

Konzeption und Implementierung einer Makrosprache in C++

Roland Jäger
360 956

Hochschule Bremen



11. Juni 2016

2016-06-11

Konzeption und Implementierung einer Makrosprache in C++

Konzeption und Implementierung einer
Makrosprache in C++

Roland Jäger
360 956
Hochschule Bremen

11. Juni 2016



1. Moin, ich denke ihr kennt alle meinen Namen und ...
2. Da wir nur wenig Zeit haben, muss ich euch in den nächsten 20 min 15.000 Zeilen Code erklären. Das sind 750 Zeilen pro Minute ...

1. Was?
2. Anforderungen
3. Konzeption
4. Realisierung
5. Demo

2016-06-11

Konzeption und Implementierung einer Makrosprache in C++

└ Themen

- Themen
1. Was?
 2. Anforderungen
 3. Konzeption
 4. Realisierung
 5. Demo

1. Das kann ich allerdings keinem zumuten, weshalb ich in den folgenden Folien meine Arbeit sehr abstrakt erklären werde. In der Demo werde ich auch nur ein wenig auf den Code eingehen. Wodurch euch C++ Code größten Teils erspart bleibt.
2. Wer im Anschluss Fragen zum Code hat – ich werde gerne Fragen beantworten (und wahrscheinlich länger über C++ reden, als euch recht ist).
3. Fragen zu der Präsentation und der Demo können vor und nach der Demo gestellt werden. Nachdem meine Professoren ihre Fragen gestellt haben, werde ich auch gerne Fragen von den Zuschauern beantworten, da ich möchte, dass jeder etwas von meiner Präsentation mitnehmen kann.

Was?

2016-06-11

Konzeption und Implementierung einer Makrosprache
in C++

└ Was?

Was?

1. Fangen wir mit einer einfachen Frage an: 'Was?'. ...

Was ist eine Makrosprache?

2016-06-11

Konzeption und Implementierung einer Makrosprache
in C++

└ Was?

└ Was ist eine Makrosprache?

1. Was ist eine Makrosprache?

Was ist eine Makrosprache?

2016-06-11

Konzeption und Implementierung einer Makrosprache in C++

└ Was?

└ Was ist eine Makrosprache?

• Eine Programmiersprache

- Eine Programmiersprache

1. Eine Makrosprache ist mit euch bekannten Programmiersprachen zu vergleichen.
2. Makros werden genutzt, um eintönige Arbeitsabläufe zu automatisieren.
3. Makros führen also Befehle aus, die der Nutzer hätte ausführen können. Somit ist eine Makrosprache eine der höchsten (abstraktesten) Programmiersprachen.
4. Zu vergleichen ist das am ehesten mit den Sprachen, die eine Virtuelle Maschine nutzen. Im Fall der Makrosprache ist die Anwendung die Virtuelle Maschine. Es sollte also kein Unterschied machen, ob ein Makro auf Windows geschrieben wurde und dann auf Linux oder XOS ausgeführt wird.

- Eine Programmiersprache
- Eine Möglichkeit statische Systeme zu dynamischen umzuwandeln

2016-06-11

Konzeption und Implementierung einer Makrosprache in C++

└ Was?

└ Was ist eine Makrosprache?

- Eine Programmiersprache
- Eine Möglichkeit statische Systeme zu dynamischen umzuwandeln

1. Makros sind dynamisch.
2. Makros sind Strings – also eine Reihenfolge von Zeichen – die erst zur Laufzeit der Anwendung erstellt werden. Diese werden entweder von dem Benutzer erzeugt oder z.B. von einer Datei gelesen. Das bedeutet, dass die Strings nicht zur Zeit der Entwicklung oder der Auslieferung des Softwaresystems bekannt sein müssen.
3. Makros bieten den Nutzer an, eine Anwendung um Funktionen zu erweitern, ohne dass die Anwendung erweitert werden muss.

- Eine Programmiersprache
- Eine Möglichkeit statische Systeme zu dynamischen umzuwandeln
- Kein Script

2016-06-11

Konzeption und Implementierung einer Makrosprache in C++

└ Was?

└ Was ist eine Makrosprache?

- Eine Programmiersprache
- Eine Möglichkeit statische Systeme zu dynamischen umzuwandeln
- Kein Script

1. Eine Makrosprache soll keine Scriptsprache sein. Zwar kann sich eine Makrosprache wie ein Script anfühlen, sollte aber keine sein.
2. Der Unterschied von Makros und Scripts liegt vor allem darin, dass Scripts komplett neue Funktionalitäten erzeugen und Makros genutzt werden um vorhandene Funktionen zu automatisieren. – Also ein Subset von Script darstellen.
3. Es ist ein kleiner Unterschied – der allerdings große Folgen hat. Eine Scriptsprache sollte eine wesentlich größere API anbieten, z.B. call-backs und threads support. Diese beiden würden Scripts ermöglichen, die nicht durch den Benutzer direkt angestoßen werden müssen und asynchron ihre Funktionen ausführen – z.B. eine automatische Sicherung nach 5 Minuten.

Was war meine Ausgangssituation?

2016-06-11

Konzeption und Implementierung einer Makrosprache
in C++

└ Was?

└ Was war meine Ausgangssituation?

1. Was war meine Ausgangssituation?

Was war meine Ausgangssituation?

2016-06-11

Konzeption und Implementierung einer Makrosprache
in C++

└ Was?

└ Was war meine Ausgangssituation?

• C++

- C++

1. Die Anwendung, für die diese Makrosprache entwickelt wurde, ist in C++ geschrieben.
2. Das hat den Vorteil, dass die Anwendung das Potenzial hat extrem effizient und schnell zu arbeiten.
3. Ein Nachteil ist, dass C++ eine statische Sprache ist – also nach dem Kompilieren nicht verändert werden, kann was ausgeführt werden soll.
4. Wäre sie in z.B. Python geschrieben, wären Makros nur ein weiteres Python Script welches geladen und abgearbeitet werden würde. (Was ziemlich einfach wäre zu implementieren.)

- C++
- Command-Pattern

2016-06-11

Konzeption und Implementierung einer Makrosprache
in C++

└ Was?

