Die Umwandlung von Quelltext in Maschinensprache

Silas Groh, Mik Müller

17. Mai 2023

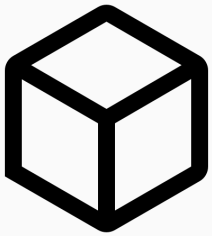
Carl-Fuhlrott-Gymnasium

Einstieg & Motivation

Einstieg & Motivation

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

Erweiterbarkeit & Reperatur



Portabilität & Platformunabhängigkeit

Geschwindigkeit & Einfachheit

[Goo23]

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  }
```

- Programme sollten einfach zu schreiben sein

$\Rightarrow$ Ein Computer muss diese jedoch auch einfach verarbeiten

Methoden zur Programmausführung

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

Interpreter

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

Ausführung 

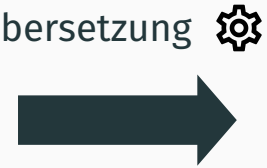
Exit code: 5

- Python, Javascript, PHP, usw.
- Keine Übersetzung notwending

$\Rightarrow$ Interpretiert den Syntaxbaum direkt

Compiler

```rust
fn main() {
    foo(2);
}

fn foo(n: int) {
    let mut m = 3;
    exit(n + m);
}
```

Übersetzung 

```
; RISC-V binary
457f 464c 0102
0002 00f3 0001
0040 0000 0000
0005 0000 0040
0003 7000 0004
0000 0000 0000
0048 0000 0000
0001 0000 0000
0000 0000 0000
```

- Rust, C, Go, usw.
- Zusätzlicher Prozess
- Umwandlung in ein anderes Format

$\Rightarrow$ Muss vor der Ausführung stattfinden

Die Programmiersprache „rush“

Fakten über rush

- ca. vier Monate intensive Entwicklung
- 814 Git Commits
- 17526 Zeilen Programmtext[1] in Git Commit ‘9953dd8’

[1] Leerzeilen und Kommentare werden nicht gezählt.

Inhalte des Projektes

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

Fähigkeiten von rush TODO: MAYBE SPLIT FEATURES ACROSS SLIDES

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

Datentypen in rush TODO: MAYBE SPLIT TYPES ACROSS SLIDES

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

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 }
```

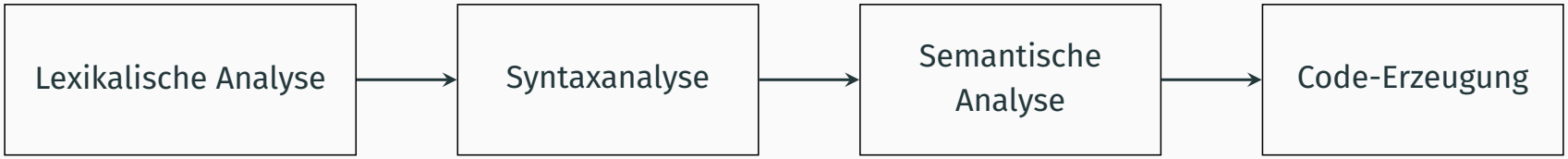
$$f_n = f_{n-2} + f_{n-1} \quad ; n \geq 3$$

