Besondere Lernleistung im Fach Informatik

Entwicklung einer eigenen Programmiersprache

Autor: ...
Kurs: MSS 12 Stammkurs Englisch
Lehrer: ...

Abgabedatum: 03.07.2024

Inhaltsverzeichnis

1 Einleitung

Programmiersprachen sind einer der wichtigsten Bestandteile der Informatik. Mit ihnen kann man einem Computer Anweisungen geben, die dieser dann ausführt. Heutzutage gibt es viele verschiedene Programmiersprachen, die alle ihr Besonderheiten haben. Weil ich mich für die Funktionsweise von Programmiersprachen interessiere, habe ich mich dazu entschieden im Rahmen einer besonderen Lernleistung eine eigene Programmiersprache bzw. einen eigenen Compiler zu entwickeln. In dieser Arbeit wird auf die Eigenschaften meiner Sprache, sowie auf den Entwicklungsprozess des Compilers eingegangen.

2 Die Sprache

2.1 Ziele

Mein Ziel für die Sprache war es, eine recht einfache, aber dennoch featurereiche Sprache zu entwickeln. Sie sollte eine einfache, Python-ähnliche Syntax haben, aber auch komplexere Konzepte wie Generics und Operatorüberladung unterstützen.

2.2 Syntax & Semantik

Wie bereits erwähnt ist die Syntax weitgehend an Python angelehnt. So sieht zum Beispiel ein einfaches Hello World Programm in meiner Sprache aus:

```
use "std/io.mx"

def main() -> int {
    print("Hello, World!");
    return 0;
}
```

Wie in anderen Sprachen ist die `main` Funktion der Einstiegspunkt in ein Programm. Diese muss immer vorhanden sein und muss den Rückgabetyp `int` haben. Dieser zurückgegebene Wert ist dann auch der Exit-Code des Programms.

Variablen können mit dem Schlüsselwort `let` deklariert werden. Standardmäßig sind Variablen nicht veränderbar. Um eine veränderbare Variable zu deklarieren, muss diese mit `let mut` deklariert werden. Eine Typangabe ist immer optional.

```
let x = 5;
let mut y: int = 10;

y = x + 5;
```

Man kann auch Zeiger verwenden. Der Datentyp eines Zeigers sieht so aus: `*T`, wobei `T` der Typ des Wertes ist, auf den der Zeiger zeigt. Um einen Zeiger zu erstellen, der auf einen existierenden Wert zeigt, wird das Zeichen `&` verwendet. Um den Zeiger zu dereferenzieren, wird das Zeichen ~ verwendet.

```
let x = 5;
let y = &x;
let z = ~y;
# z = 5
```

Funktionen können mit dem Schlüsselwort `def` deklariert werden. Parameter einer Funktion werden in Klammern hinter dem Namen angegeben, der Typ der Parameter muss angegeben werden. Wenn die Funktion einen Wert zurückgibt, dann muss der Rückgabetyp mit einem Pfeil `->` angegeben werden.

```
def add(x: int, y: int) -> int {
    return x + y;
}
```

Es gibt auch While-Schleifen und Fallunterscheidungen (if, else, else-if). Ich habe mich bewusst gegen die Implementierung von For-Schleifen entschieden, weil es damit nur einen Weg gibt, eine Schleife zu durchlaufen. Dadurch ist der Code in meiner Sprache einheitlicher und einfacher zu lesen.

```
let mut x = 0;
while x < 10 {
    x = x + 1;
}

if x == 10 {
    print("x ist 10");
} else if x == 5 {
    print("x ist nicht 5");
} else {
    print("x ist weder 10 noch 5");
}
```

Es ist auch möglich, eigene Datentypen in Form von Klassen zu definieren. Dies ist mit dem Schlüsselwort `class` möglich.

```
class Vector {
    x: int,
    y: int,
}

def main() -> int {
    let v = Vector { x: 1, y: 2 };
    return v.x + v.y;
}
```

Klassen können auch Methoden haben. Diese werden wie eine normale Funktion deklariert, nur das nach den Parametern das Schlüsselwort `for` gefolgt von dem Klassennamen steht. Wenn eine Methode einen Parameter der Klasseninstanz (`self`) besitzt, dann ruft man diese mit einem Punkt (`.`) gefolgt von dem Methodennamen auf. Ist die Methode statisch (verwendet also keinen `self` Parameter), dann wird sie mit einem Doppelpunkt (`::`) hinter dem Klassennamen aufgerufen.

```
def vector_add(self, other: Vector) for Vector -> Vector {
    return Vector {
        x: self.x + other.x,
        y: self.y + other.y
    };
}

def default() for Vector -> Vector {
    return Vector { x: 0, y: 0 };
}

def main() -> int {
    let v1 = Vector { x: 1, y: 2 };
    let v2 = Vector::default();

    let v3 = v1.vector_add(v2);
    return v3.x + v3.y;
}
```

Methoden können auch für primitive Typen implementiert werden:

```
def is_even(self) for int -> bool {
    return self % 2 == 0;
}
```

Andere Datein können mit dem Schlüsselwort `use` eingebunden werden.

foo.mx

```
def foo() -> int {
    return 5;
}

def bar() -> int {
    return 10;
}
```

main.mx

```
use "foo.mx"

def main() -> int {
    return foo() + bar();
}
```

2.3 Besonderheiten

2.3.1 Typsystem

Ich habe mich für ein statisches Typsystem entschieden, weil dadurch schon beim Kompilieren viele Fehler gefunden werden können. Typen von Variablen und Argumenten können vom Compiler fast immer ermittelt werden, und sollte man irgendwo einen falschen Typen verwenden, wird ein präziser Fehler[1] ausgegeben, der auf die Stelle im Code hinweist, wo der Fehler gemacht wurde.

[1] Codeverweis Parserfehler: `src/parser/error.rs`

```
def main() -> int {
    let x = 5;
    let y = 1.0;
    return x + y;
}
```

Dieser Code würde beim Kompilieren zu diesem Fehler führen:

```
error[5]: wrong type
example:4:20

  2 let x = 5;
  3 let y = 1.0;
  4 return x + y;
            ^^^^^ expected: "int", got: "float"

Error:
    0: compiler error: 5
```

Typen müssen immer explizit umgewandelt werden, *Type Coercion*[2] gibt es daher nicht. Man würde den Code also so schreiben:

```
def main() -> int {
    let x = 5;
    let y = 1.0;
    return x + (y as int);
}
```

2.3.2 Generics

In meiner Sprache ist es möglich, generische Datentypen zu verwenden. Funktionen können auch generisch sein:

[2]Type Coercion: Automatische Umwandlung von Typen `https://developer.mozilla.org/en-US/docs/Glossary/Type_coercion`

```
def echo<T>(x: T) -> T {
    return x;
}

def main() -> int {
    return echo(5);
}
```

Und auch Klassen und deren Methoden können generisch sein:

```
class Vector<T> {
    x: T,
    y: T,
}

def invert<T>(self) for Vector<T> {
    self.x = self.y;
    self.y = self.x;
}
```