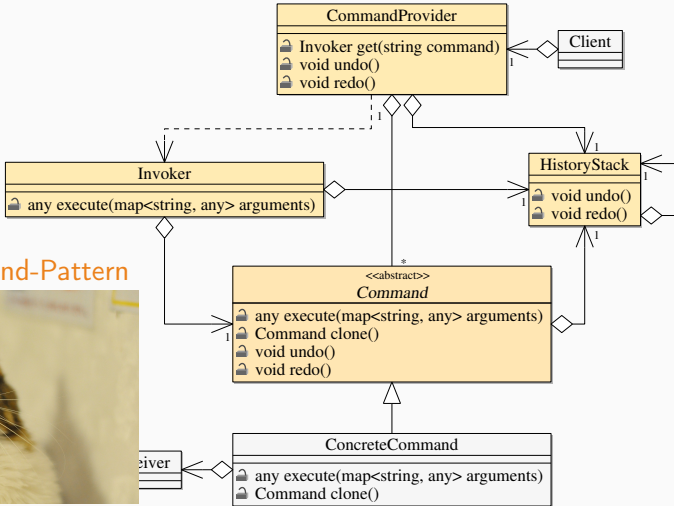
└ Was war meine Ausgangssituation?

1. Ein zentrales Element der vorhandenen Software ist eine Implementation des CommandPatterns.
2. Das Command-Pattern der Anwendung, ist die Schnittstelle, die die Makrosprache nutzt, um mit der Anwendung zu interagieren. ...

• C++
• Command-Pattern

Was war meine Ausgangssituation?


- C++
- Command-Pattern

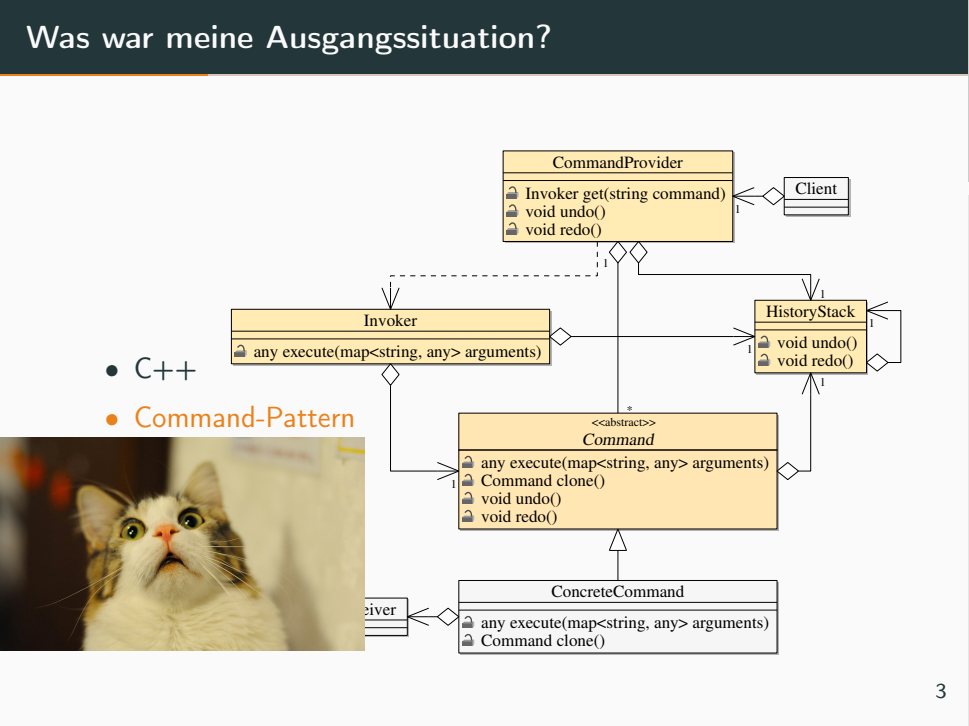


```
classDiagram
    class Client
    class CommandProvider {
        Invoker get(string command)
        void undo()
        void redo()
    }
    class Invoker {
        any execute(map<string, any> arguments)
    }
    class HistoryStack {
        void undo()
        void redo()
    }
    class Command {
        <<abstract>>
        any execute(map<string, any> arguments)
        Command clone()
        void undo()
        void redo()
    }
    class ConcreteCommand {
        any execute(map<string, any> arguments)
        Command clone()
    }
    Client --> CommandProvider
    CommandProvider o--> Invoker
    CommandProvider o--> HistoryStack
    Invoker o--> Command
    Invoker o--> HistoryStack
    CommandProvider ..> Invoker
    CommandProvider ..> HistoryStack
    CommandProvider ..> Command
    HistoryStack o--> Command
    Command <|-- ConcreteCommand
```

The diagram illustrates the Command Pattern structure. It includes classes: **Client**, **CommandProvider**, **Invoker**, **HistoryStack**, **Command** (abstract), and **ConcreteCommand**. **Client** interacts with **CommandProvider**. **CommandProvider** manages **Invoker** and **HistoryStack**, and interacts with **Command**. **Invoker** interacts with **Command** and **HistoryStack**. **HistoryStack** interacts with **Command**. **ConcreteCommand** inherits from **Command**. A small image of a cat is visible in the bottom left corner.

3

- 




2016-06-11

Konzeption und Implementierung einer Makrosprache in C++

Was?


Was war meine Ausgangssituation?



Was war meine Ausgangssituation?

• C++

• Command-Pattern




2016-06-11

Konzeption und Implementierung einer Makrosprache in C++

Was?


Was war meine Ausgangssituation?



Was war meine Ausgangssituation?

• C++

• Command-Pattern




2016-06-11

Konzeption und Implementierung einer Makrosprache in C++

Was?


Was war meine Ausgangssituation?



Was war meine Ausgangssituation?

• C++

• Command-Pattern




2016-06-11

Konzeption und Implementierung einer Makrosprache in C++

Was?


Was war meine Ausgangssituation?



Was war meine Ausgangssituation?

• C++

• Command-Pattern



- 2016-06-11

Konzeption und Implementierung einer Makrosprache in C++

Was?

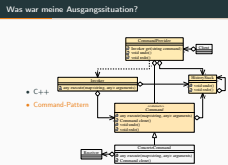
Was war meine Ausgangssituation?

Was war meine Ausgangssituation?

2016-06-11

Was?

Was war meine Ausgangssituation?



