



Die Umwandlung von Quelltext in Maschinensprache

Silas Groh, Mik Müller

17. Mai 2023

Carl-Fuhlrott-Gymnasium

Einstieg & Motivation

- Programme werden in speziellen Sprachen verfasst
- Vorteile eines hohen Abstraktionsgrades [Dan05, S. 9]



Erweiterbarkeit &
Reperatur



Portabilität &
Platformunabhängigkeit

[Goo23]



Geschwindigkeit &
Einfachheit

```
1 fn main() {  
2     foo(2);  
3 }  
4  
5 fn foo(n: int) {  
6     let mut m = 3;  
7     exit(n + m);  
8 }
```

Vorgang 



- Programme sollten **einfach** zu schreiben sein
- ⇒ Ein Computer muss diese jedoch auch **einfach** verarbeiten

- Man unterscheidet zwischen **Compilern** und **Interpretern**
- Compiler: übersetzt das Programm in ein Zielformat
- Interpreter: führt das Programm direkt aus (keine Übersetzung)

```
1 fn main() {  
2     foo(2);  
3 }  
4  
5 fn foo(n: int) {  
6     let mut m = 3;  
7     exit(n + m);  
8 }
```


Ausführung ⚙️



Exit code: 5

- Python, Javascript, PHP, usw.
 - Keine Übersetzung notwendig
- ⇒ Interpretiert den Syntaxbaum direkt

```
1 fn main() {  
2     foo(2);  
3 }  
4  
5 fn foo(n: int) {  
6     let mut m = 3;  
7     exit(n + m);  
8 }
```

Übersetzung 



```
1 ; RISC-V binary  
2 457f 464c 0102  
3 0002 00f3 0001  
4 0040 0000 0000  
5 0005 0000 0040  
6 0003 7000 0004  
7 0000 0000 0000  
8 0048 0000 0000  
9 0001 0000 0000  
10 0000 0000 0000
```

- Rust, C, Go, usw.
 - Zusätzlicher Prozess
 - Umwandlung in ein anderes Format
- ⇒ Muss vor der Ausführung stattfinden

Die Programmiersprache „rush“



- ca. vier Monate intensive Entwicklung
- 814 Git Commits
- 17526 Zeilen Programmtext¹ in Git Commit '9953dd8'

¹Leerzeilen und Kommentare werden nicht gezählt.

- Lexer
- Parser
- Semantikanalyse
- zwei Interpreter
- ein Transpiler
- vier Compiler

| Bezeichnung | Beispiel |
|---------------------|--|
| Schleife | <code>loop { }</code> |
| while-Schleife | <code>while x < 5 { }</code> |
| for-Schleife | <code>for i = 0; i < 5; i += 1 { }</code> |
| if-Verzweigung | <code>if true { } else { }</code> |
| Funktionsdefinition | <code>fn foo(n: int) { }</code> |
| infix-expression | <code>1 + n; 5 ** 2</code> |
| prefix-expression | <code>!false; -n</code> |
| let-statement | <code>let mut answer = 42</code> |
| cast-expression | <code>42 as float</code> |

| Bezeichnung | Instanziierung einer Variable |
|-------------|-----------------------------------|
| „int“ | <code>let a: int = 0;</code> |
| „float“ | <code>let b: float = 3.14;</code> |
| „bool“ | <code>let c: bool = true;</code> |
| „char“ | <code>let d: char = 'a';</code> |
| „()“ | <code>let e: () = main();</code> |
| „!“ | <code>let f = exit(42);</code> |

Berechnung von Fibonaccizahlen in rust

```
1 fn main() {  
2     exit(fib(10));  
3 }  
4  
5 fn fib(n: int) -> int {  
6     if n < 2 {  
7         n  
8     } else {  
9         fib(n - 2) + fib(n - 1)  
10    }  
11 }
```

$$f_n = f_{n-2} + f_{n-1} \quad ; n \geq 3$$

$$f_1 = f_2 = 1$$

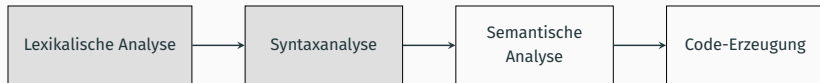
Umfang der einzelnen Komponenten

| Komponente | Zeilen Programmtext |
|--------------------------|---------------------|
| Lexer / Parser | 2737 |
| Tree-walking interpreter | 578 |
| VM compiler / runtime | 1288 |
| WASM compiler | 1641 |
| LLVM compiler | 1450 |
| RISC-V compiler | 2234 |
| x86 compiler | 2751 |



