Programmierkurs: Kontrollstrukturen & Funktionen

Manfred Hauswirth | Open Distributed Systems | Einführung in die Programmierung, WS 24/25

Rückblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git
VL 10 „Debugging und Stack“: Fehlverhalten/Bugs, Fehlersuche Strategien und Werkzeuge

Vorbemerkung: Syntax vs. Semantik

Syntax und Semantik

- Syntax
  - Legt fest, welche Zeichenketten Teil einer Sprache sind, d.h. die “Wörter” und “Sätze” der Sprache
- Semantik
  - Legt fest, was sie bedeuten
- Beispiel:
  - Syntax: a + b
  - Bedeutung: Addition von a und b

Syntax vs. Semantik – Hello World

Human

```
Say "Hello World"
```

Pseudocode

```
print "Hello World"
```

Python

```
print("Hello World")
```

R

```
print("Hello World")
```

Java

```
class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

C

```
#include <stdio.h>

int main()
{
    printf("Hello World");
    return 0;
}
```

Syntax vs. Semantik – Powers of 2

Human

```
Set m to 0 and p to 1.
As long as p is less than or equal to n,
    print "two to the power "
       value of m " ist " value of p.
    Increment m.
    Double p.
```

Pseudocode

```
n = 20;
m = 0;
p = 1;
while (p <= n)
    Ausgabe: "2^m ist p";
    m = m + 1;
    p = p * 2;
```

Python

```
n = 20
m = 0
p = 1
while p <= n:
    print("2^%d ist %d" % (m, p))
    m = m + 1
    p = p * 2
```

R

```
n <- 20
m <- 0
p <- 1
while(p < n){
    print(sprintf("2^%d ist %d", m, p))
    m = m + 1
    p = p * 2
}
```

Syntax vs. Semantik – Powers of 2

Java

```
public class Main {
    public static void main(String[] args) {
        int n = 20;
        int m = 0;
        int p = 1;
        while(p < n){
            System.out.printf("2^%d ist %d%n", m, p);
         m = m + 1;
         p = p * 2;
        }
    }
}
```

C

```
#include <stdio.h>

int main() {
    int n = 20;
    int m = 0;
    int p = 1;
    while (p < n) {
        printf("2^%d ist %d\n", m, p);
        m = m + 1;
        p = p * 2;
    }
}
```

Syntax-Fehler: Beispiel Absolutwert

- Beispiel 1:

```
if (x < 0) {
    x = -x;
}
```

Semikolon

- Beispiel 2:

```
if(x < 0){
    x = -x;
}
```

Runde Klammern

- Beispiel 3:

```
if(x < 0)x = -x;
```

Runde Klammern

Korrekt u.a.:

```
if (x < 0) {
    x = -x;
}
```

Oder:

```
if (x < 0)
    x = -x;
```

Besser?

Konsequenz: Programm nicht kompilierbar/übersetzbar

Exkurs: if – geschweifte Klammern

Falsch :-(

C ignoriert Einrückungen. Code wird so interpretiert:

```
if ( x < 0 )
    printf("x negative");
y = -x;
```

Richtig :-)

```
if ( x < 0 )
    printf("x negative");

if ( x < 0 )
    printf("x negative");
    y = -x;

if ( x < 0 )
    printf("x negative");
y = -x;

if ( x < 0 ){
    printf("x negative");
    y = -x;
}
```

Korrekt u.a.:

```
if (x < 0) {
    x = -x;
}
```

Oder:

```
if (x < 0)
    x = -x;
```

Besser?

Bitte **immer** geschweifte Klammern benutzen!

Semantik-Fehler: Bsp. Absolutwert

- Beispiel 1:

```
if (x > 0) {
    x = -x;
}
```

Zeichen falsch rum

- Beispiel 2:

```
if (x < 0) {
    x = -2 * x;
}
```

Falsch. Warum 2?

Korrekt u.a.:

```
if (x < 0) {
    x = -x;
}
```

Oder:

```
if (x < 0)
    x = -x;
```

Konsequenz: Programm kompiliert aber tut nicht das, was es soll…
Meistens viel schwieriger zu debuggen.

Fehlertypen

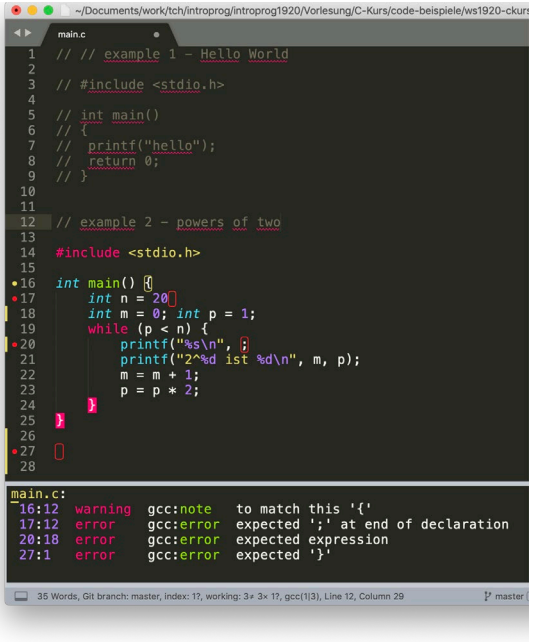
- Syntaxfehler
  - Konsequenz: Programm lässt sich nicht kompilieren/übersetzen.

- Semantikfehler (auch häufig Programmlogikfehler)
  - Konsequenz: Programm kompiliert aber tut nicht das Gewünschte.

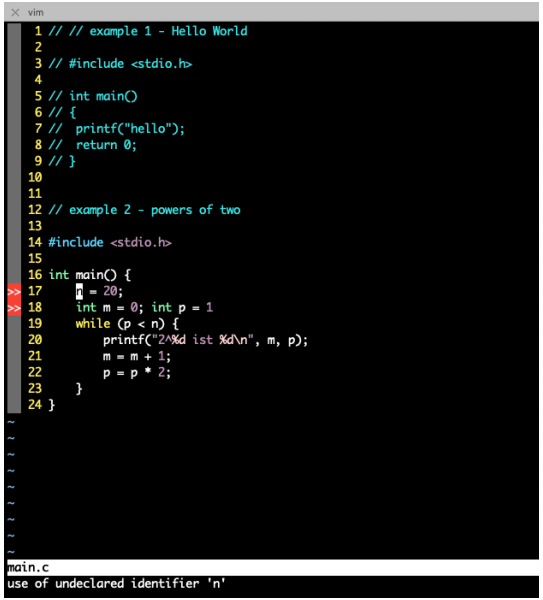
Automatische Syntaxprüfung - Editoren

Technische Universität Berlin

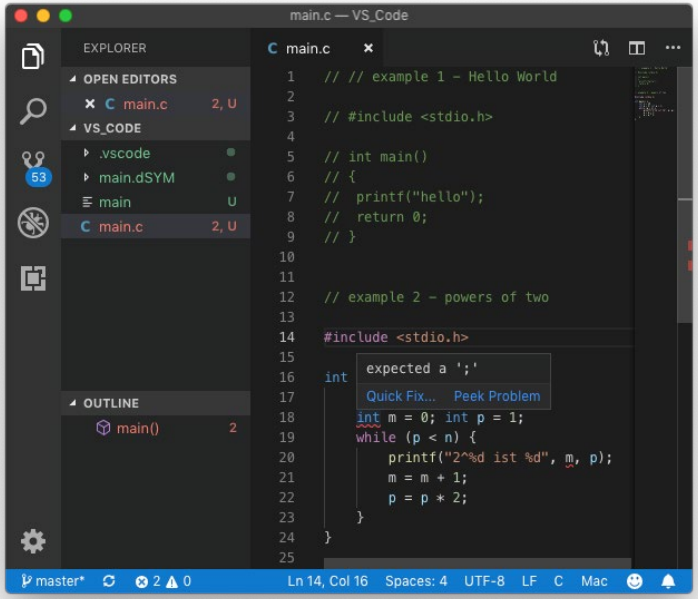
Sublime Text



vim



VS Code

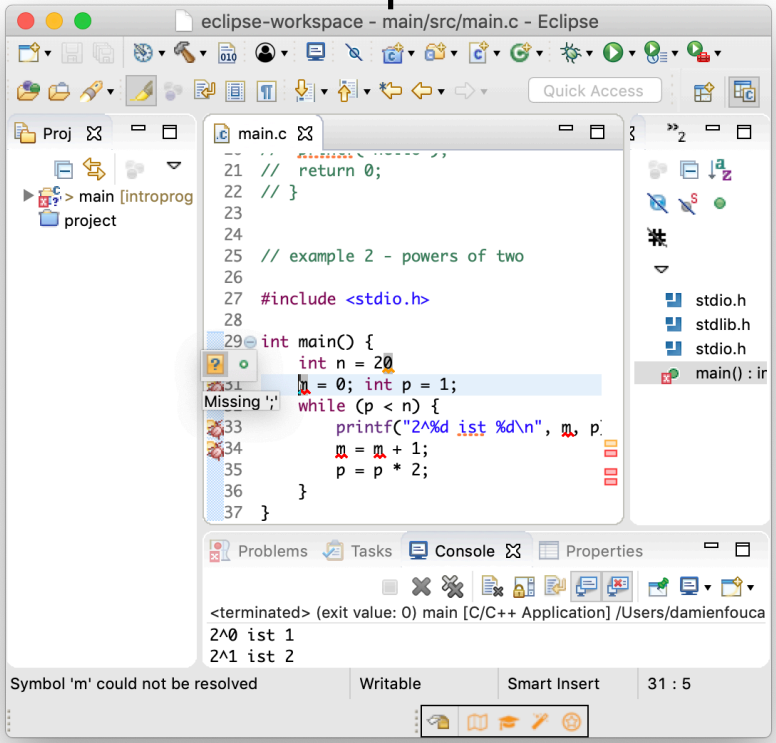


Open Distributed Sytems
ODS

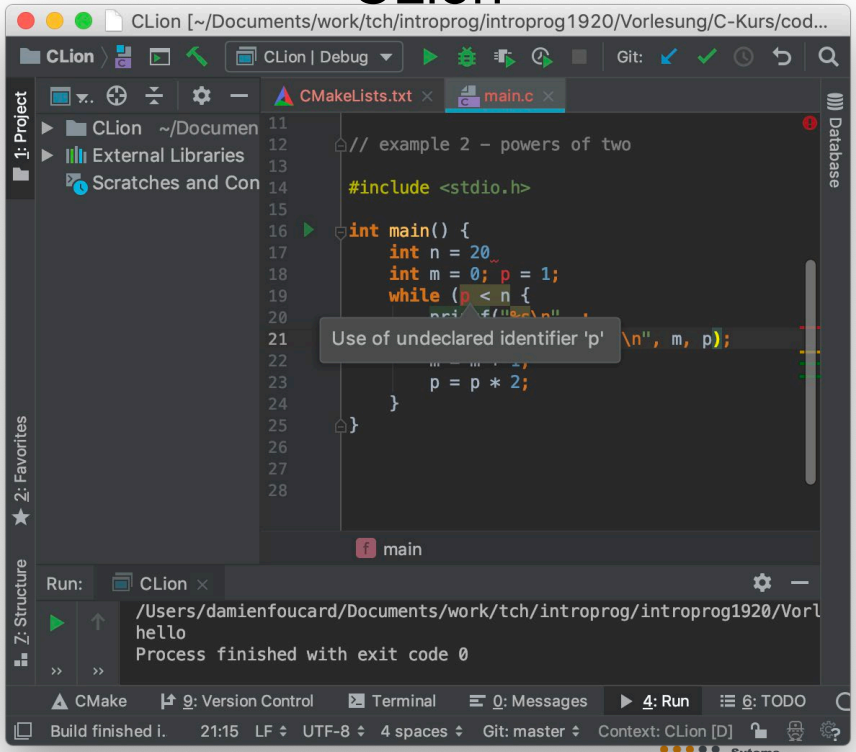
Automatische Syntaxprüfung - IDEs

Technische Universität Berlin

Eclipse



CLion

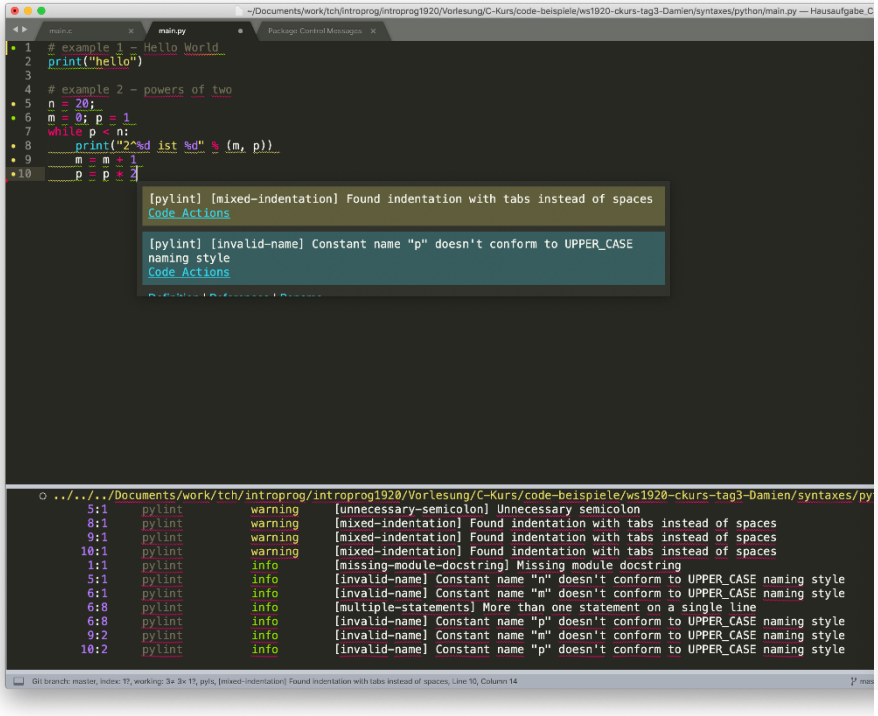


Sytems ODS

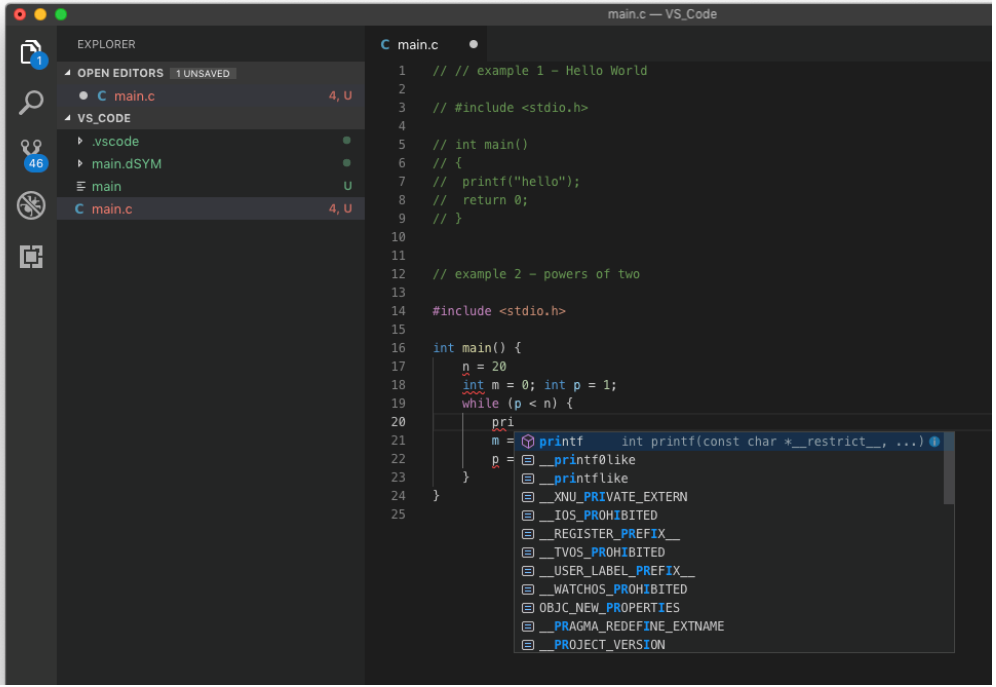
Automatische Ergänzung

Technische Universität Berlin

Sublime Text



VS Code



Open Distributed Sytems
ODS

Richtige Formatierung

Schlechte Formatierung