-
- ```

classDiagram
 class Client
 class CommandProvider {
 Invoker get(string command)
 void undo()
 void redo()
 }
 class Invoker {
 any execute(map<string, any> arguments)
 }
 class HistoryStack {
 void undo()
 void redo()
 }
 class Command {
 <<abstract>>
 any execute(map<string, any> arguments)
 Command clone()
 void undo()
 void redo()
 }
 class ConcreteCommand {
 any execute(map<string, any> arguments)
 Command clone()
 }
 class Receiver

 Client --> "1" CommandProvider
 CommandProvider --> "1" Invoker
 CommandProvider --> "1" HistoryStack
 Invoker --> "1" Command
 Invoker --> "1" Receiver
 HistoryStack --> "1" Command
 HistoryStack --> "1" ConcreteCommand
 CommandProvider --> "*" Command
 Command <|-- ConcreteCommand

```
- The diagram illustrates the Command Pattern structure:
- Client**: Interacts with **CommandProvider**.
  - CommandProvider**: Contains methods `Invoker get(string command)`, `void undo()`, and `void redo()`. It manages **Invoker**, **HistoryStack**, and **Command**.
  - Invoker**: Contains the method `any execute(map<string, any> arguments)`. It interacts with **Command** and **Receiver**.
  - HistoryStack**: Contains methods `void undo()` and `void redo()`. It interacts with **Command** and **ConcreteCommand**.
  - Command**: An abstract class defining methods `any execute(map<string, any> arguments)`, `Command clone()`, `void undo()`, and `void redo()`.
  - ConcreteCommand**: Implements the **Command** interface, containing methods `any execute(map<string, any> arguments)` and `Command clone()`. It interacts with **Receiver**.
  - Receiver**: The target that performs the request.

1. Der **Invoker** wird von dem **CommandProvider** an den **Client** gegeben – der **Client** kann z.B ein Knopf in einer GUI sein.
2. Der **HistoryStack** sorgt dafür, dass **Commands** rückgängig gemacht bzw. wiederhergestellt werden können.
3. Das **Command** ist die Elternklasse von dem **ConcreteCommand** welche ihre Funktionalität in der **execute** Methode implementiert. Die einen Rückgabewert vom Typ **any** erlaubt.
4. Ebenso zu beachten ist die **map<string, any>**. Diese erlaubt es beliebig viele Daten, mit beliebigen Datentypen, typischer als Parameter zu übergeben – ohne die Methodensignatur zu verändern.
5. Der **any** Typ – der beliebige Datentypen typischer wrapped – wird in C++17 enthalten sein.

2016-06-11

# Konzeption und Implementierung einer Makrosprache in C++

## └ Anforderungen

Anforderungen

## Anforderungen

---

### 1. Anforderungen an die Lösung. . .

- Dynamisches Typesystem

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└─Anforderungen

└─Anforderungen

• Dynamisches Typesystem

1. Die Makros sollten davon profitieren, dass die Commands Rückgabewerte und Parameter haben. Das heißt, dass es eine Möglichkeit geben muss, dass “unbekannte” Datentypen zwischengespeichert werden.
2. Unbekannt impliziert, dass die Makrosprache mit allen Datentypen arbeiten können muss, nicht nur den primitiven, die als Literals in dem Makro erstellt werden können.
3. Durch das dynamische Typsystem soll es also möglich sein, so etwas zu schreiben ...

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└─Anforderungen

└─Anforderungen

- Dynamisches Typesystem `var a = 10; a = "10";`

1. Hier wird der Variable **a** beim ersten Mal ein Integer zugewiesen und beim zweiten Mal ein String.
2. Die Lösung für das zwischenspeichern ist der any Typ, der von dem Command-Pattern genutzt wird.

- Dynamisches Typesystem
- Kontrollfluss

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Anforderungen

└ Anforderungen

• Dynamisches Typesystem  
• Kontrollfluss

1. Die Makros sollten auch eine Möglichkeit haben, auf den Zustand der Anwendung zu reagieren.  
Ein Beispiel wäre ...



- Dynamisches Typesystem

- **Kontrollfluss**

```
if(has_unsaved()) { save(); }
```

1. Das ein Makro nur probiert zu speichern, wenn es etwas zum speichern gibt.
2. Generell sollte speichern schnell gehen, allerdings kann es passieren, dass eine mehrere Gigabyte große Datei geschrieben werden muss, was dem Nutzer nicht zumuten ist.

- Dynamisches Typesystem
- Kontrollfluss
- Benutzerfreundlichkeit

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Anforderungen

└ Anforderungen

- Dynamisches Typesystem
- Kontrollfluss
- Benutzerfreundlichkeit

1. Außerdem soll die Makrosprache für den Benutzer leicht zu verstehen sein. Da der Benutzer auch der Endkunde sein kann ...

- Dynamisches Typesystem
- Kontrollfluss
- Benutzerfreundlichkeit

```
def main() {
 var bar = "1 ";

 return fun(foo:bar)
}
```

```
def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

### └ Anforderungen

### └ Anforderungen

1. Der Syntax sollte also deutlich machen, was passieren wird.
2. Durch **def** wird automatisch klar, das eine Funktion definiert wird und **var** definiert eine Variable.
3. Da die Implementation des CommandPatterns eine Map nutzt, um die Parameter des Commands anzugeben, habe ich mich entschieden named parameter zu nutzen (**foo:bar**). Diese schreiben keine Reihenfolge vor und ermöglichen das überladen von Funktionen anhand der Parameternamen.
4. Aber in meine Augen ist Syntax nur ein kleiner Punkt der Benutzerfreundlichkeit ...

Anforderungen

- Dynamisches Typesystem
- Kontrollfluss
- Benutzerfreundlichkeit

```
def main() {
 var bar = "1 ";

 return fun(foo:bar)
}

def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```

- Dynamisches Typesystem
- Kontrollfluss
- Benutzerfreundlichkeit

Anon:9:5: In the 'main' function defined here:

```
def main() {
```

^

Anon:12:22: Expected a ';'

```
 return fun(foo:bar)
```

^

1. In dem Beispiel wurde ein Semikolon in der main Methode vergessen, was die beiden Fehlermeldungen schnell deutlich machen.
2. Wenn der Nutzer einen Fehler macht, soll dieser möglichst einfach zu finden sein – man will mit einem Tool arbeiten, nicht dagegen kämpfen.
3. Die Fehlermeldungen müssen dem Nutzer nicht nur sagen was falsch ist, sondern auch wo und weshalb.

