Schriftliche Ausarbeitung der Besonderen Lernleistung im Fach Informatik

Helmholtz-Gymnasium Essen

Für das Abitur 2019

Void — Entwicklung und Gestaltung einer eigenen Programmiersprache

Autor:
Julian Koch

Fachlehrer:
Dennis Großkamp

31. März 2019

Inhaltsverzeichnis

1 Vorwort **2**

2 Einleitung **2**

3 Die Sprache **3**
3.1 Das Typsystem . . . 3
3.2 Die Grundtypen in Void . . . 3
3.3 Generelle Notationen . . . 4
3.4 Funktionen in Void . . . 5
3.4.1 Termination der Argumentenliste . . . 6
3.5 Expression-ception . . . 6
3.6 Scoping . . . 6
3.7 Exceptions . . . 7
3.8 Control-Flow . . . 8
3.8.1 If-Abfragen . . . 8
3.8.2 While-Schleifen . . . 8
3.9 Punkt-Notation . . . 9
3.10 Implizite Typumwandlung—Type Coercion . . . 10
3.10.1 Vergleiche . . . 10
3.11 with-Konstruktionen . . . 10
3.11.1 class-Funktionen . . . 10
3.12 Standardbibliothek . . . 11

4 Compiler-Design **12**
4.1 Tokenisierung . . . 12
4.2 Syntaktische Analyse . . . 13
4.3 Interpretation . . . 14
4.3.1 Visitor Design Pattern . . . 14
4.3.2 Stacks im Interpreter . . . 15
4.3.3 Interpreter-Modus . . . 16

5 Entstehungsprozess **19**

1 Vorwort

Programmiersprachen sind ein grundlegendes Werkzeug für die technologische Progression der Menschheit. Jedoch ist nur ein sehr geringer Anteil von Software-Erstellern an der Entwicklung einer Programmiersprache beteiligt.

2 Einleitung

Void orientiert sich hauptsächlich an JavaScript. Die Sprache verfolgt mehrere Ziele, in absteigender Wichtigkeit:

1. **Zugänglichkeit & Lesbarkeit.** Void ist speziell für Anfänger eine geeignete Sprache, da viele unnötige syntaktische Elemente weggelassen und eine Vielfalt an traditionell wortreichen Konstruktionen deutlich vereinfacht sind.

2. **Universelle Einsetzbarkeit.** Trotz der ersten Priorität ist Void für eine große Anzahl an Einsatzmöglichkeiten geeignet. Der zugrundeliegende Compiler und Interpreter ist in der JVM-Sprache Kotlin geschrieben und daher auf allen gängigen Systemen ausführbar.

3. **Erweiterte Möglichkeiten.** Viele Features, die in Sprachen wie Java nicht oder nur über Umwege implementiert sind, werden von Void nativ unterstützt. In etwa sind Funktionen *first-class citizens*, können also beispielsweise als Argumente anderen Funktionen übergeben werden.

4. **Praktikabilität in *Prototyping*-Szenarien.** JavaScript ist besonders aufgrund dessen Eignung für das schnelle Testen und Erproben von ersten Ideen beliebt. Wie in dieser Scripting-Sprache kann in Void ein Programm aus nur einer Zeile bestehen.

5. **Erweiterbarkeit.** Das Kotlin-Backend ist modular konzipiert und erlaubt die einfache Extension der Standardbibliothek mit nativen Ausdrücken.

3 Die Sprache

Im Folgenden wird Void als Ganzes vorgestellt, und es wird auf einige Design-Entscheidungen eingegangen, um einen Einblick in den Entstehungsprozess der Sprache zu gewähren.

3.1 Das Typsystem

Void ist eine dynamische, schwach-typisierte Sprache. Das bedeutet, der Compiler beinhaltet keinen Typ-Checker; eine Variable ist nicht mit einem Typen assoziiert, und Typ-Fehler werden erst zur Laufzeit erkannt.

Diese Entscheidung bietet mehrere Vor- und Nachteile. Einerseits ist ein dynamisches Typsystem eine traditionelle Fehlerquelle in Sprachen wie Java; andererseits erlaubt es eine hohes Niveau an Flexibilität und erleichtert das schnelle *Prototyping* von Ideen, ohne die Erstellung eines Typ-Frameworks vorauszusetzen.

Aufgrund des Ziels Voids, so einsteigerfreundlich wie möglich zu sein, überwiegen hier die Vorteile deutlich; neue Programmierer müssen sich nicht mit einem Typsystem "herumschlagen" und können sich auf die Bearbeitung der jeweiligen Aufgabe konzentrieren.

3.2 Die Grundtypen in Void

Ungeachtet des dynamischen Typsystems beinhaltet Void eine festgelegte Menge an Typen. Diese werden im Folgenden näher beleuchtet.

