Liste der noch zu erledigenden Punkte

Hochschule Bremen

Bachelorarbeit

Thesis

Konzeption und Implementierung einer Makrosprache in C++

Autor:
Roland Jäger
360 956

22. 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 22. 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 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, 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 wird und nicht durch eine weitere Ebene übersetzt werden muss (C ist eine native Programmiersprache). Somit ist ein Makrosystem eher mit einer Sprache zu vergleichen, die sich einer *virtuellen Maschine* (VM) bedient – wie Java – als mit C.

Bei Java gibt die VM den Befehlssatz vor und muss bei der Ausführung des Programms diese Befehle, in die entsprechenden Hardwarebefehle, übersetzen (write once, run anywhere). Ähnlich verhält es sich mit dem Makrosystem, das Makro nutzt die vorhandenen Befehle der Applikation, um einen Arbeitsvorgang zu automatisieren. Da die Befehle jedoch nur als Zeichenketten vorliegen, müssen diese zu den richtigen Funktionen übersetzt werden. Der Befehlssatz dieses Makrosystems kommt aus dem vorhandenen System, da dort das Command-Pattern [8, 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 [6, S.46] integriert, um die Stücke der Zeichenkette (String) in ein Format zu überführen, die vom Interpreter unterstützt werden. 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 kompilierten Programmen, zu vereinen.

2.1.2 Parser

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

- Links nach rechts, links Auflösung (LL) [6, 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) [6, 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. [10]
- Rekursiv Absteigend (recursive descent)
  Recursive descent Parser arbeiten wie LR Parser 'rückwärts' und gelangen am Ende auch am Anfang des Baumes an. Im Gegensatz zu LL und LR Parsern arbeiten sie nicht mit Zustandstabellen sondern mit Rekursion. Für jedes Konstrukt, welches geparst werden soll, gibt es eine Methode. In dieser werden all die Methoden aufgerufen, die ein Element produzieren, welches sich in dem der äußeren Methode befinden darf.

2.1.3 Das vorhandene System

Das vorhandene Softwaresystem ist eine C++ Anwendung, die mit Hilfe von Modulen ihre Funktionalität erweitern kann. Ein Teil des zentralen Herzstückes ist eine Implementierung des Command-Patterns, siehe Abbildung 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. `Client`s sind über den `CommandProvider` in der Lage einen `Invoker` zu bekommen, der ein konkretes `Command` wrapped. Der `Invoker` bekommt bei der Erzeugung über den Konstruktor den momentanen `HistoryStack`. Wenn der `Invoker` durch einen Aufruf aus einem `Command` (auf den `CommandProvider`) entstanden ist, ist es der `HistoryStack` des `Command`s. Ansonsten ist es der `HistoryStack` des `CommandProvider`s. Durch den `HistoryStack` ist es möglich die Veränderungen von ausgeführten `Command`s rückgängig zu machen oder wiederherzustellen. Dies wird zum Beispiel von dem `Invoker` genutzt, wenn das aufgerufene `Command` einen Fehler verursacht. An dem `CommandProvider` können sich alle `Command`s registrieren, die in anderen Teilen der Anwendung genutzt werden können.

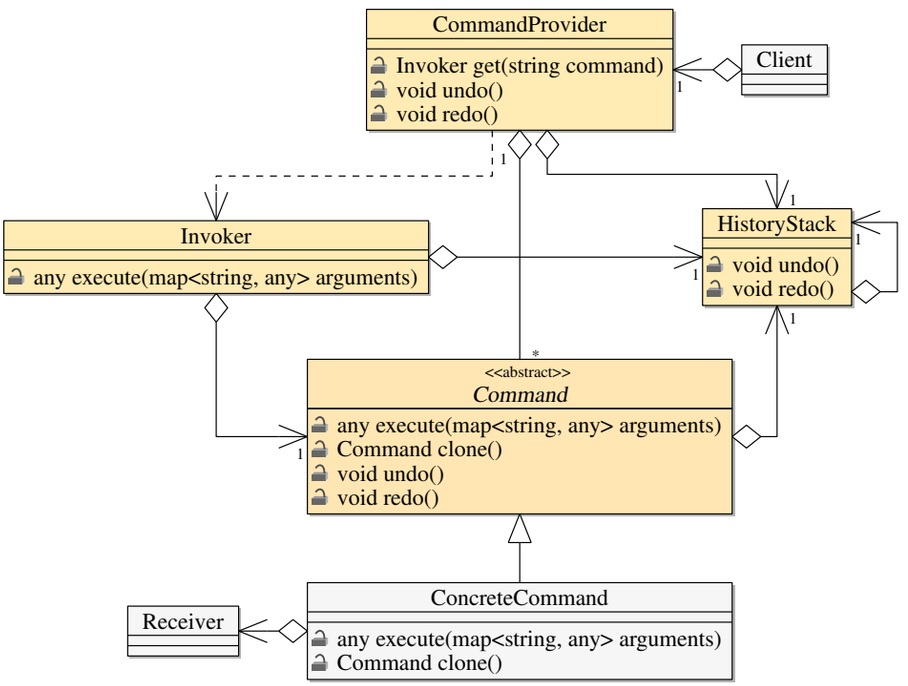

Abbildung 2: Abstrakte Command-PatterImplementation

Da C++ keine Reflexion [17, 7] unterstützt, ist die Signatur, der `any execute(map<string, any> arguments)` Methode, ein wichtiger Punkt der Architektur. Durch den `any` [2] Typ, in der Implementation des Command-Patterns, wurde die fehlende Reflektion umgangen[3]. `any` ist in der Lage, jeglichen Typ aufzunehmen und typsicher[4] 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 `ConcreteCommand`s 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 `Command`s nicht nur hintereinander abgearbeitet werden. Also ohne dass sie wissen, dass andere `Command`s vor bzw. nach ihnen ausgeführt werden – dieser Ablauf ist in Abbildung 3 zu sehen.