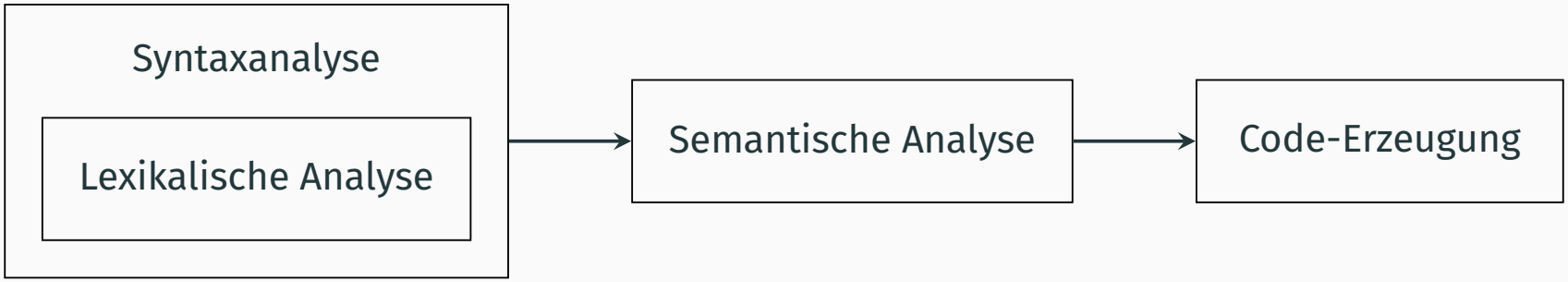
$$f_1 = f_2 = 1$$

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

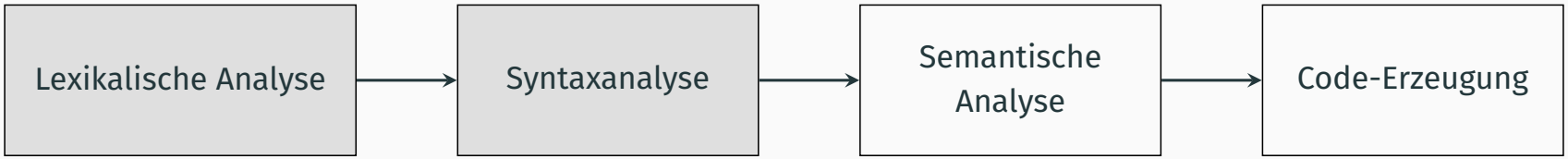
Stufen der Übersetzung [Wir05, S. 6–7] TODO: MAYBE DELETE





Lexikalische & Syntaktische Analyse

Etappen der Übersetzung: Semantische Analyse



[Wir05, S. 6–7]

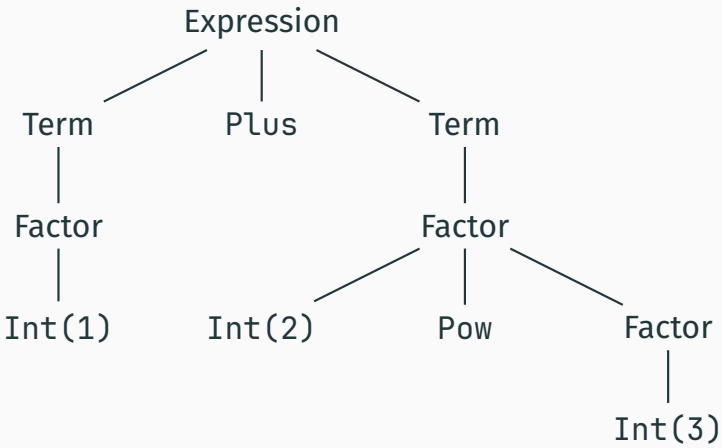
Lexikalische & Syntaktische Analyse TODO: DELETE

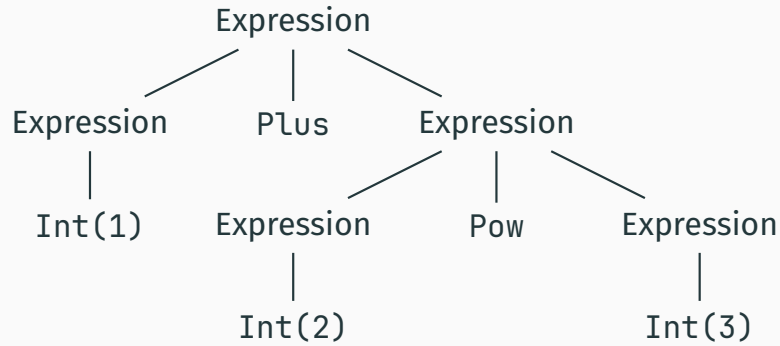
- Gruppieren des Programmtextes in Tokens
- Analyse der Syntax des Programms
- Erzeugung eines abstrakten Syntaxbaums
- Festlegen der formalen Regeln in Form einer Grammatik

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

Ein Beispiel für eine kontextfreihe Grammatik (EBNF)

