Der C Compiler

EINE KLEINE EINFÜHRUNG IN DIE STRUKTUR VON COMPILERS IN ALLGEMEIN, GCC UND CLANG/LLVM

Überblick

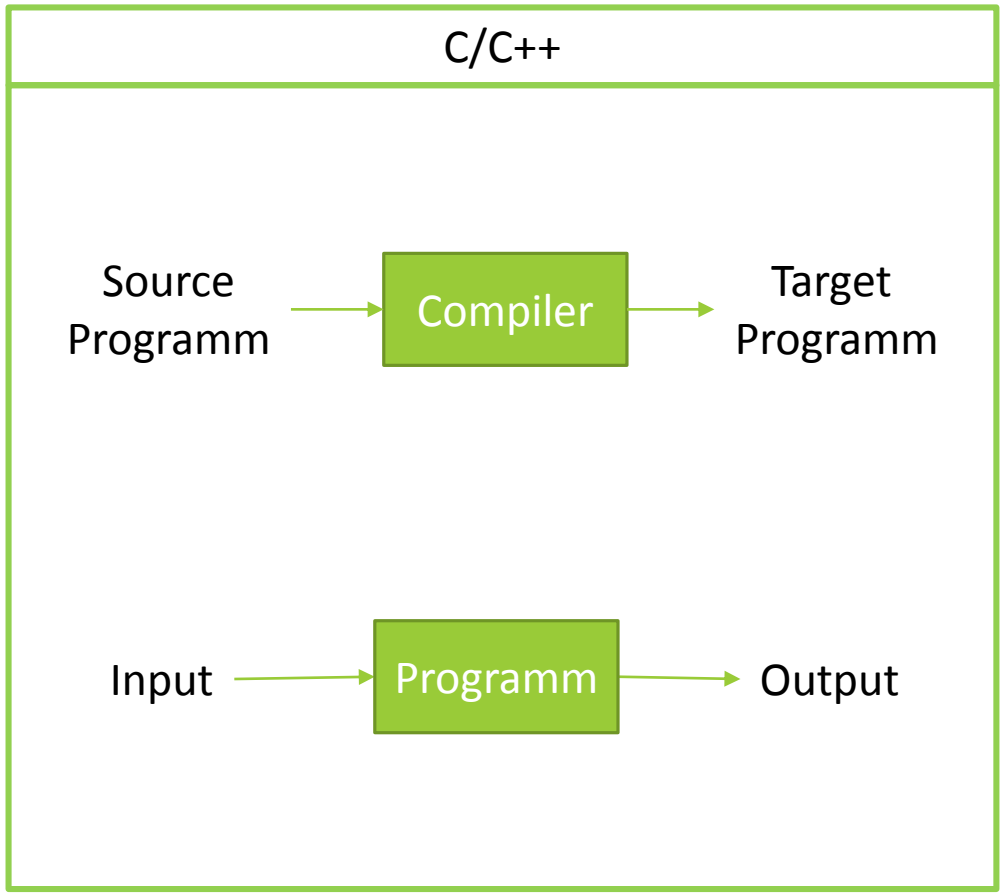
- Was ist ein Compiler
  - Compilers vs. Interpreter
  - Input und Output bei einem C Compiler
- GCC/Clang Crash Course
- Struktur eines Compilers
  - Grammatik
  - Lexer
  - Parser, Parse Tree, Abstract Syntax Tree
  - Semantische Analyse
- GCC: Geschichte, Motivation und Architektur
- Clang: Geschichte, Motivation und Architektur
- Assembler, Linker und Loader

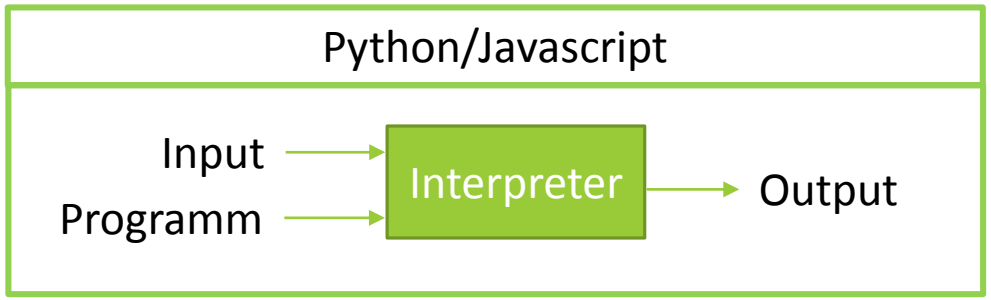
Was ist ein Compiler?

