The Conversion of Source Code to Machine Code

Silas Groh, Mik Müller

17. Mai 2023

Carl-Fuhlrott-Gymnasium

TODO

TODO

- **TODO: translate everything into German**
- **TODO: fix lirstings captions**

Einstieg & Motivation

Einstieg & Motivation

- Computerprogramme werden oft in speziell entwickelten Programmiersprachen verfasst
- Vorteile eines hohen Abstraktionsgrades [Dan05a, S. 9]:
  - Entwicklung ist einfacher und schneller
  - Programme sind portabel
  - Reperaturen und Erweiterungen sind einfacher

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   }
```

Übersetzung

```
1   ; RISC-V binary
2   0000000 457f 464c 0102
3   0000010 0002 00f3 0001
4   0000020 0040 0000 0000
5   0000030 0005 0000 0040
6   0000040 0003 7000 0004
7   0000050 0000 0000 0000
8   0000060 0048 0000 0000
9   0000070 0001 0000 0000
10  0000080 0000 0000 0000
```

- Programme sollten einfach zu schreiben sein

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

Methoden zur Programmausführung

- Man unterscheidet zwischen Compilern und Interpretern
- Ein Compiler übersetzt die Sprache in ein zielspezifiesches Format, welches der Computer versteht
- Ein Interpreter führt das Programm direkt aus, ohne es vorher zu übersetzen

Etappen der Übersetzung [Wir05, S. 6–7]

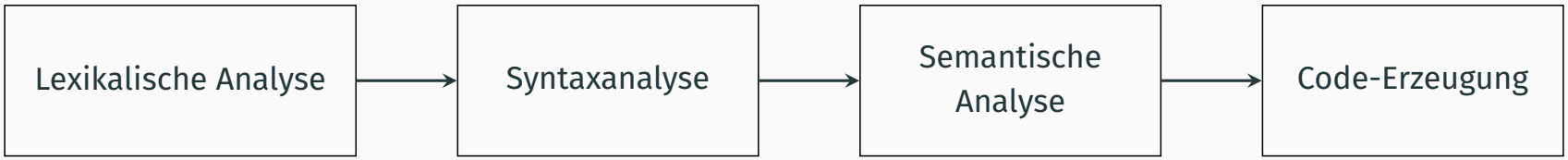


Abbildung 1 – Etappen der Übersetzung.

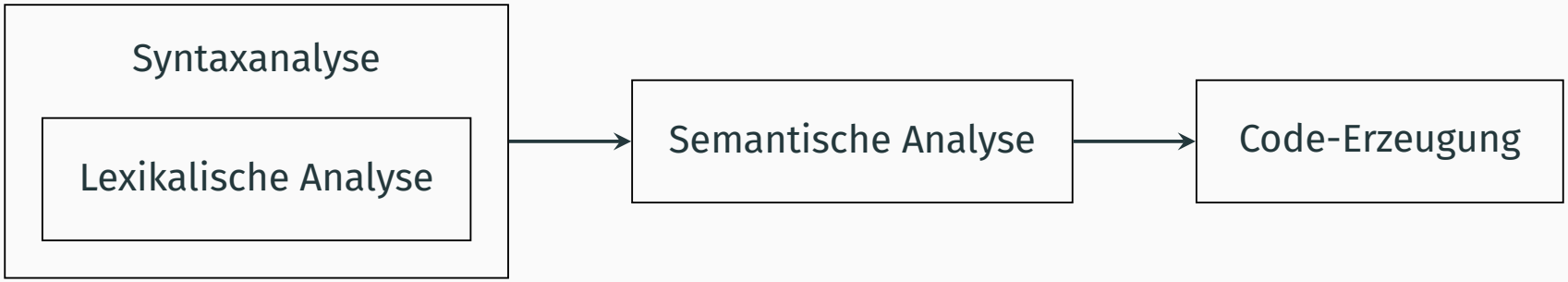


Abbildung 2 – Etappen der Übersetzung (angepasst).

Fähigkeiten von rush

Tabelle 1 – Die wichtigsten Fähigkeiten von rush.

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

Tabelle 2 – Datentypen in rush.

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);`

Beispiel Programmtext 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 }
```

Listing 1.1 – Beispiel: Berechnung von Fibonaccizahlen in rush.

Fakten über rush

- Im Git Commit ‘9953dd8’ umfasste das Projekt 17526 Zeilen Programmtext[1]
- Das Projekt enthält zwei Interpreter, einen Transpiler und vier Compiler
- Die Komponenten verwenden alle die selbe Semantikanalyse und den selben Lexer und Parser

[1] Leerzeilen und Kommentare werden nicht gezählt.

Programmtext der einzelnen Komponenten

Tabelle 3 – Zeilen Programmtext pro Komponente.

