JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Rainer Gutkas

Angefertigt am
Institut für Systemsoftware

Beurteiler / Beurteilerin
Günther Blaschek

November 2016

Design und Implementierung einer interaktiven Programmiersprache

Masterarbeit

zur Erlangung des akademischen Grades

Dipl.Ing.

im Masterstudium

Computer Science

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

DESIGN UND IMPLEMENTIERUNG EINER INTERAKTIVEN PROGRAMMIERSPRACHE

In dieser Arbeit stelle ich eine von mir entworfene, interaktive Programmiersprache vor, die den Namen Chill trägt. Chill ist als Unterstützung für den Unterricht gedacht und soll es ermöglichen, Programmieren zu lernen.
In Chill können einzelne Anweisungen eingeben und direkt ausgewertet werden, oder es können ganze Programme erstellt werden. Chill bietet Möglichkeiten der strukturierten Programmierung, wie Verzweigungen, Schleifen und Funktionen. Chill bietet integrierte und benutzerdefinierte Datentypen. Chill unterstützt eine auf Klassen und Vererbung basierte Objektorientierung.
Chill ist in eine Verwaltungsumgebung eingebunden, die es erlaubt, Kurse zu erstellen und abzuhalten. Die Verwaltungsumgebung und die Sprache Chill wurde für den Einsatz in Web Browsern in Javascript implementiert. Die Verwaltungsumgebung benötigt einen Web-Server mit PHP-Unterstützung und MySQL-Datenbank.

In this thesis I introduce an interactive programming language, called Chill. Chill is meant for supporting tutition of learning how to program.
Chill allows to enter single instructions, which are executed at once and Chill allows to enter and execute whole programs. Chill offers possibillities of structured programming, such as branching, loops and functions. Chill offers built in and user defined Data types. Chill supports Object oriented programming, based on classes and inheritance.
Chill is embedded in an enviroment, which can be used to design and offer courses. This enviroment as well as the language implementation are written in Javascript, so they can be executed in Web Browsers. The Sytsem is served by a Web-Server with PHP and MYSQL support.

Inhaltsverzeichnis

1. Einleitung

1.1. Aufgabenstellung

Interaktive Programmiersprachen erlauben es den Anwendern, einzelne Anwei- sungen oder Ausdrücke einzugeben und sofort auswerten zu lassen. Darüber hi- naus gibt es meist einfache Möglichkeiten für die Definition von Funktionen oder zum Speichern von Programmen für die spätere Benutzung.
Interaktive Sprachen wie BASIC und APL waren in den 1960er Jahren populär; heute ist es um dieses Konzept still geworden. In dieser Masterarbeit soll eine neue interaktive Sprache entwickelt werden, die vor allem in der Lehre, aber auch für kleine Alltagsaufgaben verwendet werden kann.
Die Masterarbeit umfasst die Sprachdefinition, die Implementierung eines Compi- lers und Laufzeitsystems und die Entwicklung einer einfachen Programmierumge- bung, in der man Anweisungen eingeben und auswerten lassen kann. Die Pro- grammierumgebung soll die Wirkung von Anweisungen veranschaulichen und bei der Behebung von Fehlern behilflich sein.
Der Sprachentwurf und die Gestaltung der Benutzerschnittstelle sind in Absprache mit dem Betreuer vorzunehmen. Die Programmierumgebung soll möglichst platt- formunabhängig sein, idealerweise als web-basiertes System, das in einem Web- Browser verwendet werden kann.

1.2. Überblick

Am Beginn gebe ich einige Beispiele von interaktiven Programmiersprachen. Dann werde ich die Sprache Chill im Detail vorstellen, das Ergebnis meines Sprachentwurfs. Ich beginne dabei bei Sprachelementen, die ausschlaggebend waren für das Design der interaktiven Umgebung und setze dann bei Sprachkonstrukten fort, die die Programmierumgebung von Chill beeinflusst haben.
Danach stelle ich die Verwaltungsumgebung vor, die es erlaubt, Programmierkurse zu erstellen und durchzuführen.
In Kapitel 4 gehe ich auf die Implementierung ein, beginnend bei der Ansteuerung des MySQL – Servers, der die Daten für die Verwaltungsumgebung verwaltet, über die Verwaltungsumgebung zur Programmierumgebung, Compiler und Laufzeitumgebung.
Danach möchte ich 2 Beispiele anführen, die zeigen, wie Chill erweitert werden kann.
Abschließend werde ich mit einer kritischen Betrachtung meiner Arbeit.

2. Grundlagen

Eine interaktive Programmiersprache erlaubt es, Kommandos einzugeben und sofort auswerten zu lassen. Bekannte Beispiele von interaktiven Programmiersprachen aus den 60'er Jahren sind APL und Basic. Während APL, dessen Syntax hauptsächlich auf Symbolen basiert, für algorithmische Notation bei IBM entwickelt wurde, war das Ziel, dass mit Basic verfolgt wurde, eine für Anfänger geeignete Allzweck Programmiersprache zu entwerfen. Der Name BASIC steht für „Beginner's All purpose Symbolic Instruction Code“.
Programme wurden in BASIC mit Zeilennummern versehen. Möglichkeiten der strukturierten Programmierung, wie Funktionen, fehlten. Für Sprünge zu Unterprogrammen standen GOTO <Zeilennummer> und in einigen BASIC Dialekten GOSUB <Zeilenummer> - RETURN zur Verfügung.
Eine aus den 90'er Jahren stammende Programmiersprache, Python, ist heutzutage sehr beliebt. Da Python Interpreter auch einen interaktiven Modus erlauben und auch Python Shell's existieren, kann Python als interaktive Programmiersprache gesehen werden.
Python verfolgt Konzepte der strukturierten-, der objektorientierten- und auch manche Konzepte der funktionalen Programmierung. Das Designziel von Python war es, eine einfache und übersichtliche Sprache zu bieten. Python beschränkt sich daher auf wenige syntaktische Konstrukte. Es besitzt zum Beispiel nur zwei Schleifenformen, for und while. Für Verzweigungen benutzt Python eine if elif else Anweisung. Ein switch existiert nicht. Code in Python wird durch Einrückungen strukturiert, nicht durch geschwungene Klammern, wie in vielen anderen Programmiersprachen. Python unterstützt Funktionen.
Alles in Python ist ein Objekt, die zahlreichen eingebauten Typen und auch Klassen. Es gibt keine statischen Typen, die Typvergabe und Typ - Prüfung sind dynamisch.
Python Implementierungen sind für fast jede Plattform verfügbar. Python ist in den meisten Unix basierten System als Standard enthalten. Trinket.io bietet sogar die Möglichkeit, Python im Browser auszuführen.

3. Benutzung

3.1. Die Sprache Chill

Die Sprache Chill kann auf 2 Arten verwendet werden, interaktiv durch Eingabe von einzelnen Anweisungen, oder durch Eingabe eines Programmes, das dann gesammelt ausgeführt werden kann.
Die interaktive Eingabe ist gedacht, um sich mit grundlegenden Sprachelementen vertraut machen zu können. Die Eingabe von Programmen, um sich mit fortführenden Konzepten vertraut machen zu können.
Im Folgenden wird die Sprache aufbauend erklärt, beginnend bei dem Sprachumfang für Interaktion.
Für die Beschreibung der Sprachelemente wird EBNF verwendet. Ziffern werden als Digit bezeichnet, Buchstaben als Letter, sonstige Zeichen als Symbol.

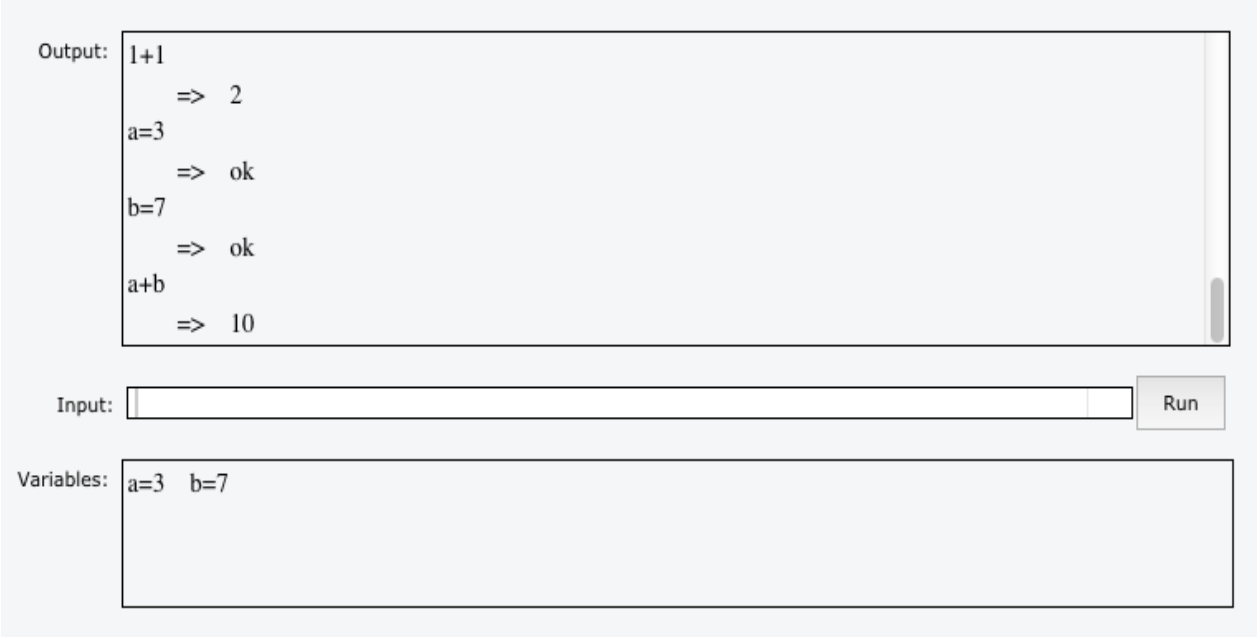
```
Digit     := "0" ... "9"

Letter    := "a"  ... "z" | "A" ... "Z"

Symbol    :=  "^"  ...  "*"
```

3.1.1. Interaktiv

Für die interaktive Verwendung von Chill steht eine einfache Bedienungsoberfläche zur Verfügung. Im oberen Bereich des Bildschirms steht die Ausgabe im Feld Output. Darunter folgt die Eingabezeile, Input. Hier können Anweisungen eingegeben und durch die Eingabetaste bestätigt werden. Daraufhin wird die Eingabe übersetzt und von der Laufzeitumgebung bearbeitet. Das Ergebnis der Verarbeitung wird dann in Output angezeigt.



Das unterste Feld Variables dient der Anzeige von Werten, die in der Laufzeitumgebung gespeichert sind. Das werde ich in Kapitel 3.1.1.5 genauer behandeln. Zuerst beginne ich mit einfachen Funktionen wie dem Lösen von Rechnungen.

3.1.1.1. Ganze Zahlen

Es gibt zwei Arten von Zahlen in Chill. Die erste sind ganze Zahlen, die in der Folge mit „integer“ bezeichnet werden.
Ganze Zahlen bestehen aus einer Ziffer, gefolgt von beliebig vielen Ziffern.

```
Factor      := Integer
Integer     := Digit {Digit}
```

Beispiele:

```
0
123
875218
```

Mit Zahlen kann man rechnen. Dabei wird * für eine Multiplikation, / für eine Division und % für eine Restwertrechnung verwendet. Rechnungen mit ganzen Zahlen ergeben wieder ganze Zahlen. Eine Division durch 0 ist *nicht* erlaubt!

```
Product     := Factor {("*" | "/" | "%") Factor}
```

Beispiele:

```
5 * 3           =>   15
5 /  3          =>   1
5%3             =>   2
5/0             =>   Error : Division by Zero
```

Zahlen können Vorzeichen haben. Das Zeichen + steht in Chill für Plus und das Zeichen – für Minus. Diese beiden Zeichen lassen sich auch für Addition und Subtraktion einsetzen.

```
Sum             := ["+" | "-"] Product {("+" | "-") Product}
```

Beispiele:

```
-5+1*3          =>   -2
5*2-1           =>    9
```

Punktrechnung geht in Chill vor Strichrechnung. Es ist aber auch möglich, Klammersetzung zu verwenden. Es können beliebig viele runde Klammern verwendet werden, allerdings muß jede geöffnete Klammer wieder geschlossen werden.

```
Factor          := "(" Sum ")" | Integer
```

Beispiele:

```
(5*2)-1         =>   9
5*(2-1)         =>   5
```

3.1.1.2. Kommazahlen

Die Zweite Art stellen Kommazahlen dar, die in der Folge als „number“ bezeichnet werden. Kommazahlen beginnen mit einer Ziffer, gefolgt von beliebig vielen Ziffern. Als Kommazeichen dient ein . . Danach folgt wieder mindestens eine Ziffer.

```
Number              := Digit {Digit} "." Digit {Digit}
```

Beispiele:

```
0.01
123.321
87.5218
```

Somit bestehen Zahlen in Chill aus Ganzen- und Kommazahlen.

Factor := "(" Sum ")" | Integer | Number

Kommazahlen können wie ganze Zahlen addiert, subtrahiert, multipliziert und dividiert werden. Kommazahlen und ganze Zahlen können in Rechnungen auch gemeinsam angewendet werden, wobei das Ergebnis eine Kommazahl ist. Sollte nur eine 0 hinter dem Komma stehen, wird nur die Zahl vor dem Komma angezeigt.

Beispiele:

```
5.5*2-1         =>   10
5*(2.5-1)       =>   2.5
```

Restwertrechnungen dürfen keine Kommazahlen beinhalten. Auch eine Division durch 0 ist *nicht* erlaubt.

3.1.1.3. Zeichen und Zeichenketten

Zeichen können auch in Chill angegeben werden. Um Zeichen wie Buchstaben, Ziffern oder Symbole als Zeichen zu kennzeichnen werden sie zwischen zwei Hochkommata ' ' geschrieben. Zeichen werden in der Folge als „character“ bezeichnet.

Character := "'" (Letter | Digit | Symbol) "'"

Beispiele:

```
'c'
'9'
'$'
```

Mit einem + kann man Zeichen zusammenhängen. Zusammengehängte Zeichen ergeben eine Zeichenkette, die im Folgenden als „string“ bezeichnet wird.

String := "" {Letter | Digit | Symbol} ""

Eine Zeichenkette kann man auch direkt angeben, indem man sie zwischen zwei Anführungszeichen setzt " ". Auch Zeichenketten lassen sich mit einem + zusammenhängen. Genau genommen lassen sich Zeichen mit Zeichenketten, Zahlen und Wahrheitswerten durch + zu Zeichenketten zusammenhängen.

Beispiele:

```
'c'+9+'$'            => "c9$"
"Hallo"+' '+“Welt“  => “Hallo Welt“
```

Zeichen und Zeichenketten zählen sowie Zahlen zu den grundlegenden Werten in Chill.

Factor := "(" Sum ")" | Integer | Number | Character | String

3.1.1.4. Wahrheitswerte

Ein Wahrheitswert kann entweder wahr, „true“ oder falsch, „false“ sein. Wahrheitswerte werden im folgenden als „boolean“ bezeichnet. Wahrheitswerte kann man in Chill durch die Eingabe der Wörter *true*, oder *false angeben.*

Boolean := "true" | "false"

Wahrheitswerte zählen zu den grundlegenden Werten.

Factor := "(" Expr ")" | Integer | Number | Character | String | Boolean

Ein Vergleich zweier Werte ergibt einen Wahrheitswert. Es gibt in Chill mehrere Möglichkeiten zwei Werte zu vergleichen.

A == B	ist A gleich B?
A != B	ist A ungleich B?
A <= B	ist A kleiner oder gleich B?
A >= B	ist A größer oder gleich B?
A < B	ist A kleiner als B?
A > B	ist A größer als B?

```
Comparison    := Sum {("==" | "!=" | "<=" | ">=" | "<" | ">" | "in") Sum}
```

Der Vergleich mit "in" stellt einen Spezialfall dar, den ich später erläutern werde.

