

Compiler

Proseminar „C-Programmierung - Grundlagen und Konzepte“

Torsten Zühlke

`9zuehlke@informatik.uni-hamburg.de`

3. Juni 2011

- 1 Grundlagen
- 2 2-Phasen-Modell
 - Allgemeines
 - Analysephase
 - Synthesephase
- 3 Compilertypen

Gliederung

- 1 Grundlagen
- 2 2-Phasen-Modell
 - Allgemeines
 - Analysephase
 - Synthesephase
- 3 Compilertypen

Definition

Compiler¹

Ein Compiler (auch Übersetzer oder Kompilierer genannt) ist ein Computerprogramm, das eine in einer Quellsprache geschriebene Eingabe in eine semantisch äquivalente Ausgabe einer Zielsprache umwandelt.

¹Frei nach: Compiler. <http://de.wikipedia.org/wiki/Compiler> 2011-05-24

²Cooper & Torczon: Engineering a Compiler. Morgan Kaufmann 2003

Definition

Compiler¹

Ein Compiler (auch Übersetzer oder Kompilierer genannt) ist ein Computerprogramm, das eine in einer Quellsprache geschriebene Eingabe in eine semantisch äquivalente Ausgabe einer Zielsprache umwandelt.

Grundsätze²

The compiler must preserve the meaning of the program being compiled.

The compiler must improve the input program in some discernible way.

¹Frei nach: Compiler. <http://de.wikipedia.org/wiki/Compiler> 2011-05-24

²Cooper & Torczon: Engineering a Compiler. Morgan Kaufmann 2003

Definition

Compiler¹

Ein Compiler (auch Übersetzer oder Kompilierer genannt) ist ein Computerprogramm, das eine in einer Quellsprache geschriebene Eingabe in eine semantisch äquivalente Ausgabe einer Zielsprache umwandelt.

Grundsätze²

The compiler must preserve the meaning of the program being compiled.

The compiler must improve the input program in some discernible way.

Beispiele

- `tpic` überführt `pic`-Grafiken in \LaTeX
- `gcc` überführt Quellcode diverser Hochsprachen in Maschinencode

¹Frei nach: Compiler. <http://de.wikipedia.org/wiki/Compiler> 2011-05-24

²Cooper & Torczon: Engineering a Compiler. Morgan Kaufmann 2003

Motivation

Vorteile von Compilern

- Programmierung in Hochsprache möglich
 - Programmierung für mehrere Plattformen einfacher
- ⇒ Schnellere Softwareentwicklung

Motivation

Vorteile von Compilern

- Programmierung in Hochsprache möglich
 - Programmierung für mehrere Plattformen einfacher
- ⇒ Schnellere Softwareentwicklung
- Erkennt Fehler
 - Erzeugt bessere Programme
- ⇒ Zuverlässigere Software

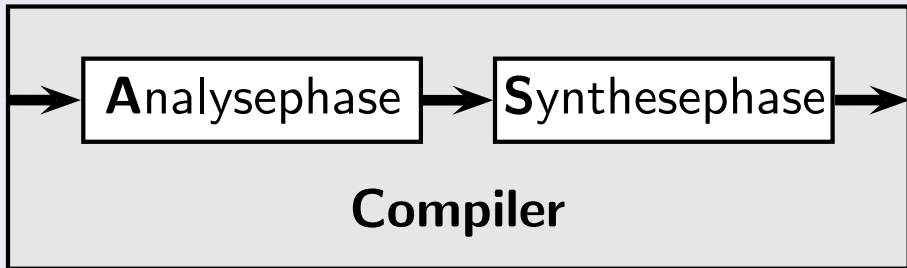
Gliederung

- 1 Grundlagen
- 2 2-Phasen-Modell
 - Allgemeines
 - Analysephase
 - Synthesephase
- 3 Compilertypen

Die Compilerphasen

Analyse- und Synthesephase

Compiler arbeiten üblicherweise in 2 Phasen, die weiter unterteilt sind.



