

# Workshop Programming Languages: Nim

Niklas Peter

# Contents

<b>1</b>	<b>Einleitung</b>	<b>2</b>
1.1	Einleitung . . . . .	2
<b>2</b>	<b>Vorstellung der Sprache</b>	<b>3</b>
2.1	Entstehung . . . . .	3
2.1.1	Einflüsse anderer Sprachen . . . . .	3
2.1.2	Syntax . . . . .	3
2.1.3	Allgemeines . . . . .	4
2.2	Nutzung heute . . . . .	4
2.3	Programmierungsparadigmen . . . . .	5
2.3.1	Funktionale Programmierung . . . . .	5
2.3.2	Objektorientierte Programmierung . . . . .	5
2.3.3	Metaprogrammierung . . . . .	7
2.3.4	Foreign function interface . . . . .	7
2.3.5	Parallelisierung und Nebenläufigkeit . . . . .	8
2.4	Infrastruktur . . . . .	8
2.4.1	Kompilierer . . . . .	8
2.4.2	Entwicklungswerkzeuge . . . . .	9
2.4.3	Bibliotheken . . . . .	13
2.4.4	Dokumentation . . . . .	14
<b>3</b>	<b>Implementierung</b>	<b>15</b>
3.1	Herangehensweise . . . . .	15
3.2	Benutzte Bibliotheken . . . . .	17
3.2.1	os . . . . .	17
3.2.2	osproc . . . . .	17
3.2.3	strutils . . . . .	17
3.2.4	terminal . . . . .	17
3.3	Verwendete Elemente der Sprache . . . . .	18
3.4	Optimierungen . . . . .	18
<b>4</b>	<b>Fazit</b>	<b>19</b>
4.1	Tauglichkeit der Sprache zur Entwicklung einer <i>grep</i> Alternative	19
4.2	Zusammenfassung der Sprache . . . . .	19

# Chapter 1

## Einleitung

### 1.1 Einleitung

In diesem Bericht möchte ich die Programmiersprache Nim vorstellen. Ich werde zeigen, welche Merkmale und Eigenschaften sie hat und wie sie sich von anderen Programmiersprachen unterscheidet. Des Weiteren teile ich meine Erfahrungen mit der Verwendung von Nim in einem Programmierprojekt. Genauer gesagt die Verwendung zur Entwicklung eines simplen Klons des Kommandozeilen Werkzeugs **grep**, welches zur Suche bestimmter Zeichenketten in Dateien verwendet wird.

## Chapter 2

# Vorstellung der Sprache

### 2.1 Entstehung

Andreas Rumpf hat 2005 mit der Entwicklung der Sprache, die effizient, ausdrucksvoll und elegant sein sollte, angefangen. In 2008 hat er sein Projekt, damals noch Nimrod genannt, veröffentlicht. Mit Veröffentlichung der Version 0.10.2 im Dezember 2014 wurde Nimrod zu Nim umbenannt.<sup>1</sup> Am 23. September 2019 kam schließlich Version 1.0 raus. Zu diesem Zeitpunkt war die Sprache schon so weit fortgeschritten, dass eine gewisse Stabilität garantiert werden konnte.<sup>2</sup> Mittlerweile wird Nim von einem Team bestehend aus Andreas Rumpf und weiteren Freiwilligen weiter entwickelt. Die Programmiersprache ist quelloffen und wird unter der MIT Lizenz veröffentlicht.

#### 2.1.1 Einflüsse anderer Sprachen

Nim's Speichermanagement ist inspiriert von C++ und Rust. Es ist deterministisch und anpassbar durch Destruktoren und Move Semantics.<sup>3</sup> Nim kombiniert Python's leserliche Syntax, Ada's starkes statisches Typisierungssystem und Lisp's flexibles Macrosystem.<sup>4</sup>

#### 2.1.2 Syntax

Nim's Syntax gleicht der von Python. Das bedeutet, Codeblöcke werden durch Einrückung ausgedrückt. Viele Schlüsselwörter sind identisch zu denen von Python. So wird beispielsweise im Gegensatz zu anderen Programmiersprachen bei einer Verneinung einer Bedingung kein Ausrufezeichen sondern ein *not* verwendet. Anders als Python setzt Nim auf statische Typisierung. Nim ermöglicht allerdings einfache Typ Umwandlungen. Um verschiedene Programmierstile (zum Beispiel camelCase oder snake\_case)

---

<sup>1</sup><https://nim-lang.org/blog/2014/12/29/version-0102-released.html>

<sup>2</sup><https://nim-lang.org/blog/2019/09/23/version-100-released.html>

<sup>3</sup><https://nim-lang.org/>

<sup>4</sup><https://www.youtube.com/watch?v=-9SGIB946lw>

zu ermöglichen, ist Nim *style-insensitive*. Das bedeutet, zwei Bezeichner (zum Beispiel Variablennamen) mit unterschiedlicher Groß- und Kleinschreibung und mit oder ohne Unterstrich können als der selbe Bezeichner interpretiert werden, so lange der erste Buchstabe identisch ist. Dies ermöglicht eine Mischung an Stilen über Bibliotheken ohne Probleme.

Zusätzlich unterstützt Nim Uniform Function Call Syntax (UFCS). Dabei können Funktionen wie Methoden aus objektorientierten Sprachen benutzt werden. Hierzu wird der Empfänger als erster Parameter und die gegebenen Argumente als die restlichen Parameter verwendet. Beispiel:

$x = \text{multiply}(y, z)$  entspricht  $x = y.\text{multiply}(z)$ .

Das ist besonders nützlich, wenn man mehrere Funktionen aneinander ketten:

$x = x.\text{multiply}(y).\text{multiply}(z)$

### 2.1.3 Allgemeines

Nim ist wie Python eine imperative Sprache. Allerdings ist sie nicht wie Python auch eine Interpreter Sprache, sondern kompiliert zu C. Das bringt Vorteile bei der Performance gegenüber Python bei ungefähr gleichbleibender Einfachheit der Programmierung aufgrund der Syntax.