Lexikalische & Syntaktische Analyse

Etappen der Übersetzung: Semantische Analyse

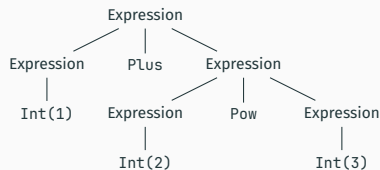
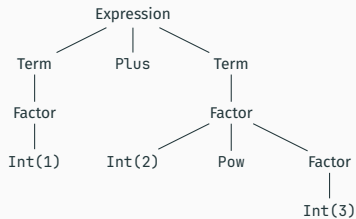


[Wir05, S. 6–7]

- Gruppieren des Programmtextes in Tokens
- Analyse der Syntax des Programms
- Erzeugung eines abstrakten Syntaxbaums
- Festlegen der formalen Regeln in Form einer Grammatik

```
1 Expression = Term , { ( '+' | '-' ) , Term } ;
2 Term       = Factor , { ( '*' | '/' ) , Factor } ;
3 Factor     = ( integer
4             | '(' , Expression , ')' ) , [ '**' , Factor ] ;
5 integer    = { '0' | '1' | '2' | (* ... *) | '9' }- ;
```

Ein Beispiel für eine kontextfreie Grammatik (EBNF)



Zwei verschiedene Syntaxbäume für „1+2*3“

Semantische Analyse

Etappen der Übersetzung: Semantische Analyse



[Wir05, S. 6–7]

- Validiert die semantische Eigenschaften
- Meistens: Definition in einer natürlichen Sprache

Beispiel 1: Invalides rush Programm

```
1 fn main() {  
2     let num = 3.1415;  
3     num + 1;  
4 }
```



Fehlerausgabe

TypeError at incompatible_types.rush:3:5

```
2 |     let num = 3.1415;  
3 |     num + 1;  
   |           ^^^^^^^  
4 | }
```

infix expressions require equal types on both sides, got `float` and `int`

Beispiel 2: Invalides rush Programm

```
1 fn main(_n: int) -> bool {  
2     return true;  
3 }
```



Fehlerausgabe

SemanticError at invalid_main_fn.rush:1:8

```
1 | fn main(_n: int) -> bool {  
    ^^^^^^^^^^^  
2 |     return true;
```

the ``main`` function must have 0 parameters, however 1 is defined

note: remove the parameters: ``fn main() { ... }``

SemanticError at invalid_main_fn.rush:1:21

```
1 | fn main(_n: int) -> bool {  
    ^^^^^  
2 |     return true;
```

the ``main`` function's return type must be ``()``, but is declared as ``bool``

note: remove the return type: ``fn main() { ... }``

Beispiel 3: Warnung aufgrund einer unbenutzten Variable

```
1 fn main() {  
2     let variable = 42;  
3     let mut code = 3;  
4     exit(code);  
5 }
```



Ausgabe

Info at unused_var.rush:3:13

```
2 |     let variable = 42;  
3 |     let mut code = 3;  
   |                   ~~~~  
4 |     exit(code);
```

variable `code` does not need to be mutable

Warning at unused_var.rush:2:9

```
1 | fn main() {  
2 |     let variable = 42;  
   |     ~~~~~  
3 |     let mut code = 3;
```

unused variable `variable`

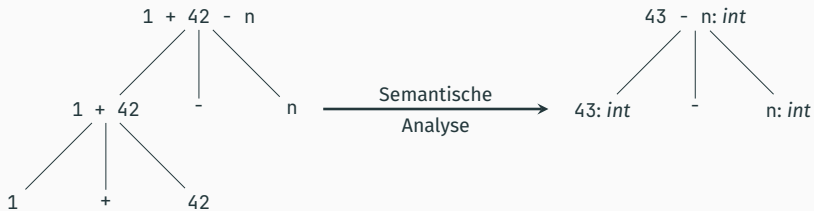
note: if this is intentional, change the name to `_variable` to hide this

↪ warning

Anforderungen an die semantische Analyse (für rush)

