

Algorithmen und Programmierung

Skript zur Vorlesung

Prof. Dr.-Ing. habil. Matthias Werner

Stand: 22. November 2021



Dieses Skript wurde mit L^AT_EX unter Nutzung verschiedener Klassen/Pakete erstellt.
Eigene Grafiken sind mit TikZ/pgf, OmniGraffle oder Affinity Designer erstellt.

Benutzte Versionen:

- pdfT_EX: 3.141592653-2.6-1.40.23 (TeX Live 2022/dev) kpathsea version 6.3.4/dev
- L^AT_EX-Format: 2021-06-01
- KOMA-Script: 2021/06/25 v3.34
- Beamer (beamerarticle): 2021/10/18 v3.64
- TikZ/pdf: 3.1.9a

Dieses Skript unterliegt der Creative Commons-Lizenz CC BY-NC-SA (i s c) 3.0.

Dies bedeutet im Wesentlichen, dass Inhalte dieses Skriptes für nichtkommerzielle Zwecke genutzt werden können, wenn das abgeleitete Werk den Autor dieses Skriptes nennt und sein Werk unter gleichen Bedingungen weitergibt.

Details unter <http://creativecommons.org/licenses/by-nc-sa/3.0/de/>

Inhaltsverzeichnis

1. Einführung	1
1.1. Worum geht es?	1
1.2. Ziele, Vorgehen und Herausforderungen	2
1.3. Literatur	6
1.4. Knobeln & Denksport	10
2. Einführung in Algorithmen	11
2.1. Beispiele	11
2.2. Algorithmenbegriff	21
2.3. Elemente der Datenmanipulation	28
2.4. Konstruktion von Algorithmen	32
3. Einführung in Programmiersprachen	35
3.1. Grundsätzliches	35
3.2. Geschichte	38
3.3. C am Beispiel	43
3.4. Der erste (richtige) Algorithmus	57
3.5. Python	61
3.6. Fehler	66
4. Typen	75
4.1. Einführung	75
4.2. Grundtypen	78
4.3. Typenkonvertierung	87
5. Variablen, Speicher und Zeiger	91
5.1. Speicher	91
5.2. Benannte Variablen	92
5.3. Adressen	98
5.4. Anonyme Variablen	102
5.5. Parameter	106
6. Abgeleitete Typen	113
6.1. Verbundtyp	114
6.2. Aufzählungstyp	118
6.3. Arrays	119
6.4. Parameter von main()	128
6.5. Komplexe Typen in Python	130
6.6. Nochmals Speicherklassen und -attribute	134

7. Iterationen	137
7.1. Einführung	137
7.2. Schleifen in C und Python	139
7.3. Iteration und Rekursion	149
8. Entwurf Algorithmen	153
8.1. Idee	153
8.2. Spezifikation	162
8.3. Beispiel: Ferienplanung	165
9. Logik und Automaten	173
9.1. Boolesche Algebra	173
9.2. Logik in C	178
9.3. Automaten	192
10.Syntax	199
10.1. Einführung	199
10.2. Formalisierung	201
10.3. Anwendung	209
11.Analyse von Algorithmen	217
11.1. Einführung	217
11.2. Partielle Korrektheit	223
11.3. Terminierung	226
11.4. Komplexität	229
11.5. Die $\mathcal{O}$ -Notation	233
12.Von Türmen und Damen - Rekursion und Backtracking	243
12.1. Einführung	243
12.2. Fibonacci-Zahlen	247
12.3. Türme von Benares	250
12.4. Backtracking	253
13.Dateien und Textsuche	261
13.1. Einführung	261
13.2. Dateien	262
13.3. Einfache Suche	270
13.4. Wildcards	277
14.Let's play	293
14.1. Einführung	293
14.2. Hauptschleife & E/A	299
14.3. Strategie	305
15.Linker und Bibliotheken	317
15.1. Wiederholung: Module und Linker	317
15.2. Linker	320
15.3. Bibliotheken	325
15.4. Fallstudie: Taschenrechner	329

16. Besserer Code	341
16.1. Layout	342
16.2. Dokumentation	347
16.3. Portabilität	359
16.4. Über den Tellerrand	363
 Anhang	 367
A. Programmiersprachen allgemein	369
A.1. Galerie von Computerpionieren	369
A.2. Wenn Programmiersprachen Filmcharaktere wären...	370
A.3. Zeittafel	370
 B. Die Sprache C	 391
B.1. Schlüsselwörter	391
B.2. Syntax	392
B.3. Operatorenvorrang und Assoziativität	411
B.4. Aus- und Eingabeformatierung	411
 C. Python	 415
C.1. Schlüsselwörter	415
C.2. Syntax	415
 D. Code	 437
D.1. Textsuche	437
D.2. Tic Tac Toe	445
D.3. Taschenrechner	453
 Literatur	 471
 Index	 473

Danksagung

Ein Skript wie dieses ist sicher niemals vollständig und niemals fehlerfrei. Alle Fehler in diesem Text sind mir zuzurechnen.

Jedoch möchte ich mich bei folgenden eifrigen Lesern dafür bedanken, dass sie mich auf Fehler aufmerksam gemacht und damit dazu beigetragen haben, dieses Skript zu verbessern:¹

- Jens Pönisch
- Mario Haustein
- Michael Kunz
- Martin Däumler
- Alexej Schepeljanski
- Robert Baumgartl
- Tobias Loose
- Marco Seidel
- Andreas Löscher
- Alexander Pöllmann
- Martin Zimniak
- Henrik Kretzschmar
- Robert Kiesel
- Peter Tröger
- Thomas Oechsle
- Kai Bergermann
- Colin Pohle
- Jennifer Hofmann
- Nils Trampel
- Wolf Weidner
- Marianne Ziesmer
- Kevin Matthes
- Tobias Morgenstern

¹Namen in der Reihenfolge der „Erstmeldung“

Vorwort

Dieses Skript wurde von mir begleitend zur der Vorlesung „Algorithmen und Programmierung“ an der TU Chemnitz in den Semestern 2010/2011 und 2011/2012 erstellt. Es handelt sich im Wesentlichen nicht um einen eigenständigen Text, sondern um die aneinandergereihten Folien der Vorlesung, ergänzt um Zwischentexte, Anhänge und nicht zuletzt um die illustrierenden xkcd-Comics von Randall Munroe. In diesem Vorwort möchte ich zwei didaktische Entscheidungen erläutern, die den Aufbau der Vorlesung und somit auch dieses Skriptes maßgeblich beeinflusst haben.

In Kursen über Algorithmen ist es in der Regel üblich, entweder zuerst eine Programmiersprache zu vermitteln, um danach zu Betrachtungen von Algorithmen im Allgemeinen überzugehen; oder mit letzteren anzufangen und die Programmierung (falls sie überhaupt zum Kurs gehört und nicht nur in Pseudocode erfolgt) nachzuschieben. Ich habe mich für einen dritten Weg entschieden: einzelne abstrakte Betrachtungen zu Algorithmen wechseln sich mit Elementen der Programmierung ab; beides ist miteinander verzahnt. So wird z. B. das `switch`-Statement in C im Laufe der Betrachtungen zu Automaten eingeführt. Ziel dieses Vorgehens ist die engere Verknüpfung von Theorie und Praxis, so dass die Theorie leichter **durch** die Praxis motiviert und schneller **in** der Praxis angewendet werden kann. Außerdem sollen lange „Trockenstrecken“ vermieden werden.

Eine zweite Entscheidung war, neben der durch die Curricula der Fakultät für Informatik der TU Chemnitz vorgegebene Programmiersprache C auch Elemente der Sprache Python in diesem Kurs zu präsentieren. Hier bestand das Ziel darin zu verdeutlichen, welche Konzepte der (imperativen) Programmierung von der benutzten Programmiersprache unabhängig sind und über C hinaus Gültigkeit besitzen, und welche lediglich dem Sprachdesign von C geschuldet sind. Entsprechend dient dieser Kurs **nicht** dazu, den Studentinnen und Studenten Python beizubringen: Viele wesentliche Elemente von Python wie z. B. die Objektorientierung finden keine Erwähnung. Vielmehr sollen Lernende ein Gespür für die Möglichkeiten von Programmiersprachen entwickeln und ermutigt werden, sich weitere anzueignen.

Auch wenn es für den Kursteilnehmer durch diese beiden Entscheidungen zur Didaktik vielleicht etwas schwerer ist, gezielt Informationen zu dem einen oder anderen Thema „herauszupicken“, hoffe ich, dass die Entscheidungen im Gegenzug dazu beitragen, das gesamtheitliches Verständnis der Algorithmen und Programmierung zu verbessern.

Vorwort 2016

Nach ein paar Jahren erhält das Skript zu „Algorithmen und Programmierung“ zum Wintersemester 2016/2017 eine Aktualisierung, die im Wesentlichen den Entwicklungen der Sprache C geschuldet ist. Zum einen hat sich neben dem GNU-Compiler der CLANG-Compiler mit seiner moderneren Architektur etabliert, so dass er in diesem Skript ab diesem Semester als Standardcompiler eingesetzt wird.

Zum anderen hat sich der C99-Standard (insbesondere die C++-artigen Kommentare und der entspanntere Umgang mit Variablendeklarationen) weitgehend durchgesetzt. Daher wird bei allen Beispielen die C99-Sprachvariante genutzt, ohne auf alle Feinheiten des C99-Standards einzugehen². Wir werden also überall dort C99-Features nutzen, wo diese „bequemer“ sind. Außerdem wird auf Inkompatibilitäten zwischen den Sprachvarianten besonders hingewiesen.

Des Weiteren wurden einige Kapitel gestrafft, während an anderer Stelle ausführlichere Erläuterungen eingefügt wurden – beides in der Hoffnung, dass damit die Verständlichkeit des Stoffes gesteigert wird.

Chemnitz, September 2016

Matthias Werner

Vorwort 2021

Zum (Corona-)Wintersemester 2021/2022 haben sich wieder ein paar Änderungen angesammelt. Neben kleineren Korrekturen und Ergänzungen gab es vor allem eine strukturelle Änderung: Die ausführlichere Diskussion über das Vorgehen beim Algorithmen- und Programmentwurf, die die Eingangsdiskussion zu dieser Thematik im Kapitel 2 vertieft, wurde abgegrenzt und leicht ausgebaut als ein eigenes Kapitel nach der Iteration (Kapitel 7) eingeschoben. Dafür wurden die qualitativen Aspekte (Programmkorrektheit) und die quantitativen Aspekte (Komplexität) der Bewertung von Algorithmen und Programmen zu einem Analyse-Kapitel (Kapitel 11) zusammengefasst.

Chemnitz, September 2021

Matthias Werner

²C99 hat viele Eigenheiten eingeführt, die nur bei wenigen, speziellen Architekturen wirksam werden (z.B. Paddingbits).

Kapitel 1

Einführung in die Lehrveranstaltung

And programming computers was so fascinating. You create your own little universe, and then it does what you tell it to do.

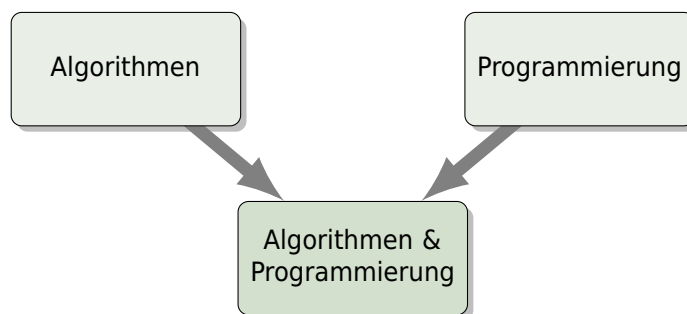
— Vint Cerf

Computer Science is embarrassed by the computer.

— Alan J. Perlis

1.1 Worum geht es?

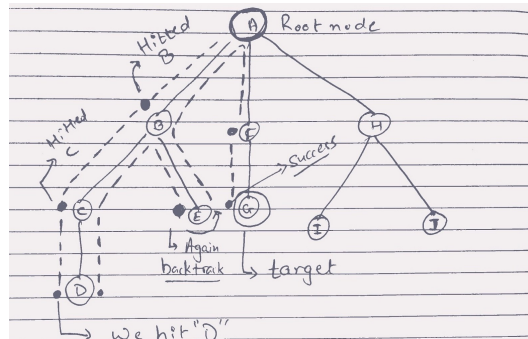
- Diese Lehrveranstaltung berührt zwei große Gebiete¹



Algorithmen vs. Programmierung

- Der Entwurf von **Algorithmen** ist vollständig unabhängig von Computern und verlangt das **Verstehen** von Problemen, **Abstraktionsvermögen**, **Mathematikkenntnisse**, sowie **Kreativität**

¹Wie überraschend! 😊



- **Programmierung** ist (in der praktischen Umsetzung) an Computer gebunden und verlangt das (abstrakte) Verständnis der Vorgänge im zu programmierenden Computer, Kenntnisse der **Syntax** und **Semantik**² von Programmiersprachen, sowie Beherrschung der **Entwicklungstools**.

```

1 data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
2
3 traverseDF :: Tree a -> [a]
4 traverseDF Empty = []
5 traverseDF (Node a l r) = a : (traverseDF l) ++ (traverseDF r)

```

Bitte seien Sie sich stets bewusst, dass Algorithmen und Programmieren zwei zwar (in diesem Kurs) eng verknüpfte, aber **unterschiedliche** Dinge sind!

- Wenn Sie in diesem Kurs erfolgreich sein wollen, müssen Sie die Fähigkeiten entwickeln...
 - beide Bereiche auseinander zu halten und zwischen beiden Bereichen gedanklich „umschalten“ **← d.h. jedes Problem in seinem Bereich lösen!**
 - beide Bereiche miteinander zu verknüpfen **← d.h. Abstraktionsebenen zu wechseln und die Auswirkungen auf die andere Ebene zu berücksichtigen**

1.2 Ziele, Vorgehen und Herausforderungen

1.2.1 Ziele

- Nach erfolgreichem Absolvieren dieser Lehrveranstaltung sollten Sie...
 - ... in Abstraktionen und Algorithmen denken können;
 - ... einige grundlegende Algorithmen kennen und verstehen;
 - ... grundsätzliche Ansätze für algorithmische Lösungen kennen und anwenden können;
 - ... die imperative Programmiersprache C grundsätzlich (und ein bisschen Python) beherrschen;

²Diese Begriffe werden wir später noch genauer untersuchen.

- ... Algorithmen entwickeln und programmieren können;
- ... Algorithmen und Programme bewerten können;
- ... mit den wichtigsten Entwicklungswerkzeugen (Editor, Compiler, Linker, Debugger) umgehen können;
- ... kleinere Probleme mit Hilfe von C lösen können;
- Zusätzlich sollten Sie natürlich jede Menge mehr oder weniger nützliches Wissen angesammelt und anwendungsbereit parat haben.

Hinweis

- In diesem Kurs sollen Sie **theoretische** Kenntnisse sowie **technische** und **praktische** Fähigkeiten erwerben
- Der wichtigste ist dabei das **Selbststudium**

Merke:

Programmieren kann man **nur** durch Programmieren lernen!
(Mit leichten Abstrichen gilt dies analog beim Algorithmen)

1.2.2 Vorgehen

- In diesem Kurs werden (u.a.) zwei didaktische Prinzipien genutzt:
- Algorithmische und Programmierkonzepte werden eng verzahnt behandelt
 - + Schulung des Wechsels der Abstraktionsebenen
 - + besseres Verstehen
 - schlecht zum Nachschlagen
 - ungünstig beim Auswendiglernen
- Einige Gebiete werden wiederholt erörtert und vertieft
 - + "Wiederholung ist die Mutter der Weisheit"
 - + besseres Verstehen
 - schlecht zum Nachschlagen
 - "Lying for kids" ist manchmal notwendig
- Zum Nachschlagen nutzen Sie bitte Literatur eigener Wahl (► Literaturliste, Abschnitt 1.3)
- Nach allgemeiner Erfahrung sind die Konzepte, die in diesem Kurs die größten Schwierigkeiten bereiten

- **Rekursion** im Algorithmenentwurf
- **Zeiger** (Pointer) beim Programmieren.

[The] ability to understand pointers and recursion is directly correlated with the ability to be a great programmer.

– JOEL SPOLSKY

- Versuchen Sie, diese Konzepte so früh wie möglich zu verstehen und lernen Sie, sicher damit umzugehen
 - Beide Konzepte sind im Grunde sehr einfach, die Schwierigkeit besteht eher in der nicht den Alltagserfahrungen entsprechenden³ Denkweise
 - Um Sie darin zu unterstützen, werden diese Konzepte recht früh im Kurs eingeführt und dann wiederholt angewendet

1.2.3 Herausforderungen

- Algorithmen und Programme zu entwickeln macht Spaß!
- Es sind aber (gerade am Anfang) ein paar **Klippen** zu umschiffen, um Frustration zu vermeiden
- Daher wollen wir hier die Ziele und Schwierigkeiten etwas genauer diskutieren...

Klippen

- Einige (eigentlich sehr positive) Eigenschaften können unter Umständen Ihren Erfolg in diesem Kurs gefährden.
- Sich dieser Gefahren bewusst zu werden kann helfen, entsprechende Fehler zu vermeiden
- Prüfen Sie sich selbst und treffen Sie ggf. Vorkehrungen, damit Sie nicht an diesen Klippen scheitern!
- **Sie beherrschen bereits eine andere Programmiersprache.**
 - Unterschiedliche Programmiersprachen nutzen unterschiedliche Konzepte, z.T. haben diese Konzepte aber sogar den gleichen Namen.
 - In C muss man sich um einige Dinge (z.B. Speicherverwaltung) kümmern, die in anderen Sprachen versteckt sind
 - Achten Sie auf die Unterschiede!
- **Sie beherrschen bereits C und alles ist Ihnen zu einfach.**
 - Manchmal ist es tatsächlich so – in diesem Fall: Herzlichen Glückwunsch ➡ suchen Sie sich schwierigere Herausforderungen!

³...aber informatik-typischen

- Häufig aber verpassen Sie neue Aspekte (z.B. beim Algorithmieren), weil Ihnen alles so bekannt vorkommt.
- Erweitern Sie ihr Verständnis!
- **Sie haben eine rasche Auffassungsgabe und sind es (aus der Schule) gewohnt, alles sofort (d.h. direkt im Unterricht) zu lernen.**
 - Zum Programmieren und Algorithmieren gehört auch **praktische Erfahrung**. Diese kann nicht durch Auswendiglernen erworben werden.
 - Erwerben Sie Praxiserfahrungen!
- **Sie sind es (aus der Schule) gewohnt, sich alles erst mal anzuhören und dann später (evtl. erst vor der Prüfung) im Zusammenhang zu lernen bzw. zu begreifen.**
 - Die Lernkurve an einer Universität ist in der Regel **steiler** als z.B. in der Schule.
 - Bei neuen Stoff wird das Verständnis des bisherigen vorausgesetzt
 - Wenn Sie etwas nicht verstehen, haken Sie sofort nach, sonst verlieren Sie den Anschluss!
- **Sie sind es gewohnt, an Dinge immer ganz konkret heranzugehen.**
 - Sie versuchen Abstraktionen zu vermeiden.
 - Durch die enge Verknüpfung von abstrakten und konkreten Konzepten führt eine solche Vermeidung zu einer schnellen Frustration auch (und gerade) bei konkreten Arbeiten.
 - Überwinden Sie Ihre Abscheu: **Abstraktionen sind Ihre Freunde!**
- **Sie sind (z.B. aus einem früheren Studium) zwar an abstraktes Denken, aber nicht an formales Denken gewöhnt.**
 - Unkonkrete Abstraktionen können Sie schnell **in die Irre** führen.
 - Letztendlich machen Computer genau das, was Sie ihnen sagen, nicht das, was Sie meinen.
 - Trainieren Sie präzises und formales Denken!
- **Sie sind es gewohnt, Dinge schnell in großen Zusammenhängen zu verstehen.**
 - Mitunter werden dabei die **präzisen Details vernachlässigt**, diese sind wegen des Zusammenspiels der verschiedenen Ebenen aber wichtig
 - Lernen Sie, auf die Details zu achten!
- **Sie sind es gewohnt, stets das volle Lehrangebot auszuschöpfen.**
 - Jede/r hat einen eigenen Lernstil, daher stellen wir Ihnen eine Vielzahl von Möglichkeiten zur Verfügung⁴, die Sie nutzen können.
 - Es ist Ihre Aufgabe, sich **das für Sie Passende und Wirksame** herauszusuchen – alles zu nutzen, könnte den einen oder die andere **überfordern**.

⁴mehr dazu später...

- Finden Sie das für sich Richtige heraus, und trauen Sie sich, anderes wegzulassen – am Ende zählt nur das Erlernte (in der Prüfung)

Tipp

Wenn Sie entdecken, dass Sie auf eine (oder mehrere) dieser Klippen zusteuern oder mit anderen Schwierigkeiten konfrontiert sind, agieren Sie sofort:

- Sprechen Sie mit den Lehrenden, z.B. in den Sprechstunden
- Fragen Sie „dienstältere“ Studierende
- Gehen Sie zum Fachschaftsrat
- Überprüfen Sie Ihren Lernstil und ändern ihn gegebenenfalls

1.3 Literaturempfehlungen

- Diese Veranstaltung folgt **keinem einzelnen** Lehrbuch⁶; folgende kommentierte Liste gibt daher allgemein Literatur an, die ggf. nützlich sein kann
 - Anmerkung: Die Kommentare sind z.T. subjektiv
- Jede(r) kann (und sollte) sich die Literatur suchen, die ihr/ihm liegt
- Wenn jemand der Überzeugung ist, ein besonders gutes Buch gefunden zu haben, bin ich für Empfehlungen dankbar (wenn möglich mit Angabe der Gründe, **warum** dieses Buch gut ist)

Funktion/Bewertung von Algorithmen

 [AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983

 [AHU75] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Design and Analysis of Computer Algorithms*. Addison-Wesley, 1975

Anm: Sind (oder waren zumindest jahrelang) an vielen US-Universitäten die Standardlehrbücher für Studenten im Grund- [AHU83] bzw. Hauptstudium [AHU75].

 [Cor+01] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd ed. MIT Press, 2001

Anm: Die wahrscheinlich umfangreichste Algorithmensammlung in einem einzelnen Buch; mathematisch fundiert; gut als Nachschlagewerk

 [Sed90] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1990

⁶falls Sie dieses Skript nicht als ein Lehrbuch auffassen

Anm: *Das Buch diskutiert ein breites Spektrum von Algorithmen (wenn auch nicht ganz der Umfang von [Cor+01] erreicht wird). Verzichtet wird auf (zuviel) Mathematik, aber auch auf Aspekte des Algorithmenentwurfs (siehe nächster Abschnitt).*

Neben [Sed90] gibt es vom gleichen Autor noch die Bücher [Sed83] und [Sed03] Alle drei Bücher sind ähnlich und unterscheiden sich im Wesentlichen nur in der Programmiersprache. [Sed83] benutzt Pascal/Pseudocode, [Sed03] Java.

Achtung: Viele Fehler (zumindest in älteren Ausgaben), Errata beachten!

 [Knu69] Donald E. Knuth. *Fundamental Algorithms*. Vol. 1. The Art of Computer Programming. Addison-Wesley, 1969

 [Knu81] Donald E. Knuth. *Seminumerical Algorithms*. Vol. 2. The Art of Computer Programming. Addison-Wesley, 1981

 [Knu98] Donald E. Knuth. *Sorting and Searching*. Vol. 3. The Art of Computer Programming. Addison-Wesley, 1998

Anm: *Donald Knuth schreibt seit über 40 Jahren an einem Standardwerk über Informatik, das auf 5 – 7 Bände angelegt ist und dessen erste drei Bände (plus einiger „Vorschauhefte“ auf Band 4 und Korrekturhefte zu den Bänden 1 – 3) bereits erschienen sind.*

Knuth ist für seine Präzision bekannt. Mathematische Betrachtungen können ggf. übersprungen werden. Benutzte Programmiersprache ist ein künstlicher Assembler.

Entwurf von Algorithmen

*Die folgenden Bücher legen besonderen Wert auf den **Entwurf** von Algorithmen – ein Thema, welches in vielen Algorithmen-Büchern zu kurz kommt.*

 [Ski08] Steven S. Skiena. *The Algorithm Design Manual*. 2nd ed. Springer, 2008

Anm: *Unterhaltsam durch häufigen Bezug auf interessante Probleme aus dem wahren Leben™ („war stories“).*

 [Man89] Udi Manber. *Introduction to Algorithms. A Creative Approach*. Addison-Wesley, 1989

Anm: *Autor nutzt ein induktives Vorgehen beim Algorithmenentwurf. Er stellt auch falsche Lösungen vor, um sukzessive zum korrekten Entwurf zu kommen.*

 [Edm08] Jeff Edmonds. *How to Think About Algorithms*. Cambridge University Press, 2008

Anm: *Besonderer Schwerpunkt auf Schleifeninvarianten und Rekursion*

Programmier Sprache C

Bücher über C gibt es in Hülle und Fülle. Deshalb gibt es hier nur eine sehr kleine Auswahl.

Nutzen Sie ruhig dasjenige Buch, das Ihnen am besten liegt. Achten Sie aber darauf, dass es mindestens die ANSI-Variante von C enthält.

 [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Prentice Hall, 1988

Anm: *Das Standardwerk über C, von den Erfindern der Sprache.*

- 📖 [Oua97] Steve Oualline. *Practical C*. 3rd ed. O'Reilly, 1997
 Anm: *Typisches O'Reilly-Buch: Pragmatisch, knapp im Stil, ausführlich im Umfang.*
- 📖 [Wol16] Jürgen Wolf. *Grundkurs C: C-Programmierung verständlich erklärt*. 2nd ed. Rheinwerk Computing, 2016
 Anm: *Das aktuellste der hier aufgelisteten Bücher, enthält auch C11*
- 📖 [Reg03] Regionales Rechenzentrum Niedersachsen, ed. *Die Programmiersprache C. Ein Nachschlagewerk*. 18th ed. Universität Hannover. 2003
 Anm: *Billig (3,50 €); erhältlich über das Rechenzentrum, siehe https://www.tu-chemnitz.de/urz/handbuecher_intern.html#bestellformular*
- 📖 [Sch14] Helmut O.B. Schellong. *Moderne C-Programmierung*. 3rd ed. Springer, 2014
 Anm: *Als E-Book über <http://dx.doi.org/10.1007/978-3-642-54437-8> für TU-Studenten kostenlos*
- 📖 [Goo04] Dan Gookin. *C For Dummies*. 2nd ed. Wiley Publishing, 2004
 Anm: *Falls sonst nichts mehr hilft.*
- Programmiersprache Python**
Auch über Python gibt es zahlreiche Bücher – auch hier deshalb nur eine kurze Auswahl.
- 📖 [Zel03] John M. Zelle. *Python Programming: An Introduction to Computer Science*. Franklin Beedle & Associates, 2003
- 📖 [Chu06] Wesley J. Chun. *Core Python Programming*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006
 Anm: *Die Empfehlung von Jens Pönisch 😊*
- 📖 [Lut13] Mark Lutz. *Learning Python*. 5th. O'Reilly, 2013
 Anm: *Für TUC-Angehörige kostenlos als E-Book über die Bibliothek verfügbar: <https://katalog.bibliothek.tu-chemnitz.de/Record/0008951133>.*
- Andere Programmiersprachen**
Hier sind nur Bücher gelistet, die keine reinen Programmiersprachelehrbücher sind, sondern (auch) Hintergrundwissen über Algorithmenentwurf etc. bieten.
Jedoch sind sie so stark sprachgebunden, dass sie nicht in den allgemeinen Algorithmen-Teil gehören.
- 📖 [HW94] Brian Harvey and Matthew Wright. *Simply Scheme: Introducing Computer Science*. MIT Press, 1994
 Anm: *Didaktisch sehr gut, z.T. unterhaltsam. Sprache: Scheme*
- 📖 [Wir86] Niklaus Wirth. *Algorithms & Data Structures*. Prentice-Hall, 1986
 Anm: *Niklaus Wirth ist der Guru der Strukturierten Programmierung. Sprache: Modula-2*
- 📖 [Har97c] Brian Harvey. *Symbolic Computing*. 2nd ed. Vol. 1. Computer Science Logo Style. MIT Press, 1997

 [Har97a] Brian Harvey. *Advanced Techniques*. 2nd ed. Vol. 2. Computer Science Logo Style. MIT Press, 1997

 [Har97b] Brian Harvey. *Beyond Programming*. 2nd ed. Vol. 3. Computer Science Logo Style. MIT Press, 1997

Anm: *Diese drei Bände präsentieren (nahezu) die komplette Informatik mit Hilfe der Programmiersprache Logo*

Vielleicht auch interessant

 [HLW06] Helmut Herold, Bruno Lurz, and Jürgen Wohlrab. *Grundlagen der Informatik*. Pearson Studium, 2006

Anm: *Auf über 700 Seiten werden theoretische, technische und praktische Grundlagen der Informatik präsentiert; gut als Nachschlagewerk.*

 [Are+15] Tilo Arens et al. *Mathematik*. 3rd ed. Spektrum Akademischer Verlag, 2015

Anm: *Auf über 1600(!) Seiten werden gut verständlich die Grundlagen der Mathematik dargestellt; allein im ca. ersten Viertel im Wesentlichen der Abiturstoff (der in dieser Vorlesung vorausgesetzt wird).*

 [Ben06] Mordechai Ben-Ari. *Understanding Programming Languages*. For personal use free available: <http://www.weizmann.ac.il/sci-tea/benari/books/upl.zip>. John Wiley & Sons, 2006

 [Mit02] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2002

Anm: *Beide Bücher betrachten Programmiersprachen im Vergleich.*

 [Har92] David Harel. *Algorithmics. The Spirit of Computing*. Addison-Wesley, 1992

Anm: *Feuilletonartige Betrachtung algorithmischer Probleme (was dem Einsteiger/Fachfremden das Lesen erleichtert, manche aber stört). Enthält ein Kapitel über Programmiersprachen und viele Bibelzitate.*

 [Bia09] Federico Biancuzzi. *Masterminds of Programming. Conversations with the Creators of Major Programming Languages*. O'Reilly, 2009

Anm: *Interviews mit den Schöpfern berühmter Programmiersprachen. Gibt Einblicke in die Motivation und Ziele beim Entwurf von Programmiersprachen.*

 [Ben00] Jon Bentley. *Programming Pearls*. 2nd ed. Addison-Wesley, 2000

 [Ben88] Jon Bentley. *More Programming Pearls. Confessions of a Coder*. 1st ed. Addison-Wesley, 1988

Anm: *Keine Lehrbücher oder Monographien, sondern ausgewählte Beiträge aus der Zeitschrift „Communications of the ACM“ über Probleme beim Entwurf von Software.*

Sehr unterhaltsam, trotzdem lehrreich.

Zur Beachtung

Außer [Sch14] und [HLW06] ist alle hier gelistete Literatur in englischer Sprache.

Von vielen Büchern existiert eine Übersetzung ins Deutsche. Seien Sie aber vorsichtig: Häufig enthalten die Übersetzungen inhaltliche Fehler!

Da Sie für Ihr Studium sowieso nicht um englische Literatur herumkommen, fangen Sie am besten gleich mit Originalliteratur an.

1.4 Knobeln & Denksport**Aufgabe 1.1**

Betrachten Sie die Reihe folgender Zahlen:

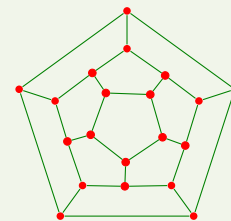
9 44 32 12 7 42 34 92 35 37 41 8 20 27 83 64 61 28 39 93 29 17 13 14 55 21 66 72 23
73 99 1 2 88 77 3 65 83 84 62 5 11 74 68 76 78 67 75 69 70 22 71 24 25 26

- (a) Streichen Sie so wenig Zahlen wie möglich, so dass die verbleibenden Zahlen eine aufsteigende Reihe bilden!
- (b) Lösen Sie die Aufgabe erneut, nur soll die Ergebnisreihe diesmal absteigend sein!
- (c) Beschreiben Sie, wie man vorgehen muss, um für eine beliebige Zahlenreihe zum optimalen Ergebnis bei den beiden vorstehenden Aufgaben zu kommen!