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

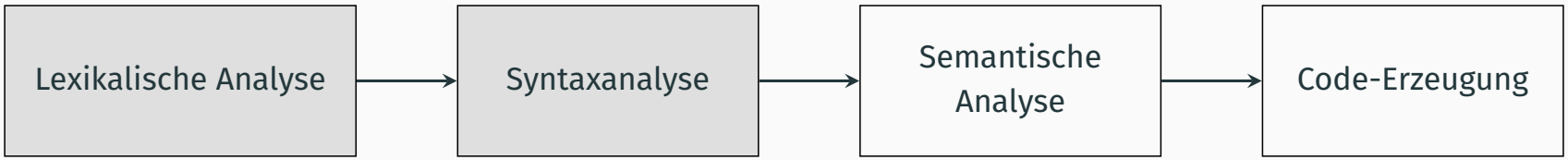


Abbildung 3 – Etappen der Übersetzung: Syntaxanalyse.

[Wir05, S. 6–7]

Lexikalische & 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' }- ;
```

Listing 1.2 – Ein Beispiel für eine kontextfreihe Grammatik (EBNF).

Abstrakter Syntaxbaum

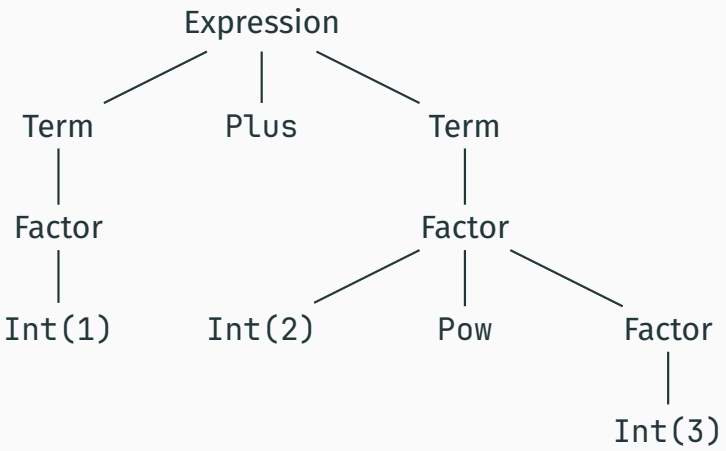


Abbildung 4 – Abstakter Syntaxbaum für ‘1+2**3’.

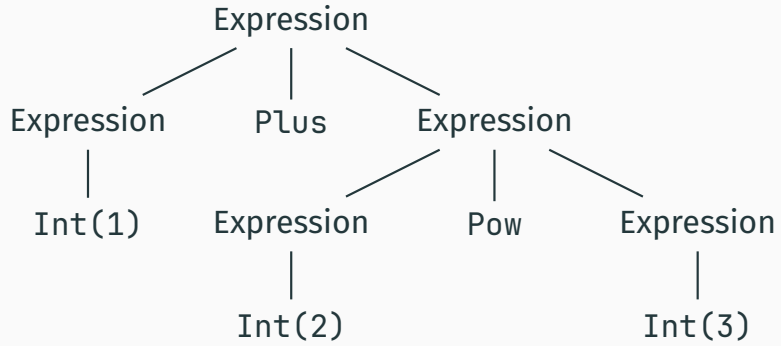


Abbildung 5 – Abstakter Syntaxbaum für ‘1+2**3’, erstellt durch Pratt-Parsing.

Semantische Analyse

Etappen der Übersetzung: Semantische Analyse

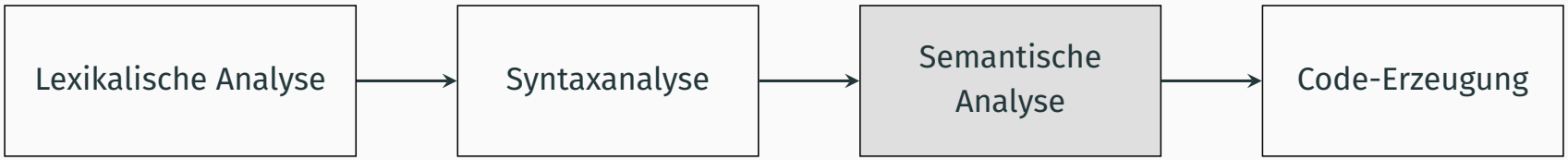


Abbildung 6 – Etappen der Übersetzung: Semantische Analyse.

[Wir05, S. 6–7]

Semantische Analyse & Semantikregeln

- Findet nach der Syntaxanalyse und vor der Übersetzung statt
- Validierung der semantischen Eigenschaften des Programms
- Die semantischen Regeln einer Programmiersprache werden oft mittels einer natürlichen Sprache beschrieben
- Für rush wurde ein Dokument erstellt, welches die meisten Semantikregeln erklärt

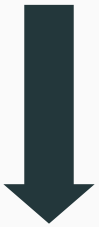
Beispiele für die Semantikregeln von rush

- Jede Variable, jede Funktion und jeder Parameter besitzt einen Datentyp, der nach Definierung nicht mehr geändert werden kann
- Eine Funktion muss immer mit den Argumenten aufgerufen werden, die zu den Parametern passen
- Jeder Funktionsname muss eindeutig sein
- Die „`main`“ Funktion liefert immer den „`()`“ Datentyp und akzeptiert keine Parameter
- Jede Variable muss Definiert sein, bevor diese Verwendbar ist
- Logische und mathematische Operationen erfolgen nur, wenn die Operanten den selben Datentypen besitzen
- Eine definierte Variable *sollte* verwendet werden

...

Beispiel: Invalides rush Programm

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

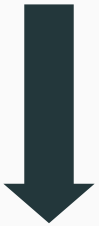


Fehlerausgabe