- **Undefined.** Dieser Typ hat nur einen möglichen Wert zur Laufzeit: *undefined*. Allerdings hat eine Variable zu keinem Zeitpunkt diesen Wert, da er praktisch die Nichtexistenz eines Wertes bedeutet. Das Setzen einer Variable auf *undefined* ist also als Löschen dieser zu verstehen. Ebenfalls gibt das "Anfragen" einer unbekannten Variable diesen Wert zurück.

- **Bool.** Ähnlich zu anderen Sprachen kann eine Variable des Typs Bool nur die Werte *True* und *False* annehmen (die Kapitalisierung ist zu beachten!).

- **Num.** Im Kontrast zu vielen Sprachen, einschließlich Java und JavaScript, unterstützt Void nur einen Typ, der für das Speichern einer Zahl geeignet ist: Num. Er kann positive sowie negative Werte in den reellen Zahlen annehmen (mit einer Präzision von 64 Bit—dies entspricht einem *double* in Java).

- **Str.** Der Str-Typ speichert eine Zeichenfolge beliebiger Länge. Werte können entweder mithilfe von Str-Literals in der Form `"Text"` oder über Enum-Literals in der Form `\Text` erzeugt werden. Ein Enum-Literal

ist ein Backslash gefolgt von einem einzigen validen Identifier und ist äquivalent zu `\Text` (die Nützlichkeit zeigt sich beispielsweise bei der *for*-Funktion).

- **Obj.** Werte diesen Typs können als Mapping von Namen zu weiteren Werten beliebiger Typen interpretiert werden. Der Typ ist äquivalent zu JavaScripts Objekten. Ein Obj kann mithilfe von Obj-Literals erzeugt werden:

```
[
    myText: "Hallo, Welt!"
    pi: 3.1415
]
```

  Es ist zu beachten, dass die Werte (im Gegensatz zur traditionellen Form in JavaScript) nicht durch Kommata getrennt werden, sondern durch Neuzeilen.

- **Vec.** Dies ist der standardmäßige Listen-Typ von Void. Er speichert eine geordnete, beliebig große Menge von Werten und kann durch Vec-Literals erzeugt werden:

```
("a"|"b"|3.1415)  ; 3-dimensionaler Vektor
()                ; leerer Vektor
Vec "x"           ; 1-dimensionale Vektoren werden
                  ; über die Funktion Vec erstellt
```

  Vec stellt viele nützliche Funktionen bereit, die später näher betrachtet werden.

- **Function.** Alle Funktionen werden durch diesen Typen repräsentiert. Er enkapsuliert Name, Parameterliste und Körper.

- **Block.** Dies ist ein ausschließlich intern genutzter Typ und repräsentiert einen in geschweiften Klammern eingeschlossenen Code-Block, welcher zu einem späteren Zeitpunkt ausgeführt werden kann.

3.3 Generelle Notationen

In Void ist jede Abfrage einer Variable durch einen Hashtag (`#`)–Präfix gekennzeichnet. Dies erhöht die allgemeine Lesbarkeit deutlich, da sofort ersichtlich ist, ob eine Funktion aufgerufen oder eine Variable gelesen wird.

Ein Funktionsaufruf wird durch den Namen der Funktion eingeleitet, gefolgt von beliebig vielen Identifiern und Expressions. Unterschiedlich zu vielen Sprachen müssen diese Argumente nicht in Klammern eingeschlossen werden, was die Lesbarkeit von Methodenaufrufen deutlich erhöht. Zum Beispiel besteht das traditionelle *Hallo, Welt!*–Programm in Void aus nur einer Zeile:

```
print "Hallo, Welt!"
```

3.4 Funktionen in Void

In Void werden Funktionen wie folgt definiert:

```
fun <name>(<parameter>) <koerper>
```

Der `<name>` ist optional und besteht aus genau einem Identifier. Der `<koerper>` ist entweder ein Statement oder eine in geschweiften Klammern eingeschlossene, Neuzeilen-separierte Liste von beliebig vielen Statements. Die `<parameter>` setzen sich aus einer (möglicherweise leeren) Komma-separierten Liste von Parametern zusammen, die im Folgenden konkret definiert werden.

Void unterstützt zwei verschiedene Arten von Parametern: *Normale* und *Enum*-Parameter. Gemeinsam ist beiden Arten die anfängliche Notation `<externerName> #<internerName> [= <standardwert>]`, wobei der externe Name eine beliebig lange Leerzeichen-getrennte Liste von Namen und der interne Name genau ein Name ist. Der Standardwert ist eine beliebige Expression, die evaluiert wird, wenn dem Parameter kein Wert übergeben wird.

Folgend wird zur näheren Erläuterung dieser Definition beispielhaft die Funktion *summe* definiert:

```
fun summe(von #a, und #b = 0) return #a + #b

summe von 1 und 2.3  ; 3.3
summe von 4          ; 4
```

Hier werden zwei normale Parameter definiert, welche beide einen externen und einen internen Namen besitzen. Der externe Name muss beim Aufruf der Funktion immer mit angegeben sein; der interne ist der Name einer funktionslokalen Variable, in welcher der übergebene Wert gespeichert wird.