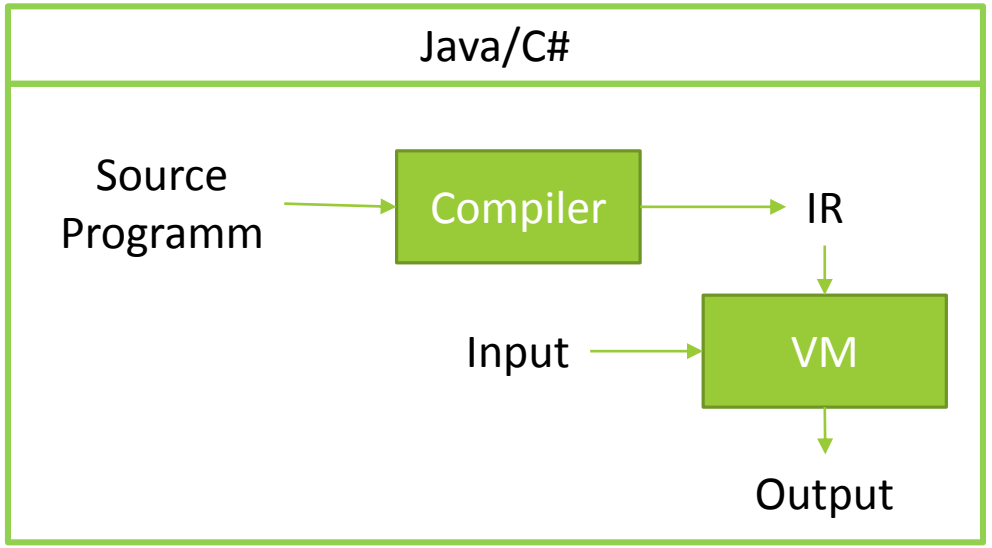
Was ist ein Compiler?

Ein Compiler ist nichts weiteres als ein Programm, das ein Programm in einer Sprache lesen kann und es in eine andere Spache umwandelt.

Compiler vs Interpreter







Input ->IR-> Output mit Clang/LLVM

```c
#include<stdio.h>

main(){

   int  a = 2;

   int  b = 3;

   int  c =  a + b;

   int  d =  a * b;

   int  e =  c + d;

   return e;

}
```

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-
i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-
a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-freebsd9.0"

define i32 @main() nounwind {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  %e = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 2, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %c, align 4
  %2 = load i32* %a, align 4
  %3 = load i32* %b, align 4
  %mul = mul nsw i32 %2, %3
  store i32 %mul, i32* %d, align 4
  %4 = load i32* %c, align 4
  %5 = load i32* %d, align 4
  %add1 = add nsw i32 %4, %5
  store i32 %add1, i32* %e, align 4
  %6 = load i32* %e, align 4
  ret i32 %6
}
