Liste der noch zu erledigenden Punkte

Hochschule Bremen

Bachelorarbeit

Thesis

Konzeption und Implementierung einer Makrosprache in C++

Autor:
Roland Jäger
360 956

27. März 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 27. März 2016 _____

Roland Jäger

Danksagung

Danke an Mutter, Vater und alle Profs

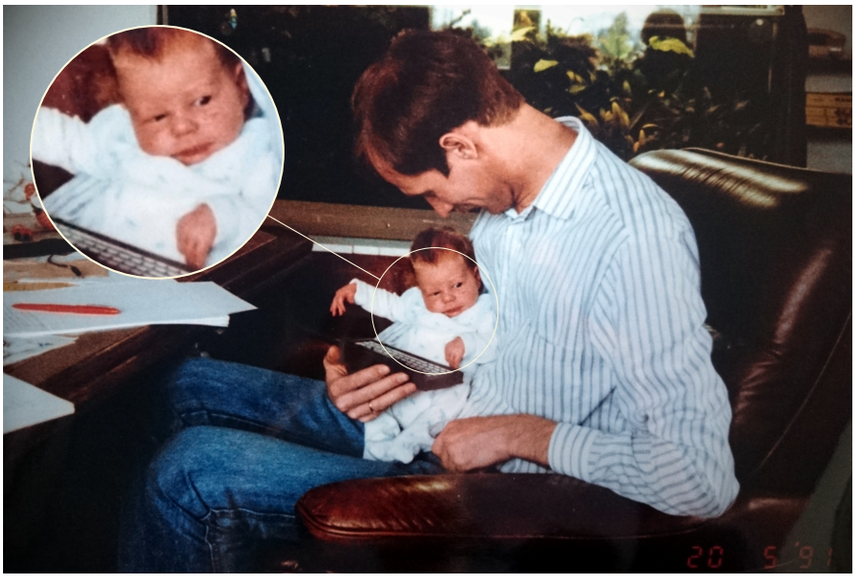

Abbildung 1: *“Das soll meine Zukunft sein?”*

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.

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.

- 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.

[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

Überschriften ohne Text sind böse, aber was kann ich hier schreiben?

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*[3, 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 integriert, welcher die Stücken des Strings in ein Format bringt, welches der *Interpreter*[3, S.274] verstehen kann. Das Format ist meist ein *abstrakter Syntaxbaum (AST)*[2], 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 ist alles ziemlich komprimiert – muss das noch ausführlicher durchgekaut werden, oder ist das 'Allgemeinwissen'?

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 durch komplett neue Funktionen zu erweitern, was die Vorteile von z.B Python größtenteils beseitigt.

2.2 Anforderungen an die angestrebte Lösung

Die Herausforderung, ein Makrosystem/-sprache in C++ zu implementieren fängt dann an, wenn man von den Makros will, dass diese nicht nur hintereinander abgearbeitet werden, ohne dass sie wissen, dass andere Makros vor ihnen bzw. nach ihnen ausgeführt werden – wie in Abbildung 2 zu sehen ist.

[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.

Abbildung 2: Sequenzielles Abarbeiten von Prozessschritten.

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

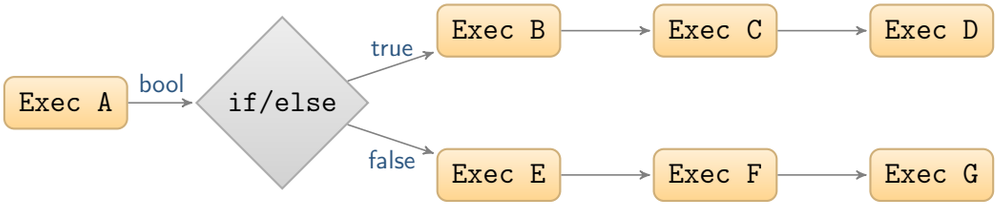

Abbildung 3: Logische Ausdrücke um bedingte Anweisungen zuzulassen.

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[7, 2] unterstützt. Dies ist – da es die Zeit, die für die Bachelorarbeit bereitsteht, mehrfach sprengt – schon durch eine Implementation[4] des Command-Patterns von gelöst worden. Somit kommt in diesem Schritt 'nur' noch hinzu, dass es Schleifen geben kann, siehe Abbildung 4, und Rückgabewerte in Variablen speicherbar sind.