Wird kein externer Name angegeben, ist der effektive externe Name derselbe wie der interne. Soll kein externer Name definiert werden, muss dies durch einen einzelnen Unterstrich (_) explizit gekennzeichnet werden.

Enum-Parameter sind spezielle Parameter, die nur einen Wert aus einer vordefinierten Liste von Identifiern annehmen können. Hier ist der interne Name von dem Schlüsselwort `of` und einer Vertikalstrich-separierten Liste von Identifiern gefolgt. Dies entspricht folgender beispielhafter Form:

```
fun berechne(
     _ #modus of summe|produkt|durchschnitt,
     von #a, und #b
)
     return if (#modus == \summe)         #a + #b
            else if (#modus == \produkt) #a * #b
            else                          (#a + #b) // 2
            ; Division erfolgt durch ein Doppelslash

berechne summe   von 2 und 3  ; 5
berechne produkt von 2 und 3  ; 6
berechne durchschnitt von 2 und 3  ; 2.5
```

Hinweis: Innerhalb der Funktion nimmt die `#modus`-Variable einen Wert des Typs Str an und muss somit mit einem Str, in diesem Fall einem Enum-Str-Literal, verglichen werden.

3.4.1 Termination der Argumentenliste

Standardmäßig werden alle Identifier und Expressions, die einem initialen Funktionsaufruf folgen, diesem als Argumente angehängt. Dieses Verhalten kann durch ein Ausrufezeichen unterbunden werden:

```
fun quadrat(#von) return #von * #von

quadrat von 3 + 1     ; (3+1)^2 = 16
(quadrat von 3) + 1   ; 3^2 + 1 = 10
quadrat von 3! + 1    ; 3^2 + 1 = 10
```

3.5 Expression-ception

In Void sind fast alle traditionellen Statements tatsächlich Expressions, geben also einen Wert zurück. Beispielsweise wird folgend eine Funktion definiert, die rekursiv die Fakultät einer natürlichen Zahl ausrechnet:

```
fun fakultaet(von #n) return
     if (#n > 0) #n * fakultaet von #n-1
     else 1

fakultaet von 5   ; 5! = 120
```

Hier ist ersichtlich, dass `if` eine Expression ist, sofern der Körper der If-Abfrage nicht in geschweiften Klammern eingeschlossen ist.

3.6 Scoping

Wird der *Flow* eines Programms zur Laufzeit geändert, etwa durch *Control-Flow*–Strukturen oder das Aufrufen einer Funktion, erstellt Void intern ein neues Scope. Dieses kann als Umgebung und Speicherort für lokale Variablen interpretiert werden. Wird in einem Scope eine Variable erstellt und auf einen Wert (nicht `undefined`) gesetzt, ist diese nur innerhalb dieses und aller weiteren von diesem aus erstellten Scopes sichtbar. Sobald die Programmexekution das Scope verlässt, wird die Variable unsichtbar und ihr Wert gelöscht.

Allerdings können mit standardmäßigen Variablenzuweisungen Werte von bereits existierenden Variablen überschrieben werden. Nutzt eine Funktion also intern eine Variable `x`, ist eine mögliche Fehlerquelle, dass vor Aufruf der Funktion eine Variable desselben Namens erstellt worden ist. In diesem Fall wäre der Wert dieser Variable "verloren".

Um dies zu umgehen, können Variablenzuweisungen als lokal gekennzeichnet werden:

```
fun foo() {
     local #x = 2
     ; #x hat den Wert zwei
}
#x = 1
foo
; #x hat den Wert 1
```

Nach einmaliger Variablendeklaration mit `local` ist die weitere Verwendung des Schlüsselwortes nicht mehr nötig. Hier wird davon gesprochen, dass die lokale Variable die globale *schattiert* (engl. "shadows").

3.7 Exceptions

Wie JavaScript und Java bietet Void die Möglichkeit, in speziellen Fällen die Exekution des Programms frühzeitig zu beenden. Dies wird mithilfe von *Exceptions* ermöglicht.

Eine Exception ist ein beliebiger Wert, der durch das Schlüsselwort `throw` *geworfen* wird. Hiernach bricht die Ausführung des Codes sofort ab, und die Exception wird über alle Methodenaufrufe propagiert, bis das Programm völlig unterbrochen oder die Exception *gefangen* wird.

Um eine Exception zu fangen und so wieder Kontrolle über die Programmausführung zu erlangen, kann ein `try-catch`–Konstrukt genutzt werden. Der Code, in welchem eine Exception geworfen werden kann, steht innerhalb von geschweiften Klammern, die dem Schlüsselwort `try` folgen. Die abschließende Klammer wird von `catch`, einem Hashtag, Variablennamen und einem weiteren Code-Block gefolgt.

