dass die `any` Parameter für die `Executable`s geprüft werden, bevor mit diesen gearbeitet wird.

Die bis hier her beschriebene Architektur lässt sich sehr einfach in das Ziel (siehe Abbildung 19) konvertieren. Der `Interpreter` hat als Rückgabewert `any`, so wie `Executable`. Als Parameter kann der `Interpreter` eine Liste von `any` Werten – die einem `string` zugewiesen sind – annehmen, sowie zwei `string`s (Makro und Makro Name). `Executable` können nur eine Liste von `any` Werten, die einem `string` zugeweisen sind, annehmen. Zwei `any` Werte mehr in der Liste, als Makro und Makro Name löst das Problem der verschiedenen Anzahl von Parametern. Also muss ein `Interpreter` nur von einem Wrapper, der das `Executable` Interface implementiert, umschlossen werden und das anfängliche Ziel ist erreicht.

Ist der letzte Absatz zu abstrakt?

4 Exemplarische Realisierung

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)

```
def fun(foo){
  while(!foo){
    foo = foo + 1;
  }
  return foo;
}

def main() {
  var foo = 1;

  return fun(foo:foo);
}
```

Code 2: Interpreter Beziehungen.

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

```
def
fun
(
foo
)
{
while
(
!
foo
)
{
foo
=
foo
+
1
;
}
return
```

Code 3: Tokenized Makro

4.2 Parser

Lorem ipsum dolor sit amet ...

```
@Scope {
  l:0 c:0 t:
  @Define {
    l:1 c:1 t:def
    @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 {
```

Code 4: Geparsed Makro

4.3 Interpreter

Lorem ipsum dolor sit amet ...

Lorem ipsum dolor sit amet ...

5 Evaluation

6 Zusammenfassung und Ausblick

6.1 Ausblick

1. Ausnutzung des ASTs
   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 werde, 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 möglichen einen Stack von Fehlermeldungen zu produzieren. 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 geparst werden.

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.

   Dies Hätte zur Folge das der Nutzer mehr Fehler auf einmal beseitigen kann.

7 Literatur

[1] *cppreference.com*. 2015. URL: http://en.cppreference.com/w/cpp (besucht am 29. 12. 2015).

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

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

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

[5] 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).

[6] Roman R Redziejowski. „Parsing expression grammar as a primitive recursive-descent parser with backtracking“. In: *Fundamenta Informaticae* 79.3-4 (2007), S. 513–524.

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

[8] *The C++ Standards Committee*. 2016. URL: http://www.open-std.org/JTC1/SC22/WG21/ (besucht am 17. 01. 2016).

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

8 Anhänge