



# Die Umwandlung von Quelltext in Maschinensprache

---

Silas Groh, Mik Müller

17. Mai 2023

Carl-Fuhlrott-Gymnasium

- Wir schreiben Programme in speziellen Sprachen
- Vorteile eines hohen Abstraktionsgrades



Erweiterbarkeit



Erweiterbarkeit



Plattformunabhängigkeit



Erweiterbarkeit



Plattformunabhängigkeit



Geschwindigkeit

# Zentrales Problem

```
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
  - Die Ausführung sollte jedoch auch **einfach** sein
- ⇒ Zentrales Problem unserer Arbeit

- Verschiedene Vorgehensweisen
- Unterscheidung zwischen **Interpretern** und **Compilern**

```
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.
  - Direktes Ausführen des Syntaxbaums
- ⇒ Kein zusätzlicher Prozess notwendig



```
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.
  - Umwandlung in ein anderes Format
  - Muss vor der Ausführung stattfinden
- ⇒ Zusätzlicher Prozess

- Entwicklung einer eigenen Programmiersprache
- Jeder hat jeweils **einen Interpreter** und **zwei Compiler** entwickelt

## **Die Programmiersprache „rush“**

---



- ca. vier Monate intensive Entwicklung
- 816 Git Commits
- 17548 Zeilen Programmtext in Git Commit „dbcbfa8“

Komponente	Zeilen Programmtext
Lexer / Parser	2737
Analyzer	2392
Tree-walking Interpreter	578
VM Compiler / Runtime	1288
WebAssembly Compiler	1641
LLVM Compiler	1450
C Transpiler	1185
RISC-V Compiler	2234
x64 Compiler	2773

Komponente	Zeilen Programmtext
Lexer / Parser	2737
Analyzer	2392
Tree-walking Interpreter	578
VM Compiler / Runtime	1288
WebAssembly Compiler	1641
LLVM Compiler	1450
C Transpiler	1185
RISC-V Compiler	2234
x64 Compiler	2773

Bezeichnung	Beispiel
Schleife	<code>loop { }</code>
„while“-Schleife	<code>while x &lt; 5 { }</code>
„for“-Schleife	<code>for i = 0; i &lt; 5; i += 1 { }</code>

Bezeichnung	Beispiel
Schleife	<code>loop { }</code>
„while“-Schleife	<code>while x &lt; 5 { }</code>
„for“-Schleife	<code>for i = 0; i &lt; 5; i += 1 { }</code>
„if“-Verzweigung	<code>if true { } else { }</code>



Bezeichnung	Beispiel
Schleife	<code>loop { }</code>
„while“-Schleife	<code>while x &lt; 5 { }</code>
„for“-Schleife	<code>for i = 0; i &lt; 5; i += 1 { }</code>
„if“-Verzweigung	<code>if true { } else { }</code>
Funktionsdefinition	<code>fn foo(n: int) { }</code>

Bezeichnung	Beispiel
Schleife	<code>loop { }</code>
„while“-Schleife	<code>while x &lt; 5 { }</code>
„for“-Schleife	<code>for i = 0; i &lt; 5; i += 1 { }</code>
„if“-Verzweigung	<code>if true { } else { }</code>
Funktionsdefinition	<code>fn foo(n: int) { }</code>
Variablendefinition	<code>let mut answer = 42</code>

Bezeichnung	Beispiel
Schleife	<code>loop { }</code>
„while“-Schleife	<code>while x &lt; 5 { }</code>
„for“-Schleife	<code>for i = 0; i &lt; 5; i += 1 { }</code>
„if“-Verzweigung	<code>if true { } else { }</code>
Funktionsdefinition	<code>fn foo(n: int) { }</code>
Variablendefinition	<code>let mut answer = 42</code>
Infix-Ausdruck	<code>1 + n; 5 ** 2</code>
Präfix-Ausdruck	<code>!false; -n</code>

# Fähigkeiten von rush

Bezeichnung	Beispiel
Schleife	<code>loop { }</code>
„while“-Schleife	<code>while x &lt; 5 { }</code>
„for“-Schleife	<code>for i = 0; i &lt; 5; i += 1 { }</code>
„if“-Verzweigung	<code>if true { } else { }</code>
Funktionsdefinition	<code>fn foo(n: int) { }</code>
Variablendefinition	<code>let mut answer = 42</code>
Infix-Ausdruck	<code>1 + n; 5 ** 2</code>
Präfix-Ausdruck	<code>!false; -n</code>
Pointer	<code>let b = &amp;a; *b</code>

Bezeichnung	Beispiel
Schleife	<code>loop { }</code>
„while“-Schleife	<code>while x &lt; 5 { }</code>
„for“-Schleife	<code>for i = 0; i &lt; 5; i += 1 { }</code>
„if“-Verzweigung	<code>if true { } else { }</code>
Funktionsdefinition	<code>fn foo(n: int) { }</code>
Variablendefinition	<code>let mut answer = 42</code>
Infix-Ausdruck	<code>1 + n; 5 ** 2</code>
Präfix-Ausdruck	<code>!false; -n</code>
Pointer	<code>let b = &amp;a; *b</code>
Typumwandlung	<code>42 as float</code>

Bezeichnung	Instanziierung einer Variable
„int“	<code>let a: int = 0;</code>
„float“	<code>let b: float = 3.14;</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>

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>



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>
„()“ oder „Unit“	<code>let e: () = main();</code>
„!“ oder „Never“	<code>let f = exit(42);</code>

## Berechnung von Fibonaccizahlen in rush