Aufgabe 1.2

Betrachten Sie die nebenstehende Figur.
Finden Sie einen geschlossenen Weg (d.h. Start = Ende) entlang der grünen Linien, so dass jeder rote Punkt genau einmal besucht wird!

Amerkung: Es muss *nicht* jede grüne Linie genutzt werden.



Kapitel 2

Einführung in Algorithmen

*Precept may point to us the way, but it is
silent continuous example, conveyed to us
by habits, and living with us in fact, that
carries us along.*

— Samuel Smiles

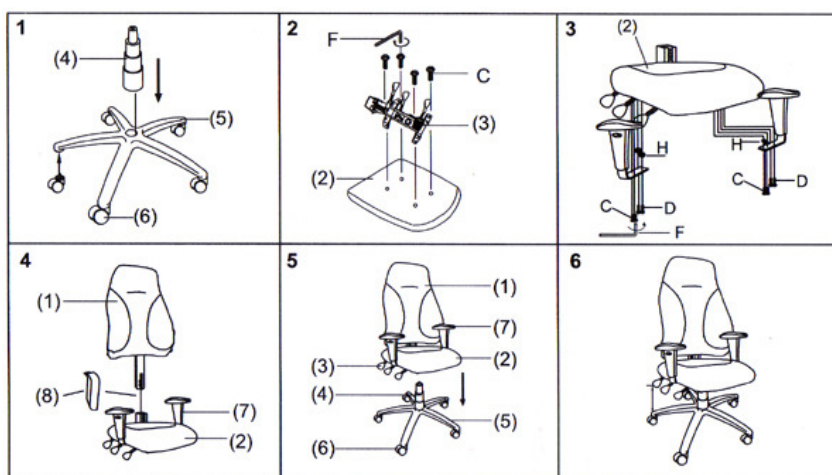
*We enjoy seeing our algorithms expressed
clearly and persuasively.*

— Samuel P. Harbison

2.1 Einführende Beispiele

Um ein intuitives Verständnis zu erreichen, betrachten wir zu Beginn eine Anzahl von Beispielen, die (mehr oder weniger) dem Alltag entnommen sind.

Beispiel 1: Bauanleitung



Quelle: *IKEA*

Beispiel 2: Rezept

Caipirinha

- **Zutaten:**

- 3 cl Zuckerrohrschnaps (Cachaca)
- 1 Limette
- 2 TL brauner Zucker
- crushed Eis

- **Zubereitung:**

- Limette in Achtelstücke schneiden, in ein Longdrinkglas geben und den braunen Zucker darüber streuen.
- Limettenstücke mit einem Holzmörser gut zerdrücken und mit dem Zucker vermischen
- Anschließend den Zuckerrohrschnaps darübergießen und das Glas mit crushed Eis auffüllen
- Gut mischen und mit einem oder zwei Trinkhalmen servieren.

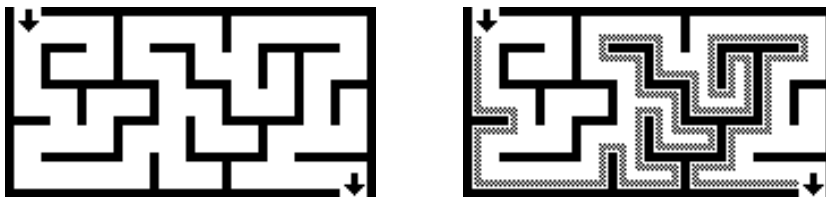


Beispiel 3: Quadratische Gleichung

- **Gegeben:** Gleichung $a \cdot x^2 + b \cdot x + c = 0$, wobei $a \neq 0$
- **Gesucht:** Alle x , die die gegebene Gleichung erfüllen

Algorithmus¹

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

Beispiel 4: Weg durch das Labyrinth

Regel

- Gehe immer so, dass Du mit der rechten Hand die Wand berührst!

¹Falls jemand es nicht mehr aus der Schule parat hat, hier die Herleitung:

$$\begin{array}{ll}
 ax^2 + bx + c = 0 & | -c \\
 ax^2 + bx = -c & | \cdot 4a \\
 4a^2x^2 + 4abx = -4ac & | + b^2 \text{ (quadratische Ergänzung)} \\
 (2ax)^2 + 2 \cdot 2axb + b^2 = b^2 - 4ac & | \text{Umformen mit binomischer Formel} \\
 (2ax + b)^2 = b^2 - 4ac & | \pm \sqrt{} \\
 2ax + b = \pm \sqrt{b^2 - 4ac} & | -b \\
 2ax = -b \pm \sqrt{b^2 - 4ac} & | : (2a) \\
 x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} &
 \end{array}$$

Beispiel 5: Schriftliche Addition

- **Problem:** Addiere zwei mehrstellige Zahlen
- **Algorithmus:**
 1. Schreibe die Zahlen rechtsbündig so hin, dass jeweils zwei Ziffern in Spalten untereinander stehen
 2. Beginne mit der Spalte, die am weitesten rechts ist
 3. Addiere die Zahlen in dieser Spalte
 4. Schreibe die Einerstelle der Summe unter die Spalte
 5. Schreibe die Zehnerstelle der Summe über die Spalte links der aktuellen
 6. Wiederhole ab Schritt (3) mit nächster Spalte links der aktuellen, bis keine Spalte mehr übrig ist
 7. Die unterste Zeile ist das Ergebnis

$$\begin{array}{r}
 1 1 0 \\
 7 8 2 \\
 3 4 1 \\
 \hline
 1 1 2 3
 \end{array}$$

Beispiel 6: Magisches Quadrat**Hexeneinmaleins**

Du mußt versteh'n!
 Aus Eins mach Zehn,
 Und Zwei laß geh'n,
 Und Drei mach gleich,
 So bist Du reich.
 Verlier die Vier!
 Aus Fünf und Sechs,
 So sagt die Hex',
 Mach Sieben und Acht,
 So ist's vollbracht

GOETHE, Faust I

10	2	3
0	7	8
5	6	4

Beispiel 7: Bruch kürzen

- **Problem:** Kürze einen Bruch $\Rightarrow$ teile Zähler und Nenner durch ihren größten gemeinsamen Teiler (ggT)
- **Neues Problem:** Finde den größten gemeinsamen Teiler zweier Zahlen
 - Wenn; x und y **gleich** sind, so hat der größte gemeinsame Teiler den Wert x (oder y) $\Rightarrow x$

- Wenn x und y **ungleich** sind, so ist ihr größter gemeinsamer Teiler gleich dem größten gemeinsamen Teiler vom kleineren der beiden Werte und der Differenz zwischen dem größeren und dem kleineren $\Rightarrow \text{ggT}(\min(x, y), \text{abs}(x - y))$

-
- **Beispiel:** Kürze $\frac{15}{25} \Rightarrow \frac{3}{5}$

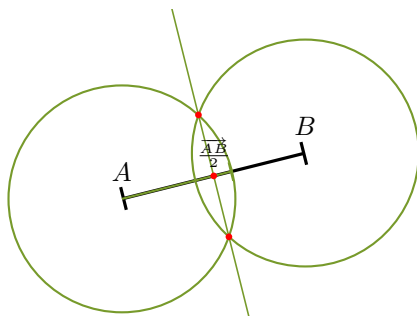
$$\begin{aligned}
 \text{ggT}(15, 25) &= \text{ggT}(\min(15, 25), \text{abs}(15 - 25)) = \text{ggT}(15, \text{abs}(-10)) \\
 &= \text{ggT}(15, 10) = \text{ggT}(\min(15, 10), \text{abs}(15 - 10)) \\
 &= \text{ggT}(10, 5) = \text{ggT}(\min(10, 5), \text{abs}(10 - 5)) \\
 &= \text{ggT}(5, 5) = 5
 \end{aligned}$$

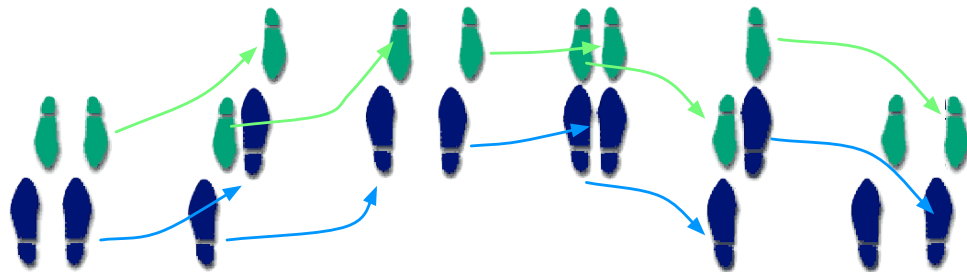
Der Algorithmus zum Bestimmen des größten gemeinsamen Teilers stammt von Euklid (Euklid von Alexandria, ca. 360 v. Chr. bis ca. 280 v. Chr.) und gilt als der älteste (nichttriviale) Algorithmus. Er war von Euklid ursprünglich als Aussage über geometrischen Stecken gedacht und lautete etwa:

„Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.“

Beispiel 8: Geometrische Streckenhalbierung

- **Problem:** Halbieren Sie eine gegebene Strecke nur mit Hilfe eines Zirkels und eines (unbemaßten) Lineals
- **Algorithmus:**
 1. Stelle am Zirkel eine Entfernung ein, die deutlich größer als die halbe Strecke ist
 2. Zeichne mit dieser Einstellung um das eine Ende der Strecke einen Kreis
 3. Zeichne mit der gleichen Einstellung um das andere Ende der Strecke einen Kreis
 4. Verbinde mit dem Lineal die Schnittpunkte der Kreise mit einer Geraden
 5. Wo die Gerade die Strecke schneidet, befindet sich die Mitte der Strecke



Beispiel 9: Tanzschritte

Langsamer Walzer, Grundschrift

Beispiel 10: Musik**Beispiel 11: Ableitung einer Funktion**

- **Problem:** Finde die Ableitung nach x einer Funktion $f(x)$
- **Regeln:** ➔ Bilden zusammen einen Algorithmus!
 1. Ist $f(x)$ eine Summe so ist die Ableitung die Summe der Ableitungen der Summanden
 2. Ist $f(x)$ ein Produkt so ist die Ableitung die Summe des Produkts des ersten Faktors mit der Ableitung des zweiten Faktors und des Produkts des zweiten Faktors mit der Ableitung des ersten Faktors
 3. Ist $f(x)$ eine Funktion einer anderen Funktion von x , so ist die Ableitung das Produkt aus der Ableitung der äußeren Funktion nach der inneren Funktion und der Ableitung der inneren Funktion
 4. Ist $f(x)$ nicht von x abhängig, so ist die Ableitung 0
 5. Ist $f(x) = x^k$, so ist die Ableitung $k \cdot x^{k-1}$
 6. Ist $f(x) = a^x$, so ist die Ableitung $a^x \cdot \ln a$

7. Ist $f(x) = \sin(x)$, so ist die Ableitung $\cos(x)$

8. Ist $f(x) = \log_a(x)$, so ist die Ableitung $\frac{1}{x} \log_a e$

Man beachte:

Wie bei Beispiel 7 wird der Algorithmus von sich selbst aufgerufen (Regeln 1, 2 und 3)

Beispiel 12: Sortieren

- **Problem:** Sortiere folgende Spielkarten nach ihrem Wert beim Skat²:



- **Algorithmus:**

1. Man beginnt mit den beiden Karten, die am weitesten links liegen
2. Wenn die größere der beiden Karten rechts liegt, vertauscht man beide Karten
3. Gehe eine Karte nach rechts und wiederhole Schritt 2
4. Wenn man bei den beiden Karten ganz rechts angekommen ist, beginnt man wieder bei Schritt 1, falls mindestens einmal getauscht wurde

Runde 1:											getauscht: ✗
Runde 1:											getauscht: ✗
Runde 1:											getauscht: ✗
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓

²Etwas vereinfacht: Kartenwert über Farbwert, Kartenwertreihenfolge: B,A,10,K,D,9,8,7; Farbreihenfolge: Kreuz, Pik, Herz, Karo

Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 1:											getauscht: ✓
Runde 2:											getauscht: ✗
Runde 2:											getauscht: ✗
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											

Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 2:											getauscht: ✓
Runde 3:											getauscht: ✗
Runde 3:											getauscht: ✗
Runde 3:											getauscht: ✓
Runde 3:											getauscht: ✓
Runde 3:											getauscht: ✓
Runde 3:											getauscht: ✓
Runde 3:											getauscht: ✓
Runde 3:											getauscht: ✓
Runde 3:											getauscht: ✓
Runde 3:											getauscht: ✓
Runde 3:											getauscht: ✓
Runde 3:											getauscht: ✓

Runde 4:		getauscht: ✗
Runde 4:		getauscht: ✓
Runde 4:		getauscht: ✓
Runde 4:		getauscht: ✓
Runde 4:		getauscht: ✓
Runde 4:		getauscht: ✓
Runde 4:		getauscht: ✓
Runde 4:		getauscht: ✓
Runde 4:		getauscht: ✓
Runde 5:		getauscht: ✗
Runde 5:		getauscht: ✗
Runde 5:		getauscht: ✗
Runde 5:		getauscht: ✗
Runde 5:		getauscht: ✗
Runde 5:		getauscht: ✗
Runde 5:		getauscht: ✗
Runde 5:		getauscht: ✗

Runde 5:  getauscht: ✕

Runde 5:  getauscht: ✕

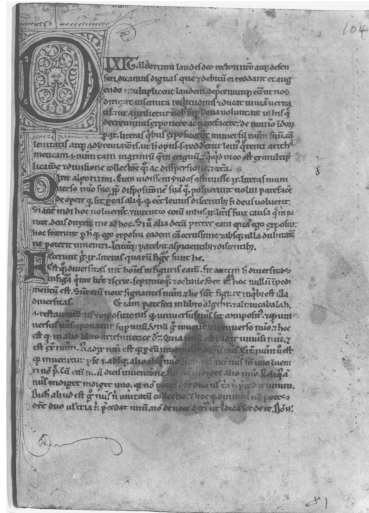
2.2 Begriff und Eigenschaften

- *ABU JAFAR MUHAMMAD IBN MŪSĀ AL-CHWĀRIZMĪ*, latinisiert: *ALGORISMUS*
 - Universalgelehrter 780 – ca. 842
 - Stammte aus **Chwarizm** (Gebiet in Zentralasien südl. des Aralsees)
 - Wirkte in Persien (heute Iran)
 - Führt die (indische) Null ins arabische Zahlensystem ein
 - Rechenregeln mit diesem modernen Zahlensystem legte er in einem Buch vor, dessen lateinische Version den Titel „**Algorismi** de numero indorum“ (Das Werk des Algorismus’ über indische Zahlen) trägt
 - Daher stammt der Begriff „Algorithmus“ für Rechenregeln, später allgemeiner gebraucht



Bildquelle: Alain Juhel

- Allerdings existieren schon vor al-Chwarizmi Rechenregeln (Algorithmen), vermutlich seit etwa 1800 v. Chr.



Seite aus dem lateinischen Text zu indischen Zahlen, die mit „DIXIT algorizmi“ (Algorithmus hat gesagt) beginnt.

Bildquelle: Cambridge, University Library, ff. 6.5.

Definition

- Es gibt verschiedene Formalisierungen des Algorithmen-Begriffs
- Benutzen zunächst recht allgemeinen Begriff³:

Definition 2.1: Algorithmus

Ein Algorithmus ist eine Handlungsvorschrift, die aus einer **endlichen** Anzahl von **wohldefinierten** und **effektiven** Einzelschritten besteht, und die dazu dient, ein bestimmtes Ziel zu erreichen.

- **Endliche Anzahl:** Die Beschreibung der Algorithmus besitzt endliche Länge ➔ **Finitheit**
- **Wohldefinierter Schritt:** Alle Schritte müssen eindeutig sein ➔ **Definiertheit**
- **Effektiver Schritt:** Alle Elemente müssen direkt ausführbar sein ➔ **Effektivität, Ausführbarkeit**

Finitheit

- Algorithmen können auf verschiedene Weise beschrieben werden ➔ siehe Beispiele weiter vorn
- **Aber:** Es muss stets gewährleistet sein, dass zur Darstellung nur eine Maximalmenge „Datenträger“ verbraucht wird
 - Text
 - Platz

³Es gibt derzeit keine allgemein akzeptierte formale Definition von „Algorithmus“.

- Größe Zeichnung
- etc.

Definiertheit

Betrachten folgenden „Algorithmus“ zur Übernahme von Google:

1. Gehe nach New York
2. Mache eine Million Dollar
3. Spekuliere an der Börse bis Du 1,6 Billionen Dollar zusammen hast
4. Kaufe alle Aktien von Alphabet Inc.⁴ auf

- Die Schritte 2 und 3 sind **nicht** genau definiert (und wahrscheinlich auch nicht definierbar)

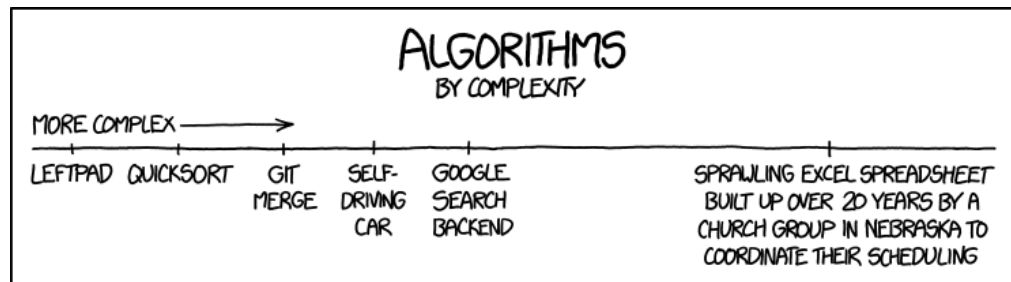
Ausführbarkeit

- Jeder Teil eines Algorithmus sollte **direkt** oder mit Hilfe eines **anderen Algorithmus** ausführbar sein
- **Test:** Prinzipiell muss alles mit Stift und Papier nachvollziehbar sein
- Betrachten folgenden Algorithmus zu Primzahlzwillingen:

1. Lese zwei Zahlen x_1 und x_2 ein
2. Wenn x_1 oder/und x_2 keine Primzahl ist, dann gebe „Keine zwei Primzahlen“ aus und gehe zu 5
3. Wenn $\text{abs}(x_1 - x_2) \neq 2$, dann gebe „Keine Primzahlzwillinge“ aus und gehe zu 5
4. Wenn x_1 und x_2 die größten Primzahlzwillinge sind, gebe „größte Primzahlzwillinge“ aus, sonst gebe „Primzahlzwillinge“ aus
5. Beende Algorithmus

- Alle Schritte sind exakt definiert
- **Aber:** Bis heute (Herbst 2021) weiß niemand, **ob** es ein größtes Primzahlzwillingspaar gibt oder **wie** man dieses ggf. bestimmt

⁴Alphabet Inc. ist die Muttergesellschaft aller Google-Unternehmen; Börsenwert vom März 2021, Quelle: (www.wikipedia.de)



Quelle: *xkcd* - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1667/>

Weitere Eigenschaften

Mitunter werden noch mehr Eigenschaften verlangt

- **Abstrahierung:** Der Algorithmus soll nicht nur ein Problem lösen, sondern eine Klasse von Problemen
- **Terminierung:** Nach einer endlichen Anzahl von Schritten ist der Algorithmus beendet
- **Determinismus:** Zu jedem Zeitpunkt gibt es höchstens eine Möglichkeit der Fortsetzung
- **Determiniertheit:** Unter den gleichen Anfangsbedingungen gelten am Ende stets die gleichen Endbedingungen
- **Dynamische Finitheit, Ressourcenbeschränkung:** Ein Algorithmus darf stets nur eine endliche Menge von Ressourcen benutzen

Merke

- Es gibt Algorithmen, die eine oder mehrere dieser Eigenschaften **nicht** haben
- Für uns sind (hier) aber nur die Algorithmen **interessant**, die abstrahierend, terminierend, determiniert, deterministisch und ressourcenbeschränkt sind
 (... und natürlich aus einer endlichen Menge von wohldefinierten und effektiven Schritten bestehen)

- **Außerdem:** Uns interessieren Algorithmen, die (vor allem) auf einem *Computer* ausgeführt werden

Betrachten diese Eigenschaften im Folgenden genauer

Computeralgorithmen

- **Computer:** der, *von lat.* („computare“ = zusammenrechnen) ⇒ *eng.* („to compute“ = rechnen)
- Computer können **Daten** manipulieren, aber **nicht** ohne weiteres Objekte der **realen** Welt

- Objekte der realen Welt benötigen eine **Daten-Repräsentation** (⇒ Abbildung)
- Abbildung erfolgt über Computerperipherie (Ein-/Ausgabegeräte)
- Eigentlicher Algorithmus betrachtet nur Daten
 - Struktur der Daten kann sehr komplex sein ⇒ siehe Vorlesung „Algorithmen und Datenstrukturen“

Abstrahierung

- Um den gleichen Algorithmus für eine Vielzahl von Fällen nutzen zu können, wird der Algorithmus **parameterisiert**
- Parameterisierung erfolgt über die **Eingabedaten**
- **Ergebnis** des Algorithmus wird in den **Ausgabedaten** dargestellt



- Dies entspricht dem Modell einer **mathematischen Funktion**:

$$A = f(E)$$

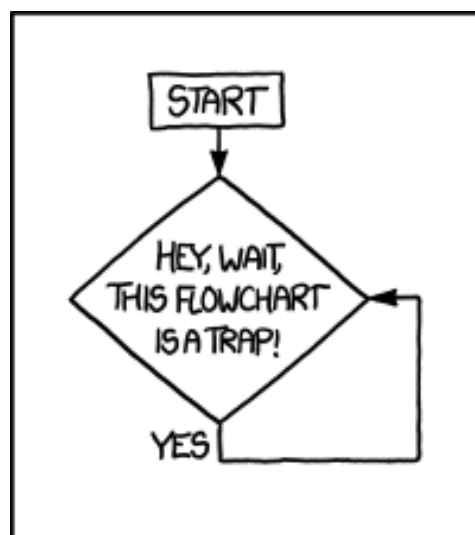
- E entspricht der Eingabe, A der Ausgabe, und f dem Algorithmus
- Durch die Äquivalenz mit einer Funktion wird ein Computeralgorithmus zu einem **Rechnungsproblem**
- Lässt sich alles berechnen (also programmieren)?
- **Nein!**
 - Z.B. gibt es keinen Algorithmus, der für zwei beliebige(!) andere Algorithmen berechnen kann, ob sie dieselbe Funktion berechnen
 - Es gibt noch nicht mal einen Algorithmus, der stets berechnen kann, ob ein beliebig gegebener anderer Algorithmus zum Ende kommt (terminiert) oder nicht
- Solche Probleme der Grenzen der Machbarkeit sind Gegenstand der *theoretischen Informatik*

Terminierung

Betrachten folgenden Algorithmus:

1. Gehe zu Schritt 2
2. Gehe zu Schritt 1
3. Beende Algorithmus

- Obwohl die Beschreibung des Algorithmus **endlich** ist, ist die Ausführung **nie beendet**
➔ Schritt 3 wird nie erreicht
- Falls ein Algorithmus Ausgaben produziert, weiß man bei einem nichtterminierenden Algorithmus nicht, wann die Ausgabe vollständig/gültig ist
- Es gibt nichtterminierende Algorithmen, die aber hier nicht betrachtet werden



Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/1195/>

Effizienz

- In der Praxis sollen Algorithmen nicht nur nicht unendlich lange laufen, sondern sogar **nicht sehr lange**
- Die Dauer der Ausführung wird als Gütekriterium eines Algorithmus angesehen
- Die Dauer hängt typischerweise von der **Größe des Problems** ab
 - **Beispiel:** Der Sortieralgorithmus aus Beispiel 12 dauert für 20 Spielkarten in der Regel länger als für 10 Karten

- Um den Algorithmus unabhängig von der ausführenden Hardware beurteilen zu können, wird meist ein normiertes Maß benutzt:
 - ➔ **(Zeit-)Komplexität**
 - Später mehr darüber (➔ Kapitel 10)

Determinismus und Determiniertheit

Betrachten folgenden Algorithmus:

1. Lasse Dir eine Zahl geben
2. Werfe einen Würfel
3. Wenn der Würfel mehr als 3 anzeigt, multipliziere die Zahl mit 2 und gehe zu Schritt 2
4. Gib die Zahl aus

- Dieser Algorithmus ist **nicht** deterministisch: Es ist nicht (a priori) klar, welcher Schritt nach Schritt 3 ausgeführt wird
- Er ist auch nicht determiniert: Auch bei gleichem Eingabewert können unterschiedliche Ausgangswerte vorkommen
- Nichtdeterministische Algorithmen werden auch **randomisierte** Algorithmen genannt
- Ein deterministischer Algorithmus kann mit einem Zufallswert „gefüttert“ werden

Merke

- Ein terminierender und deterministischer Algorithmus ist **stets** auch determiniert
- **Aber:** Das Ergebnis eines randomisierten Algorithmus kann auch determiniert sein

- Randomisierte Algorithmen bilden eine wichtige Klasse von Algorithmen
- Mitunter sind sie schneller oder effizienter als deterministische Algorithmen
 - ➔ mehr bei PROF. GOERDT

Ressourcenbeschränkung

- Theoretisch kann ein Algorithmus *fast* unendlich viel Ressourcen nutzen, in der Praxis steht immer nur eine beschränkte Anzahl zur Verfügung
- Betrachten folgenden Fall: Sie erhalten einen Kettenbrief mit 8 Namen und folgenden Instruktionen:

Schicke an die Person, die oben auf der Liste steht, einen Eurocent.

Nun streiche den obersten Namen und füge Deinen unten an die Liste an. Dann fertige 12 Kopien dieses Briefes und schicke ihn an 12 Menschen, die noch nicht an diesem Spiel teilgenommen haben.

Nach einiger Zeit wirst Du Eurocents im Wert von fast 5 Millionen Euro erhalten.

- Selbst, wenn derjenige, der sich dieses Spiel ausgedacht hat, Ihnen einen der ersten zwölf Briefen geschickt hat, wären 5628851293 (ca. 5,6 Milliarden⁵) verschiedene Menschen nötig, die sich am Spiel beteiligen, damit dieses Schneeballsystem funktioniert und Sie ihren „Gewinn“⁶ erhielten
- Das Beispiel zeigt, dass der Ressourcenverbrauch auch scheinbar einfacher Algorithmen sehr schnell steigen kann
- Bei Computeralgorithmen bezieht sich der Ressourcenverbrauch auf den benötigten **Speicher**
- Wie die Ausführungsdauer wird auch der Speicherverbrauch normiert, um Algorithmen vergleichen zu können ➔ **(Speicher)-Platzkomplexität**
- Die Platzkomplexität ist neben der Zeitkomplexität das wichtigste Bewertungsmaß für Algorithmen

2.3 Elemente der Datenmanipulation

- Algorithmen, die durch Computer ausgeführt werden, berechnen Daten
 - Wenn sie in der realen Welt Wirkung zeigen sollen, werden entsprechende Interfacegeräte benötigt (Bildschirm, Soundkarte+Lautsprecher, Motoren, etc...)
 - Wir betrachten hier (erst mal nur) Modellansätze zur Datenmanipulation
- Ein Modell für die Abarbeitung von Algorithmen haben wir bereits kennengelernt: die mathematische Funktion ➔ **Funktionsmodell**
 - ➔ Ausgangsdaten als Funktion von Eingangsdaten
 - Die Eingangsdaten sind von den Ausgangsdaten unabhängig
- Grundsätzlich könnte man mit dem Funktionsmodell allein alle möglichen (Computer-)algorithmen erstellen
- In der Praxis kommen jedoch in der Regel auch noch andere Ansätze zum tragen.

⁵Zum Vergleich: Die Gesamtbevölkerung der Erde wurde zum Mai 2020 auf ca. 7,8 Mrd. geschätzt.

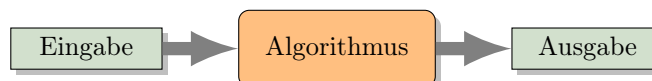
⁶Der übrigens auch „nur“ ca. 4,3 Millionen € betragen würde.

Zustandsmodell

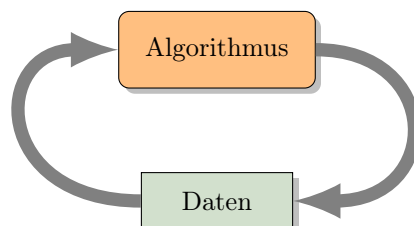
- Im Funktionsmodell kommt es nur auf **Werte** an, die berechnet werden
 - Da Eingangsdaten und berechnete Daten erst mal nichts miteinander zu tun haben, können beide (nach Berechnung bzw. Definition) als **unveränderlich** angesehen werden
- Anders im **Zustandsmodell**
 - Es wird **nicht** strikt zwischen Eingangswerten und Ausgangswerten unterschieden, sondern eine Gesamtheit von Daten
 - Daten werden durch den Algorithmus **bearbeitet**, d.h. verändert
 - Zu jedem gegebenen Zeitpunkt während des Ablauf des Algorithmus besitzen die Daten damit einen bestimmten **Zustand**, sie sind **veränderlich**

Die folgenden Abbildungen zeigt den konzeptionellen Unterschied zwischen beiden Modellen:

- **Funktion:**



- **Zustandsänderung:**



Zustandsänderung als Zeitreihe

- Im Zustandsmodell⁷ können die Werte der Daten als Reihe $d_1, d_2, d_3, \dots$, bzw. als Zeitreihe: $d_{t_1}, d_{t_2}, d_{t_3}, \dots$ aufgefasst werden
- Ein einzelner Schritt des Algorithmus kann wieder als Funktion aufgefasst werden:

$$d_{t+1} = f_{S_t}(d_t)$$

- Um nicht ständig Indizes zu schreiben, lässt man sie bei der Beschreibung von Algorithmen häufig weg und schreibt:

$$d \leftarrow f_{S_t}(d)$$

- Dabei ist auf der linken Seite der „neue“ und auf der rechten Seite der „alte“ Wert von d gemeint

⁷Achtung, der Begriff „Zustandsmodell“ hat in anderen Gebieten eine z.T. völlig andere Bedeutungen, z.B. in der Regelungstechnik

- Interpretation: Folgezustand ist Funktion des Vorzustandes

Achtung

Viele Programmiersprachen (u.a. auch C, Java und Python) nutzen die Schreibweise

$$d = f(d)$$

wenn eigentlich

$$d \leftarrow f(d)$$

gemeint ist.

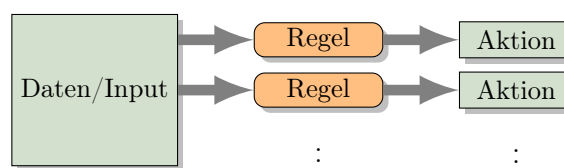
Das Gleichheitszeichen „=“ ist hier **nicht** als mathematisches Gleichheitszeichen zu verstehen

- Bei anderen Programmiersprachen wird versucht, diese Verwirrung zu vermeiden, indem dort „:=“, „<-“ o.ä. benutzt wird

Konditionalmodell

- Prinzipiell kann jeder Algorithmus aus einer Menge von **Wenn-Dann-Regeln (Konditional-klauseln)** bestehen:

- Wenn A gilt, dann geschehe X
- Wenn B gilt, dann geschehe Y
- Wenn C gilt, dann geschehe Z



- Die Reihenfolge der Regeln ist dabei nicht (unbedingt) entscheidend, sondern hängt in erster Linie von den Bedingungen, Daten oder Input ab
 - Auch im Funktionsmodell oder Zustandsmodell gibt es Fallunterscheidungen, die aber nur zu bestimmten Zeitpunkten ausgewertet werden
- Die Aktion kann sowohl ein Wert/eine Ausgabe (wie im Funktionsmodell), als auch eine Änderung eines Zustands sein (wie im Zustandsmodell)
- Beispielalgorithmus 11 (Ableitung) ist nach diesem Modell gestaltet

Diskussion

- Prinzipiell stellt ein Modell nur eine **Betrachtungsweise** dar
- Alle drei Modelle können gegenseitig simuliert
- Ein wesentlicher Unterschied der Modelle ist der Umgang mit **Zeit**
 - Das Funktionsmodell hat *keinen Ausdruck von Zeit* ⇒ Auswertung findet „auf einmal“ statt
 - Das Zustandsmodell hat ein *explizites Zeitmodell* ⇒ Zeitschritt bei Wechsel von Vor- zum Nachzustand
 - Das Konditionalmodell erlaubt *Nebenläufigkeit* ⇒ Aktionen können ggf. auch „gleichzeitig“ stattfinden
- Welches Modell zur Formulierung eines Algorithmus tatsächlich genutzt wird, hängt ab von...
 - der Art des Problems (manche lassen sich besser in einem bestimmten Modell ausdrücken)
 - den vorgegebenen Möglichkeiten zur Problembeschreibung (z.B. Programmiersprache)
 - persönlichen Vorlieben

Anmerkung

Vielen Menschen fällt das Denken mit Zuständen meist am leichtesten, da es am ehesten der Alltagserfahrung entspricht.

