

Advanced Programming SS 2020: LLVM



<https://xkcd.com/303/>

```
# Compute Ackermann's function
# for two parameters a and b.
def ack(a, b)
    if a == 0 then
        b + 1
    else if b == 0 then
        ack(a - 1, 1)
    else
        ack(a - 1, ack(a, b - 1))

def run(a, b)
    ack(a, b)
```

- Wie arbeitet ein Compiler?

```
#include <stdio.h>

int main(void)
{
    // My first C program!
    printf("Hello, world!\n");
    return 0;
}
```

- Wie arbeitet ein Compiler?

...

```
int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

Präprozessor

- Includes (Copy-Paste von Headern, rekursiv!)
- Kommentare entfernen
- Trigraphen ersetzen
- Makrosubstitution
- Bedingtes Übersetzen (`#ifdef` und Co.)

- Wie arbeitet ein Compiler?

...

```
keyword(`int`) ident(`main`) paren(`(`) keyword(`void`) paren(`)`)  
paren(`{`)  
ident(`printf`) paren(`(`) literal(`"Hello, world!\n"`) paren(`)`)  
semi(`;`)  
keyword(`return`) literal(`0`) semi(`;`)  
paren(`}`)
```

Lexer

- Zerlegung des Codes in Tokens
- Entfernung von Whitespaces

- Wie arbeitet ein Compiler?

```
func_def
(
  prototype: (name: main, ret: int, params: []),
  body:
  [
    call_stmt(func: printf, params: [str_literal("Hello, world!\n")]),
    ret_stmt(int_literal(0))
  ]
)
```

Parser

- Gliedern der Tokens in höbersprachliche Konstrukte
- Hierarchische Anordnung → „Abstract Syntax Tree“ (AST)
- Verifikation: Nicht jedes Token ist überall erlaubt

- Wie arbeitet ein Compiler?

```
func_def
(
  prototype: (name: main, ret: int, params: []),
  body:
  [
    call_stmt(func: puts, params: [str_literal("Hello, world!\n")]),
    ret_stmt(int_literal(0))
  ]
)
```

Optimierer

- Extrem kompliziert!
- Evtl. Überführung in andere Zwischenrepräsentationen
- Teilweise plattformabhängig (z.B. durch Registeranzahl)

- Wie arbeitet ein Compiler?

```
sub    rsp,0x8
lea    rdi,[rip+0xfb9]
call   0x1030 <puts@plt>
xor     eax,eax
add     rsp,0x8
ret
```

Codegenerator

- Überführung in Assembler für die Zielplattform

- Wie arbeitet ein Compiler?