Ähnlich wie bei Rust, kann man sich bei der Definition einer Variable entscheiden, ob sie *immutable* sein soll, indem man statt dem Schlüsselwort *var* das Schlüsselwort *let* benutzt. Dadurch kann der Variable nur einmal ein Wert zugewiesen werden. Das ist dann wichtig, wenn man ungewollte Speicherzugriffe vermeiden möchte.

## 2.2 Nutzung heute

Aufgrund der einfachen Syntax eignet sich Nim hervorragend zum Shell Scripting. Aber auch für anspruchsvollere Projekte wie Frontend und Backend Entwicklung für Webdienste, Machine Learning oder sogar eingebettete Systeme ist Nim geeignet, da Nim zu C, C++ und JavaScript kompilieren kann.

Aufgrund der breiten Einsatzmöglichkeiten gibt es jede Menge Projekte, die in Nim geschrieben wurden. Im Folgenden eine kleine Auswahl an Projekten:

**Chroma** Eine Bibliothek um mit Farben und Farbräumen zu arbeiten. Einfache Transformierung von Farben.<sup>5</sup>

**dimscord** Ein Discord Bot<sup>6</sup>

**godot-nim** Nim Bindings für die Game Engine Godot<sup>7</sup>

---

<sup>5</sup><https://github.com/treeform/chroma>

<sup>6</sup><https://github.com/krisppurg/dimscord>

<sup>7</sup><https://github.com/pragmagic/godot-nim>

**httpx** Ein plattformübergreifender Web Server<sup>8</sup>

**netty** Eine UDP Verbindungsbibliothek für Spiele in Nim<sup>9</sup>

**nim-regex** Eine Bibliothek, um reguläre Expressionen zu parsen, kompilieren und auszuführen. Zur Laufzeit und zur Kompilierungszeit.<sup>10</sup>

**nitter** Ein Front End für Twitter.<sup>11</sup> Vermutlich eines der am weit verbreitetsten Projekte, welches in Nim geschrieben wurde.

**rod** Eine plattformübergreifende 2D und 3D Game Engine<sup>12</sup>

**Stardust** Ein Online Spiel im Browser<sup>13</sup>

## 2.3 Programmierungsparadigmen

### 2.3.1 Funktionale Programmierung

Nim unterstützt funktionale Programmierung. Es gibt First-Class-Funktionen. Diese können anderen Funktionen als Argument übergeben werden, von einer anderen Funktion als Wert zurückgegeben werden, einer Variable zugewiesen werden, in einer Datenstruktur gespeichert werden und zur Laufzeit erzeugt werden.<sup>14</sup>

Zur Kompilierzeit hat man die Möglichkeit Funktionen auf Side Effects zu überprüfen. Dazu markiert man Funktionen mit dem Schlüsselwort *func* oder man beschriftet eine Prozedur mit `{.noSideEffect.}`. Wenn solche Side Effects gefunden werden, weigert sich der Kompilierer zu kompilieren. Zu diesen Side Effects gehören Mutation, globaler Zugriff oder Modifikation, asynchroner Code, Threaded Code und Input/Output. In zukünftigen Versionen ist es geplant, Mutationen durch *ref* und *ptr* ebenfalls zu verbieten.<sup>15</sup> Es ist möglich, verschiedene Funktionen aneinander zu ketten. Dabei wird das Ergebnis der ersten Funktion an die nächste Funktion als Parameter weitergegeben.

Außerdem unterstützt Nim Pattern Matching.

### 2.3.2 Objektorientierte Programmierung

In erster Linie ist Nim eine imperative und funktionale Sprache. Trotzdem kann darüber hinaus auch objektorientiert programmiert werden. Allerdings mit eingeschränktem Umfang. Ein Objekt wird mit dem Schlüsselwort *type* initialisiert und ähneln *structs* aus der Programmiersprache C. In Ab-

---

<sup>8</sup><https://github.com/ringabout/httpx>

<sup>9</sup><https://github.com/treeform/netty>

<sup>10</sup><https://github.com/nitely/nim-regex>

<sup>11</sup><https://github.com/zedeus/nitter>

<sup>12</sup><https://github.com/yglukhov/rod>

<sup>13</sup><https://stardust.dev/play/>

<sup>14</sup><https://de.wikipedia.org/wiki/First-Class-Funktion>

<sup>15</sup><https://forum.nim-lang.org/t/9716>

```
type Animal = object
  name: string
  age: int
```

Figure 2.1: Objekt Deklaration

bildung 2.1 wird die Deklaration des Objekts *Animal* mit den Feldern *name* und *age* gezeigt. Beim Erzeugen eines Objektes können gleich die Werte für die Felder mit angegeben werden. Werden diese Werte nicht mit angegeben, werden sie automatisch auf 0 oder, im Falle eines Strings, auf einen leeren String gesetzt. Das ist wichtig, da es im Gegensatz zu anderen objektorientierten Sprachen keine Kontruktor- oder Dekonstruktor-Methoden gibt und so mit den Werten in Funktionen, die einen Parameter verlangen, gerechnet werden kann. Selbst wenn die Werte auf 0 gesetzt wurden, können sie trotzdem nachträglich noch verändert werden. Es sei denn, man initialisiert ein Objekt mit dem Schlüsselwort *let*, welches die Werte der Felder unveränderbar machen.