- Die Analysephase prüft auf Korrektheit
- Die Synthesephase erzeugt und optimiert Maschinencode

Gliederung

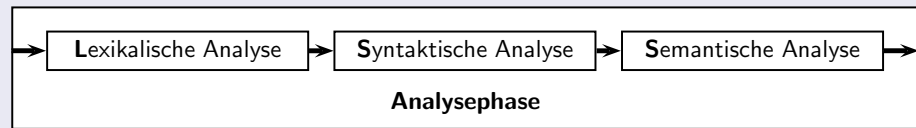
- 1 Grundlagen
- 2 2-Phasen-Modell
 - Allgemeines
 - **Analysephase**
 - Synthesephase
- 3 Compilertypen

Die Analysephase (Frontend)

Analyseschritte

Die Analysephase ist aufgeteilt in

- Lexikalische Analyse
- Syntaktische Analyse
- Semantische Analyse



Die Lexikalische Analyse

Aufgabe

Bei der Lexikalischen Analyse wird der Quellcode in Tokens umgewandelt.

Token

Token (engl.): Terminalsymbol \Leftrightarrow zusammengehörende Grundeinheit

Mögliche Tokens

- Schlüsselwörter
if, for, while, switch, return, ...
- Bezeichner
- Typen
int, float, double, char, ...
- Konstanten
Ganzzahlen, Fließkommazahlen, Zeichen, Zeichenketten
- Operatoren

Veranschaulichung I

C-Code

```
1 int main(int argc , char *argv [])
2 {
3   int *pointer , value ;
4   value = 0 ;
5   pointer = &value ;
6
7   for(int i = 0 ; i < 5 ; i++){
8     value += i ;
9     printf("%d\n" , *pointer ) ;
10  }
11
12  return *pointer ;
13 }
```

Veranschaulichung II

Tokens

```

<Typ, int>, <"main">, <"(">, <Typ, int>, <"argc">, <",">,
  <Typ, char>, <Op, "*">, <"argv">, <Op, []>, <"(">, <"{">
<Typ, int>, <Op, "*">, <"pointer">, <"Value">, <";">
<"value">, <Op, "=">, <Ganzzahl, 0>, <";">
<"pointer">, <Op, "=">, <Op, "&">, <"value">, <";">
<Schl.wort, "for">, <"(">
  <Typ, int>, <"i">, <Op, =>, <Ganzzahl, 0>, <";">
  <"i">, <Op, <>>, <Ganzzahl, 5>, <";">
  <"i">, <Op, ++>, <"(">, <"{">
<"value">, <Op, +=>, <"i">, <";">
<"printf">, <"(">, <String, "%d\n">, <"(">,
  <Op, *>, <"pointer">, <"(">, <";">, <"}">
<Schl.wort, "return">, <Op, *>, <"pointer">, <";">, <"}">

```

Die Syntaktische Analyse

Aufgaben

- Prüfung der Eingabe auf syntaktische Korrektheit
- Erstellen eines Syntaxbaums

Vergleich der Eingabe mit einer Definition der Eingabesprache.
Hierfür muss eine formale Definition der Eingabesprache vorliegen.

Exkurs - Sprachen & Grammatiken I

Formale Sprache

Eine formale Sprache L ist eine Menge von erlaubten Wörtern.

z.B. $L = \{w \mid w \text{ ist C-Programm}\}$

Exkurs - Sprachen & Grammatiken I

Formale Sprache

Eine formale Sprache L ist eine Menge von erlaubten Wörtern.

z.B. $L = \{w \mid w \text{ ist C-Programm}\}$

Formale Grammatik

Mathematisches Konstrukt zur Beschreibung formaler Sprachen.

Nichtterminale werden mittels Produktionsregeln zu Terminalen abgeleitet.

Terminalen sind (Teil-)Wörter.