Abbildung 3: Sequenzielles Abarbeiten von Prozessschritten

Abbildung 4 zeigt den ersten Schritt zu einer nützlichen Implementierung – Logik. Hierbei bietet man an, dass der Makro-Entwickler durch Rückgabewerte aus `Command`s entscheiden kann, welche weiteren `Command`s er ausführen möchte.

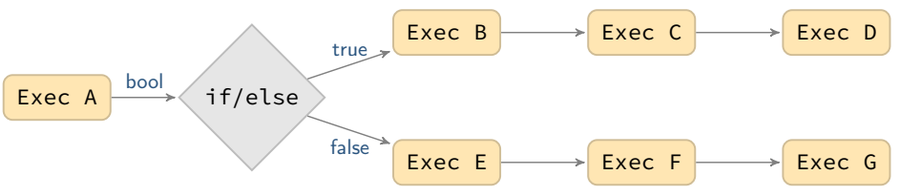

Abbildung 4: Logische Ausdrücke um bedingte Anweisungen zuzulassen

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.

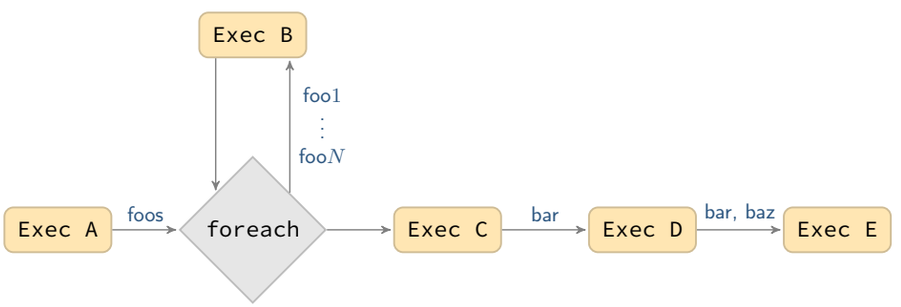

Abbildung 5: Schleife, die Anweisungen für ein Element aus der Liste aufrufen

Letztendlich kann man sagen, dass ein solches Makrosystem/-sprache eine Programmiersprache mit Interpreter [8, S.274] sein sollte, deren Laufzeitumgebung eine anderes Softwaresystem ist.

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

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

Listing 1: Clang Fehlermeldung

Ref Section in implementation / architekture

3 Konzeption

Dieses Kapitel beschäftigt sich mit der theoretischen Problemlösung.

Durchgängiges Beispiel?!?!!?

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 [3], Python [12], JavaScript [5] und Swift [15] 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. `*` bedeutet null, `+` bedeutet ein, oder mehr vom vorausgehenden Zeichen. `?` bedeutet null oder ein Mal, in Verbindung mit `*?` wird beim ersten validen Regex abgebrochen. `[]` ist eine Sammlung von Zeichen, bei dem alle Zeichen einzeln verglichen werden – `[^]` enthält alle Zeichen, die nicht genannt sind. In einer Sammlung kann `-` genutzt werden, um eine Reihe von Zeichen zu bestimmen. `.` kann jedes Zeichen sein, `\s` sind Whitespaces und `\d` sind Zahlen. `|` bedeutet oder und letztlich escaped `\` das folgende Zeichen.

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.

[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.

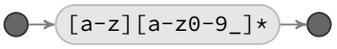

Abbildung 6: Bezeichner

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

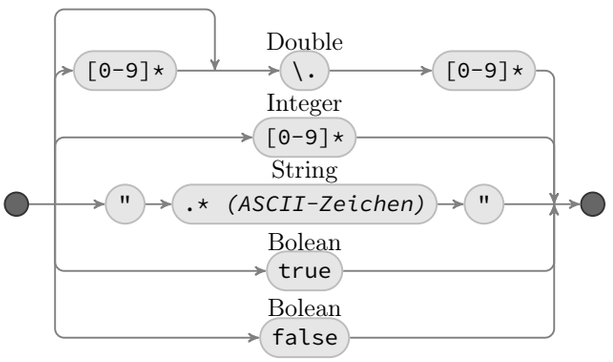

Abbildung 7: Syntax von Literals

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

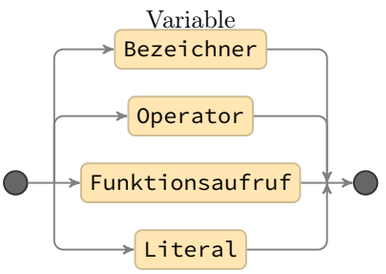

Abbildung 8: Werterzeuger

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.

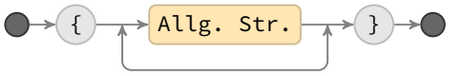

Abbildung 10: Syntax vom Scope

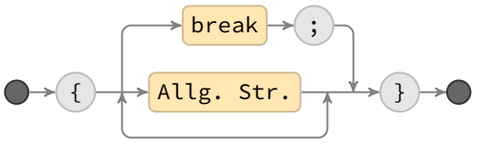

Abbildung 11: Syntax vom Loop Scope

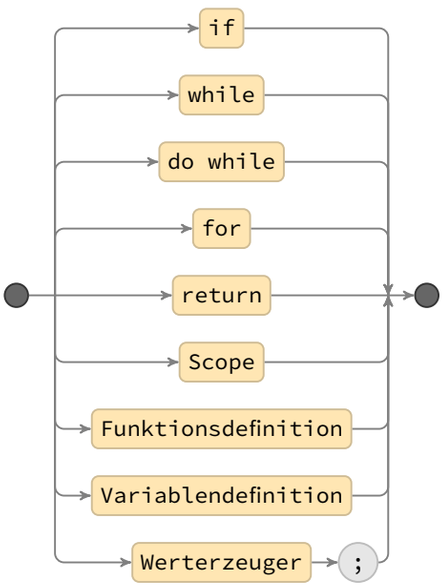

Abbildung 12: Syntax allgemeiner Strukturen

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.