Beispiele:

```
0==0           => true
'a'!='b'       => true
0<=0           => true
0>=0           => true
0<0            => false
0>0            => false
true==false    => false
```

Will man wissen, ob ein Wahrheitswert und ein zweiter Wahrheitswert gilt, kann man dies in Chill mit einer & Verknüpfung berechnen.

```
And            := Comparison {"&" Comparison}
```

Eine Verknüpfung mit & ist nur dann wahr, wenn beide Werte wahr sind.

Beispiele:

```
true & true    => true
true & false   => false
false & true   => false
false & false  => false
```

Vergleiche gehen dabei vor einer Verknüpfung mit &. Und eine Verknüpfung mit & geht vor einer Verknüpfung mit | . Mit | kann man testen, ob ein Wahrheitswert oder ein zweiter Wahrheitswert gilt.

```
Expr           := And {"|" And}
```

Eine Verknüpfung mit | ist dann wahr, wenn einer der beiden Wahrheitswerte wahr ist.

Beispiele:

```
true | true    => true
true | false   => true
false | true   => true
false | false  => false
```

Wahrheitswerte können durch Voranstellen des Zeichens ! umgekehrt werden.

Factor := "!" Expr | "(" Expr ")" | Integer | Number | Character | String | Boolean

Beispiele:

```
!true          => false
!false         => true
```

3.1.1.5. Variablen und Namen

Variablen in Chill kann man sich als Platzhalter für Werte vorstellen. Eine Variable hat einen Namen. Die Namen in Chill müssen mit einem Buchstaben beginnen, gefolgt von beliebig vielen Buchstaben und Zahlen.

Ident := Letter {Letter | Digit}

Groß- und Kleinbuchstaben werden in Chill unterschieden, was soviel heisst, dass der Name a und der Name A zwei unterschiedliche Namen in Chill sind.
Um eine Zahl, einen Wahrheitswert, ein Zeichen oder eine Zeichenkette in einer Variable zu speichern, muß man diesen Wert der Variable zuweisen.

Assignment := Ident "=" Expr

Beispiele:

```
a=0            => erzeugt eine Variable a mit dem Wert 0
s=“Hallo“      => erzeugt eine Variable mit dem Wert“Hallo“
b=true         => erzeugt eine Variable mit dem Wert true
```

Wenn noch keine Variable dieses Namens existiert, wird eine neue Variable erzeugt. Diese wird dann im unteren Bereich, Variables, angezeigt. Existiert bereits eine Variable des Namens wird der Wert der Variable geändert.
Auf existierende Variablen kann durch Angabe des Namens „zugegriffen“ werden.

Beispiele:

```
a              =>   0
a+1            =>   1
b=!b           =>   ok
b              =>   false
c              =>   Error : Identifier c is not declared
```

3.1.1.6. Weitere Möglichkeiten

Viele der Möglichkeiten, die in Kapitel 3.1.2 vorgestellt werden, stehen, je nach Konfiguration, bei der interaktiven Verwendung von Chill zur Verfügung. Ein Beispiel ist, dass es auch möglich ist, mehrere Befehle hintereinander, durch das Symbol ; getrennt, in eine Zeile zu schreiben. Diese Befehle werden dann hintereinander ausgeführt. Als Ausgabe erscheint dann das Ergebnis des letzten Befehls.

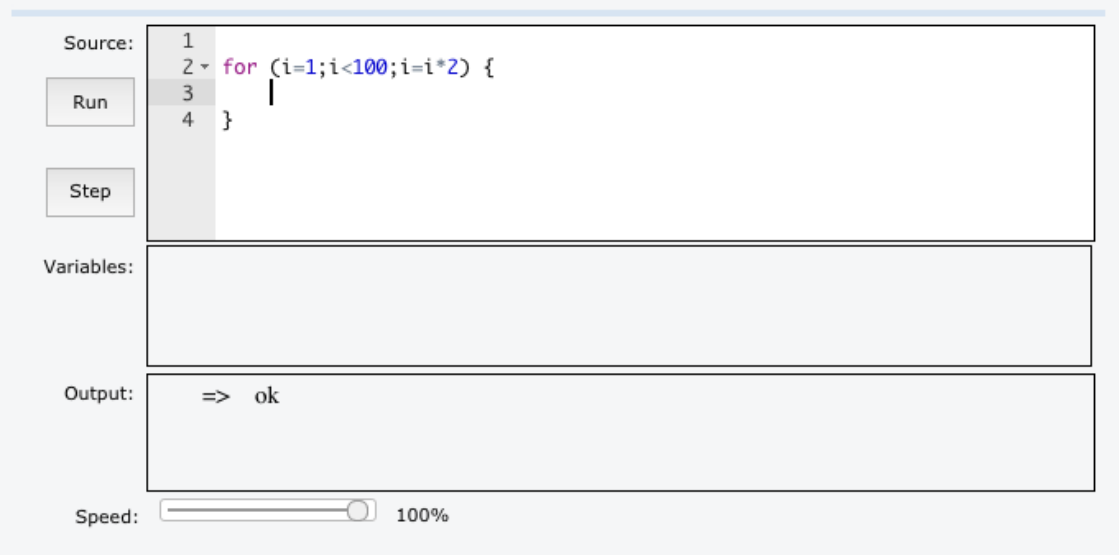
3.1.2. Programme in Chill

Ein Programm in Chill stellt eine Abfolge von Befehlen dar.

Program := {Statement}

Für die Eingabe von Programmen bietet Chill eine Oberfläche.

Befehle werden hintereinander ausgeführt. Damit ein Befehl vom anderen unterschieden werden kann, muß am Ende eines Befehles ein ; stehen.



Statement := (Expr | Assignment | Return) ";"

Mit Run werden alle Befehle ausgeführt, mit Step jeweils ein Befehl nach dem anderen. Chill zeigt dabei die Variablen, die gerade verwendet werden, an.
Der Befehl return beendet die Ausführung des Programmes an der Stelle, an der er steht. Wird ein Wert hinter return angegeben, wird dieser als Ergebnis zurückgeliefert.

Return := "return" [Expr]

Beispiele:

```
return 1;      =>   1
b=true;
return b;
b=false;  =>      true
```

3.1.2.1. Kommentare

Kommentare dienen dazu, um Anmerkungen in einem Programm einzufügen zu können, die dem Verständnis des Programmes dienen, aber die Übersetzung und Ausführung nicht beeinflussen.
Um Kommentare in Programme zu schreiben gibt es in Chill 2 Varianten. Steht in einer Zeile // werden alle Zeichen bis zum Zeilenende als Anmerkung verstanden und nicht übersetzt.

Beispiel:

```
a=a+1;          // Addiere 1 zu a
```

Die zweite Variante besteht darin einen Kommentar zwischen /* ***/** zu schreiben. Alle Zeichen zwischen /* und */ werden dabei ignoriert. Solche Kommentare können in Chill geschachtelt werden, also es kann innerhalb eines Kommentars ein weiterer stehen. Einzig die Zahl der /* und */ müssen übereinstimmen.

Beispiel:

```
/*Dies ist ein Kommentar
/*Und dies ein Kommentar in einem Kommentar*/
*/
```

3.1.2.2. Standard Datentypen

Bisher habe ich gezeigt, wie Ganze- und Kommazahlen, wie Zeichen- und Zeichenketten und wie Wahrheitswerte in Chill angegeben werden können. Diese unterschiedlichen Arten von Werten werden durch die in Chill vorhandenen Datentypen repräsentiert.

boolean	…	Wahrheitswerte
integer	…	Ganze Zahlen
number	…	Kommazahlen
character	…	Einzelne Zeichen
string	…	Zeichenketten

3.1.2.3. Variablen Deklarationen

Außerdem habe ich gezeigt, wie Variablen durch Zuweisungen erzeugt werden können. Für die interaktive Verwendung von Chill ist diese Art Variablen zu erzeugen zwar praktisch, allerdings hat sie auch Nachteile.
Ein Nachteil ist, dass diese Art in Programmen unübersichtlich sein kann, da nicht immer auf den ersten Blick ersichtlich ist, ob eine Variable bei einer Zuweisung erzeugt oder verändert wird.
Ein weiterer Aspekt ist, dass solche Variablen in Chill keinen bestimmten Typ haben, der ihnen zugewiesen werden kann, sondern sie können Werte jedes in Chill unterstützten

Typ enthalten. Das ist zwar auf der einen Seite praktisch, auf der anderen Seite macht es eine exakte Prüfung, ob eine Variable einen bestimmten Typ enthalten darf oder nicht, sehr aufwendig.
Aus diesen und weiteren Gründen wird in Chill eigentlich zwischen der Erzeugung von Variablen und Zuweisungen von Werten an Variablen unterschieden. Ein Programm besteht dann in der Folge aus der Erzeugung, Deklaration, von Variablen und Befehlen.

Program := {VarDecl | Statement}

Variablen müssen in Chill erst erzeugt worden sein, bevor sie verwendet werden können.
Für die Erzeugung von Variablen in Chill gibt es 4 Varianten. Die Entscheidung, welche Möglichkeit bei einer bestimmten Aufgabe zum Einsatz kommt, obliegt dem Lehrer der die Aufgabe erstellt.

3.1.2.3.a Untyped

Dies ist die bereits vorgestellte Variante. Variablen werden per Zuweisung deklariert. Die Variablen können jeden beliebigen Typ beinhalten.

VarDecl := Ident "=" Expr ";"

Beispiele:

```
a=17;           //Erzeugt eine Variable a mit dem Wert 17
b=a+4;          //Erzeugt eine Variable b mit dem Wert 21
a="Hallo Welt";//Ändert den Wert der Variable a auf "Hallo..
```

3.1.2.3.b Optional Declaration

Variablen können explizit erzeugt werden, müssen es aber nicht. Variablen können „nur“ für einen bestimmten Typ erzeugt werden. Der Programmierer entscheidet.
Außerdem können Konstanten erzeugt werden, also Variablen, denen nur ein einziges Mal ein Wert zugewiesen werden kann.

VarDecl := (Ident "=" Expr | [Ident] ("variable" | "constant") Ident ["=" Expr])";"

Beispiele:

```
integer variable a=17; //Erzeugt eine Variable a mit dem
                       // Wert 17
constant b=a+4;        //Erzeugt eine Konstante b mit dem
```

```
                        // Wert 21
a=“Hallo Welt“;         //Fehler, da a nur Ganze Zahlen
                        // beinhalten darf
b=9;                    //Fehler, da b nicht verändert werden
                       // darf
c=true;                //Erzeugt c mit dem Wert true
```

3.1.2.3.c Declaration, optional type

Variablen müssen deklariert werden, eine Erzeugung durch Zuweisung ist nicht mehr zulässig.

Es muß kein Typ für eine Variable angegeben werden.

VarDecl := [Ident] ("variable" | "constant") Ident ["=" Expr] ";"

Beispiele:

```
integer variable a=17;//Erzeugt eine Variable a mit dem Wert
                      // 17
constant b=a+4;       //Erzeugt eine Konstante b mit dem
                      //Wert 21
a=“Hallo Welt“;       //Fehler, da a nur Ganze Zahlen
                      //beinhalten darf
b=9;                  //Fehler, da b nicht verändert werden
                      //darf
c=true;               //Ergibt einen Fehler, da c nicht
                      // deklariert wurde.
```

3.1.2.3.d Strict Typing

Variablen müssen deklariert werden und es muß ein Typ angegeben werden.

VarDecl := Ident ("variable" | "constant") Ident ["=" Expr] ";"

Beispiele:

```
integer variable a=17;//Erzeugt eine Variable a mit dem Wert
                      // 17
constant b=a+4;       //Fehler, da kein Typ angegeben wurde
```

3.1.2.4. Kontrollstrukturen

Im folgenden werde ich Strukturen vorstellen, die in Chill verwendet werden können um, den Programmablauf zu steuern.

Statement := If | For | Do | While | Switch | (Expr | Assignment | Return) ";"

3.1.2.4.a Blöcke

Um mehrere Kommandos zusammenzufassen können diese in Chill zu Blöcken zusammengefasst werden. Diese Blöcke werden durch geschwungene Klammern begrenzt. Innerhalb der Klammern können Variablendeklarationen und Anweisungen stehen. Variablen, die in einem Block deklariert werden, sind nur innerhalb dieses Blocks sichtbar. Variablen, die außerhalb eines Blockes deklariert wurden, sind innerhalb des Blocks sichtbar. Sollte innerhalb eines Blocks eine Variable deklariert werden, die einen identischen Namen hat, wie eine Variable außerhalb des Blocks, ist innerhalb des Blocks die Variable die innerhalb deklariert wurde, sichtbar.

CodeBlock := "{" {VarDecl | Statement} "}"

3.1.2.4.b if

Eine einfache Verzweigung stellt die if Anweisung dar. Wenn eine Bedingung gilt, dann wird der dahinter folgende Codeblock ausgeführt.

If := "if" "(" Expr ")" CodeBlock ["else" CodeBlock]

Optional kann nach einem „if – Zweig“ eine else Anweisung, gefolgt von einem Codeblock stehen. Der „else – Block“ wird dann ausgeführt, wenn die Bedingung der if – Anweisung nicht erfüllt ist.

Beispiel:

```
if (a==0) {
     a=1;
} else {
     a=0;
}
```

In vielen Programmiersprachen kann auch ein einzelnes Statement nach einer if – Anweisung stehen. In Chill ist dies nicht möglich. Dadurch gibt es in Chill aber auch kein „dangling else“.

3.1.2.4.c switch

Eine switch Anweisung führt unterschiedliche Blöcke, je nach dem Wert, den eine angegebene Variable gerade hat, aus. In Chill muss als letzte Möglichkeit in einer Switch Anweisung immer ein „default“ Block angegeben werden. Dieser wird ausgeführt, wenn kein anderer Ausführungszweig gefunden wird.

Switch := "switch" "(" Ident ")" "{" {Option} "default" ":" CodeBlock " "}"

Die unterschiedlichen Möglichkeiten, die der Wert annehmen kann werden in case Anweisungen angegeben. Anders als in vielen Programmiersprachen müssen Blöcke nach einem Case verwendet werden, um Konsistenz mit anderen in Chill verwendeten Kontrollstrukturen zu haben.

Option := "case" Expr {"," Expr} ":" CodeBlock

Beispiel:

```
switch (a) {
     case 0: {/* Do something */}
     case 1,3,5,7,9:{/* Do something else */}
     case  2,4,6,8:{/* Do another thing */}
     default: {/* Do something */}
}
```

3.1.2.4.d while

Bei einer while Schleife werden die Anweisungen innerhalb des darauffolgenden Blocks solange ausgeführt, solange die Bedingung, die hinter while angegeben wird, erfüllt ist. Die Bedingung wird vor jedem Betreten der Schleife geprüft.

While := "while" "(" Expr ")" CodeBlock

Beispiel:

```
a=398; b=256;
while (b!=0) {
     h=a%b;
     a=b;
     b=h;
}
return a;
```

3.1.2.4.e do – while

Auch bei der do – while Schleife werden die Anweisungen des darin enthaltenen Blocks solange ausgeführt, solange die Bedingung, die hinter while angegeben wird, erfüllt ist. Die Bedingung wird nach der Ausführung des Blocks geprüft. Daher wird eine do – while Schleife mindesten einmal durchlaufen.