```
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

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

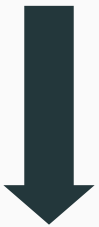


Fehlerausgabe

```
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

```
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

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
```

Semantische Analyse für rush

- Muss in der Lage sein, ein invalides von einem validen Programm zu unterscheiden
- Kann zusätzlich hilfreiche Warnungen und Informationen generieren
- Fügt Informationen über Datentypen zu dem (vom Parser erstellten) AST hinzu
- Führt triviale Optimierungen der Programmstruktur durch

Hinzufügen von Informationen über Datentypen

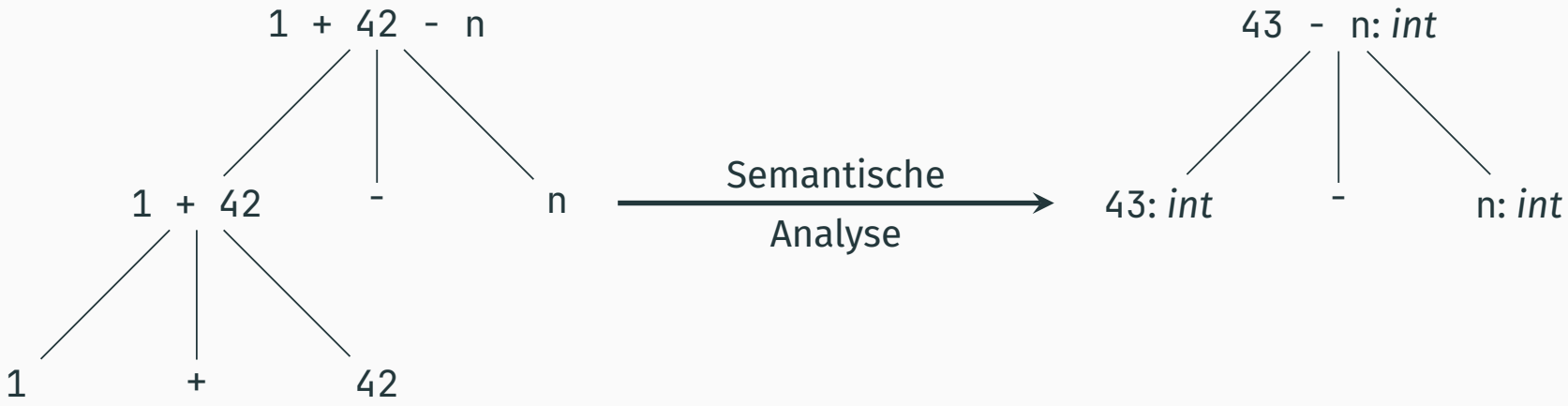


Abbildung 7 – Auswirkungen der semantischen Analyse auf den AST.

Interpreter

Tree-walking Interpreter

TODO: RubixDev Write this

Virtuelle Maschine

Virtuelle Maschine

- Oft bezeichnet eine *virtuelle Maschine* (VM) ein Softwareprogramm, welches einen echten Computer simuliert
- Hierbei werden oft auch Geräte wie das Display, Lautsprecher oder die Festplatte miteinbezogen
- In diesem Kontext bezeichnet der Begriff jedoch eine Software, die wie die CPU eines Rechners funktioniert

Wie eine CPU Programme ausführt

- 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

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>,
}
```

Listing 1.3 – Struct Definition der VM.

- **TODO: fix broken caption**
- „`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

Speicherstruktur der rush VM.