- Programmiersprachen nutzen meist mehrere Modelle, manche
 - **C** und **Python** sind stark auf das Zustandsmodell orientiert (nutzen aber auch das Funktionsmodell)
 - * **C++**, **C#** oder **Java** nutzen darüber hinaus durch das Exceptions-Konzept auch das Konditionalmodell
 - In „Algorithmen und Programmierung“ wird keine rein-funktionale⁸ Sprache behandelt; ein Beispiel für eine solche Sprache ist **Haskell**
 - **make** (siehe Kapitel 13) orientiert sich am Konditionalmodell
- Aus der Perspektive der Algorithmenmodell entspricht die Hardware der heutigen Rechner⁹ am ehesten dem Zustandsmodell
 - Ursache ist der Standardzyklus der Befehlsverarbeitung (⇒ siehe Rechnerorganisation oder EFC)

⁸so werden Sprachen genannt, die nur das Funktionsmodell unterstützen

⁹das war bei den historischen Analogrechnern anders, und auch heute gibt es z.B. Datenflussarchitekturen

- Da die Modelle sich gegenseitig simulieren können, ist die Hardware für die benutzte Sprache aber (nahezu) egal

2.4 Konstruktion von Algorithmen

- Aus den drei Grundmodellen kann man die (abstrakten) Elemente zusammenstellen, die einen Algorithmus bilden
- Wir gehen aber in erster Linie (wie in C) von Zustandsmodell aus
- **Berechnen, Ausdruck auswerten**, z.B.:
 - $\frac{-b + \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$
 - ...die Spielkarte, die am weitesten links liegt
 - ... der erste Buchstabe im Wort
- **Zustand verändern**, z.B.:
 - Linken Fuß an den rechten Fuß heranziehen
 - $i \leftarrow i + 1$
 - Tausche zwei Karten
- **Entscheidung treffen**, z.B.:
 - Wenn $x > y$, dann ... sonst ...
 - Bei Feueralarm (rufe die Feuerwehr)
 - Mache um 16.30 Uhr Feierabend

Zusammensetzen der Elemente

- Um einen Algorithmus zu formen, werden einzelne dieser Elemente miteinander verknüpft
- Diese Verknüpfung ist in der Regel **sequentiell**...
 - Mach dies** $\Rightarrow$ dann das $\Rightarrow$ dann jenes
- ...jedoch können die Elemente auch parallel miteinander verknüpft sein (insbesondere im Konditionalmodell)
- Mitunter muss etwas mehrfach gemacht werden
 - Rekursion (wie im Beispiel 7) $\Rightarrow$ ausführlicher im nächsten Kapitel und im Kapitel 11
 - Iteration (Schleifen) (wie im Beispiel 5) $\Rightarrow$ ausführlicher in Kapitel 6
- Wenn Sie einen Algorithmus entwerfen wollen, überlegen Sie...
 - welche(s) Ding(e) Sie manipulieren/berechnen wollen
 - $\Rightarrow$ „Dinge“ meint hier z.B. Gegenstände, Objekte, aber auch Ausdrücke oder Werte

- wie Sie bei mehreren zu betrachtenden Dingen diese unterscheiden können
- wie das Ergebnis aussieht
- welche Schritte Sie unternehmen müssen, um zum Ergebnis zu kommen
- Es ist in der Regel hilfreich, sich die vom Algorithmus betroffenen Dinge und ihre **Beziehungen** auf einem Blatt Papier zu visualisieren

Telefontest

Tipp

Ein guter Hinweis, ob Ihr Algorithmus tatsächlich den Anforderungen an einen solchen genügt, ist der **Telefontest**:

Überlegen Sie sich, wie Sie jemanden, der die vom Algorithmus betroffenen Dinge vor sich sieht, am Telefon erklären, was er machen soll, **ohne dass Sie** diese Dinge sehen.

- Tatsächlich ist es oft hilfreich, anderen sein Vorgehen zu erklären (es muss ja nicht unbedingt am Telefon sein...)
- Wir Betrachten den Entwurf (und die Korrektheit!) von Algorithmen noch einmal genauer im Kapitel 9, nachdem wir gelernt haben, wie die einzelnen Algorithmenelemente in der Programmiersprache C aussehen

Knobelaufgaben

Aufgabe 2.1

Überlegen Sie sich, wie oft sie im Algorithmus aus Beispiel 12 jeweils im günstigsten und im ungünstigsten Fall zwei Karten miteinander tauschen müssen, wenn Sie auf diese Weise n Karten sortieren wollen.

Aufgabe 2.2

Beweisen Sie, dass der Algorithmus aus Beispiel 8 tatsächlich die Strecke $\overline{AB}$ halbiert!

Aufgabe 2.3

Sie müssen den Wert von 2^{64} durch Multiplikation berechnen. Was ist die kleinste Anzahl von Multiplikationen, die Sie vornehmen müssen?

Kapitel 3

Einführung in Programmiersprachen

The most disastrous thing that you can ever learn is your first programming language.

— Alan C. Kay

C is quirky, flawed and enormous success.

— Dennis Ritchie

3.1 Grundsätzliches

- Die Ausführung eines Algorithmus auf einem Computer setzt voraus, dass der Computer den Algorithmus **interpretieren** kann:
 - Er muss „verstehen“, was jeder Schritt bedeutet
 - Er muss die jeweilige Operation ausführen können
- Jeder Prozessor¹ kann ein Bitmuster interpretieren
 - Dieses Bitmuster wird **Maschinencode** genannt
 - Der Maschinencode ist typisch für jede Prozessorfamilie
- Beispiel: Im Maschinencode der x86-Prozessorfamilie bedeutet das Bitmuster

11111010

dass der Prozessor sich nicht „ablenken“ lassen soll, d.h., dass keine Interrupts² beachtet werden sollen

¹Präziser: jeder **digitale** Prozessor

²Das Konzept des Interrupts wird in den Fächern der technischen Informatik genauer behandelt

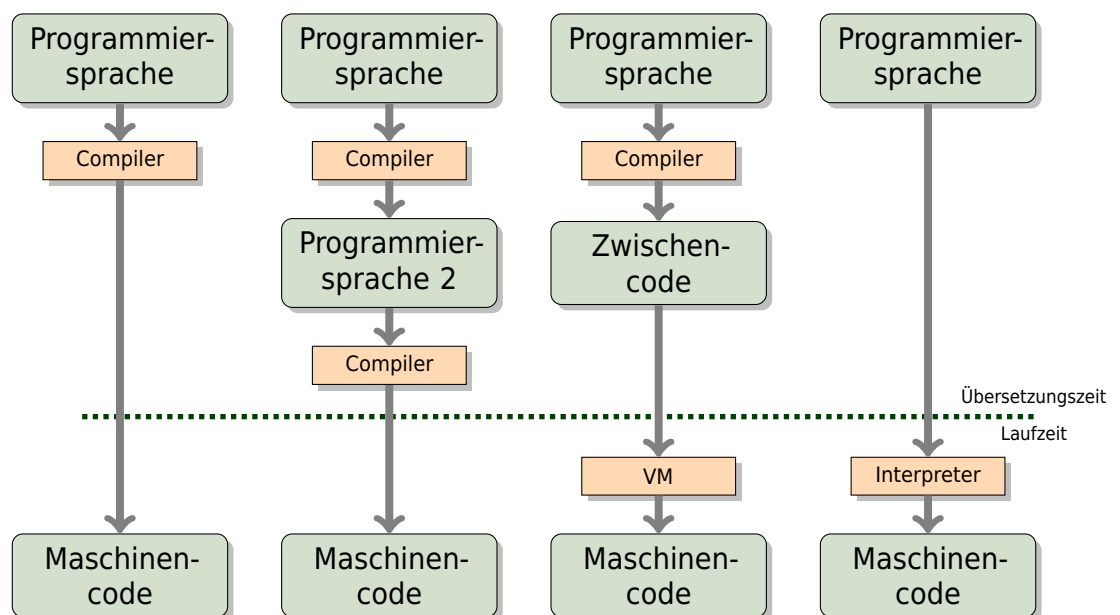
Compiler und Interpreter

- Für den Menschen ist Maschinencode nicht besonders „handlich“ 😞
- Daher gibt es Programmiersprachen, die dem Denken des Menschen entgegenkommen
- Programmiersprachen benutzen verschiedene Modelle und Abstraktionen
 - Z.B. gibt es für jeden Maschinencode (mindestens) eine sogenannte **Assembler-Sprache**, die den Maschinencode in leicht(er) merkbaren Abkürzungen (**Mnemonics**) darstellt

11111010 → cli (für **clear interrupt**)

- Programme werden entweder in Maschinencode **übersetzt**, oder durch ein anderes Programm **interpretiert**
 - Übersetzer = **Compiler** ← zwei Schritte: 1. Übersetzen in Maschinencode, 2. Ausführen
 - Interpreter** ← ein Schritt: Übersetzen während der Ausführung

(Einige) Varianten der Programmausführung



- VM: Virtuelle Maschine: Simulation eines Prozessors, der den Zwischencode „versteh“
- Neben den hier dargestellten Formen der Programmausführung gibt es zahlreiche Mischformen und Zwischenvarianten

Tabelle 3.1: Einige Programmierparadigmen

Eigenschaft	(vereinfachte) Erläuterung	typische Beispiele
<i>imperativ</i>	betont das Zustandsübergangsmodell der Programmausführung	C, C++, C#, Java, Pascal
<i>funktional</i>	betont das Funktionsmodell der Programmausführung	Haskell, Scheme, Logo, ML
<i>deklarativ</i>	ermöglicht die Beschreibung eines Programmziels, statt des Weges dorthin	SQL, Prolog
<i>modular</i>	ein Programm kann aus verschiedenen Teilen „komponiert“ werden, wobei von den externen Teilen nur bestimmte Informationen bekannt werden	Modula 2, C#
<i>objektorientiert</i>	betont die Zusammengehörigkeit von Daten und Operationen	Simula, Smalltalk, Eiffel, Java, C++
<i>strikt</i>	Ausdrücke werden bei Zuweisung berechnet	C, Java, Ada
<i>nicht-strikt</i>	Ausdrücke werden bei Bedarf berechnet	Miranda, Haskell
<i>reflexiv</i>	Programm kann sich selbst modifizieren	Lisp, C#, Python

- Bei allen Varianten wird aber am Ende ein Maschinencode ausgeführt, der dem Hochsprachenprogramm entspricht
- Die hier betrachtete Sprache C funktioniert nach der ersten Variante (links), Java nach der dritten, und Python (im Wesentlichen) nach der vierten

Paradigmen

- Häufig werden Sie hören, dass eine Programmiersprache diesem oder jenem **Paradigma** folgt

Paradigma

Paradigma, das, Pl: Paradigmen, griech., παραδειγμα: Beispiel, Muster

- Gemeint ist, dass die Sprache gewisse **Eigenschaften** hat oder bestimmte **Programmierstile** (besonders gut) unterstützt
- Typischerweise unterstützen Sprachen mehrere Paradigmen
- Die Befolgung eines Paradigma sagt **nichts** über die Ausdrucksmächtigkeit einer Sprache
- Tabelle 3.1 listet einige Programmierparadigmen auf
- Prinzipiell kann **jedes** lösbare Problem in **jeder Sprache** gelöst werden

- Die Befolgung von Paradigmen können häufig auch durch Selbstbeschränkung oder Tricks erreicht werden, wenn die Sprache sie nicht unterstützt
- **Beispiele:** Die Programmiersprache C unterstützt keine Objektorientierung und C++ ist keine funktionale Sprache.
Es ist aber sehr wohl möglich, in C objektorientiert oder in C++ funktional zu programmieren
 - Objektorientierung in C: siehe z.B. [Sch01]
 - Funktionales Programmieren in C++ siehe z.B.: [MS00]

3.2 Geschichte

1822 Erster funktionsfähiger Entwurf einer **Analytical Engine** durch CHARLES BABBAGE; besitzt alle Einheiten eines modernen Rechners wie Rechenwerk, Zahlenspeicher, Steuerwerk, E/A-Möglichkeiten;
Programmwurf von AUGUSTA ADA KING BYRON zur Berechnung der Bernoulli-Zahlen

1936 ALAN M. TURING entwirft die sogenannte „Turing-Maschine“

1941 KONRAD ZUSE entwickelt den ersten funktionsfähigen Computer **Z3**

1944 **ENIAC** (Electrical Numerical Integrator And Computer): erster elektronischer Universalrechner

Computer wurden in dieser Zeit entweder durch „Verkabelung“ oder mit Lockkarten/-steifen in Maschinencode programmiert

1952 GRACE HOPPER entwickelt den ersten *Compiler* (Assembler)

Höhere Programmiersprachen

~1946 KONRAD ZUSE entwirft **Plankalkül**, die vermutlich erste höhere Programmiersprache; erste Implementierung 1972

1953 **FORTRAN** (**F**ormula **T**ranslating System): erste implementierte höhere Programmiersprache von JOHN W. BACKUS

- Viele bis heute verwendete Konzepte, Stammvater aller imperativen Sprachen
- Insbesondere für numerische Berechnungen vorgesehen und optimiert
- Immer wieder aktualisiert, letzte Version *Fortran 2008*

1958 **Lisp** (**L**ist **P**rocessing), entwickelt von JOHN MCCARTHY am MIT

- Bis heute in Verwendung, mit vielen Abkömmlingen, Stammvater aller funktionalen Sprachen
- Angelehnt an das λ -Kalkül
- Symbolische Programmierung

- Funktionen können auch lediglich teilweise ausgewertet und als Parameter übergeben werden
- automatische Speicherverwaltung (*garbage collector*)
- Mechanismen zur Robustheit (statische Zusicherungen, Alignment, Grenzüberprüfungen)

1958 **ALGOL** (**A**lgorithmic **L**anguage)

- Vorläufer vieler Programmiersprachen, z.B. C, Pascal, Java, ...
- allgemeine Schleifen
- Blockkonzept

1962 **Simula** (**S**imulation **l**anguage) entwickelt von OLE-JOHAN DAHL und KRISTEN NYGAARD

- Idee der Objektorientierung
- Vorgesehen für den Entwurf von Simulationssoftware
- Konzept von Co-Routinen

1970 **Smalltalk** von ALAN KAY, DAN INGALLS und ADELE GOLDBERG

- Erste vollständig objektorientierte Sprache: alles ist ein Objekt
- Virtuelle Maschine (VM)

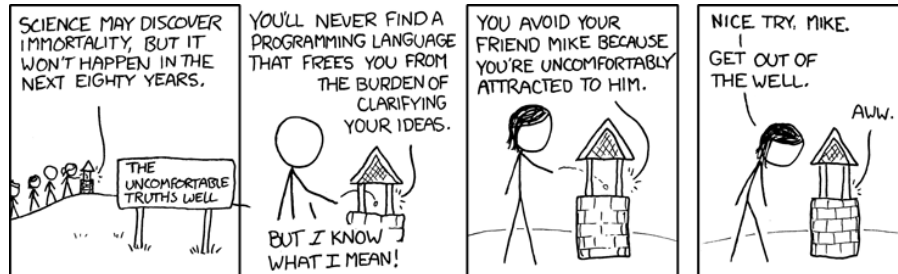
1972 **Prolog** (**P**rogrammation en **L**ogique) von ALAIN COLMEAUER

- Deklarative Programmiersprache zur Beschreibung logischer Sachverhalte
- Hauptanwendungszweck: Künstliche Intelligenz

Man beachte:

Damit wurden im Prinzip alle wesentlichen Programmiersprachkonzepte schon in den frühen Sprachen etabliert.

- Bis zum heutigen Tag wurden über 5000 Computersprachen entwickelt
- Die Online Historical Encyclopaedia of Programming Languages (HOPL) <http://hop1.info> hat sich zur Aufgabe gemacht, eine möglichst umfassende Liste zu erstellen um die Ideengeschichte von Programmiersprachen zu dokumentieren
- Im Anhang A sind für die wichtigsten 50 Sprachen Erscheinungszeiten und Abhängigkeiten in einem Graphen dargestellt



Quelle: *xkcd* - An webcomic of romance, sarcasm, math, and language
<http://xkcd.com/568/>

Die Programmiersprache C

Etwa 1972 entstand die erste Version von **C**

- Entwickelt von **BRIAN W. KERNIGHAN** und **DENNIS RITCHIE**
- Der Erfolg von C ist eng mit dem Erfolg des Betriebssystems **UNIX** verknüpft
 - Die ersten Versionen von UNIX waren in Assembler entwickelt
 - 1973 wurde UNIX nahezu komplett in C reimplementiert, was maßgeblich zum Erfolg beitrug
- Vorgängersprachen: **BCPL** ⇒ **B** ⇒ **C**
- Eine Vielzahl von Sprachen stammen von C ab oder wurden durch C inspiriert, u.a.: **C++**, **C***, **Concurrent C**, **C+@**, **Objective C**, **Cmm**, **Java**, **C#**, **Python**, ...

Die Sprache war im Laufe ihrer Geschichte vielen Wandlungen unterworfen

- **K&R C**: Erste (öffentliche) Version von C: 1978
- 1983 begann das ANSI³-Komitee **X3J11** die Sprache zu standardisieren.
- 1989 wurde der Standard ANSI X3.159-1989 verabschiedet, bekannt als **C89** oder **ANSI C**
 - U.a. geänderte Funktionsdeklarationen
- 1990 wurde der Standard von der ISO⁴ als ISO/IEC 9899:1990 (oder **C90**) übernommen
- Weiterentwicklung des Standards führten zu **C95** (ISO/IEC 9899:1994-09), das nur kleinere Änderungen gegenüber C90 einführt
- Größere Änderungen gab es mit ISO/IEC 9899:1999, bekannt als **C99**
 - Änderungen bei Typen, u.a. optionale Typen mit festgelegter (Mindest-)breite
 - Felder variabler Größe

³American National Standards Institute

⁴International Organization for Standardization

- Mehr Freiheiten bei der Deklaration
- Schlüsselwörter für Optimierung
- Kommentare von von C++ übernommen
- Verbot von impliziten Typen oder Funktionsdeklarationen
- Später kleinere Änderungen in *Technical Corrigenda*
- **C11** (Dezember 2011, ISO/IEC 9899:2011)
 - Explizites Maschinen- und Speichermodell
 - Bessere Unterstützung von Unicode
 - Viele Änderungen an der Standardbibliothek, z.B. Unterstützung von Multithreading
 - Unterstützung von Typengenerizität in Makros
 - Von den vielen Compilern derzeit noch nicht vollständig implementiert
- Aktueller Standard: **C18** (Juni 2018, ISO/IEC 9899:2018)
 - Keine neuen Sprachfeatures, sondern ein „Bugfix“ für C11
- Wir werden in diesem Kurs i.d.R. vom C99-Standard ausgehen, ohne alle Möglichkeiten von C99 auszuschöpfen
- Viele Compiler unterstützen mehrere Standards
 - Gewünschter Standard muss ggf. dem Compiler mitgeteilt werden
- Wo es wesentliche Unterschiede gibt, wird darauf hingewiesen: 

Achtung

Einige Standards, insbesondere C11/C18, werden von verschiedenen Compilern nicht vollständig unterstützt, was zu merkwürdigen Effekten führen kann.

Versuchen Sie, möglichst wenig versionsabhängig zu programmieren.



Eigenschaften von C

- Ist nahe an der **Hardware**
 - ➡ manchmal wird nicht von einer Hochsprache, sondern von einer „Mittelsprache“ gesprochen
 - ➡ Alle relevanten Betriebssysteme sind in C geschrieben
- Programme sind sehr **kompakt**
 - ➡ sie brauchen geringen Overhead und Speicher
- Ist weit **verbreitet**
 - ➡ vermutlich immer noch die Sprache, in der weltweit die meisten Programme geschrieben werden

- In C lassen sich sehr **effiziente** Programme schreiben
 - ➔ Geschwindigkeit von C-Programmen ist meist Maßstab

Größter **Vorteil** von C

C gibt dem Programmierer viele Freiheiten.

Größter **Nachteil** von C

C gibt dem Programmierer viele Freiheiten.

Geschichte von Python

- Da in diesem Kurs gelegentlich auch Python genutzt wird, gibt es auch davon eine (eher kurze) geschichtliche Darstellung
 - Erste Implementation von GUIDO VAN ROSSUM (Centrum Wiskunde & Informatica, Amsterdam) im Jahre 1989
 - Name abgeleitet von *Monty Python's Flying Circus*
 - 2000 erschien Version 2.0 und wurde zu einem OSS-Projekt
 - Nach mehreren Zwischenversionen erschien 2008 die **nicht** rückwärtskompatible Version 3.0
 - Aktuell: 3.7.4 (Juli 2019), jedoch wird 2.7 weiter gepflegt (aktuell: 2.7.16)
- Python hat trotz kurzer Geschichte schon einige andere Sprachen beeinflusst, u.a.: **Cobra**, **ECMAScript**, **Go**, **Groovy**

Philosophie

- Sowohl C/UNIX als auch Python werden häufig (nur halb im Scherz) mit der Philosophie des Zen-Buddhismus verglichen
- Beispiel: UNIX-Kōans über Meister Foo von ERIC STEVEN RAYMOND⁵

A Unix zealot, having heard that Master Foo was wise in the Great Way, came to him for instruction. Master Foo said to him: „When the Patriarch Thompson invented Unix, he did not understand it. Then he gained in understanding, and no longer invented it.“

„When the Patriarch McIlroy invented the pipe, he knew that it would transform software, but did not know that it would transform mind.“

„When the Patriarch Ritchie invented C, he condemned programmers to a thousand hells of buffer overruns, heap corruption, and stale-pointer bugs.“

„Truly, the Patriarchs were blind and foolish!“

The zealot was greatly angered by the Master's words.

⁵<http://www.catb.org/~esr/writings/unix-koans/index.html>

„These enlightened ones,” he protested. „gave us the Great Way of Unix. Surely, if we mock them we will lose merit and be reborn as beasts or MCSEs.“

„Is your code ever completely without stain and flaw?” demanded Master Foo.

„No,” admitted the zealot, „no man’s is.“

„The wisdom of the Patriarchs“ said Master Foo, „was that they knew they were fools.“

Upon hearing this, the zealot was enlightened.

The Zen of Python

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren’t special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one– and preferably only one –obvious way to do it.

Although that way may not be obvious at first unless you’re Dutch.

Now is better than never.

*Although never is often better than *right* now.*

If the implementation is hard to explain, it’s a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea – let’s do more of those!

3.3 C am Beispiel

Warnung

Bevor wir mit dem Programmieren in C beginnen, ein Wort der Warnung:

Achtung!

C-Compiler (insbesondere ältere) gehen davon aus, dass Programmierer wissen, was sie wollen. Viele Fehler treten dann erst zur Laufzeit auf.

- **Beispiel:** Der folgende Code stellt ein übersetz- und ausführbares C-Programm dar...

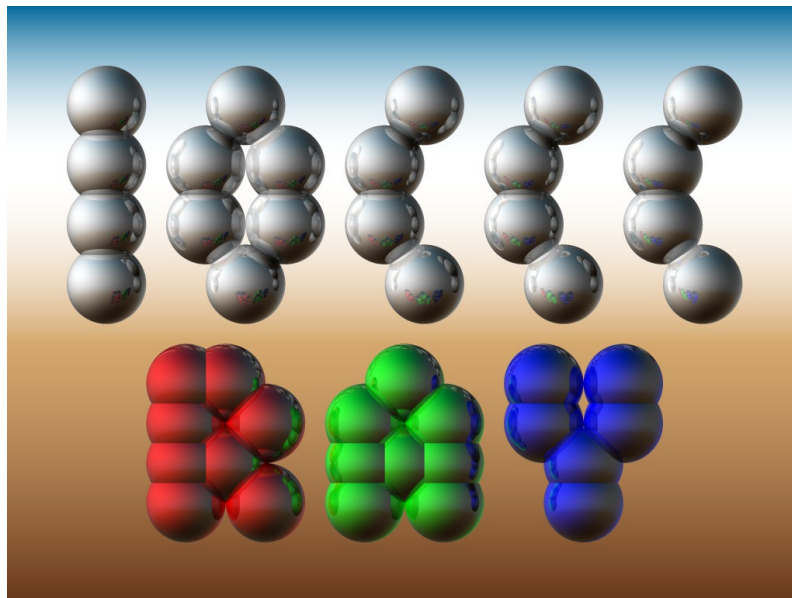
```

X=1024; Y=768; A=3;

J=0;K=-10;L=-7;M=1296;N=36;O=255;P=9;_=-1<<15;E;S;C;D;F(b){E="1""111886:6:??AAF"
"FHHMM005557799@@>>>BBBGGIIKK"[b]-64;C="C@=:C@=:C@=:C@=:C5""31/513/5131/"
"31/531/53"[b]-64;S=b<22?9:0;D=2;I(x,Y,X){Y?(X^=Y,X*X>x?(X^=Y):0,I(x,Y/2,X
)): (E=X);JH(x){I(x,_,0);Jp;q(c,x,y,z,k,l,m,a,b){F(c
);x-=E*M;y-=S*M;z-=C*M;b=x*x/M+y*y/M+z
*z/M-D*D*M;a=-x*k/M-y*l/M-z*m/M;p=(b=a*a/M-
b)>0?(I(b*M,_,0),b=E,a+(a>b?-b:b)):-1.0);JZ;W;o
(c,x,y,z,k,l,m,a){Z!=c?-1:Z;c<44?(q(c,x,y,z,k,
l,m,0,0),(p>0&&c!=a&&(p<W||Z<0))?(W=
p,Z=c):0,o(c+1,x,y,z,k,l,m,a)):0;JQ;T;
U;u;v;w;n(e,f,g,h,i,j,d,a,b,V){o(0,e,f,g,h,i,j,a);d>0
&&Z>0?(e+=h*W/M,f+=i*W/M,g+=j*W/M,F(Z),u=e-E*M,v=f-S*M,w=g-C*M,b=(-2*u-2*v+w)
/3,H(u*u+v*v+w*w),b/=D,b*=b,b*=200,b/=(M*M),V=Z,E!=0?(u=-u*M/E,v=-v*M/E,w=-w*M/
E):0,E=(h*u+i*v+j*w)/M,h=-u*E/(M/2),i=-v*E/(M/2),j=-w*E/(M/2),n(e,f,g,h,i,j,d-1
,Z,0,0),Q/=2,T/=2,U/=2,V=V<22?7:(V<30?1:(V<38?2:(V<44?4:(V==44?6:3))))
,Q+=V&1?b:0,T+=V&2?b:0,U+=V&4?b:0):(d=P?(g+=2
,j=g>0?g/8:g/20):0,j>0?(U=j*j/M,Q=255-250*U/M,T=255
-150*U/M,U=255-100*U/M):(U=j*j/M,U<M/5?(Q=255-210*U
/M,T=255-435*U/M,U=255-720*U/M):(U=-M/5,Q=213-110*U
/M,T=168-113*U/M,U=111-85*U/M),d!=P?(Q/=2,T/=2
,U/=2):0);Q=Q<0?0:Q>0?0:Q;T=T<0?0:T>0?0:T;U=U<0?0:
U>0?0:U;JR;G;B;t(x,y,a,b){n(M*J+M*40*(A*x+a)/X/A-M*20,M*K,M
*L-M*30*(A*y+b)/Y/A+M*15,0,M,0,P,-1,0,0);R+=Q;G+=T;B+=U;+a<A?t(x,y,a,
b):(++b<A?t(x,y,0,b):0);}r(x,y){R=G=B=0;t(x,y,0,0);x<X?(printf("%c%c%c",R/A/A,G
/A/A,B/A/A),r(x+1,y)):0;}s(y){r(0,--y?s(y),y:y);}main(){printf("P6\n%i %i\n255"
"\n",X,Y);s(Y);}

```

- ... das folgendes Bild generiert:



- Mehr davon beim **International Obfuscated C Code Contest**:
<http://www.de.ioccc.org/main.html>

Programmentwicklung

- Wir werden uns in diesem Kurs auf die Programmierung in C konzentrieren, aber gelegentlich auch Python benutzen
- Die Programmerstellung folgt stets gleichem Arbeitsablauf:

Schritt 1: Programmcode bearbeiten

Schritt 2: Programm übersetzen und linkern

– Falls dabei Fehler auftreten, zurück zu *Schritt 1:*

Schritt 3: Programm testen

– Falls dabei Fehler auftreten, zurück zu *Schritt 1:*

Anmerkung:

Es wird davon ausgegangen, dass jeder Kursteilnehmer in der Lage ist, mit einer Shell (tcsh, bash, ...) soweit umzugehen, dass sie/er ein Verzeichnis anlegen/wechseln/löschen, mit einem Editor Textdateien bearbeiten, und allgemein Befehle in eine Shell eingeben kann.

- Gegebenenfalls muss man sich dies im Eigenstudium beibringen
- Die Übungsleiter helfen gern

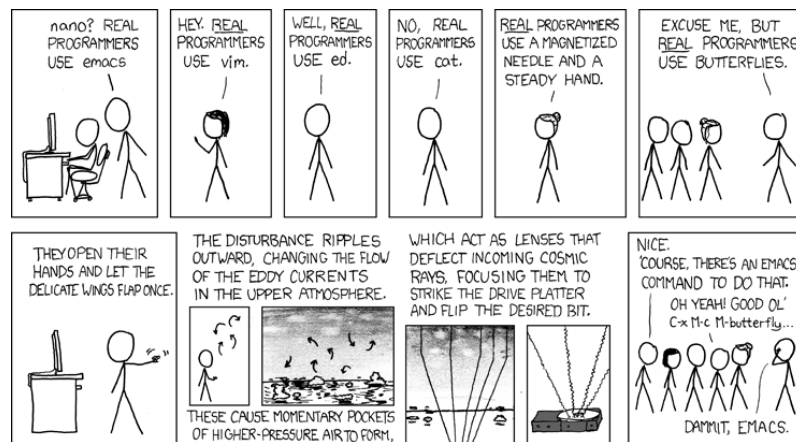
Werkzeuge

- Für die C-Programmerstellung braucht man mehrere Werkzeuge, mindestens zwei: einen Editor und einen Compiler⁶
- Als Editor kann jedes Programm genutzt werden, dass **einfachen** (unformatierten) Text bearbeiten kann
 - Editoren auf Shellebene: vim, emacs, nano, mcedit, ...
 - Grafische Editoren: kedit, kate, xemacs, ...
 - Gänzlich ungeeignet: ~~MS Word~~, ~~OpenOffice~~, ~~KWriter~~, ...

Hinweis

Benutzen Sie den Editor, der Ihnen am besten liegt.

Wenn Sie aber einen Editor einer **IDE** (Integrierte Entwicklungsumgebung, *integrated development environment*, z.B. Eclipse) benutzen, achten Sie darauf, dass Sie **nicht** die automatische Übersetzung der IDE benutzen



Quelle: xkcd - An webcomic of romance, sarcasm, math, and language
<http://xkcd.com/378/>

- Als Compiler werden wir stets den **clang**-Compiler oder den Compiler der **GNU Compiler Collection** (*gcc* nutzen)⁷
 - Beide sind auf jeder relevanten Plattform (Linux, MacOS, Windows, FreeBSD, ...) verfügbar
 - Prinzipiell sind jedoch auch andere C-Compiler möglich

Hinweis

Auch wenn dies einigen von Ihnen, die schon etwas Programmiererfahrung haben, vermutlich ungewohnt vorkommt:

Wir werden **den Compiler zunächst stets von der Kommandozeile** (oder über ein Makefile, siehe Kapitel 13) starten.