Generische Typen können vom Compiler fast immer ermittelt werden. Ausnahmen sind Funktionen oder Methoden, die den generischen Parameter nicht als Parameter verwenden. In diesem Fall muss der generische Typ in spitzen klammern (`<T>`) vor den Argumenten angegeben werden.

```
def new<T>() for List -> List<T> {
    ...
}

def main() -> int {
    let list = List::new<int>();
    return 0;
}
```

2.3.3 Operatorüberladung

In meiner Sprache ist es auch möglich Operatoren zu überladen. Dazu muss eine Methode mit einem speziellen Namen, der dann einem Operator zugeordnet wird, definiert werden.

```
def add(self, other: Vector<int>) for Vector<int>
-> Vector<int> {
    return Vector {
        x: self.x + other.x,
        y: self.y + other.y
    };
}
```

Es ist auch möglich, die Operatoren von primitiven Datentypen zu überladen:

```
def add(self, other: int) for int -> int {
    return 4;
}
```

Es können auch Generics verwendet werden:

```
def idx<T>(self, index: int) for List<T> {
    ...
}
```

2.3.4 Aufruf von C Funktionen

In meiner Sprache können alle Funktion der C Standardbibliothek (*libc*[3]) verwendet werden. Je nach Betriebssystem varriert der Umfang der Funktionen. Auf Windows können zum Beispiel Funktionen der *Windows API*[4] verwendet werden.

Dadurch ist es möglich, Funktionen zur Speicherverwaltung (`malloc`, `free`), Dateioperationen (`fopen`, `fclose`) und zur Ein- und Ausgabe (`printf`, `scanf`) zu verwenden.

Um eine externe C Funktion nutzen zu können, wird eine Funktion mit dem Schlüsselwort `extern` deklariert. Der Funktionskörper wird dabei nicht angegeben, sondern nur die Signatur der Funktion. Generics werden dabei nicht unterstützt.

Im folgenden Beispiel handelt es sich um die Windows Funktion `Sleep`[5].

[3]libc: C Standardbibliothek https://de.wikipedia.org/wiki/C-Standard-Bibliothek

[4]Windows API: Windows Programmierschnittstelle https://de.wikipedia.org/wiki/Windows_API

[5]Windows Sleep Funktion: https://learn.microsoft.com/de-de/windows/win32/api/synchapi/nf-synchapi-sleep

```
extern def Sleep(s: int) -> void;

def main() -> int {
    Sleep(1000);
    return 0;
}
```

Durch den Aufruf von C Funktionen ist es möglich, Datentypen mit variabler Größe (also im Heap allokiert) zu erstellen.

```
extern def calloc(num: int, size: int) -> int

def _calloc<T>(num: int) -> *T {
    return calloc(num, sizeof(T)) as *T;
}

class List<T> {
    ...
}

def new<T>() for List -> List<T> {
    let DEFAULT_CAP = 16;
    return List {
        data: _calloc<T>(DEFAULT_CAP * size_of(T)),
        len: 0,
        cap: DEFAULT_CAP,
    }
}
```

2.3.5 Makros

Es gibt in meiner Sprache auch Makros, diese werden während dem Kompilieren zu Code umgewandelt. Diese Makros sind nicht vom Benutzer definierbar, lassen sich allerdings im Code des Compilers recht einfach hinzufügen.

Ein Listen Makro könnte zum Beispiel folgenden Code erzeugen:

```
let list = list![1, 2, 3];
```

wird zu:

```
let list = {
    List::new<int>();
    list.push(1);
    list.push(2);
    list.push(3);
    return list;
}
```

Um ein Makro verwenden zu können, kann man im Compiler Bedingungen in Form von benötigten Klassen, Funktionen oder Methoden angeben. In dem oben gezeigten Beispiel wären die Bedingugen, dass es eine Klasse `List` mit den Methoden `new` und `push` gibt.

Dadurch sind Datentypen wie z.B. Strings in der Sprache selbst definiert.

2.4 Beispielprogramme & Vergleich mit anderen Sprachen

2.4.1 Bekannte Algorithmen

Im folgenden Abschnitt werden einige bekannte Algorithmen in meiner Sprache implementiert und mit den Programmiersprachen C++ und Python verglichen. Im Anschluss wird eine Laufzeitanalyse durchgeführt, hierbei wird auch die absolute Laufzeit in Millisekunden verglichen. Die Implementierungen in Python befinden sich im Anhang.

Bubble Sort (in Place)

C++

```
void bubbleSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i) {
        for (int j = 0; j < n - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Meine Sprache

```
def bubble_sort(list: *List<int>) {
    let mut list = ~list;
    let n = list.len;
    let mut i = 0;
    while i < n - 1 {
        let mut j = 0;
        while j < n - i - 1 {
            if list[j] > list[j + 1] {
                let temp = list[j];
                list[j] = list[j + 1];
                list[j + 1] = temp;
            }
            j = j + 1;
        }
        i = i + 1;
    }
}
```

Fibonacci-Algorithmus (rekursiv)

C++

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Meine Sprache

```
def fibonacci(n: int) -> int {
    if n <= 1 {
        return n;
    }

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Matrizenmultiplikation

C++

```
typedef std::vector<std::vector<int>> matrix;

int matrix_multiply(matrix a, matrix b, matrix c) {
    int i, j, k;
    int sum;
    int n = a.size();
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            sum = 0;
            for (k = 0; k < n; k++) {
                sum += a[i][k] * b[k][j];
            }
            c[i][j] = sum;
        }
    }
    return 0;
}
```

```
def matrix_multiplicaton(a: *List<List<int>>, b: *List<List<int>>)
-> List<List<int>> {
    let mut result = List::new<List<int>>();

    let a = ~a;
    let b = ~b;

    let n = a.len;
    let m = b[0].len;
    let p = b.len;

    let mut i = 0;
    while i < n {
        let mut row = List::new<int>();
        let mut j = 0;
        while j < m {
            let mut sum = 0;
            let mut k = 0;
            while k < p {
                sum = sum + a[i][k] * b[k][j];
                k = k + 1;
            }
            row.push<int>(sum);
            j = j + 1;
        }
        result.push<List<int>>(row);
        i = i + 1;
    }

    return result;
}
```

2.4.2 Laufzeitanalyse

Nun werden die oben gezeigten Algorithmen mit der erwarteten und tatsächlichen Laufzeit verglichen. Es wird auch die absolute Laufzeit in Millisekunden verglichen. Um die Zeit zu messen, wird vor und nach Aufruf der Funktion die

Zeit durch die WinAPI Funktion (C++ und meine Sprache) `GetSystemTimeAsFileTime` [6] bzw. durch die Funktion `time.time()` in Python gemessen. Die Tests werden auf dem `AMD Ryzen 7 3700X (16) @ 3.6GHz` Prozessor durchgeführt.