Exkurs - Sprachen & Grammatiken I

Formale Sprache

Eine formale Sprache L ist eine Menge von erlaubten Wörtern.

z.B. $L = \{w \mid w \text{ ist C-Programm}\}$

Formale Grammatik

Mathematisches Konstrukt zur Beschreibung formaler Sprachen.

Nichtterminale werden mittels Produktionsregeln zu Terminalen abgeleitet.

Terminalen sind (Teil-)Wörter.

Chomsky-Hierarchie

Die Chomsky-Hierarchie klassifiziert Grammatiken nach Produktionsregeln:

- Typ 0 Unbeschränkte Grammatik
- Typ 1 Kontextsensitive Grammatik
- Typ 2 Kontextfreie Grammatik
- Typ 3 Reguläre Grammatik

Exkurs - Sprachen & Grammatiken II

Erweiterte Backus-Naur-Form (EBNF)

Metasprache, die kontextfreie Grammatiken mittels ihrer Regeln darstellt.

- Menschenlesbar
- Maschinenlesbar

EBNF-Beispiel¹

```
ZifferAußerNull = "1" | "2" | "3" | "4" | "5"  
                | "6" | "7" | "8" | "9" ;  
Ziffer          = "0" | ZifferAußerNull ;  
NatuerlicheZahl = ZifferAußerNull { Ziffer } ;  
GanzeZahl      = "0" | [ "-" ] NatuerlicheZahl ;
```

¹Erweiterte Backus-Naur-Form.

C-Grammatik I

C-Syntax (Auszug, vereinfacht)¹

```
translation_unit    = function_definition;
function_definition = type_specifier IDENTIFIER declaration_list
                    compound_statement;
type_specifier      = VOID | INT | CHAR;
declaration_list    = declaration | declaration_list declaration;
declaration         = type_specifier init_declarator_list;
init_declarator_list = init_declarator
                    | init_declarator_list ',' init_declarator;
init_declarator     = declarator | declarator = initializer;
declarator          = ['*'] IDENTIFIER;
initializer         = ['&'] IDENTIFIER | CONSTANT;
compound_statement  = '{' statement_list '}'
                    | '{' declaration_list statement_list '}';
```

¹ANSI C Grammar(Yacc).

C-Grammatik II

C-Codebeispiel

```
1 int main(int argc , char *argv [])
2 {
3   int *pointer , value ;
4   value = 0 ;
5   pointer = &value ;
6
7   for(int i = 0 ; i < 5 ; i++){
8     value += i ;
9     printf ("%d\n" , *pointer ) ;
10  }
11
12  return *pointer ;
13 }
```

Semantische Analyse I

Semantische Analyse

Der Compiler sammelt Informationen über Programmdetails:

- Eigenschaften benutzter Werte/Typen
- Art der Bereitstellung und Initialisierung von Werten
- Vorhaltezeit für Werte
- Art der Speicherfreigabe
- Überladene Funktionen
- Zulässigkeit von Zuweisungen

Semantische Analyse II

Deklaration

Informationen über Variablen werden gewonnen aus

- Deklarationen
- Kontext der Nutzung

Vorwärtsdeklaration

Deklaration eines Bezeichners steht im Quellcode nach der ersten Nutzung.

Auflösen von Vorwärtsdeklarationen:

⇒ Semantische Analyse mehrfach ausführen

Gliederung

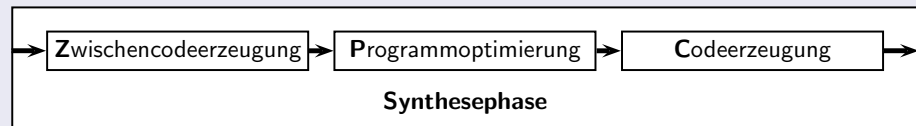
- 1 Grundlagen
- 2 2-Phasen-Modell**
 - Allgemeines
 - Analysephase
 - Synthesephase**
- 3 Compilertypen

Die Synthesephase (Backend)