Do := "do" CodeBlock "while" "(" Expr ")" ";"

Beispiel:

```
do {
     a=a*2;
} while (a<256);
```

3.1.2.4.f for Schleife

In Chill gibt es 2 Varianten der for – Schleife. Eine Variante steht nur in Verbindung mit array‘s zur Verfügung und wird daher in Kapitel 3.1.2.5.a vorgestellt.
Bei einer for – Schleife in Chill werden 3 Argumente angegeben, die durch ; getrennt werden. Das erste Argument dient der Initialisierung. Es muss eine Zuweisung enthalten.
Das zweite Argument ist eine Bedingung, die geprüft wird, bevor die Schleife ausgeführt wird. Solange diese Bedingung wahr ist, solange wird der folgende Block ausgeführt.
Das dritte Argument wird nach dem Block ausgeführt. Es wird verwendet, um z.B. eine Zählvariable zu erhöhen. Dieses Argument muß in Chill wieder eine Zuweisung enthalten.

For := "for" ("(" Assignment ";" Expr ";" Assignment ")") CodeBlock

Beispiel:

```
for (i=0;i<100;i=i+1) {
     // Do something
}
```

Würde man obige for – Schleife als while Schleife ausführen, würde sie wie folgt aussehen.

```
I=0;

while (i<100) {
     // Do something
     i=i+1;
}
```

3.1.2.5. Funktionen

Anweisungsfolgen, die mehrmals verwendet werden, können als Funktion ausgeführt werden. An eine Funktion können Werte übergeben werden. Diese werden an Parameter zugewiesen, das sind Variablen, die im Funktionskopf angegeben werden und nur innerhalb der Funktion sichtbar sind.
Außerdem kann eine Funktion einen Wert zurück liefern. Dazu dient das return – Statement, das die Funktion beendet, sobald es aufgerufen wird und den hinter ihm angegeben Wert zurückliefert.
Immer dann, wenn diese Anweisungsfolge ausgeführt werden soll, kann sie dann durch Angabe des Funktionsnamen aufgerufen werden.
Beispiel:

```
function sum(a,b) {
     return (a+b);
}

s=sum(3,4);
```

Je nach Art der Typisierung darf kein, kann oder muss ein Typ als Rückgabewert und für jeden Parameter angegeben werden. Der Typ des Rückgabewertes wird vor dem Schüsselwort function angegeben. Der Typ eines Parameter vor dem Namen des entsprechenden Parameters.
Für „Untyped“ gilt folgende Funktionsdefinition:

Function := "function" Ident "(" Ident {"," Ident }] ")" CodeBlock

Für „Optional Declaration“ und „Declaration, optional type“ gilt folgende Funktionsdefinition:

```
Function        := [Ident] "function" Ident "(" [[Ident] Ident {"," [Ident] Ident }] ")"
                   CodeBlock
```

Für „Strict Typing“ gilt folgende Funktionsdefinition:

```
Function        := [Ident] "function" Ident "(" [Ident Ident {"," Ident Ident }] ")"
                   CodeBlock
```

Beispiel:

```
integer function sum(integer a,integer b) {
     return (a+b);
 }

 s=sum(3,4);
```

Funktionen können an beliebiger Position des Programmcodes deklariert werden.

```
Program         := {VarDecl | Function | Statement}
```

3.1.2.6. Benutzerdefinierte Typen

Mit benutzerdefinierten Typen können eigene Datentypen erzeugt werden. Diese Typen können erst ab der Typisierungsart „Optional Declaration“ verwendet werden, da sonst keine Variablen des entsprechenden Typen angelegt werden können.
Benutzerdefinierte Typen dürfen an beliebiger Position im Programm stehen, müssen aber deklariert werden, bevor sie benutzt werden können.

```
Program  := {Class | StructDecl | ArrayDecl | VarDecl | Function | Statement}
```

3.1.2.6.a array

Eine Array in Chill erlaubt es mehrere Werte in einer Variablen zu speichern. Auf die einzelnen Werte kann durch Angabe eines Index, der zwischen [] steht, zugegriffen werden.
Um array‘s in Chill verwenden zu können muss davor ein Typ für diese Art von Array erzeugt werden.

ArrayDecl := "array" Ident "["Ident"]" "contains" Ident ";"

Beispiel:

```
array a [];
a variable h;

h[“de“]=“Hallo“;
h[“en“]=“Hello“;
```

Je nach Art der Typisierung können oder müssen Typen für den Index und die enthaltenen Werte angegeben werden. Der Typ des Index wird dabei zwischen [] angegeben, der Typ für die enthaltenen Werte folgt auf das Wort contains.

Beispiel:

```
array a [string] contains string;

a variable h;
h[“de“]=“Hallo“;
h[“en“]=“Hello“;
```

Array‘s in Chill sind selbstwachsend. Sobald einem Index, der noch nicht existiert, ein Wert zugewiesen wird, wird dieser automatisch erzeugt. Um die Länge eines Array‘s in Chill auslesen zu können, besitzt jedes Array ein length Attribut, das die Anzahl der im Array beinhalteten Werte anzeigt.

Beispiel:

```
return h.length;              => 2
```

Array‘s in Chill sind, im Gegensatz Array‘s in vielen anderen Programmiersprachen, wertbasiert. Wird ein Variable einer anderen zugewiesen, so werden alle Werte kopiert. Wird dann ein Wert eines der beiden Array‘s verändert, beeinflusst es das andere Array nicht.

Beispiel:

```
array a [string] contains string;

a variable h,x;

h[“de“]=“Hallo“;
h[“en“]=“Hello“;
x=h;
x[“cz“]=“Ahoj“;     // h==[“Hallo“,“Hello“],
                    // x==[“Hallo“,“Hello“,“Ahoj“]
```

Um Operationen auf alle in einem Array enthaltenen Werte ausführen zu können existiert in Chill eine spezielle Variante der for – Schleife. Ihr wird ein Argument übergeben, dass mit dem Wort each beginnt.

For := "for" "(" "each" Ident "in" Ident ")" CodeBlock

Der Anweisungs – Block dieser Schleife wird für jeden Index eines Array‘s ausgeführt. Dabei wird die Variable, der der Indexwert zugewiesen wird, hinter each angegeben. Das Array hinter in.

Beispiel:

```
for (each index in arr) {
     arr[index]=arr[index]*2;
}
```

3.1.2.6.b structure

Mithilfe einer Structure können mehrere Variablen und Konstanten in einer Variablen gespeichert werden. Um auf die in einer structure – Variablen gespeicherten Werte zuzugreifen wird hinter dem Variablennamen, durch einen . getrennt, der Name des Elements angegeben.

StructDecl := "structure" Ident "{" VarDecl {VarDecl} "}"

Beispiel:

```
structure name {
     variable vorname;
     variable nachname;
}
name variable n;
n.vorname=“Max“;
n.nachname=“Mustermann“;
```

Auch Strukturen in Chill sind wertbasiert, jede Variable einer structure enthält jedes Element einer structure. Wird eine Variable einer anderen zugewiesen wird jeder Wert kopiert.

3.1.2.6.c class

Mit class können Klassen in Chill erzeugt werden. Klassen in Chill können Variablen bzw. Konstanten und Funktionen beinhalten.

Class := "class" Ident "{" {VarDecl | Function} "}"

Beispiel:

```
class c {
     variable x;
     function f() {
          // Do something
     }
}
```

Klassen dienen in Chill als Schablonen für Objekte. Mit dem Schlüsselwort new kann ein Objekt erzeugt werden, das alle Eigenschaften, also alle Variablen und Funktionen, einer Klasse hat.

Beispiel:

```
o=new c();

o.x = 100;
return o.f();
```

Um Objekte initialisieren zu können gibt es in Chill Konstruktoren. Ein Konstruktor ist in Chill eine Funktion, die aufgerufen wird, wenn ein Objekt mit new erzeugt wird. Eine Klasse in Chill besitzt automatisch einen Konstruktor, egal, ob er explizit angegeben wird oder nicht. Wird ein Konstruktor angegeben, dann muss dies am Beginn einer Klassendeklaration geschehen.

```
Class := "class" "{" ["constructor" "(" [Ident] Ident ")" CodeBlock]
         {VarDecl | Function} "}"
```

Beispiel:

```
class c {
     constructor (a) { x=a;}

     variable x;

     function f() { // Do something }
}

o=new c(100);
```

Objekte sind im Gegensatz zu allen anderen Typen in Chill nicht wert-, sondern referenzbasiert. Eine Variable der ein Objekt zugewiesen wird enthält nicht das Objekt, sondern verweist auf das Objekt. Es können also mehrere Variablen auf das selbe Objekt verweisen.

Beispiel:

```
a=new c(100);// Erzeugt ein neues Objekt, auf das a verweist
b=a;         // a und b verweisen auf das Selbe Objekt
b.x=200;     // Auch a.x ist 200, denn x wurde im Objekt
             // geändert
```

Eine Klasse in Chill kann von einer andere Klasse erben. Diese Klasse besitzt alle Eigenschaften der Basisklasse, sie besitzt zumindest dieselben Variablen, Konstanten und Funktionen. Soll eine Klasse eine andere erweitern wird das durch das Wort extends angegeben.

```
Class := "class" Ident ["extends" Ident] "{"
            ["constructor" "(" Ident Ident ")" CodeBlock]
            {VarDecl | Function}
         "}"
```

Beispiel:

```
class d extends c {
     function g () {
          return x;
     }
}

o=new d(100);       // d hat den Konstruktor von c geerbt.
o.f();              // d hat auch die Funktion f von c
                    // geerbt.
return o.g();       // d hat zusätzlich zu c die funktion g
```

In Chill können Eigenschaften überschrieben, aber nicht überladen werden. Werden die selben Namen in einer abgeleiteten Klasse verwendet, die in einer Basisklasse verwendet wurden, dann müssen sie identisch deklariert werden.

Beispiele:

```
class d extends c {
     function f (a) {//Fehler, da in c eine Funktion f
                     //deklariert
          return a; // wurde, der kein Parameter übergeben
                    // wird
     }
  }

class d extends c {
     function f () {//Identische Deklaration von f()
                    //in d und c
          return x; //Anweisungen in d haben sich verändert.
     }
}
```

Um innerhalb eines Objektes auf Werte, die in der Klasse deklariert wurden, zuzugreifen existert eine spezielle Variable in jedem Objekt, die auf dieses Objekt verweist. Diese Variable heisst this.

Beispiel:

```
class c {
     constructor (x) {
          this.x=x;
     }

     variable x;

     function f() {
          // Do something
     }
}
```

Um auf ein Element zugreifen zu können, das in einer Basisklasse deklariert wurde, existiert in jedem Objekt eine Variable super, die auf ein Objekt mit allen Eigenschaften der Basisklasse verweist.

Beispiel:

```
class d extends c {

     function f() {
          a=super.f();
          // Do something
     }
}
```

3.1.2.7. Javascript

In Chill kann javascript Code eingebunden werden. Das Einbinden von Javascript erlaubt es die Funktionalität von Chill zu erweitern. Damit das Laufzeitsystem von Chill nicht beeinflusst werden kann, wird der Code „gekapselt“ ausgeführt. Sollen Variablen, die in Chill verwendet werden auch innerhalb des javascript zur Verfügung stehen, müssen sie wie Parameter angegeben werden. Änderungen, die im Javascript - Code

an Variablen vorgenommen werden bleiben auch in Chill erhalten. Der Javascript code steht zwischen # und #end.

JavaScript := "javascript" "(" [[Ident] Ident {"," [Ident] Ident }] ")"

"#" {Letter | Digit | Symbol} "#end"

Beispiel:

```
number function abs(number x){//Returns the abs of x
     number variable ret=0;
     javascript(x,ret) # ret=Math.abs(x); #end
     return ret;
}
```

Statement:= If | For | Do | While | Switch | Javascript | (Expr | Assignment | Return) ";"

Array‘s, Strukturen und Objekte aus Chill können nicht im Javascript Code verwendet werden.

3.1.3. Zusammenfassung – EBNF

Program := {Class | StructDecl | ArrayDecl | VarDecl | Function | Statement}

Class := "class" Ident ["extends" Ident] "{" ["constructor" "(" Ident Ident ")" CodeBlock] {VarDecl | Function} "}"
StructDecl := "structure" Ident "{" VarDecl {VarDecl} "}"
ArrayDecl := "array" Ident "["Ident"]" "contains" Ident ";"

VarDecl := (Ident "=" Expr | [Ident] ("variable" | "constant") Ident ["=" Expr]) [";"]

Function := [Ident] "function" Ident "(" [[Ident] Ident {"," [Ident] Ident }] ")" CodeBlock
CodeBlock := "{" {VarDecl | Statement} "}"

Statement := If | For | Do | While | Switch | Javascript | (Expr | Assignment | Return) [";"]

If := "if" "(" Expr ")" CodeBlock ["else" CodeBlock]

```
For := "for" ("(" Assignment ";" Expr ";" Assignment ")") CodeBlock
Do := "do" CodeBlock "while" "(" Expr ")" ";"
While:= "while" "(" Expr ")" CodeBlock
Switch := "switch" "(" Ident ")" "{" {Option} "default" ":" CodeBlock " "}"
Option := "case" Expr {"," Expr} ":" CodeBlock

JavaScript := "javascript" "(" [[Ident] Ident {"," [Ident] Ident }] ")" "#"Letter | Digit | Symbol}
                "#end"

Expr := And {"|" And}
And := Comparison {"&" Comparison}
Comparison := Sum {("==" | "!=" | "<=" | ">=" | "<" | ">" | "in") Sum}
Sum := ["+" | "-"] Product {("+" | "-") Product}
Product := Factor {("*" | "/" | "%") Factor}

Factor := IdentSequence | "new" Ident "(" [Expr {"," Expr}] ")" | "!" Expr | "(" Expr ")"
          | Boolean | Integer | Number | Character | String

IdentSequence := [("this" | "super") "."]  ("constructor" "(" [Expr {"," Expr}] ")"
                    | Ident {{ArrayIndex} | "." Ident } ["(" [Expr {"," Expr}] ")"])

Boolean := "true" | "false"
Integer := Digit {Digit}
Number := Digit {Digit} "." Digit {Digit}
Character := "'" (Letter | Digit | Symbol) "'"
String := "" {Letter | Digit | Symbol} ""

Digit := "0" ... "9"
Letter := "a"  ... "z" | "A" ... "Z"
Symbol :=  "^"  ...  "*"
```

3.2. Die Verwaltungsumgebung

3.2.1. Vorraussetzungen

Für den Server wird ein Web-Server mit PHP ab Version 5 und MySQL-Datenbank benötig. Der Client ist mit jedem aktuellen Webbrowser verwendbar,der HTML5 unterstützt. Javascript muss im Browser aktiviert sein.

3.2.2. Installation

Die Implementierung wird in das documentRoot – Verzeichnis der Web Server kopiert. Danach ruft man

http://<serveradresse>/setup.php

im Webbrowser auf.