```
#include <stdio.h>

int
main() {
            int n = 20; int m = 0; int p = 1;
    while (p
        < n) {
printf("2^%d ist %d\n",


     m, p);
         m = m + 1;
     p = p * 2;}
         }
```

Auf einer Zeile bitte

Nicht alles in einer Zeile definieren. ***Debugging*** *schwerer.*

Warum diese komischen Einrückungen? 👀

Gute Formatierung

```
#include <stdio.h>

int main() {
    int n = 20;
    int m = 0;
    int p = 1;
    while (p < n) {
        printf("2^%d ist %d\n", m, p);
        m = m + 1;
        p = p * 2;
    }
}
```

Debugging

- Wörtlich: „Entwanzen“
  - Die ersten Computer verwendeten Relais
  - 1 Relais = 1 Bit
  - offen „0“, geschlossen: „1“

- Moderner Gebrauch:
  - Bug = Programm-Fehler
  - Debuggen = Fehlersuche und -behebung

Bedingte Anweisungen

Blöcke (Wiederholung)

- Ein Block ist eine Zusammenfassung einer Folge von Anweisungen.

```
{ // begin of block
    int z = x;
    x = y;
    y = z;
} // end of block
```

- Eine Zusammenfassung von Ausdrücken wird in C durch geschweifte Klammern { ... } realisiert.
- 1 Block = 1 Scope: definiert Variablen, die zur Verfügung stehen

Blöcke – Beispiel 1: Scope

Technische Universität Berlin

Falsch:

```
#include <stdio.h>
int main() {
    {
        int n = 20;
    }
    printf("%d\n", n);
}
```

Error: Use of undeclared identifier n

Richtig (aber sinnlos):

```
#include <stdio.h>
int main() {
    int n = 20;
    {
        printf("%d\n", n);
    }
}
```

„20“

Ein Block öffnet einen Scope („Sichtbarkeit“). Innerhalb eines Scopes, ist alles von „außerhalb“ sichtbar. Von „außen“ kann nicht in einen Scope „hineingeblickt“ werden.

Blöcke

- Hilfreich für die Lesbarkeit des Programms und für die Fehlersuche ist eine an der Blockstruktur orientierte Einrückung („Indentation“).
- Blöcke können geschachtelt sein.

```
<block> ::= {<statements>}
<statements> ::= <statement> |
                 <statements> <statement>
```

Syntax Beschreibung: Backus-Naur-Form (BNF)

```
<block> ::= {<statements>}
<statements> ::= <statement> |
          <statements> <statement>
```

- Diese Schreibweise heißt *Backus-Naur-Form* (BNF) und wird oft zur Syntax-Definition von Programmiersprachen benutzt
- Sie beschreibt die Syntax in formalisierter Weise.

