

# Beschreibung von Programmiersprachen

Bisher ist die Sprache C und C++ nur informal definiert und an hand von Beispielen und informalen Regeln erklärt.

Zur exakten Definition von Programmiersprachen werden Formalismen gebraucht, von denen einige hier vorgestellt werden.

## Inhalt

1.	Grundlagen .....	2
1.1.	Strings, Sprachen.....	2
1.2.	Sprachen und Grammatiken.....	5
2.	Syntax .....	14
2.1.	EBNF .....	14
2.1.1.	Allgemeine Form .....	14
2.1.2.	Syntax von C++ .....	17
3.	Semantik .....	59

# 1. Grundlagen

## 1.1. Strings, Sprachen

Zunächst betrachten wir Zeichenketten (Strings) und Sprachen.

### Definition 1-1

Ein **Alphabet** ist eine endliche, nicht leere Menge.

Die Elemente eines Alphabetes heißen Zeichen oder Symbole. Sie sind unteilbare Einheiten.

Sei  $\Sigma$  ein Alphabet. Eine endliche Folge von Symbolen  $a_1a_2\dots a_n$  mit  $a_i \in \Sigma$  ( $1 \leq i \leq n$ ) heißt **Wort** oder **String** (über  $\Sigma$ ).

Die **Länge** eines Strings  $w = a_1a_2\dots a_n$  wird mit  $|w|$  notiert und ist als Anzahl der Symbole von  $w$  definiert.

### Beispiel 1-1

Sei  $\Sigma = \{a, b\}$

$w = aaabbb$  ist ein Wort über  $\Sigma$ .

$w = aaacccbbb$  ist kein Wort über  $\Sigma$ .

Sei  $w = a_1a_2\dots a_n$  ein Wort über  $\Sigma$  dann ist  $|w| = n$ .

## Notation 1-1

Bei Worten ist es sinnvoll, ein spezielles Wort mit Länge 0 besonders zu notieren. Dieses **leere Wort** wird mit  $\epsilon$  (Epsilon) bezeichnet, d.h.  $|\epsilon| = 0$ .

Wenn  $\Sigma$  ein Alphabet ist, dann bezeichnet  $\Sigma^*$  die Menge aller Wörter über  $\Sigma$  inklusive  $\epsilon$ .

$\Sigma^+$  ist die Menge aller Wörter über  $\Sigma$  ohne das leere Wort, also:  $\Sigma^+ = \Sigma^* - \{\epsilon\}$ .

## Beispiel 1-2

Sei  $\Sigma = \{a, b\}$ .

Dann ist  $\Sigma^* = \{\epsilon, a, b, aa, ab, ac, ba, bb, bc, \dots, cc, aaa, \dots\}$

## Definition 1-2

Seien  $x = x_1x_2\dots x_n$ ,  $y = y_1y_2\dots y_m \in \Sigma^*$ .

Die Konkatenation „.“ ist definiert als binäre Relation auf  $\Sigma^*$ , so dass  $x.y = x_1x_2\dots x_ny_1y_2\dots y_m$ .

Der Konkatenationsoperator „.“ wird häufig einfach weggelassen, d.h.  $xy = x.y$ .

Die Struktur  $(\Sigma^*, .)$  ist ein Monoid mit neutralem Element  $\epsilon$ .

Erinnerung:

$(A,+)$  ist ein **Monoid**, falls gilt:

1. **Abgeschlossenheit**:  $\forall a,b \in A: a+b \in A$

2. **Assoziativität**:  $\forall a,b,c \in A: (a+b)+c = a+(b+c)$

3. **neutrales Element**:  $\exists e \in A: \forall a \in A (a+e = e+a = a)$

(1,2=Halbgruppe, 1,2,3=Monoid)

Zur exakten Definition von  $\Sigma^*$  benötigt man noch Operationen auf Mengen.

### Definition 1-3

Seien  $X$  und  $Y$  Mengen. Das **Produkt** von  $X$  und  $Y$  ist definiert als:  $XY = \{x.y \mid x \in X, y \in Y\}$

Die **Exponentiation** ist somit definiert als:

- $X^0 = \{\epsilon\}$
- $X^1 = X$
- $X^{n+1} = X^n X$

Der **Kleene'sche Abschluss** der Menge  $X \subseteq \Sigma$  ist definiert als:  $X^* = \bigcup_{i \geq 0} X^i$ .

Der (Halbgruppen) Abschluss der Menge  $X \subseteq \Sigma$  ist definiert als:  $X^+ = \bigcup_{i \geq 1} X^i$ .

## Notation 1-2

Sei  $w \in \Sigma^*$ , dann ist  $w^n = \underbrace{www\dots w}_n$ .

## Definition 1-4

Sei  $\Sigma$  ein Alphabet und  $L \subseteq \Sigma^*$ .  $L$  heißt (**formale**) **Sprache** über  $\Sigma$ .

Damit ist nun formal beschrieben, was unter einer Sprache zu verstehen ist.

## 1.2. Sprachen und Grammatiken

**Sprachen** sind i.a. **unendliche Mengen**. Damit sie algorithmisch behandelbar sind (z.B. geprüft werden kann, ob ein String ein korrektes C++ Programm ist) ist ein **endliches Beschreibungsmittel** für sie erforderlich.

Eine **Lösung** sind **Grammatiken** als endliche Mechanismen zur Erzeugung von Wort-Mengen. Eine andere Lösung sind Automaten. Wir behandeln kurz Grammatiken. Ausführlicher wird dies und das Konzept der Automaten in der Veranstaltung „Theoretische Informatik“ behandelt.