Abstrakter Syntaxbaum TODO: DELETE

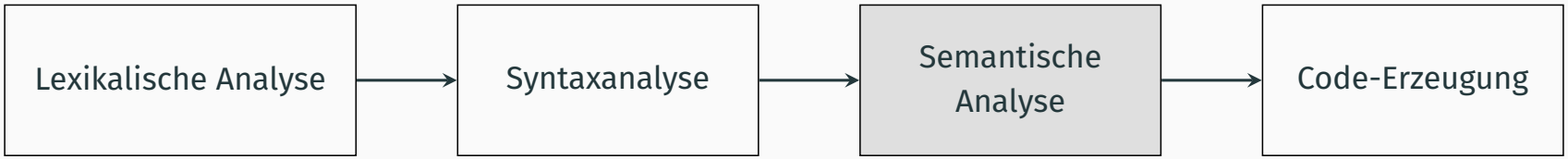




Zwei verschiedene Syntaxbäume für „1+2**3“

Semantische Analyse

Etappen der Übersetzung: Semantische Analyse



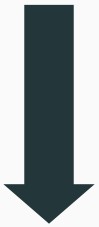
[Wir05, S. 6–7]

Semantische Analyse & Semantikregeln

- 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  }
```

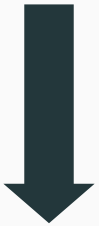


Fehlerausgabe

```
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  }
```

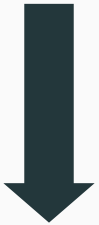


Fehlerausgabe

```
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

```rust
fn main() {
    let variable = 42;
    let mut code = 3;
    exit(code);
}
```



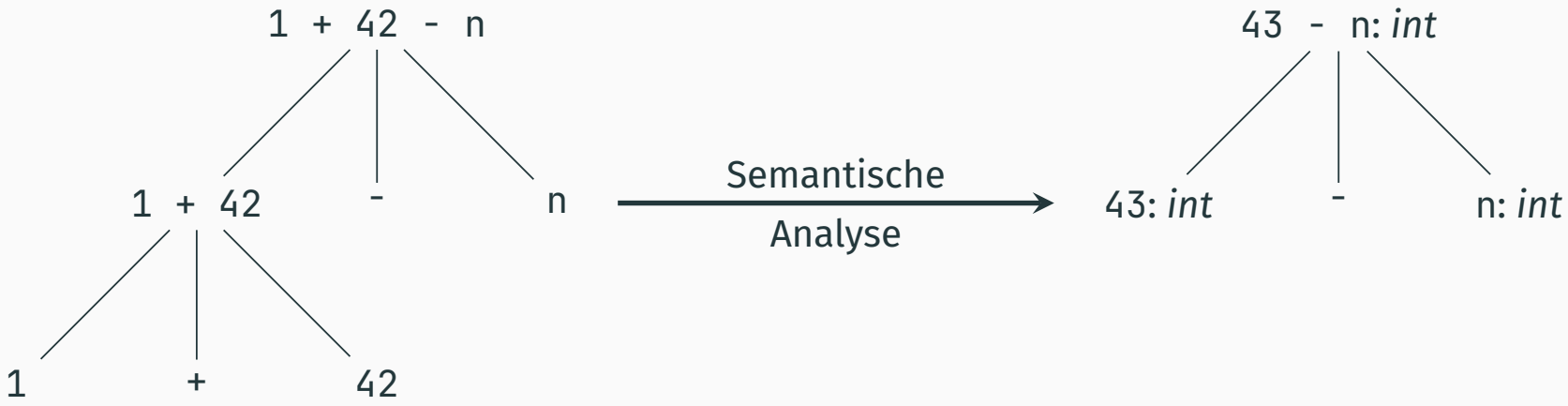
Ausgabe

```
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

Tree-Walking Interpreter

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

Felder des Interpreters

crates/rush-interpreter-tree/src/interpreter.rs

```rust
type Scope<'src> = HashMap<&'src str, Rc<RefCell<Value>>>;