Zum Beispiel wird folgend eine Funktion `sicherPositiv` definiert, die ein Argument entgegennimmt und eine Exception genau dann wirft, wenn der numerische Wert des übergebenen Wertes null oder negativ ist. Diese Funktion wird mit einem negativen Argument aufgerufen, die Exception gefangen und auf der Konsole ausgegeben.

```
fun sicherPositiv(_ #x)
     if (#x <= 0) throw "Wert " + #x + " ist nicht positiv!"

try sicherPositiv(-4)
catch #ex {
     print #ex  ; "Wert -4 ist nicht positiv!"
}
```

Wird eine Exception mithilfe der Funktion `Exception` erstellt, wird dieser der aktuelle *Stack Trace* angehangen; ein Vektor, der die Namen aller Scopes umfasst, in welchen sich die Programmexekution momentan befindet. Optional kann der Exception mit dem Parameter `withMessage` eine Nachricht hinzugefügt werden, die in dem Member `message` gespeichert wird.

```
fun c() throw Exception withMessage "Hallo, Welt!"
fun b() c
```

```
fun a() b

try a
catch #ex print #ex.#message   ; "Hallo, Welt!"
```

3.8 Control-Flow

Control-Flow bezeichnet die Menge von Möglichkeiten, die eine Sprache zum Steuern bzw. Leiten der Programmexekution bereitstellt. Void bietet hier zwei grundlegende Strukturen: `if` und `while`.

3.8.1 If-Abfragen

In Void folgen If-Abfragen einer sehr "lenienten" (nachsichtigen) Syntax. Eingeleitet werden sie durch das Schlüsselwort `if`, worauf die Bedingung folgt. Diese ist eine einzelne Expression und muss nicht in Klammern eingeschlossen werden.

Hierauf folgt der Körper, welcher entweder aus einem einzelnen Statement oder einer in geschweiften Klammern eingeschlossenen beliebig langen Liste von Statements besteht. Es ist jedoch zu beachten, dass, sollte die Bedingung nicht in Klammern stehen, der Körper nicht in derselben Zeile beginnen darf, es sei denn, er ist in geschweiften Klammern eingeschlossen. Hierdurch werden unleserliche Konstruktionen größtenteils verhindert.

Abschließend folgt optional das Schlüsselwort `else` und ein weiterer Körper, den bereits beschriebenen Regeln entsprechend. Es existiert keine "spezielle" `if-else`–Konstruktion; der Körper des `else`-Teils kann ebenfalls eine If-Abfrage sein.

Beispielsweise wird im Folgenden der Kontostand (bzw. der Wert der `#kontostand`-Variable) geprüft.

```
if #kontostand >= 10000 {
     print "Du bist reich!"
} else if #kontostand >= 0 {
     print "Dein Kontostand ist positiv!"
} else {
     print "Du bist um " + #kontostand + "€ verschuldet!"
}
; Oder:
print "Der Kontostand ist: " +
     (if (#kontostand >= 0) "positiv" else "negativ")
```

3.8.2 While-Schleifen

While-Konstruktionen erlauben das wiederholte Ausführen von Code basierend auf einer Bedingung. (In Void werden For-Schleifen nicht nativ, sondern mittels einer Funktion unterstützt.)

Das `while`-Schlüsselwort wird wie bei der If-Abfrage von der Bedingung gefolgt, die vor jeder Exekution des Codes abgefragt wird. Hiernach steht der Körper, denselben Regeln der If-Abfrage entsprechend.

Letztlich kann die Schleife optional von einer `else`-Konstruktion gefolgt werden; dieser Code wird ausgeführt, wenn der Schleifenkörper kein einziges Mal aufgerufen worden ist.

```
#v = ("a"|"b"|"c")
#idx = 0
while #idx < #v.dim! {
                    ; Ohne das ! würde der Code-Block als
                    ; Argument der dim-Funktion interpretiert
                    ; werden
      print "Element " + #idx + ": " + #v.(#idx)
      #idx = #idx + 1
} else {
      print "Der Vektor ist leer."
}
```

3.9 Punkt-Notation

Alle Typen definieren mehrere innere Variablen, sogenannte *Member*, die von diesem abhängen. Bei Num und Str sind dies vordefinierte Methoden; in Objekten können vom Nutzer festgelegte Namen-Werte–Paare enkapsuliert werden.

Die Syntax, um einen Member abzufragen bzw. aufzurufen, ist bei allen Wertetypen gleich: Auf den äußeren/einschließenden Wert folgt ein Punkt und hierauf ...

- `#xxx`, um den Wert des Members namens `xxx` abzufragen.
- `xxx (...)`, um die Member-Funktion namens `xxx` mit den Argumenten `(...)` aufzurufen.
- `0`, um den Wert des Members mit dem Index `0` abzufragen (hier kann jede nichtnegative ganze Zahl eingesetzt werden).
- `(...)`, um den Wert des Members
  - mit dem Index, zu dem die eingeklammerte Expression evaluiert wird, sofern dieser Wert ein Num ist, bzw.
  - mit dem Namen, der dem Str-Wert des Resultats der eingeklammerten Expression entspricht,

  abzufragen.

Sollte ein Member den angefragten Bedingungen entsprechend nicht gefunden werden, wird `undefined` zurückgegeben.