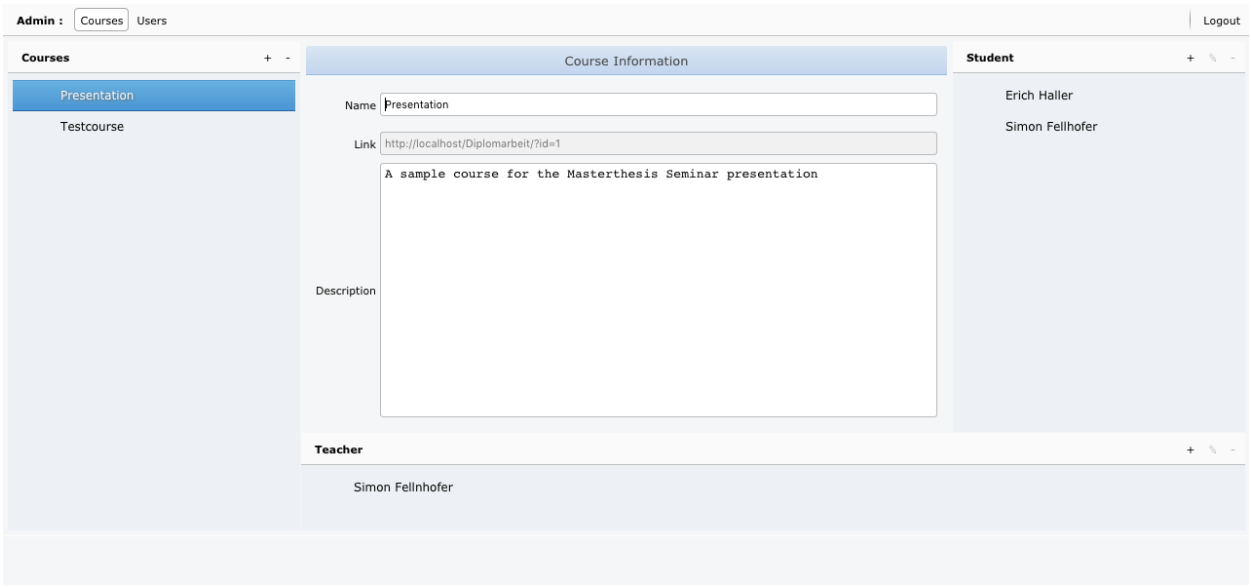
Dann wird man nach der Adresse, einem Benutzer und dessen Passwort für den MySQL – Server gefragt. Der Benutzer benötigt die Rechte, um Datenbanken anlegen zu können.
In der Folge wird die für Chill erforderliche Datenbankstruktur aufgebaut.
Zum Abschluss wird man noch nach einem Benutzernamen und einem Passwort für den Chill - Administrator gefragt.

Nach erfolgreicher Installation ruft man im Webbrowser

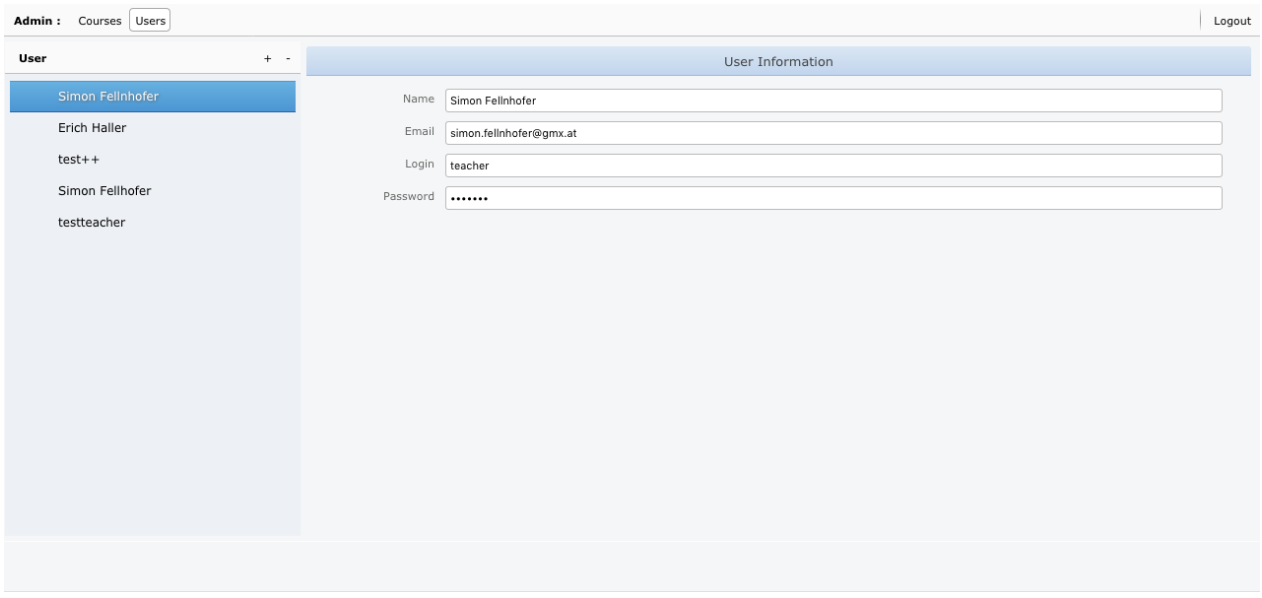
http://<serveradresse>/

auf und kann sich als Administrator anmelden.

3.2.3. Administration



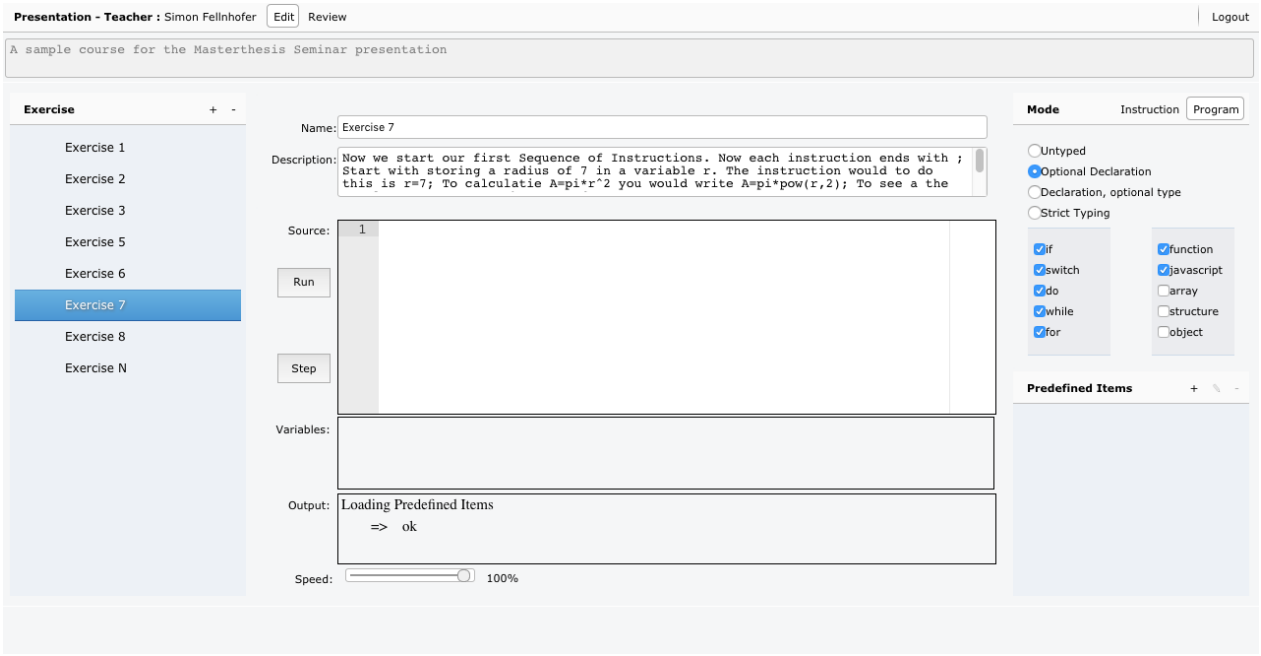
Der Administrator kann Kurse anlegen und löschen. Er schreibt die Kursbeschreibung und legt den Lehrer für den Kurs fest. Der Administrator kann Schüler zu einem Kurs hinzufügen und aus einem Kurs löschen.
Außerdem kann der Administrator Benutzer verwalten. Er kann zu Chill Benutzer hinzufügen, die Daten ändern und Benutzer löschen.



3.2.4. Lehrer

3.2.4.1. Erstellen von Aufgaben

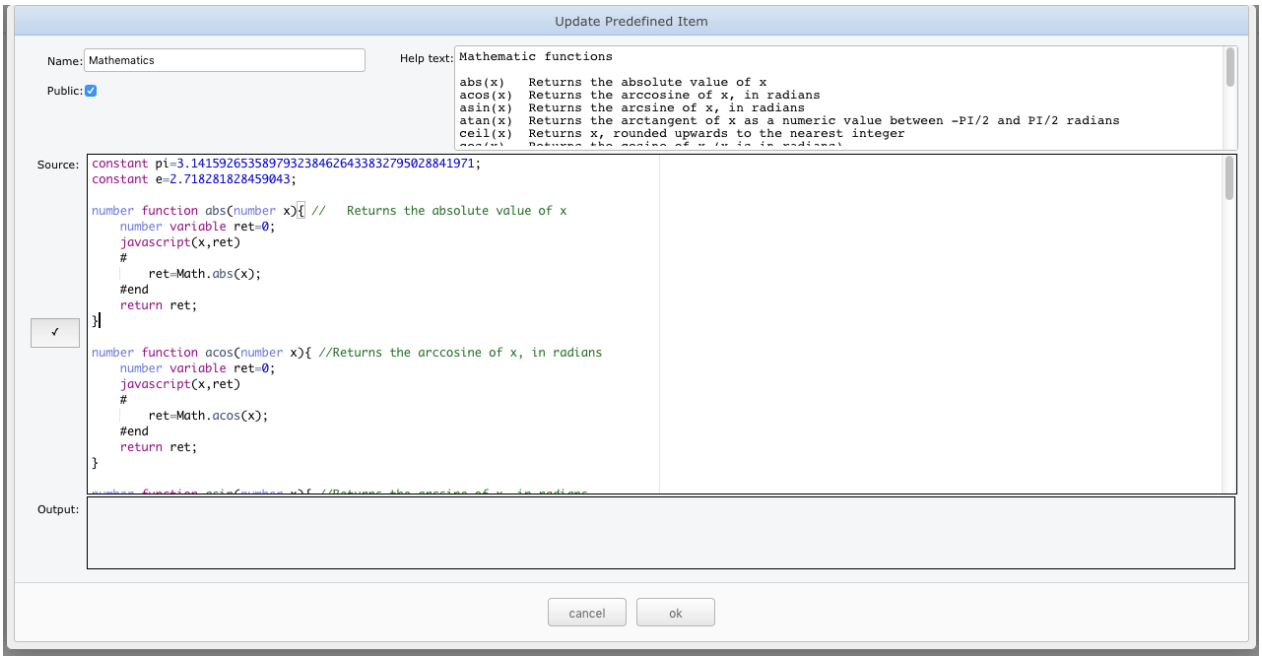
Der Lehrer kann Aufgaben für einen Kurs erstellen. Er legt den Namen für jede Aufgabe fest und beschreibt die Aufgabe. Außerdem kann er für jede Aufgabe die Spracheigenschaften von Chill bestimmen. Der Lehrer kann für die Aufgabe festlegen, ob sie im Interaktiven- oder im Programmiermodus gelöst werden soll. Er legt fest, welche Typisierungsart verwendet werden soll und kann alle Spracheigenschaften an- und ausschalten.



Er hat auch eine Programmierumgebung mit den eingestellten Eigenschaften zur Verfügung, um selbst mit der gestellten Aufgabe experimentieren zu können und eventuell eine Musterlösung zu erstellen.

3.2.4.1.a Predefined Items

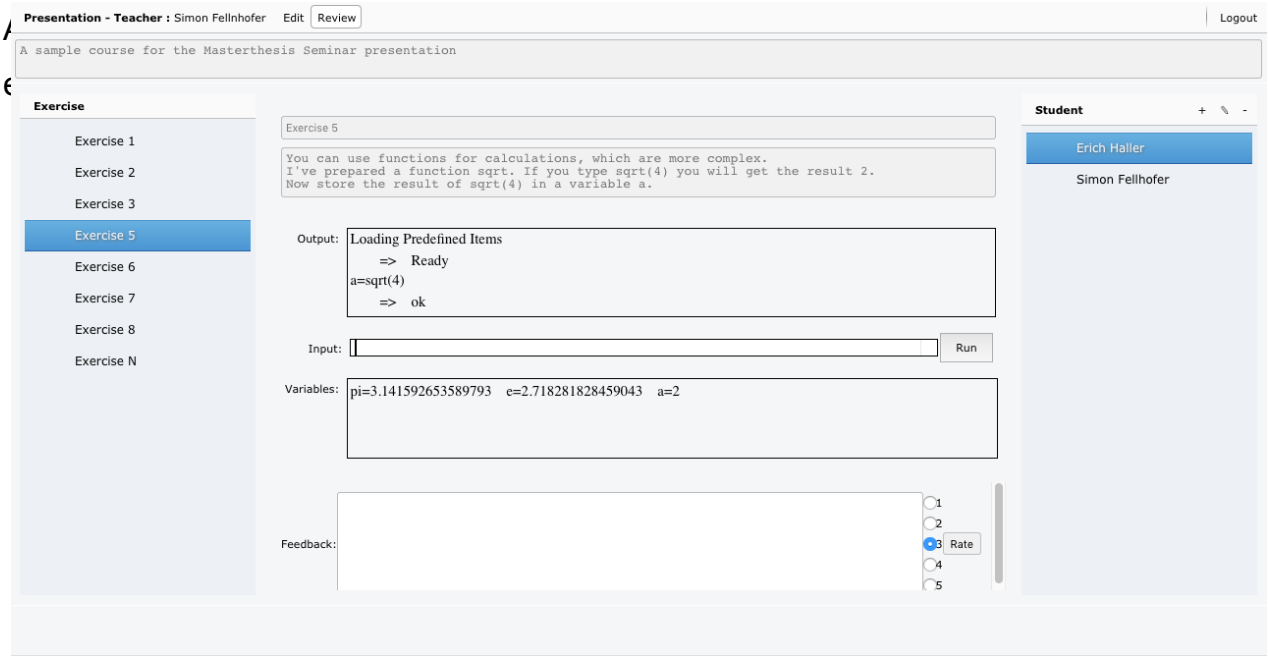
Ein Lehrer kann den Schülern vordefinierte Elemente, „predefined Items“, zur Verfügung stellen. Diese Elemente können genutzt werden, um zum Beispiel Typen, Variablen, Konstanten zu deklarieren oder Funktionen zu erstellen, die dann den Schülern für das Lösen der Aufgabe zur Verfügung stehen.



Für vordefinierte Elemente kann auch ein Hilfetext angegeben werden. Diese Elemente können auch bei anderen Aufgaben und Kursen verwendet werden.
Für vordefinierte Elemente steht der komplette Sprachumfang von Chill zur Verfügung, die eingestellten Limitierungen gelten für sie nicht. Wenn eine Aufgabe geladen wird, werden sie automatisch geladen und ausgeführt.

3.2.4.2. Korrektur

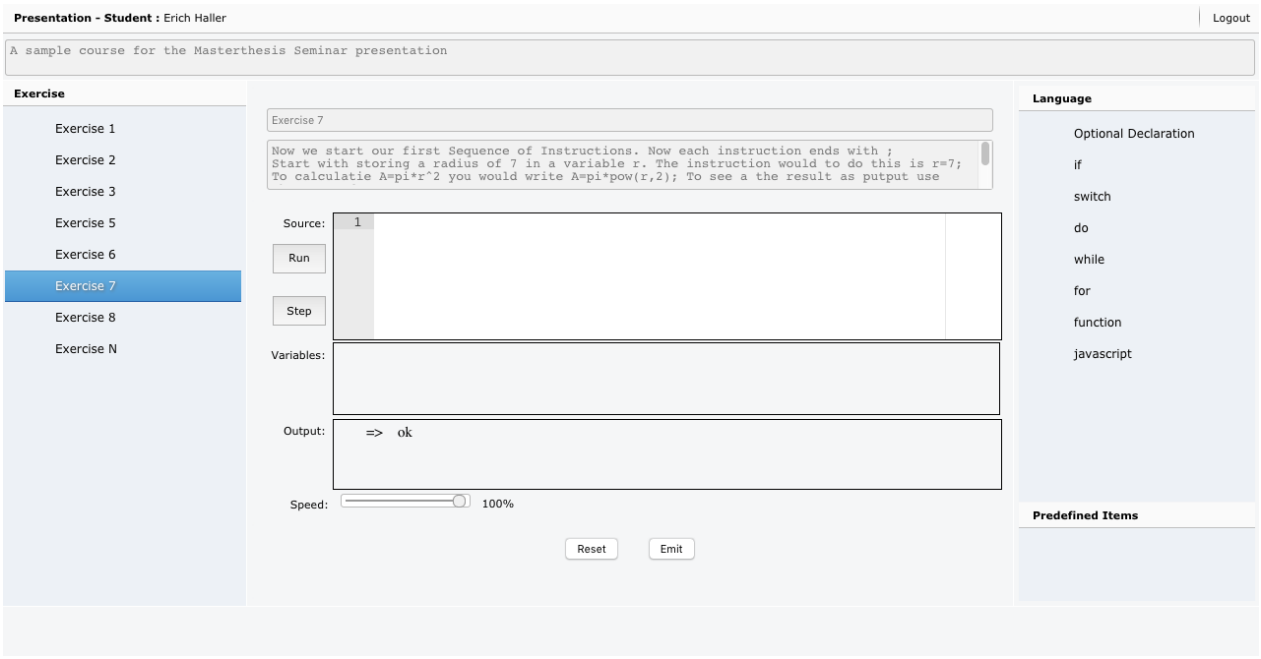
Für die Korrektur der abgegebenen Aufgaben steht dem Lehrer ein „Review“ - Modus zur Verfügung. In diesem Modus hat der Lehrer auch die Möglichkeit, Schüler zu einem Kurs hinzuzufügen, oder von einem Kurs zu löschen.