Es ist möglich ein Objekt zu referenzieren. Anstatt das Objekt auf dem *Stack* zu speichern, wird bei einer Referenz nur auf eine Stelle des *Heaps* im Speicher gezeigt. Dafür wird das Schlüsselwort *ref* verwendet.<sup>16</sup> Referenzen sind für die Vererbung erforderlich. Um Vererbung überhaupt möglich zu machen, muss das Objekt von *RootObj* erben. In dem Beispiel mit dem *Animal* Objekt würde das so aussehen: *type Animal = object of RootObj*. Um von diesem Objekt zu erben, deklariert man ein weiteren Objekt wie folgt: *type Pet = ref object of Animal*. Es ist möglich während der Laufzeit den Typ eines Objektes zu überprüfen. Dies geschieht mit dem *of* Operator, welcher vergleichbar mit dem Operator *instanceof* in Java ist.<sup>17</sup>

Methoden der Objekte werden nicht im Kontext des Objektes definiert, sondern können einfach als Prozedur mit *proc* implementiert werden. Durch UFCS können die Methoden dann, wie von anderen objektorientierten Sprachen gewohnt, verwendet werden: *instance.method()* anstatt *method(instance)*. Um die Werte eines Objekts beispielsweise mithilfe von Setter-Methoden zu verändern, muss das Argument für das Objekt der Methode durch *var* gekennzeichnet werden, da die Argumente von Prozeduren standardmäßig unveränderbar sind und *var* diese Beschränkung aufhebt. In Verbindung mit Vererbung können statt Prozeduren *proc* auch Methoden *method* benutzt werden. Der Vorteil liegt darin, dass Methoden durch Methoden des Subtypen überschrieben werden. Dazu benutzt man den Typ des Subtyps in den Parametern. Man sollte allerdings nur begrenzt darauf zurückgreifen, da Methoden eine dynamische Bindung und Prozeduren eine statische Bindung haben und diese performanter ist gegenüber der dynamischen ist.<sup>18</sup>

<sup>16</sup><https://nim-by-example.github.io/types/objects/>

<sup>17</sup><https://nim-lang.org/docs/manual.html>

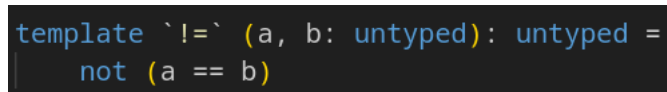
<sup>18</sup><https://nim-by-example.github.io/oop/>

### 2.3.3 Metaprogrammierung

Metaprogrammierung ist die Verwendung von Metaprogrammen, die beim Prozess der Programmierung mitarbeiten und so Fehler vermindern und die Effektivität steigern.<sup>19</sup> In Nim stehen dazu Generics, Templates und Macros zur Verfügung.

Generics erlauben die Parametrisierung von Prozeduren, Iteratoren und Typen mit Typ Parametern, wodurch die Typisierung erst später stattfindet. Dadurch wird sich duplizierender Code verringert. Parameter werden in eckigen Klammern geschrieben. Sie erweisen sich gerade für effiziente typsichere Container als sehr nützlich.<sup>20</sup>

Ein Template wird wie eine Prozedur aufgerufen. Sie sind eine einfache Form von Makros und eine Mechanik zum einfachen Austausch, welche im semantischen Durchgang im Kompilierer verarbeitet werden. Beispielsweise sind Nim's `!=`, `¿`, `¿=`, `in`, `notin` und `isnot` Operatoren als Templates umgesetzt worden.<sup>21</sup>



```
template `!=` (a, b: untyped): untyped =  
  not (a == b)
```

Figure 2.2: Umsetzung des `!=` Operators in Nim's System Modul

Ein Macro ist eine Funktion, die bei der Kompilierung ausgeführt wird und einen Nim Syntax Tree in einen anderen transformiert.<sup>22</sup> Macros eignen sich gut, um den Code übersichtlicher zu gestalten.

### 2.3.4 Foreign function interface

Ein mächter Teil von Nim ist das FFI (Foreign function interface). Diese Schnittstelle erlaubt es dem Programmierer, innerhalb seines Nim Programmes auf in anderen Programmiersprachen geschriebenen Bibliotheken zuzugreifen, indem diese importiert und deren Funktionen dann ganz normal verwendet werden können. Nim unterstützt Bibliotheken in den Sprachen C, C++, Objective-C und JavaScript. Allerdings können nicht alle vier Sprachen gleichzeitig mit eingebunden werden. Man muss sich zwischen C/C++/Obj-C oder JavaScript entscheiden, denn JavaScript ist inkompatibel zu den anderen drei Sprachen. Diese drei Sprachen können aber gleichzeitig in Nim verwendet werden, da C++ und Obj-C mit C rückwärts kompatibel sind. Nim unterstützt sowohl statisches Linken, bei dem die importierte Bibliothek in das Programm eingebettet wird, als auch dynamisches Linken, bei dem die Bibliothek auf dem PC des Nutzers installiert sein muss. Nim bevorzugt dynamisch gelinkte Bibliotheken. Wie in Abbildung 2.3 zu sehen,

<sup>19</sup><https://de.wikipedia.org/wiki/Metaprogrammierung>

<sup>20</sup><https://nim-lang.org/docs/tut2.html>

<sup>21</sup><https://nim-lang.org/docs/manual.html>

<sup>22</sup><https://nim-lang.org/docs/tut3.html>



```
proc printf(formatstr: cstring) {.header: "<stdio.h>", importc: "printf", varargs.}
```

Figure 2.3: Die Funktion `printf` aus der C Bibliothek `stdio`

wird gegebenenfalls eine Typkonvertierung benötigt, denn Nim's Typen sind teilweise nicht mit denen von C kompatibel.<sup>23</sup>

### 2.3.5 Parallelisierung und Nebenläufigkeit

Um Parallelisierung durch Threads zu aktivieren, muss das Programm mit der Option `-threads:on` kompiliert werden. Eine Funktion, die in einem Thread ausgeführt wird, sollte mit dem Pragma `{.thread.}` gekennzeichnet werden. Eine Thread Prozedur kann durch `spawn()` oder `createThread()` gestartet werden und auf laufende Threads kann mit `joinThreads()` gewartet werden. Anstatt Threads einzeln zu starten, kann die `threadpool` Bibliothek verwendet werden. Dort stehen strukturierter Parallelismus mit dem Statement `parallel` und unstrukturierter Parallelismus mit dem Statement `spawn` zur Verfügung. Die Verwendung dieser Statements wird durch gewisse Regeln eingeschränkt, um gewisse Sicherheiten zu gewährleisten. Beispielsweise sind Speicherzugriffe eingeschränkt und jeder Thread hat seinen eigenen Garbage Collector.