#[derive(Debug, Default)]
pub struct Interpreter<'src> {
    scopes: Vec<Scope<'src>>,
    functions: HashMap<&'src str, Rc<AnalyzedFunctionDefinition<'src>>>,
}
```

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

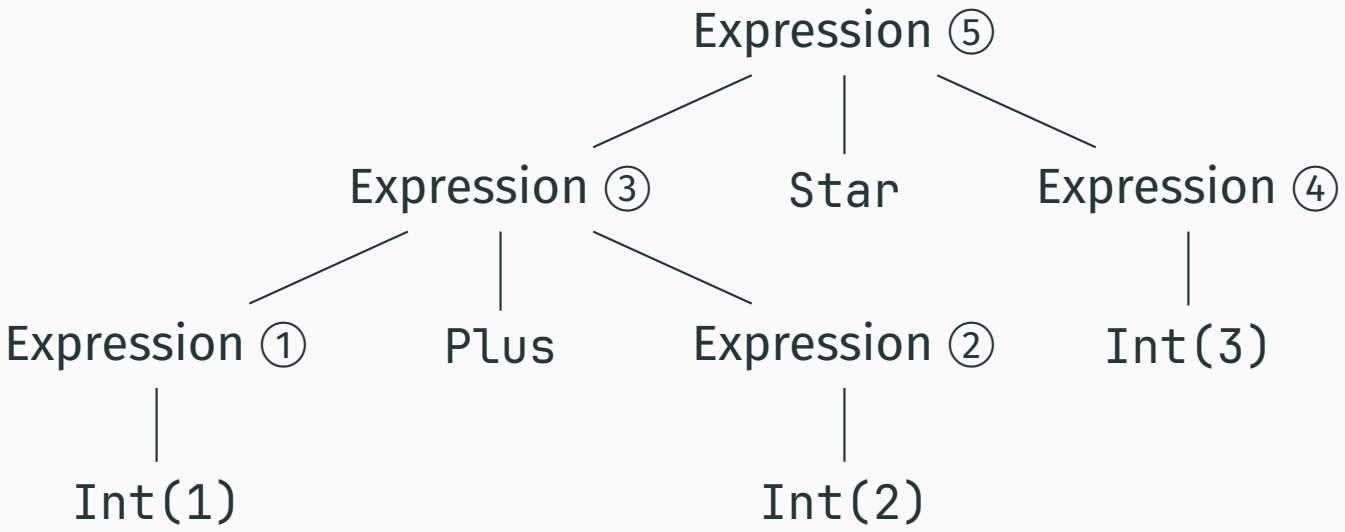
Weitere Typdefinitionen

crates/rush-interpreter-tree/src/value.rs