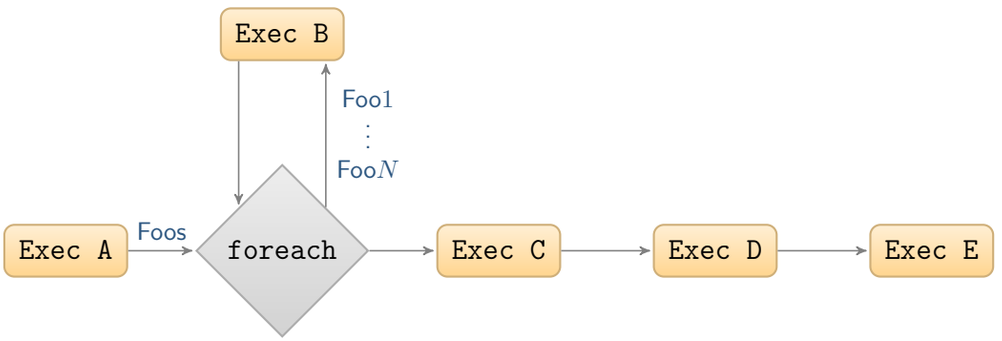

Abbildung 4: Schleife, die Anweisungen für ein Element aus der Liste aufrufen.

[4]Das Command-Pattern wurde mit Hilfe des `any`[5] sowie `optional`[6] Types 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

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

3 Konzeption

3.1 Syntax

Der Syntax ist an C, Python, JavaScript und Swift angelegt. C liefert den größten Anteil des Syntax, von Python wurde `def` übernommen, von JavaScript `var` und der named parameter Syntax von Swift `fun(foo:gun());`. `def` und `var` sorgen dafür, dass der Programmierer nach den ersten drei Zeichen weiß, was als passieren wird. Das die Makrosprache named parameter unterstützt liegt daran, dass das CommandPattern so implementiert wurde, dass Parameter alias Bezeichnungen benutzen können, um eine hohe Kompatibilität zwischen unabhängig entwickelten Modulen zu gewährleisten.

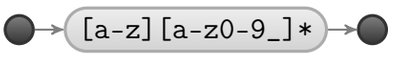

Abbildung 5: Bezeichner

Bezeichner müssen dem Regex aus Abbildung 5 entsprechen. Das heißt, dass Bezeichner nur aus keinen 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 entsprechen verboten.

Liste von keywords.

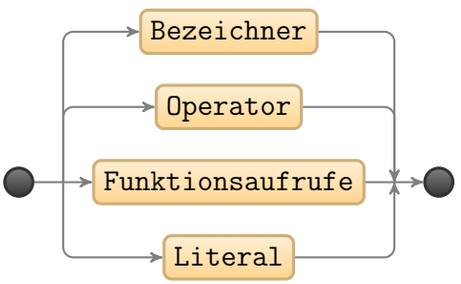

Abbildung 6: Werterzeuger

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

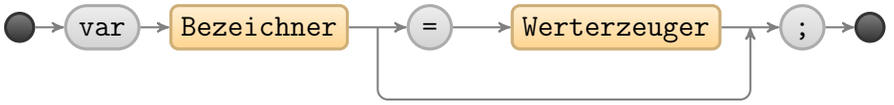

Abbildung 7: Syntax der Variablendeklaration.

Variablen können, wie in Abbildung 7 zu sehen ist, definiert werden: `var foo;`. Um nach der Definition einen Wert der Variablen zu zuweisen, ist es erlaubt unter anderem dies zu tun: `var foo = fun();` oder `var foo = true == false;`.

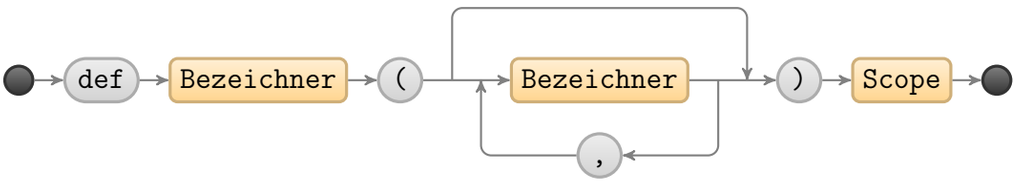

Abbildung 8: Syntax der Funktionsdeklaration.

Funktionen können über `def fun(){...}` definiert werden, was Abbildung 8 zeigt. Um eine parametrisierte Funktion zu definieren, gibt man die Parameternamen, Komma getrennt, nach dem Funktionsnamen an: `def fun(foo, bar){...}`. Der Einstiegspunkt eines jeden Makros ist eine `def main(){...}` Funktion. Der Syntax ist der selbe zu normalen Funktionen, und erlaubt es daher auch Parameter übergeben zu bekommen. Deswegen können die Makros aus anderen Makros, oder aus der C++ Ebene über einen äquivalenten Syntax aufgerufen werden.