Syntheseschritte

Die Synthesephase ist aufgeteilt in

- Zwischencodeerzeugung
- Programmoptimierung
- Codeerzeugung



Die Zwischencodeerzeugung I

Aufgabe

Die Eingabe wird in eine systemnahen Zwischencode übersetzt. Dieser Zwischencode hat folgende Eigenschaften:

- Verwendet Prozessorbefehle
- Verwendet Speicher unbegrenzter Größe
- Kann unbegrenzt viele Register adressieren

Die Zwischencodeerzeugung II

ILOC¹

$$w \leftarrow x \times 2 \times y \times z$$

loadAI	$r_{arp}, @w$	$\Rightarrow r_w$	// load w
loadI	2	$\Rightarrow r_2$	// constant 2 into r_2
loadAI	$r_{arp}, @x$	$\Rightarrow r_x$	// load x
loadAI	$r_{arp}, @y$	$\Rightarrow r_y$	// load y
loadAI	$r_{arp}, @z$	$\Rightarrow r_z$	// load z
mult	r_w, r_2	$\Rightarrow r_w$	// $r_w \leftarrow w \times 2$
mult	r_w, r_x	$\Rightarrow r_w$	// $r_w \leftarrow (w \times 2) \times x$
mult	r_w, r_y	$\Rightarrow r_w$	// $r_w \leftarrow (w \times 2 \times x) \times y$
mult	r_w, r_z	$\Rightarrow r_w$	// $r_w \leftarrow (w \times 2 \times x \times y) \times z$
storeAI	r_w	$\Rightarrow r_{arp}, 0$	// write r_w back to w

¹Cooper & Torczon: Engineering a Compiler. Morgan Kaufmann 2003

Optimierung

Optimierung

Bei der Optimierung wird der Quellcode in Bezug auf seinen Ressourcenbedarf verbessert.

Minimierung von z.B.

- Speicherzugriffe
- Speicherbedarf
- Befehlszahl
- Wartezeiten

Optimierungen

Einige mögliche Optimierungen

- Schleifen abwickeln
- Toten Code eliminieren
- Neuordnung von Befehlen
- Statische FormelAuswertung
- Erkennung von Konstanten
- Einfügen von Unterprogrammen
- Verwendung schnellerer Anweisungen
- Paging verhindern

Optimierungsprobleme

It (the compiler) will almost always fail to produce optimal code.¹

¹Cooper & Torczon: Engineering a Compiler. Morgan Kaufmann 2003

Optimierungsprobleme

It (the compiler) will almost always fail to produce optimal code.¹

Optimierung vs. Optimierung

Eine Optimierung einer Eigenschaft kann negative Auswirkungen auf andere Eigenschaften haben.

Speicherzugriffe \Leftrightarrow Befehlszahl
Speicherzugriffe \Leftrightarrow Speicherbedarf

¹Cooper & Torczon: Engineering a Compiler. Morgan Kaufmann 2003

Die Codeerzeugung

Aufgabe

Entscheidung über die Zuweisung von Registern und Speicherstellen

Die Codeerzeugung

Aufgabe

Entscheidung über die Zuweisung von Registern und Speicherstellen

Abhängig von:

- Art und Zahl der Register
- Verwendung der Variablen
- Inkonsistenzen bei der Nutzung von Registern

Veranschaulichung I

C-Codebeispiel

```
1 int main(int argc , char *argv [])
2 {
3   int *pointer , value ;
4   value = 0 ;
5   pointer = &value ;
6
7   for(int i = 0 ; i < 5 ; i++){
8     value += i ;
9     printf ("%d\n" , *pointer ) ;
10  }
11
12  return *pointer ;
13 }
```