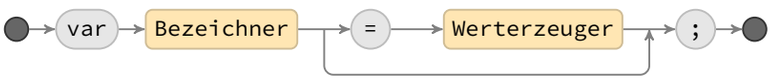

Abbildung 13: Syntax der Variablendeklaration

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.

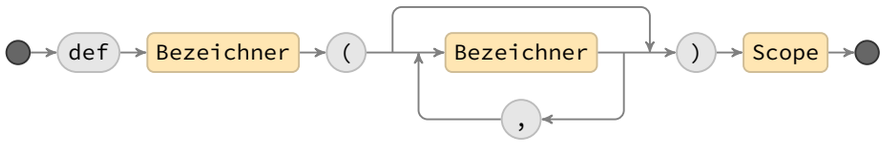

Abbildung 14: Syntax der Funktionsdeklaration

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`**, **`do`**-/**`while`** und **`for`** aus den Abbildungen 15, 16, 17 und 18 sollten wie erwartet aussehen.

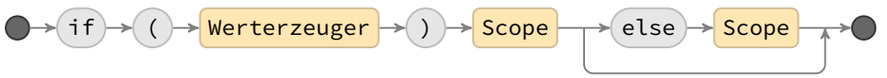

Abbildung 15: Syntax von if

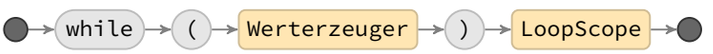

Abbildung 16: Syntax von while

Abbildung 17: Syntax von do-while

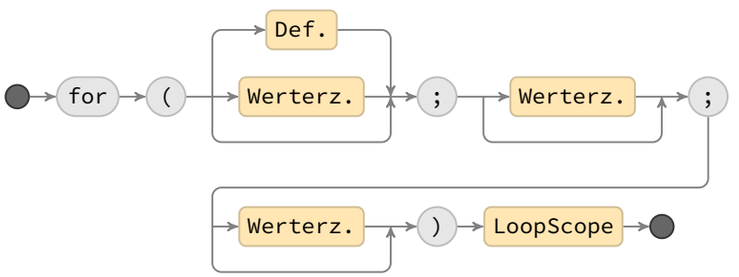

Abbildung 18: Syntax von for

3.1.4 Befehle

Abbildung 19 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 er Klammer zu haben.

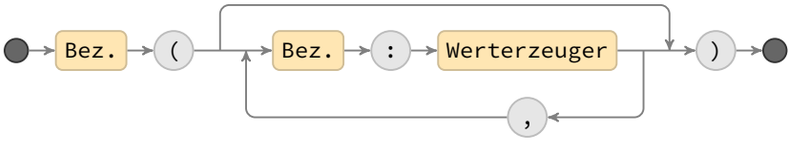

Abbildung 19: Syntax von Funktionsaufrufen

Die Operator Syntax aus Abbildung 20 folgt, wie die **`return`** Syntax, den Vorbildern dieser Syntax (C, Python, Javascript, Swift).

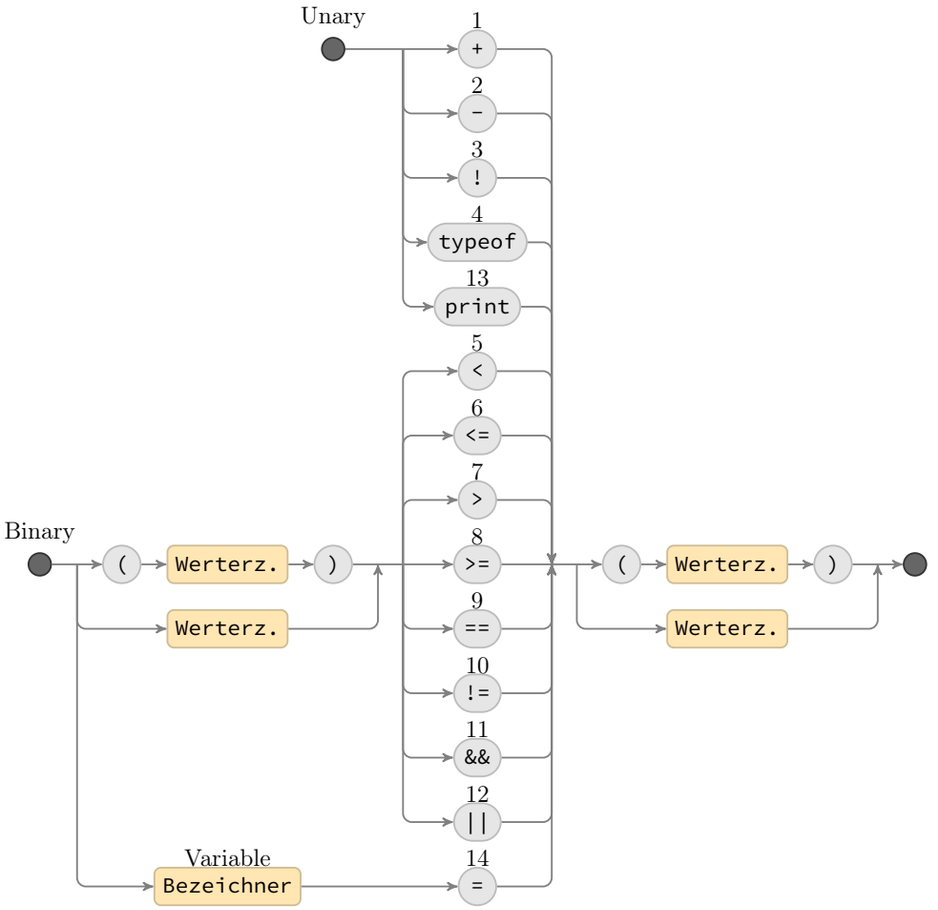

Abbildung 20: Syntax von Operatoren

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 **`print`** (der vor `=` kommt) 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`→`"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.