```
48 83 ec 08
48 8d 3d b9 0f 00 00
e8 e0 ff ff ff
31 c0
48 83 c4 08
c3
```

Assembler

- Generierung von Maschinencode
- Überführung in Objektdaten (z.B. ELF)

- Wie arbeitet ein Compiler?

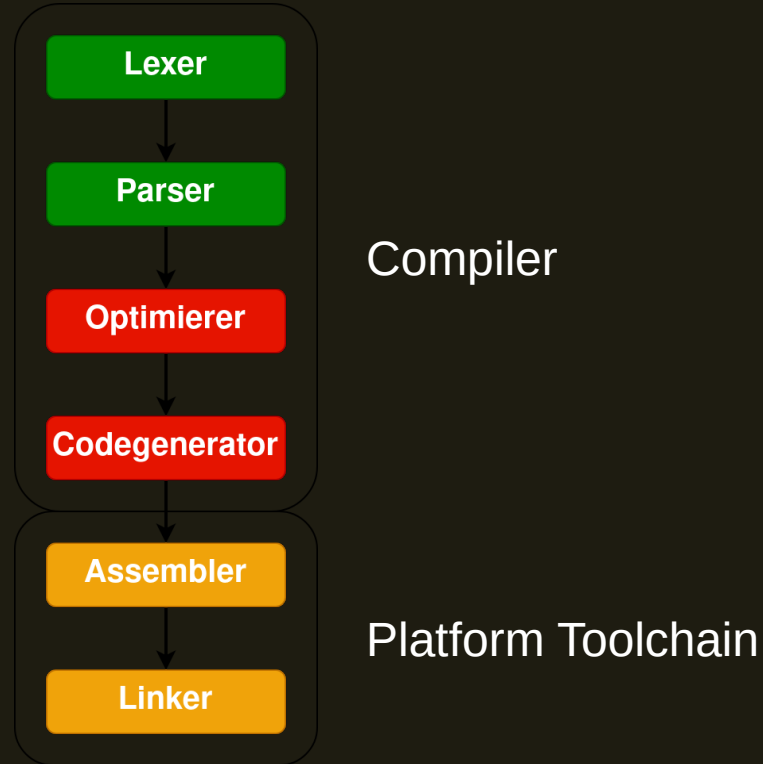
```
48 83 ec 08
48 8d 3d b9 0f 00 00
e8 e0 ff ff ff
31 c0
48 83 c4 08
c3
```

Linker

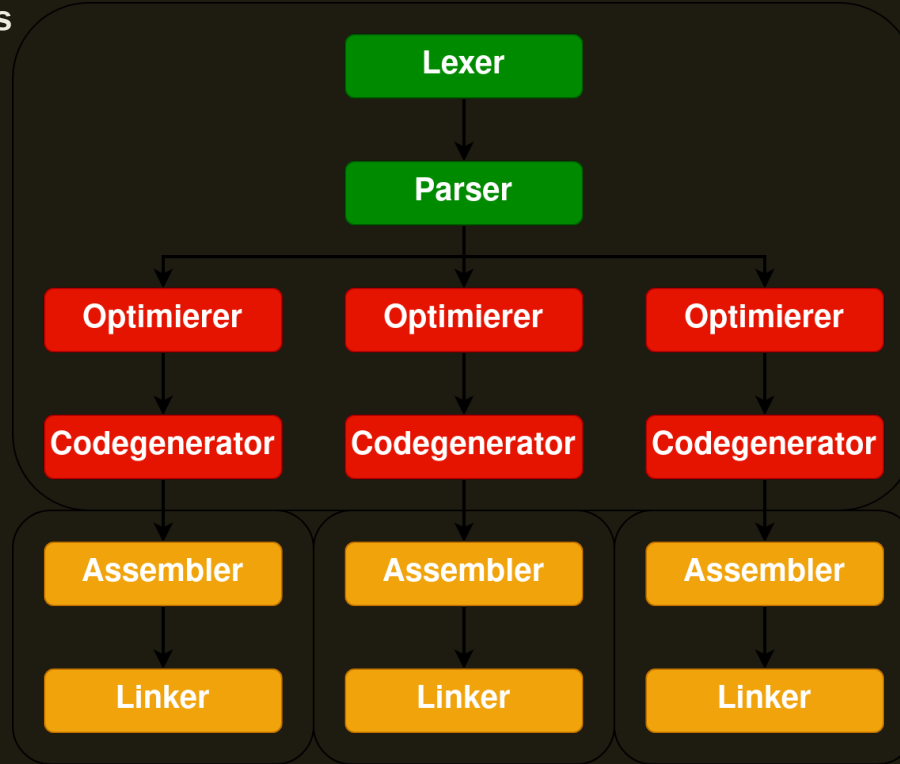
- Verknüpfung mit weiteren Objektdaten
- Auflösung von Funktionsverweisen

Compilerbau

- **Monolithisches Modell**



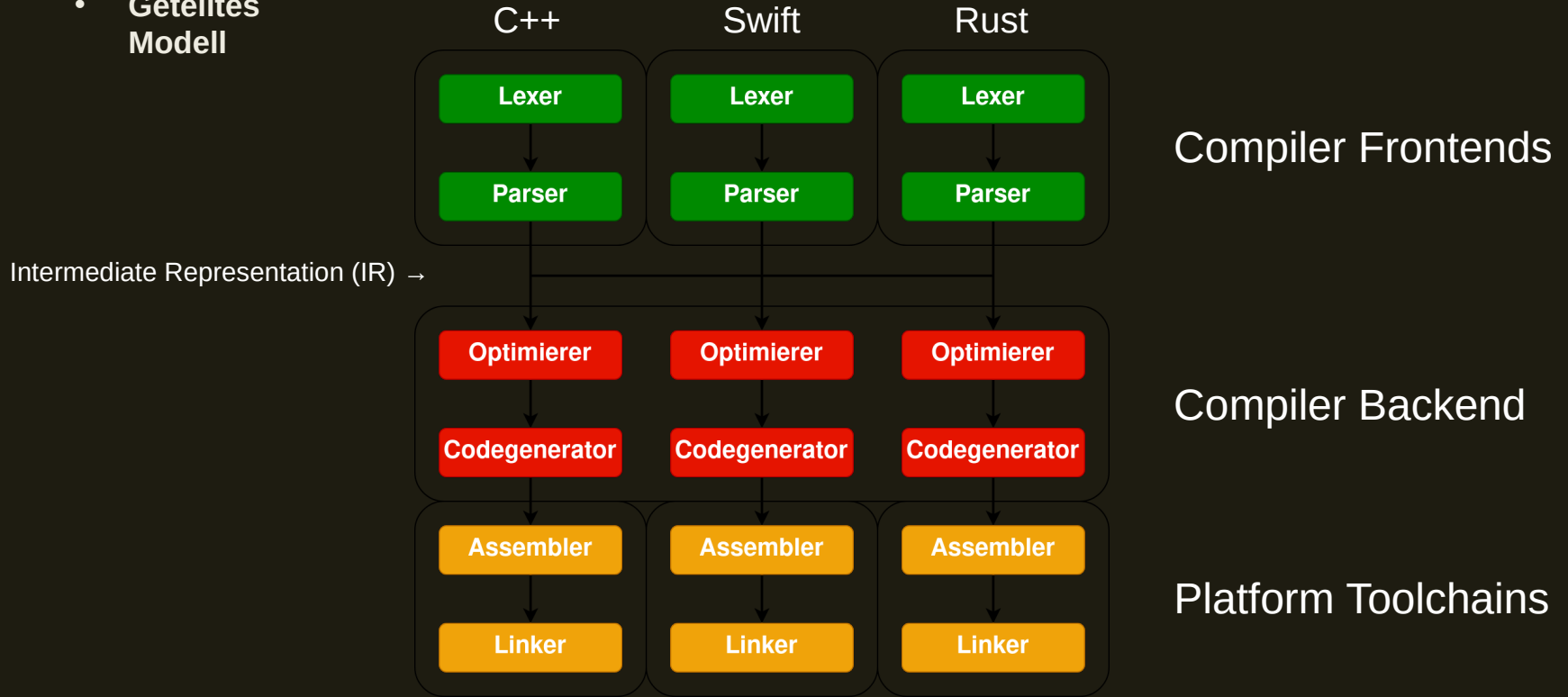
- Monolithisches Modell



Compiler

Platform Toolchain

- **Geteiltes Modell**



- LLVM



- **LLVM**

- Open-Source-Projekt an der Universität von Illinois (2000, Chris Lattner / Vikram Adve)
- Ursprünglich: „*Low Level Virtual Machine*“
- LLVM-IR: Zwischensprache, ähnlich einem High-Level-Assembler
- Schnittstellen zur Generierung von LLVM-IR aus ASTs
- Komplexe, ausgereifte Optimierung von LLVM-IR („Passes“)
- Übersetzung in plattformspezifischen Assembler
- JIT-Compiler-Unterstützung



- **Wer nutzt LLVM?**

- **clang**: Compiler für C / C++, Drop-In-Replacement für gcc
- **GHC**: Glasgow Haskell Compiler
- **Julia**: Programmiersprache mit Fokus auf numerische / wissenschaftliche Berechnungen und hohe Performanz