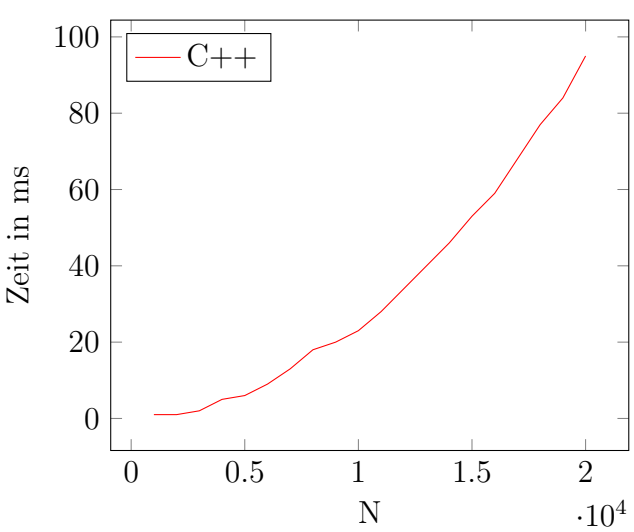
Angaben in Millisekunden. Programme in C++ und in meiner Sprache wurden mit `clang++` [7] auf der höchsten Optimierungsstufe kompiliert. Die Python Programme wurden mit `Python 3.10.11` ausgeführt.

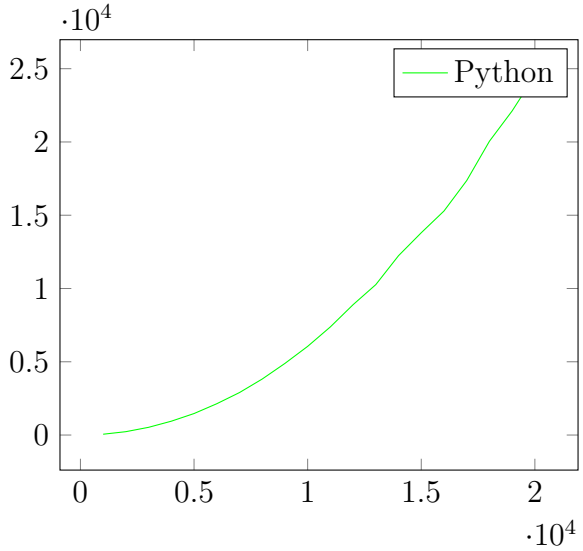
[6] GetSystemTimeAsFileTime Funktion `https://learn.microsoft.com/de-de/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsystemtimeasfiletime`

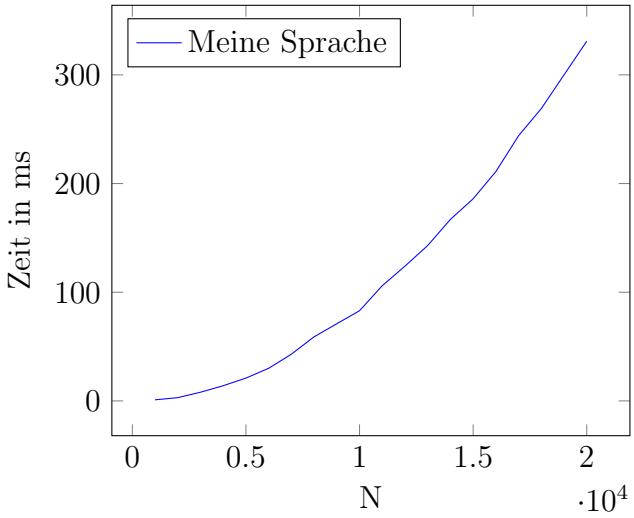
[7] Clang Compiler toolchain `https://de.wikipedia.org/wiki/Clang`

Bubble Sort (Worst Case, Listen sind absteigend sortiert)

N	C++	Python	Meine Sprache
1000	1	56	1
2000	1	230	3
3000	2	527	8
4000	5	945	14
...	...	...	...
20000	95	24531	331







Wie erwartet, hat der Bubble Sort Algorithmus in allen Sprachen eine quadratische Laufzeit $\mathcal{O}(n^2)$. Wenn die absolute Laufzeit des C++ Programm als Basiswert (also 100 %) genommen wird, dann liegt die prozentuale Laufzeit meiner Sprache bei 348,21 %. Die prozentuale Laufzeit von Python liegt bei 25822,10 %.

Weil sich C++ und meine Sprache zu LLVM-IR kompilieren lassen, kann man sich nun anschauen, warum meine Sprache langsamer ist als C++. Um Platz zu sparen, wurde die IR an manchen Stellen gekürzt.

Bubble Sort LLVM-IR

C++

```
define dso_local void @bubbleSort(ptr %0) #2 {
    %2 = alloca ptr, align 8
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    %5 = alloca i32, align 4
    %6 = alloca i32, align 4
    store ptr %0, ptr %2, align 8
    %7 = load ptr, ptr %2, align 8
    %8 = call i64 @"vector.size"(ptr %7) #2
    %9 = trunc i64 %8 to i32
    store i32 %9, ptr %3, align 4
    store i32 0, ptr %4, align 4
    br label %10

    ...

    ret void
  }

  attributes #2 = { ... }
```

Meine Sprache

```
define void @bubble_sort(%List--int* %_list)  {
entry:
    %_list_0 = alloca %List--int*
    store %List--int* %_list, %List--int** %_list_0

    ...

    %_845 = alloca i64
    %_760 = alloca i64
    %_777 = alloca i64
    %_785 = alloca i64
    %_temp_789 = alloca i64
    %_809 = alloca i64
    %_817 = alloca i64
    %_824 = alloca i64
    %_832 = alloca i64
    %_839 = alloca i64
    %_j_749 = alloca i64
    %_751 = alloca i64
    %_745 = alloca i64
    br label %while_head_748

    ...

    ret void
}
```

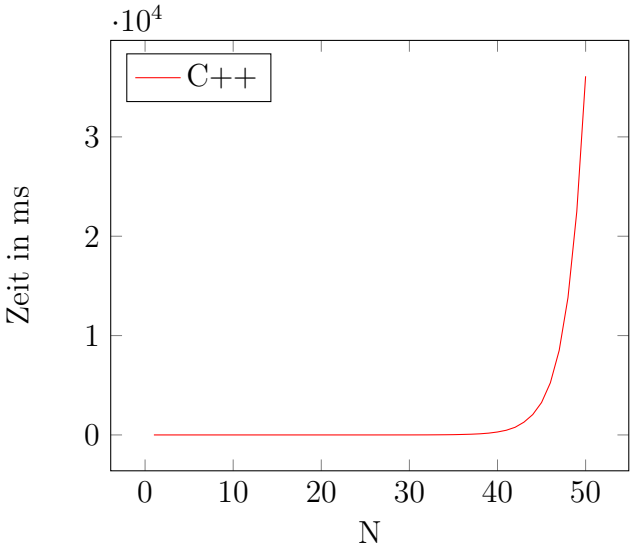
Zunächst ist zu sehen, dass in meiner Sprache mehr Speicher allokiert wird als in C++. In meiner Sprache werden 13 Variablen allokiert (Z. 8-20), in C++ nur 5 (Z. 2-6). In der C++ IR werden auch Funktionsaufrufe mit Attributen [8] versehen. Dadurch können Funktionen beim Kompilieren optimiert werden, z.B. durch spezifische Optimierung für spezifische Prozessorarchitekturen. Mein Compiler unterstützt dies nicht.