```
<komplexes Konstrukt> ::= <einfachere Konstrukte>
```

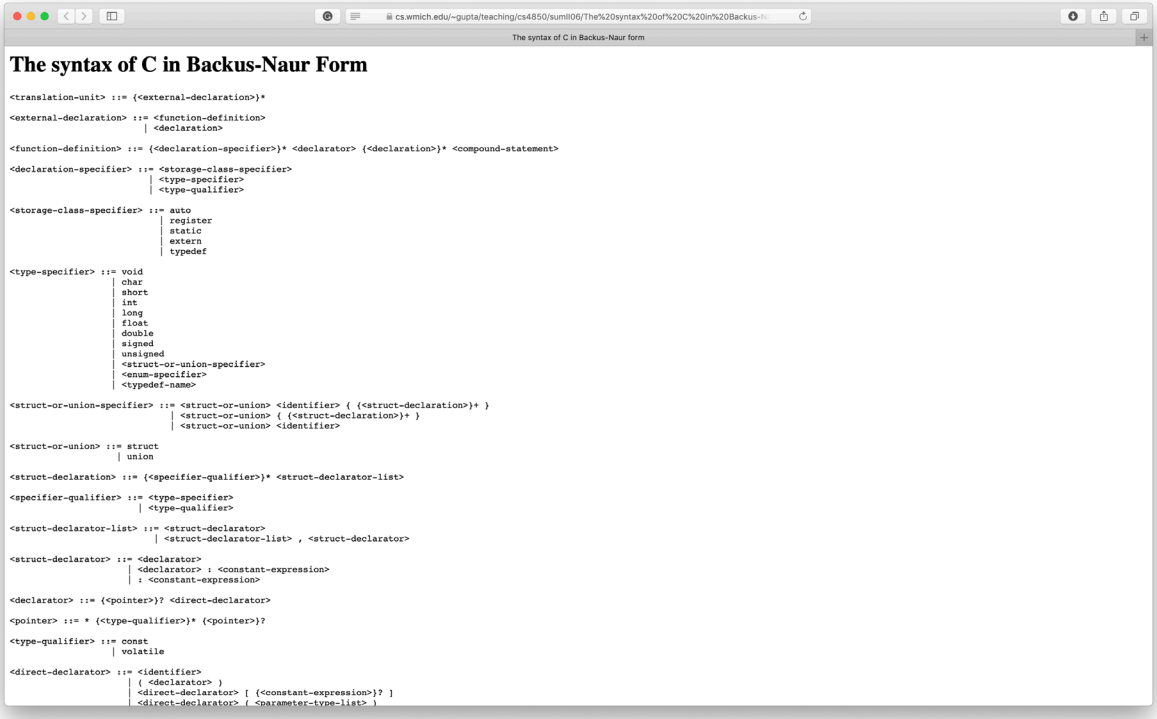
Blöcke

- Hilfreich für die Lesbarkeit des Programms und für die Fehlersuche ist eine an der Blockstruktur orientierte Einrückung („Indentation“).
- Blöcke können geschachtelt sein.

```
<block> ::= {<statements>}
<statements> ::= <statement> |
   <statements> <statement>

<statement> ::= <assignment>; | ...
<assignment> ::= <variable> = <expression>
<expression> ::= ...
```

C-Syntax in BNF



The syntax of C in Backus-Naur Form

```
<translation-unit> ::= {<external-declaration>}*

<external-declaration> ::= <function-definition>
                         | <declaration>

<function-definition> ::= {<declaration-specifier>}* <declarator> {<declaration>}* <compound-statement>

<declaration-specifier> ::= <storage-class-specifier>
                          | <type-specifier>
                          | <type-qualifier>

<storage-class-specifier> ::= auto
                            | register
                            | static
                            | extern
                            | typedef

<type-specifier> ::= void
                   | char
                   | short
                   | int
                   | long
                   | float
                   | double
                   | signed
                   | unsigned
                   | <struct-or-union-specifier>
                   | <enum-specifier>
                   | <typedef-name>

<struct-or-union-specifier> ::= <struct-or-union> <identifier> { {<struct-declaration>}+ }
                              | <struct-or-union> { {<struct-declaration>}+ }
                              | <struct-or-union> <identifier>

<struct-or-union> ::= struct
                    | union

<struct-declaration> ::= {<specifier-qualifier>}* <struct-declarator-list>

<specifier-qualifier> ::= <type-specifier>
                        | <type-qualifier>

<struct-declarator-list> ::= <struct-declarator>
                           | <struct-declarator-list> , <struct-declarator>

<struct-declarator> ::= <declarator>
                      | <declarator> : <constant-expression>
                      | : <constant-expression>

<declarator> ::= {<pointer>}? <direct-declarator>

<pointer> ::= * {<type-qualifier>}* {<pointer>}?

<type-qualifier> ::= const
                   | volatile

<direct-declarator> ::= <identifier>
                      | ( <declarator> )
                      | <direct-declarator> [ {<constant-expression>}? ]
                      | <direct-declarator> ( <parameter-type-list> )
```

https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm

Bedingte Anweisung

- Manche Anweisungen sollen nur unter bestimmten Bedingungen ausgeführt werden.

  Z.B.: Berechne den Absolutwert einer Variable:

```
if (x < 0) {
   x = -x;
}
```

- Syntaktische Form:

```
if (<condition>) <block>
```

Bedingte Anweisung ohne Alternative

- Wenn n = 0, mache etwas:

```
if (n == 0)
{
    // do something
}
```

Achtung!

Bedingte Anweisung mit Alternative – naive Version

- Wenn n = 0, mache etwas.
- Sonst, d.h., wenn n ≠ 0, mache etwas Anderes:

```
if (n == 0)
{
        // do something
}
if (n != 0)
{
        // do something else
}
```

Bedingte Anweisung mit Alternative – naive Version

- Wenn n = 1, tue etwas.
- Sonst, d.h., wenn n ≠ 1, tue etwas Anderes:

```
if (n == 1)
{
        // do something
}
if (n != 0)
{
        // do something else
}
```

Neue Anforderung

Bedingung anpassen

Fehlt was?

Anpassung der Gegenbedingung übersehen => Code jetzt inkorrekt 

Bedingte Anweisung mit Alternative – naive Version

- Wenn n = 1, tue etwas.
- Sonst, d.h., wenn n ≠ 1, tue etwas Anderes, d.h. „sonst“

```
if (n == 1)
{
        // do something
}
else
{
        // do something else
}
```

Man kann hier einfach „sonst“ sagen! ☺

Bedingte Anweisung mit Alternative

- Verallgemeinerte Form der If-Anweisung lautet

```
if (<condition>) <block> else <block>
```

- Abhängig von der Bedingung wird eine der Alternativen ausgeführt.

- Beispiel: Maximum zweier Zahlen finden

```
// set z to maximum of x and y
if (x > y) {  // condition
   z = x;  // then-part
} else {
   z = y;  // else-part
}
```

Bedingte Anweisung mit mehreren Alternativen – naive Version

- Wenn n = 1, mache etwas.
- Wenn n = 2, mache etwas Anderes
- Wenn n weder 1 noch 2, mache etwas ganz Anderes.