```
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 }
```

## **Lexikalische und syntaktische Analyse**

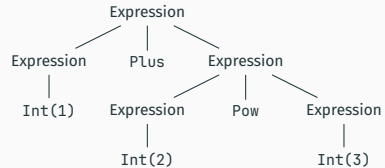
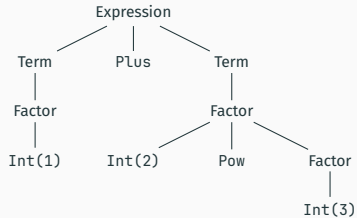
---



- 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' | '3' | '4' | '5' | '6' | '7' | '8' | '9' }- ;
```

# Abstrakter Syntaxbaum



Zwei verschiedene Syntaxbäume für „1+2\*\*3“

## **Semantische Analyse**

---





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

- Unterscheidung zwischen validen und invaliden Programmen

- Unterscheidung zwischen validen und invaliden Programmen
- hilfreiche Warnungen und Informationen

- Unterscheidung zwischen validen und invaliden Programmen
- hilfreiche Warnungen und Informationen
- Hinzufügen von Typinformationen zu dem Syntaxbaum

- Unterscheidung zwischen validen und invaliden Programmen
- hilfreiche Warnungen und Informationen
- Hinzufügen von Typinformationen zu dem Syntaxbaum
- Triviale Optimierungen der Programmstruktur

## Beispiel 1: Typkonflikt

```
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: Warnung aufgrund einer unbenutzten Variable

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



Ausgabe

**Warning** at unused\_var.rush:2:9

```
1 | fn main() {  
2 |     let x = 42;  
   |         ^  
3 |     let mut y = 3;
```

**unused variable** `x`

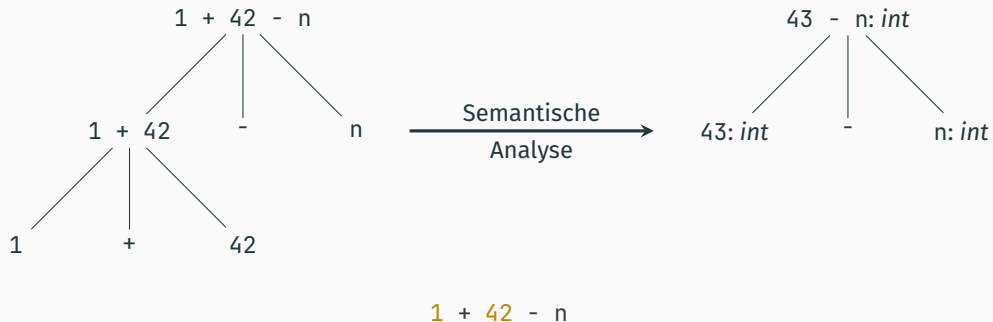
**note:** if this is intentional, change the name to `\_x` to hide this warning

**Info** at unused\_var.rush:3:13