### Definition 1-5

Eine (allgemeine Regel-) Grammatik ist ein 4-Tupel  $G = (V, \Sigma, P, S)$  mit:

- $V$  ist eine endliche Menge von Nichtterminalen
- $\Sigma$  ist das Terminalalphabet
- $\Sigma \cap V = \{\}$
- $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$  ist eine endliche Menge von Regeln (Produktionen)
- $S \in V$  ist das Startsymbol

### Konvention

Für  $(a, b) \in P$  wird  $a \rightarrow b \in P$  oder einfach  $a \rightarrow b$  geschrieben.

## Beispiel 1-3

$G_1 = ($   
    { Satz, Subjekt, Objekt, Prädikat, Artikel, Attributphrase, Attribut, Substantiv },  
    {  $\epsilon$ , der, die, das, kleine, müde, große, Philipp, Hanna, streichelt, ".", ",", " },  
    { P },  
    Satz  
)

mit

P :

Satz  $\rightarrow$  Subjekt Prädikat Objekt.

Subjekt  $\rightarrow$  Artikel Attributphrase Substantiv

Artikel  $\rightarrow \epsilon$

Artikel  $\rightarrow$  der

Artikel  $\rightarrow$  die

Substantiv  $\rightarrow$  Philipp

Substantiv  $\rightarrow$  Hanna

Prädikat  $\rightarrow$  streichelt

Objekt  $\rightarrow$  Artikel Attributphrase Substantiv

## Beispiel 1-4

$G_2 = (\{E, T, F\}, \{(\cdot), +, *, i\}, \{P\}, E)$

mit

P :

$E \rightarrow T$

$E \rightarrow E + T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow i$

## Beispiel 1-5

$G_3 = ( \{S, B, C\}, \{a, b, c\}, \{P\}, S )$

mit

P :

$S \rightarrow aSBC$

$S \rightarrow aBC$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

Um zu zeigen, welche Sprache von einer Grammatik erzeugt wird, sind einige Relationen zu definieren.

## Definition 1-6

Sei  $\Rightarrow$  eine Relation auf der Menge  $A$  (d.h.  $\Rightarrow \subseteq (A \times A)$ ).

Die reflexive transitive Hülle, i.Z.  $\Rightarrow^*$  ist die kleinste Relation, die  $\Rightarrow$  enthält und reflexiv und transitiv ist, d.h.

$$x \Rightarrow^* y \text{ gdw. } \exists x_1, x_2, \dots, x_n \in A \text{ mit} \\ x_1 = x, x_i \Rightarrow x_{i+1} \ (1 \leq i \leq n), x_n = y$$

## Definition 1-7

Sei  $G = (V, \Sigma, P, S)$  eine Grammatik und  $u, v \in (V \cup \Sigma)^*$ .

Die Relation **direkte Ableitung**, i.Z.  $\Rightarrow_G$  ist definiert als:

$$u \Rightarrow_G v, \text{ falls} \\ u = u_1 u_2 u_3 \text{ und } v = u_1 v_2 u_3 \\ \text{und } u_2 \rightarrow u_2 \text{ ist eine Produktion aus } P.$$

Die Relation **Ableitung**, i.Z.  $\Rightarrow_G^*$  ist die reflexive, transitive Hülle der Relation direkte Ableitung, d.h. in 0 oder mehreren Schritten ableitbar.

## Beispiel 1-6

Betrachten wir  $G_2$  aus dem letzten Beispiel.

$$\begin{aligned} E &\Rightarrow_{G_2} E + T \Rightarrow_{G_2} T + T \Rightarrow_{G_2} T * F + T \Rightarrow_{G_2} F * F + T \Rightarrow_{G_2} \dots \\ &\Rightarrow_{G_2} (E + t) * F + T \Rightarrow_{G_2} \dots \Rightarrow_{G_2} (i + i) * i + i \end{aligned}$$

Betrachten wir  $G_1$  aus dem letzten Beispiel.

*Satz*  $\Rightarrow_{G_1}^*$  der große Philipp streichelt die kleine, müde Hanna.

Betrachten wir  $G_3$  aus dem letzten Beispiel.

$$\begin{aligned} S &\Rightarrow_{G_3} aSBC \Rightarrow_{G_3} aaSBCBC \Rightarrow_{G_3} aaaBCBCBC \Rightarrow_{G_3} aaabCBCBC \Rightarrow_{G_3} \dots \\ &\Rightarrow_{G_3} a^3b^3c^3 \end{aligned}$$

Nun können wir definieren welche Sprache eine Grammatik definiert.

## Definition 1-8

Sei  $G = (V, \Sigma, P, S)$  eine Grammatik.

Die von **G erzeugte Sprache**, i.Z.  $L(G)$  ist definiert als:

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$$

also die Menge aller vom Startsymbol aus ableitbaren Terminalwörter.

## Beispiel 1-7

Betrachten wir  $G_3$  aus dem letzten Beispiel.

$$S \Rightarrow_{G_3}^* a^3 b^3 c^3$$

$$S \Rightarrow_{G_3}^* a^4 b^4 c^4$$

...

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}$$

Das **Wortproblem** ist ein bekanntes Problem:

Gegeben ist eine Grammatik  $G = (V, \Sigma, P, S)$  eine Grammatik und ein Wort  $w \in L(G)$ .

Ist  $w$  ein Wort der Sprache, d.h. gilt  $w \in L(G)$ ?

Dieses Problem ist für allgemeine Regelsprachen nicht entscheidbar. Deshalb werden verschiedene Restriktionen an Grammatiken gelegt.