```rust
pub enum Value {
    Int(i64),
    Float(f64),
    Char(u8),
    Bool(bool),
    Unit,
    Ptr(Rc<RefCell<Value>>),
}
// ...
pub enum InterruptKind {
    Return(Value),
    Break,
    Continue,
    Error(interpreter::Error),
    Exit(i64),
}
```

- Aufzählung zum Speichern verschiedener Datentypen
- Verschiedene Unterbrechungen des Programmflusses

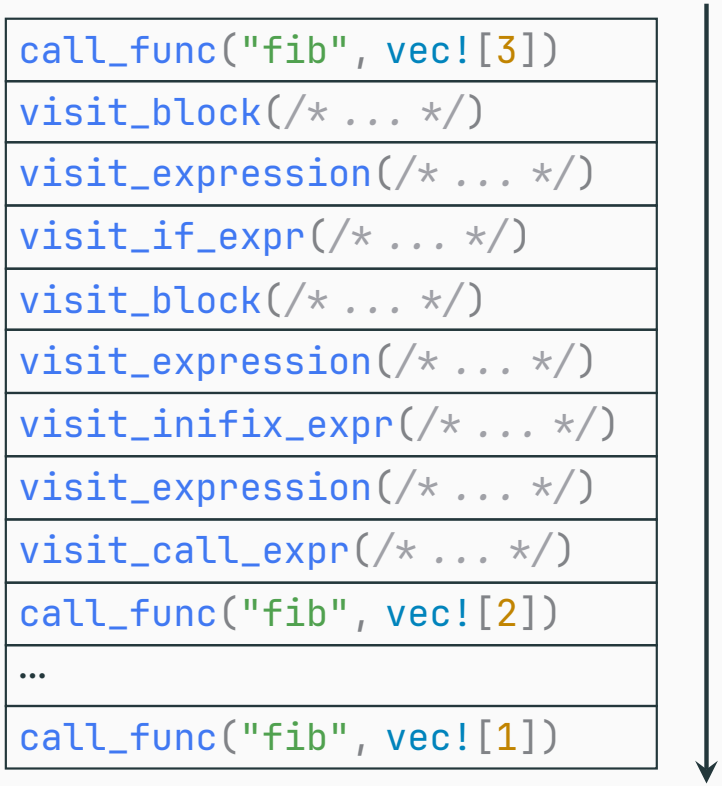
Traversierung



Syntaxbaum zu „`1 + 2 * 3`“

Effizienz

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



Virtuelle Maschine

Virtuelle Maschine

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

Wie eine CPU Programme ausführt TODO: DELETE

- 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.

[2] Engl: „arithmetic logic unit“ (ALU).
[3] Engl: „control unit“.
[4] Engl: „fetch-decode-execute cycle“.

Übertragung der Konzepte auf die rush VM TODO: DELETE

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

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

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 *Befehlsähler* und den *Funktionszähler* für jeden Aufruf speichert

Struktur der Programme der rush VM

- 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)

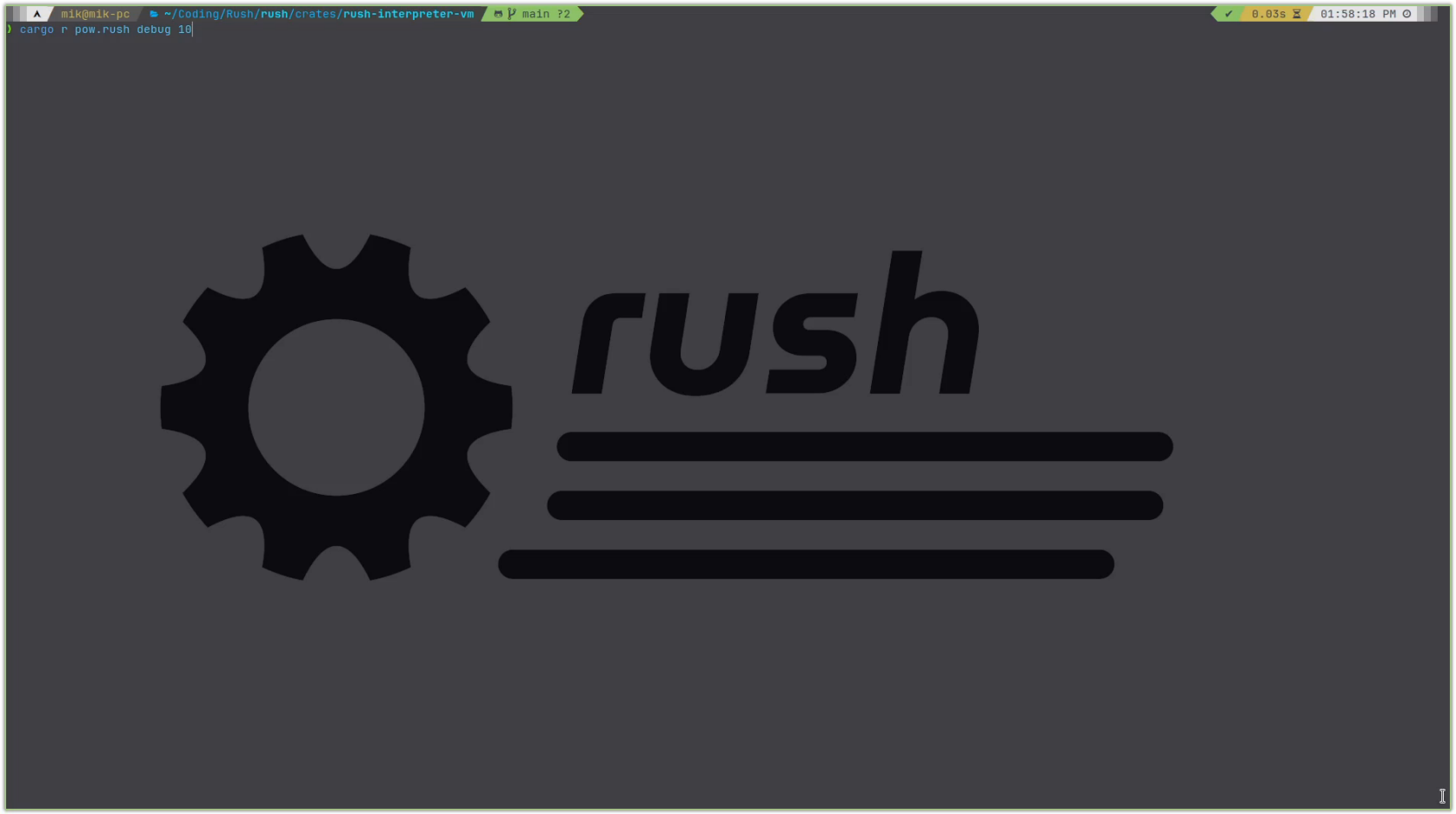
Demonstration: Ein-/Ausgabe