```
2 |     let x = 42;  
3 |     let mut y = 3;  
   |               ^  
4 |     exit(y);
```

**variable** `y` does not need to be mutable

## Hinzufügen von Informationen über Datentypen





## Tree-walking Interpreter

---

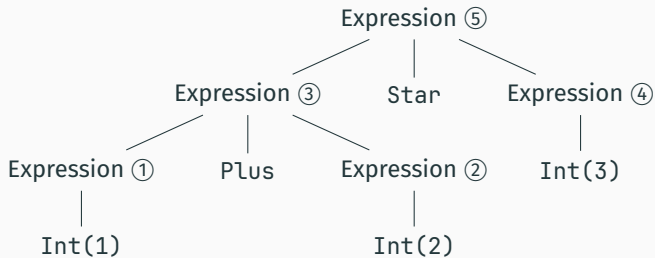
- *Traversiert* den Syntaxbaum
- *Interpretiert* das Programm direkt

```
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 dem entsprechenden Knoten im Syntaxbaum

```
6  pub enum Value {  
7      Int(i64),  
8      Float(f64),  
9      Char(u8),  
10     Bool(bool),  
11     Unit,  
12     Ptr(Rc<RefCell<Value>>),  
13 }
```



Syntaxbaum zu „1 + 2 \* 3“

```
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 }
```

...
call_func("fib", vec![10])
visit_block(/* ... */)
visit_expression(/* ... */)
visit_if_expr(/* ... */)
visit_block(/* ... */)
visit_expression(/* ... */)
visit_infix_expr(/* ... */)
visit_expression(/* ... */)
visit_call_expr(/* ... */)
call_func("fib", vec![8])
...
call_func("fib", vec![6])
...
call_func("fib", vec![4])
...
call_func("fib", vec![2])
...
call_func("fib", vec![0])



## **Virtuelle Maschine**

---

## Etappen der Übersetzung: Code-Erzeugung





- Häufig: Eine *Virtuelle Maschine* (VM) simuliert echte Computer
  - Display
  - Lautsprecher
  - Festplatte
  - ...

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

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

Compiler 



```
4 1: (main)  
5     setmp 0  
6     push 2  
7     call 2  
8 2: (foo)  
9     setmp 2  
10    svari *rel[0]  
11    push 3  
12    svari *rel[-1]  
13    push *rel[0]  
14    gvar  
15    push *rel[-1]  
16    gvar  
17    add  
18    exit  
19    setmp -2  
20    ret
```

VM 



Exit code: 5

- Java, usw.
- Umwandlung in ein anderes Format

⇒ Anschliessende Ausführung des Programmes

- Führt ein zuvor übersetztes Programm aus

- Führt ein zuvor übersetztes Programm aus
- Besitzt eine selbst entwickelte Architektur
  - Stackbasiertes Design
  - Hoher Abstaktionsgrad

**stack** Für temporäre Werte

**stack** Für temporäre Werte

**mem** Für Variablen

**mem\_ptr** Für Speicherverwaltung

**stack** Für temporäre Werte

**mem** Für Variablen

**mem\_ptr** Für Speicherverwaltung

**call\_stack** Aufrufstapel (*Befehlsähler* und *Funktionszähler*)



- Unterteilung in Funktionen
  - Ohne Namen
  - numerische Identifizierung
  - Enthält mehrere Anweisungen

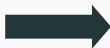
- Unterteilung in Funktionen
  - Ohne Namen
  - numerische Identifizierung
  - Enthält mehrere Anweisungen
- Struktur der Anweisungen: „call 2“
  - Befehlscode (call)
  - Optionaler Operand (2)