- **Swift**: Objektorientierte Programmiersprache für Apple-Ökosystem (inzwischen auch Linux), App-Programmierung (siehe z.B. „Swift-UI“), funktionale Aspekte, ARC, hohe Performanz
- **Rust**: Systemnahe Programmiersprache, Performanz im Bereich von C / C++, Speichersicherheit durch Ownership / Borrowchecker, objektorientierte Anleihen (aber keine Vererbung), funktionale Aspekte
- **Kotlin Native**: Kotlin außerhalb der JVM



- **LLVM-Tutorial**

- Bester Einstieg: Entwurf einer Spielzeug-Programmiersprache
- Implementierung von Lexer / Parser, Generierung von LLVM-IR
- Tutorial: Ocaml (<https://llvm.org/docs/tutorial/>)
- C++ (<https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html>)
- Third-Party-Ports für andere Sprachen
- Heute und in den nächsten drei Veranstaltungen: **LLVM-Tutorial in C#**
 - Lexer
 - Parser
 - IR-Generierung
 - Optimierung, Feinschliff



- **Kaleidoscope:** Spielzeug-Programmiersprache
 - Nur ein Datentyp: Double-Precision Floating Point (IEEE 754) → keine Annotationen
 - Funktionen: ein Rückgabewert, beliebig viele Parameter
 - „Expression-based“: wie Kotlin (bzw. noch näher an Rust), aber gänzlich ohne Statements und Variablen