```
if (n == 1)
{
        // do thing 1
}
if (n == 2)
{
        // do thing 2
}
else
{
        // do thing 3
}
```

Macht der Code das, was wir wollen? 

Nope!  : Tut „thing 3“ auch wenn n = 1

Bedingte Anweisung mit mehreren Alternativen – naive Version

- Wenn n = 1, tue etwas.
- Wenn n = 2, tue etwas Anderes
- Wenn n weder 1 noch 2, tue noch etwas Anderes.

```
if (n == 1)
{
      // do thing 1
}
else
{
      // do thing 3
}
```

```
if (n == 2)
{
      // do thing 2
}
else
{
      // do thing 3
}
```

Mach der Code das, was wir wollen?

Nope! : Macht „thing 3“, wenn n = 1 und auch wenn n = 2!

Bedingte Anweisung mit mehreren Alternativen – naive Version

- Wenn n = 1, tue etwas.
- Wenn n = 2, tue etwas Anderes
- Wenn n weder 1 noch 2, tue noch etwas Anderes.

```
if (n == 1)
{
      // do thing 1
}
else if (n == 2)
{
      // do thing 2
}
else
{
      // do thing 3
}
```

Macht der Code das, was wir wollen?

Yep! :

- Macht „thing 1“ wenn n = 1
- Sonst wir geprüft, ob n = 2.
  - Wenn n = 2, wird „thing 2“ gemacht
  - Sonst „thing 3“

Bedingte Anweisung mit Alternative

- Das `else if` wird verwendet, um abhängig von einer Bedingung zwischen verschiedenen Blöcken zu wählen:

```
if (n == 1) {            // execute block #1
}
else if (n == 2) {       // execute block #2
}
else if (n == 3) {       // execute block #3
}
else {   // if all fails, execute block #4
}
```

Logische Ausdrücke (Boolean Expressions)

- Logische Ausdrücke in C:

  Wert `==` 0 → false / falsch

  Wert `!=` 0 → true / wahr

- Vergleichsoperatoren liefern Integer Werte 0 oder 1:

  `==` gleich, `!=` ungleich, `<` kleiner, `>` größer,

  `<=` kleiner gleich, `>=` größer gleich

- Verknüpfung von logischen Ausdrücken:

  `&&` logisches und (beides wahr)

  `||` logisches oder (mindestens eines von beiden wahr)

Schleifen und Iterationen

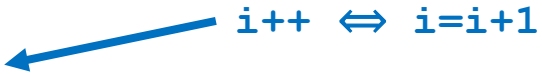
Schleifen und Iterationen

- Es gibt häufig Situationen, in denen ein Programmstück mehrmals mit jeweils sich ändernden Werten durchlaufen werden soll: Schleifen.

- `while`-Schleife: Anzahl der Iterationen wird durch eine Bedingung bestimmt.
- `for`-Schleife: Anzahl der Iterationen ist bekannt.

for-Schleife

- Beispiel: Zählt von 0 bis 10.




```
int i;
for (i = 0; i <= 10; i++){
   printf("i: %d\n", i);
}
```

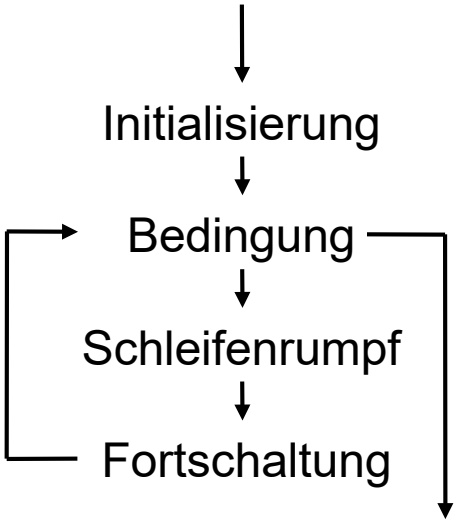
- Allgemein lautet die Syntax:

```
<for-statement> ::=
   for(<for-init>; <condition>; <for-update>)
   <block>

<for-init> ::= <statement>
<for-update> ::= <statement>
```

`for`-Schleife

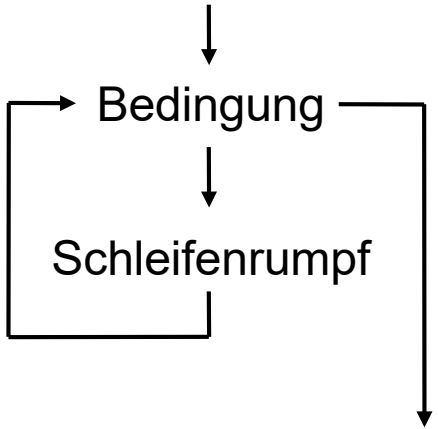
1. **Initialisierung:** Deklaration und Initialisierung der Schleifenvariable
2. **Bedingung:** Logischer Ausdruck, der „wahr“ sein muss. Wird vor jeder Ausführung des Schleifenrumpfs getestet.
3. **Schleifenrumpf:** Anweisung(en), die wiederholt ausgeführt werden, solange die Bedingung den Wert „true“ ergibt.
4. **Fortschaltung**



`while`-Schleife

- Allgemeine Form:

```
<while-statement> :=
   while (<condition>) <block>;
```



- Unter Umständen wird der Schleifenrumpf nie ausgeführt! (kann auch bei einer **`for`**-Schleife passieren)
- **`while`**- und **`for`**-Schleifen sind semantisch äquivalent, d.h. jede **`while`**-Schleife kann als **`for`**-Schleife geschrieben werden und umgekehrt.

`while`-Schleife

- Schleifen können ineinander geschachtelt werden:

```
int a, b;
a = 1;
while (a < 10) {
    b = 1;
    while (b < 10) {
        printf("%d ", a * b);
        b = b + 1; // kürzer: b++;
    }
    printf("\n");  // neue Zeile
    a = a + 1;     // kürzer: a++;
}
```

Ausgabe:
1 2 3 4 …… 9
2 4 6 8 …… 18
3 6 9 12 ….. 27
.
.
9 18 27 …… 81

Vergleich: `while`-/`for`-Schleife Zählen von 0 bis 10

Technische Universität Berlin

`while`-Schleife