Ziel ist (zunächst) nicht die Programmier-Bequemlichkeit, sondern dass Sie verstehen, was Sie genau machen.

Das erste C-Programm

- Ein C-Programm besteht aus **Funktionen** (mindestens einer)

Wir betrachten folgendes Minimalprogramm:

⁶Eigentlich verbergen sich hinter dem Compiler häufig noch mehr Programme.

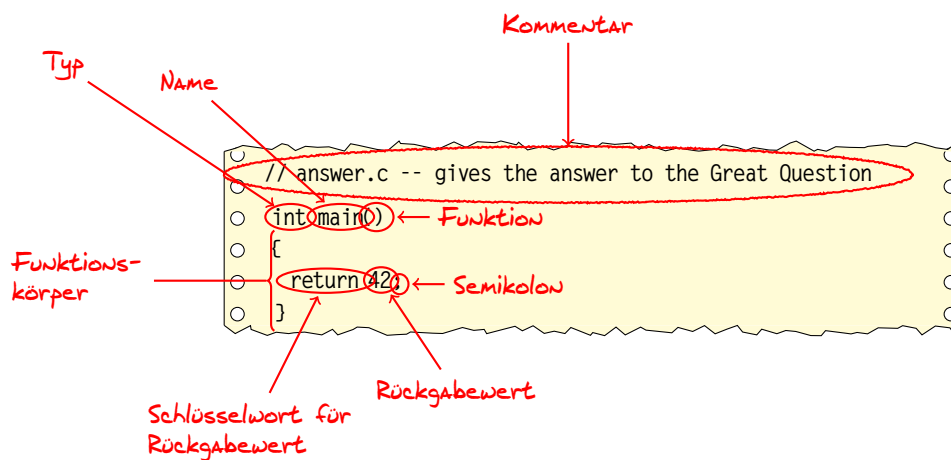
⁷Die Kommandozeilenoptionen sind für clang und GNU-Compiler in den meisten Fällen gleich.

```

// answer.c -- gives the answer to the Great Question
int main()
{
    return 42;
}

```

Untersuchen wir die einzelnen Elemente genauer:



- **Kommentar:** wird bei der Programmausführung ignoriert; enthält Informationen für *menschliche* Leser des Programms

Ein Kommentar ist entweder in `/*...*/` eingeschlossen, oder (ab C99) beginnt mit einem `///
" und geht dann bis zum Zeilenende.`

- **Typ:** Wertebereich der Funktion; `int` steht für „Integer“ $\hat{=}$ ganze Zahl

Wertebereich ist (wie bei aus der Mathematik bekannten Funktionen) eine Menge von möglichen Werten, siehe Kapitel 4.

- Jede Funktion hat einen **Namen**; „main“ ist die „Startfunktion“

Sie wird stets als erste gerufen, wenn ein C-Programm gestartet wird. Alle anderen Funktionen müssen von `main()` aufgerufen werden.

- Die runden Klammer kennzeichnen „main“ als **Funktion**

Innerhalb der Klammern können bei jeder Funktion auch die Parameter stehen, mit der sie aufgerufen wird, siehe Kapitel 4.

- Der **Funktionskörper** (oder Funktionsrumpf) wird in geschweifte Klammern eingeschlossen

- Das **Schlüsselwort** „return“ kündigt den Rückgabewert an



- Der Wert „42“ wird von der Funktion zurückgegeben⁸ ➔ **Rückgabewert**



Der Rückgabewert von `main()` wird an das Betriebssystem zurück gegeben und kann dort ausgegeben werden (siehe unten). Wird *kein* Wert zurückgegeben, d.h., wenn `main()` kein `return`-Statement besitzt, wird ab C99 der Wert 0 zurückgegeben. Bei ANSI-C ist der Rückgabewert unbestimmt.

- Das Semikolon markiert das Ende einer **Anweisung** (Statement)

⁸Warum 42? Wer es wirklich nicht weiß, sei auf folgendes Zitat aus „The Hitchhiker’s Guide to the Galaxy“ von Douglas Adams [Ada79] verwiesen:

“Good morning,” said Deep Thought at last.
 “Er ... Good morning, O Deep Thought,” said Loonquawl nervously, “do you have ... er, that is...”
 “An answer for you?” interrupted Deep Thought majestically. “Yes. I have.”
 The two men shivered with expectancy. Their waiting had not been in vain.
 “There really is one?” breathed Phouchg.
 “There really is one,” confirmed Deep Thought.
 “To Everything? To the great Question of Life, the Universe and Everything?”
 “Yes.”
 Both of the men had been trained for this moment, their lives had been a preparation for it, they had been selected at birth as those who would witness the answer, but even so they found themselves gasping and squirming like excited children.
 “And you’re ready to give it to us?” urged Loonquawl.
 “I am.”
 “Now?”
 “Now,” said Deep Thought.
 They both licked their dry lips.
 “Though I don’t think,” added Deep Thought, “that you’re going to like it.”
 “Doesn’t matter!” said Phouchg. “We must know it! Now!”
 “Now?” inquired Deep Thought.
 “Yes! Now...”
 “Alright,” said the computer and settled into silence again. The two men fidgeted. The tension was unbearable.
 “You’re really not going to like it,” observed Deep Thought.
 “Tell us!”
 “Alright,” said Deep Thought. “The Answer to the Great Question...”
 “Yes...!”
 “Of Life, the Universe and Everything...” said Deep Thought.
 “Yes...!”
 “Is.” said Deep Thought, and paused.
 “Yes...!”
 “Is.”
 “Yes...!!!!...?”
 “Forty-two,” said Deep Thought, with infinite majesty and calm.

Schlüsselwörter

- Wir haben in unserem ersten Programm zwei sogenannte **Schlüsselwörter** (*keywords*) benutzt
 - `int` ➔ (Markierung einer) Ganzzahl
 - `return` ➔ veranlasst Rückgabe eines Wertes und das Verlassen der Funktionsberechnung
 - * Im Fall der Hauptfunktion (d.h. `main()`) wird damit auch das gesamte Programm beendet
- Schlüsselwörter sind Wörter im Programmtext, die für den Compiler eine vordefinierte Bedeutung haben
- C kennt (in der Version C99) 37 Schlüsselwörter⁹
 - Wir werden die meisten in diesem Kurs kennenlernen
 - `main` ist **kein** Schlüsselwort: man kann den Compiler auch dazu bringen, die Programmausführung mit einer anderen Funktion zu starten; diese Möglichkeit sollte aber nur in Ausnahmefällen genutzt werden



Es gibt für jede Sprache genaue Regeln, die für den Einsatz von Schlüsselwörtern gelten. Zunächst werden diese Regeln nur verbal und am Beispiel beschrieben.

Im Kapitel 7 werden dann Möglichkeiten vorgestellt, diese Regeln formal zu beschreiben, und im Anhang B sind diese Regeln für C aufgelistet.

Übersetzen eines C-Programms

Um das Programm zu übersetzen (compilieren) wird der Compiler aufgerufen:

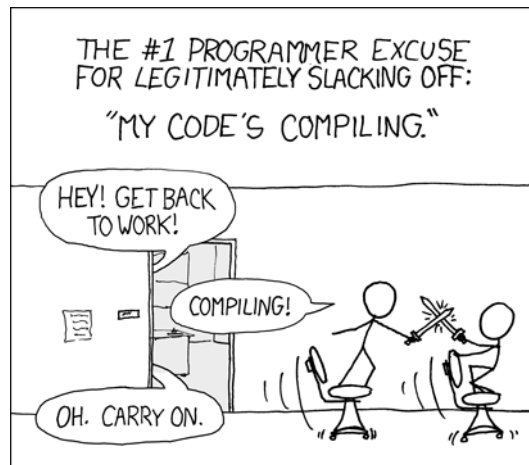
```
> cc -std=c99 -Wall answer.c -o answer
```

- `cc`: Der Name des Compilers
- `-std=c99`: Aktiviert den Sprachstandard C99
- `-Wall`: Gibt Warnungen aus
- `answer.c`: Name des Quellcode-Files
- `-o answer`: Bestimmt den Namen des übersetzten Programms

Aufruf des Programms:

```
> ./answer
> echo $?
42
>
```

⁹Eine Übersicht finden Sie im Anhang B.



Quelle: xkcd - An webcomic of romance, sarcasm, math, and language
<http://xkcd.com/303/>

Anmerkung

Bei der Nutzung einer IDE, egal ob direkt auf Ihrem Rechner (z.B. Xcode oder Eclipse) oder web-basiert (z.B. Codingblocks) wird ebenfalls der Compiler mit den entsprechenden Optionen aufgerufen.

Dieser Aufruf (und ggf. weitere) erfolgen jedoch **versteckt**, so dass der Entwickler keine unmittelbare Übersicht hat, was vorsichgeht.

Ausdrücke

- Statt einer Konstanten¹⁰ kann auch ein **Ausdruck** (expression) berechnet und zurückgegeben werden:

```

/* answer2.c -- calculates the answer to the[-75ex] Great Question */
int main ()
{
    return 23+19; Ausdruck
}

```

```

> cc -std=c99 -Wall answer2.c -o answer2 > ./answer2
> echo $?
42
>

```

¹⁰Genaugenommen ist es programmiertechnisch keine Konstante, sondern ein *Literal*, aber dazu später mehr

- C kennt die Operatoren für die Grundrechenarten und die Punkt-vor-Strich-Regel
- Folgende Ausdrücke haben z.B. das gleiche Resultat (den Wert 42):
 - $65-23$
 - $126/3$
 - $3*7+7*3$
 - $100-2*29$
- Ein **Ausdruck** ist etwa das, was in der Mathematik ein **Term** ist

Achtung

Bei der Division ganzer Zahlen gelten auch die Regeln im Ganzzahlbereich, es wird also **nicht** gerundet!

- z.B.: ergibt der Ausdruck $11/12$ den Wert 0 (mit dem Rest 11)
- Auch für die Ermittlung des Rests bei der Ganzzahldivision gibt es einen Operator: %
- So ergibt der Ausdruck $141\%99$ wieder 42
- Es gibt noch viele andere Operatoren in C, die wir im Laufe dieses Kurses kennenlernen werden
 - In manchen Programmiersprachen kann man sich **eigene** Operatoren definieren – in C geht dies nicht

Deklaration

- Wie in Termen in der Mathematik können auch in C-Ausdrücken **Funktionen** enthalten sein
- **Problem:** C (genauer: der C-Compiler) kennt (erst mal) keine Funktionen.¹¹
- **Aber:** Für manche Funktionen¹² reicht es, wenn man dem Compiler mitteilt, dass sie **existieren**

- Beispielsweise teilt die Zeile

```
int abs(int);
```

dem Compiler mit, dass eine Funktion mit dem Namen `abs` existiert, die eine Ganzzahl als Parameter nimmt und eine Ganzzahl zurückgibt

- Was die Funktion **berechnet**, ist dem Compiler egal
- Eine solche „Existenzmitteilung“ an den Compiler nennt man **Deklaration** der Funktion

Für uns als Programmierer ist das natürlich nicht egal: `abs()` berechnet den Absolutwert

¹¹außer `main()`

¹²konkret für alle Funktionen, die zusammen mit dem Compiler in einer *Bibliothek* mitgeliefert worden; später mehr dazu

- Die Funktion `int abs(int)` berechnet den Absolutwert, d.h.: $\text{abs}(x) = \begin{cases} x & \text{wenn } x \geq 0 \\ -x & \text{sonst} \end{cases}$
- Das folgende Programm gibt also wieder die Antwort auf die Große Frage:

```
/* answer3.c -- Great Question, again */
int abs(int);
int main()
{
    return abs(100-58);
}
```

```
% > cc -std=c99 -Wall answer3.c -o answer3
> ./answer3
> echo $?
42
>
```

- **Ohne** die Deklaration gibt der Compiler eine Warnung (je nach Option auch einen Fehler) aus:

```
/* answer3b.c -- warning */
int main()
{
    return abs(100-58);
}
```

```
> cc -std=c99 -Wall answer3b.c -o answer3b
answer3b.c:5:10: warning: implicit declaration of function 'abs'
[-Wimplicit-function-declaration]
   5 |     return abs(100-58);
     |               ^~~~
```

Achtung!

Insbesondere sehr alte Compiler nehmen das Fehlen einer Deklaration so einfach hin und machen dann Annahmen, die zu eigenartigen Fehlern führen können.

Vergessen Sie daher nicht, Funktionen, die Sie nutzen wollen, auch zu deklarieren.

Bibliotheken und Header

- Nochmal zurück zur Funktion `abs()`:
 - Wir haben sie deklariert, aber nirgends geschrieben, **was** sie macht ➔ **definiert**
 - Sie ist **kein** Schlüsselwort
- **Frage:** Wenn der Compiler `abs()` nicht kennt, wo kommt eigentlich die Definition der Funktion her?
- **Antwort:** Aus der **C-Standardbibliothek**
- C-Standard schreibt eine Mindestmenge an Funktionen in der Standardbibliothek vor
- Einzelne Hersteller können zusätzliche Funktionen zur Verfügung stellen

Erläuterung

Der C-Standard lässt verschiedene **Ausführungsumgebungen** (*execution environments*) zu.

Für den in der Praxis häufigsten Fall „**hosted environment**“ muss zu den C-Tools auch noch eine Sammlung von fertig übersetzten Funktionen mitgeliefert werden, eben die Standard-Bibliothek.

- Neben der Standardbibliothek kann es noch weitere Bibliotheken geben, die aber ggf. erst auf einem System installiert werden müssen
- Deklarationen zu Bibliotheken sind in **Headerdateien** zusammengefasst ➔ Dateiname endet mit `.h`
- Headerdateien werden über die `#include`-Direktive in Code eingefügt
- Die `#include`-Direktive gibt es in zwei Versionen:
 - `#include<Dateiname>`
Sucht nach der Datei „*Dateiname*“ im **Verzeichnis für System-Include-Dateien**
 - `#include"Dateiname"`
Sucht nach der Datei „*Dateiname*“ im **lokalen Verzeichnis**
 - Wenn im letzten Fall die Datei nicht gefunden wird, wird sie im Systemverzeichnis gesucht
- Die gefundene Datei wird jeweils so behandelt, als ob ihr Inhalt **Bestandteil der Quelldatei** ist
- Wenn wir die Deklaration über die Einbindung der Headerdatei über die `#include`-Direktive realisieren, sieht unserer Programm jetzt so aus:

```

/* answer3.c -- Great Question, again */
#include<stdlib.h>

int main()
{
    return abs(100-58);
}

```

Was eigentlich ist eine „Direktive“?

Direktiven in C (erkennbar durch das Nummernzeichen #) werden vom **Präprozessor** ausgeführt und verändern den Quelltext, **ehe** der Compiler ihn „sieht“.

- Verschiedene Teile der Standardbibliothek sind in verschiedenen Headerdateien deklariert
 - Auch wenn es der Name suggeriert, `stdlib.h` enthält **nicht** die vollständige Deklaration der Standardbibliothek; andere ggf. nötige Headerdateien sind z.B. `stdio.h` oder `assert.h`
- Bibliotheken werden nochmal genauer im Kapitel 14 betrachtet
- **Frage:** Woher soll ich wissen, was eine Funktion macht und welche Headerdatei nötig ist?
- **Antwort:** Dafür gibt es Dokumentationen
- Generell kann man die Beschreibung einer Funktion (in UNIX-artigen) Betriebssystemen mit dem `man`-Befehl in der Shell nachschlagen

```

ABS(3)                                BSD Library Functions Manual                                ABS(3)

NAME
    abs - integer absolute value function

LIBRARY
    Standard C Library (libc, -lc)

SYNOPSIS
    #include <stdlib.h>

    int
    abs(int i);

DESCRIPTION
    The abs() function computes the absolute value of the integer i.

```

SEE ALSO

cabs(3), fabs(3), floor(3), hypot(3), imaxabs(3), labs(3), llabs(3), math(3)

:

- Was ist, wenn es die Funktion **nicht** in der Standardbibliothek gibt?

```
/* answer4.c -- Great Question, again */
int min(int, int);
int main()
{
    return min(100, 42);
}
```

```
> cc -std=c99 -Wall answer4.c -o answer4
Undefined symbols for architecture x86_64: "_min":
ld: symbol(s) not found for architecture x86_64
collect2: error: ld returned 1 exit status
```

- Man schreibt sie sich einfach selbst!
- Sprachregelung: man **definiert** die Funktion ➔ **Funktionsdefinition**

```
/* answer4.c -- Great Question, again */
int min(int, int);
int main()
{
    return min(100, 42);
}
int min(int x, int y){
    if (x<y) return x;
    else return y;
}
```

- Wenn eine Funktion **vor der Nutzung** definiert wird, wird sie auch gleich deklariert

Fallunterscheidungen

- Wir haben gerade zwei neue Schlüsselwörter kennengelernt: `if` und `else`
- Sie dienen zur **Ausführung** von **Fallunterscheidungen**
- **Regeln:**
 - `if` untersucht eine **Bedingung** (hier: ob `x` kleiner als `y` ist) und führt das folgende Statement (hier: `return x;`) aus, wenn die Bedingung eingetroffen ist (*wahr ist*)
 - Die Bedingung ist ein **Ausdruck**, der in Klammern geschrieben wird
 - `else` erklärt die **Alternative**: Es sagt, was ausgeführt wird (hier: `return y;`), wenn die Bedingung **nicht** eingetroffen ist
 - `if` und `else` gehören zusammen: es gibt **nie** ein `else` ohne ein vorheriges `if`
 - Allerdings braucht ein `if` nicht ein folgendes `else` ➔ dann wird als Alternative einfach **nichts** gemacht
- In der Bedingung werden zwei Zahlen miteinander verglichen: `x<y`
- Dabei ist das „<“ auch ein Operator, genauer ein **Vergleichsoperator**
- Es gibt noch mehr Vergleichsoperatoren:
 - `>` größer als
 - `<=` kleiner oder gleich
 - `>=` größer oder gleich
 - `==` gleich

Achtung

Der Gleichheitsoperator „`==`“ besteht aus **zwei** Gleichheitszeichen.

Es gibt in C auch noch einen Operator mit einem Gleichheitszeichen, aber der hat eine **andere Bedeutung**.

Blöcke

- Nach dem `if` darf nur **genau ein** Statement stehen:

```
if (x<y) return x;
```

- Was, wenn wir aus irgendeinem Grund mehr als ein Statement bräuchten?
- In diesem Fall kann man mehrere Statements in einen **Block** setzen, der durch geschweifte Klammern markiert wird
- Der Funktionskörper, der ja auch durch geschweifte Klammern gekennzeichnet ist, kann als ein „Spezialfall“ eines Blockes angesehen werden

- Beispiel¹³:

```

if (x==y) {
    x++; // die Bedeutung dieser Zeile lernen wir noch... :-)
    return x;
}

```

- Aus Compilersicht bildet ein Block (nach „außen“) ein einzelnes Statement

3.4 Der erste (richtige) Algorithmus

```

1  /* euclid.c -- calculates GCD */
2  #include<stdlib.h> // for abs()
3
4  int min(int, int);
5  int euclid(int, int);
6
7  int main ()
8  {
9      return euclid(45, 30);
10 }
11
12 int euclid(int x, int y)
13 {
14     if (x==y) return x;
15     else return euclid(abs(x-y), min(x,y));
16 }
17
18 int min(int x, int y){
19     if (x<y) return x;
20     else return y;
21 }

```

- Das vorstehende Programm implementiert eine Variante des Euklidschen Algorithmus (siehe Beispielalgorithmus 7 aus Kapitel 2)
- Zur (hoffentlich!) besseren Übersichtlichkeit, enthält das Listing Zeilennummern und farbige **Schlüsselwörter**
 - Schlüsselwörter sind Wörter, die in einer Programmiersprache eine vordefinierte Bedeutung haben.

#3 - #5: Macht den Compiler mit den neuen Funktionen bekannt ➔ **Deklaration**

¹³Das Beispiel demonstriert nur die Schreibweise für Blöcke, der Code des Originalalgorithmus wäre durch diese Änderung nicht mehr korrekt.

- `int abs(int);`: Funktion hat den Namen „abs“, hat ein Ganzzahlargument und gibt eine Ganzzahl zurück
- `int min(int, int);`: Funktion hat den Namen „min“, hat zwei Ganzzahlargument und gibt eine Ganzzahl zurück
- `int euclid(int, int);`: Funktion hat den Namen „euclid“, ebenfalls hat zwei Ganzzahlen als Argumente und gibt auch eine Ganzzahl zurück

#7 - #10: Definition der Hauptfunktion `main()`

- `main()` braucht (als einzige Funktion) keine Deklaration

#9: Der Rückgabewert der Funktion `main()` (d.h. des Gesamtprogrammes) ist Wert der Funktion `euclid` mit den Parametern 45 und 30

#12 - #16: Definition der Funktion `euclid()`, die den eigentlichen Algorithmus enthält

#18 - #21: Definition der (schon bekannten) Funktion `min()`

Übersetzen und aufrufen:

```
> cc -std=c99 -Wall euclid.c -o euclid
>
> ./euclid
> echo $?
15
>
```

An dieser Stelle soll noch einmal betrachtet werden, warum der Algorithmus des Euklid in der dargestellten Form funktioniert.

Es wird dabei das Prinzip der **Rekursion** genutzt.

Zum Verständnis ist es wichtig, den Unterschied zwischen einem Algorithmus und seiner Ausführung zu beachten. Wenn **innerhalb** der Funktion „euclid“ die Funktion „euclid“ erneut aufgerufen wird, entspricht das **nicht** einfach einen Sprung an den Anfang der Funktion. Vielmehr wird der Algorithmus (Funktion) neu gestartet. Das „Modell der kleinen Menschen“ zeigt den Unterschied.

- Nochmalige Betrachtung, **warum** der Algorithmus funktioniert
- Man nehme an, dass Berechnungen nicht von einem Computer durchgeführt werden, sondern von einer Gruppe tausender fleissiger kleiner Menschen.¹⁴
- Den **Algorithmus** (Anleitung) kennen alle, bei der **Durchführung** werden die Aufgaben verteilt.
- **Beispiel „Euklidischer Algorithmus“:**
 - *Manfred* ist für die Funktion `main` verantwortlich. Dazu geht er zu *Ernst*, und sagt ihm die Werte 45 und 30

¹⁴Dieses Modell geht auf BRIAN HARVEY zurück, siehe z.B. [HW94]; [Har97]

- *Ernst* (für `euclid` verantwortlich) vergleicht die Zahlen, die *Manfred* ihm gegeben hat. Da sie ungleich sind, zieht er die kleinere von der größeren ab ($45 - 30 = 15$) und sagt nun *Eva* die Werte 15 und 30
- *Eva* (ebenfalls für `euclid` verantwortlich) vergleicht die Zahlen, die *Ernst* ihr gegeben hat. Da sie ungleich sind, zieht sie die kleinere von der größeren ab ($30 - 15 = 15$) und sagt nun *Egon* die Werte 15 und 15
- *Egon* (ebenfalls für `euclid` verantwortlich) vergleicht die Zahlen, die *Eva* ihm gegeben hat. Da sie gleich sind, gibt er *Eva* den Wert 15 zurück
- *Eva* gibt den Wert *Ernst*, *Ernst* gibt ihn *Manfred*, und *Manfred* teilt ihn der Öffentlichkeit mit
- Das Beispiel soll Folgendes verdeutlichen:
 - Jeder der kleinen Menschen weiß nicht, was der andere für Werte hat. Z.B. Egon hat **keine Ahnung**, welche Zahlen Ernst erhalten hat
 - Obwohl die Zahlen in der Gebrauchsanweisung immer `x` und `y` heißen, sind es für jeden der kleinen Menschen **unterschiedliche** Zahlen
 - Obwohl keiner der kleinen Menschen weiß, mit welchen Zahlen ein anderer arbeitet, **funktioniert** der Algorithmus

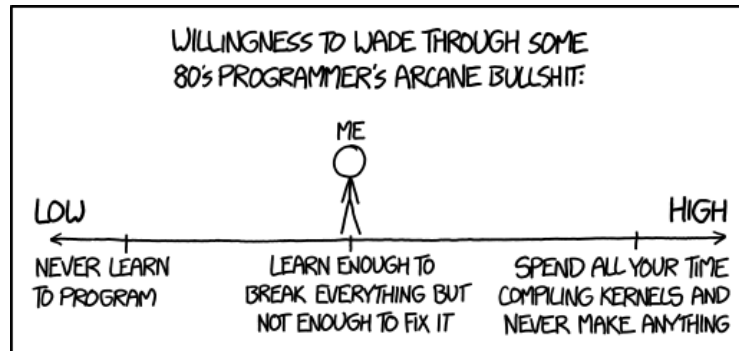


Parameterisierung und Ausgabe

- Unser Programm berechnet nur den größten gemeinsamen Teiler von 45 und 30
- Wenn wir ihn für andere Werte berechnen wollen, müssen wir den Quelltext ändern und das Programm neu übersetzen ➔ unbequem
- Daher wollen wir den Algorithmus **parameterisierbar** machen
- **Problem:** die tatsächlichen Parameter von `main()` sind etwas kompliziert
- Daher benutzen wir für diesen zunächst(!) Kurs eine „Spezialversion“ von `main()`, nämlich `Main()`:
 - An die neue Startfunktion¹⁵ `Main()` können von der Befehlszeile Ganzzahlen übergeben werden
 - `Main()` ist in der Headerdatei `aup-main.h` deklariert, die Sie für eigene Zwecke über folgende URL herunterladen können: <https://osg.informatik.tu-chemnitz.de/lehre/aup/support/aup-main.h>

¹⁵Für Fortgeschrittene: genau genommen ist `Main()` ein Makro

- Die genauen Parameter von `main()` werden wir im Kapitel 4 betrachten



Quelle: *xkcd* - An webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1671/>

- Außerdem wollen wir das Ergebnis ausgeben, ohne es in der Shell extra abzufragen
 - **Anmerkung:** Der bisher benutzte Mechanismus ist eigentlich nur für Fehlercodes gedacht
- Wir benutzen die Bibliotheksfunktion `printf()`, die in `stdio.h` deklariert ist
 - `printf` steht für „*print formatted*“, also etwa „gib formatiert aus“
- Die neue Variante des Algorithmus ist im folgenden Code dargestellt (neue Funktionen **hervorgehoben**)

```

1  /* euclid2.c -- calculates GCD with terminal output */
2  #include "aup-main.h" // for Main()
3  #include <stdlib.h>    // for abs()
4  #include <stdio.h>    // for printf()
5  int min(int, int);
6  int euclid(int, int);
7
8  int Main(int x, int y) {
9      printf("GCD is %d\n", euclid(x,y));
10     return 0;
11 }
12
13 int euclid(int x, int y) {
14     if (x==y) return x;
15     else return euclid(abs(x-y),min(x,y));
16 }
17
18 int min(int x, int y) {
19     if (x<y) return x;
20     else return y;
21 }

```

```
> cc -std=c99 -Wall euclid2.c -o euclid2
> ./euclid2 35 49
GCD is 7
```

- Was macht `printf()`?
 - Das erste Argument ist der **Formatierungstext** ➔ bestimmt die **Formatierung** der folgenden Ausdrücke, die ausgegeben werden sollen
 - Jedes **Prozentzeichen** (%) steht für einen **zu formatierenden Ausdruck**
 - Auf das Prozentzeichen folgt die **Formatierungsanweisung**, z.B. steht „d“ für Formatierung einer Ganzzahl im Dezimalsystem, und „x“ für Formatierung einer Ganzzahl im Hexadezimalsystem¹⁶
 - Alles, was **nicht** Formatierungsanweisung ist, wird direkt ausgegeben
 - Die Möglichkeiten der Formatierungsanweisung sind sehr umfangreich, siehe man 3 `printf` (oder Anhang B.4 für einen Überblick)

• Ausgabe: 42 - 1a

- Nun kann das Programm mit Parametern aufgerufen werden:

```
> ./euclid2 15 45
GCD is 15
> ./euclid2 42 23
GCD is 1
> ./euclid2 12 16
GCD is 4
>
```

3.5 Python

In mancher Beziehung ist C etwas „eigen“. Daher kann es Lernenden mitunter schwerfallen zu entscheiden, ob eine bestimmte Eigenheiten eine Eigenschaft des Algorithmus oder der Programmiersprache ist. Darum wird in diesem Kurs zum Vergleich eine zweite Programmiersprache benutzt: Python.

Es ist nicht Sinn des Kurses, den Studierenden die vollständige Beherrschung von Python zu lehren. Vielmehr werden nur Strukturen vermittelt, die in etwa eine Äquivalenz in C haben. Anderes, wie z.B. Objektorientierung oder Reflexion wird an dieser Stelle ausgelassen.

¹⁶Hexadezimalsystem ist ein Positionssystem mit der Basis 16 (siehe Mathematik Klassenstufe 5). Das Hexadezimalsystem wird wegen seiner „kompakten“ Darstellung in der Informatik häufig genutzt, siehe auch Kapitel 4

Das Gleiche in Blau/Gelb

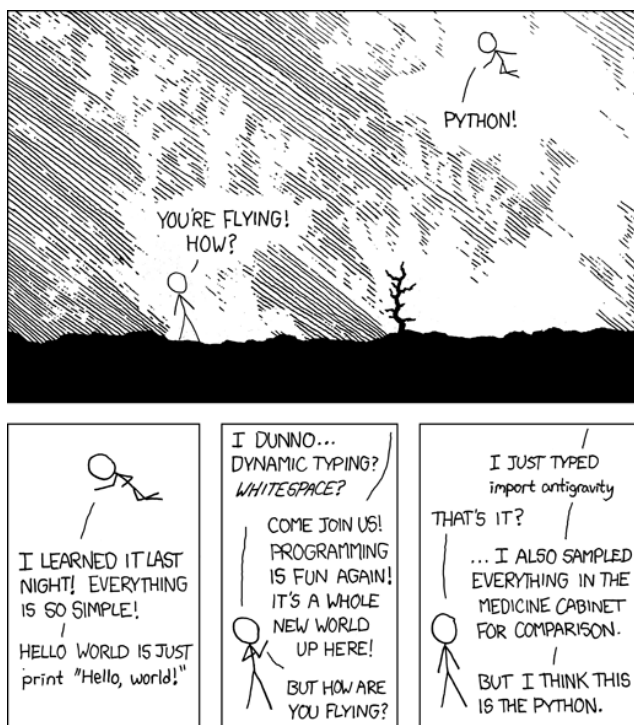
- Python wird zur Laufzeit interpretiert ➔ **Skriptsprache**
- Präzisiert: Ein Python-Programm wird zur Laufzeit in einen Zwischencode übersetzt, der dann interpretiert wird.
- Das Great-Answer-Programm sieht in Python so aus:



```
1 # answer.py -- gives the answer to the Great Question
2
3 exit(42)
```

- Zur Ausführung muss der Interpreter, also das Programm, das den Code „versteht“, angegeben werden:

```
> python answer.py
> echo $?
42
```



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/353/>

- Unter UNIX (z.B. Linux) kann alternativ auch in der 1. Zeile der Interpreter angegeben werden:

```

1 #! /usr/bin/env python
2 # answer2.py -- gives the answer to the Great Question
3
4 exit(42)

```

- Außerdem muss die Datei ausführbar gemacht werden, z.B.: `chmod a+x answer.py`
- Dann kann das Python-Skript direkt aufgerufen werden

```

> ./answer.py
> echo $?
42

```

Euklid

- Der Euklid-Algorithmus kann hier ähnlich wie in C beschrieben werden
 - Im Gegensatz zu C sind in Python `min()` und `abs()` in die Sprache eingebaute Funktionen (*built-in functions*)
- Es gibt in Python keinen Startpunkt, sondern das Programm wird von oben nach unten abgearbeitet

```

1 # euclid.py -- calculates GCD
2
3 def euclid(x,y):
4     if x==y: return x
5     else: return euclid(abs(x-y),min(x,y))
6
7 exit(euclid(45,30))

```

Achtung!

