



The Conversion of Source Code to Machine Code

Silas Groh, Mik Müller

19. April 2023

Carl-Fuhlrott-Gymnasium

Einstieg & Motivation

- Computerprogrammer werden oft in speziell entwickelten Programmiersprachen verfasst
- Vorteile eines hohen Abstraktionsgrades [Dan05, S. 9]:
 - Entwicklung ist einfacher und schneller
 - Programme sind portabel
 - Instandhaltung ist 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 00000000 457f 464c 0102  
3 00000010 0002 00f3 0001  
4 00000020 0040 0000 0000  
5 00000030 0005 0000 0040  
6 00000040 0003 7000 0004  
7 00000050 0000 0000 0000  
8 00000060 0048 0000 0000  
9 00000070 0001 0000 0000  
10 00000080 0000 0000 0000
```

- Programme sollten einfach zu schreiben sein

⇒ Ein Computer muss diese jedoch auch *einfach* verarbeiten

- Man unterscheidet zwischen *Compilern* und *Interpretern*
- Ein Compiler (auch *Übersetzer*) übersetzt die Sprache in ein zielspezifisches Format, sodass ein Computer dieses verstehen kann
- Ein Interpreter führt das Programm direkt aus, ohne es vorher zu bearbeiten



Abbildung 1 – Etappen der Übersetzung.



Abbildung 2 – Etappen der Übersetzung (angepasst).

Tabelle 1 – Die wichtigsten Fähigkeiten von rush.

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>

Tabelle 2 – Datentypen in rush.

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>

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.

- Im Git Commit '9953dd8' umfasste das Projekt 17526 Zeilen Programmtext¹
- Das Projekt enthält einen Lexer, einen Parser, fünf Compiler, einen Interpreter und einen Transpiler

¹Leerzeilen und Kommentare werden nicht gezählt.

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



Abbildung 3 – Etappen der Übersetzung: Syntaxanalyse.

[Wir05, S. 6–7]

- Gruppieren des Programmtextes in Tokens
- Analyse der Syntax des Programms
- Generation 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'
6             | '9' }- ;
```

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

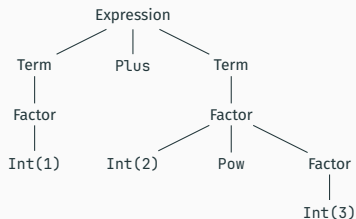


Abbildung 4 – Abstrakter Syntaxbaum für '1+2*3'.

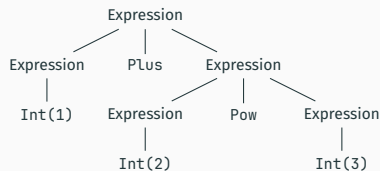


Abbildung 5 – Abstrakter Syntaxbaum für '1+2*3', erstellt durch Pratt-Parsing.

Semantische Analyse



Abbildung 6 – Etappen der Übersetzung: Semantische Analyse.

[Wir05, S. 6–7]

- 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

```
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

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

- 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

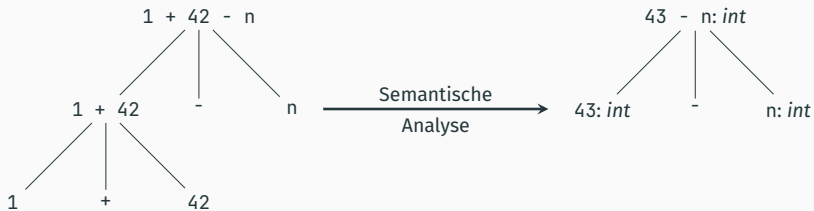


Abbildung 7 – Auswirkungen der semantischen Analyse auf den AST.

Interpreter

TODO: RubixDev Write this

Virtuelle Maschinen

- 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

- 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

```
_____ 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 }
```

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 *Befehlszähler* 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]“

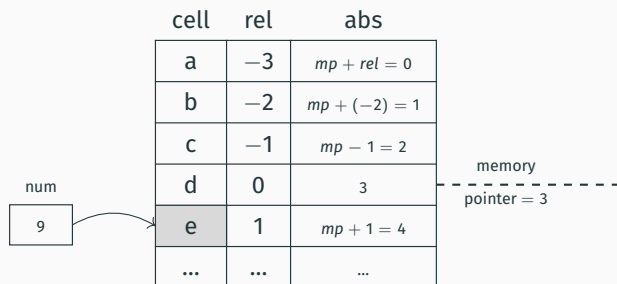


Abbildung 8 – Speicherstruktur der rush VM.

Ein Beispielprogramm der rush VM

```
1  0: (prelude)
2      setmp 0
3      call 1
4  1: (main)
5      setmp 0
6      push 1000
7      call 2
8      exit
9  2: (rec)
10     setmp 1
11     svari *rel[0]
12     push *rel[0]
13     gvar
14     push 0
15     eq
16     jmpfalse 9
17     push 0
18     jmp 14
19     push *rel[0]
20     gvar
21     push 1
22     sub
23     call 2
24     setmp -1
25     ret
```

Listing 1.4 – Beispielprogramm 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“ beinhaltet die VM ca. verschiedene 30 Befehlscodes
- Jede Anweisung besteht aus einem Befehlscode mit einem optionalen Operanden
- Beispiele: „`add`“ oder „`call 2`“

- 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

Kompilierung zu high-level Architekturen

Wie ein Compiler den AST traversiert

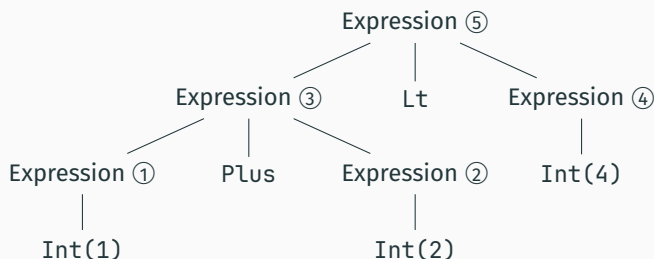


Abbildung 9 – AST zu „1 + 2 < 4“.

```
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

TODO: Write this

Kompilierung zu 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 Optimierungsmaßnahmen
 - LLVM stellt eine API zur Verfügung, mit welcher eine sogenannte *intermediate representation* (IR) erzeugt werden kann [Hsu21, preface]
- ⇒ LLVM ist das Backend eines Compilers

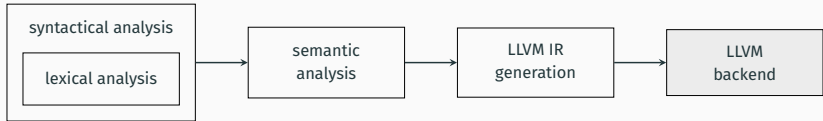


Abbildung 10 – Etappen der Übersetzung mit Verwendung von LLVM

Kompilierung zu low-level Architekturen

TODO: Write this

Finale Anmerkungen & Fazit

TODO: 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.