```
int i = 0;
while (i <= 10) {
  printf("i: %d\n", i);
  i++;
}
```

`for`-Schleife

```
int i;
for (i=0; i <= 10; i++) {
  printf("i: %d\n", i);
}
```

Wann `for` benutzen?

Man weiß, wie oft man etwas machen will, hier 11 mal:

```
int i;
for (i=0; i <= 10; i++) {
      // do something for the i-th time
}
```

Wann `while` benutzen?

Man will etwas mehrmals machen, aber es ist nicht im voraus klar, wie oft:

```
int i;
while (sun_is_shining) {
        // do something you can only do
        // when the sun is shining
}
```

Funktionen

Funktionen: Motivation

```
// do a complex operation on first input

// do the same complex operation on second input

// do the same complex operation on third input

// etc.
```

```
// implement complex operation once (in a function)


// call function on first input (reuse code)

// call function on second input (reuse code)

// call function on third input (reuse code)

// etc.
```

- **Einfachheit:** Funktion nur einmal korrekt implementieren; funktioniert dann immer gleich.
- **Wartbarkeit**: Änderung nötig? Nur eine Stelle (die Funktion) muss angepasst werden.

Funktionen: Motivation

```
#include <stdio.h>

int main()
{
    int x1 = 10;
    int x2 = 2;
    ...
    int x100 = 3;

    if (x1 > x2) {
        printf("value of max of x1 and x2: %d\n", x1);
    } else {
        printf("value of max of x1 and x2 is %d\n", x2);
    }
...
    if (x99 > x100) {
        printf("value of max of x99 and x100 is %d\n", x99);
    } else {
        printf("value of max of x99 and x100 is %d\n", x100);
```

```
#include <stdio.h>

int main()
{
    int x1 = 10;
    int x2 = 2;
    ...
    int x100 = 3;

    int max12 = max(x1, x2)
    printf("value of max of x1 and x2 is %d\n",
           max12);
    ...
    int max99100 = max(x99, x100)
    printf("value of max of x99 and x00 is
           %d\n", max99100);
}
```

```
int max(int a, int b){
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Funktionen: Wartbarkeit

```
#include <stdio.h>

int main()
{
    int x1 = 10;
    int x2 = 2;
    ...
    int x100 = 3;

    if (x1 < x2) {
        printf("value of max of x1 and x2: %d\n", x1);
    } else {
        printf("value of max of x1 and x2 is %d\n", x2);
    }
...
    if (x99 < x100) {
        printf("value of max of x99 and x100 is %d\n", x99);
    } else {
        printf("value of max of x99 and x100 is %d\n", x100);
```

Fehler – Zeichen falsch rum

```
#include <stdio.h>

int main()
{
    int x1 = 10;
    int x2 = 2;
    ...
    int x100 = 3;

    int max12 = max(x1, x2);
    printf("value of max of x1 and x2 is %d\n",
           max12);
    ...
    int max99100 = max(x99, x100);
    printf("value of max of x99 and x00 is
           %d\n", max99100);
}
```

```
int max(int a, int b){
    if (a < b) {
        return a;
    } else
        return b;
    }
}
```

Fehler – Zeichen falsch rum

Funktionen: Wartbarkeit

```
#include <stdio.h>

int main()
{
    int x1 = 10;
    int x2 = 2;
    ...
    int x100 = 3;

    if (x1 < x2) {
        printf("value of max of x1 and x2: %d\n", x1);
    } else {
        printf("value of max of x1 and x2 is %d\n", x2);
    }
...
    if (x99 < x100) {
        printf("value of max of x99 and x100 is %d\n", x99);
    } else {
        printf("value of max of x99 and x100 is %d\n", x100);
```

100 Korrekturen notwendig

```
#include <stdio.h>

int main()
{
    int x1 = 10;
    int x2 = 2;
    ...
    int x100 = 3;

    int max12 = max(x1, x2);
    printf("value of max of x1 and x2 is %d\n",
           max12);
    ...
    int max99100 = max(x99, x100);
    printf("value of max of x99 and x00 is
           %d\n", max99100);
}
```

```
int max(int a, int b){
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

eine Korrektur notwendig :)

Funktionen: Motivation

- Strukturierte Programmierung:
  - Modularisierung
  - Vermeidung von komplexen Kontrollstrukturen
  - Kapselung
  - Dokumentation
- Vorteile
  - Übersichtlicher
  - Lesbarer
  - Testbarkeit
  - Wiederverwendbarkeit
  - Wartbarkeit

Funktionen – Beispiel

Berechnung des Maximums

```
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Funktionen – Beispiel

Berechnung des Maximums, aber gut dokumentiert

```
// function to calculate the maximum of a and b
int max(int a, int b) {
   if (a > b) {     // condition
         return a; // a is max => return its value
   } else {
         return b; // b is max => return its value
   }
}
```

Funktionen (vereinfacht)

- Vereinfachte Form der Funktion ist

```
type name(parameters) <block>
```

- Rückgabe eines Wertes mittels Schlüsselwort **`return`**

- Beispiel: Maximum

```
// function to calculate the maximum of a and b
int max (int a, int b) {
    if (a > b) {  // condition
        return a; // a is max => return its value
    } else {
        return b; // b is max => return its value
    }
}
```

Funktionsaufruf

```
... // Definition of max
int main() {
  int r1, r2;
  int n = 10;
  int m = 11;

  r1 = max(10, 11);  // Aufruf mit festen Werten
  r2 = max(n, m);    // Aufruf mit Variablen

  printf("1: max of 10, 11: %d\n", r1);
  printf("2: max of n, m: %d\n", r2);
  // Aufruf innerhalb eines anderen Aufrufs
  printf("3: max of n, m: %d\n", max(n, m));
}
```

Funktionen

- Funktionen bilden das Grundgerüst jedes Programms:
  - Modularisierung
  - Vermeidung von komplexen Kontrollstrukturen
  - Kapselung
  - Dokumentation
- Gültigkeit von Variablen/Scoping
  - Immer innerhalb des Blockes, in dem sie definiert sind.
  - Gilt insbesondere für die Variablen in einer Funktion.
  - Wertübergaben von einer Funktion an eine andere mittels Parameter und Rückgabewert

Funktionsaufrufe

- Jede Funktion kann von jeder Funktion aufgerufen werden
- Beispiele:
  - `max()` von `main()` aus
  - `max()` von `printf()` aus
- Insbesondere kann eine Funktion auch sich selbst aufrufen! → Rekursion