```
def sqr(x)
  x * x
```

- Operatoren: +, -, *, / mit üblicher Rangfolge, außerdem: Klammern

```
def avg(a, b)
  0.5 * (a + b)
```

- **Kaleidoscope:** Spielzeug-Programmiersprache
 - Boolesche Operatoren: `==`, `<`, `>`, `<=`, `>=` liefern Double zurück (1.0 für `true`, 0.0 für `false`)
 - Bedingte Anweisungen: `if-then-else` (alles erforderlich)

```
def abs(x)
  if x < 0 then
    -1 * x
  else
    x
```

- Kommentare mit `#`, Funktionsaufrufe wie in C:

```
# Calculates the absolute average
def abs_avg(a, b)
  abs(avg(a, b))
```

- **Kaleidoscope:** Spielzeug-Programmiersprache
 - Rekursion:

```
# Computes the n'th Fibonacci number
def fib(n)
    if n < 2 then
        n
    else
        fib(n - 2) + fib(n - 1)
```

- Generell gilt: Funktionen müssen vor ihrem Aufruf deklariert sein.
- Statt Definition: Forwärtsdeklaration

```
# Computes the n'th Fibonacci number
dec fib(n)
```

- **Kaleidoscope:** Spielzeug-Programmiersprache
 - Gegen `libm` linken → Funktionen aus der Mathebibliothek nutzen

```
dec sin(x)
dec cos(x)
dec pow(base, epx)