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

## Demonstration: Ein-/Ausgabe

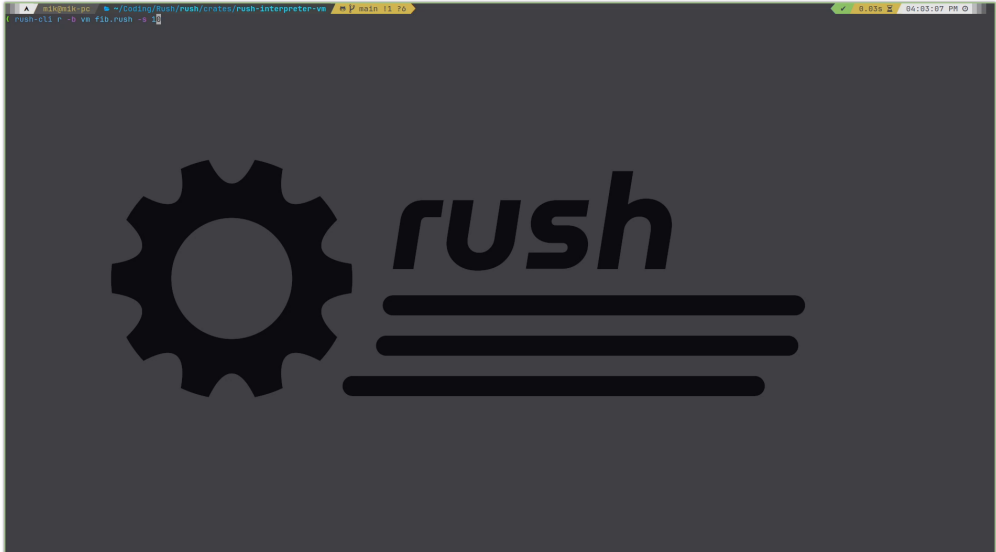
```
1  fn main() {  
2      exit(fib(10));  
3  }  
4  
5  fn fib(n: int) -> int {  
6      if n < 2 {  
7          n  
8      } else {  
9          fib(n - 2) + fib(n - 1)  
10     }  
11 }
```

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     # ...  
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

- Ca. 2,7 mal schneller als der Tree-walking Interpreter
- Einfache Implementierung des Compilers
  - Stack-basierte Architektur
  - Hoher Abstraktionsgrad
  - Gleichzeitige Entwicklung von VM und Compiler (**Feedbackschleife**)

## Kompilierung zu WebAssembly

---



# Was ist WebAssembly?

- Sicheres, portables, kompaktes und effizientes Format
- Hauptsächlich für leistungsstarke Webanwendungen

- Sicheres, portables, kompaktes und effizientes Format
- Hauptsächlich für leistungsstarke Webanwendungen
- Alleinstehende Spezifikation

- Sicheres, portables, kompaktes und effizientes Format
- Hauptsächlich für leistungsstarke Webanwendungen
- Alleinstehende Spezifikation
- Implementation durch Browser oder separate Runtimes

## Beispiel 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

```
00000000: 0061 736d 0100 0000 010d 0360 017f 0060 .asm.....`...`
00000010: 0000 6001 7e01 7e02 2401 1677 6173 695f ..`.~.~.$..wasi_
00000020: 736e 6170 7368 6f74 5f70 7265 7669 6577 snapshot_preview
00000030: 3109 7072 6f63 5f65 7869 7400 0003 0302 1.proc_exit.....
00000040: 0102 0503 0100 0007 1302 065f 7374 6172 ....._star
00000050: 7400 0106 6d65 6d6f 7279 0200 0801 010a t...memory.....
00000060: 2902 0a00 420a 1002 a710 0000 0b1c 0020 )...B.....
00000070: 0042 0253 047e 2000 0520 0042 027d 1002 .B.S.~ .. .B.}..
00000080: 2000 4201 7d10 027c 0b0b 002a 046e 616d .B.}..|...*.nam
00000090: 6501 1903 000b 5f5f 7761 7369 5f65 7869 e.....__wasi_exi
000000a0: 7401 046d 6169 6e02 0366 6962 0208 0201 t..main..fib....
000000b0: 0002 0100 016e .....n
```

```
1  (module
2    ;; ...
3    (import "wasi_snapshot_preview1" "proc_exit" (func $__wasi_exit (type 0)))
4    (func $main (type 1)
5      i64.const 10
6      call $fib
7      i32.wrap_i64
8      call $__wasi_exit
9      unreachable)
10   (func $fib (type 2) (param $n i64) (result i64)
11     local.get $n
12     i64.const 2
13     i64.lt_s
14     if (result i64) ;; label = @1
15       local.get $n
16       else
17         local.get $n
18         i64.const 2
19         i64.sub
20         call $fib
21         local.get $n
22         i64.const 1
23         i64.sub
24         call $fib
25         i64.add
26       end)
27   (memory (;0;) 0)
28   (export "_start" (func $main))
29   (export "memory" (memory 0))
30   (start $main))
```

# Hoher Abstraktionsgrad

rush	WebAssembly
<pre>fn fib(n: int) -&gt; int {     // ... }</pre>	<pre>(func \$fib   (param \$n i64)   (result i64)   ;; ... )</pre>
<pre>if /* ... */ {     n } else {     // ... }</pre>	<pre>;; ... if (result i64)   local.get \$n else   ;; ... end</pre>
<pre>loop {     // ... }</pre>	<pre>loop   ;; ...   br 0 end</pre>