- Unterscheidung zwischen zwei Arten der Adressierung
- *relative Adressierung*: „`svari *rel[0]`“
- *absolute Adressierung*: „`svari *abs[0]`“

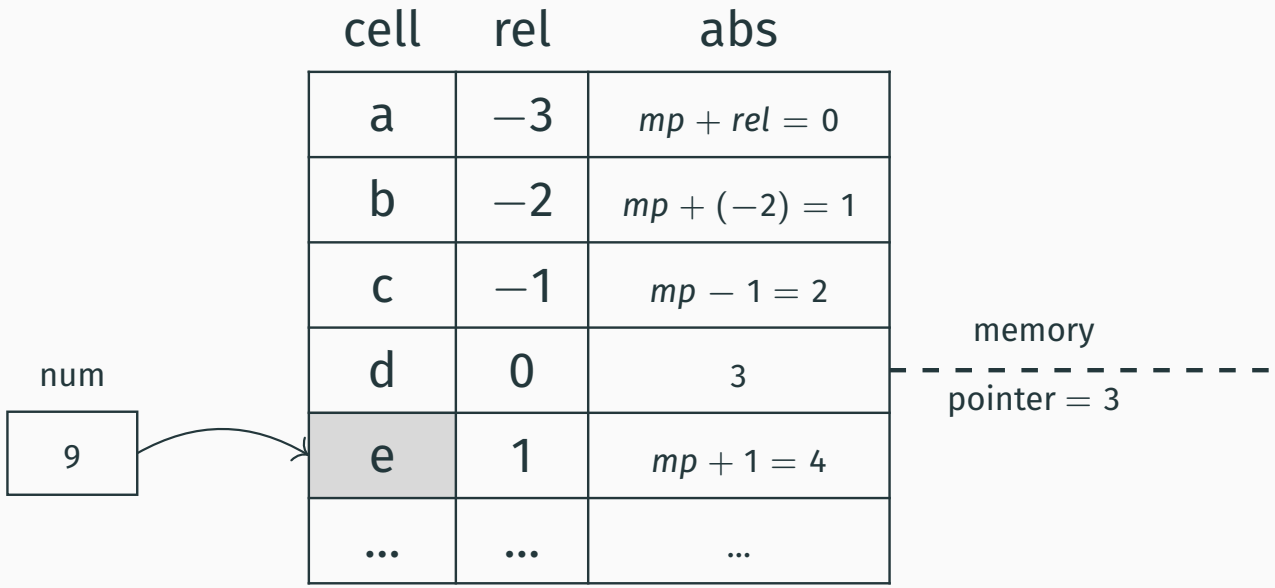


Abbildung 8 – Speicherstruktur der rush VM.

Ein Beispielprogramm der rush VM

```
0: (prelude)
        setmp 0
        call 1
1: (main)
        setmp 0
        push 1000
        call 2
        exit
2: (rec)
        setmp 1
        svari *rel[0]
        push *rel[0]
        gvar
        push 0
        eq
        jmpfalse 9
        push 0
        jmp 14
        push *rel[0]
        gvar
        push 1
        sub
        call 2
        setmp -1
        ret
```

Listing 1.4 – Beispielprogramm der rush VM.

Struktur der Programme der rush VM

- Verwendung eines Stacks für Funktionsaufrufe und temporäre Operationen
- Unterteilung in Funktionen
- Jede Funktion enthält mehrere Anweisungen
- Die Funktions- und Variablennamen wurden durch Indizes ersetzt
- Im Git Commit „9953dd8“ beinhaltete die VM ca. verschiedene 30 Befehlscodes
- Jede Anweisung besteht aus einem Befehlscode mit einem optionalen Operanden
- Beispiele: „`add`“ oder „`call 2`“