def trig(x)
    pow(sin(x), 2) + pow(cos(x), 2)
```

- **Kaleidoscope:** Spielzeug-Programmiersprache

The command line arguments are listed and explained below.

- `-h / --help` : Print a short manual.
- `-o / --output-path` : Define the directory where output files shall be written to. By default, they land in the same directory as the source file.
- `-s / --stage` : Define up to which stage the compiler shall proceed and which kind of output shall be generated. The six stages are:
 - `lex` : Run the lexer on the source file and dump the resulting token stream into a `.lex` file.
 - `ast` : Run the parser on the output of the lexer and dump the resulting abstract syntax tree (AST) into a `.ast` file.
 - `ir` : Transform the AST to LLVM's object model, generate IR code from it and dump it into a `.ll` file.
 - `asm` : Translate the IR into the platform's native assembly syntax and dump it into a `.s` file.
 - `obj` : Assemble to native machine code and dump it into a `.o` file.
 - `exe` : Create a Makefile project including the object code file and a light C shim to provide an entry point for the operating system.
- `-O / --optimize` : Run a bunch of LLVM's optimizer passes on the resulting IR to generate more efficient code.
- `-t / --target` : By passing a [target triple](#) with this option, `kalc` supports a basic cross-compilation mode. Use with care, this does not work well with the `exe` stage.

In addition to the described arguments, a non-optional input file path has to be specified. Its file name determines the name of the output (and is assigned to the generated module).

- **Lexer**

- Gegeben: Token-Klasse
- Algebraischer Datentyp: Enum mit Payloads, die von der Variante abhängen (Swift, Rust) → Emulation in C#

```
public enum TokenType
{
    EndOfFile,
    Keyword,
    Bracket,
    Identifier,
    Number,
    Operator,
    ParameterSeparator,
    Comment,
}
```

- **Lexer**

- Gegeben: Token-Klasse
- Algebraischer Datentyp: Enum mit Payloads, die von der Variante abhängen (Swift, Rust) → Emulation in C#

```
public enum TokenType
{
    EndOfFile,
    Keyword,
    Bracket,
    Identifier,
    Number,
    Operator,
    ParameterSeparator,
    Comment,
}
```

```
public enum Keyword
{
    Dec,
    Def,
    If,
    Then,
    Else,
}
```

- **Lexer**

- Gegeben: Token-Klasse
- Algebraischer Datentyp: Enum mit Payloads, die von der Variante abhängen (Swift, Rust) → Emulation in C#

```
public enum TokenType
{
    EndOfFile,
    Keyword,
    Bracket,
    Identifier,
    Number,
    Operator,
    ParameterSeparator,
    Comment,
}
```

```
public enum Bracket
{
    RoundStart,
    RoundEnd,
}
```


- **Lexer**

- Gegeben: Token-Klasse
- Algebraischer Datentyp: Enum mit Payloads, die von der Variante abhängen (Swift, Rust) → Emulation in C#

```
public enum TokenType
{
    EndOfFile,
    Keyword,
    Bracket,
    Identifier,
    Number,
    Operator,
    ParameterSeparator,
    Comment,
}
```

```
public enum Operator
{
    Add,
    Subtract,
    Multiply,
    Divide,
    Equal,
    LowerThan,
    LowerThanEqual,
    GreaterThan,
    GreaterThanEqual,
}
```

- **Lexer**

- Gegeben: Token-Klasse
- Algebraischer Datentyp: Enum mit Payloads, die von der Variante abhängen (Swift, Rust) → Emulation in C#
- Type-Property zum Abfragen
- Spezifische Properties, die im Getter den Typ validieren: `Keyword`, `Bracket`, `Identifier` etc.
- Privater Konstruktor, stattdessen statische Methoden mit Payload-Parametern: `NewKeyword(Keyword keyword)`, `NewBracket(Bracket bracket)`, `NewIdentifier(string identifier)` etc.
- `ToString()`-Methode für „Pretty Printing“

- **Lexer**

- Gegeben: `Lexer`-Klasse
- `Character`-Struct: ebenfalls algebraisch (`char` oder `EndOfFile`)
- `Private` `Character`-Feld, wird über die `private EatChar()`-Methode konsumiert
- Konstruktor kapselt einen `TextReader`
- **(Haus-)Aufgaben für heute:**
 - In den Quellcode einarbeiten (auch `Compiler.cs` mal anschauen!)
 - Die `Lexer`-Klasse vervollständigen
 - Mit den gegebenen Beispieldateien (`samples` bzw. `samples/lex`) testen