## Kompilierung zu LLVM

---

- Startete als **Forschungsprojekt**

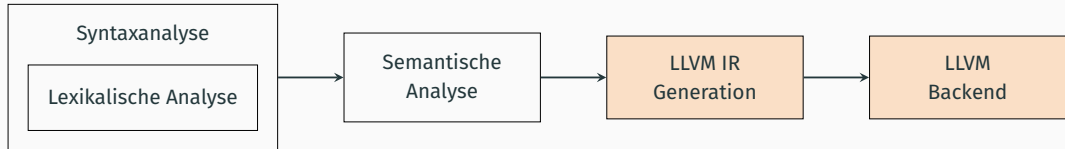


- Startete als **Forschungsprojekt**
- Erzeugung von Code aus einer **Zwischendarstellung** (IR)
- Die IR kann mittels APIs erzeugt werden

- Startete als **Forschungsprojekt**
- Erzeugung von Code aus einer **Zwischendarstellung** (IR)
- Die IR kann mittels APIs erzeugt werden
- Aggressive **Optimierung**

- Startete als **Forschungsprojekt**
- Erzeugung von Code aus einer **Zwischendarstellung** (IR)
- Die IR kann mittels APIs erzeugt werden
- Aggressive **Optimierung**
- Auch Rust und Swift nutzen LLVM

- Startete als **Forschungsprojekt**
  - Erzeugung von Code aus einer **Zwischendarstellung** (IR)
  - Die IR kann mittels APIs erzeugt werden
  - Aggressive **Optimierung**
  - Auch Rust und Swift nutzen LLVM
- ⇒ Das Backend eines Compilers