Veranschaulichung II

Ausgabe bei Ausführung des Programms

0

1

3

6

10

Rückgabe: 10

Veranschaulichung III

Assemblercode (Initialisierung)

```
main:
.LFB0:
    pushq  %rbp                //Stackframe setzen
    movq   %rsp, %rbp          // I
    subq   $32, %rsp           // I
    movl   $0, -20(%rbp)       //value = 0
    leaq   -20(%rbp), %rax      //pointer = &value
    movq   %rax, -16(%rbp)     //pointer auf Stack Speichern
    movl   $0, -4(%rbp)        //i = 0 (for)
    jmp    .L2                 // -> Schleifenbedingung
```

Veranschaulichung IV

Assemblercode (Schleifenrumpf)

```
.L3:                                //Schleifenrumpf
    movl   -20(%rbp), %eax           //value laden
    addl   -4(%rbp), %eax           //value += i
    movl   %eax, -20(%rbp)         //value zurückspeichern
    movq   -16(%rbp), %rax         //printf("%d\n", *pointer)
    movl   (%rax), %eax            // I
    movl   %eax, %esi              // I
    movl   $.LC0, %edi             // I
    movl   $0, %eax                // I
    call  printf                    // I
    addl   $1, -4(%rbp)            // i++
```

Veranschaulichung V

Assemblercode (Schleifenbedingung & Ende)

```
.L2:                                //Schleifenbedingung
    cmpl    $4,          -4(%rbp)    //i < 5 (for)
    jle     .L3           // -> Schleifenrumpf
    movq    -16(%rbp), %rax          //return *pointer
    movl    (%rax),      %eax        // I
    leave                               //Stack aufräumen
    ret                               //Funktion verlassen
```

Veranschaulichung VI

Assemblercode (Optimiert)

```
main:
.LFB0:
    subq    $8,        %rsp        //Stackframe setzen
    xorl    %esi,      %esi        //printf("0\n")
    movl    $.LC0,     %edi        // I
    xorl    %eax,      %eax        // I
    call   printf      // I
    movl    $1,        %esi        //printf("1\n")
    movl    $.LC0,     %edi        // I
    xorl    %eax,      %eax        // I
    call   printf      // I
    ...
```


Veranschaulichung VII

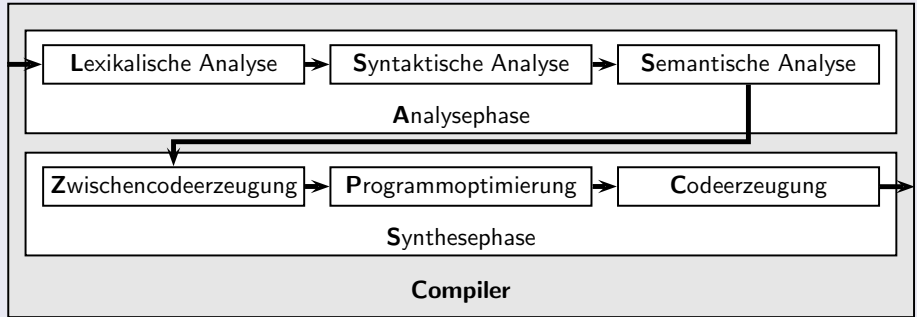
Assemblercode (Optimiert)

...

```
movl    $10,      %esi    //printf("10\n")
movl    $.LC0,    %edi    // I
xorl    %eax,     %eax    // I
call    printf    // I
movl    $10,      %eax    //return 10
addq    $8,       %rsp    //Stack aufräumen
ret     //Funktion verlassen
```