Funktionen (vereinfachte Syntax)

```
<function> ::= <declarator-specifier><declarator><block>

<declarator-specifier> ::= <type-specifier> | ...
<type-specifier> ::= void | int | char | ...
<declarator> ::= <identifier> () |
                 <identifier> (<parameter-list>) |
                 ...

<parameter-list> ::= <type-specifier> <identifier> |
       <type-specifier><identifier>, <parameter-list>
<identifier> ::= ...
```

Ausblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git
VL 10 „Debugging und Stack“: Fehlverhalten/Bugs, Fehlersuche Strategien und Werkzeuge

Slides für Interessierte

for-Schleife

- Schleifen können ineinander geschachtelt werden:

```
int a, b;
for (a = 1; a < 10; a++) {
    for (b = 1; b < 10; b++) {
        printf("%d ", a * b);
    }
    printf("\n");
}
```

Ausgabe:

```
1 2 3 4 ..... 9
2 4 6 8 ..... 18
3 6 9 12 ..... 27
.
.
9 18 27 ..... 81
```

Von Semantik zu Syntax

Semantik kommt zuerst – Bsp. 1 – Iterationen aufzählen

Technische Universität Berlin

Semantik falsch: Unsinn

```
#include <stdio.h>
int main(){
    int n = 10;
    n = n + 1;
    n = 20;
    for (int i = 0; i < 10; ++i)
    {
        if (i < n)
        {
            printf("We have covered %d
iterations so far\n", i + 2);
            printf("We still have %d
iterations to go\n", n - i);
        }
    }
    printf("We're done.\n");
}
```

Semantik richtig: Code nachvollziehbar

```
#include <stdio.h>
int main(){
    int n = 20;
    for (int i = 0; i < n; ++i)
    {
        printf("We have covered %d iterations so
far\n", i );
        printf("We still have %d iterations to
go\n", n - i);
    }
    printf("We're done.\n");
}
```

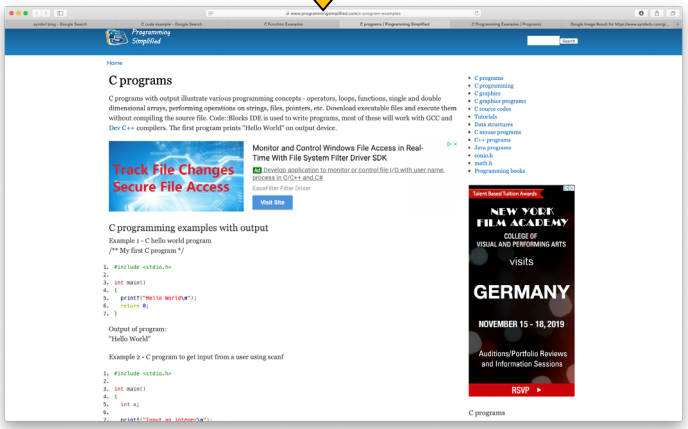
Zuerst die Semantik gut überlegen ☺, dann kommt die Syntax (fast) von selbst

Semantik kommt zuerst – Bsp. 2 - Teilbarkeit durch 7

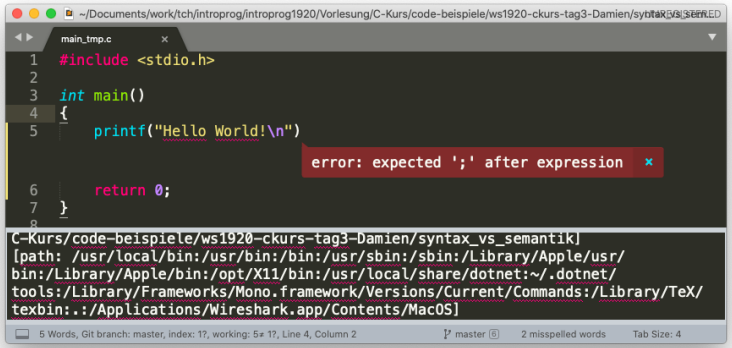
1. Wie sieht eine C-Code-Datei noch mal aus?

Suche

=> zweites Ergebnis nutzbar



https://www.programmingsimplified.com/c-program-examples




```
#include <stdio.h>
int main()
{
    printf("Hello World\n")
    return 0;
}
```

Test 1:
Error: expected ‘;‘ after expression

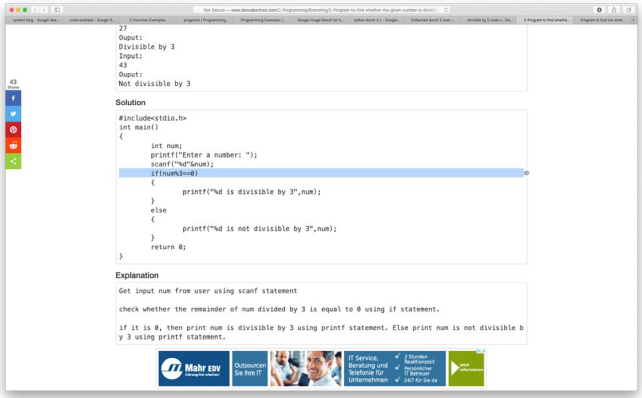
Test 2:
Hello World

Semantik kommt zuerst – Bsp. 2 - Teilbarkeit durch 7

2. Teilbar durch 7 in C?

Suche

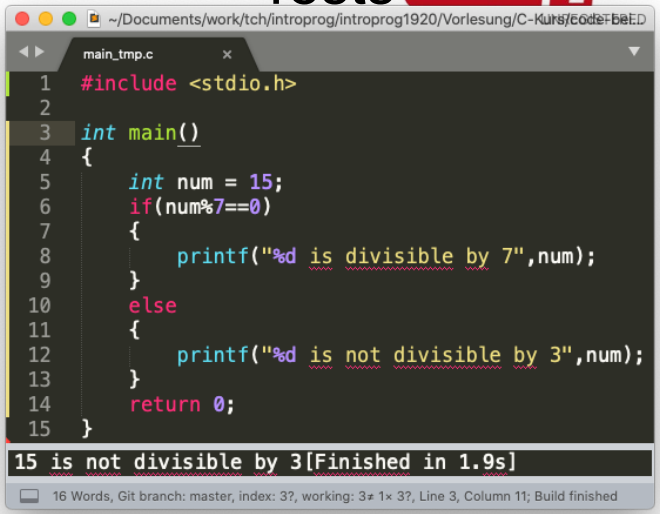
=> erstes Ergebnis gut genug



link

```c
#include <stdio.h>
int main()
{
    int num = 15;
    if(num%7==0)
    {
        printf("%d is divisible by 7",num);
    }
    else
    {
        printf("%d is not divisible by 3",num);
    }
    return 0;
}
```