Er kann ein verbales Feedback erstellen und eine Note für die Lösung vergeben.

3.2.4.3. Schüler

Ein Schüler kann die einzelnen Aufgaben anwählen und eine Lösung erstellen.
Er kann seine Programmierumgebung auch jederzeit in ihren Ausgangszustand bringen, um eine „saubere“ Lösung erstellen zu können.

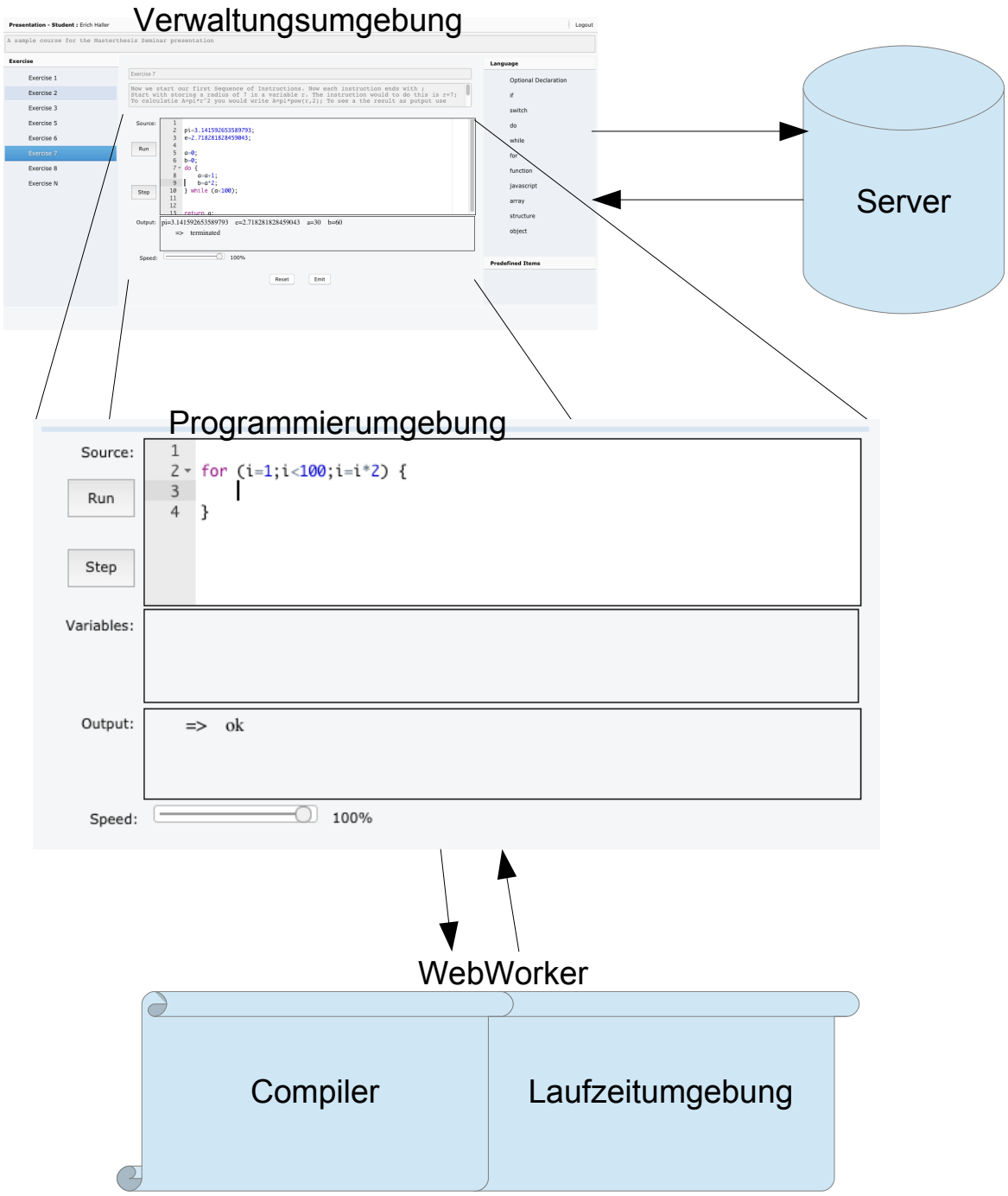


Zu den Sprachfunktionen stehen dem Schüler Hilfestellungen parat. Wenn er eine Aufgabe gelöst hat kann er sie abgeben. Die Aufgabe bleibt dann gesperrt, bis der Lehrer die Aufgabe bewertet hat. Dann sieht der Schüler sowohl das Feedback, als auch die Note. Er kann auch seine Aufgabe wieder bearbeiten und so durch das Feedback entstandene Erkenntnisse ausprobieren.

4. Implementierung

4.1. Struktur

Eine „Verwaltungsumgebung“ wird vom Server bereitgestellt. Sie kommuniziert mit dem Server und enthält die Programmierumgebung.
Die Programmierumgebung arbeitet mit einem WebWorker, der Compiler und Laufzeitumgebung enthält.



4.2. Server

Die Daten, die für die Funktion von Chill benötigt werden, werden in einer MySql Datenbank am Server gespeichert. Die Datenbank hat dabei folgende Struktur.

Datenbank Struktur

Tabelle	Felder	Beschreibung
users	Id … Eindeutige Benutzer Id user ... Benutzername pass ... Passwort name … Name des Benutzers email … Emailadresse role … „admin“ oder „user“	Beinhaltet alle Benutzer
courses	Id … Eindeutige Kurs Id name ... Kursname description … Kursbeschreibung	Beinhaltet alle Kurse
teachers	course … id des Kurses user … id des Benutzers	Ordnet Benutzer Kursen als Lehrer zu, mithilfe Ihrer Id‘s.
students	course … id des Kurses user … id des Benutzers	Ordnet Benutzer Kursen als Schüler ,mit Hilfe Ihrer Id‘s, zu.
exercises	id … Eindeutige Aufgaben Id course ...Id des Kurses, zu dem die Aufgabe gehört. name… Name der Aufgabe description… Beschreibung der Aufgabe mode … 'instruction','program' features …Eigentschaften der Sprache	Beinhaltet alle Aufgaben, die Zuordnung der Aufgabe zu dem jeweiligen Kurs, den jeweiligen Modus der Programmierumgebung und die Einstellungen der Sprache.
items	id … Eindeutige Id des Items name ... Name des Items course ...Id des Kurses, in dem das Item vorhanden ist.	Beinhaltet alle predefined Items

	public … Ist das Item öffentlich? code … Programmcode help ... Help Text	
submissions	vode … Programmcode exercise … Id der Aufgabe user … Id des Schülers time … Zeitpunkt der Abgabe	Beinhaltet alle Abgaben
ratings	feedback … Feedback zur Aufgabe grade … Note exercise … Id der Aufgabe user ... Id des Schülers time ... Zeitpunkt der Bewertung	Beinhalted alle Bewertungen

Tabelle 1: Datenbank Struktur

Der Zugriff auf die Datenbank erfolgt mithilfe von server.php. Diese Server Implementierung übersetzt folgende Befehle in SQL Kommandos, um auf die Datenbank zuzugreifen.

Server Befehle

Befehl	Destination	Schlüssel (wenn nicht weiter erklärt wird eine Numerische id erwartet)	Erklärung
get	user, course, item, exercise	id	Liefert einen Eintrag mit der Kennung id
get	predefindedItems	exercise	Liefert alle predefined Items einer gegeben Aufgabe
get	exercise	id user	Wenn zusätzlich eine Benutzer Kennung angegeben wird, werden außerdem die Felder

			submitted rated zurückgeliefert, die angeben ob die Lösung abgegeben und bewertet wurde.
get	submission, rating	user … id des Schülers exercise … id der Aufgabe	Liefert die Lösung oder Bewertung einer Lösung einer Aufgabe eines Benutzers.
add	exercise,course,user,item	-	Fügt einen Eintrag in eine Tabelle hinzu, liefert die id des Eintrags.
add	userToCourse	role … student | teacher user course	Fügt einen Benutzer zu einem Kurs hinzu. role gibt an, ob der Benutzer Schüler oder Lehrer in dem Kurs ist.
del	course,user,item,exercise	id	Löscht einen Eintrag und all seine Abhängigkeiten
del	student,teacher	user course	Löscht einen Benutzer aus einem Kurs
del	itemInExercise	Item exercise	Löscht ein predefined Item aus einer Aufgabe
update	exercise,course,user,item	id	Verändert einen Eintrag in einer Tabelle
list	user	course (optional)	Zählt alle Benutzer auf. Wird eine course Id

			angegeben, zählt es alle Benutzer außer den bereits im Kurs eingetragenen Benutzern auf.
list	Item	exercise (optional) course	Zählt alle predefined Items auf, die in dem Kurs definiert, oder öffentlich, aber nicht in der Aufgabe in Verwendung sind.
list	exercise	course	Zählt alle Aufgaben eines Kurses auf.
submit	-	user exercise	Gibt eine Lösung eines Benutzers für Aufgabe ab
rate	-	user exercise	Fügt die Bewertung einer Lösung eines Benutzers für Aufgabe hinzu

Tabelle 2: Server Befehle

Es werden Einzeleinträge, oder Array‘s zurückgeliefert. Sollte ein Fehler auftreten, wird ein Fehlercode und die entsprechende Fehlermeldung zurückgeliefert.

4.3. Verwaltungsumgebung

Für Administrator, Lehrer und Schüler existiert jeweils eine Verwaltungsoberfläche. Sie ist in w2ui erstellt, einer auf Jquery basierenden Javascript – Bibliothek, für die Entwicklung von HTML basierten Programmen. Im Unterschied zu vielen anderen Javascript – Bibliotheken setzt w2ui weniger auf optische Raffinessen, sondern stellt ein nützliches Set an Bedienelementen zur Verfügung.
Für die Erzeugung von UI – Elementen, die öfter verwendet werden stellt template.js die nötigen Funktionen zur Verfügung.
In der Datei ui.js sind Hilfsmittel, die für alle 3 Verwaltungsoberflächen benötigt werden, zusammengefasst. Neben Funktionen zum Öffnen und Schließen von Dialogen und

einem Array, das die unterschiedlichen Programmierumgebungen beinhaltet, bietet sie auch die Funktion sendCommand, zur Kommunikation mit dem Server.

```
sendCommand(command,destination,data, function(data){ … } );
```

Das erste Argument von sendCommand ist der Befehl, als String, der an den Server geschickt wird. Das zweite Argument ist ein String mit dem Namen der Tabelle, auf die der Befehl angewendet werden soll. Das dritte Argument sind die Daten, als Javascript Objekt übergeben, die an den Server übermittelt werden sollen. Die Daten werden im JSON-Format an den Server geschickt. Sollten Strings in den Daten stehen, werden diese Base64 kodiert. Das ist notwendig, da sonst Sonderzeichen bei der Übermittlung an den Server verlorengehen könnten.
Das letzte Argument von sendCommand ist eine Funktion, die ausgeführt wird, sobald vom Server eine Antwort empfangen wurde.
Die Daten, die vom Server empfangen werden, werden dekodiert und überprüft. Wenn eine korrekte Antwort empfangen wird, wird sie als Javascript Objekt an die Callback – Funktion übergeben.

4.4. Programmierumgebung

Für die beiden Modi, Interaktiv und Programmierung, gibt es jeweils eine eigene HTML – Vorlage, die die Grundstruktur der einzelnen Bedienelemente vorgibt. Sie werden mit Hilfe eines iframes in die Verwaltungsoberfläche eingebunden.
Die Programmierumgebung wird beim Laden der Seite initialisiert und der aktuelle Zustand beim Verlassen der Seite in localStorage gesichert. Die Implementierung für die Ein- und Ausgabe, sowie die Ansteuerung von Compiler und Laufzeitumgebung befinden sich in editor.js.
Als Eingabe-Editor wird ace.js verwendet, ein in Javascript geschriebener Code – Editor verwendet, der Hervorhebung von Codeelementen erlaubt.
Angesteuert wird er über ein Objekt namens editor, das von den HTML-Seiten in der übergeordneten Umgebung angelegt wird. Dadurch kann von der Verwaltungsumgebung einfacher auf den Editor zugegriffen werden. Die Ausgabe übernimmt ein Singleton namens log.
Für die Bedienelemente steht ein Singleton namens button zur Verfügung.

4.5. WebWorker

Für Compiler und Laufzeitumgebung wird in editor.js ein WebWorker namens compiler erzeugt. Ein WebWorker ist ein Javascript, das im Hintergrund in einem eigenen Thread ausgeführt wird. Ich habe mich für die Ausführung von Compiler und Laufzeitumgebung als WebWorker entschieden, damit die Benutzeroberfläche auch während rechenintensiveren Aufgaben weiterhin Eingaben akzeptiert.
Nachrichten an den Compiler und die Laufzeitumgebung werden an diesen WebWorker mit dem Kommando postMessage übergeben.

Steuerung des WebWorkers

Nachricht	Zusätliche Daten	Beschreibung
reset	-	Versetzt Compiler und Laufzeitumgebung in Ihren Ausgangszustand.
features	features	Legt die Spracheigenschaften fest
library	source … Source Code	Übersetzt ein predefined Item und fügt alle Elemente der Laufzeitumgebung hinzu.
item	source … Source Code	Überprüft ob ein predefined Item übersetzt werden kann.
execute	source … Source Code kind … instruction | program	Übersetzt eine Befehlsfolge oder ein Programm, mit Rücksicht auf die Ausführungsart. Führt dann die Befehlsfolge aus, oder leitet die Ausführung eines Programms ein.
debug	source … Source Code kind … instruction | program	Übersetzt eine Befehlsfolge oder ein Programm, mit Rücksicht auf die Ausführungsart. Und Leitet dann die Ausführung des Codes ein.
run	-	Ändert die Ausführungsart des Programms von „debug“ auf „execute“. Das Programm wird dann in Folge ab dem Nächsten Kommando ausgeführt.

step	-	Führt das Nächste Kommando des Programms aus.

Tabelle 3: Steuerung des WebWorkers

Der WebWorker generiert Rückmeldungen an die Benutzeroberfläche und sendet diese mit dem Kommando postMessage. In editor.js wird die Reaktion auf Nachrichten von Compiler und Laufzeitumgebung festgelegt.