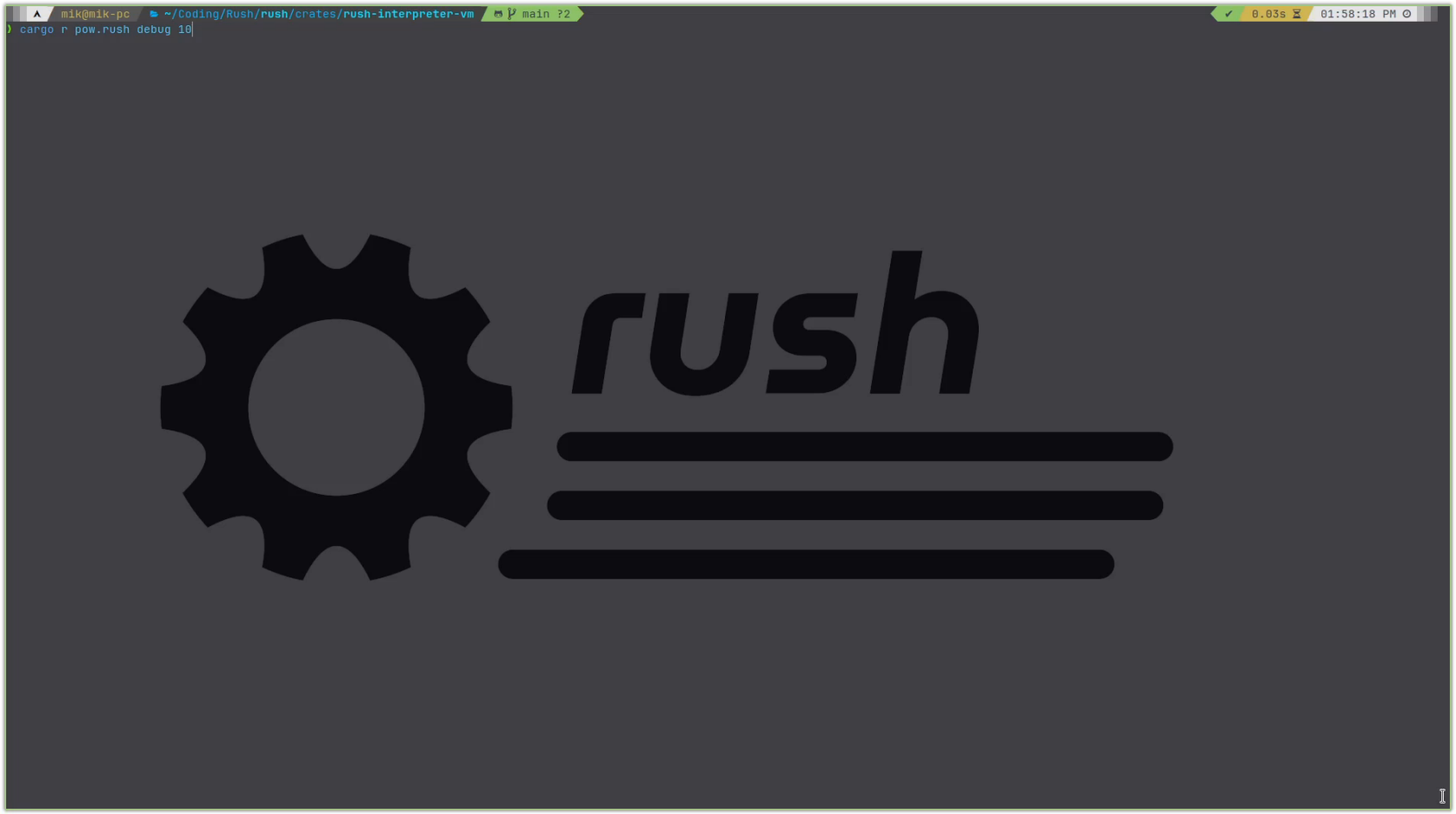
VM: Fazit

- Ca. 2.7 mal schneller als der tree-walking interpreter
- Implementierung des Compilers stellte sich als eher einfach heraus
- Die Stack-basierte Architektur erleichterte die Implementierung des Compilers
- Gleichzeitige Entwicklung der VM und des Compilers erleichterte einige Vorgänge
- Der Compiler profitiert von dem hohen Abstraktionsgrad der VM

VM: Demonstration

```
1   fn main() {
2       exit(pow(2, 4)); // 2 ** 4 = 16
3   }
4
5   fn pow(mut base: int, mut exp: int) -> int {
6       if exp == 0 {
7           return 1;
8       }
9       if exp < 0 {
10          return 0;
11      }
12
13      let mut acc = 1;
14
15      while exp > 1 {
16          if (exp & 1) == 1 {
17              acc *= base
18          }
19          exp /= 2;
20          base *= base;
21      }
22
23      acc * base
24  }
```