Nebenläufigkeit wird durch `async/await` Syntax erzielt. Eine Prozedur mit Nebenläufigkeit wird mit `{.async.}` gekennzeichnet. Dadurch kann sie das `await` Schlüsselwort verwenden, um auf asynchrone Prozeduren zu warten.

## 2.4 Infrastruktur

### 2.4.1 Kompilierer

Nim's Kompilierer ist eine der großen Stärken der Sprache, da dieser standardmäßig zu schnellem und optimiertem C Code kompiliert. Das macht es einfach in Nim zu programmieren, da Python's Syntax, auf der Nim basiert, anfängerfreundlicher und effizienter ist als direkt in C zu programmieren. Außerdem ist ein zu C kompiliertes Programm auch wesentlich schneller als ein interpretiertes Programm, wie es zum Beispiel bei Python der Fall ist. Die Kompilierung zu Objektcode wird an einen externen Kompilierer abgegeben, um von bereits vorhandenen Optimierungen zu profitieren. Zu den unterstützten C Kompilierern zählen beispielsweise *GCC* (*GNU Compiler Collection*), *MinGW* und *Clang*. Nim kann nicht nur Bibliotheken geschrieben in C, C++, Objective-C und JavaScript einbinden, sondern diese auch beim Kompilieren ausgeben. Das erleichtert die Verwendung von APIs (Application Programming Interface) in diesen Sprachen. Ursprünglich war der Kompilierer in Pascal geschrieben. Doch 2008 wurde eine

<sup>23</sup><https://livebook.manning.com/book/nim-in-action/chapter-8/102>

in Nim geschriebene Version des Kompilers veröffentlicht, sodass dieser selbsttragend ist. Nim läuft auf Linux, Windows und MacOS aber auch der Export nach diesen Betriebssystem wird von dem Compiler unterstützt, was gerade für Anwendungen für eingebettete Systeme und unübliche Architekturen von Nutzen ist. Durch Cross-Compilation ist es möglich für Android, iOS oder sogar für die Nintendo Switch zu kompilieren.<sup>24</sup>

Standardmäßig erstellt der Compiler ein Debug Build. Das ist für den Programmierer sehr nützlich, da dieses mehr Runtime Checks beinhaltet. Allerdings ist dieses Debug Build nicht großartig für Schnelligkeit optimiert. Um schnellere Builds mit weniger Runtime Checks zu erstellen, wird die Option `-d:release` benutzt. Mit der Option `-d:danger` werden für maximale Schnelligkeit sämtliche Runtime Checks entfernt.

Der Compiler implementiert verschiedene Garbage Collectors. Eine Garbage Collection ist eine automatische Speicherverwaltung, die zur Vermeidung von Speicherproblemen beiträgt, indem zur Laufzeit versucht wird, nicht länger benötigte Bereiche im Speicher zu finden, um diese dann wieder freizugeben. Einen Garbage Collector auszuwählen geht mit der Option `-gc`. Wobei standardmäßig ORC verwendet wird. Dieser versucht durch das automatische Zählen von Referenzen mit *move* Semantik Optimierungen freizugebenden Speicher zu identifizieren und bietet voll-deterministische Performance für harte Echtzeitsysteme. Referenzzyklen, die zu Memory Leaks führen würden, werden durch einen Zyklus Collector vermieden. Es ist möglich auf einen Garbage Collector zu verzichten, indem man die Option `-gc:none` angibt. Hierbei gibt es keine Speichermanagement Strategie und es wird kein Speicher mehr automatisch freigegeben. Der Entwickler muss sich also in diesem Fall selbst um die Speichernutzung in seinem Programm kümmern.

## 2.4.2 Entwicklungswerkzeuge

Die folgenden Werkzeuge werden mit der Standard Nim Installation automatisch mitgeliefert.<sup>25</sup>

### Nimble

Das wohl wichtigste Entwicklungswerkzeug für Nim ist Nimble. Es ist Nim's Paketmanager. Entwickelt wurde Nimble von einem der Hauptentwickler Nim's Dominik Picheta und ist seit 2015 Nim's offizieller Paketmanager. Nimble Pakete werden durch `.nimble` Dateien definiert, welche Informationen über die Paketversion, Autoren, Lizenzen, Abhängigkeiten und mehr enthalten. `.nimble` Dateien unterstützen NimScript, einen Teil von Nim's Syntax. Mit diesen Skripten können Test Prozeduren verändert werden und benutzerdefinierte Aufgaben geschrieben werden. Eine Liste mit verfügbaren Paketen im JSON (*JavaScript Object Notation*) Format wird

---

<sup>24</sup><https://nim-lang.org/docs/nimc.html>

<sup>25</sup><https://nim-lang.org/docs/tools.html>

im `nim-lang/packages` Repository auf Github frei zur Verfügung gestellt.<sup>26</sup> Diese Liste beinhaltet die Namen und Repositories der verschiedenen Pakete. Da es kein zentrales Repository mit allen Paketen gibt sondern jeder sein eigenes Repository hosten kann, ist eine gute Zugänglichkeit des Ökosystems gegeben. Nimble benötigt Git um richtig zu funktionieren. Mit dem Commandline Interface lassen sich Pakete der Module installieren, aktualisieren und deinstallieren.

## atlas

*atlas* ist ein einfaches Paket Kloner Werkzeug. Es verwaltet eine isolierte Arbeitsumgebung und beinhaltet Projekte und Abhängigkeiten. Es ist mit Nimble kompatibel, da es das Nimble Datei Format unterstützt. Es gibt drei Konzepte: Arbeitsflächen, Projekte und Abhängigkeiten.

Jede Arbeitsfläche ist isoliert und zwischen den Arbeitsflächen wird nichts geteilt. Jede Arbeitsfläche ist ein Verzeichnis und beinhaltet eine *atlas.workspace* Datei. Mit *atlas init* lassen sich Arbeitsflächen von dem derzeitigen Arbeitsverzeichnis erstellen.

Eine Arbeitsfläche enthält ein oder mehrere Projekte. Diese Projekte können sich gegenseitig benutzen und es ist einfach mehrere Projekte gleichzeitig zu entwickeln.