Durchgängiges Beispiel?!?!!?

Abbildung 21 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.

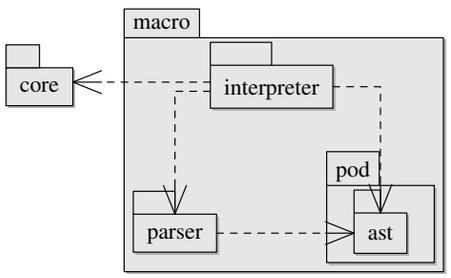

Abbildung 21: Abhängigkeiten von dem Makro Modul

3.2.1 Token und Parser Paket

Der `Parser` aus Abbildung 22 bedient sich des `Tokenizer`, um eine `TokenList` von `Token`s 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.

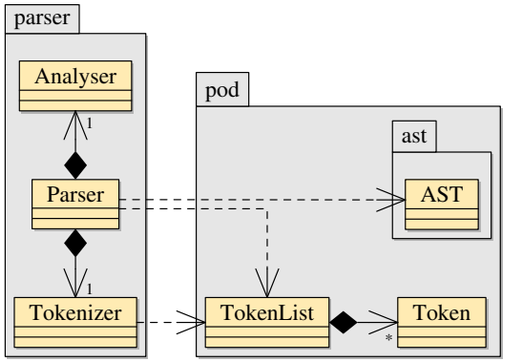

Abbildung 22: Parser Paket UML

Tokenizer Der `Tokenizer` wandelt den String, der das Makro beschreibt, in eine Reihenfolge von `Token`s um. Tokens sind alle Zeichen, die von whitespace (\s*) getrennt sind, die nicht den Anforderungen als Bezeichner genügen ([^a-zA-Z0-9_], siehe Abbildung 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.

Token `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 (zB. `if`).

Parser 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[14]. 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[9] und GCC[11] 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, escapete Symbole in einem String müssen umgewandelt werden (z.B.: `"t\tt"`→`"t  t"`).

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

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, das diese Reihenfolge eindeutig und korrekt ist.

Variablendeklarationen mit anschließender Wertzuweisung müssen in zwei Schritte aufgeteilt werden (erst deklarieren und dann der Variablen den Wert zuweisen[7]).

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

Der `Analyser` wird den AST ablaufen und vor und nach jedem AST Element, ein Signal abschicken. Signals sind mit dem Visitor-Pattern [8, S.366] sehr eng verwand. Ein Visitor wird bei dem entsprechenden Signal registriert und durch das absenden des Signals aufgerufen – es können pro Signal mehrere Visitoren verbunden werden. Wenn der `Analyser` dann zum Beispiel einen Funktionsaufruf, auf die `main()` Methode findet, kann er dies als Fehler melden. Im Gegensatz zu dem Visitor-Pattern werden bei Signalen alle Visitoren gesammelt und in einem Durchlauf abgearbeitet. Der Vorteil hierbei ist, das der AST nur einmal durchlaufen werden muss, was schneller ist und keine Nachteile mit sich bringt.

3.2.2 Abstrakter Syntaxbaum Paket

Wie Abbildung 23 zeigt[8], 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 24), 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 einigele Klassen hinter allgemeinen Begriffen – wie `Loop` – versteckt, um etwas Übersicht zu bewahren.

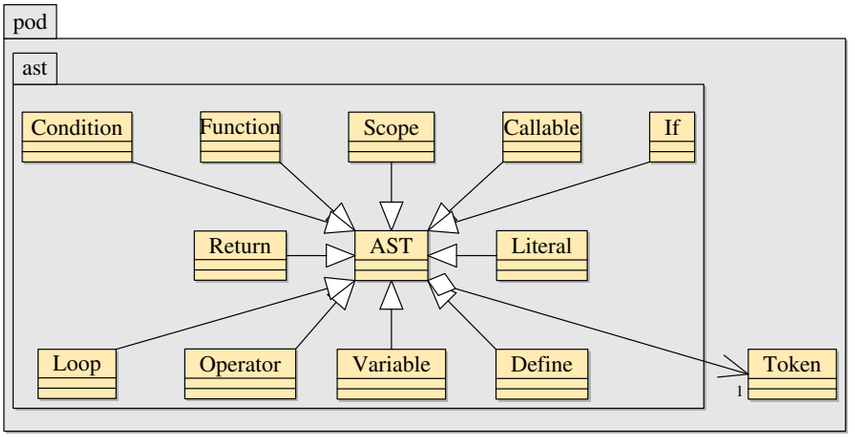

Abbildung 23: Stammbaum der AST Klassen

Abbildung 24 zeigt die `Scope` Klasse, diese Klasse kann beliebig viele andere `AST` Instanzen aufnehmen.

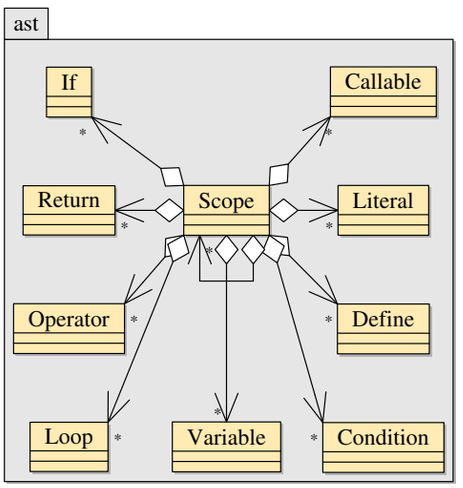

Abbildung 24: Verbindungen vom Scope

In Abbildung 25 sind die restlichen Abhängigkeiten zwischen den `AST` Klassen zu sehen.