Rückmeldungen des WebWorker

Nachricht	Zusätliche Daten	Beschreibung
error	line …. Position des Fehlers content …. Ausgabe	Ein Fehler ist aufgetreten
finished	**-**	Die Ausführung wurde beendet
sucess	content …. Ausgabe	Die Ausführung war erfolgreich
executing	-	Die Übersetzung ist abgeschlossen und die Ausführung beginnt
step	Line …. aktuelle Position im Programm	Ein Schritt der Ausführung ist abgeschlossen
variables	content …. array aller im Moment sichtbaren Variablen. Für jede Variable werden die Felder geliefert : name … Variablenname type … Typ der Variable value … Wert der Variable	Liefert alle im Moment sichtbaren Variablen

Tabelle 4: Rückmeldungen des WebWorker

Ein weiterer Vorteil eines WebWorkers ist, dass mit dem Kommando importScripts zusätzliche Scripte im Javascript Code importiert werden können. Ohne Webworker müssten zusätzliche Scripte im HTML-Code importiert werden. Durch den importScripts wird klarer ersichtlich, welche zusätzlichen Scripte verwendet werden.
Der WebWorker delegiert seine Aufgaben an Compiler und Laufzeitumgebung, welche ich im Folgenden genauer beschreiben werde.

4.5.1. Compiler

4.5.1.1. Lexer

Die Aufgabe des Lexers ist es, die Zeichen des Sourcecode zu scannen und in Symbole zu übersetzen. Der Lexer erzeugt eine Symbolkette, die dann vom Parser weiter verarbeitet wird.
Leerzeichen und Kommentare spielen für die Übersetzung keine Rolle und werden ignoriert. Der Lexer merkt sich allerdings Kommentare, die dann vom Parser ausgelesen werden können, um später im übersetzten Code als Kommentar eingefügt zu werden.
Der Lexer übernimmt auch die Aufgabe, deaktivierte Schlüsselwörter zu filtern. Stößt er auf ein Schlüsselwort, das nicht aktiviert ist, erzeugt er eine Fehlermeldung.
Stösst der Lexer auf Namen im Programmcode, dann stellt er ihnen ein _$ voran. Dadurch wird ein eigener Namensraum generiert, der verhindert, dass es bei der Ausführung des Programms zu einem Namenskonflikt mit der Implementierung von Compiler und Laufzeitumgebung kommt.

4.5.1.2. Parser

Der Parser übergibt den Source-Code an den Lexer, der ihn in eine Kette von Symbolen übersetzt. Diese Symbolkette wird dann vom Parser nach dem Prinzip des rekursiven Abstiegs in einen abstrakten Syntax Baum übersetzt. Gleichzeitig wird eine Symboltabelle erstellt.

Der Parser kennt 3 Modi, die er für die Übersetzung verwenden kann, library, instruction und program. Im Modus library werden die voreingestellten Spracheigenschaften ignoriert, es sind alle Befehle erlaubt. Für die Typisierung wird „Optional Declaration“ verwendet. Somit hat man bei der Erstellung eines „predefined Items“ größtmögliche Freiheit.
Im Modus program und instruction werden die voreingestellten Spracheigenschaften berücksichtigt.Im Modus instruction darf der ; nach dem letzten Befehl fehlen.
Ein weiterer Unterschied zwischen den Modi ist der Umgang mit der Symboltabelle. Einträge in die Symboltabelle, die im library - und instruction - Modus vorgenommen werden, bleiben für weitere Übersetzungsvorgänge erhalten. Einträge im program – Modus werden bei einer neuerlich Übersetzung gelöscht.

Deaktivierte Spracheigenschaften werden normalerweise vom Lexer erkannt. Eine Ausnahme stellt das Schlüsselwort while dar, da es sowohl in der while – als auch in der do – while - Schleife verwendet wird. Wenn die do – while - Schleife aktiviert ist, die while – Schleife aber deaktiviert, kann dies nicht mit einfachem Verbieten des Schlüsselwortes while gelöst werden, da es sich sowohl um den Beginn einer while – Schleife als auch um das Ende einer do – while – Schleife handeln kann. Die Unterscheidung ist also nur mit Hilfe der Grammatik möglich. Daher wird in diesem Fall die Verwendung einer while – Schleife im Parser und nicht im Lexer verhindert.

Die Grammatik der Sprache Chill verändert sich an 3 Stellen, je nach Art der Typisierung. Die Grammatik für VarDecl, Function und ArrayDecl variiert dabei wie folgt:

(i) Untyped

```
VarDecl     := Ident "=" Expr ";"
Function    := "function" Ident "(" Ident {"," Ident }] ")"  CodeBlock
```

(ii) Optional Declaration

```
VarDecl     :=  (Ident "=" Expr | [Ident] ("variable" | "constant") Ident ["=" Expr])";"
Function    := [Ident] "function" Ident "(" [[Ident] Ident {"," [Ident] Ident }] ")" CodeBlock
ArrayDecl   := "array" Ident "["[Ident]"]" ["contains" Ident]
```

(iii) Declaration, optional type

```
VarDecl     := [Ident] ("variable" | "constant") Ident ["=" Expr] ";"
Function    := [Ident] "function" Ident "(" [[Ident] Ident {"," [Ident] Ident }] ")" CodeBlock
ArrayDecl   := "array" Ident "["[Ident]"]" ["contains" Ident]
```

(iv) Strict Typing

```
VarDecl     := Ident ("variable" | "constant") Ident ["=" Expr] ";"
Function    := [Ident] "function" Ident "(" [Ident Ident {"," Ident Ident }] ")" CodeBlock
ArrayDecl   := "array" Ident "["Ident"]" "contains" Ident
```

Die Unterschiede in der Grammatik sind für jeden dieser Fälle dadurch gekennzeichnet, ob ein Name (Ident) angegeben werden darf, kann oder muss. Daher wird der Funktion des Parsers zur Erkennung von Namen, ein Flag forced

übergeben, das angibt, ob ein Name an dieser Stelle zwingend erforderlich ist, oder nicht.

```
Ident(forced) // forced … true | false
```

Wenn forced true ist, wird im Falle, dass kein Ident erkannt wird eine Fehlermeldung generiert. Sonst wird dieser Umstand ignoriert.

Die Funktionen VarDecl(), Function() und ArrayDecl() des Parsers setzen das Flag forced abhängig davon, ob die Angabe eines Typs erforderlich ist. Darf kein Typ angegeben werden wird die Funktion Ident() nicht aufgerufen.

4.5.1.3. Übersetzung

Der abstrakte Syntaxbaum und die Symboltabelle werden einem Objekt mit dem Namen „translator“ übergeben. In diesem Objekt wird der abstrakte Syntaxbaum rekursiv durchlaufen und abhängig vom jeweiligen Knoten der entsprechende Code erzeugt. Während der Übersetzung wird mit Hilfe der Symboltabelle die Symantik des Programmes überpürft.
Eine einfache Übersetzung in Javascript Code ist nicht möglich, da Javascript Programme, Befehl für Befehl, abgearbeitet werden und dabei nicht unterbrochen werden können. Sollte im Programmcode eine Endlosschleife oder eine Endlosrekursion stehen, würde diese dazu führen, dass das Programm, ohne unterbrochen werden zu können, weiterläuft. Als einzige Möglichkeit bleibt dann den Browser neu zu starten.

4.5.1.3.a Event Basierter Code

Ein WebWorker, wie er für Compiler und Laufzeitumgebung verwendet wird, bietet zwar eine Möglichkeit ihn mit dem Befehl terminate zu beenden, allerdings wird vor der Beendigung des WebWorkers der gesamte noch anstehende Code ausgeführt.
Die Laufzeitumgebung von Javascript, wie auch die eines WebWorkers besteht aus einer Event – Queue. In dieser Queue werden abhängig von Ereignissen Programme bzw. Funktionen eigereiht und eine nach der anderen ausgeführt. Es wird zwar der gesamte Code eines Programmes bzw. einer Funktion, ohne es unterbrechen zu können, bis zum Ende ausgeführt, allerdings kann diese Event-Queue zwischen dem Ausführen zweier Ereignisse unterbrochen werden. Für einen WebWorker steht dafür der Befehl terminate zur Verfügung.

Damit nun eine Endlos - Schleife bzw. Rekursion den Browser blockiert, werden Befehle in Chill in einzelne Funktionen übersetzt, die dann als zeitverzögertes Ereignis gestartet werden. Es würde zwar ausreichen, bei jedem Schleifen – Kopf und jedem Funktionseintritt einen neuen Event in die Queue einzureihen, doch wären die Konsequenzen die selben, wie die Zerlegung auf einzelne Befehle. Außerdem kann so die genaue Position im Programm während der Ausführung angezeigt werden.

```
function m(){…jump(x["0_1"],2);}; x["0_1"]=function(){… jump(x["0_2"],3);};
x["0_2"]=function({… jump(x["0_3"],5);}; … m();
```

Die Funktion m() dient dabei als Einstieg in das Programm. Und dann wird eine Funktion nach der anderen aufgerufen.
Diese Vorgehensweise hat für die Übersetzung weitreichende Konsequenzen, denn Variablen, die innerhalb einer Funktion erzeugt werden, sind nicht außerhalb der Funktion sichtbar. Daher sind sie auch nicht in der Folgefunktion sichtbar. Daher ist es nötig, Variablen global zu erzeugen, damit sie auch in den folgenden Programmabschnitten sichtbar sind. Daraus folgt dann auch dass ein eigenes Speichermanagment erforderlich ist, um die korrekte Sichtbarkeit von Variablen im Programm sicherzustellen. Auch die Übergabe von Parametern an Funktionen kann nicht mehr über Parameter von Javascript – Funktionen, sondern muß über globale Variablen erfolgen.
Um nicht eine unübersichtliche Flut an globalen Variablen zu erzeugen und eine klare Struktur für die Übersetzung zu haben, habe ich mich dazu entschlossen, die Programme in Chill für eine in Javascript implementierte Stack – Maschine zu übersetzen, die ich im Kapitel Laufzeitumgebung vorstellen werde.

4.5.1.3.b Sprungadressen

Würde ein Programm nur aus der sequentiellen Abfolge von Kommandos bestehen, würde es ausreichen, die Funktionen, die die einzelnen Kommando‘s enthalten, fortlaufend zu nummerieren. Doch ein Programm kann Verzweigungen, Schleifen, Funktionen, etc. enthalten. Für diese Fälle ist es nicht mehr ausreichend, fortlaufende Nummern zu vergeben, um eindeutige Sprungadressen zu gewährleisten.
Wenn bei einer if – Anweisung die Eingangsbedingung nicht erfüllt ist, würde 2 Nummern weiter gesprungen. Steht in dem if Zweig aber eine Schleife, oder ein weiteres if, dann verschiebt sich diese Zahl nach hinten.

Um die Sprungadressen nicht gesondert auflösen zu müssen, habe ich mich entschieden, die Übersetzung in Sektionen zu unterteilen.
Sektionen funktionieren wie die Nummerierung von Kapiteln. Jeder Sektionsbeginn spannt eine neue Unterebene in der Nummerierung auf. Das Sektionsende beendet die Ebene wieder.
Alle Verzweigungen, Schleifen, etc. erhalten eigene Sektionen. Da innerhalb der Sektion die Sprungadressen konstant bleiben, lassen sie sich leicht berechnen.

4.5.1.3.c Zeilennummern

Die Zeilennummer, die ein Befehl im Programmtext hat, spielt für die Darstellung der Programmausführung eine wesentliche Rolle. In der Übersetzung wird immer dann, wenn eine neue Zeile begonnen wird, und bei jedem Beginn eines Codeblocks einer Variable l die aktuelle Zeilennummer zugewiesen. Damit ist der Laufzeitumgebung bekannt, welche Position des Programmes gerade ausgeführt wird und sie kann diese Position an die Programmierumgebung liefern.

4.5.1.3.d Übersetzungstabelle

Die folgende Tabelle beschreibt, wie Sprachkonstrukte in Chill übersetzt werden. Die Zielumgebung für die Übersetzung ist die in Javascript implementierte Laufzeitumgebung von Chill.
In der Tabelle verzichte ich auf die Darstellung von Sprüngen, die nur für die Visualisierung, aber nicht für die Funktion relevant sind, da sie die Übersetzung unübersichtlich machen, aber die Funktion nicht beeiflussen.

Übersetzungstabelle

Code	Übersetzung
`true`	`lit['_$boolean'].get(true)`
`1`	`lit['_$integer'].get(1)`
`1.0`	`lit['_$number'].get(1.0)`
`'a'`	`lit['_$character'].get('a')`
`"str"`	`lit['_$string'].get("str")`
`n=v` `/* n existiert bereits */`	`push(v);` `global['_$n'].assign();`

`n=v`
`/* n existiert nicht */`
	`push(v);`
	`v.create('_$n','_$any');`
	`global['_$n'].assign();`

variable n	v.create('_$n','_$any')
t variable n	v.create('_$n','_$t')
constant n	c.create('_$n','_$any')
t constant n	c.create('_$n','_$t')
+r	push(r); plus();
-r	push(r); minus();

l+r	push(l);
	push(r);
	add();

l-r	push(l);
	push(r);
	sub();

l*r	push(l);
	push(r);
	mul();

l/r	push(l);
	push(r);
	div();

l%r	push(l);
	push(r);
	mod();

function n
(a,...) {
}
	{f.create('_$n',f.enter('x["..."]',
	[{'n':'_$a','t':'_$any','c':false},...,]));};
	x["..."]=function(){
	s.enter();
	s.exit();
	f.exit('_$any');
	};

return v	push(v); f.exit('_$t',pop(),1);
n(a,..);	push(a); ... run(global['_$n'],'x["..."]')}; x["..."]=function(){...

if (cond) {}
else {}
	push(cond);
	/*if-begin*/
	if (pop().value)run(x["..._1"]);else
	run(x["..._2"]);};
	x["...1"]=function(){/*if-block*/
	s.enter();
	s.exit();
	run(x["..._3"]);
	};

	x["..._2"]=function(){/*else-block*/
	  s.enter();
	  s.exit();
	  run(x["..._3"]);
	};
	x["..._3"]=function(){/*else-end*/ ...

while(cond)
{}
	push(cond);
	/*while-begin*/if (pop().value)run(x["..._1"]);else
	run(x["..._2"]);}; x["..._1"]=function(){
	  s.enter();
	  s.exit();
	  push(cond);
	  if (pop().value)run(x["..._1"]);else
	run(x["..._2"]);
	};
	x["..._2"]=function(){/*while-end*/...

do {
}while(cond)
	.../*do-begin*/run(x["..._1"]);};
	x["..._1"]=function(){
	  s.enter();
	  s.exit();
	  push(cond);
	  /*while*/if (pop().value)run(x["..._1"]);else
	run(x["..._2"]);/**/
	};
	x["..._2"]=function(){/*do-end*/....

for(i=init;i
<limit;i=i)
{
}
	.../*for-init*/
	push(init);
	global['_$i'].assign();
	/*for-cond*/
	push(global['_$i']);
	push(limit);
	lt();
	if (pop().value)run(x["..._1"]);else
	run(x["..._2"]);}; x["..._1"]=function(){
	  /*for-begin*/
	  s.enter();
	  s.exit();
	  /*for-inc*/
	 push(global['_$i']);
	 push(lit['_$integer'].get(1));

	add();
	global['_$i'].assign();
	/*for-cond*/push(global['_$i']);
	push(llimit);
	lt();
	if (pop().value)run(x["..._1"]);else
	run(x["..._2"]);
	};
	x["..._2"]=function(){/*for-end*/...

array name[iType] contains eType	t.addArray('_$name','_$iType','_$eType');
a[i]	push(i); push(global['_$a].access(pop()));

struct name
{
t variable a;
u constant b;
}
	...run(mx)};
	t.addClass('_$name,'struct','null');
	type=t['_$name'];
	type.fields['_$a']={'typeName':'_$t','name':'_$a','
	constant':false,'bounds':null};
	type.fields['_$b']={'typeName':'_$u','name':'_$b','
	constant':true,'bounds':null};
	function mx(){...