Die Syntax von Programmiersprachen lässt sich durch kontextfreie Grammatiken beschreiben:

### **Definition 1-9**

Sei  $G = (V, \Sigma, P, S)$  eine Grammatik.

$G$  ist eine **kontextfreie Grammatik**, wenn

für alle  $w_1 \rightarrow w_2 \in P$  gilt :

$|w_1| \leq |w_2|$  und

$w_1 \in V$

also wird  $w_1$  ohne Berücksichtigung des Kontextes, in dem es in einer Ableitung auftaucht, durch  $w_2$  ersetzt.

## 2. Syntax

Die Syntax der Sprache C++ kann ganz formal als kontextfreie Grammatik angegeben werden. Dazu hat sich die Erweiterte Backus Naur Form (EBNF) etabliert.

Man kann aber auch eine graphische Darstellung verwenden. Dazu sind Syntaxdiagramme geeignet.

### 2.1. EBNF

#### 2.1.1. Allgemeine Form

##### Produktionen

Eine EBNF besteht aus einer Liste von Produktionen. Jede Produktion beschreibt die Syntax eines bestimmten Grammatikfragmentes. Produktionen werden als eine Art Gleichungen geschrieben. Auf der linken Seite steht ein Name für das definierte Grammatikfragment; auf der rechten Seite steht eine Folge von Symbolen, die den Aufbau des Grammatikfragmentes festlegen. Zwischen linker und rechter Seite wird das Trennzeichen  $::=$  gesetzt. Das Ende einer Produktion wird mit einem Punkt gekennzeichnet, also schematisch:

*linke Seite ::= rechte Seite .*

##### Terminale und Nichtterminale

Von den Symbolen, die auf der rechten Seite einer Produktion vorkommen, gibt es zwei Arten:

- Die *Terminale* stehen für sich selbst, sie sind wörtlich zu nehmen.

- Die *Nichtterminale* benennen eine andere Produktion, die an dieser Stelle einzusetzen ist.

Zur Unterscheidung werden Terminale oft in Gänsefüßchen gesetzt, Nichtterminale unterstrichen oder einfach überhaupt nicht markiert.

## Optionale Symbolfolgen

Um eine Reihe von Symbolen auf der rechten Seite einer Produktion als optional zu kennzeichnen, wird diese in eckige Klammern gesetzt.

## Wiederholbare Symbolfolgen

Eine Symbolfolge wird in geschweifte Klammern gesetzt, wenn sie beliebig oft wiederholt werden darf. Das schließt Weglassen, d.h. null-maliges Wiederholen, mit ein. Das Beispiel für Syntaxdiagramme als EBNF:

```
integer ::= [sign] digit {digit}.
```

In diesem Zusammenhang ist eine alternative Notation üblich, die in der EBNF eigentlich nicht vorgesehen ist: Hinter die geschweifte Klammer kann auch ein Plus-Zeichen gesetzt werden, um ein- oder mehrmalige Wiederholung auszudrücken. Das obige Beispiel würde dann kürzer lauten:

```
integer ::= [sign] {digit}+.
```

Um die ursprüngliche, beliebige Wiederholung davon klar abzuheben, wird diese dann ausdrücklich mit einem Stern nach der geschweiften Klammer markiert. Die folgende Schreibweise ist umständlicher, aber inhaltlich gleichwertig mit der vorhergehenden:

```
integer ::= [sign] digit {digit}*.
```

## Alternativen

Wenn mehrere Symbolfolgen auf der rechten Seite einer Produktion zur Auswahl stehen, werden die Möglichkeiten nacheinander aufgeführt und mit senkrechten Strichen getrennt. Das folgende Beispiel zeigt die komplette EBNF für eine `integer`-Konstante in "C"-Syntax:

```
integer ::= [sign] digit {digit}.
sign ::= "+" | "-".
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

## Gruppierung

Symbolfolgen können mit einfachen runden Klammern gruppiert werden, um Alternativen einzugrenzen. Auch die anderen Klammer-Arten (eckige und geschweifte) können so benutzt werden. Das folgende Beispiel ist zwar ungeschickt, illustriert aber diese Möglichkeit:

```
integer ::= [sign]
  ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9")
  {digit}.
sign ::= "+" | "-".
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

## Metasyntax

Die EBNF folgt einer eigenen Syntax, die mit der durch die EBNF beschriebenen Syntax nichts zu tun hat. Man nennt die EBNF-eigene Syntax ihre "*Metasyntax*". Die Metasyntax der EBNF kann fast in ihrem eigenen Formalismus beschrieben werden:

```
ebnf ::= {production}.
production ::= leftside "::=" rightside ".".
```

```

leftside ::= nonterminal.
rightside ::= alternative {"|" alternative}.
alternative ::= {sequence}.
sequence ::= "{" rightside "}" |
  "[" rightside "]" |
  "(" rightside ")" |
  symbol.
symbol ::= terminal | nonterminal.
terminal ::= doublequote text doublequote.
nonterminal ::= identifier.
identifier ::= letter {letter | digit}.

```

Die folgenden Nichtterminale sind nicht mehr aufgelöst:

- letter steht für einen Buchstaben;
- digit steht für eine Ziffer;
- text steht für eine Folge von beliebigen Zeichen außer Gänsefüßchen;
- doublequote steht für das Terminal mit Gänsefüßchen als Inhalt;

### 2.1.2. Syntax von C++

Die hier aufgeführte **kontextfreie Grammatik** entspricht der des **Standard-Entwurfs** des ANSI-Komitees auf dem Stand von 1997. Sie ist direkt aus dem ANSI-Dokument übernommen worden -- die Namen aller Symbole wurden in Englisch belassen.