```

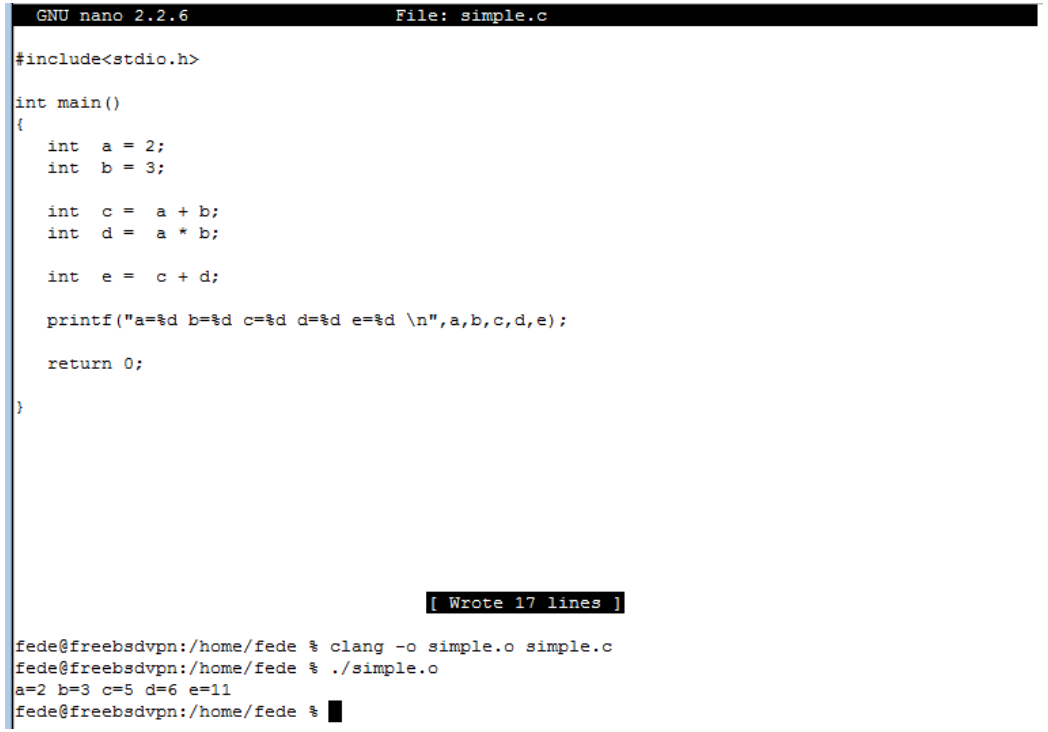
```
        .file   "simple.ll"
        .text
        .globl  main
        .align  16, 0x90
        .type   main,@function
main:                                   # @main
# BB#0:                                 # %entry
        movl    $0, -4(%rsp)
        movl    $2, -8(%rsp)
        movl    $3, -12(%rsp)
        movl    -8(%rsp), %eax
        addl    $3, %eax
        movl    %eax, -16(%rsp)
        movl    -8(%rsp), %eax
        imull   -12(%rsp), %eax
        movl    %eax, -20(%rsp)
        addl    -16(%rsp), %eax
        movl    %eax, -24(%rsp)
        ret
.Ltmp0:
        .size   main, .Ltmp0-main


        .section        ".note.GNU-stack","",@progbits
```

GCC und Clang - Crash Course

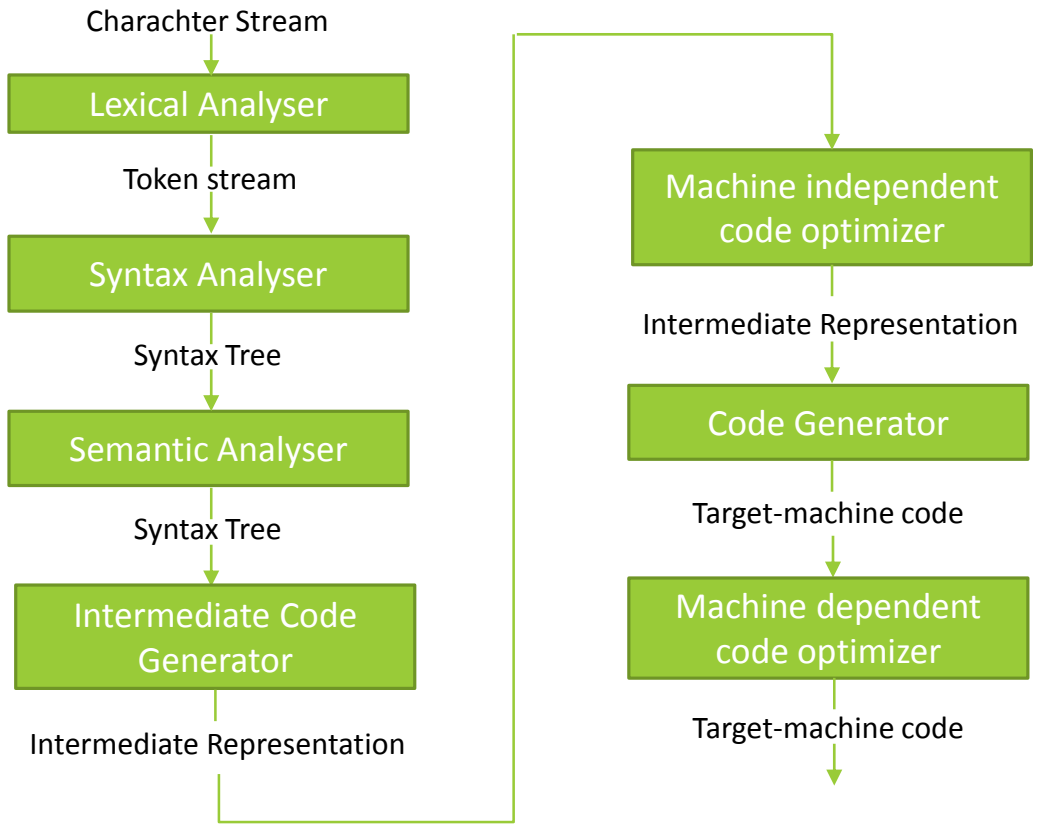
Gleich werden wir auf der Command Line folgendes zeigen:

- Wie kompiliert man eine oder mehrere Dateien.
- Wie bestimmt man wo der Output hinkommt.
- Wie kann ich eine bestimmte Spezifikation erzwingen (z.B. ANSI)
- Wie kann man aus Warnungen Fehler machen.
- Wie kann man das Optimization Level bestimmen.