- Dynamisches Typesystem
- Kontrollfluss
- Benutzerfreundlichkeit
- Wartbarkeit und Macht

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

### └ Anforderungen

### └ Anforderungen

- Dynamisches Typesystem
- Kontrollfluss
- Benutzerfreundlichkeit
- Wartbarkeit und Macht

1. Neben der Wartbarkeit, war auch die Macht dieser Makrosprache wichtig.
2. Wie der Syntax gezeigt hat, geht die Sprache in die Richtung von Scripts, man kann Funktionen anlegen, primitive Datentypen nutzen und hat Kontrollstrukturen zur Verfügung.
3. Da die Makros letztendlich nur auf Commands arbeiten können, ist das Gefahrenpotenzial – was von unwissenden Nutzern ausgeht – größtenteils eingeschränkt.

# Konzeption

---

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

### └ Konzeption

Konzeption

---

1. Kommen wir nun zur Konzeption
2. Die folgende Liste zeigt nicht nur, was entwickelt werden musste, sondern auch den abstrakten Ablauf, der beim Ausführen eines Makros stattfindet.

- Tokenizer

2016-06-11

Konzeption und Implementierung einer Makrosprache  
in C++

└─Konzeption

└─Was wurde entwickelt?

1. Als erstes brauchen wir einen Tokenizer.
2. Der Tokenizer muss den String / das Makro zu einer Liste von Tokens – also Teilstücke – verwandeln ...

• Tokenizer

## • Tokenizer

```
def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```

→

|     |      |       |
|-----|------|-------|
| l:1 | c:1  | t:def |
| l:1 | c:5  | t:fun |
| l:1 | c:8  | t:(   |
| l:1 | c:9  | t:foo |
| l:1 | c:12 | t:)   |
| l:1 | c:14 | t:{   |
| l:2 | c:3  | t:do  |

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

### └ Konzeption

### └ Was wurde entwickelt?

• Tokenizer

```
def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```

→

|     |      |       |
|-----|------|-------|
| l:1 | c:1  | t:def |
| l:1 | c:5  | t:fun |
| l:1 | c:8  | t:(   |
| l:1 | c:9  | t:foo |
| l:1 | c:12 | t:)   |
| l:1 | c:14 | t:{   |
| l:2 | c:3  | t:do  |

1. Wichtig ist, dass bei dem erstellen der Liste, keine nützlichen Informationen verloren gehen.
2. Zwar wird die Reihenfolge durch das Tokenizen nicht verändert, allerdings gehen überflüssige Leerzeichen und Zeilenumbrüche verloren.
3. Diese sind für Fehlermeldungen extrem wichtig und müssen mitgespeichert werden.



- Tokenizer
- Abstrakter Syntaxbaum

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Konzeption

└ Was wurde entwickelt?

1. Für den Folgenden Schritt muss es Klassen geben, die einen abstrakten Syntaxbaum bilden.
2. Ein Abstrakter Syntaxbaum beschreibt den Syntax aus dem Makro, ohne Informationsverlust und ist nur ein passiver Bestandteil des Ablaufes.
3. ASTs werden genutzt, da es einfacher ist mit ihnen als Datenstruktur zu arbeiten, als mit einem String.
4. Die meisten Elemente des Syntaxes lassen sich als Klasseninstanzen in dem AST wiederfinden. Prominente Ausnahmen sind Klammern, Semikolons und Leerzeichen.

- Tokenizer
- AST
- Parser

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└─Konzeption

└─Was wurde entwickelt?

1. Das nächste aktive Element ist ein Parser.
2. Der Parser ist dafür zuständig, dass die Liste von Tokens in einen abstrakten Syntaxbaum umgewandelt werden. . . .