Auf einer Arbeitsfläche kann es ein `_deps` Verzeichnis geben, in dem die Abhängigkeiten aufbewahrt werden. Der einzige Unterschied zwischen einem Projekt und einer Abhängigkeit ist der Aufbewahrungsort. Für eine Abhängigkeitsresolution hat ein Projekt stets die höhere Priorität.

Atlas kümmert sich für den Entwickler nur um die zwei Dateien *project.nimble* und *nim.cfg*. Ansonsten wird alles so gelassen wie es ist.<sup>27</sup>

## Hot code reloading

Durch die `hotCodeReloading` Option wird ein spezieller Kompilierungsmodus aktiviert, wodurch Änderungen im Code automatisch auf ein laufendes Programm angewendet werden können. Das Neuladen des Codes passiert mit individuellen Modulen. Wenn ein Modul neu geladen wird, werden jegliche neu hinzugefügte globale Variablen initialisiert aber der restliche top-level Code in diesem Modul wird nicht erneut ausgeführt und der aktuelle Zustand von allen bereits existierenden globalen Variablen bleibt erhalten. Zurzeit funktioniert Hot Code Reloading nicht für das Hauptmodul selbst, weswegen ein Helfer Modul benutzt werden muss. Es gibt spezielle Event Handhaber, die vor und nach dem Reload ausgeführt werden, um den Zustand einer bestimmten Variable zu ändern oder um die Ausführung bestimmter Statements zu erzwingen. Diese Handhaber werden mit *beforeCodeReload* und *afterCodeReload*. Um Hot Code Reloading zu verwenden muss das `hotcodereloading` Modul importiert werden und beim Kompilieren die Option

---

<sup>26</sup><https://github.com/nim-lang/packages/blob/master/packages.json>

<sup>27</sup><https://nim-lang.org/docs/atlas.html>

`-hotcodereloading:on` gesetzt sein.<sup>28</sup>

## Document generator

Der in den Nim Kompilierer eingebaute Dokument Generator *nim doc* generiert HTML (*Hyptertext Markup Language*), Latex und JSON Dokumentation von *.nim* Quelldateien und Projekten. Der Output beinhaltet die Modulabhängigkeiten (Imports), alle obersten Dokumentationskommentare (`##`) und exportierte Symbole (`*`) inklusive Prozeduren, Typen und Variablen. Man kann eine Dokumentation in HTML mit *nim doc filename.nim* generieren.<sup>29</sup>

## Nimsuggest IDE support

Da Nim sehr schnell ist, bietet es sich an, den Texteditoren externe Abfragen über den geschriebenen Quellcode zur Verfügung zu stellen. Durch das *nimsuggest* Werkzeug kann jede Entwicklungsumgebung *.nim* Quelldateien abfragen und nützliche Informationen wie Definitionen von Symbolen oder Vorschläge zur automatischen Vervollständigung erhalten.<sup>30</sup>

## C2nim

Ein ANSI C/C++ zu Nim Quellcode Übersetzer. Das Werkzeug übersetzt C Header Files nach Nim. Deswegen ist der Preprocessor Teil des Parsers. Der Output ist leserlicher Nim code, welcher noch durch den Entwickler überarbeitet werden sollte. C2nim ist dafür gedacht C Code Fragmente zu übersetzen und folgt daher keinen `inlude` Dateien. Nicht jeder C/C++ Code kann übersetzt werden, da Konstrukte in C/C++ existieren, die in Nim nicht dargestellt werden können. Mit der Option `-nep1` generiert C2nim Nim Code, der den offiziellen Style Guides folgt.<sup>31</sup>

## niminst

Ein Werkzeug zum Generieren eines Installers für ein Nim Programm. Zurzeit können nur ein altmodischer grafischer Installer für Windows sowie Installations- und Deinstallationsskripte für Linux/Unix erstellt werden. Unterstützung für Paketmanager unter Linux ist ebenfalls geplant. Zur Erstellung des Installers liest *niminst* eine Konfigurationsdatei aus, die alle notwendigen Informationen für den Installer für die jeweiligen Betriebssysteme enthält.<sup>32</sup>

---

<sup>28</sup><https://nim-lang.org/docs/hcr.html>

<sup>29</sup><https://nim-lang.org/docs/docgen.html>

<sup>30</sup><https://nim-lang.org/docs/nimsuggest.html>

<sup>31</sup><https://github.com/nim-lang/c2nim/blob/master/doc/c2nim.rst>

<sup>32</sup><https://nim-lang.org/docs/niminst.html>

## **nimgrep**

Ein Nim Such- und Ersetzungswerkzeug in der Kommandozeile. Es kann nach Regex und Peg Patterns suchen und kann ganze Verzeichnisse auf einmal durchsuchen. Nimgrep hat eine besonders gute Unterstützung für Nim's Style Insensibilität durch die *-y* Option.

## **nimpretty**

Mit *nimpretty* lassen sich Nim Quelldateien verschönern, indem Code nach den offiziellen Style Guides formatiert wird.<sup>33</sup>

## **testament**

testament ist ein fortgeschrittener Unittest Runner für Nim Tests, welcher für die Entwicklung Nim's selbst verwendet wird. Es bietet Prozess Isolation für Tests, kann Statistiken über Testfälle und HTML Berichte generieren. Außerdem werden mehrere Programmiersprachen unterstützt.<sup>34</sup>

## **koch**

Das koch Programm ist ein Wartungsskript. Es ist als Ersatz für *make* und Shell Scripting gedacht mit dem Vorteil, dass es viel portabler ist. Es ist hauptsächlich zum Bauen des Kompilers gedacht aber es kann auch anderweitig benutzt werden.