VM: Demonstration



Kompilierung zu high-level Architekturen

Wie ein Compiler den AST traversiert

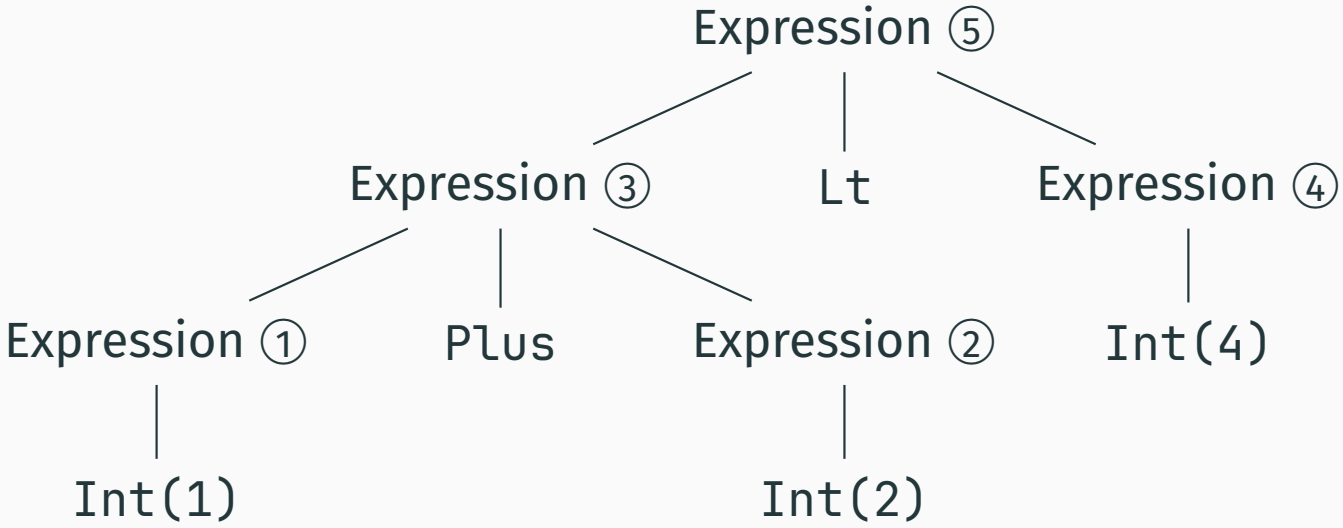


Abbildung 9 – AST zu „`1 + 2 < 4`“.

```
1  r0 = 1
2  r1 = 2
3  r2 = add r0, r1
4  r3 = 4
5  r4 = lt r2, r3
```

Listing 1.5 – Beispielausgabe zu „`1 + 2 < 4`“.

Kompilierung zu WebAssembly

Kompilierung zu WebAssembly

TODO: @RubixDev Write this

Kompilierung zu LLVM

Was ist LLVM?

- Startete als Forschungsprojekt von *Chris Lattner* [Lat02]
- Neben anderen großen Projekten verwendet auch *Rust* LLVM als sein Backend [McN21, p. 373].
- Wird auch von der *Swift* Programmiersprache (Apple) verwendet [Hsu21, preface].
- Kann aus einer Zwischendarstellung Code erzeugen, der auf der Zielmaschine ausführbar ist
- Das Framework ist bekannt für seine aggressiven Optimisierungsmaßnamen
- LLVM stellt eine API zur verfügung, mit welcher eine sogenannte *intermediate representation* (IR) erzeugt werden kann [Hsu21, preface]