Es sind dazu einige Vorbemerkungen nötig: Diese Grammatik beschreibt nämlich eigentlich eine wesentlich größere Sprache als C++.

C++ ist (wie die meisten höheren Programmiersprachen) eine **kontextsensitive Sprache**, so daß zu der hier angegebenen kontextfreien Grammatik **kontextsensitive Zusätze** erforderlich sind. Dort wird beispielsweise festgelegt, was **Gültigkeitsbereiche** von Bezeichnern sind, welche **Typen** in welchen Zusammenhängen miteinander **verträglich** sind, wann **case**-Statements erlaubt sind (nämlich nur in einem **switch**-Block), etc. Im ANSI-Dokument sind sie in ausführlicher Prosa formuliert, werden hier aber ausgelassen.

Außerdem ist die Grammatik in dieser Form **nicht eindeutig**, und es werden zusätzliche Regeln (wiederum umgangssprachlich) angegeben, um diese Mehrdeutigkeiten aufzulösen. Eine äquivalente eindeutige Grammatik wäre zu unübersichtlich geworden, ohne Vorteile zu bringen.

Die **erweiterte Backus-Naur-Form** bringt besonders bei klammer-intensiven Programmiersprachen wie C++ Probleme mit sich, da die Meta-Zeichen **( ) { } [ ] |** der EBNF mit Tokens der Sprache verwechselt werden könnten -- diese müßten dann "**{**", usw. geschrieben werden.

Deshalb wird hier wie folgt verfahren: Verschachtelte Strukturen werden durch Einführung genügend vieler Nicht-Terminalsymbole eliminiert. Dadurch kann die **Iteration** mit **{** und **}** ganz entfallen. Die **Option** [*symbol*] wird geschrieben als *symbol*<sub>opt</sub>. Die **Alternative** | ist nicht nötig, wenn mehrere Regeln zu einem Nicht-Terminal einfach untereinander geschrieben werden. Für eine Auswahl aus einer Vielzahl von Einzel-Tokens schreibt man "one of..."

Das **Startsymbol** der Grammatik ist **translation\_unit**.

**Terminalsymbole der Grammatik:**

**63 Schlüsselworte** (Identifizier-ähnlich geschriebene Terminalsymbole):

<code>asm</code>	<code>do</code>	<code>if</code>	<code>return</code>	<code>try</code>
<code>auto</code>	<code>double</code>	<code>inline</code>	<code>short</code>	<code>typedef</code>
<code>bool</code>	<code>dynamic_cast</code>	<code>int</code>	<code>signed</code>	<code>typeid</code>
<code>break</code>	<code>else</code>	<code>long</code>	<code>sizeof</code>	<code>typename</code>
<code>case</code>	<code>enum</code>	<code>mutable</code>	<code>static</code>	<code>union</code>
<code>catch</code>	<code>explicit</code>	<code>namespace</code>	<code>static_cast</code>	<code>unsigned</code>
<code>char</code>	<code>extern</code>	<code>new</code>	<code>struct</code>	<code>using</code>
<code>class</code>	<code>export</code>	<code>operator</code>	<code>switch</code>	<code>virtual</code>
<code>const</code>	<code>false</code>	<code>private</code>	<code>template</code>	<code>void</code>
<code>const_cast</code>	<code>float</code>	<code>protected</code>	<code>this</code>	<code>volatile</code>
<code>continue</code>	<code>for</code>	<code>public</code>	<code>throw</code>	<code>wchar_t</code>
<code>default</code>	<code>friend</code>	<code>register</code>	<code>true</code>	<code>while</code>
<code>delete</code>	<code>goto</code>	<code>reinterpret_cast</code>		

## 47 Spezialsymbole (für Operatoren und Interpunktionszeichen):

!	&=	++	->	/=	<<=	>>	
!=	(	+=	->*	:	<=	>>=	=
%	)	,	.	::	=	?	
%=	*	-	.*	;	==	^	}
&	*=	--	...	<	>	^=	~
&&	+	-=	/	<<	>=	{	

Die **Nicht-Terminalsymbole** sind direkt aus den folgenden **Produktionen** ablesbar. Das **Startsymbol** ist *translation-unit*.

## Keywords

***typedef-name***

*identifizier*

linke Seite der Regel

rechte Seite der Regel

***namespace-name***

*original-namespace-name*

*namespace-alias*

Alternativen

***original-namespace-name***

*identifizier*

## ***namespace-alias***

*identifier*

## ***class-name***

*identifier*

*template-id*

## ***enum-name***

*identifier*

## ***template-name***

*identifier*

## **Lexical conventions**

### ***hex-quad***

*hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit*

### ***universal-character-name***

*\u hex-quad*

*\U hex-quad hex-quad*

### ***preprocessing-token***

*header-name*  
*identifier*  
*pp-number*  
*character-literal*  
*string-literal*  
*preprocessing-op-or-punc*

## **token**

*identifier*  
*keyword*  
*literal*  
*operator*  
*punctuator*

## **header-name**

*<h-char-sequence >*  
*"q-char-sequence "*

## **h-char-sequence**

*h-char*  
*h-char-sequence h-char*

## **h-char**

any member of the source character set except new-line and >

## **q-char-sequence**

*q-char*

*q-char-sequence q-char*

## **q-char**

any member of the source character set except new-line and "

## **pp-number**

*digit*

*. digit*

*pp-number digit*

*pp-number nondigit*

*pp-number e sign*

*pp-number E sign*

*pp-number .*

## **identifier**

*nondigit*

*identifier nondigit*

*identifier digit*

## **nondigit**

*universal-character-name*

one of

a b c d e f g h i j k l <sup>M</sup>

n o p q r s t u v w x y <sup>Z</sup>

A B C D E F G H I J K L <sup>M</sup>

N O P Q R S T U V W X Y Z \_

***digit***

one of

0 1 2 3 4 5 6 7 8 9

***preprocessing-op-or-punc***

one of

{ } [ ] # ## ( )  
<: :> <% %> %: %:%: ; : ...