- Unterscheidung zwischen validen und invaliden Programmen
- Liefern von hilfreichen Warnungen und Informationen
- Hinzufügen von Typinformationen zu dem AST
- Triviale Optimierungen der Programmstruktur

Hinzufügen von Informationen über Datentypen



Auswirkungen der semantischen Analyse auf den AST

Tree-Walking Interpreter

- *Traversiert* den Syntaxbaum
- *Interpretiert* das Programm direkt

```
_____ crates/rush-interpreter-tree/src/interpreter.rs _____  
10 type Scope<'src> = HashMap<&'src str, Rc<RefCell<Value>>>;  
11  
12 #[derive(Debug, Default)]  
13 pub struct Interpreter<'src> {  
14     scopes: Vec<Scope<'src>>,  
15     functions: HashMap<&'src str, Rc<AnalyzedFunctionDefinition<'src>>>,  
16 }
```

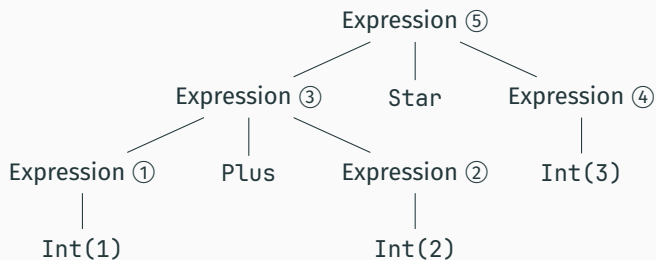
scopes Stack von Scopes; jeder Scope weist einem Variablennamen einen Laufzeitwert zu

functions Zuweisung von Funktionsnamen zu einer geteilten Referenz zu dem entsprechenden Knoten im Syntaxbaum

Weitere Typdefinitionen

```
— crates/rush-interpreter-tree/src/value.rs —  
6  pub enum Value {  
7      Int(i64),  
8      Float(f64),  
9      Char(u8),  
10     Bool(bool),  
11     Unit,  
12     Ptr(Rc<RefCell<Value>>),  
13 }  
// ...  
22 pub enum InterruptKind {  
23     Return(Value),  
24     Break,  
25     Continue,  
26     Error(interpreter::Error),  
27     Exit(i64),  
28 }
```

- Aufzählung zum Speichern verschiedener Datentypen
- Verschiedene Unterbrechungen des Programmflusses



Syntaxbaum zu „1 + 2 * 3“


```
1 fn fib(n: int) -> int {  
2   if n < 2 {  
3     n  
4   } else {  
5     fib(n - 2) + fib(n - 1)  
6   }  
7 }
```

| |
|------------------------------|
| call_func("fib", vec![3]) |
| visit_block(/* ... */) |
| visit_expression(/* ... */) |
| visit_if_expr(/* ... */) |
| visit_block(/* ... */) |
| visit_expression(/* ... */) |
| visit_inifix_expr(/* ... */) |
| visit_expression(/* ... */) |
| visit_call_expr(/* ... */) |
| call_func("fib", vec![2]) |
| ... |
| call_func("fib", vec![1]) |



Virtuelle Maschine

- Oft: Eine *Virtuelle Maschine* (VM) simuliert echte Computer
 - Display
 - Lautsprecher
 - Festplatte
 - ...
- Hier: Software, die wie die CPU eines Rechners funktioniert

- Die meisten Prozessoren basieren auf der *von Neumann Architektur* [Led20, p. 172]
- Eine CPU enthält nach von Neumann ein *Rechenwerk*², *Steuerwerk*³, *Speicherwerk*, *Ein- / Ausgabewerk* und ein Bussystem [Led20, p. 172]
- Die Programmausführung wird durch den sog. *Befehlszyklus*⁴ modelliert [Led20, pp. 208-209]:
 1. **Fetch** (Befehl laden): Das Steuerwerk lädt die nächste Anweisung aus dem Speicher
 2. **Decode** (Befehl dekodieren): Der Befehlscode und die Operanden werden ermittelt
 3. **Execute** (Befehl ausführen): Die zuständige Einheit im Prozessor wird verwendet, um den Befehl zu verarbeiten. Beispielsweis wird das Rechenwerk für logische und mathemtische Befehle aufgerufen.

²Engl: „arithmetic logic unit“ (ALU).

³Engl: „control unit“.

⁴Engl: „fetch-decode-execute cycle“.