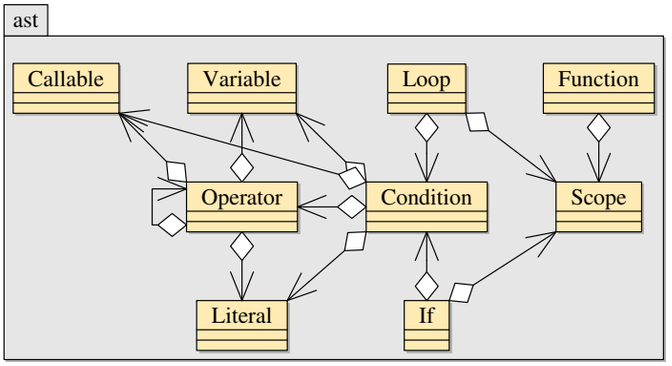

Abbildung 25: Abhängigkeiten der AST Klassen

3.2.3 Interpreter Paket

Der `Interpreter` nutzt den `parser::Parser`, um einen `ast::Scope` zu erzeugen, wie in Abbildung 26 zu sehen ist.

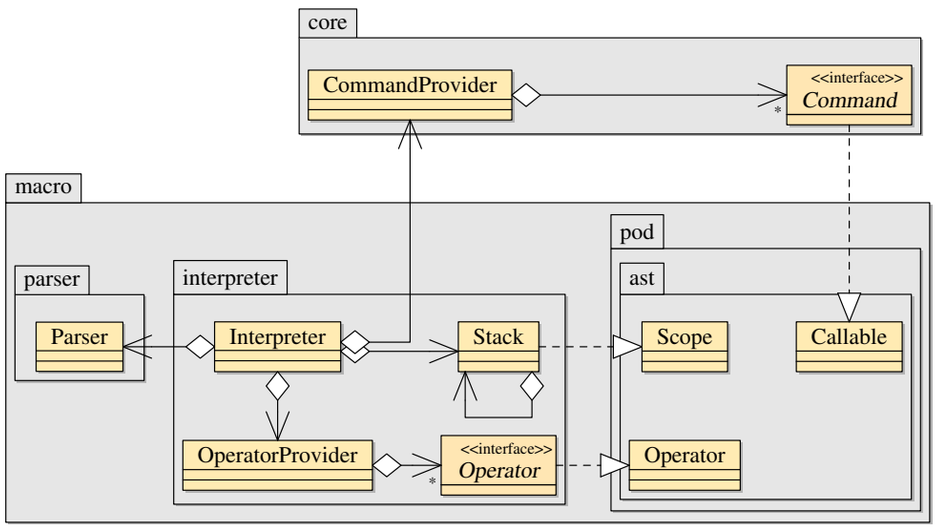

Abbildung 26: Interpreter Beziehungen

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

Stack Während der Ausführung des Makros, durch den `Interpreter`, übernimmt der `Stack` die Verwaltung der definierten Funktionen und Variablen sowie deren Werte. Dadurch repräsentiert der `Stack` die `ast::Scope`s. 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 Verketteteliste aufgebaut, bei der man nur den Kopf entfernen muss, um ein `ast::Scope` zu beenden, siehe Abbildung 27.In Unterunterkapitel 3.3.5 wird die Architektur weiter ausgeführt.

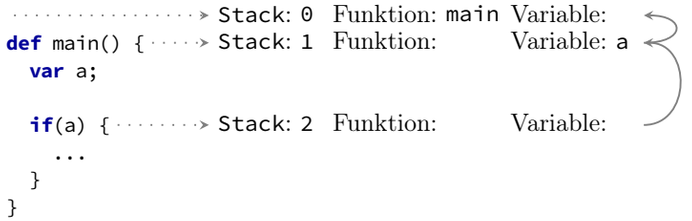

Abbildung 27: Stack Beispiel

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::Command`s die `ast::Callable` Funktionen. Funktionen, die in der Makrosprache definiert sind, realisieren ebenso die `ast::Callable`.

Interpreter Der Interpreter selber läuft den AST ab, erweitert den `Stack` bei `ast::Scope`s, fügt dem `Stack` Variablen und Funktionen bei deren Definition (durch `ast::Define`) zu und poped den `Stack` wenn er fertig ist, ein `ast::Scope` zu interpretieren. `ast::Condition`s werden mit Hilfe das `OperatorProvider`s 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 `Parser`s, `OperatorProvider`s, `Stack` und des `Interpreter`s genauer betrachtet.

Durchgängiges Beispiel?!?!!?

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

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

C++⟷C und C⟷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. 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 [13]. 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 [18, 9 Classes §10]. Da die `Token`s `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 `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 Unterkapitel 6.1: Punkt 3), 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 Analyser Architektur

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

Der erste Teil beschäftigt sich mit dem ablaufen des AST und dem absenden der Signale (der Ausführung der Visitoren). Dieser Teil ist recht einfach, da es keiner besonderen Logik bedarf. Vor dem Aufruf eines Signals, von einem `ast::Scope`, fügt der `Analyser` das Token, des `ast::Scope`s, als Referenz zu einer Liste hinzu. Diese Liste kann dann genutzt werden um Fehlermeldungen, mit der gleichen Informationsqualität des `Parser`s zu erzeugen[10].

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

Unter anderem findet der `Analyser` Fehler wie die doppelte Funktionsdeklaration. `def fun() {...}` und `def fun(foo) {...}` stehen nicht in Konflikt, da sie andere Parameter haben. Da die Reihenfolge der Parameter nicht Bestandteil der Signatur ist, ist `def fun(a, b) {...}` und `def fun(b, a) {...}` – im selben `ast::Scope` – nicht zulässig.

[10] Der `Analyser` setzt Unterkapitel 6.1: Punkt 3 um.