<code>new</code>	<code>delete</code>	<code>?</code>	<code>::</code>	<code>.</code>	<code>.*</code>				
<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>^</code>	<code>&amp;</code>	<code> </code>	<code>~</code>	
<code>!</code>	<code>=</code>	<code>&lt;</code>	<code>&gt;</code>	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	
<code>^=</code>	<code>&amp;=</code>	<code> =</code>	<code>&lt;&lt;</code>	<code>&gt;&gt;</code>	<code>&gt;&gt;=</code>	<code>&lt;&lt;=</code>	<code>==</code>	<code>!=</code>	
<code>&lt;=</code>	<code>&gt;=</code>	<code>&amp;&amp;</code>	<code>  </code>	<code>++</code>	<code>--</code>	<code>,</code>	<code>-&gt;*</code>	<code>-&gt;</code>	
<code>and</code>	<code>and_eq</code>	<code>bitand</code>	<code>bitor</code>	<code>compl</code>	<code>not</code>	<code>not_eq</code>	<code>or</code>	<code>or_eq</code>	
<code>xor</code>	<code>xor_eq</code>								

## ***literal***

*integer-literal*

*character-literal*

*floating-literal*

*string-literal*

*boolean-literal*

## ***integer-literal***

*decimal-literal integer-suffix<sub>opt</sub>*

*octal-literal integer-suffix<sub>opt</sub>*

*hexadecimal-literal integer-suffix<sub>opt</sub>*

## ***decimal-literal***

*nonzero-digit*  
*decimal-literal digit*

## ***octal-literal***

0  
*octal-literal octal-digit*

## ***hexadecimal-literal***

0x *hexadecimal-digit*  
0X *hexadecimal-digit*  
*hexadecimal-literal hexadecimal-digit*

## ***nonzero-digit***

one of

1 2 3 4 5 6 7 8 9

## ***octal-digit***

one of

1 2 3 4 5 6 7

## ***hexadecimal-digit***

one of

1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

### ***integer-suffix***

*unsigned-suffix long-suffix<sub>opt</sub>*

*long-suffix unsigned-suffix<sub>opt</sub>*

### ***unsigned-suffix***

one of

u U

### ***long-suffix***

one of

l L

### ***character-literal***

' *c-char-sequence* '

Ɑ *c-char-sequence* Ɑ

### ***c-char-sequence***

*c-char*

*c-char-sequence c-char*

### ***c-char***

any member of the source character set except the single-quote ' , backslash \ , or new-line character

*escape-sequence*

*universal-character-name*

### ***escape-sequence***

*simple-escape-sequence*

*octal-escape-sequence*

*hexadecimal-escape-sequence*

### ***simple-escape-sequence***

one of

\ ' \ " \ ? \\ \ a \ b \ f \ n \ r \ t \ v

### ***octal-escape-sequence***

\ *octal-digit*

\ *octal-digit octal-digit*

\ *octal-digit octal-digit octal-digit*

## **hexadecimal-escape-sequence**

$\backslash x$  hexadecimal-digit

hexadecimal-escape-sequence hexadecimal-digit

## **floating-literal**

fractional-constant exponent-part<sub>opt</sub> floating-suffix<sub>opt</sub>

digit-sequence exponent-part floating-suffix<sub>opt</sub>

## **fractional-constant**

digit-sequence<sub>opt</sub> . digit-sequence

digit-sequence .

## **exponent-part**

e sign<sub>opt</sub> digit-sequence

E sign<sub>opt</sub> digit-sequence

## **sign**

one of

+ -

## **digit-sequence**

digit

digit-sequence digit

## ***floating-suffix***

one of

`f l F L`

## ***string-literal***

`" s-char-sequenceopt "`

`L" s-char-sequenceopt "`

## ***s-char-sequence***

`s-char`

`s-char-sequence s-char`

## ***s-char***

any member of the source character set except the double-quote `"`, backslash `\`, or new-line character

`escape-sequence`

`universal-character-name`

## ***boolean-literal***

one of

`false true`

## **Programs**

## ***translation-unit***

*declaration-seq<sub>opt</sub>*

## **Expressions**

### ***primary-expression***

*literal*

**this**

*:: identifier*

*:: operator-function-id*

*:: qualified-id*

*( expression )*

*id-expression*

### ***id-expression***

*unqualified-id*

*qualified-id*

### ***unqualified-id***

*identifier*

*operator-function-id*

*conversion-function-id*

*~ class-name*

*template-id*

## **qualified-id**

*nested-name-specifier* **template**<sub>opt</sub> *unqualified-id*

## **nested-name-specifier**

*class-or-namespace-name* **::** *nested-name-specifier*<sub>opt</sub>

## **class-or-namespace-name**

*class-name*

*namespace-name*

## **postfix-expression**

*primary-expression*

*postfix-expression* [ *expression* ]

*postfix-expression* ( *expression-list*<sub>opt</sub> )

*simple-type-specifier* ( *expression-list*<sub>opt</sub> )

*postfix-expression* . **template**<sub>opt</sub> **::**<sub>opt</sub> *id-expression*

*postfix-expression* -> **template**<sub>opt</sub> **::**<sub>opt</sub> *id-expression*