- Tokenizer
- AST
- Parser

# Was wurde entwickelt?

- Tokenizer
- AST
- **Parser**

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

@Define {
 token: def
 @Function {
 token: fun
 parameter:
 @Variable {
 token: foo
 }
 @Scope {
 ...
 }
 }
}
```

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Konzeption

└ Was wurde entwickelt?

1. Das heißt das der Parser die Tokens in **diese** Form überführen muss.
2. Die **@-Symbole** weisen auf eine AST Objektinstanz hin.
3. Das was in den Klammern ({} ) eingeschlossen ist, ist ein Bestandteil des umschließenden Objektes.



- Tokenizer
- AST
- Parser
- **Analyser**

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Konzeption

└ Was wurde entwickelt?

- Tokenizer
- AST
- Parser
- **Analyser**

1. Als vorletztes Element bedarf es einem Analyser.
2. Der Analyser ist dafür da, um Fehler zu finden, die nicht aus dem Parsen hervorgehen. Also Fehler die nicht syntaktisch sind, oder nur schwer beim Parsen zu finden sind. ...

- Tokenizer
- AST
- Parser

• **Analyser**     **def** gun(){ **break**; }

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└─Konzeption

└─Was wurde entwickelt?

1. Ein Beispiel wäre die Funktion **gun**.
2. Der AST kann den Code darstellen. Eine Funktionsdefinition, die in dem Funktionsscope ein break Element hat.
3. Bloß macht dies wenig Sinn, da **break** nur in Schleifen eine Funktion hat.
4. Es ist also wichtig dem Nutzer zu sagen, dass das was er geschrieben hat höchst wahrscheinlich nicht das ist, was er erwartet.

- Tokenizer
- AST
- Parser
- **Analyser**     **def** gun(){ **break**; }

- Tokenizer
- AST
- Parser
- Analyser
- Interpreter

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Konzeption

└ Was wurde entwickelt?

- Tokenizer
- AST
- Parser
- Analyser
- Interpreter

1. Letztlich bedarf es einem Interpreter.
2. Der Interpreter interpretiert den AST, den der Parser erzeugt hat und ist die Komponente – bzw. der Client, aus dem CommandPattern – der die Commands ausführt.
3. Somit kommen wir zu wir ... (zu weiteren UML Diagrammen)

2016-06-11

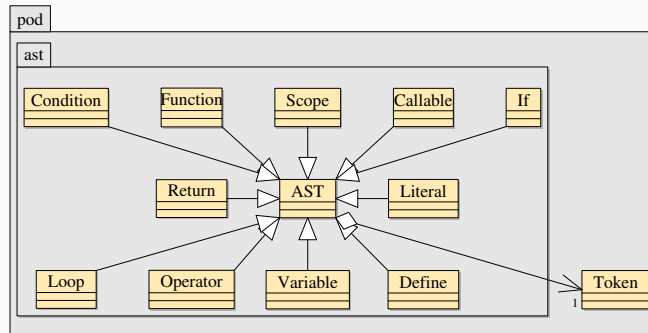
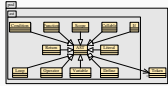
Konzeption und Implementierung einer Makrosprache  
in C++

└ Konzeption

└ AST Klassenhierarchie

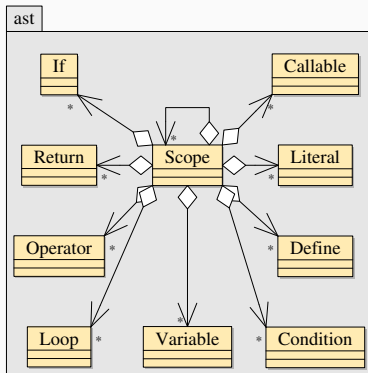


1. Zu weiteren UML Diagrammen.
2. Für meinen Geschmack helfen die Diagramme nicht genug.
3. Sie sind aber die beste Möglichkeit, die ich kenne, anderen Menschen mein Wissen, über die Datenstrukturen, zu mitzuteilen.



1. Die AST Klassenhierarchie.
2. Alle abstrakten Syntax Klassen haben die **AST** Klasse als Elternklasse. Diese hat ein **Token**, welches das Stück des Makros enthält, welches es repräsentieren soll.



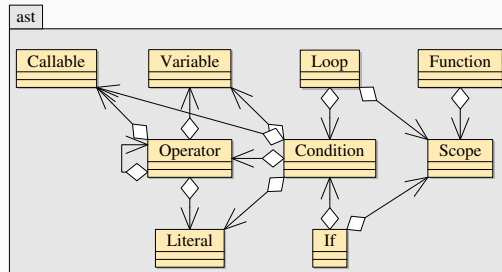


1. Das **Scope** kann Instanzen von allen AST Klassen aufnehmen.
2. Durch die Aufnahme von anderen AST Instanzen wird hier ein Teil des Baumes erstellt.

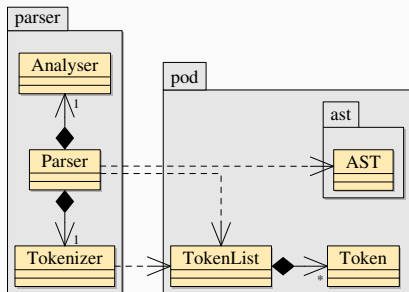
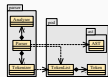
# Konzeption und Implementierung einer Makrosprache in C++

└─Konzeption

└─AST Klassen



1. Dieses UML Diagramm zeigt die restlichen Abhängigkeiten der AST Klassen – die meisten Verbindungen sollten Selbstverständlich sein.
2. Ein Bestandteil einer **Funktion** ist ein **Scope**. Ein **Loop** besteht aus einem **Scope** und einer **Condition**. Eine **Condition** kann aus **Variablen**, Funktionsaufrufen (**Callable**) usw. bestehen.



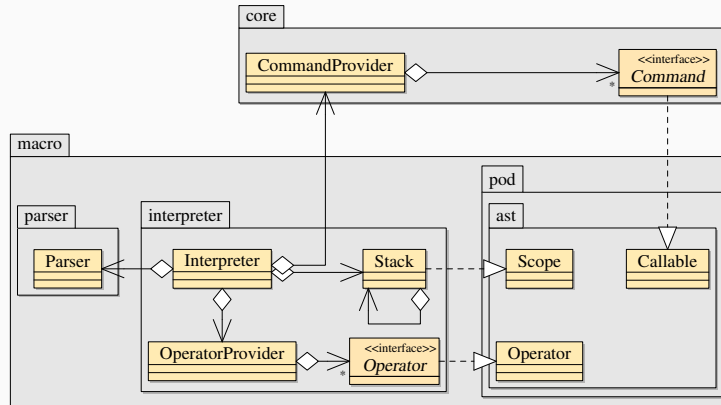
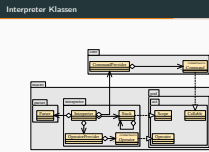
1. Das vorletzte Diagramm zeigt die Parser Klassen.
2. Der **Parser** nutzt den **Tokenizer** um eine **TokenList** von dem Makrostring zu erstellen.
3. Diese TokenListe wandelt er dann in einen **AST** um.
4. Der produzierte AST wird dann von dem **Analyser** überprüft.

2016-06-11

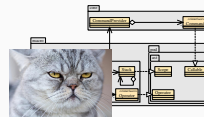
## Konzeption und Implementierung einer Makrosprache in C++

└ Konzeption

└ Interpreter Klassen



1. Letztlich die Interpreter Klassen.
2. Der **Interpreter** nutzt den **Parser** um einen AST zu erzeugen, den er dann interpretieren kann.
3. Um dies zu tun, muss der Interpreter auch Daten verwalten. Das wird von dem **Stack** übernommen.
4. Der Stack bildet einen Stack mit Instanzen von sich selber ...



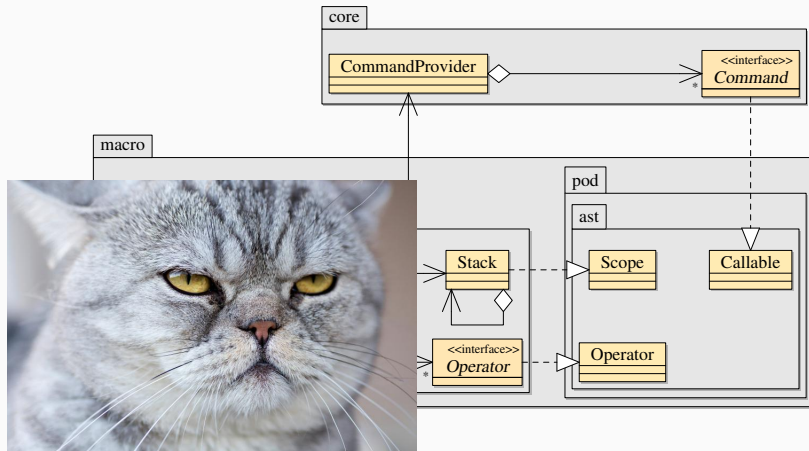
# Interpreter Klassen

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Konzeption

└ Interpreter Klassen




1. Diese Art und Weise einen Stack aufzubauen ist normal in C oder C++, da man dort Pointer zur Verfügung hat.

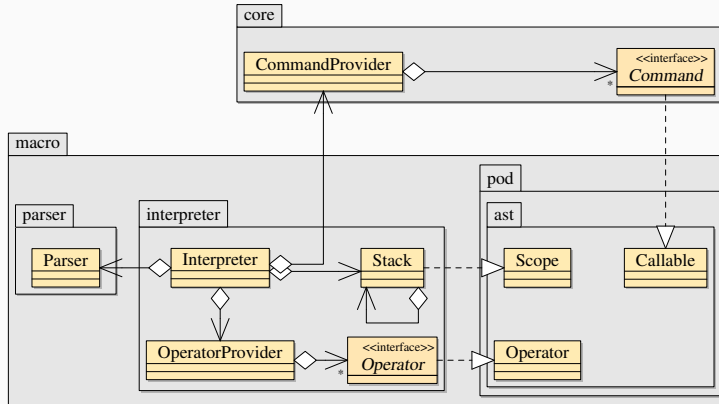
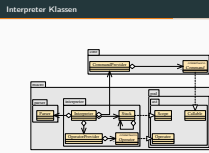
```
..... Stack: 0 Funktion: main Variable: a
def main() { Stack: 1 Funktion: Variable: a }
var a;
if(a) { Stack: 2 Funktion: Variable:
...
}
}
```