Andere nützliche Werkzeuge, die nicht automatisch mitgeliefert werden:

## **choosenim**

Choosenim ist ein Werkzeug, das die Installation und Benutzung mehrerer Versionen des Nim Kompilers ermöglicht. Man kann jede *stable* oder *development* Version mit der Kommandozeile installieren und schnell und einfach zwischen ihnen wechseln. Es macht jedoch den Eindruck, als würde dieses Tool nicht mehr weiterentwickelt werden, da das letzte Update schon 1,5 Jahre zurück liegt.<sup>35</sup>

## **nimpy**

Nimpy ist eine Bibliothek, die Python in Nim integriert.<sup>36</sup>

---

<sup>33</sup><https://nim-lang.org/docs/tools.html>

<sup>34</sup><https://nim-lang.org/docs/testament.html>

<sup>35</sup><https://github.com/dom96/choosenim>

<sup>36</sup><https://github.com/yglukhov/nimpy>

## Pixie

Pixie ist eine umfangreiche 2D Grafik Bibliothek für Nim.<sup>37</sup>

## nimterop

Nimterop ist ein Paket mit dem Ziel die Interoperabilität makellos zu gestalten. Es automatisiert die Erstellung von C Wrappern für Nim's FFI. C++ Unterstützung ist ebenfalls geplant.<sup>38</sup>

### 2.4.3 Bibliotheken

Bei Nim Bibliotheken wird zwischen Pure und Impure unterschieden. Pure bedeutet, dass die Module der Bibliothek ausschließlich in Nim geschrieben wurden ohne Verwendung irgendwelcher Wrapper, um auf Bibliotheken zuzugreifen, die in anderen Sprachen geschrieben wurden. Bei Impure sind die Nim Module auf Code aus Bibliotheken angewiesen, die in anderen Programmiersprachen geschrieben wurden wie zum Beispiel C oder C++.

Die Standardbibliothek beinhaltet Module für grundlegende Aufgaben. Im Folgenden eine Auswahl an *pure* Modulen aus der Standardbibliothek.<sup>39</sup>

**system** Wird automatisch importiert und beinhaltet Prozeduren und Operationen, die jedes Programm benötigt.

**macros** Zum Erstellen von Makros

**algorithm** Häufige generische Algorithmen wie zum Beispiel zum Sortieren von Listen.

**set** Unterstützung für *set* Strukturen.

**strutils** Zur Verarbeitung von Strings.

**times** Um mit der Zeit zu arbeiten.

**files** Um mit Dateien zu interagieren.

**math** Operationen wie Kosinus oder Berechnung der Quadratwurzel.

**uri** Um mit URIs und URLs zu arbeiten

**threadpool** Implementierung von Nim's *spawn*.

**json** Ein schneller JSON Parser.

**base64** Ein Base64 Kodierer und Dekodierer.

**logging** Ein einfacher Logger

---

<sup>37</sup><https://github.com/treeform/pixie>

<sup>38</sup><https://github.com/nimterop/nimterop>

<sup>39</sup><https://nim-lang.org/docs/lib.html>

Auf `nimble.directory` werden viele externe Bibliotheken mit kurzer Beschreibung vorgestellt.<sup>40</sup>

Nim kann Bibliotheken in den Sprachen C, C++, Objective-C und JavaScript nativ in das Programm einbinden. Aber auch auf Bibliotheken in anderen Sprachen kann mithilfe von Language Bindings zugegriffen werden. Für Bibliotheken wie GTK, OpenSSL und Vulkan gibt es bereits solche Bindings.

#### 2.4.4 Dokumentation

Obwohl Nim nicht die weit verbreitetste Sprache ist, verfügt sie über eine ausgesprochen gute und umfangreiche Dokumentation. Es gibt mehrere Tutorials für Anfänger und Experten. Tutorials extra für Programmierer, die schon C, Python oder JavaScript Erfahrung haben. Dokumentation für die Standardbibliothek inklusive der einzelnen Module ist ebenfalls verfügbar. Genauso wie Dokumentation für die einzelnen Bestandteile der Sprache, wie zum Beispiel die Bedienung des Kompilers, der Aufbau der Syntax oder einfach nur wie man Nim auf dem PC installiert. Außerdem werden auch die verschiedenen Entwicklungswerkzeuge, die ich eben schon vorgestellt habe, im Detail erklärt. Und da Nim eine aktive Community hat, kann man sich zur Not auch an andere Leute wenden, wenn man Probleme hat.

---

<sup>40</sup><https://nimble.directory/>

## Chapter 3

# Implementierung

### 3.1 Herangehensweise

Das Programm habe ich schrittweise in der Reihenfolge geschrieben, wie es dann später ausgeführt wird. Angefangen mit dem Einlesen der Parameter aus der Kommandozeile. Dafür werden die einzelnen Parameter erst einmal in einer Sequenz gespeichert. Anschließend wird mit Hilfe einer for-Schleife Parameter für Parameter überprüft, ob es sich um eine valide Option des Programms handelt. Um das umzusetzen, habe ich ein Switch Statement verwendet. Wenn eine valide Option gefunden wurde, wird die entsprechende Variable im Programm auf True oder False oder auf den entsprechenden Wert gesetzt. Da es sich nicht bei allen Parametern gleichzeitig auch um Optionen handelt, wird unter den Parametern zwischen Optionen und Pattern oder Pfad unterschieden. Die Unterscheidung kann man einfach umsetzen, weil Optionen mit einem Bindestrich beginnen. Außerdem gibt das Programm dem Nutzer die Syntax vor, die verlangt, dass erst die Optionen angegeben werden, dann das Pattern und danach optional noch den Pfad beziehungsweise die Pfade, falls mehrere Orte durchsucht werden sollen. Das heißt, das Programm kann davon ausgehen, dass der erste Parameter nach den Optionen das Pattern ist und alles, was dann noch folgt, als Pfad interpretiert wird. Die angegebenen Pfade werden in absolute Pfade umgewandelt, damit der Nutzer neben Dateien und Verzeichnissen im aktuellen Arbeitsverzeichnis auch beliebig andere Dateien von außerhalb angeben kann. Wenn kein Pfad angegeben wird, werden alle Dateien und Verzeichnisse im aktuellen Arbeitsverzeichnis durchsucht.