```
fn main() {
    exit(fib(10));
}

fn fib(n: int) -> int {
    if n < 2 {
        n
    } else {
        fib(n - 2) + fib(n - 1)
    }
}
```

Ausgabe

```
0: (prelude)
    setmp 0
    call 1
1: (main)
    setmp 0
    push 10
    call 2
    exit
2: (fib)
    setmp 1
    svari *rel[0]
    push *rel[0]
    gvar
    push 2
    lt
    jmpfalse 10
    push *rel[0]
    gvar
    jmp 21
    push *rel[0]
    gvar
    push 2
    sub
    call 2
    push *rel[0]
    gvar
    push 1
    sub
    call 2
    add
    setmp -1
    ret
```

Demonstration: Laufzeitverhalten



VM: Fazit

- 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

Kompilierung zu WebAssembly

TODO: @RubixDev Write this

Kompilierung zu LLVM

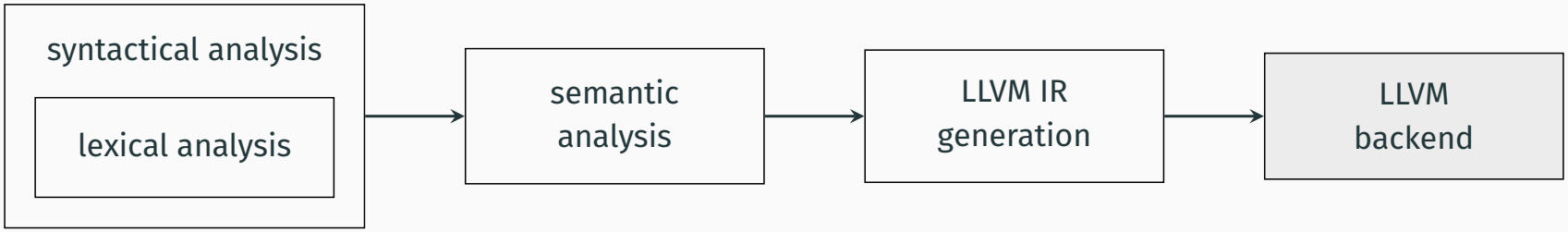
Was ist LLVM?

- Startete als Forschungsprojekt von *Chris Lattner* [Lat02]
- Auch *Rust* und *Switft* nutzen LLVM[5].
- Erzeugung von Code aus einer Zwischendarstellung
- Aggressive Optimisierungsmaßnamen
- Die sogenannte *intermediate representation* (IR) kann mittels APIs erzeugt werden [Hsu21, preface]

$\Rightarrow$ 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

Der rush LLVM Compiler

- 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)
```