3.3.4 OperatorProvider Architektur

Der `OperatorProvider` ist eine Klasse, deren Aufgabe es ist, Datentypen Funktionen zuzuordnen und zur Verfügung zu stellen. Da die `Operatoren` zustandslos sind[11], 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[12] 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 werde. In dem Kontext `i && a == b` wird für `i` der **`bool`** Operator aufgerufen aber nicht für `a` und `b`;

3.3.5 Stack Architektur

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

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 27 und Abbildung 26. 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 4) 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.

3.3.6 Interpreter Architektur

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

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

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 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 `CommandProvider` nach einer Funktion gefragt.
  Das heißt, dass die nativen Funktionen von Funktionsdefinitionen in dem Makro überschrieben und auch unerreichbar werden können.
- Wenn eine Funkion interpretiert werden muss und diese von dem `CommandProvider` kommt, werden die Parameter kopiert.
  Es ist durch die C++ Ebene vorgegeben und wird normalerweise durch Pointer beschleunigt. In der C++ Ebene ist es auch möglich die move-semantics aus C++11 anstelle von Pointern anzuwenden, das ist allerdings nicht aus der Makro Ebene möglich.
- Wenn eine Variable aus einem `ast::Scope` returned wird, also ein Funktionsrückgabewert ist, kann dies ohne eine Kopie passieren.
  Dies ist dann möglich, wenn die Variable, die returned wird in dem `Stack` angelegt wurde und keine Referenz ist.

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

1. Interpretiert er den `ast::Operator` soweit, dass er einen/zwei Werterzeuger und einen Operatortyp (zB. `==`) hat.
2. Erzeugt der `Interpreter` den/die Wert/e des/der Werterzeuger/s.
   Im Falle, dass es sich um den Operatorentyp `&&` oder `||` handelt, kann der Interpreter auch schon durch die Kurzschlusssemantik früher aufhören.
3. Nutzt er den `OperatorProvider` um die beiden Werte mit einem registrierten Opteratoren zu vergleichen.

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

3.3.7 Komplexe Rückgabewerte

Die komplexen Rückgabewerte, aus der Makro Ebene in die C++ Ebene sind durch den `any` Typ kein Problem, 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::ConcreteCommand`s 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 `string`s (Makro und Makro Name). `core::Command`s 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::Command`s aufrufen können, sind sie auch in der Lage andere Makros aufzurufen (siehe Unterkapitel 6.1: Punkt 5). 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 `Command`s. Allerdings ist ein wertvoller Vorteil dieser Kombinierbarkeit, dass Makros durch C++ Test-Frameworks ausgiebig getestet werden können.

4 Exemplarische Realisierung

Um schnell Resultate zu sehen, wird die Implementierung nicht nur inkrementell, sondern auch test-driven [1] vorgenommen. Inkrementell bedeutet, dass nicht komplett vollständige Teile der Software genutzt werden, um Abhängige Elemente zu entwickeln. Die Inkrementelle Entwicklung von Software kommt aus der Agile Softwareentwicklung [4]. Dabei werden schnell Resultate gesehen und zB. fallen Architektur Fehler schneller auf – wodurch grundlegende Probleme frühzeitig beseitigen werden können.

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

In den folgenden Unterkapiteln wird der Ablauf für das Makro aus Listing 2 durchgegangen. Die Reihenfolge ist die, die in Abbildung 1 schon beschrieben wurde.

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

Listing 2: Beispiel für die exemplarische Realisierung

4.1 Tokenizer

Der `Tokenizer` verwandelt den Code aus Listing 2 zu den `Token`s, die in Listing 3 zu sehen sind (abgesehen von der gesamten Code Zeile, die als Pointer gespeichert wird).

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

```
1 l:1  c:1  t:def
2 l:1  c:5  t:fun
3 l:1  c:8  t:(
4 l:1  c:9  t:foo
5 l:1  c:12 t:)
6 l:1  c:14 t:{
7 l:2  c:3  t:do
```

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

Dezimalzahl Wenn das Zeichen eine Zahl ist, probiert der `Tokenizer` den Regex \d*\.\d+ auf das aktuelle und die folgenden Zeichen anzuwenden. Integer werden als normale Token geparst. Siehe Unterkapitel 6.1: Punkt 6 für Verbesserungen.

String Wenn das momentane Zeichen ein " ist, werden alle Zeichen gelesen, bis ein weiteres " gefunden wird. Um das escapen von " zu unterstützen, wird die Anzahl, der aufeinander folgenden \\ gezählt. Wenn es sich um eine ungerade Zahl handelt, ist das " escaped und wird nicht als String Ende angesehen. Das Ergebnis dieses Verfahrens ist in Listing 3 Zeile 34 zu sehen. Ohne dieses Verfahren würde es drei `Token`s geben, und der Whitespace wäre "verloren" gegangen.

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

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

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

Listing 3: Tokenized Makro / TokenList

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

4.2 Parser

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

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

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

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

Listing 4: Root Scope des Beispieles

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

Aktuelle TokenList

Ich hoffe das ich hier die Regel 'brechen' darf – Ich behaupte das dies 'eine' Abbildung ist und somit Listing 4 für 'beide' reicht...

4.2.1 Definition

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

Um `ast::Funktion` zu parsen, erwartet die Methode, dass das momentane `Token` def entspricht – var um eine `ast::Variable` zu parsen. Wenn keiner der beiden Fälle eintritt, wird `false` zurückgegeben – alle Methoden zum Parsen verhalten sich so. Das `false` anstelle von einem `ast::Define` Objekt zurückgeben werden kann, wird durch den `optional<T>` Typ erreicht [2].

Funktion Wenn def geparst wurde, wird die `TokenList` an die Funktion für `ast::Funtion` übergeben. Diese erwartet, als aktuelles `Token` einen Bezeichner (siehe Abbildung 6) und, dass das nächste eine offene Klammer \(ist. Nach der Klammer kann eine beliebige

Anzahl von Komma getrennten Bezeichnern angegeben werden. Diese werden dem `ast::Function` Objekt als Array von `ast::Variable` Objekten übergeben. Nach den Parametern wird eine geschlossene Klammer \) erwartet. Zuletzt wird die Funktion zum parsen von `ast::Scope` aufgerufen und das Ergebnis in dem `ast::Function` Objekt gesetzt.