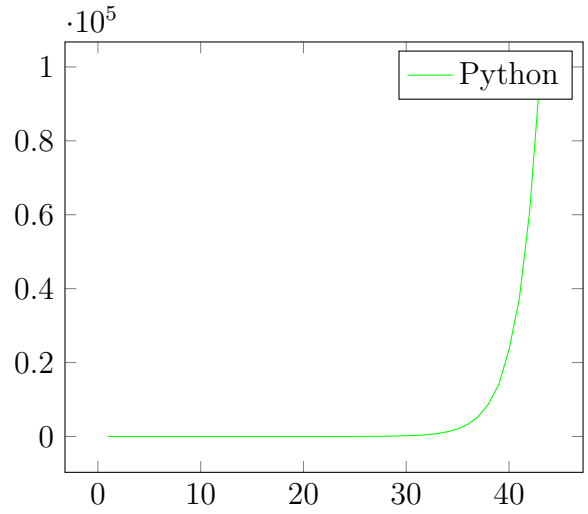
Werden diese Attribute manuell in der IR meiner Sprache hinzugefügt, dann verringert sich die relative Laufzeit von 348,21 % auf 201,05 %. Der übrige Unterschied kommt wahrscheinlich von der effizienteren Implementierung des C++ Vector. Dieser verwendet im IR, die Funktion `llvm.memmove`, um die Daten des Vector zu kopieren. In meiner Sprache wird das mit der C Funktion `memcpy` gemacht.

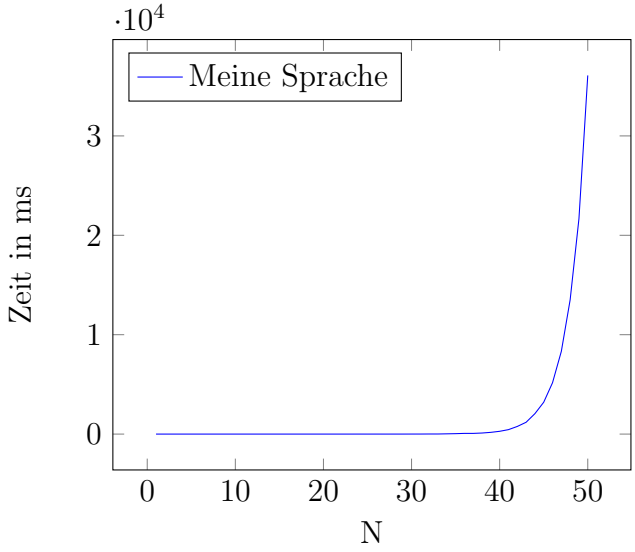
[8] Funktionsattribute in LLVM `https://llvm.org/docs/LangRef.html#attribute-groups`

Fibonacci-Algorithmus

N	C++	Python	Meine Sprache
10	0	0	0
20	0	1	0
30	2	186	3
40	296	23514	280
50	36444	-	36085







Der Fibonacci-Algorithmus hat in allen Sprachen eine exponentielle Laufzeit $\mathcal{O}(2^n)$. Die prozentuale relative Laufzeit (zu C++) meiner Sprache beträgt 99,96 %. Die prozentuale Laufzeit von Python beträgt 755,84 %. In diesem Fall ist meine Sprache sogar schneller als C++.

Wir betrachten nun den IR-Code um zu sehen, warum in diesem Fall meine Sprache schneller ist als C++.

Fibonacci-Algorithmus LLVM-IR

C++

```
define dso_local noundef i32 @"fibonacci"(i32 %0) #0 {
        %2 = alloca i32, align 4
        %3 = alloca i32, align 4
        store i32 %0, ptr %3, align 4
        %4 = load i32, ptr %3, align 4
        %5 = icmp sle i32 %4, 1
        br i1 %5, label %6, label %8
    6:
        %7 = load i32, ptr %3, align 4
        store i32 %7, ptr %2, align 4
        br label %16
    8:
        %9 = load i32, ptr %3, align 4
        %10 = sub nsw i32 %9, 1
        %11 = call noundef i32 @"fibonacci"(i32 %10)
        %12 = load i32, ptr %3, align 4
        %13 = sub nsw i32 %12, 2
        %14 = call noundef i32 @"fibonacci"(i32 %13)
        %15 = add nsw i32 %11, %14
        store i32 %15, ptr %2, align 4
        br label %16

    16:
        %17 = load i32, ptr %2, align 4
        ret i32 %17
}

attributes #0 = { ... }
```

Meine Sprache

```
define i64 @fib(i64 %_n)  {
    entry:
        %_n_0 = alloca i64
        store i64 %_n, i64* %_n_0
        %_1 = load i64, i64* %_n_0
        %_3 = alloca i64
        store i64 1, i64* %_3
        %_2 = load i64, i64* %_3
        %_4 = icmp sle i64 %_1, %_2
        br i1 %_4, label %if_5, label %end_if5
    if_5:
        %_6 = load i64, i64* %_n_0
        ret i64 %_6
        br label %end_if5
    end_if5:
        %_11 = load i64, i64* %_n_0
        %_13 = alloca i64
        store i64 1, i64* %_13
        %_12 = load i64, i64* %_13
        %_14 = sub i64 %_11, %_12
        %_9 = call i64 @fib(i64 %_14)
```

```
22              %_18 = load i64, i64* %_n_0
23              %_20 = alloca i64
24              store i64 2, i64* %_20
25              %_19 = load i64, i64* %_20
26              %_21 = sub i64 %_18, %_19
27              %_16 = call i64 @fib(i64 %_21)
28              %_22 = add i64 %_9, %_16
29              ret i64 %_22
30          }
```

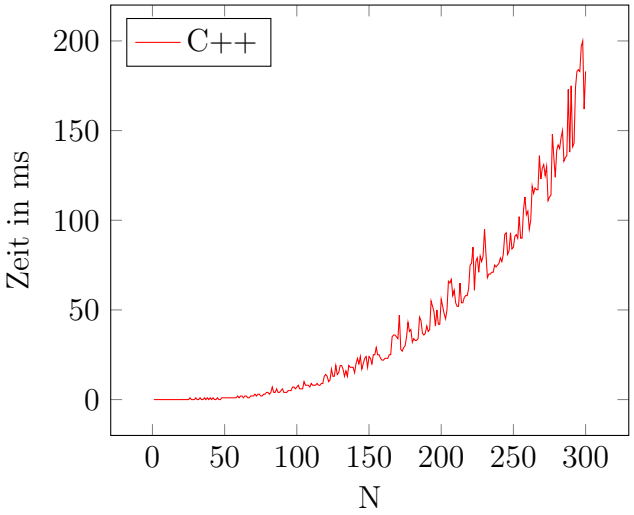
In diesem Fall ist meine Sprache schneller, weil der Datentyp int je nach nativer Wortgröße des Prozessors unterschiedlich groß ist. Auf einem 64-Bit Prozessor wird ein `int` in meiner Sprache also 64 Bit groß sein, in C++ sind `int` 32 Bit groß. Moderne Prozessoren sind für 64-Bit Operationen optimiert, daher können also Berechnungen mit 64-Bit Zahlen schneller sein.