- Rust library names **Inkwell**
- Der Compiler erzeugt LLVM IR

## Beispiel 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  ; ModuleID = 'main'
2  source_filename = "main"
3  target triple = "x86_64-pc-linux-gnu"
4
5  define internal i64 @fib(i64 %0) {
6  entry:
7      %i_lt = icmp slt i64 %0, 2
8      br i1 %i_lt, label %merge, label %else
9
10 merge:                                     ; preds
11     ↪ = %entry, %else
12     %if_res = phi i64 [ %i_sum3, %else ], [ %0, %entry ]
13     ret i64 %if_res
14
15 else:                                     ; preds
16     ↪ = %entry
17     %i_sum = add i64 %0, -2
18     %ret_fib = call i64 @fib(i64 %i_sum)
19     %i_sum1 = add i64 %0, -1
20     %ret_fib2 = call i64 @fib(i64 %i_sum1)
21     %i_sum3 = add i64 %ret_fib2, %ret_fib
22     br label %merge
23 }
24
25 define i32 @main() {
26 entry:
27     %ret_fib = call i64 @fib(i64 10)
28     call void @exit(i64 %ret_fib)
29     unreachable
30 }
31
32 declare void @exit(i64)
```

- Hoher Abstraktionsgrad



- Hoher Abstraktionsgrad
- Unabhängigkeit von der Zielmaschine

- Hoher Abstraktionsgrad
- Unabhängigkeit von der Zielmaschine
- Aggressive Optimierungsmaßnahmen

- Hoher Abstraktionsgrad
- Unabhängigkeit von der Zielmaschine
- Aggressive Optimierungsmaßnahmen
- Ausgabeprogramm ca. 1,7 mal schneller (vgl. x64 Compiler)

- Aufwendige Installation der LLVM libraries

- Aufwendige Installation der LLVM libraries
- Signifikante Größe des Compilers

- Aufwendige Installation der LLVM libraries
- Signifikante Größe des Compilers
- Unvollständige Dokumentation von Inkwell

- Aufwendige Installation der LLVM libraries
- Signifikante Größe des Compilers
- Unvollständige Dokumentation von Inkwell
- Abhängigkeit von einer C++ Codebase

## **Kompilierung zu low-level Architekturen**

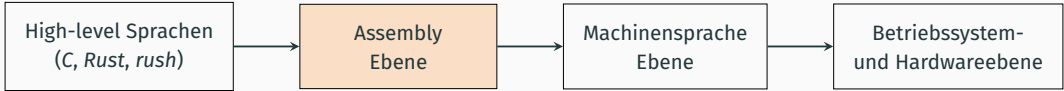
---



- Zielmaschine ist spezifisch
- Betriebssystem ist spezifisch

- Zielmaschine ist spezifisch
- Betriebssystem ist spezifisch
- Hier: die Compiler generieren Assembly

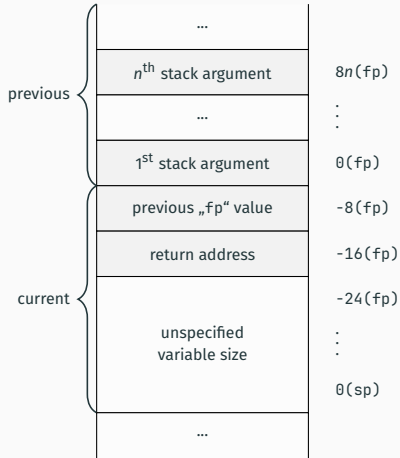
# Abstraktionsgrad von Assembly



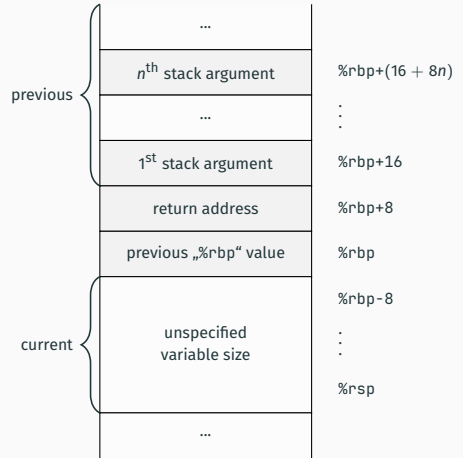
# Register

Register	Purpose
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

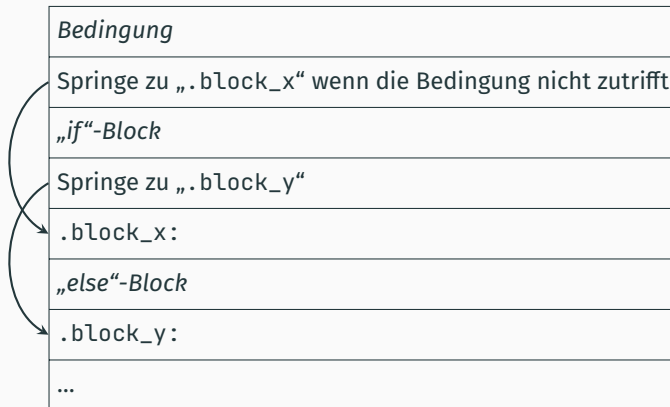
# Stacklayouts



RISC-V



x64



## Kompilierung zu RISC-V

---

- **Reduced Instruction Set Computer (RISC)**



- **Reduced Instruction Set Computer (RISC)**
- **Forschungsprojekt** der UC Berkeley

- **Reduced Instruction Set Computer (RISC)**
- **Forschungsprojekt** der UC Berkeley
- Lösen der Probleme vieler anderer Architekturen

- **R**educed **I**nstruction **S**et **C**omputer (**RISC**)
- **Forschungsprojekt** der UC Berkeley
- Lösen der Probleme vieler anderer Architekturen
- **Einfachheit** und Erweiterbarkeit

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

# Beispiel

```
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  .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, -16  
13     sd fp, 8(sp)  
14     # ...  
28  
29 main..fib:  
30     # begin prologue  
31     addi sp, sp, -32  
32     sd fp, 24(sp)  
33     sd ra, 16(sp)  
34     addi fp, sp, 32  
35     # end prologue  
36     # save params on stack  
37     sd a0, -24(fp) # param n = a0  
38     # begin body  
39     ld a0, -24(fp) # n  
40     li a1, 2
```

- Sehr neu und modern

- Sehr neu und modern
- Komplette open-source und gemeinschaftlich verwaltet

- Sehr neu und modern
- Komplette open-source und gemeinschaftlich verwaltet
- Sehr übersichtliche und simple Architektur



- Sehr neu und modern
- Komplette open-source und gemeinschaftlich verwaltet
- Sehr übersichtliche und simple Architektur
- Sehr gute und übersichtliche Dokumentation

- Geringe Verbreitung

- Geringe Verbreitung
- Weniger Online-Ressourcen

- Geringe Verbreitung
- Weniger Online-Ressourcen
- Einige Operationen sind aufwendiger

- Geringe Verbreitung
- Weniger Online-Ressourcen
- Einige Operationen sind aufwendiger
- Abhängigkeit von einem Emulator (QEMU)

## Kompilierung zu x86\_64

---

- Häufig auch x84-64 oder x64 geschrieben

- Häufig auch x84-64 oder x64 geschrieben
- **C**omplex **I**nstruction **S**et **C**omputer (CISC)
  - Um vielfaches mehr Instruktionen als RISC Architekturen



- Häufig auch x84-64 oder x64 geschrieben
- **C**omplex **I**nstruction **S**et **C**omputer (CISC)
  - Um vielfaches mehr Instruktionen als RISC Architekturen
- Sehr weit verbreitet

## Beispiel 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