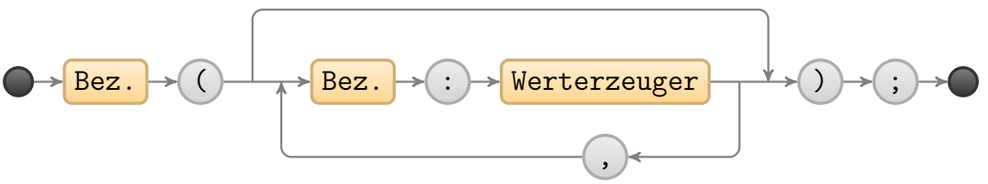

Abbildung 9: Syntax von Funktionsaufrufen.

Abbildung 9 zeigt den 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. Das zuweisen 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, im Gegensatz zu JavaScript, auch der Wert aus dem aufrufendem Scope.

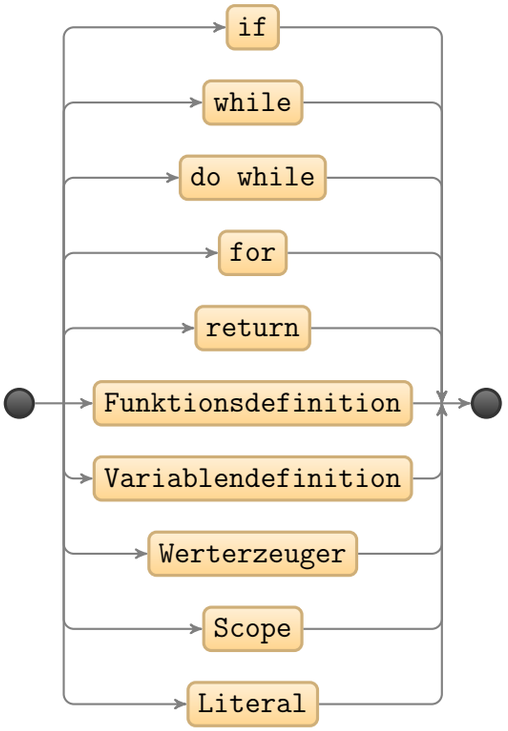

Abbildung 10: Syntax allgemeiner Strukturen.

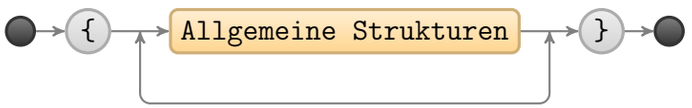

Abbildung 11: Syntax vom Scope.

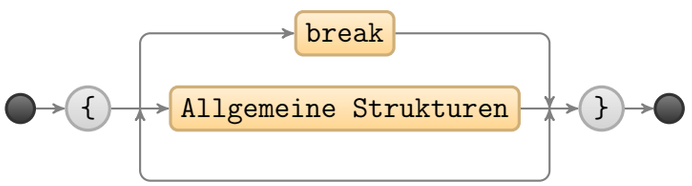

Abbildung 12: Syntax vom LoopScope.

Abbildungen 10, 11 und 12 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. Scopes verhalten sich wie C Scopes, das heißt, dass der Syntax `{var foo; {var foo;}}` richtig ist, und das erste `foo` von dem zweiten verdeckt wird.

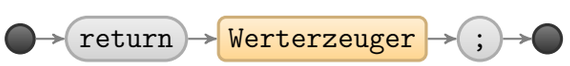

Abbildung 13: Syntax von return.

In Abbildung 13 ist der Syntax von `return` zu sehen.