Damit die Suche nach dem Einlesen starten kann, habe ich die Prozedur `checkFiles` geschrieben, welche eine Sequenz an Pfaden übergeben bekommt. In dieser Prozedur werden zuallererst die einzelnen Pfade in *Pfad zu einer Datei* und *Pfad zu einem Verzeichnis* unterteilt. Das habe ich gemacht, damit ich erst die einzelnen Dateien untersuchen kann und anschließend die Verzeichnisse rekursiv weiter nach weiteren Dateien und Verzeichnissen untersuchen kann, indem die Prozedur `checkFiles` erneut mit den neuen Pfaden aufgerufen wird.

Um die einzelnen Dateien zu untersuchen, gibt es die `checkFile` Proze-



dur, welche zum Einen den Pfad zur Datei und zum Anderen das Pattern übergeben bekommt. Bevor in der Datei nach dem Pattern gesucht werden kann, muss allerdings erst einmal überprüft werden, ob die Datei überhaupt Text enthält. Dafür rufe ich im Programm den file-Befehl<sup>1</sup> auf und lese anhand des MIME-Typs aus dessen Output, ob es sich bei der Datei um eine Textdatei handelt oder nicht. Wenn es keine Textdatei ist, wird sie übersprungen und nicht durchsucht.

Um eine Datei zu durchsuchen, wird Zeile für Zeile überprüft, ob diese das Pattern beinhaltet. Wenn das Pattern gefunden wurde, wird der Index der entsprechenden Zeile in einer Sequenz gespeichert. Außerdem wird bei jeder Datei gezählt, wie viele Zeilen sie hat, weil in einem späteren Schritt die letzte Zeile beziehungsweise das Ende wichtig wird.

Sobald alle Zeilen überprüft wurden und mindestens ein Treffer gefunden wurde, wird die nächste Prozedur aufgerufen: `printResult`. Ihr werden Pfad zur Datei, Anzahl aller Zeilen der Datei und die Sequenz der Indizes der Zeilen, in der das Pattern gefunden wurde, übergeben. Falls der Nutzer angegeben hat, dass das Ergebnis mit Kontext Zeilen ausgegeben werden soll, wird im nächsten Schritt die Sequenz mit den Indizes um jeweils den entsprechenden Kontext erweitert.

Wenn beispielsweise der Index 70 in der Sequenz beinhaltet ist und Kontext davor von 2 und Kontext danach von 3 erwünscht ist, werden die entsprechenden Indizes der Sequenz hinzugefügt, sodass nachher 68, 69, 70, 71, 72, 73 gemeinsam in der Sequenz sind.

Jetzt wird die Datei erneut Zeile für Zeile durchgegangen. Allerdings nicht, um sie nochmal nach dem Pattern zu durchsuchen, sondern wird hier bei jeder Zeile geschaut, ob ihr Index in der Sequenz mit beinhaltet ist. Wenn das der Fall ist, wird die Zeile ausgegeben. Hier kann ebenfalls unterschieden werden, ob es sich um eine Zeile mit Pattern handelt oder ob es nur eine Kontext Zeile ist, indem zusätzlich überprüft wird, ob der Index auch in der Sequenz bestehend nur aus den Zeilen mit Pattern und ohne Kontext Zeilen ist. Gleichzeitig wird der Output entsprechend der Optionen gestaltet, je nachdem er farbig sein soll und mit oder ohne Heading.

Zusätzlich habe ich noch ein paar Helferprozeduren geschrieben. `printHelp` gibt den Syntax und alle validen Optionen für das Programm aus und wird aufgerufen, wenn der Nutzer die Hilfe Option benutzt oder wenn ein Eingabefehler vorgefallen ist. In diesem Fall wird zusätzlich ebenfalls eine Fehlermeldung ausgegeben.

`isHiddenFile` überprüft, ob eine Datei versteckt ist, indem sie schaut, ob der Dateiname mit einem Punkt beginnt. Dafür wird erst nach dem letzten Schrägstrich gesucht und dann geschaut, ob das Zeichen rechts daneben ein Punkt ist. Leider arbeitet die Prozedur etwas ineffizient, da sie die einzelnen Zeichen von links nach rechts nach dem letzten Schrägstrich absucht. Effizienter wäre es gewesen, wenn man von hinten angefangen hätte zu suchen allerdings habe ich keinen Weg gefunden, die `for`-Schleife von hinten laufen

---

<sup>1</sup><https://darwinsys.com/file/>

zu lassen.

`findIndices` erhält eine Zeile Text und sucht, an welchen Stellen das Pattern gefunden wurde. Zurückgegeben wird eine Sequenz an Indizes. Die jeweiligen Indizes geben an, wo das Pattern startet.

`formatPatternInLine` sorgt dafür, dass in einer Zeile mit Pattern das Pattern farbig dargestellt wird, sofern dies auch vom Nutzer erwünscht ist. Dafür wird meine andere Helferprozedur `findIndices` benutzt, um die jeweiligen Stellen der Patterns zu finden und diese dann durch formatierte Patterns auszutauschen.

## 3.2 Benutzte Bibliotheken

### 3.2.1 os

Diese Bibliothek beinhaltet grundlegende Dienstleistungen des Betriebssystems, wie der Umgang mit Umgebungsvariablen, Arbeit mit Dateien und Verzeichnissen usw.. Ich habe sie verwendet für die Interaktion mit Dateien und Verzeichnissen. Beispielsweise die Prozedur `getFileInfo`, um zu überprüfen, ob es sich um eine Datei oder um ein Verzeichnis handelt. Oder um zu überprüfen, ob ein Pfad überhaupt existiert. Außerdem benutzte ich die Prozedur `getAppDir`, um herauszufinden, in welchem Arbeitsverzeichnis sich der Nutzer zurzeit befindet, da dies wichtig für die Erstellung der absoluten Pfade war.

### 3.2.2 osproc

`osproc` ist dafür da, um Prozesse des Betriebssystems auszuführen und mit Prozessen zu kommunizieren. Ich habe die Prozedur `execProcess` verwendet, um den Befehl `find` in mein Programm einzubauen.