```
6  _start:  
7      call    main..main  
8      mov     %rdi, 0  
9      call    exit  
10  
11  main..main:  
12      mov     %rdi, 10  
13      call    main..fib  
14      mov     %rdi, %rax  
15      call    exit  
16  main..main.return:  
17      ret  
18  
19  main..fib:  
20      push    %rbp  
21      mov     %rbp, %rsp  
22      sub     %rsp, 16  
23      mov     qword ptr [%rbp-8], %rdi  
24      cmp     qword ptr [%rbp-8], 2  
25      jge     .block_0  
26      mov     %rax, qword ptr [%rbp-8]  
27      jmp     .block_1  
28  .block_0:  
29  # ...  
39  .block_1:  
40  main..fib.return:  
41      leave  
42      ret
```

Merkmal	RISC-V	x64
Instruktionen	<code>addi a0, a0, 3</code>	<code>add %rax, 3</code>

Merkmal	RISC-V	x64
Instruktionen	<code>addi a0, a0, 3</code>	<code>add %rax, 3</code>
Register	<code>a0, a1, fa0, sp, ...</code>	<code>%rax, %rdi, %xmm0, %rsp, ...</code>

Merkmal	RISC-V	x64
Instruktionen	<code>addi a0, a0, 3</code>	<code>add %rax, 3</code>
Register	<code>a0, a1, fa0, sp, ...</code>	<code>%rax, %rdi, %xmm0, %rsp, ...</code>
Pointer	<code>-1(fp)</code>	<code>byte ptr [%rbp-1]</code>

Merkmal	RISC-V	x64
Instruktionen	<code>addi a0, a0, 3</code>	<code>add %rax, 3</code>
Register	<code>a0, a1, fa0, sp, ...</code>	<code>%rax, %rdi, %xmm0, %rsp, ...</code>
Pointer	<code>-1(fp)</code>	<code>byte ptr [%rbp-1]</code>
Größe eines Worts	4 Byte	2 Byte

- Höherer Abstraktionsgrad als RISC-V
- Weite Verbreitung
- Viele Online-Ressourcen

- Kompliziertere Übersetzung von z.B. Division
- Sehr alt und unübersichtlich
- Weniger übersichtliche Dokumentation



- Deutlich anspruchsvoller

- Deutlich anspruchsvoller
- Signifikanter Lernaufwand

- Deutlich anspruchsvoller
- Signifikanter Lernaufwand
- Detailliertes Verständnis notwendig

- Deutlich anspruchsvoller
- Signifikanter Lernaufwand
- Detailliertes Verständnis notwendig
- Sehr fehleranfällig

## **Finale Anmerkungen & Fazit**

---

- Vertiefung
  - Lexer und Parser
  - Tree-walking Interpreter

- Vertiefung
  - Lexer und Parser
  - Tree-walking Interpreter
- Pratt Parsing

- Vertiefung
  - Lexer und Parser
  - Tree-walking Interpreter
- Pratt Parsing
- LLVM und WebAssembly



- Vertiefung
  - Lexer und Parser
  - Tree-walking Interpreter
- Pratt Parsing
- LLVM und WebAssembly
- Assembly und low-level Programmierung

**rush Website** <https://rush-lang.de>

**rush Website** <https://rush-lang.de>

**Paper** <https://paper.rush-lang.de>

**rush Website** <https://rush-lang.de>

**Paper** <https://paper.rush-lang.de>

**Playground** <https://play.rush-lang.de>

**rush Website** <https://rush-lang.de>

**Paper** <https://paper.rush-lang.de>

**Playground** <https://play.rush-lang.de>

**GitHub** <https://github.com/rush-rs>