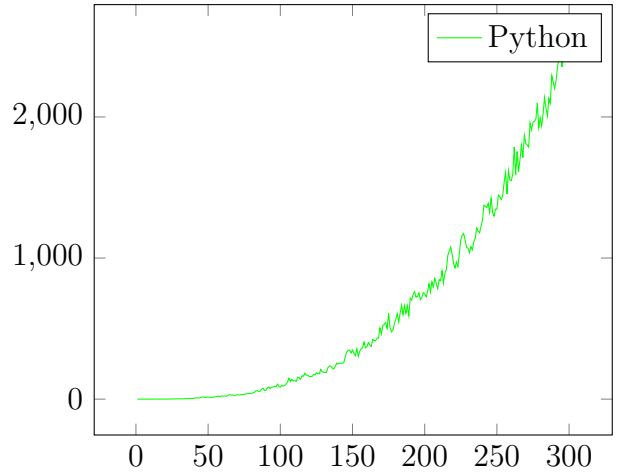
Die folgende Tabelle zeigt die Durchschnittsmesswerte aus 10 Durchläufen des C++ Algorithmus, bei denen die 47. Fibonacci-Zahl berechnet wurde. Die Integer-Größe wurde im LLVM IR nachträglich geändert.

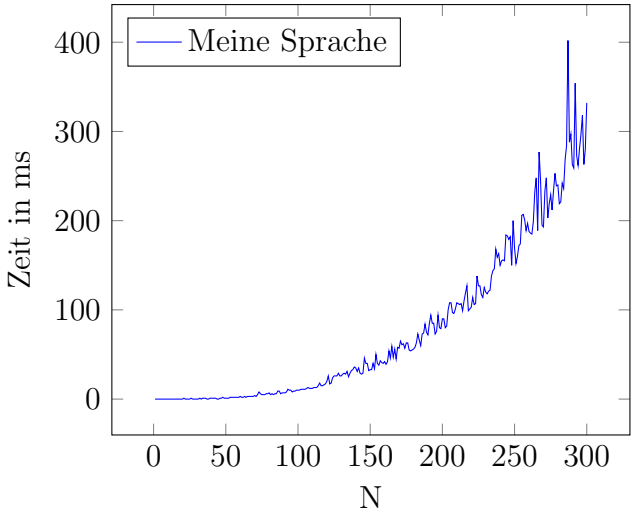
Integer Größe	Zeit in ms
32	8582
64	8501

Matrizenmultiplikation (NxN Matrizen)

N	C++	Python	Meine Sprache
50	1	13	1
100	7	85	10
150	24	348	33
200	56	748	90
250	85	1346	169
300	183	2491	322







Wie erwartet hat die Matrizenmultiplikation in allen Sprachen eine kubische Laufzeit $\mathcal{O}(n^3)$. Die prozentuale relative Laufzeit (zu C++) meiner Sprache beträgt 181,42 %. Die Laufzeit von Python beträgt 1361,20 %.

Matrizenmultiplikation LLVM-IR

C++

```
1
2       define dso_local i32 @matrix_multiply(ptr %0, ptr %1, ptr %2) # 2 {
3       entry:
4           %4 = alloca ptr, align 8
5           %5 = alloca ptr, align 8
6           %6 = alloca ptr, align 8
7           %7 = alloca i32, align 4
8           %8 = alloca i32, align 4
9           %9 = alloca i32, align 4
10          %10 = alloca i32, align 4
11          %11 = alloca i32, align 4
12          store ptr %2, ptr %4, align 8
13          store ptr %1, ptr %5, align 8
14          store ptr %0, ptr %6, align 8
15
16          ...
17
18          br label %14
19
20      14:
21          ...
22
23      28:
24          %38 = call dereferenceable(24) "vector.index"(%0, i64 %37) #12
25          ...
26          %41 = call dereferenceable(24) "vector.index"(%1, i64 %37) #12
27          %42 = load i32, ptr %41, align 4
28          %43 = mul nsw i32 %35, %42
29          %44 = load i32, ptr %10, align 4
30          %45 = add nsw i32 %44, %43
31          store i32 %45, ptr %10, align 4
32          br label %46
33      46:
34          %47 = load i32, ptr %9, align 4
35          %48 = add nsw i32 %47, 1
36          store i32 %48, ptr %9, align 4
37          br label %24, !llvm.loop !13
38      64:
39          ...
40
41          ret i32 0
42      }
43
44      attributes #2 = { ... }
45      atributtes #12 = { ... }
```

Meine Sprache

```
1       define %List @matrix_multiplication(%List %_a, %List %_b) {
2       entry:
3           %_a_0 = alloca %List*
4           store %List* %_a, %List** %_a_0
5           %_b_0 = alloca %List*
6           store %List* %_b, %List** %_b_0
7
8           ...
9       while_body_1607:
10          %_1610 = load i64, i64* %_sum_1597
```

```
11          %_1615 = load i64, i64* %_i_1578
12          %_1619 = load i64, i64* %_i_1578
13          %_1617 = call %List* @List_Index_List_int--List(%List* %_a_1550,i64 %_1619)
14          %_1620 = load i64, i64* %_k_1600
15          %_1625 = load i64, i64* %_i_1578
16          %_1629 = load i64, i64* %_i_1578
17          %_1627 = call %List* @List_Index_List_int--List(%List* %_a_1550,i64 %_1629)
18          %_1630 = load i64, i64* %_k_1600
19          %_1622 = call i64* @List_Index_List_int--int(%List* %_1627,i64 %_1630)
20          %_1612 = load i64, i64* %_1622
21          %_1634 = load i64, i64* %_k_1600
22          %_1638 = load i64, i64* %_k_1600
23          %_1636 = call %List* @List_Index_List_int--List(%List* %_b_1554,i64 %_1638)
24          %_1639 = load i64, i64* %_j_1589
25          %_1644 = load i64, i64* %_k_1600
26          %_1648 = load i64, i64* %_k_1600
27          %_1646 = call %List* @List_Index_List_int--List(%List* %_b_1554,i64 %_1648)
28          %_1649 = load i64, i64* %_j_1589
29          %_1641 = call i64* @List_Index_List_int--int(%List--int* %_1646,i64 %_1649)
30          %_1631 = load i64, i64* %_1641
31          %_1650 = mul i64 %_1612, %_1631
32          %_1651 = add i64 %_1610, %_1650
33          store i64 %_1651, i64* %_sum_1597
34          ...
35      end_while_1596:
36          %_1676 = load %List--List--int, %List--List--int* %_result_1547
37          ret %List--List--int %_1676
38      }
```

Meine Sprache ist hier wieder langsamer als C++. Dies liegt erneut an den Funktionsattributen, aber auch an der Anzahl der Anweisungen, die durchgeführt werden. In C++ werden 15 Anweisungen (Z. 24f) benötigt, um das Produkt beider Matrixelemente zu berechnen, in meiner Sprache sind es 30 (Z. 10f). Mein Compiler erzeugt IR, die an vielen Stellen mehr Anweisungen verwendet als absolut nötig.

3 Entwicklung & Verwendung des Compilers

3.1 Verwendung des Compilers

Der Compiler wird in der Kommandozeile ausgeführt. Um den Programmcode zu kompilieren, wird der Pfad der Hauptdatei (also die Datei, in der sich die `main` Funktion befindet) als Argument übergeben. Sofern eine lokale Installation des Clang Compilers vorhanden ist, verwendet mein Compiler diese um die LLVM IR in eine ausführbare Datei umzuwandeln. Die ausführbare Datei wird im cwd (current working directory, also in dem Pfad in dem der Compiler ausgeführt wird) erstellt.
Es gibt folgende optionale Argumente:

- `-e, --emit-llvm`: Der Compiler gibt die LLVM-IR aus.
- `-o, --output`: Der Pfad der ausführbaren Datei (standardmäßig `./out.exe`).
- `-h, --help`: Gibt die Hilfe aus.
- `-v, --version`: Gibt die Version des Compilers aus.