$\Rightarrow$ LLVM ist das backend eines Compilers

Rolle von LLVM in einem Compiler

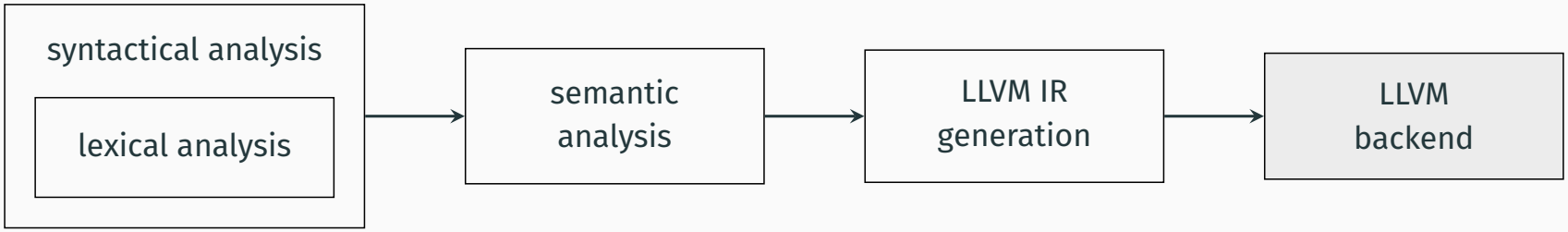


Abbildung 10 – Etappen der Übersetzung mit Verwendung von LLVM

Der rush LLVM Compiler

- Verwendet eine Rust library names Inkwell, um mit LLVM zu interargieren
- Erzeugt mittels dieser API die 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  }
```

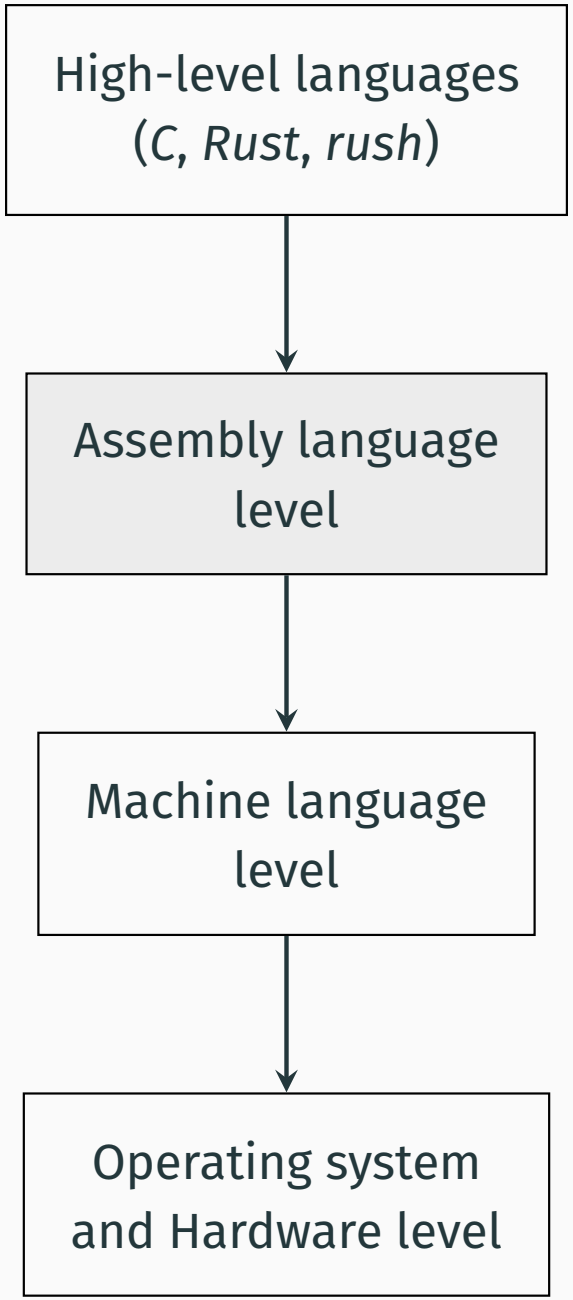
Ein LLVM Beispielprogramm: Ausgabe