Während bei C Einrückungen nur der besseren Lesbarkeit dienen, sind sie bei Python verpflichtend.

Anmerkung: Die Frage, ob Einrückungen wie in Python (aber auch z.B. in Haskell oder Occam) eine semantische Funktion haben sollten – d.h., dass durch zusätzliche Einrückungen oder Leerzeichen die **Bedeutung** eines Programmes geändert werden kann – hat zu großen Kontroversen über den Entwurf von Programmiersprachen geführt, vergleiche <http://wiki.c2.com/?SyntacticallySignificantWhitespaceConsideredHarmful>

Interaktivität

- Python kann interaktiv benutzt werden
- Dazu wird das `exit`-Statement aus dem Skript gelöscht

```

1 # euclid2.py -- calculates GCD
2
3 def euclid(x,y):
4     if x==y: return x
5     else: return euclid(abs(x-y),min(x,y))

```

```

>python
Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 24 2018, 02:44:43)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from euclid2 import euclid
>>> euclid(15,45)
15
>>> euclid(42,23)
1
>>>

```

- Die Datei „euclid2.py“ bildet ein **Modul**, dessen Funktion etwa der der Bibliothek in C entspricht

Hinweis

Durch die Interaktivität eignet sich Python sehr gut, um mit algorithmischen Ideen „herumzuspielen“.

Auch wenn das Erlernen von Python in diesem Kurs kein offizielles Ziel ist, empfiehlt es sich daher häufig bei der Ausarbeitung eines Algorithmus (nach dem Entwurf auf Papier!) ihn zunächst in Python umzusetzen. Damit werden Sie nicht durch typische C-Schwierigkeiten wie die C-Speicherverwaltung (siehe Kapitel 5) „abgelenkt“.

Zwischencode

- Nach den letzten Beispielen wird man im Arbeitsverzeichnis die Datei „euclid2.pyc“ finden
- Dies ist der Zwischencode (Bytecode), in den Python das Programm übersetzt, bevor es interpretiert wird
- Bei einem erneuten Aufruf wird – falls die Quelldatei nicht geändert wurde – direkt der Zwischencode ausgeführt

- Man kann ein Skript „xyz.py“ übersetzen, ohne es auszuführen:

```
1 >>> import py_compile
2 >>> py_compile.compile('xyz.py')
```

- Alternativ können alle Python-Dateien eines Verzeichnisses übersetzen werden:

```
> ls
answer.py      euclid.py      euclid2.py     euclid3.py
> python -mcompileall .
Listing . ...
Compiling ./answer.py ...
Compiling ./euclid.py ...
Compiling ./euclid2.py ...
Compiling ./euclid3.py ...
> ls
answer.py      euclid.py      euclid2.py     euclid3.py
answer.pyc     euclid.pyc     euclid2.pyc    euclid3.pyc
```

Ein- und Ausgabe

- Analog zur Standardbibliothek in C gibt es jede Menge vordefinierte Module
- Ein Beispiel ist das Modul „sys“, welches ein Interface zum Betriebssystem bereitstellt
 - `argv[1]` und `argv[2]` sind das erste und zweite auf der Kommandozeile übergebene Argument
 - `print()` ist eine eingebaute Funktion
 - Anmerkung: vor Python 3 war `print` ein Schlüsselwort

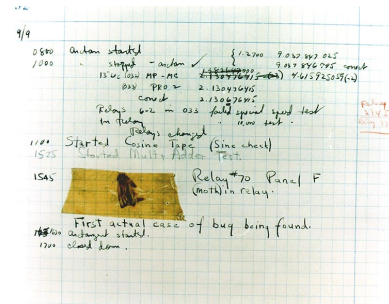
```
1 # euclid3.py -- calculates GCD
2 from sys import argv
3
4 def euclid(x,y):
5     if x==y: return x
6     else: return euclid(abs(x-y),min(y,x))
7
8 print('GCD is', euclid(int(argv[1]),int(argv[2])))
```

3.6 Fehler

Die wenigsten Programme sind von Anfang an fehlerfrei, viele sind es nie. Daher besteht ein großer Teil der Programmierarbeit in der Suche und der Beseitigung von Fehlern (⇒ **Debugging**¹⁷). Daher sollen kurz typische Arten von Fehlern diskutiert werden.

*Auszug aus dem Logbuch des elektromechanischen Computers
Harvard Mark II*

Bildquelle: Wikipedia



Blick unter die Haube

- Die verschiedenen Schritte bei der Programmerstellung eines C-Programms macht der Compiler nicht allein
- Vielmehr werden eine Reihe anderer Programme¹⁸ aufgerufen
- Im Einzelnen passiert folgendes:

Weg zum ausführbaren Programm

1. Der **Präprozessor** wertet Direktiven aus und fügt Inhalt von Headerdateien in den Code ein
2. Der eigentliche **Compiler** übersetzt den C-Code in Assembler-Code
3. Der **Assembler** übersetzt den Assembler-Code in Maschinencode und generiert daraus eine Objektdatei
4. Der **Linker** sucht die externen Referenzen des Objektfiles und sucht entsprechende Objekte aus der **Standardbibliothek** oder anderen angegebenen Bibliotheken (und Start-Objekt) zusammen und bindet alles zu einer ausführbaren **Programmdatei**

- Man kann den Compiler dazu bringen, nur einzelne Stufen des Erstellungsprozesses durchzuführen
 - „-E“ stoppt nach dem Präprozessor
 - „-S“ erzeugt die Assemblerdatei und stoppt danach
 - „-c“ erzeugt Objektdatei, ohne diese zu linken

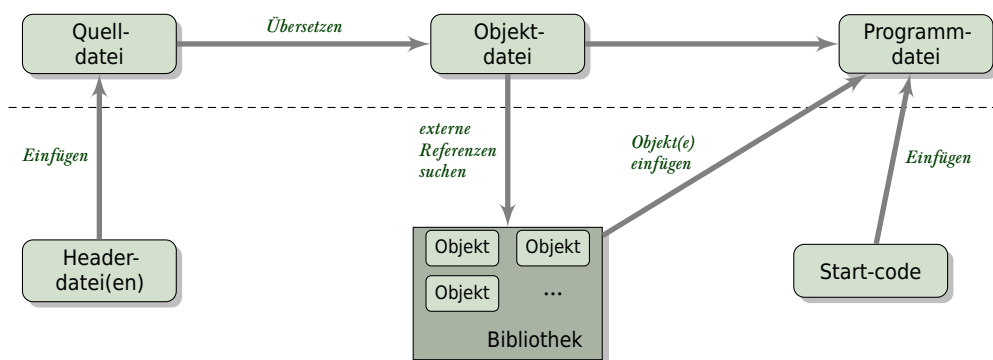
¹⁷Die Bezeichnung „bug“ (=Käfer, Insekt) für technische Probleme und Fehler ist älter als die Geschichte der Computer; jedoch ist er im Zuge der Computerisierung populär geworden.

¹⁸Mitunter werden die verschiedenen Funktionen aus Optimierungsgründen durch das gleiche Programm wahrgenommen.

- Beim GNU-Compiler kann man mit Hilfe der Option „-v“ gut sehen, was im einzelnen passiert (das folgende Beispiel ist gekürzt)

```
> gcc -ansi -pedantic -Wall -v euclid3.c -o euclid3
Using built-in specs.d
Target: i686-apple-darwin10
/usr/libexec/gcc/i686-apple-darwin10/4.2.1/cc1 -quiet -v -imultilib
x86_64 -D__DYNAMIC__ euclid3.c -fPIC -quiet -dumpbase euclid3.c -
mmacosx-version-min=10.6.4 -m64 -mtune=core2 -ansi -auxbase euclid3 -
pedantic -Wall -ansi -version -o /var/folders/WI/WI0tIF54VPPoSktWoc
+++NM/-Tmp-//ccZ1bYNd.s
/usr/libexec/gcc/i686-apple-darwin10/4.2.1/as -arch x86_64 -
force_cpusubtype_ALL -o /var/folders/WI/WI0tIF54VPPoSktWoc+++NM/-Tmp-
-//cc65KfC6.o /var/folders/WI/WI0tIF54VPPoSktWoc+++NM/-Tmp-//ccZ1bYNd
.s
/usr/libexec/gcc/i686-apple-darwin10/4.2.1/collect2 -dynamic -arch
x86_64 -macosx_version_min 10.6.4 -weak_reference_mismatches non-weak
-o euclid3 -lcrt1.10.6.o -L/usr/lib/gcc/i686-apple-darwin10/4.2.1/
x86_64 -L/usr/lib/gcc/i686-apple-darwin10/4.2.1/x86_64 -L/usr/lib/gcc
/i686-apple-darwin10/4.2.1 -L/usr/lib/gcc/i686-apple-darwin10
/4.2.1/../../../../var/folders/WI/WI0tIF54VPPoSktWoc+++NM/-Tmp-//
cc65KfC6.o -lSystem -lgcc -lSystem
```

- Hier nochmal eine Übersicht über den Weg vom Quellcode zum Programm



Hoppla!

Man beachte folgendes Programm

```

1  /* euclid-fail.c -- a faulty euclid */
2
3  // announce function to compiler
4  int euclid(int; int);
5
6  int main()
7  {
8      return euclid(45,30);
9  }
10
11 // defines the function
12 int euclid(int x, int y)
13 {
14     if (x==y) return x;
15     else if (x>y) return euclid(x-y,y);
16     else return euclid(y-x,x);
17 }

```

```

> gcc -std=c99 -Wall euclid.c -o euclid
euclid.c:4:1: error: parameter '({anonymous})' has just a forward
    declaration
euclid.c: In function 'main':
euclid.c:8:10: error: too many arguments to function 'euclid'
euclid.c:4:12: note: declared here
euclid.c: At top level:
euclid.c:12:5: error: conflicting types for 'euclid'
euclid.c:4:12: note: previous declaration of 'euclid' was here
euclid.c:4:12: warning: 'euclid' used but never defined
>

```

Und noch einmal...

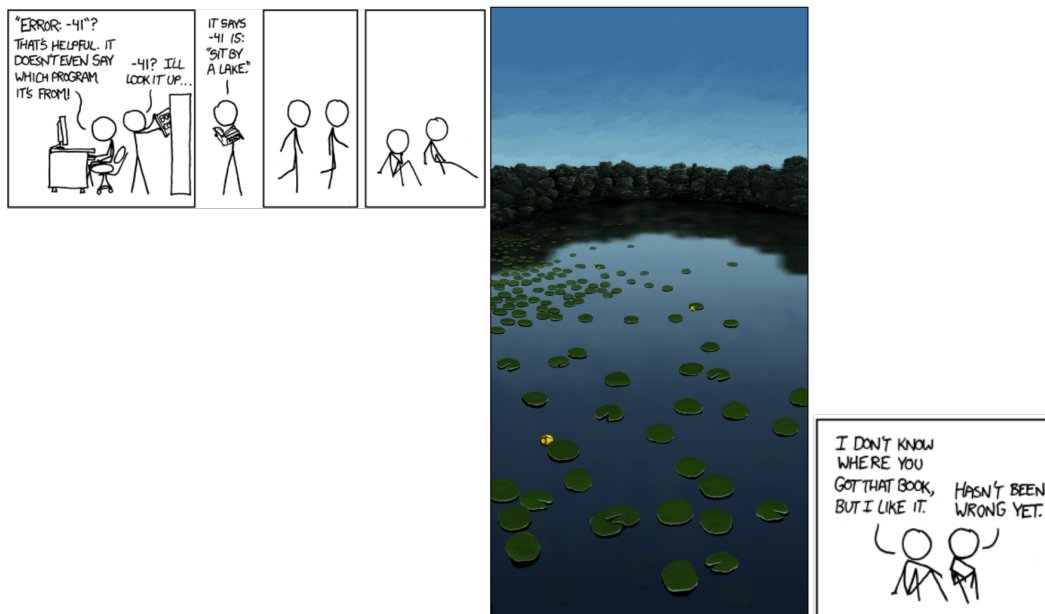
```

1  /* euclid-fail2.c -- again faulty */
2
3  // announce function to compiler
4  int euclid(int, int);
5
6  int main()
7  {
8      return euclid(45,30);
9  }
10
11 // defines the function

```

```
12 int euclid(int x, int y)
13 {
14     if (x==y) return x;
15     else if (x>y) return euclid(x-y,y);
16     else return euclid(y-x,x);
17 }
```

```
> gcc -std=c99 -Wall euclid.c -o euclid
euclid.c: In function 'main':
euclid.c:8:10: warning: implicit declaration of function 'euclid' [-Wimplicit-function-declaration]
euclid.c: At top level:
euclid.c:12:5: warning: 'euclid' defined but not used [-Wunused-function]
Undefined symbols for architecture x86_64:
  "_euclid", referenced from:
      _main in ccncBu2o.o
ld: symbol(s) not found for architecture x86_64
collect2: error: ld returned 1 exit status
>
```



Du bist gewarnt!

```

1  /* euclid-fail3.c -- faulty too */
2
3  // announce function to compiler
4  int euclid(int, int);
5
6  int main()
7  {
8      return euclid(45,30);
9  }
10
11 // defines the function
12 int euclid(int x, int y)
13 {
14     if (x=y) return x;
15     else if (x>y) return euclid(x-y,y);
16     else return euclid(y-x,x);
17 }

```

```

> cc -std=c99 euclid.c -o euclid-fail3
>
> ./euclid-fail3
> echo $?
30 ← Oops!
>
> gcc -std=c99 euclid.c -o euclid -Wall
euclid-fail3.c: In function 'euclid':
euclid-fail3.c:12: warning: suggest parentheses around assignment used as truth value
>

```

Aber ich dachte...

```

1  /* euclid3.c -- calculates GCD with I/O */
2  #include<stdio.h>    /* printf() */
3  #include"aup-main.h" /* Main() */
4
5  int euclid(int,int);
6
7  int Main(int a, int b)
8  {
9      printf("GCD is %d\n", euclid(a,b));
10     return 0;

```

```

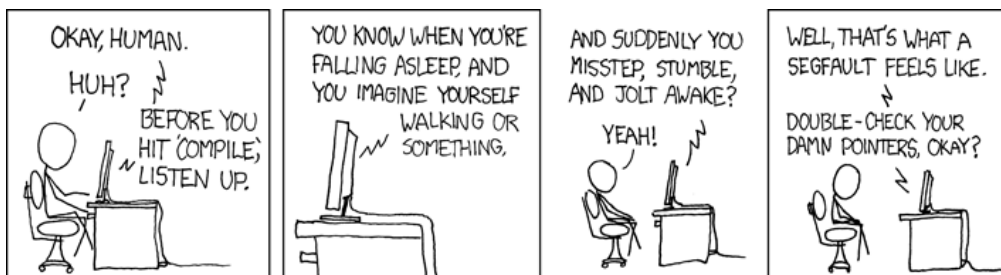
11 }
12
13 int euclid(int x, int y) {
14     if (x==y) return x;
15     else if (x>y) return euclid(x-y,y);
16     else return euclid(y-x,x);
17 }

```

```

> cc -std=c99 euclid3.c -o euclid3
>
> ./euclid3
Segmentation fault
> ./euclid3 45 30
GCD is 15

```



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/371/>

- Der Fehler lag hier darin, dass eine **Annahme** über eine Laufzeitbedingung gemacht wurde, die aber zur tatsächlichen Laufzeit **nicht** gilt
- Kein Compiler kann ahnen, was zur Laufzeit passiert ➔ keine Warnung
- Abhilfe: Annahme explizit machen und **defensiv** programmieren
 - Wenn wir die „richtige“ main()-Funktion betrachten (siehe Abschnitt 5.4), werden wir auch generische Vorsichtsmaßnahmen gegen diesen speziellen Fehlertyp diskutieren

```

1 /* euclid4.c -- calculates GCD, defensive version */
2 #include<stdio.h> /* printf */
3 #include<stdlib.h> /* atoi */
4 static int euclid(int,int);
5
6 int main(const int c, const char* v[])
7 {
8     if (c!=3) { /* 1st argument is program name */
9         printf("Error: unsufficient number of arguments\n");
10        return 1;

```

```

11 }
12 printf("GCD is %d\n", euclid(atoi(v[1]),atoi(v[2])));
13 }
14
15 int euclid(int x, int y) {
16     if (x==y) return x;
17     else if (x>y) return euclid(x-y,y);
18     else return euclid(y-x,x);
19 }

```

- Da ständig Annahmen gemacht werden, gibt es eine generische Lösung: assert
- Um assert zu nutzen, muss die Datei „assert.h“ eingebunden werden: #include<assert.h>

```

1 /* euclid5.c -- calculates GCD, using assert */
2 #include<stdio.h> /* printf */
3 #include<stdlib.h> /* atoi */
4 #include<assert.h> /* assert */
5 static int euclid(int,int);
6
7 int main(const int c, const char* v[])
8 {
9     assert(c==3);
10    printf("GCD is %d\n", euclid(atoi(v[1]),atoi(v[2])));
11 }
12
13 int euclid(int x, int y) {
14     if (x==y) return x;
15     else if (x>y) return euclid(x-y,y);
16     else return euclid(y-x,x);
17 }

```

- Es gibt offensichtlich mindestens drei Sorten von Fehlern
 - Formal „falsches“ C ⇒ **Syntaxfehler**¹⁹ (Compilerfehler)
 - Fehlende Objekte (z.B. Funktionen) oder Namenskollision ⇒ **Linkfehler**
 - Es wird etwas anderes gemacht, als beabsichtigt ⇒ **Semantikfehler**¹⁹ (Logikfehler)
- Syntaxfehler und Linkfehler i.d.R.²⁰ werden zur Übersetzungszeit bemerkt
- Semantikfehler können zwei Ursachen haben
 - Der Algorithmus wurde nicht richtig umgesetzt
 - Der Algorithmus ist falsch oder nur unter bestimmten Bedingungen korrekt

¹⁹Die Begriffe „Syntax“ und „Semantik“ werden in späteren Kapiteln noch genauer besprochen.

²⁰Es gibt pathologische Linkfehler, die erst zur Laufzeit auftreten.

Merke

Bei der ersten Art von Semantikfehler (falsche Umsetzung) schöpft der Compiler evtl.(!) einen „Verdacht“ und gibt (bei Anforderung) eine Warnung aus. Es empfiehlt sich, jeder Warnung nachzugehen, u.U. sogar die Compileroption „-Werror“ zu nutzen.

- Die Option „-Werror“ wandelt Warnungen in in Fehler um, bricht also die Übersetzung ab.
- In OpenSubmit werden für die Übersetzung der Belegaufgaben möglichst viele derartiger Warnungen (-Wall, -Wextra, ggf. -Wpedantic) eingeschaltet

Moderne Compiler (wie clang oder gcc ab Version 4.8) bieten teilweise sehr ausführliche Fehlermeldungen, die mit einem Cursor auf den genauen Fehlerort aufmerksam und Lösungsvorschläge machen. Beispielsweise sieht die Ausgabe von clang Version 7.3 für euclid-fail3.c so aus:

```
euclid.c:14:8: warning: using the result of an assignment as a condition without parentheses [-Wparentheses]
    if (x=y) return x;
       ^
euclid.c:14:8: note: place parentheses around the assignment to silence this warning
    if (x=y) return x;
       ^
       ( )
euclid.c:14:8: note: use '=' to turn this assignment into an equality comparison
    if (x=y) return x;
       ^
       =
1 warning generated.
```

Knobelaufgaben**Aufgabe 3.1**

Nehmen Sie an, dass für den Euklidischen Algorithmus, wie er z.B. in „euclid3.c“ angegeben wurde, nur Parameter aus dem Bereich $[1, \dots, 100]$ verwendet werden.

Wie häufig wird dann die Funktion „euclid(int,int)“ für eine ggT-Berechnung maximal aufgerufen?

Aufgabe 3.2

Finden Sie eine Bildungsvorschrift für die Zahlenfolge 2, 12, 1112, 3112, 132112, 1113122112, 311311222112, ..., und geben Sie den achten Wert an!

Kapitel 4

Typen

The primary purpose of the DATA statement is to give names to constants; instead of referring to pi as 3.141592653589793 at every appearance, the variable PI can be given that value with a DATA statement and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of pi change.

— Early FORTRAN manual for Xerox Computers

Some readers may need reminding that numbers are not string of bits, and $2^{33} \cdot 2^{33}$ equals 2^{66} , not overflow error.

— Leslie Lamport

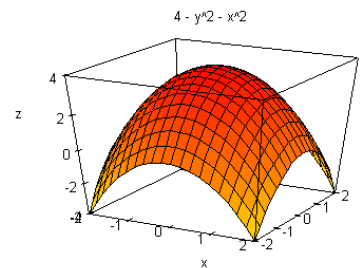
4.1 Einführung

In diesem Kapitel wird das grundlegende Konzept der Typen diskutiert.

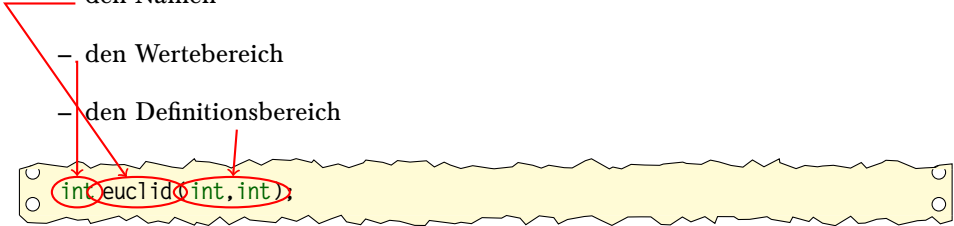
Typen sind z.B. ein allgemeines Konzept, das in vielen Programmiersprachen¹ vorkommt. Man unterscheidet getypte Sprachen (wie z.B. C), bei denen ein einmal festgelegter Typ unveränderlich bleibt, und ungetypte Sprachen (wie z.B. Python), bei denen ein Typ veränderlich ist.

¹eigentlich in allen, aber in manchen ist es praktisch nicht wahrnehmbar

- Erinnerung an Mathematik-Unterricht:
Jede Funktion hat einen **Definitionsbereich**²
und einen **Wertebereich**³
- Beides sind definierte **Mengen**
- **Beispiel:** die Funktion
 $z = f(x, y) = 4 - x^2 - y^2$, wie sie
(auszugsweise) im Graph zu sehen ist, hat
einen Definitionsbereich $D \subseteq \mathbb{R} \times \mathbb{R}$ und
einen Wertebereich $C \subset \mathbb{R}$
- **Schreibweise:** $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$



Deklaration von Funktionen

- Auch in C haben Funktionen einen Definitions- und einen Wertebereich
 - Wir haben bereits das Schlüsselwort `int` für die Menge der ganzen Zahlen kennengelernt
 - Zur Erinnerung: Eine Funktion in C muss **deklariert** werden
 - Ausnahme: Sie wird **vor** ihrer ersten Nutzung definiert
 - Ein Deklaration enthalten
 - den Namen
 - den Wertebereich
 - den Definitionsbereich
- 
- Darüber hinaus **kann** die Deklaration noch andere Dinge enthalten (später mehr)
- Mathematisch ausgedrückt: $\text{euclid} : \text{int} \times \text{int} \rightarrow \text{int}$

Typ und Signatur

- „ $\text{int} \times \text{int} \rightarrow \text{int}$ “ ist der **Typ** der Funktion „euclid“

²auch: Quellmenge, Domain

³auch: Zielmenge, Codomain

Achtung!

Unter C-Programmierern hat es sich eingebürgert, **nur** den **Wertebereich** als Typ zu betrachten.

Das ist vollständig okay für C, da bestimmte Aspekte von Typen in C keine Rolle spielen.

In anderen Sprachen ist dieser Unterschied aber wesentlich.

- Im verkürzenden C-Jargon hätte „euclid“ also den Typ „int“, korrekter wäre der Begriff **„Rückgabebetyp“**
- Der Typ bildet zusammen mit dem Namen die **Signatur** einer Funktion
 - ➔ Die Deklaration macht also den Compiler mit der Signatur einer Funktion bekannt

Wofür braucht man Typen und Signaturen überhaupt?

1. Viele Operationen (z.B. Addition ➔ „+“) sehen zwar immer gleich aus, werden aber unterschiedlich durchgeführt, je nachdem welchen Typ man betrachtet
 - Beispiel: natürliche vs. komplexe Zahlen

Typisierung hilft dem Compiler, die richtige Methode zu finden
2. Typen können dem Programmierer helfen, Denkfehler zu verhindern

- Nicht alle Programmiersprachen verlangen Deklaration von Typen ➔ andere Sprachen können Typen automatisch ableiten (**Typinferenz**)
- C ist relativ nachlässig mit den Konsequenzen aus der Typisierung
andere Sprachen sind da konsequenter (**stark typisiert**)

Typen von Ausdrücken

- Genau genommen ist der (Rückgabe-)typ einer Funktion nur ein Spezialfall
- **Alle** Ausdrücke (siehe Abschnitt 3.3) haben Typen
- **Beispiele:**
 - `-42` hat den Typ „ganze Zahl“ ($\mathbb{Z}$)
 - `23-42` hat ebenfalls den Typ „ganze Zahl“
 - `abs(23-42)` hat ebenfalls den Typ „ganze Zahl“, obwohl „natürliche Zahl“ ($\mathbb{N}$) ausgereicht hätte
 - `3.1415` hat den Typ „rationale Zahl“ ($\mathbb{Q}$)⁴
 - `abs(23-42) + 3.1415` hat ebenfalls den Typ „rationale Zahl“ ➔ Regeln werden später diskutiert

⁴Wir werden im Folgenden noch lernen, wie rationale Zahlen in C markiert werden

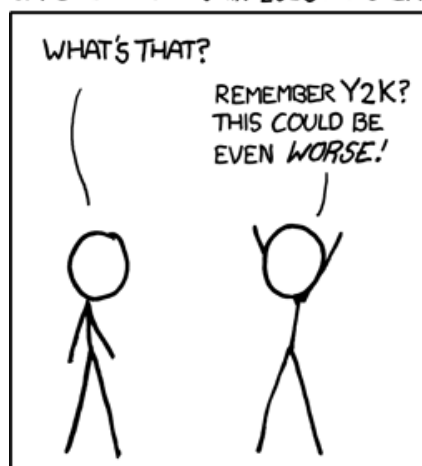
4.2 Grundtypen

- In C sind nur wenige Möglichkeiten für die Beschreibung von Definitions- und Wertebereich (Grundtypen) enthalten
 - Allerdings bietet C die Möglichkeit, weitere Typen „zu bauen“
- Die Mengen, die durch Typen beschrieben werden, sind in C nicht unendlich, bilden also z.B. nur eine Teilmenge der Menge natürlichen Zahlen ($\mathbb{N}$)
- C99 kennt folgende Grundtypen:
 - Teilmengen der **Ganzzahlwerte** ($\mathbb{Z}$)
 - Teilmengen der **rationalen Zahlenwerte** ($\mathbb{Q}$)
 - Menge der **booleschen Werte** (vollständig! 😊)
 - Teilmengen der **komplexen Zahlenwerte** ($\mathbb{C}$)
 - leere Menge



- Im Sprachumfang von C90 gab es weder boolesche noch komplexe Zahlen, sie ließen sich aber leicht „nachbauen“ und es wurden Bibliotheksfunktionen dafür zur Verfügung gestellt.

I'M GLAD WE'RE SWITCHING TO 64-BIT, BECAUSE I WASN'T LOOKING FORWARD TO CONVINCING PEOPLE TO CARE ABOUT THE UNIX 2038 PROBLEM.



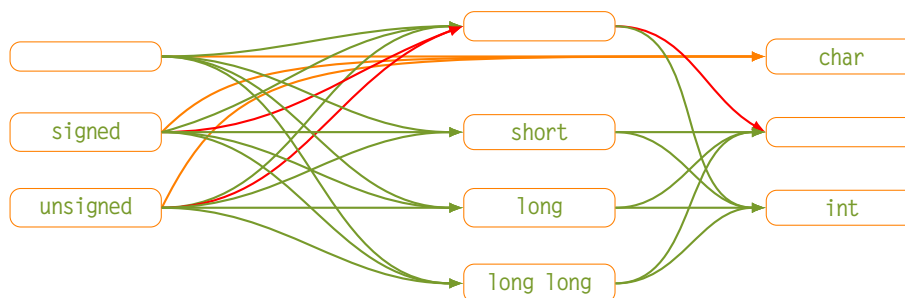
Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/607/>

4.2.1 Mengen von ganzen Zahlen

- Da die Mächtigkeit der Menge der ganzen Zahlen unendlich ist, lässt sie sich nicht ohne weiteres im Rechner darstellen
- Man beschränkt sich auf verschiedene **Untermengen** von ganzen Zahlen
- Zur Kennzeichnung der verschiedenen Untermengen stellt C folgende Schlüsselwörter zur Verfügung, die kombiniert aber auch ausgelassen werden können, zur Verfügung:

Grundgröße	Größenmodifikation	Vorzeichen
int	short	signed
char	long	unsigned

- `signed` erklärt einen vorzeichenbehafteten Typ (ganze Zahlen, Elemente aus $\mathbb{Z}$), `unsigned` einen Typ ohne Vorzeichen (natürliche Zahlen, Elemente aus $\mathbb{N}$)
 - **Ohne** `signed` oder `unsigned` gilt `int` als vorzeichenbehaftet, für `char` kann das der Compiler entscheiden
- Jede Kombination von Schlüsselworten entlang eines einfarbigen Pfads in der folgenden Grafik stellt eine gültige Beschreibung eines Ganzzahltyps in C dar
 - **Grüne** und **rote** Wege beschreiben den `int`-Grundtyp, **orange** Wege den `char`-Grundtyp



- `unsigned long func(signed char)` beschreibt also eine Funktion mit dem Namen „func“, die einen Parameter vom Typ `signed char` als Definitionsbereich und `unsigned long int` als Wertebereich (Funktionstyp) hat
- Warum gibt es so viele unterschiedliche Typen?
- ➔ Damit man alle Werte in einer Menge unterscheiden kann, braucht man je nach Menge unterschiedlich viel Speicherplatz pro Wert
- Wieviel?
- ➔ Der Computer speichert Werte mit Hilfe von Bits (=binary digits)
 - In n Bits können also 2^n verschiedene Werte gespeichert werden
 - C-Standard: alle Werte aus `char` müssen mit dem kleinsten adressierbaren Speicherobjekt auf dem jeweiligen Rechner darstellbar sein
 - Typischerweise ist die kleinste adressierbare Einheit ein **Byte** (=8 Bit)

- Werte der verschiedenen Mengen werden deshalb in einem oder mehreren Bytes gespeichert
- Die Wahl des „richtigen“ Typs ist deshalb immer ein Kompromiss zwischen der **benötigten Anzahl** unterschiedlicher Werte und dem **Speicherverbrauch** (und der Zugriffsgeschwindigkeit)
- Leider legt der Standard **nicht** die Speichergröße der einzelnen Werte fest $\Rightarrow$ kann für jeden Rechner/Compiler anders sein
- Es gilt lediglich:⁵

$$\text{card}(\text{char}) \leq \text{card}(\text{short int}) \leq \text{card}(\text{int}) \leq \text{card}(\text{long int}) \leq \text{card}(\text{long long int})$$

- Allerdings gibt es Mindestkardinalitäten, z.B. gilt: $\text{card}(\text{long long int}) \geq 18446744073709551616$
- Woher weiß ich denn dann, wieviel „Platz“ ein Wert eines Typs einnimmt?
- Zwei Möglichkeiten:
 1. RTFM⁶
 2. Nutzen Sie den `sizeof()`-Operator $\Rightarrow$ gibt Größe in Byte zurück

```
printf("An 'int' takes %lu bytes.\n", sizeof(int));
```

An 'int' takes 4 bytes.