3.2 Arten von Programmiersprachen

3.2.1 Interpreter

Interpreter[9] sind Programme, die interpretierte Programmiersprachen meist Zeile für Zeile ausführen. Das hat den Vorteil, dass interpretierte Sprachen leichter zu debuggen sind und auf allen Plattformen funktionieren, auf denen der Interpreter verfügbar ist. Bekannte Beispiele sind Python und JavaScript. Ein Nachteil ist, dass interpretierte Sprachen einen Interpreter benötigen, um ausgeführt zu werden und meist um ein Vielfaches langsamer sind als kompilierte Sprachen.

3.2.2 Compiler

Compiler[10] übersetzen den Quellcode eines Programms in eine Form, die direkt von Computern ausgeführt werden kann. Bekannte Beispiele sind C, C++, Rust und Go. Der Vorteil von kompilierten Sprachen ist, dass kein externes Programm benötigt wird, um Programme auszuführen.

[9] Interpreter: `https://de.wikipedia.org/wiki/Interpreter`
[10] Compiler: `https://de.wikipedia.org/wiki/Compiler`

3.2.3 Transpiler

Transpiler[11] oder Source-to-Source (S2S) Compiler übersetzen eine Programmiersprache in eine andere. Dabei sind beide Sprachen in der Regel von Menschen lesbar. Bekannte Beispiele sind TypeScript und V-lang.

3.3 LLVM IR

Bei *LLVM IR*[12] (LLVM Intermediate Representation) handelt es sich um eine plattformunabhängige Zwischensprache, die sehr wenig Abstraktion bietet und daher im Umfang und in den Funktionen recht simpel ist. Diese Zwischensprache kann von dem LLC (LLVM static compiler) oder einem Compiler, der diesen verwendet (z.B. Clang), in plattformabhängige Maschinensprache übersetzt werden. LLVM IR wird von Compilern wie Clang, Clang++ oder Rustc verwendet.

3.4 Compilerarchitektur

Nun werde ich auf die Architektur meines Compilers eingehen. Zunächst habe ich mich für eine kompilierte Sprache entschieden, da ich keine zur Ausführung der Programme keine externe Laufzeitumgebung haben wollte.

3.4.1 Frontend

Lexer[13]

Das Compiler Frontend ist der Teil eines Compilers, der den Quellcode eines Programms liest und in eine Datenstruktur übersetzt (meist ein abstrakter Syntaxbaum (AST)).

Der Quellcode wird zu Beginn von einem Lexer in eine Liste von Tokens übersetzt. Dabei wird Zeichen für Zeichen vorgegangen.

Ein Hello World Programm wird folgendermaßen in Tokens übersetzt:

[11] Transpiler: `https://en.wikipedia.org/wiki/Source-to-source_compiler`
[12] LLVM IR: `https://llvm.org/docs/LangRef.html`
[13] Codeverweis Tokens: `src/lexer/tokens.rs`, Lexer: `src/lexer/lexer_main.rs`

```
use "std/io.mx"

def main() -> int {
    println("Hello, World!");
    return 0;
}
```

Quellcode	Token
use	Keyword::Use
"std/io.mx"	String "std/io.mx"
def	Keyword::Def
main	Identifier "main"
(	Punctuation::OpenParen
)	Punctuation::CloseParen
->	Punctuation::ThinArrow
int	Identifier ïnt"
{	Punctuation::OpenBrace
println	Identifier "println"
(	Punctuation::OpenParen
"Hello, World!"	String "Hello, World!"
)	Punctuation::CloseParen
;	Punctuation::Semicolon
return	Keyword::Return
0	Literal::Integer 0
;	Punctuation::Semicolon
}	Punctuation::CloseBrace

Während der Tokenisierung werden für jedes Token auch noch die Positionen im Quellcode mitgespeichert, um später bei Fehlermeldungen die Position des Fehlers anzeigen zu können.

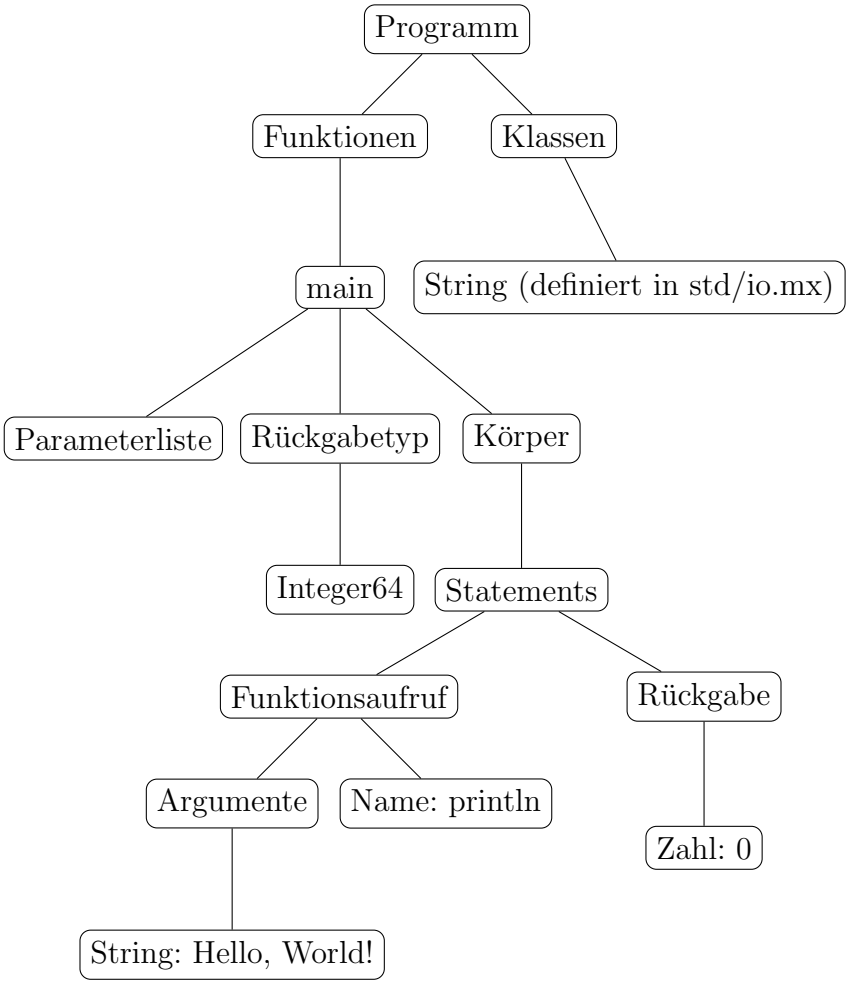
Parser und AST[14]

Anschließend wird ein `Parser` Objekt erstellt. Dieser geht nun die Liste von Tokens der Reihe nach durch und erstellt einen abstrakten Syntaxbaum (AST). Der Parser hat einen Zustand, der sich bei der Erstellung des Syntaxbaumes verändert. Der Parser speichert z.B. definierte Funktionen und

[14]Codeverweis Parser: `src/parser/parser_main.rs`, AST: `src/parser/ast.rs`

Klassen in separaten Listen. Wenn nun z.B. eine Klasse `Foo` definiert wird, erstellt der Parser intern eine Klassenstruktur und speichert diese ab. Wenn nun im Programm ein `Token::Identifier "Foo"` gefunden wird, kann der Parser auf die gespeicherte Klasse `Foo` zugreifen. Selbes gilt auch für Funktionen und Variablen.