```
.....> Stack: 0 Funktion: main Variable:
def main() {> Stack: 1 Funktion: Variable: a
var a;

if(a) {> Stack: 2 Funktion: Variable:
...
}
}
```



1. **Hier** wird der Stack durch die drei Stack Instanzen (**0, 1 und 2**) gebildet.
2. Ein späteres Beispiel wird wahrscheinlich mehr Aufschluss über die Funktionsweise und Vorteile bringen.



1. Der **OperatorProvider** wird von dem Interpreter genutzt um die **Operatoren** – wie + und – – auf Variablen anzuwenden.
2. Die Nutzung eines OperatorProviders sorgt dafür, dass Operatoren für beliebige Datentypen angeboten werden können, ohne den Interpreter anzupassen.
3. Der Code des OperatorProviders ist sehr Interessant – leider haben wir nicht genug Zeit um diesen durchzugehen. Weshalb ich euch wieder mit dem any Typ abspeisen muss, der auch hier ein wichtiger Eckpunkt ist.
4. Um Funktionsaufrufe auszuführen – die nicht zu selber definierten Funktionen aus dem Makro führen – nutzt der Interpreter den **CommandProvider**, die letztendlich die Komponente sind, die die Anwendung verändern.

## Realisierung

---

2016-06-11

Konzeption und Implementierung einer Makrosprache  
in C++

└ Realisierung

Realisierung

---

1. Das war die trockene Theorie.
2. In den folgenden Folien werde ich an dem vorherigen Beispiel die einzelnen Schritte erklären, die beim Interpretieren durchlaufen werden. Also das, was in der Demo im Hintergrund passieren wird.



2016-06-11

Konzeption und Implementierung einer Makrosprache  
in C++

└ Realisierung

└ Tokenizer

1. Der Tokenizer hat drei Hauptartenarten von Tokens

- Normale Tokens

1. Normale Tokens werden durch alle Zeichen getrennt, die keine Buchstaben, Zahlen oder Unterstriche sind.
2. Oder durch sich selber ein Token bilden z.B. eine Klammer.

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

└ Tokenizer

- Normale Tokens
- Strings

- Normale Tokens
- Strings

1. Strings sind besonders, ... (da sie ein Token bilden müssen)

- Normale Tokens

- Strings

```
print("print(\"Hallo Welt!\")")
```

1. Da sie ein Token bilden müssen. Sonst würden die Leerzeichen verloren gehen.
2. Außerdem müssen die Anführungszeichen richtig interpretiert werden.

- Normale Tokens
- Strings
- Floats

1. Dezimalzahlen sind – ähnlich wie Strings – besondere Tokens. . . (Da die beiden Teile der Zahl durch ein Punkt getrennt werden.)

- Normale Tokens
- Strings
- Floats

0.1 oder .1234

1. Da die beiden Teile der Zahl durch ein Punkt getrennt werden.

2016-06-11

Konzeption und Implementierung einer Makrosprache  
in C++

└ Realisierung

└ Parser

1. Kommen wir zu den Parser.
2. Der Parser ist der komplexeste Teil der Arbeit – aus Zeitgründen werde ich nur zwei wichtige Punkte anschneiden und ansonsten behaupten, das es einfach funktioniert.

- Recursive descent

1. Der Parser wurde recursive descent implementiert.
2. Das heißt, dass die Grammatik der Sprache durch Funktionsaufrufe abgebildet wird. Anstelle von Zustandsübergangstabellen.
3. Was das bedeutet, zeige ich an dem folgenden Punkt.



2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

└ Parser

- Recursive descent
- Operatoren

- Recursive descent
- Operatoren

1. Ein schwieriger Punkt beim parsen ist das zusammenstellen der Operatoren. . .

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

└ Parser

• Recursive descent  
• Operatoren `var foo = 1 - 1;`

- Recursive descent

- Operatoren `var foo = 1 - 1;`

1. Der Ablauf wäre wie folgt. . .

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

└ Parser

• Recursive descent  
• Operatoren `var foo = 1 - 1;`

- Recursive descent

- Operatoren

var foo = 1 - 1;

1. Parsen einer Deklaration(**var**).

- Recursive descent

- Operatoren

```
var foo = 1 - 1;
```

1. Für die Variable(**foo**).

- Recursive descent

- Operatoren

```
var foo = 1 - 1;
```

1. Nach dieser Funktion wird die Funktion zum Wertzuweisung – bzw. Operator – `parsen(=)` aufgerufen.

- Recursive descent

- Operatoren