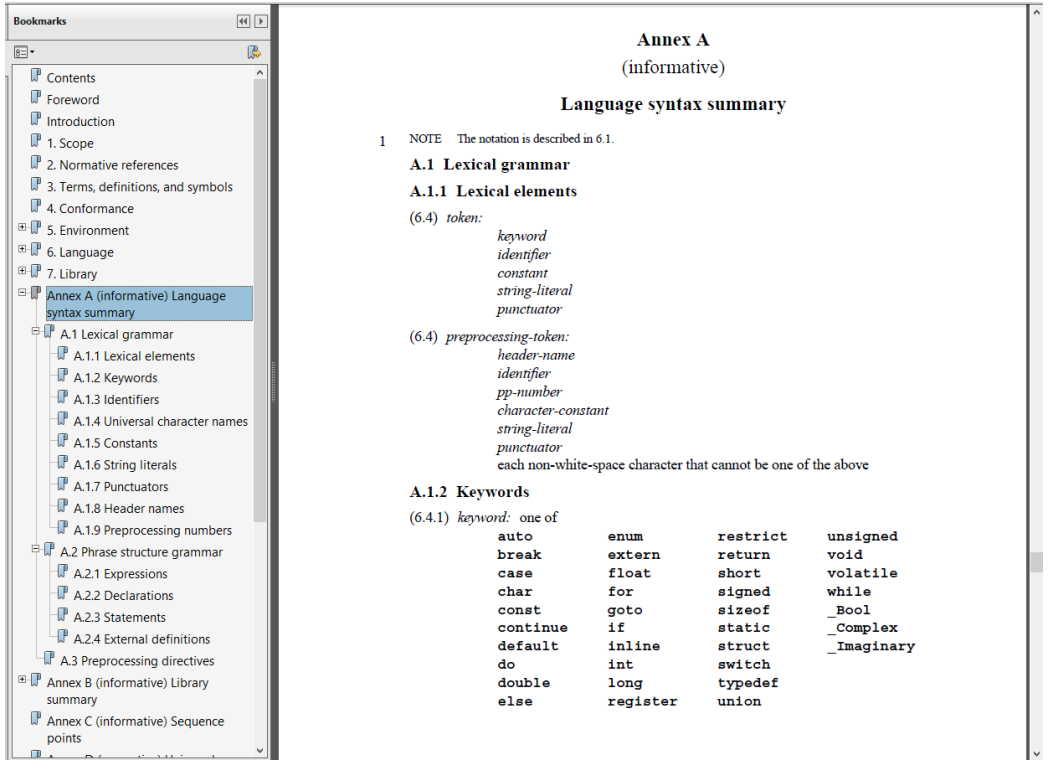
Die Strukturk eines Compilers

- Ein Compiler ist in zwei große Teile aufgeteilt: Analyse und Synthese. Diese sind auch als *Front-* und *Back-End* bekannt.
- Die Analyse teilt das Programm in seine Bestandteile auf und errichtet eine grammatikalische Struktur. Dabei werden auch weitere Informationen in der Form einer Symboltabelle gesammelt.
- Bei der Analyse soll der Compiler das Programm auf syntaktische und semantische Korrektheit überprüfen und bei Fehlern sinnvolle Rückmeldungen geben.
- Die Synthese nimmt die IR und die Symboltabelle und erzeugt daraus den erwünschten *Machine Code.*



Syntaktische Korrektheit

- Wird normaleweise durch die Angabe einer formalen Grammatik festgelegt.
- Die Definition der Formalen Grammatik erfolgt meistens mittels einer Metasprache wie zum Beispiel einer EBNF.
- In dem Fall von C wird die Gramatik im Teil 6 (*Language*) von **ISO/IEC 9899** festgelget. Eine Zusammenfassung findet man in Annex A (*Language syntax summary*).



Grammatik - Ein Beispiel mit ANTLR

- ANTLR ist ein in Java geschriebener Compilergenerator.
- Man definiert eine Grammatik und ANTLR generiert einen Compiler dazu.
- ANTLRWorks bietet die möglichkeit Grammatiken auch zu debuggen.
- Yacc und Bison sind andere Parsergeneratoren.
- Die Grammatik auf der rechten Seite akzeptiert einfache arithmetische Ausfrücke als Eingabe.