### 3.2.3 strutils

`strutils` beinhaltet einige übliche Funktionen, um mit Strings zu arbeiten. Benutzt habe ich die Funktionen `contains`, um in einer Zeile nach dem Pattern zu suchen, `toLowerAscii`, um Unterschiede der Groß- und Kleinschreibung zu entfernen, falls der Nutzer die Option `-ignore-case` aktiviert hat. Außerdem habe ich `parseInt` verwendet, um bei der Optionsangabe die Werte aus den Parametern von Strings zu Integers umzuwandeln. `substr` hat bei der Formatierung des Outputs Verwendung gefunden, da man mit dieser Funktion Teilstrings erhalten kann.

### 3.2.4 terminal

`terminal` hat ein paar Funktionen, die das Terminal kontrollieren. Ich habe `ansiForegroundColorCode` verwendet, um den Output zu farbig zu formatieren. Ursprünglich hatte ich geplant, die ANSI Escape Sequenzen ohne zusätzliche Funktionen einfach in die Strings zu schreiben aber das hat mit Nim bei mir

nicht funktioniert, weswegen ich diese Bibliothek verwendet habe, was etwas umständlicher war.

### 3.3 Verwendete Elemente der Sprache

Neben Standard Elementen einer Sprache wie *if* und *switch* Statements oder *for*-Schleifen fand ich den eingebauten Zeilen Iterator sehr nützlich. Diesen habe ich in meiner *checkFiles* Prozedur verwendet und dieser hat mir erlaubt, ohne viel Mehraufwand die einzelnen Zeilen einer Datei nacheinander durchzugehen. Einen Nutzen habe ich des weiteren aus Pattern Matching beim Aufteilen der Pfade in Dateien und Verzeichnisse gezogen.

### 3.4 Optimierungen

Leider habe ich es nicht geschafft mein Programm so zu optimieren, wie ich es gerne hätte.

Wie vorhin schon kurz erwähnt, ist es mir nicht gelungen in der Prozedur *isHiddenFile* die *for*-Schleife von rückwärts laufen zu lassen, damit nicht jedes Mal ein Großteil des Strings unnötigerweise durchlaufen werden muss.

Eine von mir von Anfang an geplante Optimierung, die in Nim's Standard Bibliothek enthaltenen Threads zur Parallelisierung zu verwenden, konnte ich letztendlich nicht umsetzen. Ich hatte vor, mehrere Verzeichnisse parallel zueinander zu durchsuchen. Doch neben der ohnehin schon recht schwierigen Umsetzung der Threads, wusste ich nicht, wie ich es schaffen soll, dass die einzelnen Threads ihren Output nicht durcheinander ausgeben und das, ohne dabei die von den Threads erhoffte Leistungssteigerung zu sehr zu mindern.

Eine Sache, die ich optimieren konnte, war, wie ich Dateien Zeile für Zeile durchsuche. In einer vorherigen Version des Programms hatte ich die komplette Datei eingelesen, in Zeilen aufgeteilt und diese dann in einer Sequenz abgespeichert. Das war für kleine Dateien zwar vorteilhaft, da ich auf die einzelnen Zeilen per Index beliebig zugreifen konnte und mir so weitere Arbeit ersparen konnte, allerdings ist dieser Ansatz mit größeren Dateien nicht mehr praktikabel, da der Speicher des PC volllaufen kann und das dann schließlich zum Abbruch des Programms führt. Mein jetziger Ansatz ist, die einzelnen Zeilen mit einem Iterator nacheinander durchzugehen, ohne alle auf einmal abzuspeichern. Dadurch ist es allerdings erforderlich, die Datei ein zweites Mal durchzugehen, um die einzelnen Zeilen auszugeben. Außerdem wird beim zweiten Durchgang bei jeder Zeile überprüft, ob dieser Index in der Sequenz der Indizes ist, die angeben, welche Zeilen ausgegeben werden sollen. Dieser Schritt kann bei vielen Zeilen mit Pattern schnell rechenintensiv werden aber ist erforderlich, um eine korrekte Ausgabe mit Kontextzeilen zu gewährleisten. Wenn keine Kontextzeilen gefordert sind, wird eine aufwendige Listenkopie übersprungen, wodurch auch Zeit eingespart wird.

# Chapter 4

## Fazit

### 4.1 Tauglichkeit der Sprache zur Entwicklung einer *grep* Alternative

Ich denke generell kann man sagen, dass sich Nim dazu eignet, eine *grep* Alternative zu schreiben. Ich für meinen Teil bin allerdings nicht richtig warm geworden mit dieser Sprache. Was mich am meisten gestört hat, war tatsächlich die Syntax. Ich hatte in der Vergangenheit zwar schon mal ein wenig mit Python gearbeitet aber die Syntax ist trotzdem noch etwas neu für mich. Von anderen Sprachen bin ich es gewöhnt, Code Blöcke mit geschweiften Klammern zu kennzeichnen, was ich auch übersichtlicher finde als dafür Einrückung zu verwenden. Manche Dinge empfand ich auch als umständlicher umzusetzen.

Aber ich gehe davon aus, dass sich das mit der Zeit legen würde und man auch effizienter arbeiten kann, wenn man erst einmal mehr Erfahrung mit der Sprache hat.

### 4.2 Zusammenfassung der Sprache

Objektiv gesehen ist Nim eigentlich **die** Programmiersprache. Obwohl sie nicht so weit verbreitet ist wie andere recht neue Sprachen wie zum Beispiel Rust, kann sie viel bieten. Mit Nim's einfacher Syntax, starker statischen Typisierung und flexiblen Macros spricht sie viele Programmierer an und eignet sich für so gut wie alle Bereiche. Vom einfachen Skript bis zum low-level Code und Mikrocontrollern wird alles abgedeckt. Es ist möglich sowohl das Frontend als auch das Backend einer Anwendung komplett in Nim zu schreiben. Außerdem ermöglicht die Kompilierung zu JavaScript eine breite Einsatzmöglichkeit in bestehenden Systemen. Das stärkste Argument für Nim meiner Meinung nach ist die Möglichkeit relativ schnell und einfach Code wie mit einer High-Level Programmiersprache zu schreiben und das Programm dann zu C oder C++ kompilieren zu können, um von der dadurch gewonnenen Performance zu profitieren.