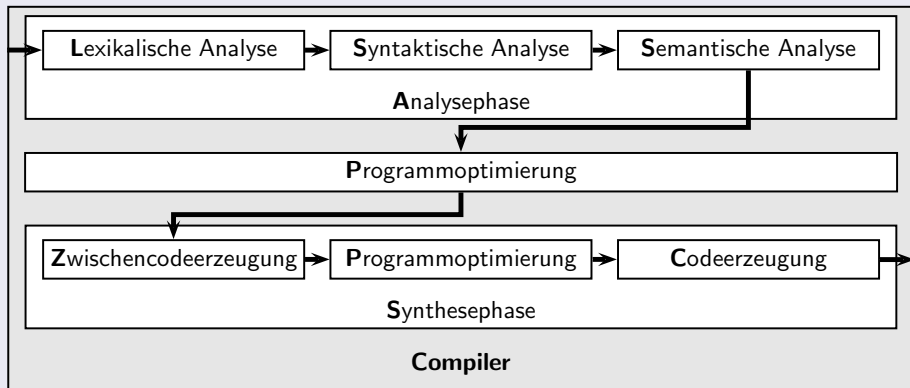
Überblick

Compilerphasen



Alternative Implementation

Compilerphasen



Gliederung

- 1 Grundlagen
- 2 2-Phasen-Modell
 - Allgemeines
 - Analysephase
 - Synthesephase
- 3 **Compilertypen**

Compilerformen I

Compilerformen

Es gibt verschiedene Compilerformen:

- **Compiler**
Ausführung und Kompilation unabhängig
- **Interpreter**
Quellcode wird zur Ausführungszeit interpretiert
- **Compreter**
Vorher erzeugter Zwischencode wird zur Ausführungszeit interpretiert
- **Just-in-time-compiler (JIT-Compiler)**
Quellcode wird direkt vor der Ausführung kompiliert
Teilweise wird unabhängig von der Ausführung Zwischencode erstellt

Compilerformen II

Gegenüberstellung

	Analysephase			Synthesephase		
	Lex. A	Syn. A	Sem. A	Zw.Code	Opt.	Code
Compiler	X	X	X	X	X	X
Compreter	X	X	X	X	X	I
Interpreter	I	I	(I)	-	-	I
JIT-Compiler	X / J	X / J	X / J	X / J	X / J	J

X: im Voraus, I: interpretiert, J: bedarfsorientiert (just in time),
 (): implementationsabhängig

Compilertypen

Native Compiler

Erzeugt Programmcode für die Plattform auf der er selbst läuft

Cross-Compiler

Erzeugt Programmcode für eine andere Plattform

Transcompiler (Transpiler)

Übersetzt aus einer Hochsprache in eine andere

Compilervarianten

Single-pass-Compiler

Erzeugt den Zielcode in einem Compilerdurchlauf

Multi-pass-Compiler

Erzeugt den Zielcode in mehreren Schritten

Zusammenfassung

1 Grundlagen

Definition, Motivation

2 2-Phasen-Modell

- Allgemeines

- Analysephase

Lexikalische -, Syntaktische -, Semantische Analyse

Tokens, Sprachen, Grammatiken, Chomsky-Hierarchie, EBNF

- Synthesephase

Zwischencodeerzeugung, Optimierung, Codeerzeugung

ILOC

Alternative Implementation mit 3 Phasen

3 Compilertypen

Compiler, Interpreter, Compreter, Just-in-Time-Compiler

Native Compiler, Cross-Compiler, Transcompiler

Single-Pass-Compiler, Multi-Pass-Compiler

Anhang: Sprachen & Grammatiken I

Alphabete & Formale Sprachen

- Ein Alphabet Σ ist eine endliche Symbolmenge
- Eine formale Sprache L ist eine Menge von Wörtern

Sprechweise

L ist Sprache über Σ :

Alle Wörter aus L bestehen nur aus Symbolen aus Σ .

Beispiel

$$\Sigma = \{0, 1\}$$

$$L = \{(ab)^+ \mid a, b \in \Sigma, a \neq b\}$$

$$L = \{01, 10, 0101, 1010, 010101, 101010, \dots\}$$

Anhang: Sprachen & Grammatiken II

Formale Grammatik

Eine formale Grammatik G ist ein mathematisches Konstrukt zur Beschreibung und Erzeugung formaler Sprachen mittels Produktionsregeln.