- Ein Beispiel-Input dafür:

  A = 2

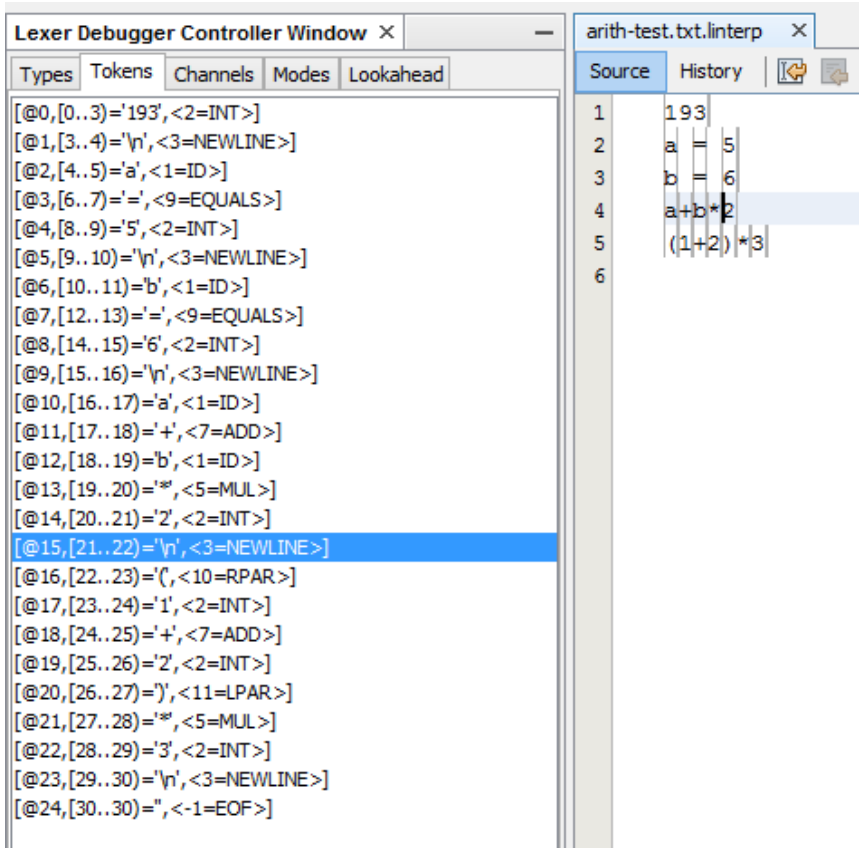
  B = A + 12 * (A + 1) – 2 * (1 + 2) + 3

```
1  grammar Arithmetic;
2  ID : [a-zA-Z]+ ;
3  INT : [0-9]+ ;
4  NEWLINE:'\r'? '\n' ;
5  WS : [ \t]+ -> skip ;
6  MUL     :   '*';
7  DIV     :   '/';
8  ADD     :   '+';
9  SUB     :   '-';
10 EQUALS  :   '=';
11 RPAR    :   '(';
12 LPAR    :   ')';
13 prog    :   stat+;
14 stat    :   expr NEWLINE # printExpr
15         |   ID EQUALS expr NEWLINE # assign
16         |   NEWLINE # blank
17         ;
18 expr    :   expr op=(MUL|DIV) expr # MulDiv
19         |   expr op=(ADD|SUB) expr # AddSub
20         |   INT # int
21         |   ID # id
22         |   RPAR expr LPAR # parens
23         ;
```

Der Lexer

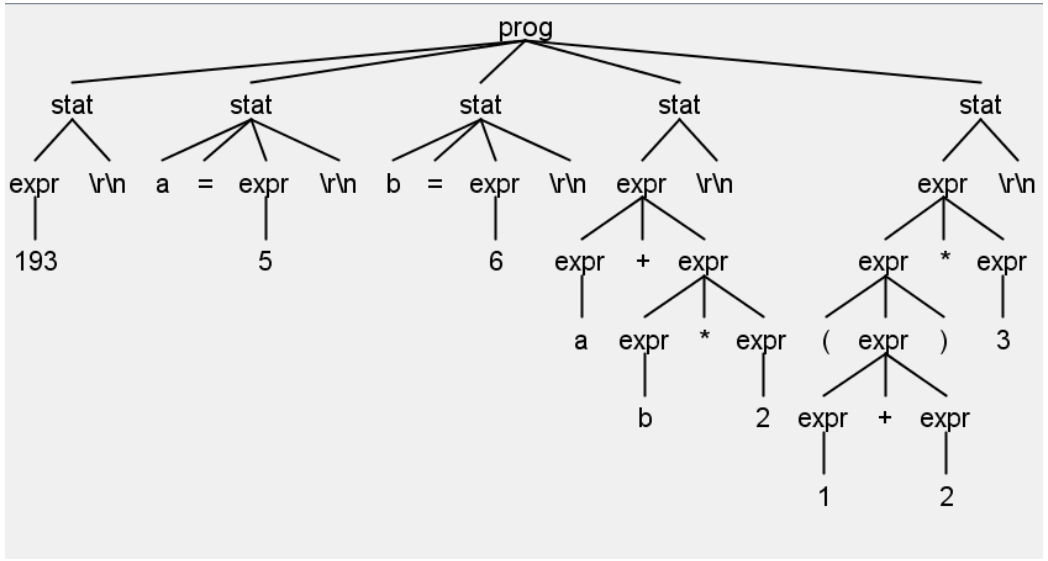
- Wird auch Scanner, Tokenizer oder *lexial Analyser* genannt.