Nachdem ein Syntaxbaum erstellt wurde, sieht er vereinfacht so aus:



Die `String` Klasse wird im Parser folgendermaßen gespeichert:

Anzeigename	String	
Name	String	
Felder	Name	Typ
	Datenzeiger	*int8
	Länge	int64
	Kapazität	int64
Subtypen	-	
Generics	-	
Subtyp von	-	

Generics[15]

Da meine Sprache auch Generics in Funktionen und Klassen unterstützt, werde ich erklären wie diese in meinem Compiler implementiert sind.

Wie in der oberen Tabelle zu sehen ist, besitzt jede Klasse die Felder `Generics`, `Subtypen` und `Subtyp von`. Diese verwendet mein Compiler, um dynamische Unterklassen zu erstellen. Dies passiert, wenn eine generische Klasse mit einem spezifischen Typen instanziiert wird. Wenn es z.B. eine generische Klasse `List<T>` gibt, und ein Datenobjekt erstellt wird, bei dem `T` durch `int` ersetzt wird (`List<int>`), wird eine neue Klasse+ `ListInt` erstellt, die alle Vorkommen von `T` in Feldern und Methoden durch `int` ersetzt.

Das folgende Beispiel veranschaulicht diese dynamische Typerstellung (im echten Compiler wird dies während der Erstellung des Syntaxbaums gemacht, Code in meiner Sprache wird für Generics also nicht erzeugt):

[15] Codeverweis: `src/parser/utils.rs`

Quellcode	"Generierter Code"

`class Foo<T> {`
`  data: T,`
`}`
 	
`class Bar {`
`  data: int,`
`}`
 	
`def main() -> int {`
`  let bar = Bar {`
`  data: 5`
`  };`
`  return bar.data;`
`}`
	`class Bar {`
	`  data: int,`
	`}`
	
	`def main() -> int {`
	`  let bar = Bar {`
	`  data: 5`
	`  };`
	`  return bar.data;`
	`}`

`class Foo<T> {`
`  data: T,`
`}`
 	
`def main() -> int {`
`  let foo = Foo<int> {`
`  data: 5`
`  };`
`  return foo.data;`
`}`
	`class FooInt {`
	`  data: int,`
	`}`
	
	`def main() -> int {`
	`  let foo = FooInt {`
	`  data: 5`
	`  };`
	`  return foo.data;`
	`}`

In den Beispielen ist zu erkennen, dass die tatsächlichen generischen Klassen nie im AST vorkommen. Jedes Mal eine Instanz dieser Klasse, in welche der generische Typ spezifiziert ist, im Quellcode vorkommt, dann wird im AST eine neue Klasse erstellt, die diesen generischen Typ ersetzt.

Methoden und Operatorüberladung

Methoden werden in meinem Compiler nicht direkt in den Klassenstrukturen gespeichert, sondern in einer separaten Map, die jedem Klassennamen eine Liste von Methoden zuordnet.

Jeder Operator kann durch eine Methode mit vorgegebenem Namen überschrieben werden. Wenn z.B. zwei Objekte `Foo` und `Bar` addiert werden,

wird überprüft, ob das Objekt `Foo` eine Methode namens `add` besitzt, die diese Signatur hat: `fn add(self, other: Bar) -> Beliebiger Typ`. Ist diese Bedingung erfüllt, wird die Addition durch einen Aufruf dieser Methode durchgeführt. Ist die Bedingung nicht erfüllt wird, sofern vorhanden eine Standardimplementierung der Addition verwendet. Diese ist z.B. bei eingebauten Typen wie `int` oder `float` vorhanden.

3.4.2 Backend & Codegenerierung

Das Backend eines Compilers ist der Teil, der den AST in die Zielsprache übersetzt, in meinem Fall LLVM IR. Die folgende Tabelle enthält alle LLVM IR Befehle, die mein Compiler verwendet:

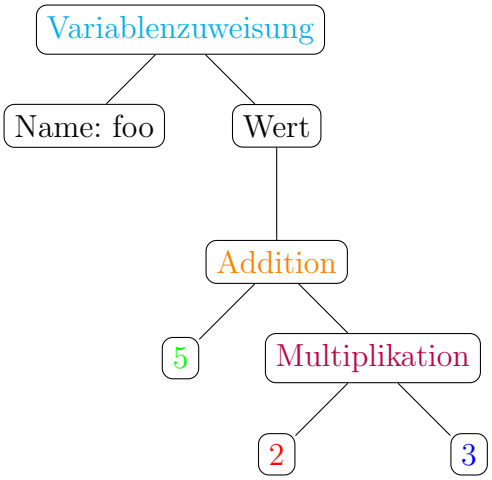
Befehlsname	Beispiel
Stack allokation	`alloca i64`
Speicherzugriff	`load i64, i64* %0`
in Speicher schreiben	`store i64 %0, i64* %1`
Element einer Struktur lesen	`getelementptr %struct.Foo, ..., i64 %0`
Pointer zu Integer cast	`ptrtoint i64*, i64`
Integer zu Pointer cast	`inttoptr i64, i64*`
binäre Operation	z.B. `add i64 %0, %1`
Variable deklaration	`%0 = alloca i64`
Sprung	`br label %1`
konditionaler Sprung	`br i1 %0, label %1, label %2`
Funktionsaufruf	`call void @foo(i64 %0)`
Integer Up-/Downcast	`sext i32 %0 to i64`

Mithilfe dieser Befehle und mit Funktionen aus der C-Standardbibliothek lassen sich in meiner Sprache komplexe Programme schreiben.

[15] Codeverweis IR Befehle: `src/codegen/llvm_instructions.rs`, Codegenerierung: `src/codegen/codegen_main.rs`

Das folgende Beispiel zeigt, wie der AST eines Rechenausdrucks meiner Sprache in LLVM IR übersetzt wird:

```
let foo = 5 + 2 * 3;
```




```
%_3 = alloca i64
store i64 5, i64* %_3
%_2 = load i64, i64* %_3
%_6 = alloca i64
store i64 2, i64* %_6
%_5 = load i64, i64* %_6
%_8 = alloca i64
store i64 3, i64* %_8
%_7 = load i64, i64* %_8
%_9 = mul i64 %_5, %_7
%_10 = add i64 %_2, %_9
%_foo_0 = alloca i64
store i64 %_10, i64* %_foo_0
```

Es ist zu erkennen, dass mein Compiler den Rechenausdruck zerlegt und die Operatorrangfolge beachtet. Jede im Ausdruck vorkommende Zahl wird in 3 Befehle übersetzt:

- `alloca i64` allokiert Speicher für die Zahl
- `store i64 <Zahl>, i64* %<Variable>` schreibt die Zahl in den allokierten Speicher
- `load i64, i64* %<Variable>` lädt die Zahl aus dem allokierten Speicher

Mein Compiler speichert die Zahl also eigentlich unnötigerweise, man könnte auch die Berechnung ohne ein Zwischenspeichern durchführen: Eine optimierte Version könnte z.B. so aussehen:

```
%_0 = add i64 5, 0
%_1 = add i64 2, 0
%_2 = add i64 3, 0
%_3 = mul i64 %_1, %_2
```

```
%_4 = add i64 %_0, %_3
%_foo_0 = alloca i64
store i64 %_4, i64* %_foo_0
```

oder sogar noch kürzer:

```
%_0 = mul i64 2, 3
%_1 = add i64 5, %_0
%_foo_0 = alloca i64
store i64 %_1, i64* %_foo_0
```

Um den generierten LLVM IR Code nun in eine ausführbare Datei umzuwandeln, verwendet mein Compiler eine lokale Installation des Clang Compilers (sofern vorhanden).