- Σ , einem endlichen Alphabet aus Terminalsymbolen
- N , einem endlichen Alphabet von Nichtterminalsymbolen
- P , einer endlichen Menge von Produktionsregeln
 - Jede Regel ist eine Relation $A \rightarrow w$ mit $A, w \in (\Sigma \cup N)^*$
 - A enthält mindestens ein Nichtterminalsymbol
- $S \in N$, einem Startsymbol

Formale Grammatik

Mit Formalen Grammatiken können Formale Sprachen beschrieben und erzeugt werden.

Anhang: Sprachen & Grammatiken III

Beispiel

Sei G folgenden Formale Grammatik: $G = (\Sigma, N, P, S)$

$$\Sigma = \{0, 1\}$$

$$N = \{A, B, S\}$$

P sei die Menge folgender Regeln:

Alternative 1

$$S \rightarrow A \mid B$$

$$A \rightarrow 0B1 \mid 01$$

$$B \rightarrow 1A0 \mid 10$$

Alternative 2

$$S \rightarrow 0A1 \mid 1A0 \mid 01 \mid 10$$

$$0A1 \rightarrow 01A01 \mid 0101$$

$$1A0 \rightarrow 10A10 \mid 1010$$

L sei die aus G erzeugte Sprache:

$$L = \{01, 10, 0101, 1010, 010101, 101010, \dots\}$$

Anhang: Sprachen & Grammatiken IV

Chomsky-Hierarchie

Die Chomsky-Hierarchie klassifiziert Grammatiken nach Produktionsregeln:

Typ 0 Unbeschränkte Grammatik

Typ 1 Kontextsensitive Grammatik

Produktionsregeln haben die Form $\alpha A \beta \rightarrow \alpha \gamma \beta$

Typ 2 Kontextfreie Grammatik

Produktionsregeln haben die Form $A \rightarrow \alpha$

Typ 3 Reguläre Grammatik

Produktionsregeln haben die Form $A \rightarrow \epsilon$ oder

$A \rightarrow aB$ (rechtsregulär) bzw. $A \rightarrow Ba$ (linksregulär)

A, B : Nichtterminal, a : Terminal

α, β, γ : Wörter aus Terminalen und Nichtterminalen, γ ist nicht leer

ϵ : leeres Wort

Anhang: Sprachen & Grammatiken V

Erweiterte Backus-Naur-Form (EBNF)

Die EBNF stellt kontextfreie (Typ-2-)Grammatiken mittels ihrer Produktionsregeln dar.

EBNF-Beispiel¹

```
ZifferAußerNull = "1" | "2" | "3" | "4" | "5"
                  | "6" | "7" | "8" | "9" ;
Ziffer           = "0" | ZifferAußerNull ;
NatuerlicheZahl = ZifferAußerNull { Ziffer } ;
GanzeZahl       = "0" | [ "-" ] NatuerlicheZahl ;
```

¹Erweiterte Backus-Naur-Form.

Quellen I



ANSI C Grammar(Yacc).

<http://www.lysator.liu.se/c/ANSI-C-grammar-y.html>.
2011-05-26.



Compiler.

<http://de.wikipedia.org/wiki/Compiler>.
2011-05-24.



Erweiterte Backus-Naur-Form.

http://de.wikipedia.org/wiki/Erweiterte_Backus-Naur-Form.
2011-05-26.



Formal grammar.

http://en.wikipedia.org/wiki/Formal_grammar.
2011-05-26.

Quellen II



Formale Grammatik.

http://de.wikipedia.org/wiki/Formale_Grammatik.
2011-05-24.



Formale Sprache.

http://de.wikipedia.org/wiki/Formale_Sprache.
2011-05-24.



GNU compiler collection.


http://en.wikipedia.org/wiki/GNU_Compiler_Collection.
2011-05-24.



Keith Cooper and Linda Torczon.

Engineering a Compiler: International Student Edition.
Morgan Kaufmann, I.S.ed edition, 12 2003.

Quellen III

-  Christoph Habel and Matthias Jantzen.
FGI-1: Formale Grundlagen der Informatik.
Universität Hamburg, SoSe 2010.