Übertragung der Konzepte auf die rush VM **TODO: DELETE**

```
crates/rush-interpreter-vm/src/vm.rs
```

```
16 pub struct Vm<const MEM_SIZE: usize> {
17     /// Working memory for temporary values
18     stack: Vec<Value>,
19     /// Linear memory for variables.
20     mem: [Option<Value>; MEM_SIZE],
21     /// The memory pointer points to the last free location in memory.
22     /// The value is always positive, but using `isize` does not require
    ↪ casts.
23     mem_ptr: isize,
24     /// Holds information about the current position (like ip / fp).
25     call_stack: Vec<CallFrame>,
26 }
```

stack Speicher für temporäre Werte bei komplexeren Operationen

mem Anhaltender Speicher mit einer festen Größe für Variablen

mem_ptr Hält den Index der letzten freien Speicherzelle in „mem“

call_stack Aufrufstapel, welcher den *Befehlszähler* und den *Funktionszähler* für jeden Aufruf speichert

- Stack für temporäre Operationen
- Weiterer Stack für Funktionsaufrufe
- Unterteilung in Funktionen
 - Ohne Namen
 - numerische Identifizierung
 - Enthält mehrere Anweisungen
- ca. 30 verschiedene Befehlscodes
- Struktur der Anweisungen: „call 2“
 - Befehlscode (call)
 - Optionaler Operand (2)

Demonstration: Ein-/Ausgabe

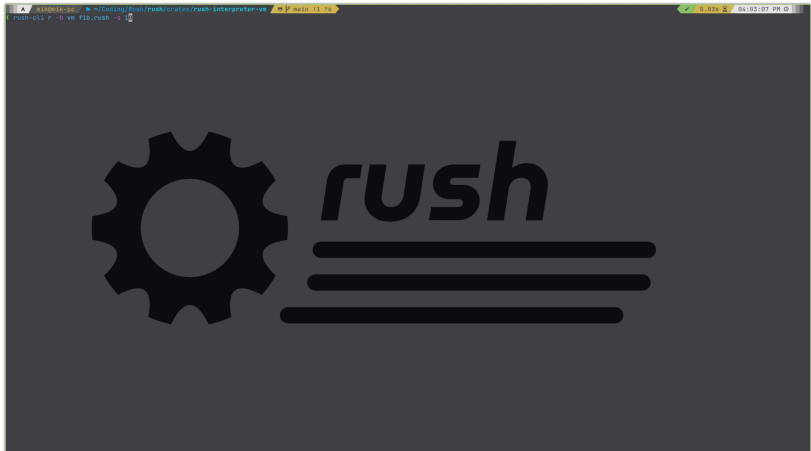
```
1  fn main() {
2      exit(fib(10));
3  }
4
5  fn fib(n: int) -> int {
6      if n < 2 {
7          n
8      } else {
9          fib(n - 2) + fib(n - 1)
10     }
11 }
```

Ausgabe



```
1  0: (prelude)
2      setmp 0
3      call 1
4  1: (main)
5      setmp 0
6      push 10
7      call 2
8      exit
9  2: (fib)
10     setmp 1
11     svari *rel[0]
12     push *rel[0]
13     gvar
14     push 2
15     lt
16     jmpfalse 10
17     push *rel[0]
18     gvar
19     jmp 21
20     push *rel[0]
21     gvar
22     push 2
23     sub
24     call 2
25     push *rel[0]
26     gvar
27     push 1
28     sub
29     call 2
30     add
31     setmp -1
32     ret
```

Demonstration: Laufzeitverhalten



- Ca. 2.7 mal schneller als der Tree-walking Interpreter
- Einfache Implementierung des Compilers
 - Stack-basierte Architektur
 - Gleichzeitige Entwicklung von VM und Compiler
 - Hoher Abstraktionsgrad

Kompilierung zu WebAssembly

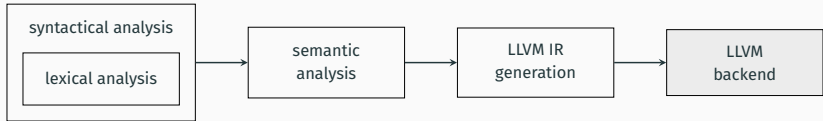
TODO: @RubixDev Write this

Kompilierung zu LLVM

- Startete als Forschungsprojekt von *Chris Lattner* [Lat02]
 - Auch *Rust* und *Swift* nutzen LLVM⁵.
 - Erzeugung von Code aus einer Zwischendarstellung
 - Aggressive Optimisierungsmaßnahmen
 - Die sogenannte *intermediate representation* (IR) kann mittels APIs erzeugt werden [Hsu21, preface]
- ⇒ LLVM ist das backend eines Compilers