*postfix-expression* . *pseudo- destructor-name*

*postfix-expression* -> *pseudo- destructor-name*

*postfix-expression* ++

*postfix-expression* --

**dynamic\_cast** < *type-id* > ( *expression* )

**static\_cast** < *type-id* > ( *expression* )

`reinterpret_cast < type-id > ( expression )`  
`const_cast < type-id > ( expression )`  
`typeid ( expression )`  
`typeid ( type-id )`

### ***expression-list***

*assignment-expression*  
*expression-list , assignment-expression*

### ***pseudo-destructor-name***

`::opt nested-name-specifieropt type-name :: ~ type-name`  
`::opt nested-name-specifieropt ~ type-name`

### ***unary-expression***

*postfix-expression*  
`++unary-expression`  
`--unary-expression`  
*unary-operator cast-expression*  
`sizeofunary-expression`  
`sizeof ( type-id )`  
*new-expression*  
*delete-expression*

### ***unary-operator***

one of

*\* & + - ! ~*

### ***new-expression***

*::<sub>opt</sub> new new-placement<sub>opt</sub> new-type-id new-initializer<sub>opt</sub>*

*::<sub>opt</sub> new new-placement<sub>opt</sub> ( type-id ) new-initializer<sub>opt</sub>*

### ***new-placement***

*( expression-list )*

### ***new-type-id***

*type-specifier-seq new-declarator<sub>opt</sub>*

### ***new-declarator***

*ptr-operator new-declarator<sub>opt</sub>*

*direct-new-declarator*

### ***direct-new-declarator***

*[ expression ]*

*direct-new-declarator [ constant-expression ]*

### ***new-initializer***

( *expression-list*<sub>opt</sub> )

### ***delete-expression***

**::<sub>opt</sub> delete** *cast-expression*

**::<sub>opt</sub> delete** [ ] *cast-expression*

### ***cast-expression***

*unary-expression*

( *type-id* ) *cast-expression*

### ***pm-expression***

*cast-expression*

*pm-expression* .\* *cast-expression*

*pm-expression* ->\* *cast-expression*

### ***multiplicative-expression***

*pm-expression*

*multiplicative-expression* \* *pm-expression*

*multiplicative-expression* / *pm-expression*

*multiplicative-expression* % *pm-expression*

### ***additive-expression***

*multiplicative-expression*

*additive-expression + multiplicative-expression*

*additive-expression - multiplicative-expression*

### **shift-expression**

*additive-expression*

*shift-expression << additive-expression*

*shift-expression >> additive-expression*

### **relational-expression**

*shift-expression*

*relational-expression < shift-expression*

*relational-expression > shift-expression*

*relational-expression <= shift-expression*

*relational-expression >= shift-expression*

### **equality-expression**

*relational-expression*

*equality-expression == relational-expression*

*equality-expression != relational-expression*

### **and-expression**

*relational-expression*  
*and-expression & equality-expression*

### ***exclusive-or-expression***

*and-expression*  
*exclusive-or-expression ^ and-expression*

### ***inclusive-or-expression***

*exclusive-or-expression*  
*inclusive-or-expression | exclusive-or-expression*

### ***logical-and-expression***

*inclusive-or-expression*  
*logical-and-expression && inclusive-or-expression*

### ***logical-or-expression***

*logical-and-expression*  
*logical-or-expression | | logical-and-expression*

### ***conditional-expression***

*logical-or-expression*  
*logical-or-expression ? expression : assignment-expression*

### ***assignment-expression***

*conditional-expression*

*logical-or-expression assignment-operator assignment-expression*

*throw-expression*

## **assignment-operator**

one of

= \*= /= %= += -= >>= <<= &= ^= |=

## **expression**

*assignment-expression*

*expression , assignment-expression*

## **constant-expression**

*conditional-expression*

## **Statements**

### **statement**

*labeled-statement*

*expression-statement*

*compound-statement*

*selection-statement*

*iteration-statement*

*jump-statement*

*declaration-statement*  
*try-block*

### ***labeled-statement***

*identifier* : *statement*  
**case** *constant-expression* : *statement*  
**default** : *statement*

### ***expression-statement***

*expression*<sub>opt</sub> ;

### ***compound-statement***

{ *statement-seq*<sub>opt</sub> }

### ***statement-seq***

*statement*  
*statement-seq* *statement*

### ***selection-statement***

**if** ( *condition* ) *statement*  
**if** ( *condition* ) *statement* **else** *statement*  
**switch** ( *condition* ) *statement*

### ***condition***

*expression*

*type-specifier-seq declarator = assignment-expression*

### ***iteration-statement***

**while** ( *condition* ) *statement*

**do** *statement* **while** ( *expression* ) ;

**for** ( *for-init-statement* *condition*<sub>opt</sub> ; *expression*<sub>opt</sub> ) *statement*

### ***for-init-statement***

*expression-statement*

*simple-declaration*

### ***jump-statement***

**break** ;

**continue** ;

**return** *expression*<sub>opt</sub> ;

**goto** *identifier* ;

### ***declaration-statement***

*block-declaration*

## **Declarations**

### ***declaration-seq***

*declaration*  
*declaration-seq declaration*

## **declaration**

*block-declaration*  
*function-definition*  
*template-declaration*  
*explicit-instantiation*  
*explicit-specialization*  
*linkage-specification*  
*namespace-definition*

## **block-declaration**

*simple-declaration*  
*asm-definition*  
*namespace-alias-definition*  
*using-declaration*  
*using-directive*