Fazit

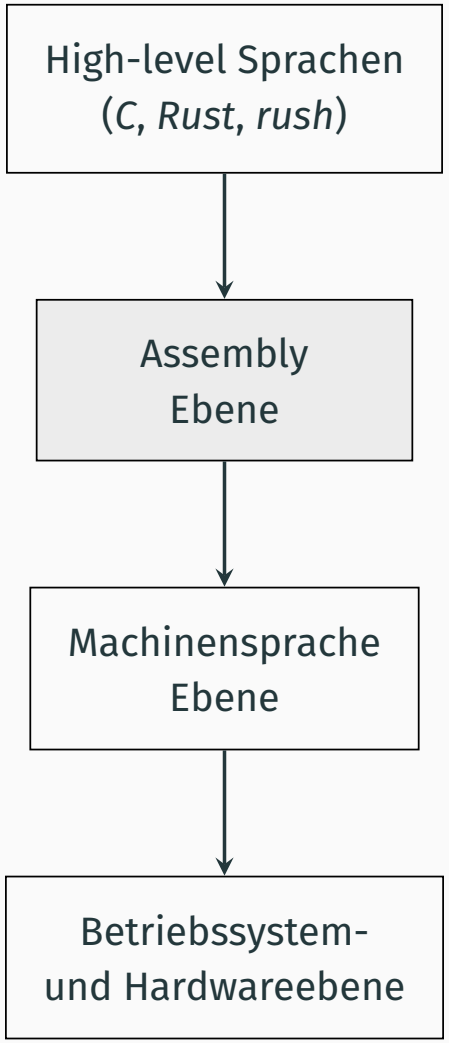
Vorteile	Nachteile
hoher Abstraktionsgrad	Aufwendige Installation der LLVM Libraries
Unabhängigkeit von der Zielmaschine	signifikante Größe der ausführbaren Datei
aggressive Optimisierungsmaßnahmen	unvollständige Dokumentation von Inkwell
Ausgabeprogramm ca. 1,7 mal schneller (vgl. x86_64 Compiler)	Abhängigkeit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