⁵[McN21, p. 373], [Hsu21, preface]

Rolle von LLVM in einem Compiler



Etappen der Übersetzung mit Verwendung von LLVM

- Verwendung einer Rust library names **Inkwell**
- Erzeugung von LLVM IR

Ein LLVM Beispielprogramm: Eingabe

```
1  fn main() {  
2      let mut res = 2f;  
3      for i = 0; i < 10; i += 1 {  
4          res /= 2f;  
5      }  
6      exit((res * 100f) as int)  
7  }
```

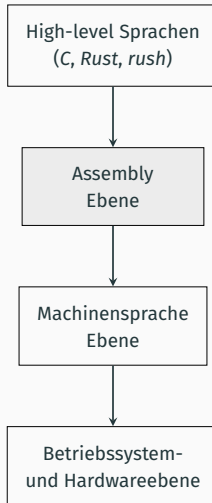

Ein LLVM Beispielprogramm: Ausgabe

```
1  ; ModuleID = 'main'
2  source_filename = "main"
3  target triple = "x86_64-alpine-linux-musl"
4
5  define i32 @main() {
6  entry:
7      br label %for_head
8
9  for_head:                                ; preds = %for_body,
    ↪ %entry
10     %res2 = phi double [ %f_prod, %for_body ], [ 2.000000e+00, %entry ]
11     %i3 = phi i64 [ %i_sum, %for_body ], [ 0, %entry ]
12     %i_lt = icmp slt i64 %i3, 10
13     br i1 %i_lt, label %for_body, label %after_for
14
15  for_body:                                ; preds = %for_head
16     %f_prod = fmul double %res2, 5.000000e-01
17     %i_sum = add i64 %i3, 1
18     br label %for_head
19
20  after_for:                                ; preds = %for_head
21     %f_prod5 = fmul double %res2, 1.000000e+02
22     %fi_cast = fptosi double %f_prod5 to i64
23     call void @exit(i64 %fi_cast)
24     unreachable
25 }
26
27 declare void @exit(i64)
```

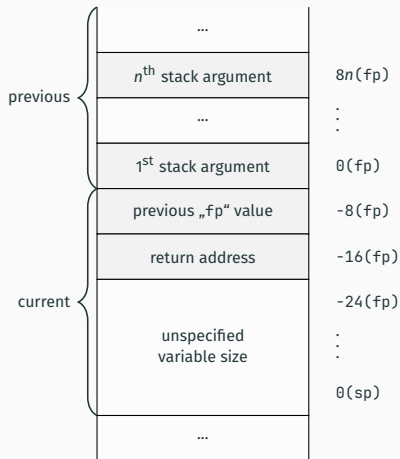
| Vorteile | Nachteile |
|--|--|
| hoher Abstraktionsgrad | Aufwendige Installation der LLVM Libraries |
| Unabhängigkeit von der Zielmaschine | signifikante Größe der ausführbaren Datei |
| aggressive Optimierungsmaßnahmen | unvollständige Dokumentation von Inkwell |
| Ausgabeprogramm ca. 1,7 mal schneller (vgl. x86_64 Compiler) | Abhängigkeit von einer C++ Codebase |

Kompilierung zu low-level Architekturen

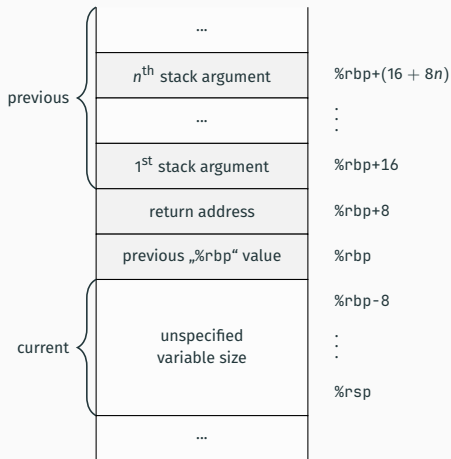
- Zielmaschine ist spezifisch
- Betriebssystem ist spezifisch
- Nutzung von Registern
- Hier: die Compiler generieren Assembly