```
var foo = 1 - 1;
```

1. Welche dann die Funktion zum parsen eines Literals(**1**) aufruft.  
Diese returned zurück zur Zuweisungsoperatorfunktion.

- Recursive descent

- Operatoren

```
var foo = 1 - 1;
```

1. Die dann die Funktion zum parsen vom Binären-Operatoren(-) aufruft.

- Recursive descent

- Operatoren

```
var foo = 1 - 1;
```

1. Welche wiederum die Funktion für Literals(1) aufruft.
2. Durch diese Reihenfolge wird allerdings nicht der richtige Baum erzeugt, sondern eine Art Liste, die aus Definition, Operator, Literal, Operator und Literal besteht.



- Recursive descent
- Operatoren

```
var foo = 1 - 1;
```

1. Die Liste muss durch einen zweiten Schritt in den Operator Funktion zu einem Operator mit zwei Literals und einem Operator mit einer Variable und einem Operator umgewandelt werden.
2. Es werden also die jeweiligen linken Werterzeuger – also Variable und Literal – zu dem rechten Operator gezogen. Dafür muss der Zuweisungsoperator sogar in die Definition der Variable greifen.

- Recursive descent

- Operatoren

`var foo = 1 --- 1;`

1. Und hier?
2. Hier wird das zweite Literal zweimal negiert und dann vom ersten abgezogen.
3. Wie genau das Abläuft kann ich leider aus Zeitgründen nicht zeigen.

2016-06-11

Konzeption und Implementierung einer Makrosprache  
in C++

└ Realisierung

└ Interpreter

1. Womit wir beim Interpreter wären.
2. Der Interpreter ist sehr einfach – um nicht primitiv zu sagen –  
wodurch er wenig Logik besitzt, die zu erklären ist.

2016-06-11

Konzeption und Implementierung einer Makrosprache  
in C++

└ Realisierung

└ Interpreter

• Läuft den AST ab

- Läuft den AST ab

1. Der Interpreter läuft den generierten AST ab.

2016-06-11

Konzeption und Implementierung einer Makrosprache  
in C++

└ Realisierung

└ Interpreter

- Läuft den AST ab
- `OperatorProvider`

- Läuft den AST ab
- `OperatorProvider`

1. Macht sich den `OperatorProvider` für Operationen zunutze

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

└ Interpreter

- Läuft den AST ab
- OperatorProvider
- CommandProvider

- Läuft den AST ab
- OperatorProvider
- CommandProvider

1. Ruft Commands über den CommandProvider auf

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

└ Interpreter

- Läuft den AST ab
- OperatorProvider
- CommandProvider
- Stack

- Läuft den AST ab
- OperatorProvider
- CommandProvider
- Stack

1. Und überlässt die Daten und Funktionsverwaltung dem Stack.

2016-06-11

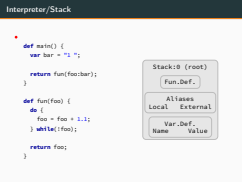
Konzeption und Implementierung einer Makrosprache  
in C++

└ Realisierung

└ Interpreter/Stack

1. Somit kommen wir zum Letzten Teil – dem Stack bzw. was beim interpretieren passiert.



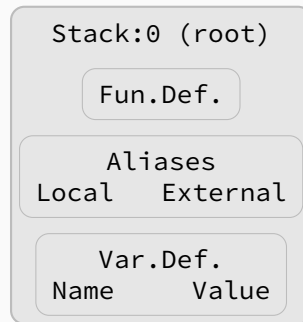


```
def main() {
 var bar = "1 ";

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

```
def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```



1. Der rote Punkt gibt die Zeile an, welche gerade interpretiert wird bzw. den Stack verändert.
2. Am Anfang gibt es eine Root Node, die durch die root Stack Instanz abgebildet wird. . .

```

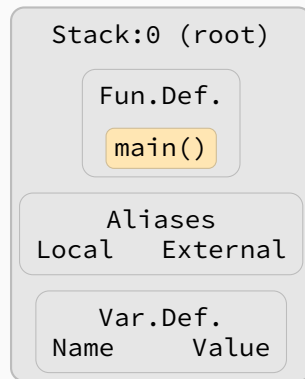
• def main() {
 var bar = "1 ";

 return fun(foo:bar);
}

def fun(foo) {
 do {
 foo = foo + 1.1;
 } while (!foo);

 return foo;
}

```



2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

└ Interpreter/Stack

```

• def main() {
 var bar = "1 ";

 return fun(foo:bar);
}

def fun(foo) {
 do {
 foo = foo + 1.1;
 } while (!foo);

 return foo;
}

```



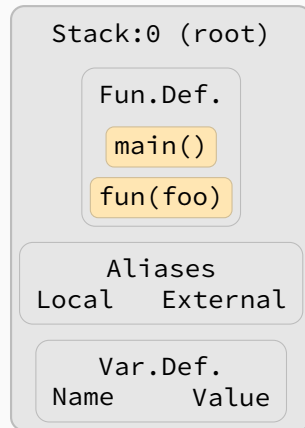
1. Als ersten Schritt definiert der Interpreter die Funktionen eines Scopes, bevor er anfängt diese zu interpretieren.
2. Dadurch kann **main...** ( vor **foo** definiert werden.)

```
def main() {
 var bar = "1 ";

 return fun(foo:bar);
}

• def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```



2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

└ Interpreter/Stack

1. Vor **foo** definiert werden.

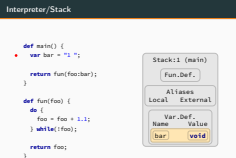
```
def main() {
 var bar = "1 ";

 return fun(foo:bar);
}

• def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```



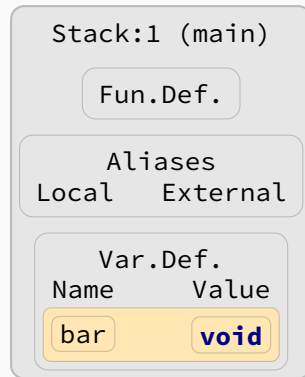