- Um bei Bedarf den Programmierer „Größensicherheit“ bei Ganzzahlwerten zu geben, gibt es seit C99 es die Headerdatei „stdint.h“
- Darin sind Ganzzahltypen mit **fester Bitbreite** definiert, **falls** sie auf der jeweiligen Plattform zur Verfügung stehen:
 - `int8_t` und `uint8_t`: vorzeichenbehaftete und vorzeichenlose Ganzzahlen mit **genau** 8 Bit $\Rightarrow$ d.h. $[-128 \dots 127]$ bzw. $[0 \dots 255]$ (Kardinalität ist 256)
 - `int16_t` und `uint16_t`: vorzeichenbehaftete und vorzeichenlose Ganzzahlen mit **genau** 16 Bit $\Rightarrow$ d.h. $[-32768 \dots 32767]$ bzw. $[0 \dots 65535]$ (Kardinalität ist 65536)
 - `int32_t` und `uint32_t`: vorzeichenbehaftete und vorzeichenlose Ganzzahlen mit **genau** 32 Bit $\Rightarrow$ d.h. $[-2^{31} \dots 2^{31} - 1]$ bzw. $[0 \dots 2^{32} - 1]$ (Kardinalität ist 4294967296)
 - `int64_t` und `uint64_t`: vorzeichenbehaftete und vorzeichenlose Ganzzahlen mit **genau** 64 Bit $\Rightarrow$ d.h. $[-2^{63} \dots 2^{63} - 1]$ bzw. $[0 \dots 2^{64} - 1]$ (Kardinalität ist 18446744073709551616)

⁵ $\text{card}(\mathbb{M})$ gibt die **Kardinalität** oder **Mächtigkeit** einer Menge $\mathbb{M}$ an, also wieviel Elemente $\mathbb{M}$ enthält.

⁶ *Read the fine manual*

Bitmuster

- Der C-Standard legt auch nicht fest, wie ein Wert im Speicher „aussieht“, d.h., welches Bit-Muster welchem Wert entspricht
- Allerdings nutzen alle gängige Plattformen ein **binäres Positionssystem**, mit **Zweierkomplement** für die Darstellung negativer Zahlen

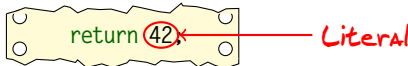
$$\begin{aligned}
 \text{vorzeichenlos:} \quad \text{Wert} &= \sum_{i=1}^n b_i \cdot 2^{i-1} \\
 \text{vorzeichenbehaftet:} \quad \text{Wert} &= \begin{cases} \sum_{i=1}^{n-1} b_i \cdot 2^{i-1} & \text{wenn } b_n = 0 \\ \left(- \sum_{i=1}^{n-1} (1 - b_i) \cdot 2^{i-1} \right) - 1 & \text{wenn } b_n = 1 \end{cases}
 \end{aligned}$$

b_i ist dabei das i . Bit

- Mehr in ➡ LV „Grundlagen Technische Informatik“ (Kapitel 2) oder „Einführung in die Funktion von Computern“ (Kapitel 1)

Literale

- Häufig kommen im Programmtext **Werte** vor, die direkt verwendet werden
- Einen solchen direkten Wert nennt man ein **Literal**⁷

– **Beispiel:**  `return 42;` **Literal**

- Wie jeder Ausdruck muss ein Literal einen Typ haben
- Einer Zahl wie 123 sieht man aber nicht an, ob sie „nur“ eine natürliche Zahl (`unsigned int`) ist, oder als ganze Zahl (`signed int`) aufgefasst werden soll, oder ob sie vielleicht sogar `unsigned long int` sein soll
- Daher gibt es bestimmte Schreibweisen, um den Typ eines Literales festlegen zu können
➡ Nutzung von Suffixen

`int`: Ohne Suffix ➡ Beispiel: 123

`unsigned int`: Wie `int`, aber mit Suffix „u“ bzw. „U“ ➡ Beispiel: 123U

`long int`: Wie `int` mit Suffix „l“ bzw. „L“ ➡ Beispiel: 123L

`long long int`: Wie `int` mit Suffix „LL“ bzw. „ll“ ➡ Beispiel: 123LL

⁷Häufig werden diese Werte auch „Konstanten“ genannt⁸. Da dieser Begriff in C noch zwei(!) weitere Bedeutungen haben kann, soll er hier vermieden werden.

- Bei `unsigned long int` und `unsigned long long int` werden die Suffixe kombiniert, wobei zuerst immer der Suffix für das (fehlende) Vorzeichen kommt ➔ Beispiel : `123ULL`
- Es gibt keine Kennzeichnung für `short`
- Außerdem können Literale in verschiedenen Zahlensystemen angegeben werden: außer dem gewohnten Dezimalsystem ist auch das **Oktalsystem** (Basis: 8) und das **Hexadezimalsystem** (Basis: 16) möglich
- Markierung durch Präfix
 - Ganzzahlliterale die mit dem Präfix `0` beginnen sind im Oktalsystem
 - Ganzzahlliterale die mit dem Präfix `0x` beginnen sind im Hexadezimalsystem
- **Frage:** Warum gerade diese beiden Zahlensysteme?
- **Antwort:** Weil bei beiden (im Gegensatz zum Dezimalsystem) jede Stelle einer ganzen Anzahl von Bits entspricht, was bequem zur Darstellung von Bitmustern ist
- **Beispiel:** 1248 (im Dezimalsystem) entspricht der Bit-Folge 010011100000
 - Oktalsystem: $\underbrace{010}_2 \underbrace{011}_3 \underbrace{100}_4 \underbrace{000}_0 \Rightarrow 02340$
 - Hexadezimalsystem: $\underbrace{0100}_4 \underbrace{1110}_E \underbrace{0000}_0 \Rightarrow 0x4E0$

Zeichen

Warum heißt ein Ganzzahltyp „char“?

- Ursprünglich gedacht für Werte im Bereich des **ASCII-Codes** (7 Bit) bzw. des **ANSI-Codes** (8 Bit) ➔ Character
- C kennt (anders als andere Sprachen) **keinen** expliziten Typ für Zeichen
 - Verwendung liegt **ausschließlich** in der **Interpretation** des Wertes einer Ganzzahl bei der Ausgabe, z.B. entsprechend einem Code
 - Je nach Plattform kann daher der gleiche Wert für unterschiedliche Zeichen stehen
- In modernen Programmen sollte für Zeichen vorwiegend der Typ `wchar_t` (und Unicode) benutzt werden
 - Vereinfacht Internationalisierung
 - Einbindung über Headerdatei `wchar.h`
- Wir werden aber zur Vereinfachung bis auf Weiteres von ASCII/ANSI-Code ausgehen
- Um die Eignung für die Kodierung von Zeichen zu unterstützen, gibt es spezielle Literale für `char` ➔ zu kodierendes Zeichen wird in Hochkomma gesetzt
- **Beispiel:** `'C'` steht für 67

- Für Zeichen im ASCII, die nicht direkt eingegeben werden können, gibt es spezielle Schreibweisen:

Bezeichnung	Schreibweise	Wert	Bezeichnung	Schreibweise	Wert
Newline	\n	10	horizontaler Tabulator	\t	9
Backspace	\b	8	vertikaler Tabulator	\v	11
Wagenrücklauf	\r	13	Seitenvorschub	\f	12
Hochkomma (')	\'	39	Anführungsstiche (")	\"	34
Piep	\a	7	Rückstrich (\)	\\	92
NUL	\0	0	Zeichen mit Hex-Code hh	\xhh	0xhh

- Das folgende Beispiel zeigt, dass ein Zeichen nur eine anders **interpretierte** Zahl ist:

```

1 /* char.c -- interpretation of char type */
2 #include<stdio.h>
3
4 char addchar(char c1, char c2)
5 {
6     return c1+c2;
7 }
8
9 int main()
10 {
11     printf("Ergebnis ist %c mit dem Code %d\n",
12           addchar('a',1),addchar('a',1));
13     return 0;
14 }

```

Interpretation

```
> ./char Ergebnis ist b mit dem Code 98
```

- Anmerkung:** Die printf()-Formatanweisung „%c“ gibt das Zeichen aus, das durch die übergebene Zahl codiert wird

ASCII-Konverter

- Wir können nun ein kleines Programm schreiben, dass uns die Zeichen des ASCII-Codes für gegebene Werte ausgibt:

```

1 // ascii.c -- prints ASCII character
2 #include<stdio.h>
3 #include"aup-main.h"
4
5 int Main(int v){
6     if (v < 32) {
7         printf("Error: code not printable\n");
8         return 1;
9     }
10    if (v > 126) {
11        printf("Error: code not printable\n");
12        return 1;
13    }
14    printf("The value %d codes the character '%c'\n",v,v);
15    return 0;
16 }

```

```

>./ascii 44
The value 44 codes the character ','
>./ascii 65
The value 65 codes the character 'A'
>

```

Code	Zeichen	Code	Zeichen
32	' '	64	@
33	!	65	A
34	"	66	B
35	#	67	C
36	\$	68	D
37	%	69	E
	⋮		⋮

4.2.2 Weitere Grundtypen

Rationale Zahlen

- Als Grundtyp hat C nur (Teilmengen) der **rationale** Zahlen ($\mathbb{Q}$), aber keine irrationalen Zahlen wie z.B. π oder $\sqrt{2}$ (also Zahlen aus der Menge $\mathbb{R} \setminus \mathbb{Q}$)
- Diese Zahlentypen werden nach der verbreiteten „wissenschaftlichen Schreibweise“ auch **Gleitkommazahlen** (*floating point numbers*) genannt

$$1234,5 = 1,2345 \cdot 10^3 = 1,2345E3$$

- C stellt folgende Typen zur Beschreibung von Gleitkommazahlen zur Verfügung:
 - `float`
 - `double`
 - `long double`
- Rationale Zahlen in C haben außer einem beschränkten Wertebereich eine beschränkte **Genauigkeit**
- Wieder wird offiziell nichts festgelegt, außer Mindestwerten
 - `float` besitzt einen Wertebereich von mindestens $\pm 10^{\pm 37}$ und eine Genauigkeit von mindestens 6 Dezimalstellen
 - `double` besitzt mindestens den Wertebereich von `float` und eine Genauigkeit von mindestens 10 Dezimalstellen
 - `long double` ist wenigstens so „gut“ wie `double`
- De facto folgen aber praktisch alle Compiler dem IEEE-754-Standard, der binäre Darstellungen für Gleitkommazahlen spezifiziert
- Auch für Gleitkommaliterale gibt es Schreibregeln:

`double`: Nur als Dezimalzahl mit Ganzzahlteil, Dezimalpunkt, Nachkommateil, Buchstabe „e“ oder „E“ und Exponent

 - Ganz- oder Nachkommateil (nicht beides) kann ausgelassen werden
 - Dezimalpunkt oder Buchstabe+Exponent (nicht beides) kann ausgelassen werden

⇒ Beispiele: `1.23e10`, `.23` oder `1e10`

`float`: Wie `double` mit Suffix „f“ oder „F“ ⇒ Beispiele: `1.23e5f`, `.23F` oder `1e5f`

`long double`: Wie `double` mit Suffix „l“ oder „L“ ⇒ Beispiele: `1.23e5L`, `.23l` oder `1e5L`
- Es gibt in C11 auch Gleitkommaliterale in Hexadezimalschreibweise, wird aber selten genutzt



Achtung

Bei Gleitkommazahlen wird entsprechend der englischen Sprache ein Punkt „.“ als Dezimalzeichen benutzt, nicht (wie im Deutschen) ein Komma „,“.

Der Ausdruck `(3,14 + 2,18)` ist in C zwar syntaktisch korrekt (und gibt daher u.U. keine Warnung), hat aber den Wert 18.

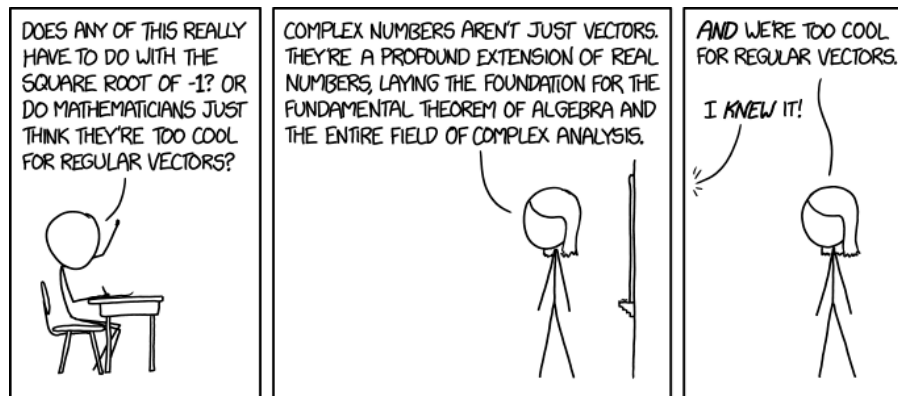
Komplexe Zahlen



- Seit C99 kennt C auch **komplexe Zahlen**
- Drei Varianten, jeweils mit dem Schlüsselwort `_Complex`:
 - `float _Complex`
 - `double _Complex`
 - `long double _Complex`
- In der Headerdatei „`complex.h`“ wird `i` als imaginäre Einheit definiert $\Rightarrow 2 + 3j$ wird als `2+3*I` geschrieben



- C11 kennt noch weitere Möglichkeiten für die Darstellung von komplexen Zahlen



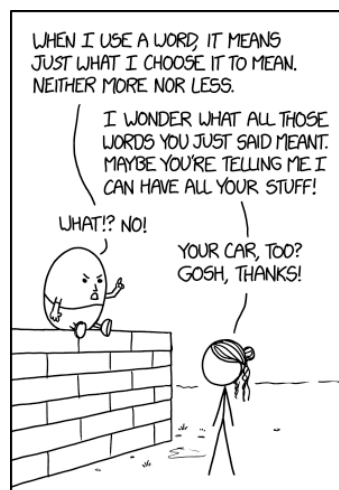
Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/2028/>

Der Typ `void`

- C kennt den Basistyp `void` („`void`“ $\Rightarrow$ *engl.* leer, nichtig)
- `void` ist eigentlich ein Anti-Typ: er wird dann genutzt, wenn man **keinen** Typ haben will $\Rightarrow$ leere Menge
- **Wozu ist das gut?**
 - C kennt nur **Funktionen** als Einheiten der Ausführung, also eine Abbildung von einer Menge ($\Rightarrow$ Funktionsparameter) in eine andere ($\Rightarrow$ Rückgabewert)
 - Mitunter kommt es nur auf die **Ausführung** an, und Parameter oder/und Rückgabewert wird nicht gebraucht.

```
1 #include<stdio.h>
2
3 void say_something(void)
4 {
5     printf("Something!\n");
6     return;
7 }
8
9 int main()
10 {
11     say_something();
12     return 0;
13 }
```

- Bei Funktionen mit Rückgabotyp `void` kann `return` weggelassen werden.
- Ist das einzige Element der Parameterliste `void`, so kann es weggelassen werden



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1860/>

4.3 Typenkonvertierung

- In Ausdrücken (z.B. `a + b`) können verschiedenen Teilausdrücke verschiedene Typen haben, z.B. `unsigned char` und `signed int`
- In diesen Fällen werden Teilausdrücke **automatisch konvertiert**
 - Allgemein gilt: es wird stets auf den höher auflösenden Typ konvertiert, mindestens auf `int`

- Neben der automatischen Konvertierung kann auch eine Konvertierung erzwungen werden, indem der Zieltyp in Klammern vor den Ausdruck gesetzt wird
 - Man spricht von einem **Cast-Operator** (*to cast* = in eine Form gießen)
 - Dabei kann auch ein kleinerer Wert erzwungen werden

Man betrachte den folgenden Code:

```

1  /* cast.c -- type cast */
2  int printf(const char*,...);
3  long ladd(long, long);
4
5  int main()
6  {
7      printf("Ergebnis: %d\n", ladd(23,42));
8      return 0;
9  }
10
11 long ladd(long x, long y)
12 {
13     return x+y;
14 }

```

- Bei der Übersetzung gibt es folgende Warnung:

```

> cc -std=C99 -Wall -pedantic -Wall -Wextra cast.c -o cast
cast.c:7:28: warning: format specifies type 'int' but the argument has
type 'long' [-Wformat]

```

- Die Warnung verschwindet, wenn man in Zeile 7 den Compiler anweist, den Wert als int zu interpretieren

```

7  printf("Ergebnis: %d\n", (int)ladd(23,42));

```

Dynamische Typen und Typinferenz

- Eine Deklaration zur Typenfestlegung wie in C ist nicht in allen Sprachen notwendig
 - In andere Sprachen mit statischen Typen kann der Compiler den Typ ableiten (Typinferenz) ➔ z.B. Haskell, ML
 - Andere Sprachen besitzen **dynamische Typen**, die zur Laufzeit bestimmt werden ➔ z.B. Python

```
>>> def foo(x):  
...     return 2*x  
...  
>>> type(42)  
<type 'int'>  
>>> type(foo)  
<type 'function'>  
>>> type(foo(42))  
<type 'int'>  
>>> type(foo(4.2))  
<type 'float'>  
>>> type(foo('Hallo'))  
<type 'str'>  
>>>
```

Python

- Python kennt auch Ganzzahlen und Gleitkommazahlen
- Darüber hinaus besitzt es Grundtypen, die es in C nicht gibt, z.B.:
 - Zahlen mit beliebiger Größe und Genauigkeit
 - Zeichenketten (Strings)

Merke

Falls es nicht auf Geschwindigkeit ankommt, eignet sich Python bei Problemen wie Textmanipulation oft besser als C.

- Größer sind die Unterschiede bei den abgeleiteten Typen ➡ später

Knobelaufgaben

Aufgabe 4.1

Archäologen finden eine Holztafel mit Symbolen. Allerdings fehlt ein Stück mit einem Symbol. Rekonstruieren Sie das fehlende Symbol!



Aufgabe 4.2

Für eine Abbildung $\mathcal{A}$ gilt:

8809 $\mapsto$ 6	7111 $\mapsto$ 0	2172 $\mapsto$ 0	6666 $\mapsto$ 4
1111 $\mapsto$ 0	3213 $\mapsto$ 0	7662 $\mapsto$ 2	9312 $\mapsto$ 1
0000 $\mapsto$ 4	2222 $\mapsto$ 0	3333 $\mapsto$ 0	5555 $\mapsto$ 0
8193 $\mapsto$ 3	8096 $\mapsto$ 5	7777 $\mapsto$ 0	9999 $\mapsto$ 4
7746 $\mapsto$ 2	6855 $\mapsto$ 3	9881 $\mapsto$ 5	5531 $\mapsto$ 0

Welchen Wert hat x in $2581 \mapsto x$?

Kapitel 5

Variablen, Speicher und Zeiger

One man's constant is another man's variable.

— Alan Perlis

Playing with pointers is like playing with fire. Fire is perhaps the most important tool known to man. Carefully used, fire brings enormous benefits; but when fire gets out of control, disaster strikes.

— John Barnes

5.1 Zustand

- Bisher nutzten unsere C-Programme im Wesentlichen das Funktionsmodell ➔ **zustandsfrei**
- Zur Realisierung des Zustandsmodells ist die Speicherung von Daten notwendig
- Prinzipiell zwei Möglichkeiten:
 - Speicherung im **Dateisystem** (gut für große Datenmengen, aber langsam)
 - Speicherung im **Hauptspeicher** (schneller, aber beschränkter)
- Betrachten hier letzteren Ansatz

Achtung!

C verlangt, dass man sich vielen Fällen explizit Gedanken über Speicherung macht.

Bei anderen Sprachen ist dies nicht so, da sie

- eine automatische Speicherverwaltung besitzen (Logo, Python) und/oder
- das Zustandsmodell gar nicht erst unterstützen (Haskell, ML)

Variablen

- Prinzipiell kann in C jede Stelle des Hauptspeichers gelesen und geschrieben werden
- *Dies ist jedoch gefährlich und wird von vielen Betriebssystemen verhindert*
- Daher gibt es Möglichkeiten, Speicher(bereiche) zu **reservieren**
- Drei prinzipielle Ansätze:
 - **benannten Speicher**, Reservierung zur Übersetzungszeit
 - **anonymen Speicher**, Reservierung zur Laufzeit¹
 - Parameter, Reservierung zur Laufzeit

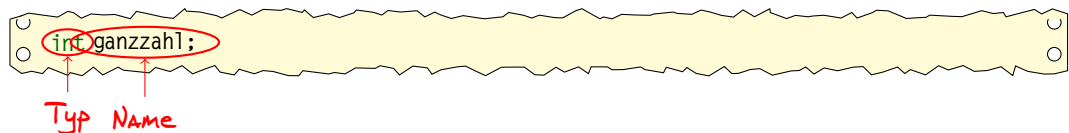
Variablen und Konstanten

Soll der Speicherinhalt eines reservierten Speicherbereichs geändert werden (was genau Sinn des Zustandsmodells ist), nennt man den Speicherbereich eine (benannte oder anonyme) **Variable**.

Soll der Speicherinhalt dagegen unverändert bleiben, spricht man von einer (Laufzeit-) **Konstanten**².

5.2 Benannte Variablen

- Benannte Variablen müssen (wie Funktionen) in C deklariert werden
- Deklaration einer Variable erfolgt analog zur Deklaration einer Funktion
 - **Denkmodell**: Eine Variable ist eine Funktion, die den ihr übergebenen Wert „berechnet“ und zurückgibt
- Die Deklaration **muss** enthalten:
 - Typ der Variablen
 - Name der Variablen



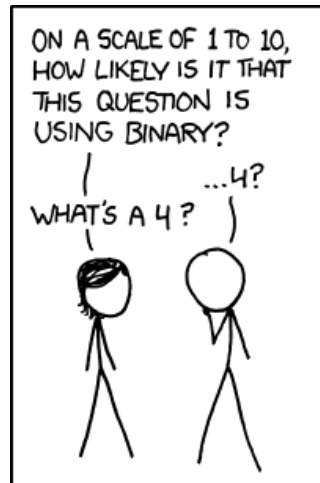
The diagram shows a variable declaration `int ganzzahl;` inside a yellow box with a torn edge. The word `int` is circled in red. Below the box, the text `Typ Name` is written in red, with a red arrow pointing from it to the circled `int`.

- Mehrere Variablen des gleichen Typs können gemeinsam deklariert werden

¹Genau genommen gibt es auch anonymen Speicher, der zur Übersetzungszeit reserviert wird.

²Der Begriff „Konstante“, kann verschiedene Bedeutungen haben: Z.B. werden Literale (siehe Seite 81) als (Übersetzungszeit-)Konstanten bezeichnet.

```
1 int x, y, z;
```



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/953>

Deklaration und Definition

- Bei einer Variablen-Deklaration **kann** zusätzlich eine **Speicherklasse** (*storage class*) und/oder **Speicherattribute** (*type qualifier*) für die Variable angegeben werden
- Die Speicherklasse wird durch eines der Schlüsselwörter
 - auto, static, extern oder register
 deklariert, Speicherattribute durch die Schlüsselwörter
 - const, volatile oder restrict³
- Wir werden die Bedeutung der Speicherklassen und Speicherattribute später betrachten
- Die Deklaration einer Variablen kann erfolgen:
 - **Außerhalb** jeder Funktion ⇒ **globale Variable**
 - Innerhalb⁴ jedes **Blockes** aus geschweiften Klammern „{ ... }“ ⇒ **lokale Variable**
- Mit Ausnahme von Deklarationen, die die Speicherklasse „extern“ tragen, werden benannte Variablen bei der Deklaration auch **definiert**, d.h. Speicherplatz reserviert
 - ⇒ **Welcher** Speicherplatz reserviert wird, legen im Zusammenspiel Compiler, Linker und Betriebssystem fest; Programmierer brauchen sich **nicht** darum zu kümmern



³erst ab C99

⁴Bis C90 dürfen Variablendeklarationen nur an **Beginn** eines Blocks stehen, d.h. **vor** allen anderen Anweisungen in diesem Block.

- Ist bei einer Deklaration einer Variablen die Speicherklasse „extern“ angegeben, so wird die Variable **nur deklariert** (dem Compiler bekannt gegeben)
 - Sie muss dann noch woanders (in einem anderen Modul, vergleiche Kapitel 14) **global** definiert werden

Bezeichner

Bezeichner

Die Namen von Funktionen, Variablen und Typen werden **Bezeichner (Identifier)** genannt.

Jeder Bezeichner darf global oder innerhalb eines Blockes jeweils **nur einmal** definiert sein.

Ein Bezeichner muss mit einem Buchstaben oder dem Unterstrich „_“ anfangen, dem eine beliebige Kombination aus Buchstaben, Zahlen oder Unterstrichen folgen kann.

Ein Schlüsselwort darf nicht als Bezeichner verwendet werden. Auch empfiehlt es sich nicht, Bezeichner aus der Standardbibliothek zu benutzen.

Achtung!

Es gibt Sprachen, bei denen es auf Groß-/Kleinschreibung nicht ankommt. In C sehr wohl: `abc`, `Abc`, `ABC` und `aBC` sind **unterschiedliche** Bezeichner.

Zuweisung

- Einer Variable kann man mit dem „=“-Operator einen Wert zuweisen
- Dies geht auch schon bei der Deklaration ➔ **Initialisierung**

```
1 int x=0;
2 x=42;
```

- Bei der Zuweisung kann auch auf den vorherigen Wert der Variable Bezug genommen werden

~~x~~=2*~~x~~ — vor Zuweisung
nach Zuweisung

Sichtbarkeit

```

1 #include<stdio.h>
2
3 int global=42;
4
5 int main()
6 {
7     int local1,local2;
8     local1=111;
9     local2=222;
10    {
11        int local1=23;
12        printf("global=%d, local1=%d, local2=%d \n",
13              global,local1,local2);
14    }
15    printf("global=%d, local1=%d, local2=%d \n",
16          global,local1,local2);
17    return 0;
18 }

```

```

> ./declare
global=42, local1=23, local2=222
global=42, local1=111, local2=222

```

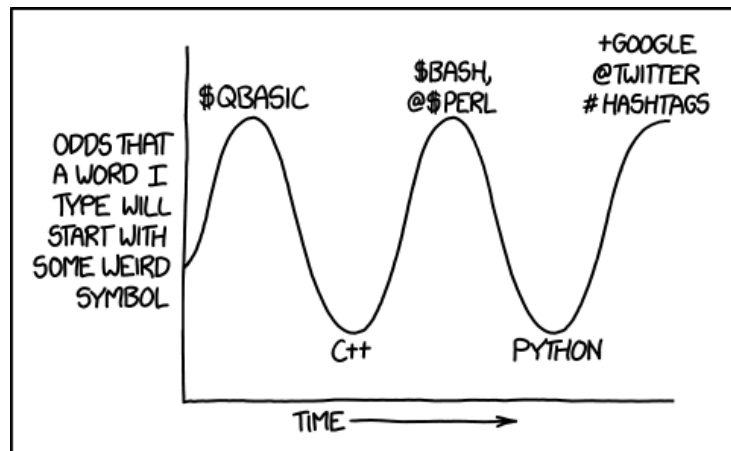
- Jede benannte Variable ist ab der Stelle, an der sie deklariert ist, **sichtbar**
- Sie verliert ihre Sichtbarkeit, wenn
 - der Block, in dem sie deklariert wurde, endet
 - wenn in einem tiefer verschachtelten Block ein gleichlautender Bezeichner deklariert wird ➔ **Überdeckung**

Lebensdauer

Jede benannte Variable hat eine **Lebensdauer (Gültigkeit)**, in der sie den jeweils letzten zugewiesenen Wert behält

Die Lebensdauer...

- ... einer **globalen Variable** ist die gesamte Laufzeit des Programms
- ... einer **automatischen lokalen Variable** die Zeit, in dem sich das Programm in dem Block befinden, in dem sie deklariert ist
- ... einer lokalen Variable der Speicherklasse „static“ (➔ **statische Variable**) die gesamte Laufzeit des Programms



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1306/>

Achtung!

Statische lokale Variablen „überleben“ das Verlassen einer Funktion.

Wird eine statische Variable bei der Deklaration initialisiert, so wird diese Initialisierung *nur einmal* vorgenommen.

- Zum Vergleich: mit `static` ...

```

1  /* static.c -- lifetime of static variables */
2  #include<stdio.h>
3
4  void count(void){
5      static int remember=1;
6      printf("remember=%d\n",remember);
7      remember=remember+1;
8  }
9
10 int main()
11 {
12     count();
13     count();
14     count();
15     return 0;
16 }

```

```
> ./static
remember=1
remember=2
remember=3
```

- ...und ohne static:

```
1 /* nonstatic.c -- lifetime of non-static variables */
2 #include<stdio.h>
3
4 void count(void){
5     int remember=1;
6     printf("remember=%d\n",remember);
7     remember=remember+1;
8 }
9
10 int main()
11 {
12     count();
13     count();
14     count();
15     return 0;
16 }
```

```
> ./nonstatic
remember=1
remember=1
remember=1
```

Python

- Die Regeln für Bezeichner sind die gleichen wie in C (allerdings andere Schlüsselwörter)
- Python besitzt eine **automatische Speicherverwaltung**
- Variablen müssen nicht deklariert werden
- Sie existieren, sobald ihnen ein Wert zugewiesen wird
- Sie sind stets lokal (und überdecken damit ggf. andere), können aber als global vereinbart werden

```
global var
```

- Python nutzt (wie z.B. auch C++) das Konzept von **Namensräumen** (*name spaces*), die eine feingranulare Nutzung auch gleicher Bezeichner erlauben

Beispiel für lokale und globale Variablen in Python:

```

1 var1=42
2 var2=23
3 var3=4711
4
5 def func():
6     global var1
7     var1=1
8     var2=2
9     print("var1=",var1," var2=",var2," var3=",var3)
10    return
11
12 print("var1=",var1," var2=",var2," var3=",var3)
13 func()
14 print("var1=",var1," var2=",var2," var3=",var3)

```

Die Ausführung dieses Codes führt zu:

```

var1= 42 var2= 23 var3= 4711
var1= 1 var2= 2 var3= 4711
var1= 1 var2= 23 var3= 4711

```

5.3 Adressen

- Jede Variable hat eine **Adresse** (⇒ Ort im Speicher)
- Der Operator & gibt die Adresse einer Variablen zurück

```

1 /* addr.c -- address of a variable */
2 #include<stdio.h>
3
4 int main()
5 {
6     int var=42;
7     printf("Variable var has the address %p and the value %d\n",
8           &var,var);
9     return 0;
10 }

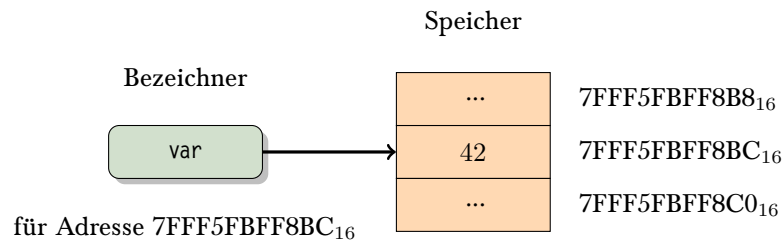
```

```

> cc -std=c99 -Wall addr.c -o addr
> ./addr
Variable var has the address 0x7fff5fbff8bc and the value 42

```


- Tatsächlich ist der Bezeichner einer Variablen nur ein Synonym für eine Adresse



- Konkret wird in C ein Variablenbezeichner als (evtl. noch durch andere Operanden modifizierte) Adresse gesehen und wie folgt behandelt:
 - Er steht **links** eines Gleichheitszeichens $\Rightarrow$ der Wert des Ausdrucks rechts des „ $=$ “ wird in die Speicherstelle mit dieser Adresse **geschrieben**
 - Sonst $\Rightarrow$ der **Inhalt** (Wert) der Speicherstelle wird **gelesen** und weiterverarbeitet
 - Da C mehrere Gleichheitszeichen in einem Ausdruck erlaubt, kann auch beides geschehen

Zeiger

- Variablen können auch selbst Adressen beinhalten
- Durch den Präfix „*“ vor dem Bezeichner wird bei der Deklaration eine Variable zu einer Adressvariablen (**Zeiger**, **Pointer**)
- Es ist für den Compiler wesentlich, **wovon** eine Adresse gebildet wird
- Entsprechend ist ein Zeiger immer ein Zeiger auf einen **bestimmten** anderen Typ
- Z.B.:
 - `int *p;` $\Rightarrow$ Zeiger auf eine Ganzzahl
 - `float *p;` $\Rightarrow$ Zeiger auf eine Gleitkommazahl
 - `unsigned int *p;` $\Rightarrow$ Zeiger auf eine natürliche Zahl
- Wenn nur allgemein eine Speicheradresse (ohne speziellen Typ) gemeint ist, wird als Grundtyp „void“ verwendet, z.B.
 - `void *p;` $\Rightarrow$ Zeiger auf eine Adresse
 - Bei **Zuweisungen** sind void-Zeiger zu allen anderen Zeiger-Typen kompatibel $\Rightarrow$ keine Warnung



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/138/>

```

1 /* addr2.c -- pointer to a variable */
2 #include <stdio.h>
3
4 int main()
5 {
6     int var=42, *pvar;
7     pvar= &var ;
8     printf("Variable var has the address %p and the value %d\n",
9           pvar,var);
10    printf("Variable pvar has the address %p and the value %p\n",
11          &pvar,pvar);
12    return 0;
13 }

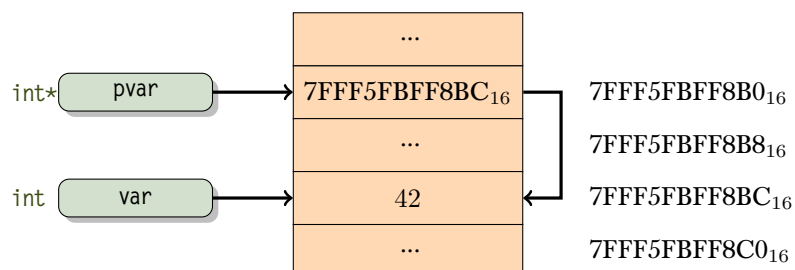
```

- Nach Ignorieren von Warnungen erhält man bei Aufruf beispielsweise:

```

> ./addr2
Variable var has the address 0x7fff5fbff8bc and the value 42
Variable pvar has the address 0x7fff5fbff8b0 and the value 0x7fff5fbff8bc

```



Achtung!