Tests



Test 1:
14 is divisible by 7

Test 2:
15 is not divisible by 3

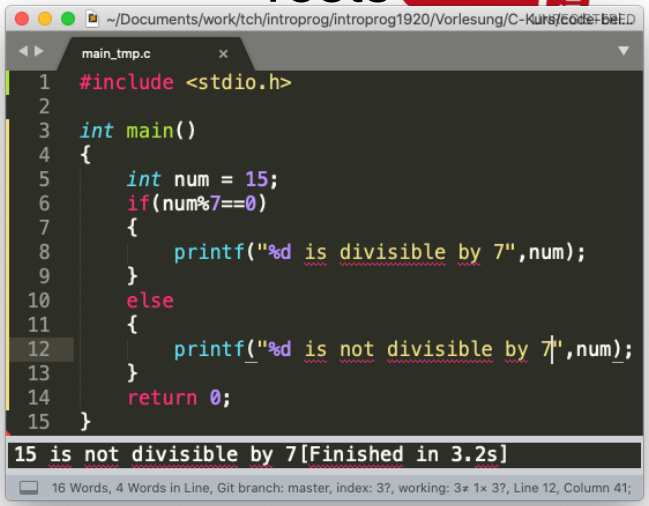
Open Distributed Sytems
ODS

Semantik kommt zuerst – Bsp. 2 - Teilbarkeit durch 7

3. Mehr Debugging: 😉

```c
#include <stdio.h>
int main()
{
    int num = 15;
    if(num%7==0)
    {
        printf("%d is divisible by 7",num);
    }
    else
    {
        printf("%d is not divisible by 7",num);
    }
    return 0;
}
```

Tests



Test 1:
14 is divisible by 7
😉

Test 2:
15 is not divisible by 7
😉

Open Distributed Sytems
ODS

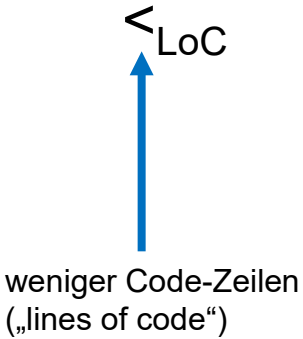
Warum Pseudo-Code ein so mächtiges Werkzeug ist

Pseudo-Code minimal – Bsp. 1: Powers of Two

Pseudo-Code kurz und knapp, Syntax minimal

→ man kann sich auf die Semantik fokussieren ☺

```
m <- 0
p <- 1
while p < n do
    Ausgabe: "2^m ist p"
    m <- m + 1
    p <- p * 2
```




```
#include <stdio.h>

int main() {
    int n = 20;
    int m = 0; int p = 1;
    while (p < n) {
        printf("2^%d ist %d", m, p);
        m = m + 1;
        p = p * 2;
    }
}
```

Pseudo-Code minimal – Bsp. 1: In der Forschung

Algorithm 4: TopPPR

Input: Graph G, source node s, decay factor α, precision ρ, parameter k

Output: $T_k(s) = \{t_1, \ldots, t_k\}$, the exact top-$k$ node set of s

1 $V_k \leftarrow \emptyset, C \leftarrow V$;

2 $n_r \leftarrow 4\sqrt{mn \log n}, r^f_{max} \leftarrow \frac{4}{\sqrt{mn \log n}}$;

3 $[r^f, \pi^f] \leftarrow \text{ForwardSearch}(r^f_{max})$;

4 $[\hat{\pi}_b, V_k, C] \leftarrow \text{CandidateUpdate}(r^f, \pi^f, V_k, C)$;

5 $n_r \leftarrow \frac{n \log n}{|C|}, r^b_{max} \leftarrow \frac{1}{\sqrt{m}}, r^f_{max} \leftarrow \frac{1}{m}$;

6 **while** $|V_k| < \rho k$ **do**

7 $\quad [r^b, \pi^b] \leftarrow \text{GroupBackwardSearch}(r^b_{max}, \hat{\pi}_b)$;

8 $\quad [r^f, \pi^f] \leftarrow \text{ForwardSearch}(r^f_{max})$;

9 $\quad [\hat{\pi}, V_k, C] \leftarrow \text{CandidateUpdate}(r^f, \pi^f, r^b, \pi^b, V_k, C)$;

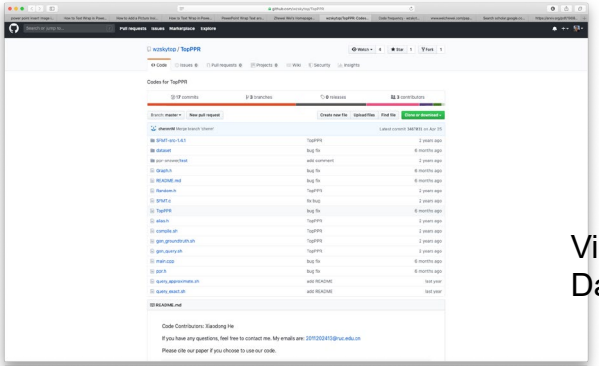
10 $\quad r^b_{max} \leftarrow \frac{r^b_{max}}{2}, r^f_{max} \leftarrow \frac{r^f_{max}}{2}, n_r \leftarrow 2n_r$;

11 **return** V_k;

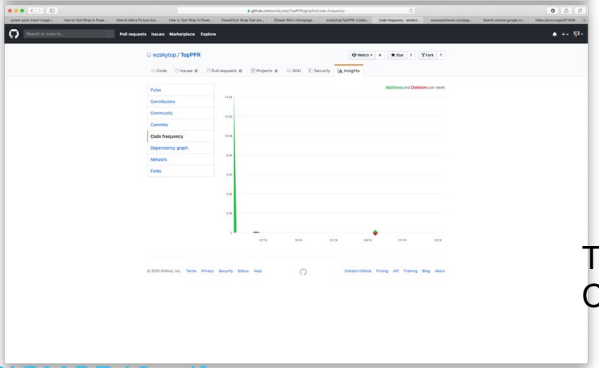
nur 11 Code-Zeilen in Pseudocode ;)

<< LoC

viel weniger Code-Zeilen



Viele Dateien



Tausende Code-Zeilen

http://www.weizhewei.com/papers/SIGMOD18.pdf