```
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)
```

Kompilierung zu low-level Architekturen

Kompilierung zu low-level Architekturen

TODO: @MikMuellerDev Write this


High-level languages
(*C*, *Rust*, *rush*)
Assembly language
level
Machine language
level
Operating system
and Hardware level

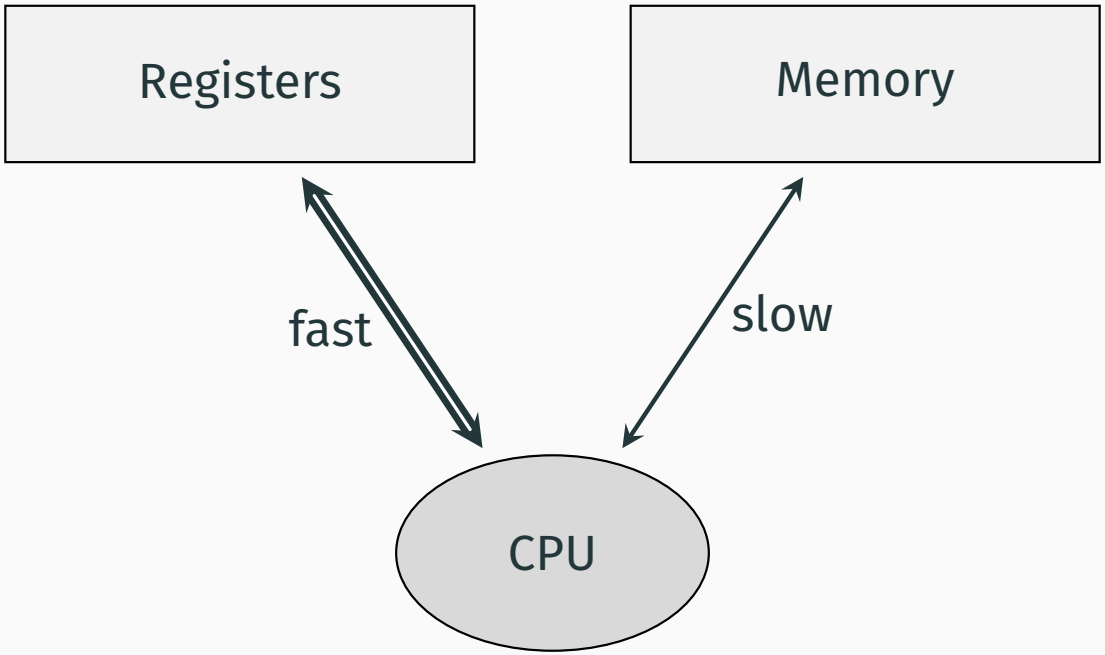


Abbildung 12 – Relationship between registers, memory, and the CPU [Dan05b, pp. 20–21].

foobar

aa

Kompilierung zu RISC-V

Was ist RISC-V?

- **TODO: @MikMuellerDev Write this**

Tabelle 4 – Registers of the RISC-V architecture [WA19, p. 155].

Register(s)	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

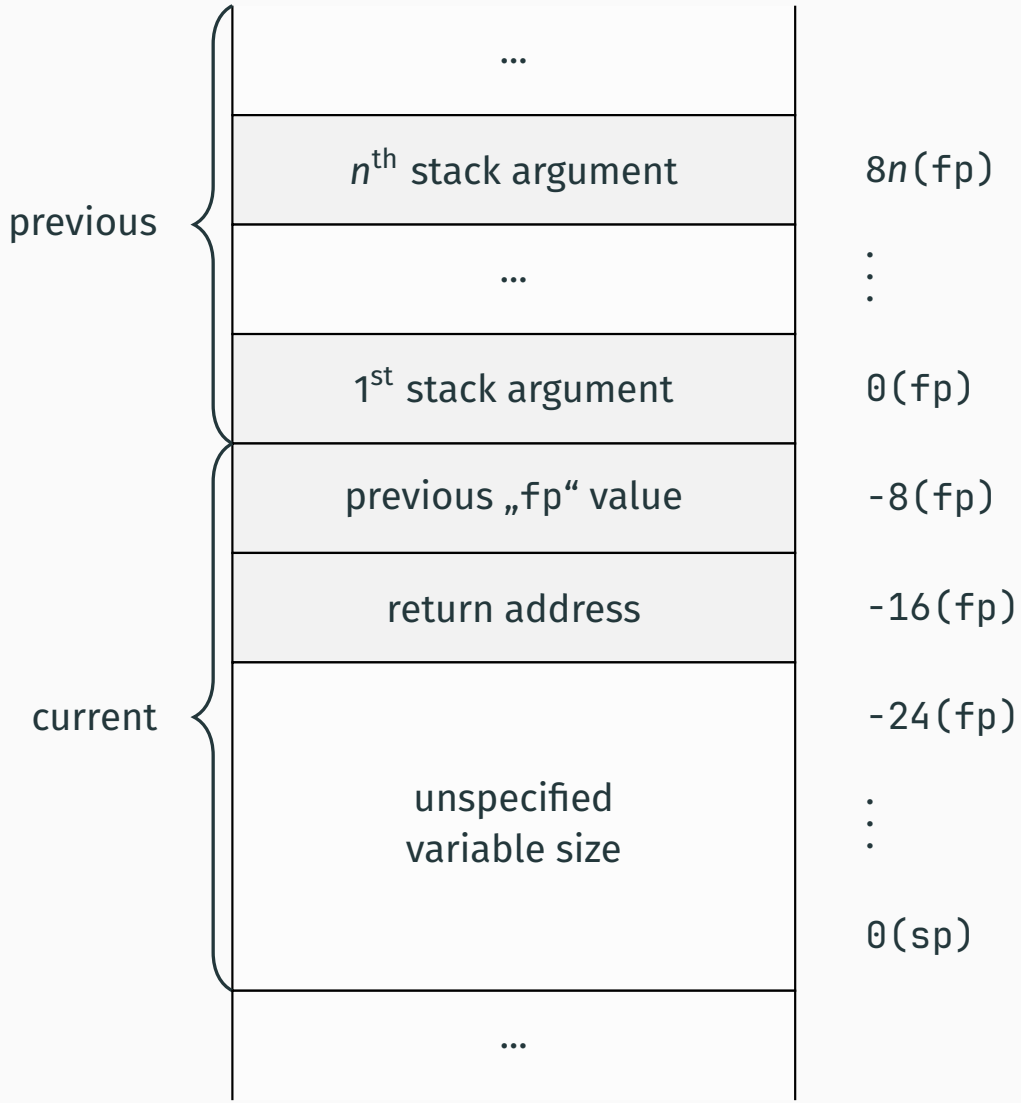


Abbildung 13 – Stack layout of the RISC-V architecture.

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

Listing 1.6 – A rush program calculating the sum of two integers.

```
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
```

Listing 1.7 – Compiler output of the rush program in Listing 1.6.

Finale Anmerkungen & Fazit

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.

[Dan05a] Sivarama P Dandamudi. *Guide to RISC processors*. Ottawa, Canada: Springer International Publishing, Feb. 2005. ISBN: 0-387-21017-2.

[Dan05b] Sivarama P. Dandamudi. *Introduction to Assembly Language Programming: For Pentium and RISC Processors*. 2. Aufl. Springer International Publishing, 2005. ISBN: 0-387-20636-1.

[Wir05] Niklaus Wirth. *Compiler Construction*. Zürich, 2005. ISBN: 0-201-40353-6.

[WA19] Andrew Waterman und Krste Asanović. „The RISC-V Instruction Set Manual Volume I: Unprivileged ISA“. In: (Dez. 2019). Hrsg. von Andrew Waterman, Krste Asanović und Sivive Inc. URL: `https://riscv.org/technical/specifications/`.

[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.