```
def main() {
 var bar = "1 ";

 return fun(foo:bar);
}

def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```



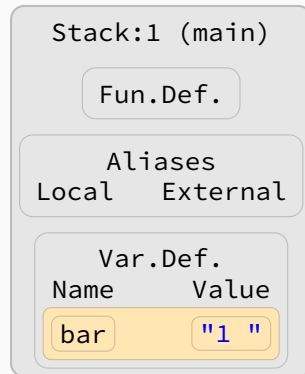
1. Als nächstes wird die **main** Methode aufgerufen.
2. Dadurch wird eine neue Stack Instanz erzeugt, dieser kann den root Stack nach Variablen und Funktionen fragen.
3. Dann wird in dem Stack die neue Variable **bar** angelegt, der (der String) ...

```
def main() {
 var bar = "1 ";

 return fun(foo:bar);
}

def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```



1. Der String "Eins Leerzeichen" zugewiesen wird.

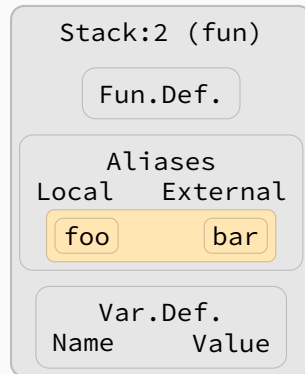


```
def main() {
 var bar = "1 ";

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

```
def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```



1. Danach wird **fun** aufgerufen, wodurch wieder eine Stack Instanz (**fun**) zu dem Stack hinzugefügt wird.
2. Der Stackinstanz wird **foo** als Alias auf **bar** mitgegeben.

```
def main() {
 var bar = "1 ";

 return fun(foo:bar);
}

def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```

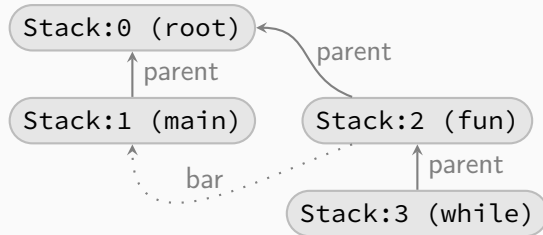


```
def main() {
 var bar = "1 ";
```

```
• return fun(foo:bar);
}
```

```
def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);
```

```
 return foo;
}
```

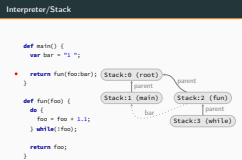


2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

└ Interpreter/Stack



1. Es gibt nun also zwei Stacks die jeweils aus zwei Stack Instanzen bestehen – bzw. im nächsten Schritt aus zwei und drei.
2. Wieso?
3. Wenn es nur einen Stack geben würde – der **fun Stack** also den **main Stack** haben würde, könnte man aus der fun Funktion die bar Variable verändern. Das will man aber nicht.

```
def main() {
 var bar = "1 ";

 return fun(foo:bar);
}

def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```

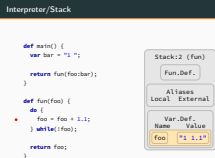


2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

└ Interpreter/Stack



1. Bei dem referenzieren von foo – also bar – ist nur lesen erlaubt.
2. Das heißt nach dem Addieren von dem String "Eins Leerzeichen" und der Dezimalzahl "1.1", wird der Alias gelöscht und eine Variable foo angelegt, die das Ergebnis speichert wird.
3. Abgesehen von Scope Sicherheit bietet dieses Verfahren weniger Speicherverbrauch und sollte deswegen generell schneller sein.

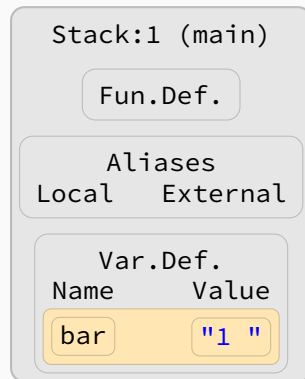


```
def main() {
 var bar = "1 ";

 return fun(foo:bar);
}

def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);

 return foo;
}
```



```
def main() {
 var bar = "1 ";
 return fun(foo:bar);
}

def fun(foo) {
 do {
 foo = foo + 1.1;
 } while(!foo);
 return foo;
}
```



1. Nach dem return aus fun, ist als bar unverändert.
2. Und der String aus fun wird in die C++ Ebene returned.

2016-06-11

## Konzeption und Implementierung einer Makrosprache in C++

└ Realisierung

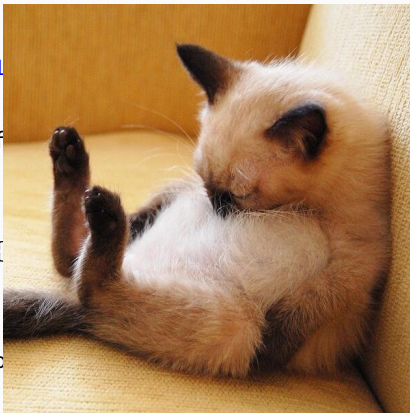
└ Interpreter/Stack

```
def main() {
 var bar = "1"
```

```
• return fun(f
}
```

```
def fun(foo) {
 do {
 foo = foo
 } while(!foo
```

```
 return foo;
}
```



:1 (main)

n.Def.

Aliases

External

r.Def.

Value

"1 "



1. Ihr habt es geschafft!
2. Welche Fragen wollt ihr nun schon stellen, bevor ich die kurze Demo zeige?

Fragen?

2016-06-11

Konzeption und Implementierung einer Makrosprache  
in C++

└ Realisierung

Fragen?

- **Katzen:** Ich habe genug theoretische und zum teil langweilige Präsentationen in meinem Studium gesehen. Dem Trend wollte ich nicht folgen – und auch wenn meine Präsentation nicht so viel Wissen vermittelt hat, hatte sie zumindest Katzen. Wer reine Theorie will, ist mit meiner Thesis besser beraten als mit mir.

2016-06-11

# Konzeption und Implementierung einer Makrosprache in C++

└ Demo

Demo

---

## Demo

---

## Weitere Fragen?

- **Katzen:** Ich habe genug theoretische und zum teil langweilige Präsentationen in meinem Studium gesehen. Dem Trend wollte ich nicht folgen – und auch wenn meine Präsentation nicht so viel Wissen vermittelt hat, hatte sie zumindest Katzen. Wer reine Theorie will, ist mit meiner Thesis besser beraten als mit mir.