4 Fazit

4.1 Ergebnisse

In meiner Sprache ist es möglich einfache, aber auch komplexe Programme zu schreiben. Vor allem durch Generics und Operatorüberladung lassen sich komplexere Probleme einfacher lösen.

Es ist möglich Programme direkt als ausführbare Datei zu kompilieren, oder sie in LLVM IR kompilieren. Fehlermeldungen sind in den meisten Fällen sehr ausführlich und hilfreich.

Weil sowohl ein eigener Lexer als auch ein eigener Parser implementiert wurde, ist der Compiler recht flexibel gestaltet und kann leicht erweitert oder angepasst werden.

Und während der Entwicklung konnte ich viel über Programmiersprachen und Compilerarchitektur lernen.

4.2 Verbeserungsmöglichkeiten

Es fehlen einige Features, die normalerweise in einer modernen Programmiersprache vorhanden sind:

- Multithreading ist nicht möglich
- Es gibt keine Bitoperationen
- Es gibt nur öffentliche Funktionen und Klassen
- Es gibt keine Namespaces

Folgende Dinge könnten verbessert werden:

- Der IR Code wird bis zum Schluss in einer Liste gespeichert. Bei größeren Programmen wäre es sinnvoller direkt bei der Generierung den IR Code in die Datei zu schreiben.
- Der Codegenerator ist single-threaded, Multithreading könnte z.B. bei dem Importieren von Dateien oder bei der Generierung des IR Codes implementiert werden.
- Eventuell könnte ich mir noch einen Namen für meine Sprache ausdenken.

- Ein Rust ähnliches Trait-Bound System wäre sinnvoll, um Generics vollständig Typensicher zu machen.
- Eine automatische Freigabe des Heap-Speichers ähnlich wie in Rust wäre sinnvoll (Ownership-System).

5 Anhang

Zugehöriges Github Repository: https://github.com/maxomatic458/compiler
Sonstige Quellen:

- https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/ (19.06.2024)
- https://github.com/jDomantas/plank/ (19.06.2024)
- https://youtu.be/apFUyLupFgE?si=unRU7SobjeODhn_d (19.06.2024)
- https://llvm.org/docs/LangRef.html (19.06.2024)
- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html (19.06.2024)
- https://de.wikipedia.org/wiki/Compiler (19.06.2024)
- https://github.com/llvm/llvm-project (19.06.2024)
- https://en.wikipedia.org/wiki/Operator-precedence_parser (20.06.2024)

Ebenfalls wurden folgende Werkzeuge verwendet:

- rust-clippy 0.1.77 https://github.com/rust-lang/rust-clippy
- typos-cli 1.15.4 https://github.com/crate-ci/typos
- rustfmt 1.7.0-stable https://github.com/rust-lang/rustfmt
- rust 1.77 stable https://www.rust-lang.org/
- Visual Studio Code 1.90.2 https://code.visualstudio.com/
- rust-analyzer v0.4.2005 (pre-release) https://github.com/rust-lang/rust-analyzer
- TeX Live 2024 3.141592653 https://tug.org/texlive/
- Vscode-ltex https://github.com/valentjn/vscode-ltex

Es wurden folgende Rust-Bibliotheken[16] bei der Entwicklung verwendet:

[16] Codeverweis: src/Cargo.toml

- clap 4.4.11 https://crates.io/crates/clap
- codespan-reporting https://github.com/brendanzab/codespan
- color-eyre 0.6.2 https://crates.io/crates/color-eyre
- derive˙more 0.99.17 https://crates.io/crates/derive_more
- indexmap 2.2.3 https://crates.io/crates/indexmap
- itertools 0.12.0 https://crates.io/crates/itertools
- lazy_static 1.4.0 https://crates.io/crates/lazy_static
- once_cell 1.18.0 https://crates.io/crates/once_cell
- phf 0.11.2 https://crates.io/crates/phf
- pretty˙assertions 1.4.0 https://crates.io/crates/pretty_assertions
- rstest 0.18.2 https://crates.io/crates/rstest
- semver 1.0.20 https://crates.io/crates/semver
- serde 1.0.188 https://crates.io/crates/serde
- serial˙test 3.0.0 https://crates.io/crates/serial_test
- strum 0.26.1 https://crates.io/crates/strum
- strum_macros 0.26.1 https://crates.io/crates/strum_macros
- termcolor 1.2.0 https://crates.io/crates/termcolor
- thiserror 1.0.47 https://crates.io/crates/thiserror
- unescape 0.1.0 https://crates.io/crates/unescape
- criterion 0.5.1 https://crates.io/crates/criterion

Es wurden die folgenden TeX-Pakete verwedet:

- babel https://ctan.org/pkg/babel
- upquote https://ctan.org/pkg/upquote

- listings https://ctan.org/pkg/listings
- inconsolata https://ctan.org/pkg/inconsolata
- breqn https://ctan.org/pkg/breqn
- hyperref https://ctan.org/pkg/hyperref
- multicol https://ctan.org/pkg/multicol
- tikz https://ctan.org/pkg/tikz
- changepage https://ctan.org/pkg/changepage
- verbatimbox https://ctan.org/pkg/verbatimbox

Während der Entwicklung und beim Schreiben der BLL wurden folgende KI-Werkzeuge verwendet (z.B. für Formattierung von Tex-Elementen oder für das Erstellen von Unit-Tests):

- ChatGPT https://chatgpt.com/
- Github Copilot https://copilot.github.com/

Python Vergleichsalgorithmen
Bubble Sort:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

Fibonacci:

```
    def fib(n: int) -> int:
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Matrizenmultiplikation:

```
def matrix_multiplication(a: list[list[int]], b: list[list[int]])
    -> list[list[int]]:
    n = len(a)
    m = len(b[0])
    k = len(b)
    c = [[0 for _ in range(m)] for _ in range(n)]
    for i in range(n):
        for j in range(m):
            for l in range(k):
                c[i][j] += a[i][l] * b[l][j]
    return c
```

Die Daten der Benchmarks befinden sich im Ordner `bll/data` als CSV-Dateien. Weitere Codebeispiele meiner Sprache (inklusive einer kleinen Standardbibliothek) sowie die verwendeten Benchmark-Algorithmen befinden sich im Ordner `bll/example`.