## **simple-declaration**

*decl-specifier-seq<sub>opt</sub> init-declarator-list<sub>opt</sub> ;*

## **decl-specifier**

*storage-class-specifier*  
*type-specifier*  
*function-specifier*  
**friend**  
**typedef**

### ***decl-specifier-seq***

*decl-specifier-seq*<sub>opt</sub> *decl-specifier*

### ***storage-class-specifier***

**auto**  
**register**  
**static**  
**extern**  
**mutable**

### ***function-specifier***

**inline**  
**virtual**  
**explicit**

### ***typedef-name***

*identifier*

### ***type-specifier***

*simple-type-specifier*  
*class-specifier*  
*enum-specifier*  
*elaborated-type-specifier*  
*cv-qualifier*

### ***simple-type-specifier***

`::opt nested-name-specifieropt type-name`  
`char`  
`wchar_t`  
`bool`  
`short`  
`int`  
`long signed`  
`unsigned`  
`float`  
`double`  
`void`

### ***type-name***

*class-name*  
*enum-name*  
*typedef-name*

### ***elaborated-type-specifier***

*class-key* ::<sub>opt</sub> *nested-name-specifier*<sub>opt</sub> *identifier*

*enum* ::<sub>opt</sub> *nested-name-specifier*<sub>opt</sub> *identifier*

**typename** ::<sub>opt</sub> *nested-name-specifier* *identifier*

**typename** ::<sub>opt</sub> *nested-name-specifier* *identifier* < *template-argument-list* >

### **enum-name**

*identifier*

### **enum-specifier**

**enum** *identifier*<sub>opt</sub> { *enumerator-list*<sub>opt</sub> }

### **enumerator-list**

*enumerator-definition*

*enumerator-list* , *enumerator-definition*

### **enumerator-definition**

*enumerator*

*enumerator* = *constant-expression*

### **enumerator**

*identifier*

### **namespace-name**

*original-namespace-name*  
*namespace-alias*

### ***original-namespace-name***

*identifier*

### ***namespace-definition***

*named-namespace-definition*  
*unnamed-namespace-definition*

### ***named-namespace-definition***

*original-namespace-definition*  
*extension-namespace-definition*

### ***original-namespace-definition***

`namespace` *identifier* { *namespace-body* }

### ***extension-namespace-definition***

`namespace` *original-namespace-name* { *namespace-body* }

### ***unnamed-namespace-definition***

`namespace` { *namespace-body* }

### ***namespace-body***

*declaration-seq<sub>opt</sub>*

### **namespace-alias**

*identifier*

### **namespace-alias-definition**

`namespace identifier = qualified-namespace-specifier ;`

### **qualified-namespace-specifier**

`::opt nested-name-specifieropt namespace-name`

### **using-declaration**

`using typenameopt ::opt nested-name-specifier unqualified-id ;`

`using :: unqualified-id ;`

### **using-directive**

`using namespace ::opt nested-name-specifieropt namespace-name ;`

### **asm-definition**

`asm ( string-literal ) ;`

### **linkage-specification**

**extern** *string-literal* { *declaration-seq*<sub>opt</sub> }  
**extern** *string-literal* *declaration*

## Declarators

### ***init-declarator-list***

*init-declarator*  
*init-declarator-list* , *init-declarator*

### ***init-declarator***

*declarator* *initializer*<sub>opt</sub>

### ***declarator***

*direct-declarator*  
*ptr-operator* *declarator*

### ***direct-declarator***

*declarator-id*  
*direct-declarator* ( *parameter-declaration-clause* ) *cv-qualifier-seq*<sub>opt</sub> *exception-specification*<sub>opt</sub>  
*direct-declarator* [ *constant-expression*<sub>opt</sub> ]  
( *declarator* )

### ***ptr-operator***

*\* cv-qualifier-seq<sub>opt</sub>*  
*&*  
*::<sub>opt</sub> nested-name-specifier \* cv-qualifier-seq<sub>opt</sub>*

### ***cv-qualifier-seq***

*cv-qualifier cv-qualifier-seq<sub>opt</sub>*

### ***cv-qualifier***

*const*  
*volatile*

### ***declarator-id***

*::<sub>opt</sub> id-expression*  
*::<sub>opt</sub> nested-name-specifier<sub>opt</sub> type-name*

### ***type-id***

*type-specifier-seq abstract-declarator<sub>opt</sub>*

### ***type-specifier-seq***

*type-specifier type-specifier-seq<sub>opt</sub>*

### ***abstract-declarator***

*ptr-operator abstract-declarator*<sub>opt</sub>  
*direct-abstract-declarator*

### **direct-abstract-declarator**

*direct-abstract-declarator* <sub>opt</sub> ( *parameter-declaration-clause* ) *cv-qualifier-seq*<sub>opt</sub> *exception-specification* <sub>opt</sub>  
*direct-abstract-declarator* <sub>opt</sub> [ *constant-expression*<sub>opt</sub> ]  
( *abstract-declarator* )

### **parameter-declaration-clause**

*parameter-declaration-list* <sub>opt</sub> . . . <sub>opt</sub>  
*parameter-declaration-list* , . . .