3.10 Implizite Typumwandlung—Type Coercion

Funktionen wie `print` sind dafür ausgelegt, mit einem Str-Argument aufgerufen zu werden. Tatsächlich können jedoch auch Werte beliebigen Typs übergeben werden, welche Void innerhalb der Funktion implizit in Strings umgewandelt.

Dieses Prinzip trifft auf viele primitive Typen zu. Im Folgenden werden einige dieser impliziten Typumwandlungen aufgezeigt.

```
convert.toBool -3.1415  ; True
!!0  ; False  implizit durch die doppelte Negation

convert.toString (10|20|"z")  ; "(10|20|z)"
""+[hallo: \welt]  ; "[hallo: welt]"  implizit durch Addition

convert.toNum True  ; 1
+False  ; 0  implizit durch das unäre Plus
```

3.10.1 Vergleiche

Die Vergleichsoperatoren `<,>,<=,>=` haben immer eine implizite Typumwandlung zu Num zur Folge. Bei `==` wird einer der beiden Werte nur dann umgewandelt, wenn sie einen unterschiedlichen Typ haben; in diesem Fall wird der Str-Wert verglichen.

Hieraus resultiert, dass Expressions wie `3 == "3"` den Wahrheitswert `True` zurückgeben. Um dieses Verhalten zu umgehen, kann, ähnlich zu JavaScript, eine *safe comparison* (sicherer Vergleich) durchgeführt werden, bei welchem immer `False` zurückgegeben wird, wenn die Typen der Werte nicht übereinstimmen.

3.11 with-Konstruktionen

`with` ist in Void eine elementare und universell einsetzbare Funktion. Sie existiert in zwei Varianten, welche hier beide erläutert werden.

`with (#wert) do {...}` führt den in geschweiften Klammern eingeschlossenen Code-Block aus, wobei `#me` den Wert der in normalen Klammern eingeschlossenen Expression annimmt. Dasselbe wird durch `#wert.with {...}` erreicht.

Dies ermöglicht eine kompaktere und in vielen Fällen klarere Syntax. So können beispielsweise einem Objekt viele Werte hinzugefügt werden, ohne das die Objektvariable jedes Mal referenziert wird.

3.11.1 class-Funktionen

Folgende Definitionen sind äquivalent:

```
fun Car(withSpeed #speed = 60) return [
     speed: #speed
```

```
     stop: fun () my #speed = 0
]

class fun Car(withSpeed #speed = 60) {
     my #speed = #speed
     my #stop = fun () my #speed = 0
}
; Das gleiche wie:
;  fun Car(...) return [].with {...}
```

Die `class fun`–Notation ermöglicht das Ausführen von beliebigem Code während der "Instanziierung" eines Objekts.

3.12 Standardbibliothek

Void bietet standardmäßig viele nützliche Funktionen. Einige dieser werden im Folgenden näher betrachtet.

```
#vec = (10|20|30)

print #vec.dim
; Dimension des Vektors (Größe)

#vec.forEach #print
; Führe die Funktion für alle Werte in #vec aus

call #print with "Hi"
; Rufe #print auf mit Argument "Hi".
; with-Parameter akzeptiert auch Vektor
;  mit mehreren Argumenten

localset "vecCopy" to #vec
; Setze lokale Variable mit dynamischem Namen

for each \item in #vec do {
     print #item
}

; Die for-Funktion ist wie folgendermaßen implementiert:
fun for(each #var, in #vec, do #action) {
     ; Lokale Variable; überschreibt keinen
     ; bereits existierenden #index
     local #index = 0
     while #index < #vec.dim! {
          localset #var to #vec.(#index)
          call #action
          #index = #index + 1
     }
}
```

4 Compiler-Design

Void ist in der JVM-Sprache Kotlin implementiert.

Das Ausführen von Code ist in drei Phasen eingeteilt: Tokenisierung, Syntaktische Analyse und Interpretation.

4.1 Tokenisierung

In der ersten Phase, auch *Lexing* genannt, wird die auszuführende Datei Zeichen für Zeichen gelesen und in einen ausgehenden Fluss von *Tokens* übersetzt. Diese beinhalten folgende Informationen:

- **Lexem.** Die Zeichenkette, aus der dieser Token generiert wurde.
- **Typ.** Der Typ eines Tokens gibt Auskunft über die Natur des zugrundeliegenden Lexems. Beispielsweise beschreibt ein Token des Typs `LIT_BOOL` ein Bool-Literal, also `True` oder `False`, während `K_FUN` das Schlüsselwort `fun` meint. Eine Liste aller Typen findet sich in der Klasse `TokenType`.
- **Position.** Die Zeile und Spalte, wo das Lexem im Originaltext erscheint.

Hier werden nur wenige Fehler entdeckt, wie illegale Symbole oder nicht abgeschlossene String-Literals.