- Der Lexer nimmt einen *Character Stream* als Eingabe und gruppiert die einzelnen Zeichen in bedeutungsvolle Einheiten die Lexeme genannt werden. Für jedes Lexem wird ein Token der Form *<token-name, token-value>* erzeugt.
- Rechts ist ein Beispiel aus ANTLR zu sehen, für die Grammatik aus Folie 8. Rechts steht die Eingabe und links die einzelnen Tokens in folgendem Format:

  [*@number* [*start..end*)=*value*,*<token-num=token-name>*]



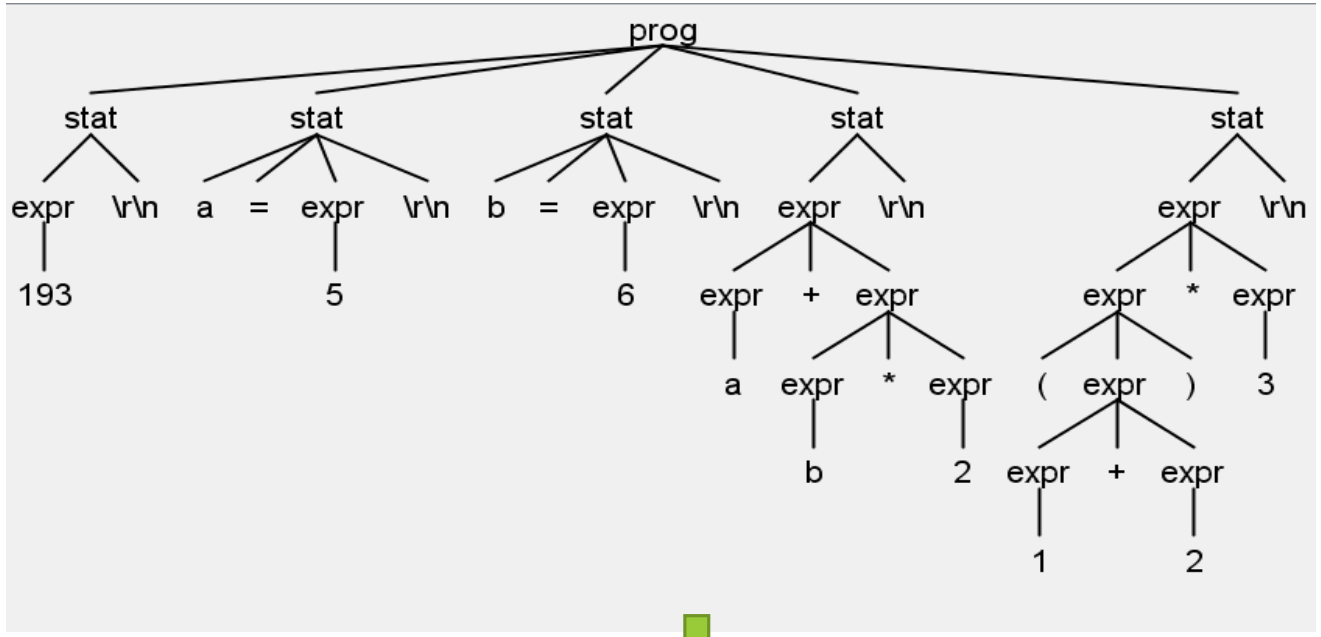
Die Syntaxanalyse *(Parsing)*

- Der Parser nimmt eine Folge von Tokens als Eingabe und erstellt daraus eine Baumartige Datenstruktur.
- Meistens wird erstmal ein sogenannter *Parse Tree* erstellt, in dem jeder Knoten einer Produktionsregel der Grammatik entspricht.
- Die Anfangsregel ist dann die Wurzel und die Tokens sind die Blätter des Baums.
- Rechts ist ein *Parse Tree* zu sehen, der aus den Tokens aus Folie 11 und der Grammatik aus Folie 10 gebaut wurde.



Parse Tree vs. Abstract Syntax Tree

Der *Parse Tree* enthält viele Informationen über die Produktionsregeln, die für einen Interpreter oder Compiler-Back-End nicht relevant sind. Deswegen werden *Parse Trees* in *Abstract Syntax Trees* umgewandelt, um den Output effizienter generieren zu können.



Semantische Analyse

- Oft können syntaktisch korrekte Ausdrücke auftauchen, die aber keinen Sinn ergeben.