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

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

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

Listing 5: Funktionsdefinition des Beispiels

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

Aktuelle TokenList

Variable Wenn das geparste `Token` var war, wird die `TokenList` an die Funktion für `ast::Variable` übergeben. Diese erwartet, als aktuelles `Token`, nur einen Bezeichner. Es ist also wichtig, dass zuerst probiert wird eine `ast::Funktion` zu parsen und erst im Anschluss eine `ast::Variable`.

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

Wie bei der Methode zum parsen von `ast::Function`, befindet sich die Logik dieser Methode in einem **`try-catch`** Block.

4.2.2 Scope

Die `ast::Scope` Klasse ist eine der meist verwendeten Klassen aus dem `ast` Paket. Die `ast::Scope` Klasse macht sich `variant<T1, T2, ..., Tn>` [16] zu Nutze – Variant ist

eine typsichere Union. Der Varianttyp des `ast::Scope`s hat, als Template Parameter, alle `ast` Typen, die in einem Loop Scope auftreten können (Abbildung 11).

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

Ähnlich wie in dem Railroad Diagramm aus Abbildung 12, müssen einige `ast` Elemente mit einem ; gefolgt werden, da sie auch in einem Kontext eingesetzt werden können, in dem kein ; benötigt ist. Diese Token werden durch das Scope gelesen und finden sich nicht im AST wieder, da sie keinen weiteren Wert haben. Sollten sie fehlen, wodurch Statements nicht zuverlässig geparst werden können, wird eine Exception geschmissen.

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

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

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

Listing 6: Funktionsscope des Beispiels

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

Aktuelle TokenList

4.2.3 do-while

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

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

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

```
1 @DoWhile {
2   line: 2 column: 3 token: do
3   Contition:
```

```
7  l:2  c:3  t:do
8  l:2  c:6  t:{
9  l:3  c:5  t:foo
```

```
4     @UnaryOperator {
5       ...
6     }
7   Scope:
8     @Scope {
9       line: 2 column: 6 token: {
10      @BinaryOperator {
11        ...
12      }
13    }
14 }
```

Listing 7: do-while Schleife des Beispiels

```
10 l:3  c:9  t:=
11 l:3  c:11 t:foo
12 l:3  c:15 t:+
13 l:3  c:17 t:1.1
14 l:3  c:20 t:;
15 l:4  c:3  t:}
16 l:4  c:5  t:while
17 l:4  c:10 t:(
18 l:4  c:11 t:!
19 l:4  c:12 t:foo
20 l:4  c:15 t:)
```

Aktuelle TokenList

4.2.4 Operator und Condition

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

Wenn der `Parser` `ast::Operator`en parsed, passiert dies in zwei Schritten. Im ersten Schritt werden alle `ast` Elemente (Operatoren und Werterzeuger) erstellt, die der `Parser` aus den `Token`s erstellen kann. Diese sind nach dem parsen allerdings nur als eine Liste angeordnet, da der `Parser` nicht wissen kann, in welcher Reihenfolge er die `Token`s zusammenzusetzen hat. Die Liste wird dann, in dem zweiten Schritt, in einen Baum verwandelt. Das geschieht, in dem der Parser einen `ast::Operator` mit seinen jeweiligen Operanden verbindet. Dabei wird die Präzedenz bzw. Reihenfolge, die in Abbildung 20 zu sehen ist, angewendet bzw. erstellt.

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

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

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

Die `ast::Operator` Methode erwartet, dass – in dem Fall, dass es sich um einen binären Operatoren handelt – das zuletzt geparste Element ein Werterzeuger ist (Abbildung 8). Dieses wird als linkes Element des `ast::Operator` gesetzt. Als nächstes erwartet die Methode, dass das nächste `Token` auch ein Werterzeuger ist.

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

Die Datenstruktur nach dem ersten Schritt zeigt Listing 8.

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

Listing 8: Erster Schritt der Variablen Addition des Beispiels

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

Listing 9: Halber zweiter Schritt der Variablen Addition des Beispiels

In dem zweiten Schritt wird die Liste von links bis zu dem `+` durchgegangen. Beim zusammensetzen des `+` greift dieses auf `foo` und auf den Double zu. Anschließend geht wird die Liste von rechts nach links bis zu dem `=` durchgegangen. Der `=` Operator greift dann, auf `foo` und den `+` Operator zu, der das andere `foo` und den Double enthält, was in Listing 9 zu sehen ist.

Der generierte AST ist in Listing 10 zu sehen.

```
@BinaryOperator {
  line: 3 column: 9 token: =
  Left operand:
    @Variable {
      line: 3 column: 5 token: foo
    }
  Operation: assignment
  Right operand:
    @BinaryOperator {
```

```
l:3  c:5  t:foo
l:3  c:9  t:=
l:3  c:11 t:foo
l:3  c:15 t:+
l:3  c:17 t:1.1
l:3  c:20 t:;
l:4  c:3  t:}
l:4  c:5  t:while
l:4  c:10 t:(
```

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

Listing 10: Variablen Addition des Beispiels

```
l:4  c:11 t:!
l:4  c:12 t:foo
l:4  c:15 t:)
l:4  c:16 t:;
l:6  c:3  t:return
l:6  c:10 t:foo
l:6  c:13 t:;
l:7  c:1  t:}
l:9  c:1  t:def
l:9  c:5  t:main
l:9  c:9  t:(
l:9  c:10 t:)
```

Aktuelle TokenList

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

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

```
Contition:
  @UnaryOperator {
    line: 4 column: 11 token: !
    Operation: not
    Operand:
      @Variable {
        line: 4 column: 12 token: foo
      }
  }
```

Listing 11: do-while Condition des Beispiels

```
l:4  c:10 t:(
l:4  c:11 t:!
l:4  c:12 t:foo
l:4  c:15 t:)
l:4  c:16 t:;
l:6  c:3  t:return
l:6  c:10 t:foo
l:6  c:13 t:;
l:7  c:1  t:}
```

Aktuelle TokenList

4.2.5 Literals

Um Literals zu parsen, wendet der `Parser` für Integer \d+, Doubles \d*\.\d+ und Booleans true| false als Regex an, um zu erkennen, ob ein `Token` ein Literal beschreibt. Für einen String muss der `Parser` nur das erste Zeichen des Tokens auf Gleichheit mit " überprüfen, da der `Tokenizer` den String als ein `Token` produziert hat.