Der Lexing-Algorithmus ist in der Klasse `Lexer` implementiert. Diese ist mit einer Liste aller Token-Typen parametrisiert. Bei Aufruf der `readToken`-Methode wird erst Whitespace (d.h. Leerzeichen, Tabs, Neuzeilen, etc.) entfernt. Wird hierbei ein Neuzeilen-Symbol gefunden, gibt die Methode einen entsprechenden Token des Typs `EOL` zurück.

Hierauf liest der Lexer die eingehende Datei Zeichen für Zeichen, während er kontinuierlich eine Liste mit allen Token-Typen aktualisiert, die auf die bereits gelesene Zeichenkombination verweisen könnten. Dies geschieht solange, bis diese Liste leer ist (unter der Bedingung, dass sie zu einem Zeitpunkt nicht leer war). Dieses Konzept heißt *Maximal Munch* und ist der Grund, warum `hallo` nicht als fünf verschiedene Identifier erfasst wird.

Nun wird das zuletzt gelesene Zeichen wieder "zurückgeschoben". Der zurückzugebende Token hat den Typen, der Element der Liste der passenden Typen und außerdem als erstes in der Liste aller Typen definiert ist.

Formal wird dies folgend mathematisch beschrieben. Es seien Z das Alphabet der einzulesenden Zeichen; n die Zahl der bereits eingelesenen Zeichen; $\vec{s} \in Z^n$ der Vektor der eingelesenen Zeichen; T die Menge aller Token-Typen; $i : T \to \mathbb{N}$ eine injektive Abbildung mit $i^{-1}(i(t)) = t$ für alle $t \in T$, die jedem Token-Typen einen einzigartigen Index zuordnet; $b : (T, Z^n) \to \{0, 1\}$ eine Wahrheitsfunktion, die aussagt, ob aus einer Zeichenkette ein Token eines bestimmten Typs generiert werden kann;

$f \in \{0, 1\}$ genau dann 1, wenn zu mindestens einem Zeitpunkt galt: $M \neq \emptyset$; und $M = \{t \mid t \in T \wedge b(t, \vec{s}) = 1\} \subseteq T$ die Menge aller Token-Typen, aus denen ein Token mit den bereits eingelesenen Zeichen generiert werden kann.

Anfangs sind $n = 0$, $\vec{s} = \vec{0}$, $f = 0$ und $M = \emptyset$. Iterativ werden nun n inkrementiert und $\vec{s}$, M und f aktualisiert, bis gilt: $f = 1 \wedge M = \emptyset$. Hierauf wird der vorausgegangene Zustand der Variablen wiederhergestellt, und es wird ein Token mit dem Typen $t \in T$ zurückgegeben, für den gilt: $t \in M \wedge i(t) = \min\{i(\bar{t}) \mid \bar{t} \in M\}$.

In Abb. 1 wird beispielhaft das Lexing des Schlüsselworts `fun` dargestellt. Zur Vereinfachung gilt: $\vec{s} = \text{“}abc\ldots\text{”} = (a\ b\ c\ \ldots)^T$.

$$T = \{\ \text{K\_FUN},\ \text{IDENT}\ \}$$

$t \in T$	$i(t)$	$b(t, \vec{s})$
K_FUN	1	$\vec{s} =$ “fun”
IDENT	2	1

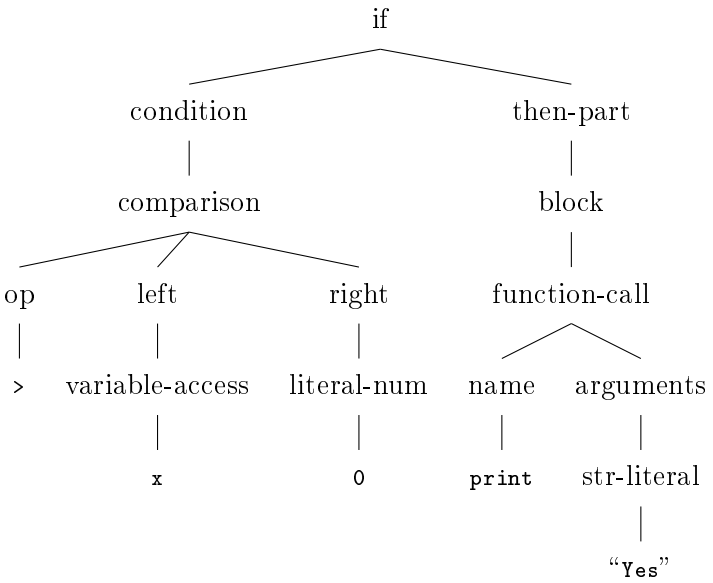
n	$\vec{s}$	M	f	$i^{-1}(\min\{i(\bar{t}) \mid \bar{t} \in M\})$
0	“”	$\emptyset$	0	—
1	“f”	{IDENT}	1	IDENT
2	“fu”	{IDENT}	1	IDENT
3	“fun”	{K_FUN, IDENT}	1	K_FUN

Abbildung 1: Beispielhaftes Lexing des Schlüsselworts `fun`. Da `K_FUN` vor `IDENT` definiert ist (das kleinere i aufweist), wird der richtige Typ zurückgegeben.