- Ein klassisches Beispiel für semantische Analyse ist *Type Checking*.
- C ist *weakly typed*, deswegen gibt es nicht viel auf der semantischen Ebene zu prüfen, wenn man sich nur an die Sprachdefinition hält.
- Dennoch geben C Compilers zahlreiche Warnungen über mögliche semantische Fehler. Entwickler können den Compiler so einstellen, dass viele Warnungen als Fehler behandelt werden, um interne *Code Style Guidelines* einzuhalten.
- Für C und C++ gibt es aber zahlreiche *static analysis tools,* um semantische Fehler im Code zu erkennen. Manche von diesen Tools greifen auf Funktionalität des Compilers zurück, um ihre Arbeit zu erleichtern.

Intermediate Representation (IR)

- Verschiedene Compilers generieren eine oder mehrere IR in verschieden Phasen, um den Input auf syntaktische und semantische Korrektheit zu prüfen oder, um Optimierungen vorzunehmen.
- *Parse Trees* und *Abstract Syntax Trees* sind auch eine Form von IR, die hauptsächlich zur Analyse verwendet werden.
- Um die Codegenerierung und Optimierungen vorzunehmen werden aber andere Intermediate Representations benutzt die näher an dem Machine-Code sind.
- In GCC wird GENERIC und GLIMPSE verwendet. Clang erzeugt LLVM IR.
- Die Erzeugung von dieser *low-level* IR aus dem AST ist zwar eine Form der Synthese ist aber oft die letzte Aufgabe des Front-Ends.

GCC – Der GNU C Compiler - Geschichte

- 27.09 1983: Richard Stallman kündigt in der *Usenet Newsgroup* `net.unix-wizards,net.usoft` das GNU Projekt an unter dem Titel *„New UNIX implementation“.* Unter anderem wird ein neuer C Compiler versprochen. Hier ein Ausschnitt aus dem Originaltext:

  “To begin with, GNU will be a kernel plus all the utilities needed to write and run C programs: editor, shell, C compiler, linker, assembler, and a few other things. “

- 22.03 1987: Die erste Betaversion von GCC wird veröffentlicht. Hier ein Ausschnitt aus dem Buch ***„Free as in Freedom“*** über die Bedeutung von GCC in der Zeit:

  It wasn't until the release of the GNU C Compiler in 1987, however, that he (Michael Tiemann) began to grasp the full potential of free software. Dubbing GCC a "bombshell," Tiemann says the program's own existence underlined Stallman's determination as a programmer.

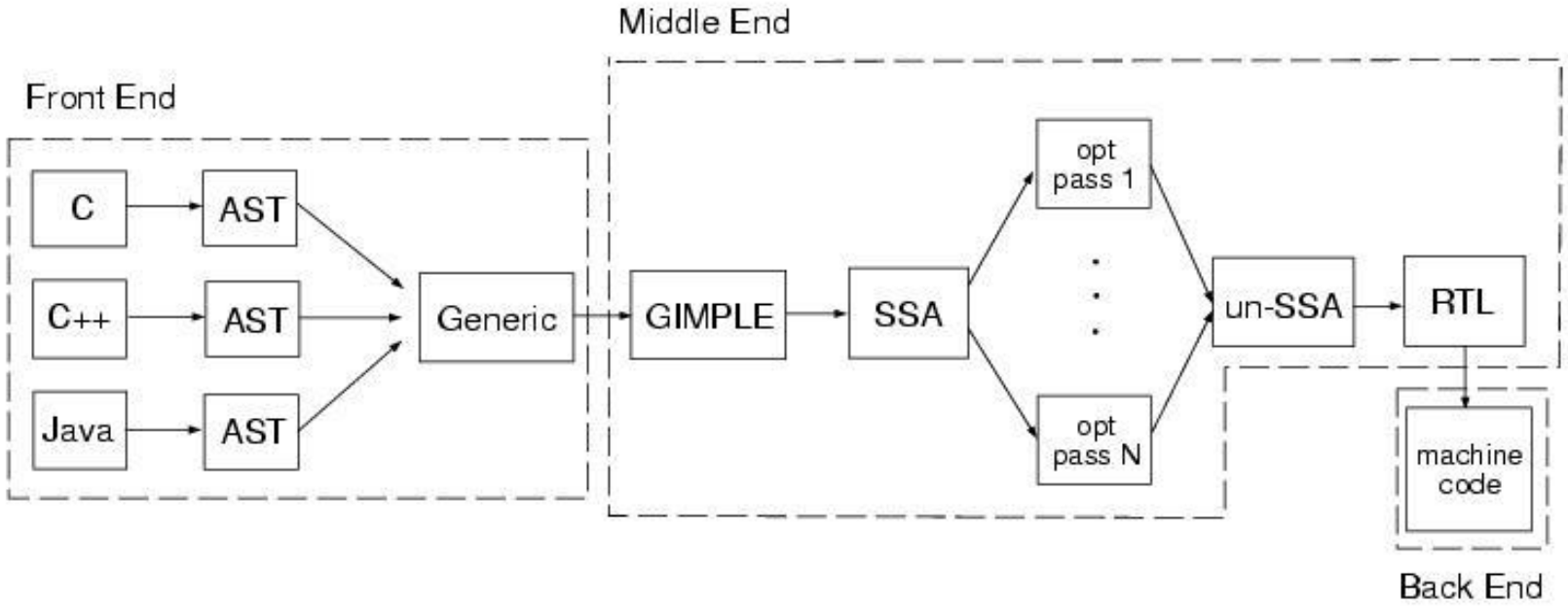
  "Just as every writer dreams of writing the great American novel, every programmer back in the 1980s talked about writing the great American compiler," Tiemman recalls. "Suddenly Stallman had done it. It was very humbling."