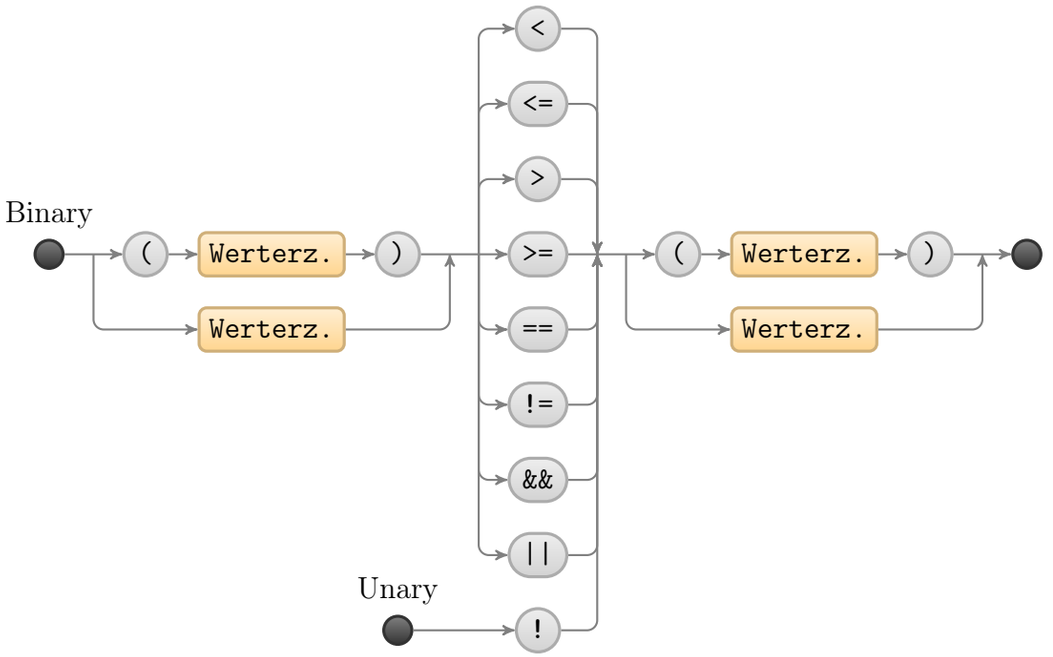

Abbildung 14: Syntax von Operatoren.

Der Operator Syntax aus Abbildung 14 sollte, ähnlich wie der `return` Syntax, nicht all zu 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.

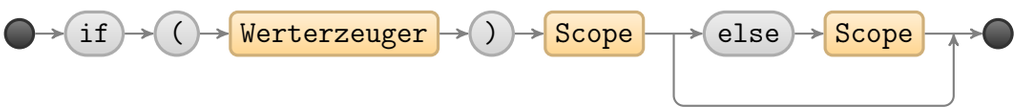

Abbildung 15: Syntax von if.

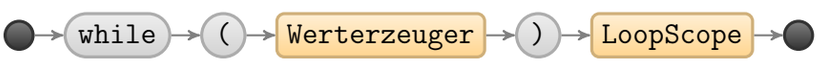

Abbildung 16: Syntax von while.

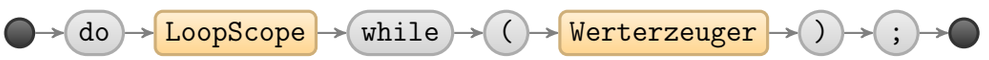

Abbildung 17: Syntax von do-while.

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

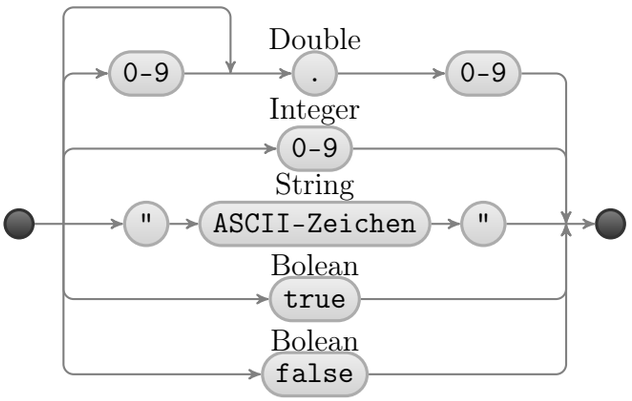

Abbildung 18: Syntax von Literals.

Literals sind entweder Doubles, Integer, Strings oder Boolean Werte, wie Abbildung 18 zeigt.

- For Schleife
- For-each Schleife

3.2 Level 1 – Grundarchitektur

3.3 Level 2 – Logik / primitive Rückgabewerte

3.4 Level 3 – Komplexe Rückgabewerte

4 Exemplarische Realisierung

4.1 Tokenizer

4.2 Abstrakter Syntaxbaum

4.3 Parser

4.4 Interpreter

4.5 Makro

5 Evaluation

6 Zusammenfassung und Ausblick

6.1 Ausblick

- Ausnutzung des ASTs
  Sofern alle Funktionen die ausgeführt werden sollen als thread-safe gekennzeichnet sind, kann das gesamte Macro parallel ausgeführt werden.
- Debugger / Stepping
  Es könnte ein Interface angeboten werde, mit dem man durch die Ausführung eines Macros Schritt für Schritt gehen kann.
- C++17 std::string_view
  for better parsing performance and less memory consumption.

7 Literatur

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

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

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

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

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

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

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

8 Anhänge