4.2 Syntaktische Analyse

Zweitens wird aus den generierten Tokens ein *Abstract Syntax Tree* (AST) erstellt. Dieser speichert alle verfügbaren Informationen über die Struktur des Programms. Folgend wird der AST für ein Beispielprogramm dargestellt:

```
if #x > 0 {
      print "Yes"
}
```



Der AST ist eine Baumstruktur von Kotlin-Klassen. Jedes Baumelement erbt von `VlElement`; dieses Interface definiert Methoden und Variablen wie Referenzen auf das übergeordnete (*parent*) und die untergeordneten (*children*) Elemente.

Jede Klasse in dieser Interface-Struktur besitzt eine konkrete Implementation. Diese folgt der Namensgebung: aus `VlXXX` wird `VlXXXImpl`. Jede Implementation beinhaltet ausschließlich nicht-finale Variablen und eine Factory-Methode mit dem Namen des Interfaces und einem Lambda-Argument. Dies entspricht folgender beispielhafter Form:

```
inline fun VlComparison(init: VlComparisonImpl.() -> Unit)
     = VlComparisonImpl().apply(init)

// Aufruf im Parser:
val comp = VlComparison {
     left = ...
}
```

In dieser Phase werden alle Fehler gefunden, die noch vor der Laufzeit entdeckt werden können, eingeschlossen nicht balancierter Klammern, dem Auslassen der Funktionsparameterliste, etc.

4.3 Interpretation

Der Interpreter ist für das tatsächliche Ausführen eines in Void geschriebenen Programms verantwortlich.

4.3.1 Visitor Design Pattern

Der Visitor Design Pattern (VDP) ist ein auf Baumstrukturen anwendbares Programmiermuster, welches das Traversieren des Baums deutlich verein-

facht.

Er schreibt vor, dass die Basisklasse der Baumelemente eine Methode `accept(Visitor)` deklariert. Des weiteren existiert ein Interface `Visitor`, welches für jede Baumklasse `VlXXX` die Methode `visitXXX` definiert. Diese Methoden werden von den jeweiligen Implementationen der `accept`-Methode aufgerufen.

Dies ermöglicht es praktisch, den bestehenden Klassen mit typsicherer Funktionalität zu erweitern, indem `Visitor` implementiert wird. Beispielsweise ist Voids Interpreter, die Klasse `Interpreter`, eine solche Implementation. Folgend werden beispielhaft die für einen Vergleich relevanten Code-Fragmente in vereinfachter Form betrachtet.

```
interface Visitor {
     fun visitComparison(comp: VlComparison)
}

interface VlElement {
     fun accept(v: Visitor)
}

interface VlComparison : VlElement
class VlComparisonImpl : VlComparison {
     override fun accept(v: Visitor) =
          v.visitComparison(this)
}

class Interpreter : Visitor {
     override fun visitComparison(comp: VlComparison) {
          ...
     }
}
```

4.3.2 Stacks im Interpreter

Der Scope-Stack. Der Kernel aller Betriebssysteme arbeitet auf fundamentaler Ebene mit einem Stack, um die Funktionalität von Funktionsaufrufen zu gewährleisten. Ein Stack ist eine dynamische, lineare Datenstruktur, die nach dem LIFO-Prinzip (*First In, Last Out*) einzelne *Frames* speichert. Ein Frame enkapsuliert alle lokalen Variablen in einer Funktion sowie eine Referenz auf die als nächstes auszuführende Instruktion. Wird eine Funktion wieder verlassen, werden diese Daten gelöscht, und die Programmexekution wird von der im übergeordneten Frame gespeicherten Instruktion fortgesetzt.

Dieser Stack findet sich auch im Interpreter von Void: der `scopeStack` ist eine Stack-Struktur, die `Scopes`, also Variablenumgebungen und Scopenamen, speichert.

Ein neues Scope wird nicht nur bei Funktionsaufrufen, sondern auch beispielsweise für If-Abfragen erstellt. Es ist somit möglich, lokale Variablen

innerhalb eines If-Körpers zu definieren, die nach Verlassen der If-Abfrage automatisch gelöscht werden. Dieses Verhalten wird in Abb. 2 gezeigt.

```
; Die Kommentare stellen den Scope-Stack dar;
;  die eingeklammerten Werte sind in jeweils
;  einem Scope gespeichert.
; Das aktuelle Scope ist immer das rechte.
;
fun foo(_ #x) {
     #b = 2
     ; [(a,foo), (x,b)]
     if #x {
          #c = 3
          ; [(a,foo), (x,b), (c)]
     }
}
#a = 1
foo True
; [(a,foo)]
```

Abbildung 2: Demonstration von Scopes in Void

Der Value-Stack. Der Void-Interpreter definiert einen weiteren Stack, um intermediäre Ergebnisse innerhalb komplexer Instruktionen zu speichern. Dieser Stack heißt Value-Stack (VS) und ist vom Typen `Value` (der Klasse aller Werte). Bei einer Summe werden etwa die linke und rechte Seite ausgeführt, welche beide ihren Wert auf den VS legen. Hiernach werden diese zwei Werte vom VS genommen, addiert und das Ergebnis wieder dem VS hinzugefügt, um weiter verwendet werden zu können.