Kompilierung zu low-level Architekturen

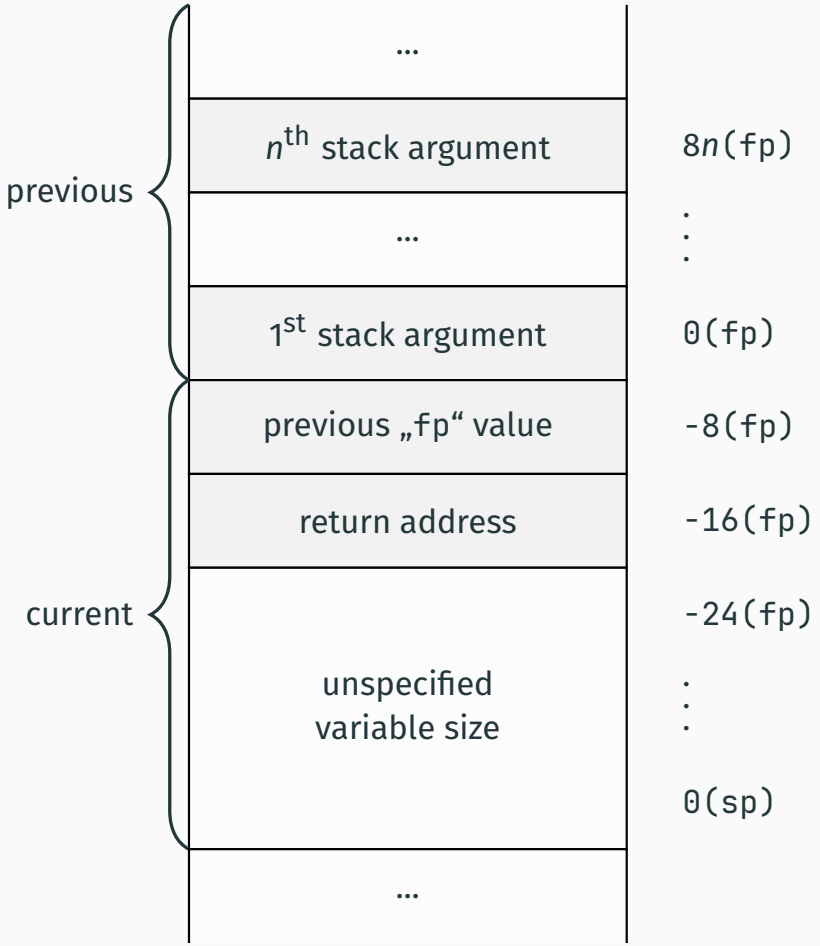
Kompilierung zu low-level Architekturen

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

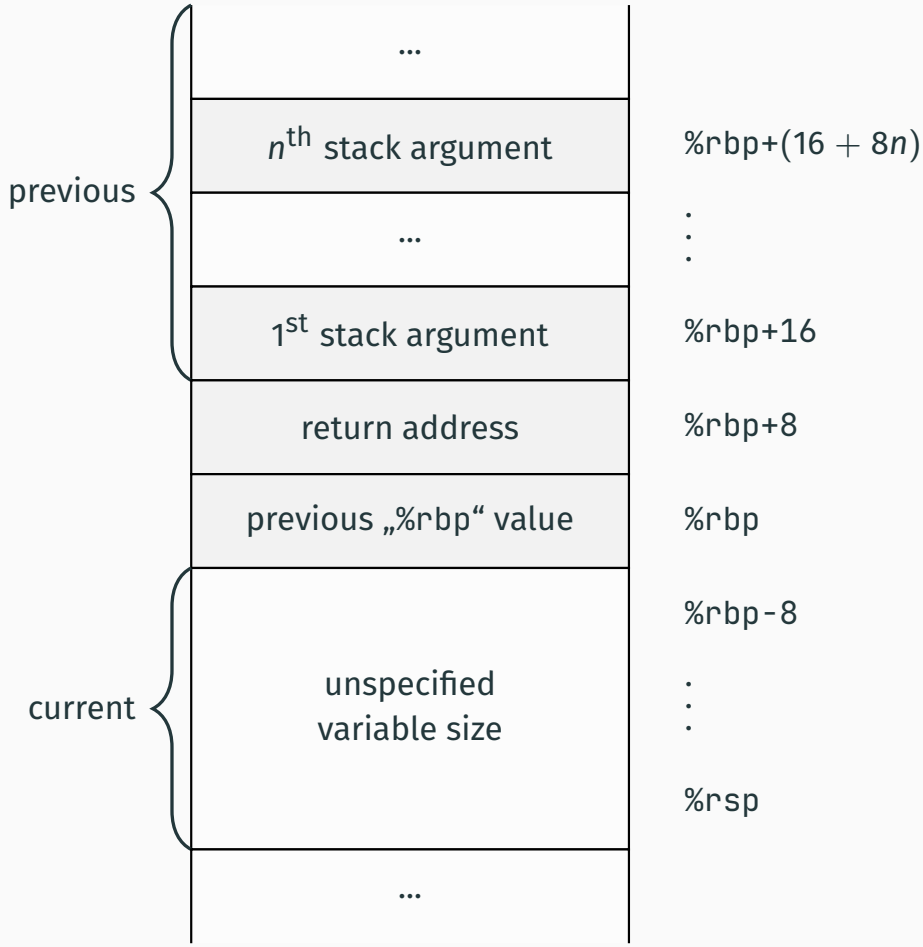
Abstraktionsgrad von Assembly



Stacklayouts



RISC-V



x64

Kompilierung zu RISC-V

Was ist RISC-V?

- **R**educed **I**nstruction **S**et **C**omputer (RISC)
- Forschungsprojekt der UC Berkeley
- Ziel: Lösen der Probleme vieler CISC Architekturen
- Simplizität und Erweiterbarkeit
- Unterstüztung 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

Beispiel

TODO: maybe fib?

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

Ausgabe

```asm
.global _start

.section .text

_start:
    call main..main
    li a0, 0
    call exit

main..main:
    # begin prologue
    addi sp, sp, -32
    sd fp, 24(sp)
    sd ra, 16(sp)
    addi fp, sp, 32
    # end prologue
    # begin body
    li a0, 1
    sd a0, -24(fp)    # let a = a0
    li a0, 2
    sd a0, -32(fp)    # let b = a0
    ld a0, -24(fp)    # a
    ld a1, -32(fp)    # b
    add a0, a0, a1
    call exit
    # end body

epilogue_0:
    ld fp, 24(sp)
    ld ra, 16(sp)
    addi sp, sp, 32
    ret
```

Fazit zu RISC-V

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

Einleitung

- **TODO: @RubixDev Write this**

Fazit zu x64

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

Vergleich mit high-level Zielen

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

Finale Anmerkungen & Fazit

Finale Anmerkungen & Fazit

TODO: @RubixDev @MikMuellerDev Write this

Quellenverzeichnis

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 Samual 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`.