s.a	push(global['_$s'].access('_$a'));

class name extends s
{
constructor(){};
variable a;
function f(){}
}
	...run(mx);};/*Class _$name*/
	t.addClass('_$name','class','_$s');
	type=t['_$name'];
	type.fields['_$a']={'typeName':'_$t','name':'_$a','
	constant':false,'bounds':undefined};
	{f.create('_$f_$name',f.enter('x["..._1"]',[]));};
	x["..._1"]=function(){
	s.enter();
	s.exit();
	f.exit('_$any');
	};
	type.methods['_$f']=global['_$f_$name'];
	
	f.create('construct_$name',f.enter('x["..._2"]',
	[]));};
	x["..._2"]=function(){
	s.enter();

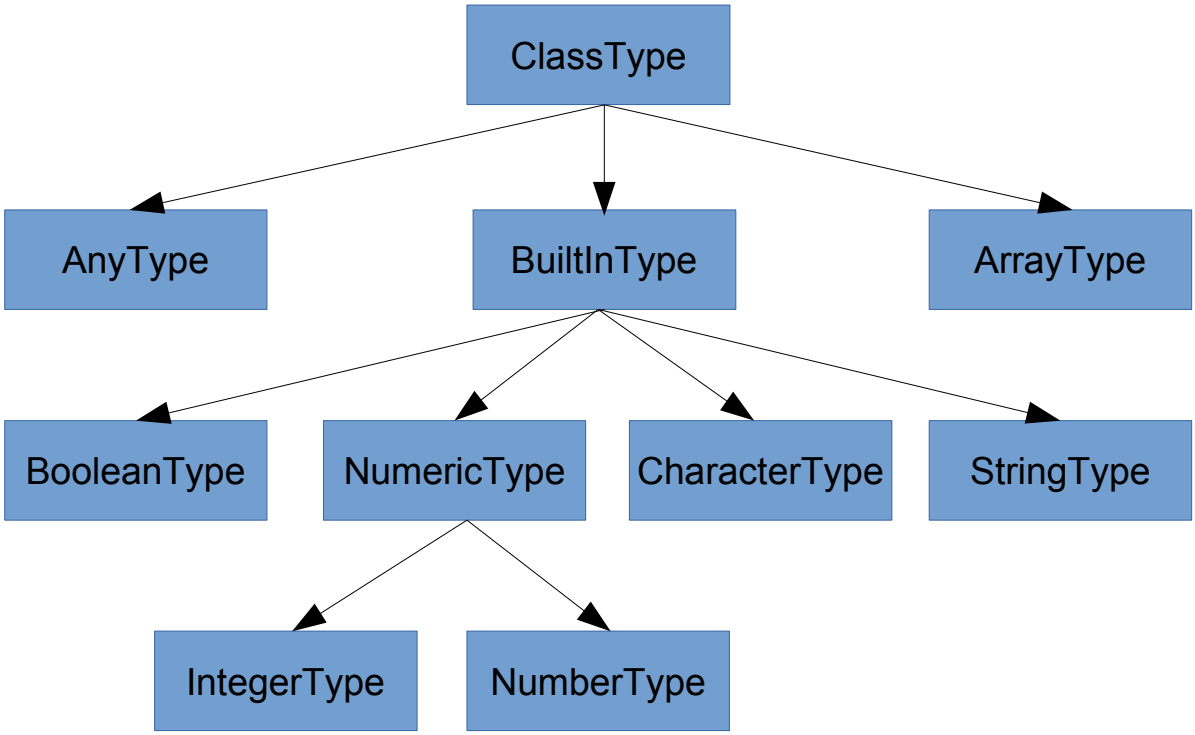
	s.exit();
	f.exit('_$name');};
	
	type.methods['construct']=global['construct_$x'];
	}
	/*Class _$name*/function mx(){...

new name();	obj=o.create('_$name'); push(obj);
name.f();	run(global['_$name'].access('_$f'),'x["..._x"]');}; x["..._x"]=function(){...
javascript(v ,...)# ….#end	use({'n':'_$v','t':'_$t'},...); runJS('...');

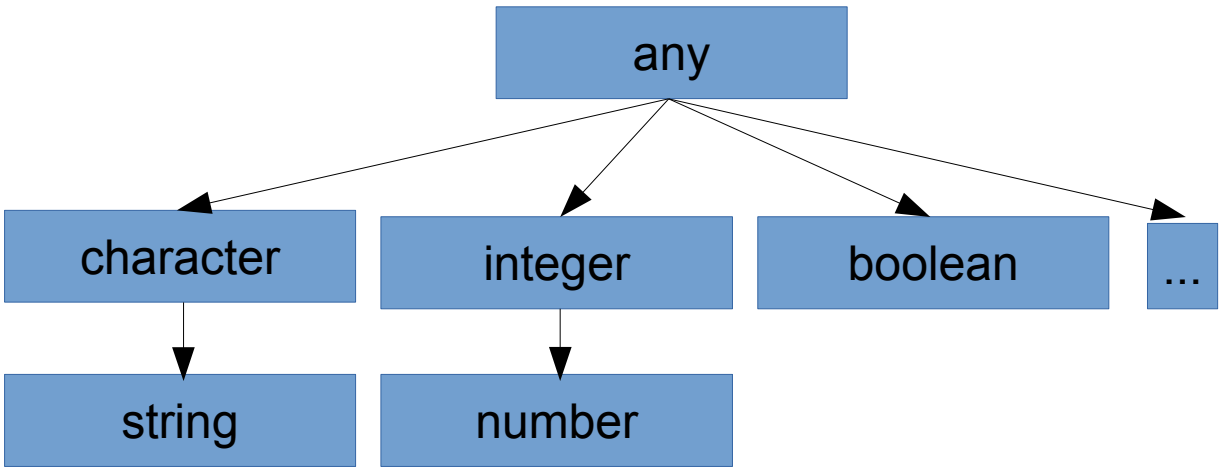
Tabelle 5: Übersetzungstabelle

4.5.2. Typ Prüfung

Das Typisierungs System von Chill habe ich in der Datei type.js implementiert. Das System wird zur Prüfung der Kompatibilität von Typen während der Übersetzung und während der Laufzeit verwendet. Die Idee dieser Implementierung ist, die Prüfung auf eine Klassenhierarchie zu basieren. Da Javascript keine Klassen kennt, sondern Prototypen, habe ich eine Klassenhierarchie mit Hilfe von Delegation nachgebaut.
Dabei leitet sich jeder Typ von einer „Basisklasse“ ClassType ab. Jeder Typ, der diese Klasse implementiert, kennt seine Basisklasse und die Klassen die von Ihm geerbt haben.
Zusätzlich kennt jeder Typ, der ClassType implementiert, seine Methoden und Felder. Und für alle Operationen der StackMaschine sind in ClassType Methoden implementiert, die überprüfen, ob die Operation auf den Typ angewendet werden kann.
Durch Delegation kann eine Struktur nachgebildet werden, die zwar noch keine korrekte Struktur für die Typprüfung erlaubt. Sie genügt aber, um Objekte als Beschreibung für alle möglichen in Chill vorkommenden Typen erzeugen zu können.



Die tatsächliche für die Prüfung relevante Struktur sieht wie folgt aus. Alle Typen sind von any abgeleitet. Werden im Programm Typen und Strukturen deklariert, werden diese auch von any abgeleitet. Klassen, die von keiner anderen Klasse abgeleitet werden, werden von any abgeleitet. Sonst werden sie von Ihrer Basisklasse abgeleitet.



Alle Typbeschreibungen werden in dem Array t gespeichert. Für jeden Typ existiert genau eine Beschreibung.

4.5.2.1. Laufzeitumgebung

4.5.2.1.a Stackmaschine

Die Basis der Laufzeitumgebung bildet eine Stackmaschine, die in stackMaschine.js implementiert ist. Der Stack wird durch das Javascript – Array stack gebildet. Die Maschine versteht folgende Kommandos:

Kommandos der Stackmaschine

Kommando	Beschreibung
push (v)	Legt den Inhalt von v auf den Stack
pop ()	Liefert den obersten Eintrag des Stacks
not()	Invertiert den obersten Eintrag des Stacks
or()	Oder - Verknüpfung der obersten Einträge des Stacks
and()	Und - Verknüpfung der obersten Einträge des Stacks
eq()	Vergleich auf Äquivalenz der obersten Einträge des Stacks
ne()	Vergleich auf Ungleichheit der obersten Einträge des Stacks
gt()	Ist der zweitoberste Eintrag größer als der oberste?
ge()	Ist der zweitoberste Eintrag größer oder gleich als der oberste?
lt()	Ist der zweitoberste Eintrag kleiner als der oberste?
le()	Ist der zweitoberste Eintrag kleiner oder gleich als der oberste?
inArray()	Ist der oberste Eintrag ein Index des Array, das der zweitoberste Eintrag des Stacks ist
plus()	Stellt dem obersten Eintrag ein plus voran
minus()	Stellt dem obersten Eintrag ein minus voran
add()	Addiert die obersten beiden Einträge des Stacks
sub()	Subtrahiert die obersten beiden Einträge des Stacks
mul()	Multipliziert die obersten beiden Einträge des Stacks
div()	Dividiert den zweit obersten Einträge des Stacks durch den Obersten
mod()	Dividiert den zweit obersten Einträge des Stacks durch den Obersten und berechnet den Restwert.

Tabelle 6: Kommandos der Stackmaschine

Bei der Ausführung der Kommandos wird die Laufzeit – Typ - Prüfung vorgenommen und auf e.v. Divisionen durch Null, etc. geprüft.

4.5.2.1.b Werte

Javascript kennt keine Typisierung. Daher wird nur der aktuelle Wert in Variablen gespeichert. Chill unterstützt Typisierung. Um eine Typ – Prüfung zur Laufzeit zu ermöglichen, muss daher zu jedem Wert auch der entsprechende Typ gespeichert

werden. Genau genommen wird zu jedem Wert ein Zeiger auf die entsprechende Typbeschreibung gespeichert.
Außerdem verfolgen Array‘s, Strukturen und Objekte in Chill andere Konzepte als die entsprechenden Pendants in Javascript.
Diesen Umständen wird in der Implementierung in value.js Rechnung getragen. Sie enthält folgende Singletons für den Umgang mit Werten.

Singleton‘s zum Umgang mit Werten

Objekt	Beschreibung
jsValue	Objekt für den Umgang mit Standard Typen.
arrayValue	Objekt für den Umgang mit Array‘s
structValue	Objekt für den Umgang mit Strukturen
objectReference	Objekt für den Umgang mit Referenzen auf Objekte
anyValue	Objekt für Werte, die jeden Typ beinhalten können

Tabelle 7: Objekte zum Umgang mit Werten

Jedes dieser Objekt enthält mindestens die folgenden Methoden.

Objekte für den Umgang mit Werten

Methode	Beschreibung
assign(thisValue,r)	Weist thisValue den Wert von r zu.
access(thisValue,index)	Liefer den Wert von thisValue, bestimmt durch index zurück
toString(thisValue,propQuote)	Konvertiert thisValue in einen String. propQuote gibt die Anführungszeichen an, die für ein Element verwendet werden sollen.
equal(l,r)	Vergleicht l und r auf Äquivalenz

Tabelle 8: Objekte für den Umgang mit Werten

Jedes Singleton kann auch weitere Methoden enthalten, wie zum Beispiel jsValue die Methode create, um Werte zu erzeugen, oder arrayValue die Methode length, die die Anzahl der Elemente des Arrays zurückliefert.

4.5.2.1.c Literale

Kommt im Quellcode des Programmes ein Literal vor, so muß dieses auch erzeugt werden. In einer naiven Herangehensweise könnte für jedes vorkommende Literal ein Wert erzeugt werden. Doch diese Vorgehensweise kann schnell sehr speicherintensiv werden. Wie zum Beispiel im folgenden Fall:

```
array arr[];
arr variable a;
for (i=0;i<100;i=i+1) {
          a[i]=0;
}
```

In diesem Fall würde 101 mal ein Literal für die Zahl 0 erzeugt, das jeweils nur einmal verwendet wird. Dann wird jedes dieser 101 Literale vom Garbage Collector des Browsers gelöscht.
Der Garbage - Collector benötigt für das Löschen Zeit, die die Ausführungsgeschwindigkeit des Programmes stark drosseln kann.
Um Fälle wie diese zu vermeiden habe ich in literal.js ein Singleton lit implementiert, das dabei hilft, Werte für Literale zu erzeugen.
Für Boolsche Werte werden in lit 2 fixe Literale zur Verfügung gestellt, eines für true und eines für false.
Für alle anderen Typen steht jeweils ein Array mit maximal 1000 Eintragungen zur Verfügung, in dem bereits verwendete Literale gespeichert werden. Wird allerdings öfter als 3 mal kein Literal in dem Array gefunden, dann geht lit davon aus, dass sich die Literale weiter verändern und speichert sie in der Folge nicht mehr.

4.5.2.1.d Speicherverwaltung

Die Speicherverwaltung ist in variable.js implementiert. Sie stellt folgende Objekte für die Speicherplatzverwaltung zur Verfügung. Da die Objekte oft in der Übersetzung verwendet werden, sind die Namen bewusst kurz gewählt, um die Übersetzung so kurz wie möglich zu gestalten.

Objekte zur Speicherplatzverwaltung

Objekt	Beschreibung	Methoden
p	Programm, beginnt und beendet die Ausführung. Im program Modus wird am Ende des Programmes der Globale Scope aufgeräumt. Im instruction – Modus bleibt er erhalten.	enter … Beginnt ein Programm exit … Beendet ein Programm

s	Scope, verwaltet Sichtbarkeitsbereiche. Beim Verlassen eines Scopes werden Deklarationen dieses Scopes automatisch gelöscht.	enter … Beginnt einen neuen Scope exit … Beendet einen Scope add … Erzeugt einen neuen Scope count … Liefert die Anzahl der Scopes
g	Verwaltet globale Objekte. Sollte bei der Erzeugung eines neuen globalen Objekt ein Namenskonflikt entstehen, wird das bisherige globale Objekt gesichert und bei Aufölsung des Namenskonflikt wieder hergestellt.	create … Erzeugt ein globales Objekt assign … Weist einem etwas einem globalen Objekt zu get … Liefert das globale Objekt zurück free … Gibt das globale Objekt frei
a	Erzeugt Objekte denen etwas zugewiesen werden kann. Solche Objekte können Variablen, Konstanten, aber auch Felder von Arrays, Strukturen, oder Objekten sein.	create … Erzeugt ein Objekt dem etwas zugewiesen werden kann. initValue … Wird verwendet um Array‘s zu initialisieren
v	Variablen	create … Erzeugt eine Variable
c	Konstanten	create … Erzeugt eine Konstante
f	Verwaltung von Funktionen. Übernimmt die Übergabe von Parametern und des Rückgabewertes.	create … Erzeugt eine Funktion enter … Beginnt die Funktion exit … Beendet die Funktion
o	Objekte	create … Erzeugt ein Objekt findField … Liefert ein Feld call … Führt eine Methode aus

Tabelle 9: Singleton‘s zum Umgang mit Werten

Eine besondere Rolle hat das Objekt a, das Objekte erzeugt, denen etwas zugewiesen werden kann. Solche Objekte können Variablen, Konstanten, aber auch Felder von Arrays, Strukturen, oder Objekten sein. Die Methoden und Eigenschaften müssen die gleichen Schnittstellen wie Werte aufweisen.