Ein Typ und dessen abgeleiteter Zeigertyp (z.B. `int` und `int*`) sind **unterschiedliche** Typen!

Warum?

Frage: Wozu braucht man Zeiger überhaupt?

Antwort: Auch wenn vielleicht die Notwendigkeit ihrer Nutzung nicht auf Anhieb ersichtlich ist, sind Zeiger in C von großer Bedeutung. Man braucht sie...

- zur Verwaltung von anonymen Variablen, die man wiederum braucht, wenn die Menge an Daten zur Übersetzungszeit noch nicht feststeht (⇒ Abschnitt 5.4)
- zum Nachbau von veränderlichen Parametern (Referenzparametern, ⇒ Abschnitt 5.5)
- zum Aufbau von Beziehungen in komplexen Datenstrukturen (⇒ Kapitel 6 und Vorlesung „Datenstrukturen“)

Dereferenzierung

- Ein Zeiger kann durch den „*“-Operator vor dem Zeiger **dereferenziert** werden
- Damit wird der Inhalt der Speicherstelle angesprochen, auf die der Zeiger zeigt
- Für dereferenzierte Zeiger gelten in Zuweisungen die gleichen Regeln wie für einfache Bezeichner

```
1  /* deref.c -- deref a pointer */
2  #include <stdio.h>
3
4  int main()
5  {
6      int y=23, *py=&y;
7
8      printf("y=%d\n",*py); /* same effect as printf("y=%d\n",y); */
9      *py=42;               /* same effect as y=42 */
10
11     printf("y=%d\n",y);
12     return 0;
13 }
```

L-Value

- Bezeichner und Zeiger sind zwei Möglichkeiten, Speicheradressen von Variablen in C auszudrücken
- Wir werden noch mehr Möglichkeiten kennenlernen
- Daher gibt es ein allgemeines Konzept: L-Value

Definition 5.1: L-Value

Jeder Ausdruck, der eine gültige **getypte** Adresse ergibt, wird **L-Value** (**L-Wert**, **Linkswert**) genannt.

- Die Bezeichnung steht für „*left value*“, da links einer Zuweisung solch ein Ausdruck stehen muss
- Eine ungetypte Adresse (void-Zeiger) ist **kein** L-Value

```

1  /* lvalue.c -- this program does not compile! */
2  int main()
3  {
4      int  i,j;
5      int  *pi;
6      void *pv;
7
8      i   = 42;    /* correct    */
9      pi  = &j;    /* correct    */
10     pv  = pi;    /* correct    */
11     *pi  = 23;    /* correct    */
12     *pv  = 23;    /* NOT correct */
13     42   = j;     /* NOT correct */
14
15     return 0;
16 }

```

- Bei diesem Programm gibt es Compilerfehler, da zweimal keine L-Values auf der linken Seite einer Zuweisung stehen

5.4 Anonyme Variablen

- **Anonyme Variablen** haben keinen Namen; es wird aber Speicherplatz reserviert
- Sie werden mit Hilfe von Funktionen der C-Standardbibliothek **zur Laufzeit** angelegt
- Anonyme Variablen werden **nicht** deklariert
- Nutzung folgender Funktionen, in `stdlib.h` deklariert

REFRESH TYPE	EXAMPLE SHORTCUTS	EFFECT
SOFT REFRESH	EMAIL [REFRESH] BUTTON	REQUESTS UPDATE WITHIN JAVASCRIPT
NORMAL REFRESH	F5, CTRL-R, ⌘R	REFRESHES PAGE
HARD REFRESH	CTRL-F5, CTRL-⇧, ⌘⇧R	REFRESHES PAGE INCLUDING CACHED FILES
HARDER REFRESH	CTRL-⇧-HYPER-ESC-R-F5	REMODEL CYCLES POWER TO DATA CENTER
HARDEST REFRESH	CTRL-⌘-⇧-⌘-R-F5-F5-ESC-O-0-0-⚡-SCROLL LOCK	INTERNET STARTS OVER FROM ARPANET

Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/1854>

- `void *malloc(size_t size)`
Reserviert `size` Bytes im Speicher
- `void *calloc(size_t count, size_t size)`
Reserviert `count`×`size` Bytes im Speicher und initialisiert sie mit dem Wert 0
- Beide Funktionen geben eine **Adresse** der anonymen Variable zurück

Der sizeof-Operator

- **Frage:** Wieviel Platz soll denn bei der Reservierung angefordert werden?
- **Antwort:**
 - Soviel, wie der Typ eines Wertes, der dort gespeichert werden soll, braucht
 - Dies mit Hilfe des `sizeof()`-Operator (vgl. Abschnitt 4.2.1) portabel möglich

Anmerkung

„sizeof“ ist ein Schlüsselwort in C und keine Funktion. Daher unterliegt sizeof() auch den Restriktionen des Typensystems.

Der `sizeof`-Operator kann sowohl auf einen **Typ** als auch auf eine **Variable** angewendet werden. Beispielsweise hat `sizeof(double)` auf MacOSX auf Intel den Wert 8.

Verwaltung anonymer Variablen

Die Verwaltung einer anonymen Variable obliegt dem Programm

- **Lebensdauer:** solange der Programmierer will $\Rightarrow$ Freigabe zur gegebenen Zeit
- **Sichtbarkeit:** keine $\Rightarrow$ der Programmierer muss dafür sorgen, dass die Existenz „gemerkt“ wird

- Freigabe über die Funktion `void free(void*)`, der die Adresse der anonymen Variablen übergeben wird

Man beachte

Die Adresse einer anonymen Variable sollte **stets** in einer anderen Zeigervariable gespeichert werden! Diese Zeigervariable kann ggf. selbst anonym sein, jedoch muss am Ende der Kette aber stets eine benannte Zeigervariable stehen.

Beispiel

```

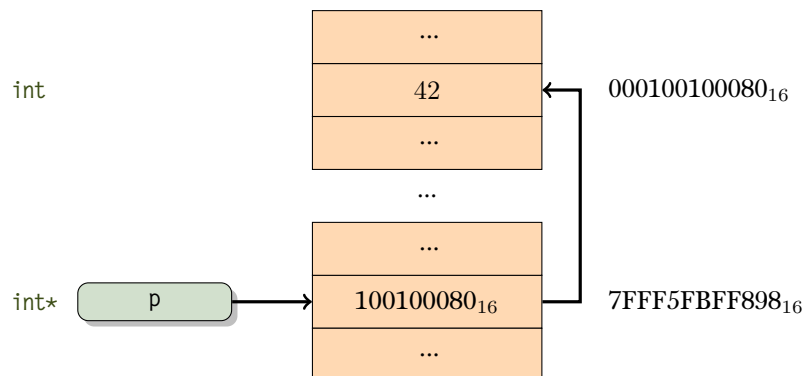
1  /* malloc.c -- anonymous variables */
2  #include<stdio.h>
3  #include<stdlib.h>
4
5  int main()
6  {
7      int *p;
8
9      p= malloc(sizeof(int)); /* reserves memory */
10     *p=42;                  /* set value      */
11     printf("Pointer p has address %p and points to %p\n",
12           (void*)&p,(void*)p);
13     printf("Anonymous variable has the value %d\n",*p);
14     free(p);                /* releases memory */
15     return 0;
16 }

```

```

> ./malloc
Pointer p has address 0x7fff5fbff898 and points to 0x100100080
Anonymous variable has the value 42

```



Fehlerquellen

- Die Nutzung anonymer Variablen hat sich als eine der wesentlichsten Fehlerquellen in C erwiesen
- Daher zwei Warnungen:

Warnung I

Das „**Vergessen**“ und eine in Folge nicht mehr mögliche Freigabe einer anonymen Variable ist eine häufige Fehlerquelle ➔ **memory leak**.

Warnung II

Die Nutzung einer **bereits freigegebenen** anonymen Variable ist ebenfalls eine häufige Fehlerquelle ➔ **dangling pointer**



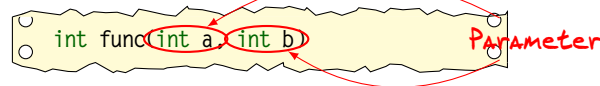
Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/379/>

Zwei Regeln helfen, diese Fehler zu vermeiden:

1. Wenn Sie Code zur Reservierung (z.B. `malloc`) schreiben, schreiben sie **sofort** auch den entsprechenden Freigabecode
2. Wenn Sie eine anonyme Variable freigeben, weisen Sie ihrem Zeiger den (symbolischen) Wert `NULL` zu, der ebenfalls in „`stdlib.h`“ definiert ist
 - Es ist garantiert, dass nie eine Variable an der durch `NULL` beschriebenen Adresse liegt
 - Zur Laufzeit führt die Dereferenzierung von `NULL` zu einem Fehler
 - Falls Bibliotheksfunktionen wie `malloc` aus irgendeinen Grund scheitern, geben sie ebenfalls `NULL` zurück
 - Im Zweifel empfiehlt es sich daher, Zeiger vor dem Gebrauch gegen den Wert `NULL` zu testen

5.5 Parameter

- Neben den benannten – zur Übersetzungszeit reservierten – und den anonymen – zur Laufzeit reservierten – Variablen gibt es noch eine dritte Sorte von Variablen, die wir schon ständig benutzt haben ➔ **Parameter**



- Parameter sind Variablen, die automatisch beim Aufruf einer Funktion **angelegt** und **initialisiert** werden
 - **Lebensdauer**: Verweilzeit des Programms in der Funktions**instanz**
 - **Sichtbarkeit**: kompletter Funktionskörper (falls nicht überdeckt)
- Parametervariablen werden mit der Funktionsdefinition deklariert und definiert
- Sie dürfen keine **Speicherklasse**, aber **Speicherattribute** haben

Parameterübergabe

Definition 5.2: Formale und tatsächliche Parameter

Die Parameter, die innerhalb eines Funktionsrumpfes verwendet werden, heißen **formale Parameter**.

Die Parameter, die beim Aufruf der Funktion verwendet werden, heißen **tatsächliche Parameter**⁵ oder **Argumente**.

- Bei der Verwendung von Parametern in Programmiersprachen gibt es zwei Probleme zu klären:
 - Welche **Zuordnung** gibt es zwischen tatsächlichen und formalen Parametern?
 - Wie erfolgt die **Übergabe** zwischen tatsächlichen und formalen Parametern?

Es gibt in Programmiersprachen verschiedene Varianten der Zuordnung von tatsächlichen und formalen Parametern

- Positionsparameter**: Die Zuordnung ergibt sich aus der Position eines tatsächlichen Parameters bei Funktionsaufruf
- Namensparameter**: Die Zuordnung ergibt sich aus beim Funktionsaufruf benutzten Namen

C

C benutzt **ausschließlich** Positionsparameter.

⁵Mitunter findet man in der (älteren) Literatur die Bezeichnung „aktueller Parameter“, die von der falschen Übersetzung des englischen Ausdrucks „*actual parameter*“ stammt.

Python

In Python können sowohl Positions- als auch Namensparameter benutzt werden.

Es gibt in Programmiersprachen verschiedene Varianten der **Übergabe** von tatsächlichen zu formalen Parametern:

- **Call by value:** Der tatsächliche Parameter wird ausgewertet und der formale Parameter erhält diesen Wert ➔ **Wertübergabe**
- **Call by name:** Der formale Parameter wird durch den Namen des tatsächlichen Parameters ersetzt ➔ **Namensersetzung**
- **Call by reference:** Der formale Parameter wird zu einem „Stellvertreterobjekt“ für den tatsächlichen Parameter, so dass alle Änderungen sofort außerhalb der Funktion wirksam werden ➔ **Referenzübergabe**
- **Call by copy/return**⁶: Der tatsächliche Parameter wird ausgewertet und der formale Parameter erhält diesen Wert zu Beginn der Funktion; bei Beendigung erhält der tatsächliche Parameter den Wert des formalen Parameters

C

In C gibt es **ausschließlich** eine Form der Parameterübergabe: „Call by value“ (Wertübergabe).

Python

Python kennt (eigentlich) ausschließlich Referenzparameter⁷ (call by reference).

Jedoch unterscheidet Python zwischen „**unveränderlichen**“ (immutable) Typen (wie z.B. Ganzzahlen oder Zeichenketten) und „**veränderlichen**“ (mutable) Typen. Erstere werden bei der Funktionsübergabe wie Werteparameter behandelt.

Ersatz von Referenzparametern

- C kennt **keine** Referenzparameter
- ➔ Konzept von **Prozeduren** (wie z.B. `var`-Parameter in Pascal) **nicht** (ohne Weiteres) realisierbar
- **Idee:** Statt eines Referenzparameters wird ein **Zeiger auf das Argument** übergeben
 - Durch Dereferenzierung können Änderungen von Daten **außerhalb** der Funktion erreicht werden

⁶In der Literatur auch: „call by value/return“, „call by value and result“ oder „copy in/copy out“

⁷Die Wirklichkeit ist etwas komplizierter, siehe Exkurs etwas später in diesem Abschnitt.

- ➔ Wirkung entspricht (im Wesentlichen) Referenzparametern
- Die Nutzung von Zeigern statt Referenzparametern ist eine **häufige Praxis** in C

Definition 5.3: Seiteneffekt

Eine Speicheränderung außerhalb der lokalen Variablen einer Funktion (zu der auch die Parameter zählen) oder eine Ein-/Ausgabe nennt man einen **Seiteneffekt**.

- Dieser Begriff „Seiteneffekt“ stammt von der inkorrekten Übersetzung des Begriffes „*side effect*“
- Veränderungen **außerhalb des Programms** (z.B. Ausgaben) werden auch zu den Seiteneffekten gezählt

```

1 /* sideeffect.c -- simulated reference parameter */
2 #include<stdio.h>
3
4 void sideeffect(int *);
5
6 int main()
7 {
8     int x=42;
9     printf("x=%d\n",x);
10    sideeffect(&x);
11    printf("x=%d\n",x);
12    return 0;
13 }
14
15 void sideeffect(int *p)
16 {
17     *p=23;
18 }

```

```

x=42
x=23

```

scanf()

- Ein Einsatzfall, bei dem man Referenzparameter braucht, ist die Eingabe von mehr als einem Wert
- Die Funktion `scanf()` aus der Standardbibliothek übernimmt genau diese Funktion
- Ähnlich wie `printf()` beginnen die Parameter mit einem Formatstring, gefolgt von einer **Liste** von Zeigern auf die Variablen

- Rückgabewert ist die Anzahl der korrekt gelesenen Variablen

Formatierung

Von welchem Typ ein einzulesender Wert ist, wird durch einen Ausdruck bestimmt, der mit einem Prozentzeichen beginnt, u.a.:

%d	ganzzahliger Dezimalwert	%i	Ganzzahl, Basis hängt vom Präfix ab
%f	Gleitkommazahl	%s	Zeichenkette

Die %-Ausdrücke können auf verschiedene Arten modifiziert werden. Eine Übersicht gibt Anhang B.4.

```

1 /* scanf.c -- input via scanf */
2
3 int scanf(char*,...);
4 int printf(char*, ...);
5
6 int main(){
7     int x,y;
8     printf("Give the point as \"(x,y)\": ");
9     if(scanf("(%d,%d",&x,&y) == 2){
10         printf("You provided: (%d,%d).\n",x,y);
11     }
12     return 0;
13 }

```

```

./scanf
Give the point as "(x,y)": (10,4)
You provided: (10,4).

```

Exkurs: Variablen und Referenzen in Python

Das Folgende stellt einen Exkurs dar und ist erst im Rahmen des Konzepts der Objektorientierung (⇒ Vorlesung Datenstrukturen) relevant.

Falls Sie Probleme beim Verständnis haben, können Sie zum Kapitel 6 weitergehen.

- Die nutzersichtbaren Variablen in Python sind eigentlich **Referenzen** auf **anonyme** Variablen (Objekte), die wiederum vom Python-Laufzeitsystem verwaltet werden
- Es gibt eine zum &-Operator ähnliche Funktion, die die Identität eines Objektes zurückgibt: `id()`
 - In manchen Implementationen (z.B. CPython) ist es tatsächlich die Speicheradresse

- ein **neues Objekt** erzeugt und referenziert
oder
- falls ein derartiges Objekt **bereits existiert**, dieses ggf. referenziert
- Im letzten Beispiel gab es schon ein Objekt, das der Ganzzahl 42 entsprach und von `b` referenziert wurde
- Einige komplexere Objekte sind **veränderlich** (*mutable*)
 - Bei Operationen mit ihnen werden sie verändert und kein neues Objekt angelegt
 - Die Referenz bleibt erhalten
- Beim Parameterruf wird die Referenz des Objekts übergeben
 - *De facto*: Kopie der bei Aufruf verwendeten Referenz
- Bei unveränderlichen Objekten führen Berechnungen in der Funktion zu Referenzierung **anderer** Objekte
- Außerhalb wird noch das gleiche Objekt referenziert ➔ Wirkung ist wie bei **Werteübergabe**
- Bei veränderlichen Objekten wird keine neue Referenz gebildet ➔ **Referenzübergabe** bleibt sichtbar

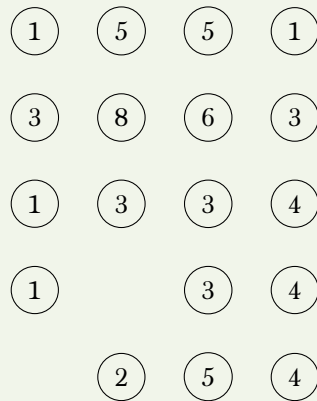


Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1906>

Knobelaufgaben**Aufgabe 5.1**

Drei Damen treffen sich zu einer Sitzung: Frau Rot, Frau Weiß und Frau Grün. Eine der Damen stellt fest: "Das ist aber merkwürdig, eine von uns trägt eine rote, eine andere eine weiße und die dritte eine grüne Bluse". "Das ist wirklich erstaunlich", meint die Dame mit der roten Bluse, "denn keine trägt die Bluse, welche ihrem Namen entspricht". "Das stimmt", stellt Frau Weiß fest.

Welche Dame trägt welche Bluse?

Aufgabe 5.2

Die Inseln in der Abbildung sollen durch Brücken (Linien) miteinander verbunden werden, die strikt nur in Nord-Süd- oder Ost-West-Richtung gehen (d.h. waagerecht oder senkrecht). Eine Brücke darf immer nur zur direkten Nachbarinsel führen. Es dürfen auch zwei parallele Brücken gebaut werden (Doppellinie), aber nicht mehr als zwei.

Verbinden Sie die Inseln so, dass zu jeder Insel genau die Anzahl von Brücken führen, die der Nummer entspricht!

Kapitel 6

Abgeleitete Typen

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

— Linus Torvalds

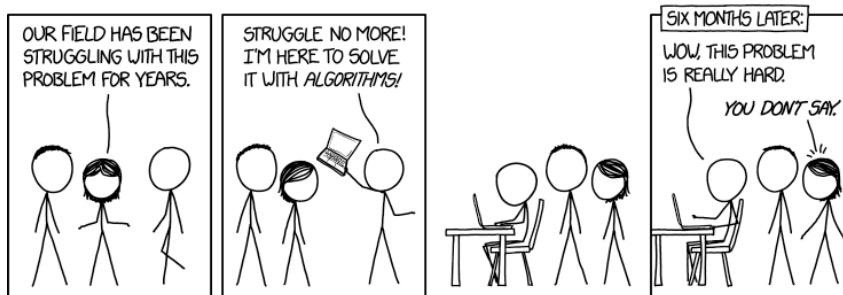
I thrive in structure. I drown in chaos.

— Anna Kendrick

Als maschinennahe Sprache ist C sehr auf sogenannte skalare Typen orientiert, insbesondere auf ganze Zahlen. Jedoch ist es günstig, wenn sich Beziehungen in Problemstellungen auf der Ebene des Datentyps darstellen lassen – skalare Typen bieten da nur sehr eingeschränkte Möglichkeiten.

Allerdings bietet C (wie viele andere Programmiersprachen auch) die Möglichkeit, eigene Typen zu erstellen,

- C kennt sogenannte **abgeleitete Typen**
- Eine Art von abgeleiteten Typen haben wir schon kennengelernt: Zeiger
- Andere Typen sind:



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1831/>

- Verbund-Datentypen (*struct* und *union*)
- Aufzählungstypen (*enum*)
- Felddatentypen (*arrays*)
- Abgeleitete Typen werden manchmal auch „komplexe Typen“ genannt

6.1 Verbundtyp *struct*

- Mit Hilfe des *struct*-Typs können mehrere Variablen (Elemente, *members*) unterschiedlichen Typs im Zusammenhang behandelt werden
- Eine Deklaration besteht aus dem Schlüsselwort *struct*, einem **Namen** (Tag), einer Liste von **Deklarationen** von Elementen bereits bekannter Typen in einem Block aus geschweiften Klammern

```

1 struct point{
2     int x;
3     int y;
4 };

```

- Für Pascal-Kenner: Der *struct*-Typ entspricht etwa dem *record*
- Mit der Deklaration ist der **Typ** bekannt – mit ihm können auf die übliche Weise Variablen deklariert/definiert werden:

```

1 struct point pt;

```

- Typen- und Variablendeklaration kann auch zusammen geschehen:

```

1 struct point{
2     int x;
3     int y;
4 } pt;

```

- In diesem Fall kann der Tag (Typname) entfallen

Vorsicht

Wenn der Tag weggelassen wird...

- ... kann keine weitere Variable dieses Typs deklariert werden
- ... ist selbst ein strukturell identischer Typ **nicht** kompatibel

- Auf ein Element eines *struct* kann mit dem „.“-Operator zugegriffen werden

- Form: variable_name.member_name

- Beispiel:

```
1 /* struct.c -- access to members */
2 #include<stdio.h>
3
4 struct point{
5     int x;
6     int y;
7 } pt;
8
9 int main()
10 {
11     pt.x=42;
12     pt.y=23;
13     printf("member of pt: x=%d, y=%d\n",
14           pt.x,pt.y);
15     return 0;
16 }
```

- Eine struct-Variable kann mit einer Liste von konstanten Ausdrücken initialisiert werden, die in geschweiften Klammern steht

```
1 /* struct2.c -- initialization of a struct */
2 #include<stdio.h>
3
4 struct point{
5     int x;
6     int y;
7 } pt = { 42, 25-2};
8
9 int main()
10 {
11     printf("member of pt: x=%d, y=%d\n",
12           pt.x,pt.y);
13     return 0;
14 }
```

- Elemente können auch¹ über ihren Namen (mit vorgesetztem Punktoperator) angesprochen werden, so dass eine nur teilweise Initialisierung oder eine andere Reihenfolge der Elemente möglich ist:



¹ab C99

```

1  /* struct2a.c -- C99 initialization */
2  #include<stdio.h>
3  #include<stdlib.h>
4
5  struct point{
6      int x;
7      int y;
8  };
9
10 int main()
11 {
12     struct point pt = { .y=23, .x=42 };
13
14     printf("member of pt: x=%d, y=%d\n",
15           pt.x,pt.y);
16     return 0;
17 }

```

- Für eine `struct`-Variable sind nur wenige Operationen erlaubt:
 - Sie darf zugewiesen/kopiert werden („=“-Operator)
 - Ihre Adresse darf ermittelt werden („&“-Operator)
 - Auf ihre Elemente darf zugegriffen werden („.“-Operator)

Zeiger auf struct

In C kommt es häufig vor, dass Zeiger auf (in der Regel anonyme) `struct`-Variablen benutzt werden.

Diese Konstruktion ist die Basis fast aller abstrakter Datentypen (ADT) in C. ADTs werden Sie im nächsten Semester (Datenstrukturen) genauer betrachtet.

Anmerkung

Andere Programmiersprachen wie C++, C#, Rust, Haskell oder Java kennen wesentlich elegantere Konzepte für ADTs: Vererbung, Typenklassen, Generics, Template.

Betrachten die Nutzung von Zeigern auf Strukturen:

```

1  /* struct3.c -- anonymous struct */
2  #include<stdio.h>
3  #include<stdlib.h>
4
5  struct point{
6      int x;
7      int y;
8  };
9
10 int main()
11 {
12     struct point *ppt = malloc(sizeof(struct point));
13     (*ppt).x=42;
14     (*ppt).y=23;
15     printf("member of pt: x=%d, y=%d\n",
16           (*ppt).x,(*ppt).y);
17     return 0;
18 }

```

- Für das Konstrukt `(*pointer).member` gibt es einen Extraoperator: „`->`“
- Der folgende Code ist also äquivalent zu `struct3.c`:

```

1  /* struct4.c -- -> operator */
2  #include<stdio.h>
3  #include<stdlib.h>
4
5  struct point{
6      int x;
7      int y;
8  };
9
10 int main()
11 {
12     struct point *ppt = malloc(sizeof(struct point));
13     ppt->x=42;
14     ppt->y=23;
15     printf("member of pt: x=%d, y=%d\n",
16           ppt->x,ppt->y);
17     return 0;
18 }

```

Typenalias mit typedef

- Die ständige Nutzung des Schlüsselworts `struct` kann umgangen werden, indem ein **Typenalias** definiert wird
- Dazu gibt es das Schlüsselwort `typedef`
- Allgemeine Form: `typedef type alias_name;`

```

1  /* typedef.c -- define alias types */
2  #include<stdio.h>
3  #include<stdlib.h>
4
5  typedef unsigned short int usint_t;
6  typedef struct {
7      usint_t x;
8      usint_t y;
9  } point_t;
10
11 int main()
12 {
13     point_t pt={42,23};
14
15     printf("member of pt: x=%d, y=%d\n", pt.x,pt.y);
16     return 0;
17 }

```

Verbundtyp union

- Der Verbundtyp `union` ist ähnlich zum Typ `struct`
- **Unterschied:** *Alle Elemente liegen an der gleichen(!) Adresse*
 - ➔ Das Schreiben eines Elements **zerstört** den Inhalt der anderen
- Wird nur selten gebraucht (typischerweise zur Optimierung oder für hardwarenahe Programmierung) und wird hier nicht weiter betrachtet

6.2 Aufzählungstyp

- Mit Hilfe von **Aufzählungstypen** können Typen mit diskreten benannten Werten definiert werden
- Schlüsselwort: `enum` (für *enumeration*)
- Beispiel:

```
enum card_suit {club, spade, heart, diamond};
```

- Für alle praktischen Belange ist ein Aufzählungstyp ein `int`
- Man kann gezielt Werte zuweisen:

```
enum month {jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};
```

```
enum escape {NL='\n', BACKSPACE='\b', TAB='\t', RETURN='\r', BELL='\a'};
```

- `enum`-Werte werden zur **Übersetzungszeit** festgelegt und bilden damit eine Möglichkeit, **Konstanten** zu definieren
 - Dies geht aber **nur** für Ganzzahl-Konstanten
- Im Gegensatz zu `const` (siehe später in diesem Abschnitt) benötigt ein „ungebrauchtes“ `enum` keinen Speicherplatz zur Laufzeit
- *Achtung:* Da `enums` im Wesentlichen wie `ints` behandelt werden, bieten sie nur eine eingeschränkte Typensicherheit

6.3 Arrays

- Häufig werden zusammengehörige gleichartige Werte benötigt
- In der Mathematik mit Index: $x_1, x_2, \dots$
- Beispiele:
 - Komponenten eines Vektors (Index: Dimension)
 - Die jeweiligen Tagesumsätze eines Monats (Index: Tag)
 - Die Pixel eines Bildschirms (Indizes: x-Achse, y-Achse)
 - ...
- C stellt dafür den abgeleiteten **Arraytyp** (häufig auch: **Feldtyp**) zur Verfügung
- Ähnlich zum Pascal-Typ „array“
- Deklaration benötigt kein Schlüsselwort, sondern geschieht mit Hilfe des **Indexoperators** (eckige Klammern), der die Elementzahl enthält und hinter den Variablenbezeichner² geschrieben wird
- Deklaration eines Arrays von 10 Ganzzahlen:

²Für Java-Programmierer: Achtung, nicht dem Typ!

```
int xa[10];
```

- Der Grundtyp eines Arrays können beliebige andere Typen sein
- Betrachte folgende Deklarationen:

```
1 int    xa[10]; // Array of 10 integers
2 int    *pxa[10]; // Array of 10 pointers to integers
3 double  f[10]; // Array of 10 floating point numbers
4 struct point{
5     int x;
6     int y;
7 } pt[10];      // Array of 10 point structs
```

- Auch der Zugriff auf Element eines Array erfolgt über den Indexoperator

```
int x1=xa[0];
```

Achtung!

Eine Array mit n Elementen hat **immer** die Indizes 0 bis $n - 1$.
Der Zugriff auf das Element n ist ein **Fehler!**

Mehrdimensionale Arrays

- Auch mehrdimensionale Arrays sind möglich
- Konzept: Array eines anderen Array-Typs:

```
int xa[4][3]; /*declares a 4x3 array */
```

- Das Beispiel deklariert ein 4×3 -Array
- Genauer: Es wird ein Array mit 4 Elementen deklariert, wobei jedes Element ein Array mit 3 Elementen ist

Achtung

Eine Separierung der Indizes durch Komma oder Semikolon (z.B. wie in Pascal) funktioniert nicht! $\Rightarrow$ ~~xa[4,3]~~

- **Beispiel:** Eine Tabelle von ganzen mit fünf Spalten und drei Zeilen lässt sich so deklarieren:

```
int t[3][5];
```

- Die Elemente würden wie folgt angesprochen:

t[0][0]	t[0][1]	t[0][2]	t[0][3]	t[0][4]
t[1][0]	t[1][1]	t[1][2]	t[1][3]	t[1][4]
t[2][0]	t[2][1]	t[2][2]	t[2][3]	t[2][4]

- Dies Prinzip lässt sich auf beliebig viele Dimensionen ausweiten

```
int xa[4][3][2]; /*declares a 4x3x2 array */
```

- Arrays können damit zur Beschreibung von vielen Dingen genutzt werden, z.B. Listen, Vektoren, Tabellen, Matrizen, Determinanten, Koordinatenräumen, ...