Um die nummerischen Tokens zu Werten zu konvertieren (lexen), nutzt der `Parser` die C++ Funktionen `std::stod(...)` und `std::stoi(...)`. Für die Booleanwerte reicht die Überprüfung, ob das `Token` ein Boolean darstellt. Und der String wird kopiert und anschließend werden alle escapeden Zeichen unescaped (`"t\tt"`→`"t  t"`[13]).

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

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

```
@Double {
  line: 3 column: 17 token: 1.1
  Data:
    1.1
}
```

Listing 12: Double Literal des Beispiels

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

Aktuelle TokenList

4.2.6 Return

Return ist leicht zu parsen, da es nur einen Werterzeuger, nach dem Keyword return erwartet.

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

Listing 13 zeigt Zeile 6 des Beispiels.

```
@Return {
  line: 6 column: 3 token: return
  @Variable {
    line: 6 column: 10 token: foo
  }
}
```

Listing 13: return Statement des Beispiels

```
l:6  c:3  t:return
l:6  c:10 t:foo
l:6  c:13 t:;
l:7  c:1  t:}
l:9  c:1  t:def
l:9  c:5  t:main
```

Aktuelle TokenList

4.2.7 Funktionsaufruf

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

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

Listing 14 zeigt das Ergebnis aus der Zeile 12 des Beispieles (**`return`** `fun(foo:foo);`).

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

```
l:12 c:3  t:return
l:12 c:10 t:fun
l:12 c:13 t:(
l:12 c:14 t:foo
l:12 c:17 t::
l:12 c:18 t:foo
l:12 c:21 t:)
```

```
8       }
9   }
10 }
```

Listing 14: Funktionsaufruf des Beispiels

```
43 l:12 c:22 t:;
44 l:13 c:1  t:}
45
```

Aktuelle TokenList

4.2.8 Analyser

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

Wie zuvor beschrieben (siehe Unterunterkapitel 3.3.3) nutzt der `Analyser` ein Signal/Visitor System[14], um bei jedem beliebigen Element Funktionen auszuführen. In dem Fall, dass überprüft werden soll, ob die `main()` Methode manuell aufgerufen wird, reicht es aus sich bei dem Signal anzumelden, welches `ast::Callable` signalisiert. In der Funktion, die das Signal ausführt, wird dann überprüft, ob das `Token` main gleicht. Wenn dem so ist, ist der AST nicht kompatible und eine Fehlermeldung wird in dem `Analyser` zu der Fehlerliste hinzugefügt. Wenn der AST durchlaufen ist, und somit alle Prüfungen ausgeführt wurden, wird die Fehlerliste an den `Parser` übergeben. Dieser schmeißt die Fehlerliste dann als Exception.

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

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

4.3 Interpreter

Lorem ipsum dolor sit amet ...

4.3.1 OperatorProvider

Lorem ipsum dolor sit amet ...

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

4.3.2 Stack

Lorem ipsum dolor sit amet ...

4.4 Fehlermeldungen

Wenn in Zeile 3 das ; vergessen wird, erhält man die Fehlermeldung aus Listing 15 – diese Fehlermeldung wird von dem `Parser` ausgegeben und führt dazu, dass das parsen abgebrochen wird.

```
1 Anonymous:1:5: In the 'fun' function defined here:
2 def fun(foo) {
3     ^
4 Anonymous:2:3: In the do-while defined here:
5   do {
6   ^
7 Anonymous:3:17: Expected a ';'
8     foo = foo + 1.1
9                 ^
```

Listing 15: Fehler bei unbekannter Variable

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

```
1 Anonymous:9:5: In the 'main' function defined here:
2 def main() {
3     ^
4 Anonymous:12:3: At return defined here:
5   return fun(foo:fo);
6   ^
7 Anonymous:12:18: Undefined variable 'fo'
8   return fun(foo:fo);
9                  ^
```

Listing 16: Fehler bei unbekannter Variable

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

2. C++17 std::string_view
   Um weniger Speicher zu verbrauchen und durch weniger Memory Allokationen schneller beim Tokenizen zu sein.

3. 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 – 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 geparst 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.

4. Verkettete Stack Liste in ein Array umwandeln
   Wenn sich herausstellt, dass die verkettete Liste von dem Stack zu langsam ist, kann der Interpreter ein Array nutzen um die 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

5. Einen Makro AST Buffer bereitstellen.
   Da das parsen und validieren der Makros ziemlich langsam ist, ist es ratsam einen Provider zu implementieren, der den Namen des Makros mit dem geparsten AST assoziiert. Der AST wird von dem `Interpreter` nicht verändert, weswegen das Makro nur geparst werden muss, wenn es sich verändert hat.

   Ein weiterer Vorteil wäre, dass das Makro nicht als String einem anderen Makro bekannt sein muss oder als `core::ConcreteCommand` implementiert sein muss, da der `Interpreter` im 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.

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

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

Literatur

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

[2] *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).

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

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

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

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

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

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

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

[10] 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).

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

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

[13] *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).

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

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

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

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

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

Abbildungsverzeichnis

Listingverzeichnis

Anhänge