Stacklayouts



RISC-V



x64

Kompilierung zu RISC-V

- **Reduced Instruction Set Computer (RISC)**
- Forschungsprojekt der UC Berkeley
- Ziel: Lösen der Probleme vieler CISC Architekturen
- Simplizität und Erweiterbarkeit
- Unterstützung durch: Google Microsoft, Samsung und IBM

Register der RISC-V Architektur

| Register | Verwendung |
|----------|-----------------------------------|
| zero | hardwired zero |
| ra | return address |
| sp | stack pointer |
| t0-t6 | temporary storage |
| fp | frame pointer |
| a0, a1 | function arguments, return values |
| a2-a7 | function arguments |
| s1-s11 | saved register |
| fa0, fa1 | float arguments, return values |
| fa2-fa7 | float arguments |
| fs0-fs11 | float saved registers |
| ft0-ft11 | float temporaries |

TODO: maybe fib?

```
1 fn main() {  
2     let a = 1;  
3     let b = 2;  
4     exit(a + b);  
5 }
```

Ausgabe



```
1 .global _start  
2  
3 .section .text  
4  
5 _start:  
6     call main..main  
7     li a0, 0  
8     call exit  
9  
10 main..main:  
11     # begin prologue  
12     addi sp, sp, -32  
13     sd fp, 24(sp)  
14     sd ra, 16(sp)  
15     addi fp, sp, 32  
16     # end prologue  
17     # begin body  
18     li a0, 1  
19     sd a0, -24(fp)    # let a = a0  
20     li a0, 2  
21     sd a0, -32(fp)    # let b = a0  
22     ld a0, -24(fp)    # a  
23     ld a1, -32(fp)    # b  
24     add a0, a0, a1  
25     call exit  
26     # end body  
27  
28 epilogue_0:  
29     ld fp, 24(sp)  
30     ld ra, 16(sp)  
31     addi sp, sp, 32  
32     ret
```

| Vorteile | Nachteile |
|---|--|
| Sehr neu und modern | Geringe Verbreitung |
| Komplett open-source und Gemeinschaftlich verwaltet | Eher experimentell |
| Sehr übersichtliche und simple Architektur | Einige Operationen sind aufwendiger |
| Weniger Online-Ressourcen | Sehr gute und übersichtliche Dokumentation |

Kompilierung zu x86_64

- **TODO: @RubixDev Write this**

| Vorteile | Nachteile |
|-------------------------------------|--|
| Höherer Abstraktionsgrad als RISC-V | Kompliziertere Übersetzung von z.B. Division |
| Weite Verbreitung | Sehr alt und unübersichtlich |
| Viele Online-Ressourcen | Weniger übersichtliche Dokumentation |

- Deutlich anspruchsvoller
- Signifikanter Lernaufwand
- Detailliertes Verständnis notwendig
- Sehr fehleranfällig
- Benötigt keine direkten Abhängigkeiten

Finale Anmerkungen & Fazit

TODO: @RubixDev @MikMuellerDev Write this

Literatur

- [Lat02] Chris Lattner. „LLVM: An Infrastructure for Multi-Stage Optimization“. Magisterarb. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dez. 2002.
- [Dan05] Sivarama P Dandamudi. *Guide to RISC processors*. Ottawa, Canada: Springer International Publishing, Feb. 2005. ISBN: 0-387-21017-2.
- [Wir05] Niklaus Wirth. *Compiler Construction*. Zürich, 2005. ISBN: 0-201-40353-6.
- [Led20] Jim Ledin. *Modern Computer Architecture and Organization*. Birmingham, UK: Packt Publishing, Apr. 2020. ISBN: 978-1-83898-439-7.
- [Hsu21] Min-Yih Hsu. *LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries*. Birmingham, UK: Packt Publishing, Apr. 2021. ISBN: 978-1-83882-495-2.
- [McN21] Timothy Samuel McNamara. *Rust in Action*. In Action. New York, NY: Manning Publications, Aug. 2021. ISBN: 978-1-61729-455-6.
- [Goo23] Google Fonts. *Google Fonts Icons*. Mai 2023. URL: <https://fonts.google.com/icons>.