Methoden und Eigenschaften eines Objektes, dem etwas zugewiesen werden kann

Name	Beschreibung
name	Name des Objekts
value	Wert des Objekts
type	Typ des Objektes
untyped	Darf das Objekt alle Typen enthalten
assign	Weist dem Objekt einen Wert zu
access	Greift auf ein Element des Objektes zu
toString	Liefert den Inhalt des Objektes als String
toJS	Konvertiert das Objekt in ein Javascript Äquivalent
fromJS	Initialisiert das Objekt mit Hilfe eines Javascript Äquivalent

Tabelle 10: Methoden und Eigenschaften eines Objektes, dem etwas zugewiesen werden kann

4.5.2.1.e Jump, Run und Done

Die Funktion run(dest) führt eine Funktion aus, der als Parameter dest übergeben wird, als Event aus. Sollte während der Ausführung ein Fehler auftreten, kümmert sich run um die Fehlerbearbeitung, schickt eine Meldung an das Benutzer Interface und räumt die Laufzeitumgebung auf.
Um ein Programm von Beginn bis zum Ende auszuführen, könnte für jeden Sprung die Funktion run verwendet werden. Dies ist nützlich für „predefined Items“, denn sie sollen unterbrechungsfrei ausgeführt werden.
Programme in Chill werden allerdings Schritt für Schritt ausgeführt. Am Ende jedes Schrittes wird der Programmierumgebung der aktuelle Zustand übermittelt. Die Ausführung wird fortgesetzt, wenn von der Programmierumgebung der Befehl dafür empfangen wird. Diese Funktionalität übernimmt die Funktion jump.Sie speichert die Funktion, die im nächsten Schritt ausgeführt werden soll und übermittelt die aktuell sichtbaren Variablen und die aktuelle Position im Programm an die Programmierumgebung. Die gespeicherte Funktion wird dann, nach der Rückmeldung der Programmierumgebung, mit Hilfe von run ausgeführt.
Diese schrittweise Ausführung ist nicht nur für das schrittweise durchlaufen eines Programmes im Debug – Modus nötig, sondern wird auch verwendet, um die Geschwindigkeit der Ausführung steuern zu können.
Am Ende eines Programmes wird die Funktion done aufgerufen. Sie liefert den Rückgabewert an die Programmierumgebung und räumt die Laufzeitumgebung auf.

4.5.2.1.f Javascript

Javascript Code wird in einer gekapselt Umgebung ausgeführt. Die dafür benötigten Funktionen werden in javascript.js implementiert.
Um globale Variablen der Laufzeitumgebung zu verstecken wird eine Maske erstellt. Diese Maske ist ein Array, in der alle globalen Objekte der Laufzeitumgebung als undefined initialisiert werden. Ausnahmen bilden im Moment nur „console“, „btoa“ und „atob“, um Ausgaben auf die Konsole und Base64 De- und Encoding zu erlauben. Weitere Ausnahmen können einfach ergänzt werden, indem sie in der Funktion allowedGlobalObject ergänzt werden.
Der Javascript Code wird der Funktion runJs übergeben. runJs kapselt den Code in eine Funktion. Diese Funktion wird dann mit der Maske als globalen Raum ausgeführt.
Um zu verhindern, dass eine Exception die im Javascript – Code entsteht, die Laufzeit Umgebung in einem unbestimmten Zustand hinterlässt, wird die Funktion innerhalb eines try Blocks aufgerufen. Die Fehlermeldung einer möglichen Exception wird an die Programmierumgebung gesendet.
Variablen aus Chill werden mit Hilfe der Funktion use in Javascript importiert. Nachdem der Code ausgeführt wurde, werden die importierten Variablen in Chill auf den aktuellen Stand gebracht.

5. Erweiterungen

5.1. Mathematik Bibliothek

Die einfachste Form Chill zu erweitern bieten predefined Items. Konstanten, Variablen, Funktionen, Typen, können so implementiert werden. Sie können sowohl im interaktiven -, als auch im Programmiermodus zur Verfügung gestellt werden. Predefined Items erlauben auch die Angabe eines Hilfetextes.
Durch die Möglichkeit, Javascript Code in Chill einzubinden, eröffnen sich zahlreiche Möglichkeiten für die Erweiterung.
Als Beispiel möchte ich hier die Implementierung einer Sammlung mathematischer Funktionen in Chill anführen. Diese Sammlung basiert auf Javascript's Math.
Eine Implementierung könnte wie folgt aussehen:

```
constant pi=3.14159265358979323846264338327950288419 71;
constant e=2.718281828459043;
number function abs(number x){//Returns the absolute value of x
    number variable ret=0;
    javascript(x,ret) # ret=Math.abs(x); #end
    return ret;
}
number function acos(number x){//Returns the acos of x,in radians
    number variable ret=0;
    javascript(x,ret) # ret=Math.acos(x); #end
    return ret;
}
number function asin(number x){//Returns the asin of x,in radians
    number variable ret=0;
    javascript(x,ret)#ret=Math.asin(x); #end
    return ret;
}
number function atan(number x){ /*Returns the arctangent of x as
a numeric value between -PI/2 and PI/2 radians */
    number variable ret=0;
    javascript(x,ret) # ret=Math.atan(x); #end
    return ret;
}
integer function ceil(number x){//Returns x, rounded upwards
```

```
    integer variable ret=0;
    javascript(x,ret) # ret=Math.ceil(x); #end
    return ret;
}
number function cos(number x){ //Returns the cos of x(in radians)
    number variable ret=0;
    javascript(x,ret) # ret=Math.cos(x.value); #end
    return ret;
}
number function exp(number x){ //Returns the value of Ex
    number variable ret=0;
    javascript(x,ret) # ret=Math.exp(x.value); #end
    return ret;
}
integer function floor(number x){//Returns x, rounded downwards
    integer variable ret=0;
    javascript(x,ret) # ret=Math.floor(x); #end
    return ret;
}
number function log(number x){//Returns the natural log of x
    number variable ret=0;
    javascript(x,ret) # ret=Math.log(x); #end
    return ret;
}
number function pow(number x,number y){//Returns x^y
    number variable ret=0;
    javascript(x,y,ret) # ret=Math.pow(x,y); #end
    return ret;
}
number function random(){//Returns a random number between 0 & 1
    number variable ret=0;
    javascript(ret) # ret=Math.random(); #end
    return ret;
}
integer function round(number x){//Rounds x
    integer variable ret=0;
    javascript(x,ret) # ret=Math.round(x); #end
    return ret;
}
```

```
number function sin(number x){ //Returns the sin of x(in radians)
    number variable ret=0;
    javascript(x,ret) # ret=Math.sin(x); #end
    return ret;
}
number function sqrt(number x){//Returns the square root of x
    number variable ret=0;
    javascript(x,ret) # ret=Math.sqrt(x); #end
    return ret;
}
number function tan(number x){ //Returns the tangent of an angle
    number variable ret=0;
    javascript(x,ret) # ret=Math.tan(x); #end
    return ret;
}
```

5.2. print

Dieses Beispiel soll zeigen, wie es möglich ist, eine Funktion print zu implementieren. Print soll es ermöglichen, Werte auszugeben.
Dazu müssen Nachrichten an die Programmierumgebung gesendet werden können und von ihr verstanden werden.
Nachrichten werden von der Laufzeitumgebung mit postMessage(...) an die Programmierumgebung gesendet. Diese Methode kann nicht direkt in den Javascript - Code eingebunden werden, da sie Ihre Laufzeitumgebung überprüft und nur von einem WebWorker aus ausgeführt werden kann. Daher muss sie in eine Funktion verpackt werden, die man zum Beispiel in javascript.js implementiert.

```
function sendMessage(type,content) {
     postMessage({“type“:type,“content“:content});
}
```

Diese Funktion kann nun in der Funktion allowedGlobalObject für die Verwendung in Javascript freigegeben werden, indem die Zeile:

case “ sendMessage“:

innerhalb des switch(elem) hinzugefügt wird.
Jetzt ist es möglich, eine Funktion print in Chill zu erstellen:

```
function print (string msg) {
     javascript(msg) # sendMessage(“print“,msg); #end
}
```

Die gesendete Nachricht muss natürlich auch von der Programmierumgebung verstanden werden. Dies wird in editor.js in der Funktion compiler.onMessage implementiert. Für die Ausgabe im Ausgabefenster kann das Objekt log verwendet werden.

```
case “print“:
     log.Msg(msg.data.content);
break;
```

6. Beurteilung

6.1. Designprozess

Der Designprozess, den ich verfolgt habe, hat Bottom – Up funktioniert. Am Anfang stand die Idee, eine vereinfachtes Javascript als Sprache zu verwenden. Während ich daran gearbeitet habe, diese Sprache zu entwerfen, habe ich mich mit Javascript vertraut gemacht. Die Erfahrungen, die ich dabei gemacht habe, haben den tatsächlichen Sprachentwurf stark beeinflusst. Ein Beispiel dafür ist die langwierige Fehlersuche, die in Javascript häufig nötig ist, da es nicht typisiert ist. Ein Schreibfehler an einer Stelle eines Programmes kann so sehr leicht zu einem Fehler führen, der von einem nicht initialisierten Funktionsparameter hervorgerufen wird, der irgendwo an einer ganz anderen Stelle des Programmes auftritt. Dieser Umstand hat zu der Entscheidung geführt, Typprüfung in Chill möglich zu machen.
Rückblickend betrachtet wäre es wahrscheinlich sinnvoller gewesen, eine Sprache als Ausgangsbasis zu wählen, in der ich schon Erfahrung sammeln konnte und diese zu vereinfachen bzw. zu erweitern.
Auch bei der Verwaltungsumgebung fehlte mir lange Zeit das Gesamtbild. Sie ist um die Programmierumgebung herum entstanden.
Zusammengefasst wäre es rückblickend betrachtet besser gewesen Top – Down zu entwickeln, mit dem Endergebnis vor Augen die Implementierung zu starten und nicht Design und Implementierung parallel zu entwickeln.
Auf der anderen Seite wären einige Ideen nicht entstanden. Die Auswahlmöglichkeit der Art der Typisierung und der Sprachfunktionen ist hier nur ein Beispiel.

6.2. AST - Interpreter vs. Übersetzung nach Javascript

Eine wesentliche Entscheidung, in einer frühen Phase der Implementierung, war es an Stelle eines AST-Interpreters auf eine Übersetzung nach Javascript zu setzen. Der Grund für diese Entscheidung war die Annahme, dass ein in Javascript übersetzter Code Vorteile in der Performance bringen würde. Zur Zeit dieser Entscheidung war mir noch nicht gelungen, das Problem zu lösen, dass Endlosschleifen die Javascript Laufzeitumgebung und damit auch den Web-Browser blockieren.

Durch die Lösung die ich in Kapitel 4.5.1.3.a beschreibe ist es sehr unwahrscheinlich, dass die Übersetzung in Javascript einen spürbaren Performance - Gewinn gegenüber einem AST-Interpreter bringt, da der übersetzte Code sehr viele Funktionsaufrufe beinhaltet.
Da der AST die Zeilennummern des Programmes enthält, wäre es auch nicht nötig gewesen, diese mit in die Übersetzung mit einfließen zu lassen. Zusätzlich hätte die Implementierung eines AST - Interpreters eine viel klarere Ausführung ergeben.

6.3. JQuery & W2ui

Die auf JQuery aufbauende User-Interface Bibliothek W2ui hat sich für mich als sehr wichtiges Werkzeug für die Entwicklung von Web-Applikationen erwiesen. Viele Javascript Frameworks setzen hauptsächlich auf visuelle Effekte und weniger auf eine konsistente Widget-Gestaltung.
Google's Closure bietet zwar ebenfalls sehr viele Gestaltungsmöglichkeiten, ist aus meiner Sicht aber viel zu umfangreich und wenig performant. Prototype greift sehr tief in Javascript ein, zum Beispiel wird der Charakter von Javascript Array's verändert, ein Umstand der mir bei der Implementierung der Laufzeitumgebung Probleme bereitete.
JQuery und W2ui wurde von allen Browsern, mit denen ich getestet habe, unterstützt und führte immer zu dem selben, einheitlichen Erscheinungsbild.

6.4. Firebug

Eine der größten Herausforderungen bei meiner Arbeit war die Fehlersuche in Javascript. Als sehr zuverlässiges und nützliches Werkzeug hat sich für mich Firebug bewährt. Hauptsächlich habe ich die Javascript – Konsole von Firebug dazu verwendet. Die Funktion debugMsg in wwOnError.js, die ich zur Fehlersuche in Übersetzer und Laufzeitumgebung verwendet habe, ist ein Beispiel dafür.
Besonders nützlich war für mich die Möglichkeit, die Struktur von Objekten genau analysieren zu können, die Firebug bietet.

7. Referenzen

W2UI User Interface Library	http://w2ui.com/web/
Jquery	http://jquery.com/
ACE Code Editor	https://ace.c9.io/
HTML5	http://www.html5rocks.com/de/
W3Schools Online Web Tutorial	http://www.w3schools.com/
SelfHTML – Javascript	https://wiki.selfhtml.org/wiki/JavaScript
Python Software Foundation	https://www.python.org
Code is you're canvas	https://trinket.io

Thomas C. Wilson, Joseph Shrott	Pascal from Begin to End
Sanford Leestma, Larry Nyhoff	Programming an Problem Solving with Modula
G.Blaschek, G.Pomberger, F.Ritzinger	Einführung in die Programmierung mit Modula-2
Bjarne Stroustrup	Die Programmiersprache C++
Commodore	Alles über den Commodore 64
Plenge	Das Grafikbuch zum Comodore 64

Lebenslauf

Persönliche Daten

Name	:	Rainer Gutkas Bakk. Techn. Inf.
Geboren	:	8.4.1974, St. Pölten
Eltern	:	Univ. Prof. Hofrat Dr. Karl & Senta Gutkas
Staatsbürgerschaft	:	Österreich
Führerscheine	:	A, B, C, F, G, eigener PKW
Familienstand	:	ledig

Ausbildungsweg

1980 - 1984	:	Daniel Gran Volksschule St. Pölten
1984 - 1988	:	Bundesgymnasium St. Pölten
1988 - 1993	:	Höhere Bundes Lehranstalt für Elektrotechnik St.Pölten, Mautra am 23.Juni 1993
1993 - 2003	:	Johannes-Kepler-Universität-Linz, Bakkalaureat Informatik
2007 – 2008	:	Lehranstalt für Heilpädagogische Berufe Freytaggasse 32, Basismodul
2009	:	Heimhelferprüfung
2013 – Heute	:	Johannes-Kepler-Universität-Linz, Masterstudium Computer - Sience

Arbeitsverhältnisse

1993 - 1994	:	Technik-Trainer im Seminarhotel Moasterhaus
1994 - 2001	:	Tutor für technische Informatik & Einführung in die Programmierung an der JKU Linz
2000 - 2002	:	Aufbau und Präsentation des Kiosksystems WorldBeat bei diversen Konferenzen
2002	:	Spielpädagogische Kinderbetreuung für die Firma AIK
2002	:	Hilfsarbeiten für Bühnenaufbau im Licht- und Tontechnikbereich
September 2003	:	Patentanwälte Miksovsky & Pollhammer OEG, Sachbearbeiter auf technisch /patentrechtlichem Gebiet(u.a. Übersetzung von Patentschriften)

2004 - 2005	:	Zivildienst - JAW - Werkstätte Innermanzing
2005	:	Kirchnersoft Gmbh., Softwareentwickler (C/C++)
2005 – 2014	:	JAW – WS&TS - Innermanzing, Päd. Mitarbeiter
2015 – Heute	:	JAW – WV – Innermanzing, Päd. Mitarbeiter, Schwerpunkt Terminisierung und Betreuung von Arztbesuchen

Projektarbeiten

1990	:	Visualisierung von animierten Lissajous Figuren
1992 - 1993	:	Konzeption und Realisierung eines miniaturisierten Industriecomputersystems (Maturaprojekt)
1996	:	WWW - Lehrveranstaltungsanmeldesystem
1997	:	Perl-Robot zum rekursiven Download von elektronischen Büchern im WWW
1998 - 1999	:	LVA Evaluierung, WWW Erfassungs- und Auswertungssystem für Fragebögen
2000 - 2002	:	NetMusic, Konzept und Realisierung eines Systems zum kooperativen Austausch musikalischer Information
2013 – Heute	:	Design und Implementierung einer interaktiven Programmiersprache (Masterarbeit)

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Ort, Datum

Unterschrift