  "You talk about single points of failure, GCC was it," echoes (Keith) Bostic. "Nobody had a compiler back then, until GCC came along."

- 15.08.1997: Das EGCS Projekt wird gestartet um verschiedene nicht offizielle Erweiterungen von GCC zusammenzubringen.

- April 1999: GCC und EGCS werden wieder zusammengeführt. EGCS wird dann die neue offizielle Version von GCC.

GCC -Architektur



Clang/LLVM – Geschichte

- Das LLVM (damals *Low Level Virtual Machine*) wurde im Jahr 2000 an der *University of Illinois* unter der Leitung von Chris Lattner und Vikram Adve angefangen.
- Das Projekt wurde ursprünglich als Forschungsarbeit zur Untersuchung dynamischer Kompilierung und Optimierungen entwickelt.
- Im Jahr 2005 hat Apple Chris Lattner eingestellt und ein Team zusammengestellt, um LLVM weiterzuentwickeln. Ziel des Projects war LLVM in den internen Entwicklungstools zu integrieren.
- Im Jahr 2007 gibt Apple den Clang-Code unter einer BSD-ähnlichen Lizenz frei.
- Am 5. November 2012 ersetzt das FreeBSD Projekt GCC durch Clang als den *default* Compiler.

Clang/LLVM – Motivationen

- GCC galt nach dem Empfinden von vielen Entwicklern als zu kompliziert und schwer zu erweitern.
- Die Architektur von GCC ist nicht besonders gut geeignet, um *Code Analysis Tools*, *IDEs* und *Refactoring Tools* anzubinden. Clang wurde von Tag eins als eine API konzipiert.
- GCC ist ab der Version 4.2.1 nur unter der GPL v3.0 zu bekommen. Viele Projekte hatten Bedenken wegen der neuen Lizenz und mussten alte Versionen von GCC benutzen.
- Clang wurde als ein *drop-in* Ersatz für GCC Entwickelt.

Assembler Linker und Loader

- Der Assembler nimmt Assemblersprache (*Assembly Language*) als Eingabe und übersetzt die mnemonischen Symbole in Opcodes. Es muss unter Anderem auch symbolische Namen in konkrete Speicheradressen umwandeln und Marcos expandieren.
- Die Ausgabe von dem Assembler ist ein ***Object File.*** Das konkrete Format eines Object Files ist vom Betriebssystem und der Prozessorarchitektur abhängig. ELF (*Executable and Linkable Format*) ist das gängige Format unter Unix und Unix-ähnlichen Betriebssystemen.
- Die meisten Programme enthalten Bestandteile oder Module, die in anderen Programmen Verwendung finden können.
- Um ein Programmmodul in einem anderen Programm verwenden zu können, müssen die symbolischen Adressen der Funktionen und Variablen des Moduls in Speicheradressen umgewandelt werden. Diese Aufgabe übernimmt der Linker.
- Der Loader muss das fertige ausführbare Programm in den Arbeitsspeicher laden, relative in absolute Arbeitsspeicheradressen umrechnen und den Linker nochmal aufrufen bei dynamischem Linking.

Überblick

- Was ist ein Compiler
- Compilers vs. Interpreter
- Input und Output bei einem C Compiler
- GCC/Clang Crash Course
- Struktur eines Compilers
  - Grammatik
  - Lexer
  - Parser, Parse Tree, Abstract Syntax Tree
  - Semantische Analyse
- GCC: Geschichte, Motivation und Architektur
- Clang: Geschichte, Motivation und Architektur
- Assembler, Linker und Loader