4.3.3 Interpreter-Modus

Der Interpreter befindet sich zu jedem Zeitpunkt in einem von zwei Modi:

- Im **Statement-Modus** (SM) werden die einzelnen Instruktionen ausgeführt, ohne dass ein Wert auf den VS gelegt wird.
- Im **Expression-Modus** (EM) führt das Ausführen *jeder* Instruktion zu dem Hinzufügen genau einen Wertes zu dem VS.

Trifft der Interpreter zum Beispiel auf ein VlStringLiteral, wird dessen Wert nur dann auf den VS gelegt, wenn der Expression-Modus aktiv ist; andernfalls wird eine Warnung ausgegeben. Abb. 3 zeigt die Implementation in Kotlin.

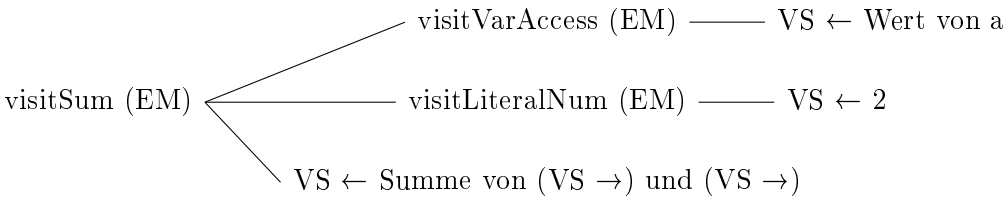
Der Interpreter startet im Statement-Modus und wechselt, wenn das Ausführen einer Instruktion durch einen zu berechnenden Wert bedingt ist.

```
override fun visitExpr_literal_string(
          stringLiteral: VlStringLiteral
) {
     if (exprMode) {
          valueStack.push(Value.Str(stringLiteral.value))
     } else {
          logWarning("Dangling literal string: " +
               "\"${stringLiteral.value}\"")
     }
}
```

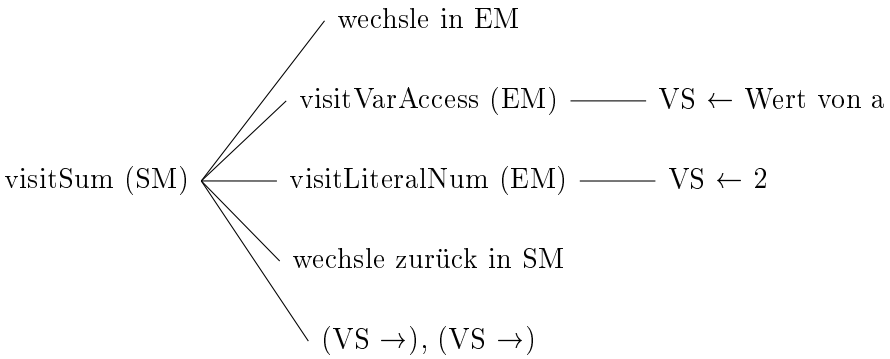
Abbildung 3: Evaluation eines String-Literals im Interpreter

Im Folgenden werden die einzelnen Zustände des Interpreters für das Beispielprogramm `#a + 2` dargestellt, wobei (VS ←x) den Wert x auf den VS legt und (VS →) den letzten Wert vom VS nimmt.

Startet der Interpreter im Expression-Modus, wird die Summe der beiden Werte dem VS hinzugefügt:



Startet er jedoch im Statement-Modus, bleibt der VS letztendlich leer:



Dieses Verhalten ist in Kotlin wie in Abb. 4 implementiert.

TODO Error Recovery.

```
override fun visitExpr_sum(sum: VlSum) {
     val inExprMode = exprMode
     exprMode = true
     val left = eval(sum.left)
     val right = eval(sum.right)
     exprMode = inExprMode
     if (inExprMode) {
          if (sum.isMinus) {
               // a - b ist die Differenz der Zahlenwerte
               val num = left.numberValue - right.numberValue
               valueStack.push(Value.Num(num))
          } else if (left is Value.Str || right is Value.Str) {
               // a + b ist String-Verkettung, wenn
               // a oder b ein Str ist
               val str = left.stringValue + right.stringValue
               valueStack.push(Value.Str(str))
          } else {
               // a + b von zwei Nicht-Strings ist
               // die Summe der Zahlenwerte
               val num = left.numberValue + right.numberValue
               valueStack.push(Value.Num(num))
          }
     }
}
```

Abbildung 4: Implementation der Summen-Interpretation

5 Entstehungsprozess

Meine Erfahrungen während der Entstehung, initiale Gedankengänge, etc.
TODO