### **parameter-declaration-list**

*parameter-declaration*  
*parameter-declaration-list* , *parameter-declaration*

### **parameter-declaration**

*decl-specifier-seq declarator*  
*decl-specifier-seq declarator* = *assignment-expression*  
*decl-specifier-seq abstract-declarator*<sub>opt</sub>  
*decl-specifier-seq abstract-declarator*<sub>opt</sub> = *assignment-expression*

### **function-definition**

*decl-specifier-seq<sub>opt</sub> declarator ctor-initializer<sub>opt</sub> function-body*  
*decl-specifier-seq<sub>opt</sub> declarator function-try-block*

### **function-body**

*compound-statement*

### **initializer**

*= initializer-clause*  
*( expression-list )*

### **initializer-clause**

*assignment-expression*  
*{ initializer-list ,<sub>opt</sub> }*  
*{ }*

### **initializer-list**

*initializer-clause*  
*initializer-list , initializer-clause*

## **Classes**

### **class-name**

*identifier*  
*template-id*

## ***class-specifier***

*class-head { member-specification<sub>opt</sub> }*

## ***class-head***

*class-key identifier<sub>opt</sub> base-clause<sub>opt</sub>*

*class-key nested-name-specifier identifier base-clause<sub>opt</sub>*

## ***class-key***

**class**

**struct**

**union**

## ***member-specification***

*member-declaration member-specification<sub>opt</sub>*

*access-specifier : member-specification<sub>opt</sub>*

## ***member-declaration***

*decl-specifier-seq<sub>opt</sub> member-declarator-list<sub>opt</sub> ;*

*function-definition ;<sub>opt</sub>*

*qualified-id ;*

*using-declaration*

*template-declaration*

## ***member-declarator-list***

*member-declarator*  
*member-declarator-list member-declarator*

### ***member-declarator***

*declarator pure-specifier<sub>opt</sub>*  
*declarator constant-initializer<sub>opt</sub>*  
*identifier<sub>opt</sub> : constant-expression*

### ***pure-specifier***

= 0

### ***constant-initializer***

= *constant-expression*

## **Derived classes**

### ***base-clause***

: *base-specifier-list*

### ***base-specifier-list***

*base-specifier*  
*base-specifier-list , base-specifier*

### ***base-specifier***

*::<sub>opt</sub> nested-name-specifier<sub>opt</sub> class-name*

**virtual** *access-specifier<sub>opt</sub> ::<sub>opt</sub> nested-name-specifier<sub>opt</sub> class-name*

*access-specifier* **virtual** *<sub>opt</sub> ::<sub>opt</sub> nested-name-specifier<sub>opt</sub> class-name*

### ***access-specifier***

**private**

**protected**

**public**

### **Special member functions**

#### ***conversion-function-id***

**operator** *conversion-type-id*

#### ***conversion-type-id***

*type-specifier-seq conversion-declarator<sub>opt</sub>*

#### ***conversion-declarator***

*ptr-operator conversion-declarator<sub>opt</sub>*

#### ***ctor-initializer***

**:** *mem-initializer-list*

#### ***mem-initializer-list***

*mem-initializer*  
*mem-initializer* , *mem-initializer-list*

### ***mem-initializer***

*mem-initializer-id* ( *expression-list*<sub>opt</sub> )

### ***mem-initializer-id***

*::*<sub>opt</sub> *nested-name-specifier*<sub>opt</sub> *class-name*  
*identifier*

## **Overloading**

### ***operator-function-id***

*operator* *operator*

### ***operator***

one of

<b>new</b>	<b>new[]</b>	<b>delete</b>	<b>delete[]</b>			<b>( )</b>	<b>[]</b>	
<b>+</b>	<b>-</b>	<b>*</b>	<b>/</b>	<b>%</b>	<b>^</b>	<b>&amp;</b>	<b> </b>	<b>~</b>
<b>!</b>	<b>=</b>	<b>&lt;</b>	<b>&gt;</b>	<b>+=</b>	<b>--</b>	<b>*=</b>	<b>/=</b>	<b>%=</b>
<b>^=</b>	<b>&amp;=</b>	<b> =</b>	<b>&lt;&lt;</b>	<b>&gt;&gt;</b>	<b>&gt;&gt;=</b>	<b>&lt;&lt;=</b>	<b>==</b>	<b>!=</b>

<= >= && || ++ -- , ->\* ->

## Templates

### ***template-declaration***

`exportopt template < template-parameter-list > declaration`

### ***template-parameter-list***

`template-parameter`

`template-parameter-list , template-parameter`

### ***template-parameter***

`type-parameter`

`parameter-declaration`

### ***type-parameter***

`class identifieropt`

`class identifieropt = type-id`

`typename identifieropt`

`typename identifieropt = type-id`

`template < template-parameter-list > class identifieropt`

`template < template-parameter-list > class identifieropt = template-name`

### ***template-id***

*template-name < template-argument-list >*

### ***template-name***

*identifier*

### ***template-argument-list***

*template-argument*

*template-argument-list , template-argument*

### ***template-argument***

*assignment-expression*

*type-id*

*template-name*

### ***explicit-instantiation***

*template-declaration*

### ***explicit-specialization***

*template < > declaration*

## **Exception handling**

### ***try-block***

*try compound-statement handler-seq*

## ***function-try-block***

*try* *ctor-initializer*<sub>opt</sub> *function-body* *handler-seq*

## ***handler-seq***

*handler* *handler-seq*<sub>opt</sub>

## ***handler***

*catch* ( *exception-declaration* ) *compound-statement*

## ***exception-declaration***

*type-specifier-seq* *declarator*  
*type-specifier-seq* *abstract-declarator*  
*type-specifier-seq*  
...

## ***throw-expression***

*throw* *assignment-expression*<sub>opt</sub>

## ***exception-specification***

*throw* ( *type-id-list*<sub>opt</sub> )

## ***type-id-list***

*type-id*  
*type-id-list* , *type-id*

### **3. Semantik**

Formale Beschreibungen für die Semantik existieren. Dies wird in Veranstaltungen, wie z.B. Compilerbau behandelt.

Wir verzichten hier darauf.