Variable Arrays

- Seit C90 können lokale Arrays angelegt werden, deren Größe erst zur Laufzeit ermittelt wird (variable length array = **VLA**)

```
1 int func(int s){
2     double a[s];
3     // ...
```

- Jedoch wird diese Möglichkeit in C11 nur noch als optional zugelassen, so dass man sich nicht unbedingt darauf verlassen sollte, wenn Portabilität angestrebt wird
- VLA haben einige Einschränkungen:
 - Sie können nicht statisch (Schlüsselwort `static`) sein
 - Der Steuerfluss darf niemals **hinter** die Deklaration eines VLA in dessen Gültigkeitsbereich gelangen
 - VLAAs können nicht Teil einer `struct` sein
- VLAAs können auch mehrere Dimensionen haben: `int array[x][y]`



Initialisierung

- Initialisierung:** wie `struct`-Typen können Arrays über über Listen nicht-variabler Ausdrücke in geschweiften Klammern initialisiert werden:

```
int xa[4]={0,1,2,3}; /*initialize elements with their indices */
```

- Dies funktioniert auch bei mehrdimensionalen Arrays:

```

1 int xa[3][4]={
2     {0,1,2,3},
3     {1,2,3,4},
4     {2,3,4,5}}; /* initialize elements with sum of their indices */

```

- Wenn ein Array bei der Definition initialisiert wird, kann die letzte (äußerste) Dimension weggelassen werden
- Das folgende Beispiel ist äquivalent zum Beispiel weiter oben:

```

int xa[]={0,1,2,3}; /*initialize elements with their indices */

```

Achtung!

Man beachte, dass die Dimensionsdeklaration von innen nach außen gelesen werden, der linkeste Wert ist also die letzte (äußerste) Dimension: `int x[4][2]` deklariert entsprechend ein Array mit vier Elementen, die jeweils wieder Integerarrays mit zwei Komponenten sind.

- Entprechen funktioniert...

```

int xa[][2]={1,2},{3,4},{5,6},{7,8}};

```

- ...aber nicht:

```

int xa[4][]={1,2},{3,4},{5,6},{7,8}};

```



- Seit C90 ist auch eine teilweise Initialisierung möglich
- Die Reihenfolge ist dabei egal

```

1 int a[10] = { [3]=42,[2]=23 };

```

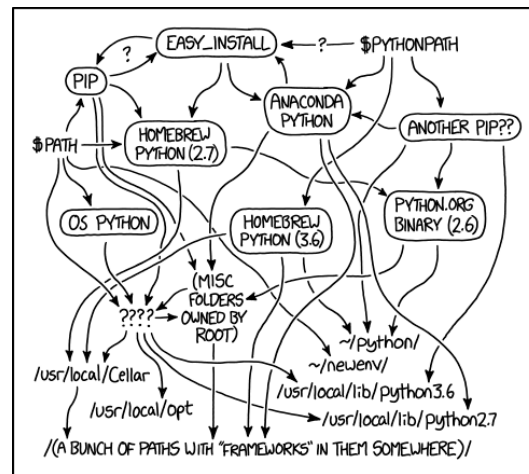
- Alle nicht explizit initialisierte Elemente sind dann 0
- Beide Initialisierungsarten lassen sich mischen
- Wie bei `enum` wird die indexlose Initialisierung für das nächst Element fortgesetzt

```

1 int a[8] = { [3]=42,4,5, [0]=23 };

```

- ...ergibt: 23, 0, 0, 42, 4, 5, 0, 0



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1987/>

Array als Parameter

- Arrays sind gegenüber anderen Typen benachteiligt: Sie lassen sich **nicht** an Funktionen übergeben
- **Genauer:** Man kann zwar Parameter als Array deklarieren, in Wirklichkeit wird aber ein **Zeiger auf den Grundtyp** des Arrays übertragen!
- Man sagt: Das Array „zerfällt“

```

1  /* array2.c -- an array decays */
2  #include <stdio.h>
3
4  typedef int myarray[10];
5
6  void func(myarray a){
7      printf("Size of a: %ld\n", sizeof(a));
8  }
9
10 int main() {
11     myarray x;
12     printf("Size of x: %ld\n", sizeof(x));
13     func(x);
14     return 0;
15 }

```

```
> ./array2
Size of x: 40
Size of a: 8
```

- Zum Glück funktioniert der Indexoperator (eckige Klammern) auch mit Zeigern
 - Array und Zeiger sind *kompatible* Typen
- Deshalb kann (ähnlich wie bei der Initialisierung) die Größe (der letzten Dimension) eines Arrays bei der Übergabe weggelassen werden, oder gleich ein Zeiger übergeben werden
- Die folgenden Deklarationen sind weitgehend äquivalent:

```
1 void func(double[]); // parameter is array of double
2 void func(double*);  // parameter is pointer to double
```

- **Problem:** Es besteht keine Möglichkeit herauszufinden, wie groß das Array vor der Übergabe war
- Deshalb benötigen viele Funktionen, die Array als Parameter haben, als zusätzlichen Parameter die Anzahl der Elemente

Beispiel:

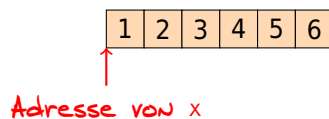
```
1 /* array_param.c -- auxiliary parameter */
2 #include<stdio.h>
3
4 void print_int_array(int,int[]);
5
6 int main() {
7     int xa[3]={1,2,3};
8     print_int_array(3,xa);
9     return 0;
10 }
11
12 void print_int_array(int count,int array[]) {
13     int i=1;
14     printf("(%d",array[0]);
15     while(i<count) {
16         printf(",%d",array[i]);
17         i=i+1;
18     }
19     printf(")\n");
20 }
```

```
> ./array_param.c
(1,2,3)
>
```

- *Achtung!* Falsche Annahmen über Array-Größen sind ein häufiger Programmierfehler

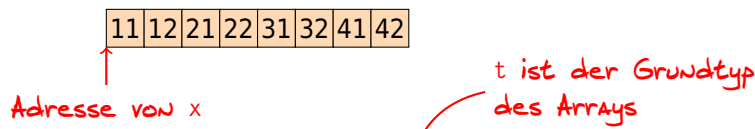
Speicherlayout

- Warum können wir ein Array als Pointer „auffassen“? ➔ festgelegtes Speicherlayout
- Die Elemente eines Arrays liegen hintereinander im Speicher: `int x[6]={1,2,3,4,5,6};`



- Dies gilt auch für mehrdimensionale Arrays:

`int x[4][2] = {{11,12},{21,22},{31,32},{41,42}};`



- Genau genommen: bei einem mit `t a[10];` deklariertem Array ist `a[i]` nur eine andere Schreibweise für `*(a+(t)(sizeof(t)*i))`

Parameterübergabe bei mehrdimensionalen Array

- Aufgrund des beschriebenen Arrayzerfalls ist man evtl. versucht, ein mehrdimensionales Array wie im folgendem Beispiel zu übergeben:

```

1 #include<stdio.h>
2
3 void print_2d(int **arr, size_t x, size_t y){
4     printf("arr[%lu][%lu] = %d\n",
5           x-1,y-1,arr[1][1]);
6 }
7
8 int array[4][4] = {{0,1,2,3},{4,5,6,7},
9                   {8,9,10,11},{12,13,14,15}};
10
11 int main(){
12     print_2d(array,4,4);
13 }

```

- Bei der Übersetzung erhalten wir die Warnung:

```

> cc -std=c99 -pedantic -Wall -Wextra fail2d.c -o fail2d
fail2d.c:17:12: warning: incompatible pointer types passing 'int [4][4]' to parameter of
type 'int **' [-Wincompatible-pointer-types]

```

- ...und bei der Ausführung:

```
>./fail2d
segmentation fault ./fail2d
>
```

- Der Arrayzerfall ist also nicht transitiv über die Dimensionen!
- Korrekt wären dagegen die folgenden beiden Versionen:

```
1 void print_2d(int arr[][4], size_t x, size_t y){
2     printf("arr[%lu][%lu] = %d\n",
3           x-1,y-1,arr[1][1]);
4
5     :
6 }
```

```
1 void print_2d(int (*arr)[4], size_t x, size_t y){
2     printf("arr[%lu][%lu] = %d\n",
3           x-1,y-1,arr[1][1]);
4
5     :
6 }
```

- Da bei der Deklaration von Arrays – auch als Parameter – nur die erste Dimension ausgelassen werden darf, gibt es keine elegante Möglichkeit, Array-Parameter mit mehreren unbekannten Dimensionen zu deklarieren ➔ Rückfall auf Zeiger-Arithmetik
- Eine Parameterdeklaration wie `char **arr` kann aber durchaus sinnvoll sein, wie im Abschnitt 6.4 gezeigt wird

6.3.1 Spezialfall Zeichenarray

- Wie schon erwähnt, kennt C keine Zeichenketten
- Daher werden `char`-Arrays als Zeichenketten „missbraucht“
- Dafür kennt C einen Spezialfall der Initialisierung

```
char str[]="Hallo!";
```

ist äquivalent zu

```
char str[]={'H','a','l','l','o','!','0'};
```

angehängter Wert "0"

- Der Wert 0 (oder `'\0'`) wird als End-Marke genutzt
 - ⇒ Er darf in keiner Zeichenkette enthalten sein
- **Genauer:** "Hallo!" ist ein **Stringliteral**, der in der internen Darstellung mit einer 0 endet

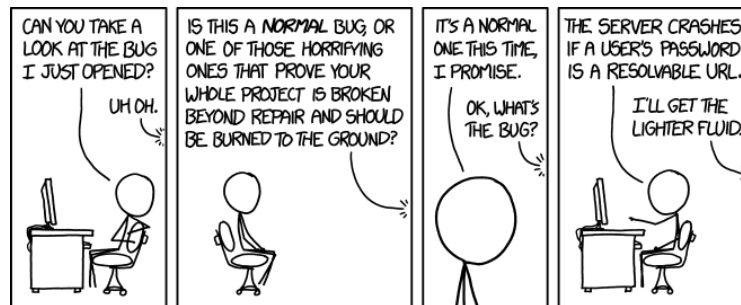
Stringfunktionen

- Die Null am Ende ist eine **Konvention**
- Viele Funktionen der Standardbibliothek verlassen sich darauf, dass `char`-Array sogenannte **C-Strings** sind
- Wir haben zwei kennengelernt:
 - `int printf(char*,...)`
 - `int scanf(char*)`
- Eine Konvention kann man brechen ⇒ **Gefahr**

```
1 /* printf-failure -- danger of C strings */
2 #include<stdio.h>
3
4 char str[]={'H','a','l','l','o','!'};
5 char bla[]="\nDas ist top secret.\n";
6
7 int main(){
8     printf(str);
9     return 0;
10 }
```

```
> ./print-failure
Hallo!
Das ist top secret.
```

- Moderne Compiler wie clang erkennen das Problem, geben eine Warnung aus und korrigieren den Fehler automatisch



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1700/>

6.4 Parameter von `main()`

- Nun haben wir alles zusammen, die Parameter der `main()`-Funktion zu betrachten
- Ihr werden die Befehlszeilenparameter als C-String übergeben
- Die korrekte Signatur von `main()` ist³:

```
int main(int argc, char *args[]);
```

- Rückgabetype `int`: wird an aufrufendes Programm oder Betriebssystem gegeben, Konvention ein Fehlercode
- `int argc`: Anzahl der Parameter des Programms
 - Da immer der Aufrufname des Programms mit übergeben wird, ist `argc` mindestens 1
- `char *args[]`: Array von Zeigern auf C-Strings
 - Jeder dieser C-Strings enthält ein Kommandozeilenargument (inklusive Programmname)

```
1 /* main.c -- gives command line arguments */
2 #include <stdio.h>
3
4 int main(int argc, char *args[])
5 {
6     printf("Program name: %s\n", args[0]);
7     if (argc > 1) { printf("1. argument: %s\n", args[1]); }
8     if (argc > 2) { printf("2. argument: %s\n", args[2]); }
9     if (argc > 3) { printf("3. argument: %s\n", args[3]); }
10    if (argc > 4) { printf("4. argument: %s\n", args[4]); }
11    if (argc > 5) { printf("5. argument: %s\n", args[5]); }
12    if (argc > 6) { printf("There are even more arguments.\n"); }
```

³Die eigentliche Deklaration ist noch etwas komplizierter.

```

13  return 0;
14  }

```

```

> ./main 1 alpha "Ein kurzer Satz."
Program name:./main
1. argument: 1
2. argument: alpha
3. argument: Ein kurzer Satz.

```

Die Deklaration...

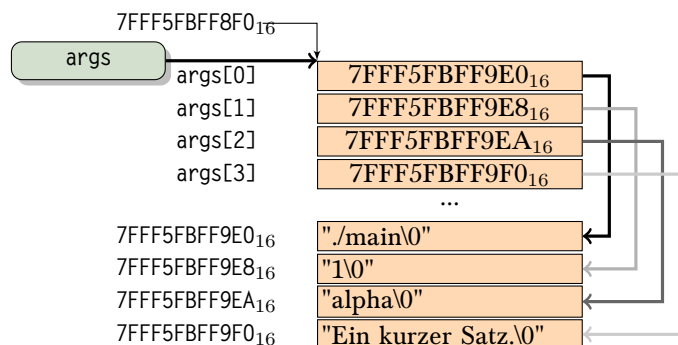
```

char *args[]

```

... bedeutet Array von Zeigern auf `char`.

Beim obigen Aufruf ergibt sich (beispielsweise) folgendes Speicherlayout:



Konvertierung

- **Frage:** Wenn an das Programm (und damit an `main()`) nur Zeichenketten übergeben werden, wie konnte dann `Main()` aus „`aup-main.h`“ mit Ganzzahlen aufgerufen werden?
- **Antwort:** Auch hier wurden Zeichenketten übergeben, jedoch wurde jeder Parameter mit der Funktion `atoi()` konvertiert
 - `atoi()` steht für „*ascii to integer*“ und wandelt eine Zeichenkette in den durch die Zeichen im Dezimalsystem dargestellten Wert um
 - In der Standardbibliothek, Deklaration in „`stdlib.h`“
 - Weitere Information mit `man atoi`

6.5 Komplexe Typen in Python

- Es gibt in Python eine Reihe von vordefinierten Typen, die ähnliche (und weitere) Fähigkeiten wie die besprochenen abgeleiteten C-Typen haben

Merke:

Da „Ableitung“ in der Objektorientierung etwas anderes bedeutet und die hier besprochenen Typen auch keine Ableitungsbeziehung im Sinne von C haben, wird nur der Begriff „komplexe Typen“ verwendet.

- Betrachten (kurz):
 - Strings (Zeichenketten)
 - Listen
 - Tupels
 - Mengen
 - Dictionaries

Python Strings

- **Strings** sind eine Zeichen-Sequenz fester Länge
- Genau genommen **kein** komplexer Typ
- String-Literale werden in einfache (') oder doppelte (") Anführungszeichen gesetzt⁴
- Ausnahmezeichen sind die gleichen wie in C-Literalen
 - Beginnt ein Literal mit einem `r` oder `R`, so werden Ausnahmezeichen nicht interpretiert ⇒ **rohe Strings** (*raw strings*)
- Strings sind **unveränderlich** ⇒ String-Operationen erzeugen also neue Strings

(Auch wenn man es als Programmierer nicht unbedingt merkt)

⁴Es gibt für Spezialzwecke noch dreifache Quotes.

Python Listen

- Listen sind das, was den C-Arrays am nächsten kommt ➡ eine geordnete Sammlung anderer „Objekte“
 - **Unterschied:** Elemente in der Liste können **unterschiedliche** Typen besitzen
- Die Elemente einer Liste werden in eckige Klammern geschrieben und durch Kommata getrennt
- **Indizierung:**
 - Klammeroperator wie in C, beginnend mit 0
 - Negative Indizes zählen von rechts (beginnend mit -1)

```

1 L=[] # empty list
2 L=[1,2,3,4] # list of numbers
3 L=['John','Paul','Georg','Ringo'] # list of string
4 L=[42,'a',3.14] # list of different types

```

- Listen sind veränderlich, wie der folgende Code demonstriert

```

1 # list1.py -- list examples
2
3 L=['John','Paul','Georg','Pete'] # list of string
4 print("list:",L)
5 print("first element:",L[0])
6 print("next to last element:",L[-2])
7 L[-1]='Ringo'
8 print("list:",L)
9 L.append('Brian')
10 print("list:",L)

```

```

list: ['John', 'Paul', 'Georg', 'Pete']
first element: John
next to last element: Georg
list: ['John', 'Paul', 'Georg', 'Ringo']
list: ['John', 'Paul', 'Georg', 'Ringo', 'Brian']

```

Python Tupels

- **Tupels** sind sehr ähnlich zu Listen
 - Unterschied: Tupels sind **unveränderlich**
- **Schreibweise:** Statt eckigen werden runde Klammern verwendet

- Sonderfall: Einertuples werden mit abschließenden Komma geschrieben, um sie von „normalen“ Klammerausdrücken unterscheiden zu können

```

1 T1=()           # empty tuple
2 T2=(0,)         # one element
3 T3=(1,'a',[42,'z']) # mixed types

```

- Da Tupels unveränderlich sind, gibt es keine `append()` Funktion
- Allerdings können Tupels verkettet werden ➡ ein neuer Tupel entsteht

```

1 # tuple2.py -- concatenation
2 T=(1,2)
3 print("tuple:",T," with id:",id(T))
4 T=T+(3,4)
5 print("tuple:",T," with id:",id(T))

```

```

tuple: (1, 2) with id: 4299594352
tuple: (1, 2, 3, 4) with id: 4299405632

```

Mengen in Python

- Seit Python 2.6 gibt es **Mengentypen**
- Zwei Ausprägungen:
 - `set` ➡ veränderlich
 - `frozenset` ➡ unveränderlich, darf auch nur unveränderliche Elemente besitzen
- **Schreibweise:** Nach dem Wort⁵ `set` oder `frozenset` folgt im Klammern eine Sequenz/-Menge
- Jedes Element wird der Menge nur **einmal** eingefügt
- Es wird keine Ordnung bewahrt

```

1 # set.py -- set types
2 s1 = set([1, 2, -9, 0, 10, 2])
3 s2 = frozenset("Python")
4 print("s1=",s1)
5 print("s2=",s2)

```

```
s1= set([0, 1, 2, 10, -9])
s2= frozenset(['h', 'o', 'n', 'P', 't', 'y'])
```

- Für Mengen sind verschiedene aus der Mengenlehre bekannte Operationen definiert, u.a.:
 - `len(s)`: gibt Mächtigkeit der Menge `s`
 - `s1 | s2`: Vereinigungsmenge von `s1` und `s2`
 - `s1 & s2`: Schnittmenge von `s1` und `s2`
 - `s1 - s2`: Differenzmenge von `s1` und `s2`
 - `s1 ^ s2`: Symmetrische Differenzmenge von `s1` und `s2`

```
1 s1=set("Python")
2 s2=set("Monty")
3 print("s1|s2=",s1|s2)
4 print("s1&s2=",s1&s2)
5 print("s1-s2=",s1-s2)
6 print("s1^s2=",s1^s2)
```

```
s1|s2= set(['h', 'M', 'o', 'n', 'P', 't', 'y'])
s1&s2= set(['y', 't', 'o', 'n'])
s1-s2= set(['h', 'P'])
s1^s2= set(['h', 'M', 'P'])
```

Python Dictionaries

- **Dictionaries** sind (ungeordnete) Mengen von Zweier-Tupeln⁶.
 - Ungeordnet ➡ nicht über Index zugreifbar
- Das erste Element dient als **Schlüssel** (*key*), über die ein Tupel auffindbar ist
- Dictionaries sind **veränderlich**
- Die Nutzung ist aus folgendem Beispiel ersichtlich

```
1 # dictionary.py -- demonstrate dictionaries
2
3 mlength = {'jan': 31, 'feb': 28,
4           'mar': 31, 'apr': 30,
5           'may': 31, 'jun': 30,
6           'jul': 31, 'aug': 31,
7           'sep': 30, 'oct': 31,
8           'nov': 30, 'dec': 31}
9
10 print("March has", mlength['mar'], "days")
```

```

11 print("February has", mlength['feb'], "days")
12
13 mlength['feb'] = 29 # leap year
14 print("February has", mlength['feb'], "days")

```

6.6 Nochmals Speicherklassen und -attribute

- Nachdem wir Typen und Speicher in C bereits betrachtet haben, können wir nochmals die Speicherklassen und -attribute diskutieren, die zwischen Typen und Speicher Beziehungen herstellen
- Schon kurz betrachtet: **Speicherklassen** „static“ und „extern“
- static:
 - Macht bei Variablen die Lebensdauer unabhängig von Stand der Abarbeitung
 - Schränkt bei Funktionen die Sichtbarkeit nach „außen“ ein
- extern:
 - Nur Deklaration, verlangt zusätzliche Definition, evtl. „außerhalb“
 - ➔ „**Außerhalb**“ heißt Bibliothek oder Modul
- Die Wirkung von static und extern bei Funktionen wird bei der Modularisierung (Kapitel 14) genauer betrachtet
- Speicherklasse auto
 - Variable wird durch den Compiler automatisch angelegt und ggf. (nach Ende der Lebensdauer) auch wieder gelöscht
 - **Standardeinstellung** wenn keine andere Speicherklasse angegeben wird
 - auto dürfte damit das am seltensten gebrauchte Schlüsselwort sein
- Speicherklasse register:
 - Compiler soll den Zugriff auf Variable (möglichst) beschleunigen
 - Nur „Hinweis“, Compiler entscheidet
 - Bei CPUs mit Universalregistern wird Variable in einem Register gehalten ➔ daher der Name
 - Programmierer darf von register-Variable **keine** Adresse ermitteln
 - Einsatz bei **zeitkritischen** Funktionen, z.B. Treibern

```

1 /* register.c -- program does not compile */
2 #include<stdio.h>
3

```

```

4 int main()
5 {
6     register int fast=42;
7     int *pf;
8
9     pf = &fast;
10    printf("fast= %d\n", fast);
11    return 0;
12 }

```

```

cc -std=c99 -pedantic -Wall register.c -o register
register.c: In function 'main':
register.c:9: error: address of register variable 'fast' requested
make: *** [register] Error 1

```

- Speicherattribut „const“: Wert der Variablen soll nicht verändert werden ➔ Variable bleibt **konstant**
- Ermöglicht Compiler-Optimierung
- Compiler **unterbindet Zuweisung** an const-Variable, außer bei Initialisierung
- Eine konstante Variable wird jedoch sonst wie jede andere Variable behandelt
 - Beispielsweise ist arr im folgendem Code

```

1 const int foo = 42;
2 int arr[foo];

```

ein Array variabler Länge und damit nicht in ANSI-C erlaubt



Achtung!

Insbesondere bei Zeigern ist die Deklaration z.T. verwirrend. Man beachte den Unterschied:

const int * p; ➔ Deklariert einen **variablen** Zeiger auf eine **konstante** Ganzzahlvariable

int * const p; ➔ Deklariert einen **konstanten** Zeiger auf eine **variable** Ganzzahlvariable

- Das Speicherattribut „volatile“ ist (ein Stück weit) das Gegenteil von „const“
- „volatile“ ➔ (*engl.*) flüchtig, sprunghaft
- **Hinweis** an Compiler: Wert kann sich seit dem letzten Schreiben **verändert** haben

- Nur bei **nebenläufigen** Programmen relevant

Hinweis

Eine Deklaration wie

```
extern const volatile int rt_clock;
```

ist durchaus sinnvoll. Der Wert wird von „außen“ (z.B. durch einen Treiber) gesetzt und kann vom aktuellen Programm nur gelesen, aber nicht geändert werden.



- Speicherattribut „restrict“ ist neu in C99
- Nur bei Zeigervariablen (und daraus abgeleitet bei Arrays)
- Zusicherung an den Compiler, dass das Speicherobjekt, auf das der Zeiger zeigt, nur (direkt oder indirekt) über diesen Zeiger zugegriffen wird
- Konkrete Definition ist etwas umfangreicher ➡ Interessierte lesen bitte §6.7.3.1 der ISO/IEC 9899
- Dient zur Programmoptimierung durch den Compiler

Knobelaufgaben

Aufgabe 6.1

Drei Mädchen streiten.

Anne sagt: „Bettina sagt die Wahrheit.“ Bettina sagt: „Claudia lügt.“ Claudia sagt: „Anne und Bettina sagen beide die Wahrheit oder lügen beide.“

Wer lügt, und wer sagt die Wahrheit?

Aufgabe 6.2

In einem Land leben drei Männer, die den Ruf haben, die Weisesten aller Menschen zu sein.

Der König möchte sie testen und ruft sie an seinen Hof. Er lässt sie hintereinander sitzen, so dass sie immer nur jeweils den/diejenigen vor ihnen Sitzenden sehen können, und spricht zu ihnen: „Ich habe hier rote und weiße Hüte, und setzt jedem von Euch einen auf, wobei mindestens einer einen weißen Hut bekommt.“ Dann frage er die Weisen einzeln, beginnend mit dem, der zwei andere vor sich hat, ob sie wüssten, welchen Hut sie aufhaben.

Die Antworten sind „Nein“, „Nein“ und „Ja.“

Welchen Hut hat der dritte Weise auf?

Kapitel 7

Iterationen

*Anything that happens, happens.
Anything that, in happening, causes
something else to happen, causes
something else to happen.
Anything that, in happening, causes itself to
happen again, happens again.
It doesn't necessarily do it in chronological
order, though.*

— Douglas N. Adams, *Mostly Harmless*

*If God himself had not willed repetition, the
world would not have come into existence.*

— Søren Kierkegaard

7.1 Einführung

Motivierendes Beispiel

„Aufgabe“

Esse alle Pralinen in einer Pralinenschachtel! 🍫

Annahme: Ich weiß, wie ich *eine* Praline essen kann.



Variante 1

Iss die erste Praline von links in der ersten Reihe.
 Iss die zweite Praline von links in der ersten Reihe.
 ⋮
 Iss die Praline ganz rechts in der ersten Reihe.
 Iss die erste Praline von links in der zweiten Reihe.
 ⋮
 Iss die erste Praline von links in der letzten Reihe.
 Iss die zweite Praline von links in der letzten Reihe.
 ⋮
 Iss die Praline ganz rechts in der letzten Reihe.

- **Problem:** Algorithmus¹ ist evtl. sehr lang

Variante 2

- Weise den Pralinen die Nummern 1 bis n zu
- Zähle von 1 bis n und iss die Praline mit der jeweiligen Nummer

- **Anmerkung:** Es findet eine **Abbildung** von der Menge der natürlichen Zahlen $\mathbb{N}$ auf die Menge der Pralinen statt

Variante 3

- Zähle die Pralinen $\Rightarrow n$
- Zähle von 1 bis n und iss bei jeder Zahl jeweils eine Praline

- **Anmerkung:** Der Vorgang des Essens wird für eine **vorher bestimme** Zahl von Wiederholungen durchgeführt

Variante 4

- Solange noch eine Praline da ist: Iss eine (beliebige) Praline

- **Anmerkung:** Der Vorgang des Essens wird für eine (zunächst) **unbestimme** Zahl von Wiederholungen durchgeführt

¹Genau genommen ist es *in der dargestellten Form* gar kein Algorithmus; vgl. Kapitel 1.

Unterschiedliche Sichtweisen

- Wiederholen...
 - ...für eine **bestimmte Anzahl** von Wiederholungen
 - ...für **Elemente** einer Menge
 - ...solange eine **Bedingung** (nicht) erfüllt ist
- Alle Varianten sind aufeinander abbildbar
- Solche Wiederholungen heißen in Programmiersprachen **Schleifen**
- Schleifen gibt es in vielen (nicht allen!) Programmiersprachen
- Allgemein gibt es bei Schleifen **immer** etwas Invariantes/Konstantes und i.d.R. etwas sich Änderndes

7.2 Schleifen in C und Python

- In C gibt es drei Arten von Schleifen:
 - `while`-Schleife
 - `do-while`-Schleife
 - `for`-Schleife
- Alle Schleifen nutzen **Bedingungen**
- Jede davon kann² die anderen simulieren
- Welche genutzt wird, ist also z.T. Geschmackssache

`while`-Schleife

- Die `while`-Schleife führt ein Statement solange aus, wie eine Bedingung wahr ist.
- Sie arbeitet wie folgt:
 1. Es wird überprüft, ob der Ausdruck in der Klammer nach dem Schlüsselwort `while` eingehalten wird
 2. Falls ja, wird **eine** Anweisung nach diesem Ausdruck ausgeführt (➡ **Schleifenkörper**) und anschließen wieder zu 1. gesprungen
 - Wenn die Schleife mehrere Anweisungen umfassen soll, müssen diese in einem Block geklammert werden
 3. Falls nein, wird mit der ersten Anweisung nach dem Schleifenkörper fortgesetzt

²...ggf unter Nutzung anderer Sprachkonstrukte...

Anweisung (Schleifenkörper)

```
while(i<10) i=i+1;
```

Bedingung

Beispiele:

```
1  /* sqr.c -- Heron's method */
2  #include<stdlib.h>
3  #include<math.h>
4  #include<stdio.h>
5
6  int main(int argc, char** args)
7  {
8      double x=1.0;
9      double y=atof(args[1]);
10
11     while(fabs(y-x*x)>0.00001){
12         x=0.5*(x+y/x);
13     }
14     printf("sqrt(%f) = %f.\n",y,x);
15     return 0;
16 }
```

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int i=3;
6      while(i>0) {
7          printf("Cthulhu!\n");
8          i=i-1;
9      }
10     return 0;
11 }
```

Schleifen und Felder

- Schleifen können zur Bearbeitung oder Belegung der Elemente eines Array zur Laufzeit genutzt werden
- Beispiel:

```

1 #include<stdio.h>
2 enum {arraysize=12};    /* constant for array size */
3
4 int main()
5 {
6     int xa[arraysize],i=0;
7
8     while(i<arraysize){    /* 0,1,...,11      */
9         xa[i]=i*i;        /* square of index */
10        i=i+1;
11    }
12
13    i=0;
14    while(i<arraysize){    /* 0,1,...,11      */
15        printf("Element %d of xa: %d\n",i,xa[i]);
16        i=i+1;
17    }
18    return 0;
19 }

```

do-while-Schleife

- Die do-while-Schleife ähnelt der while-Schleife
- Unterschied: Die Bedingung wird **nach** einem Schleifendurchlauf geprüft
- Damit der Compiler weiß, dass eine Schleife beginnt, wird zu Schleifenbeginn das Schlüsselwort **do** gesetzt

```

do i=i+1; while(i<10);

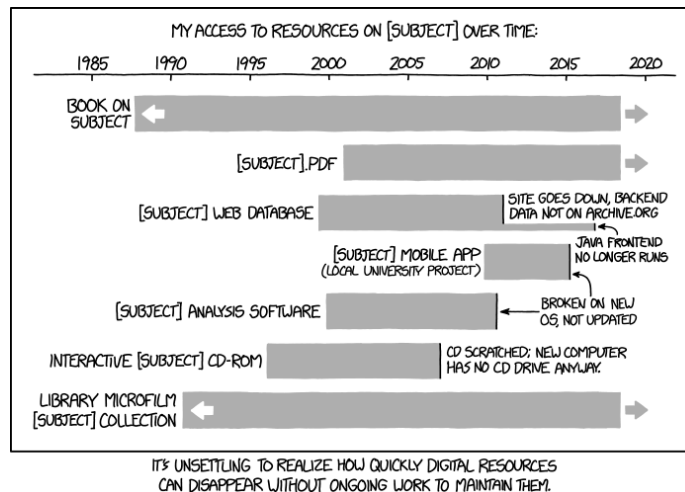
```

↑
↑
Schleifenkörper
Bedingung

- Wieder müssen bei mehr als einer Anweisung im Schleifenkörper diese in geschweifte Klammern gesetzt werden
- do-while-Schleife werden immer dann eingesetzt, wenn der Schleifenkörper **mindestens einmal** ausgeführt werden soll

for-Schleife

- Typisch:
 - Vor einer Schleife wird ein Wert initialisiert
 - Im Schleifenkörper wird ein (meist derselbe) Wert geändert



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1909/>

- Die `for`-Schleife lässt beides explizit in der Schleifendeklaration zu
- Wieder: Mehr als ein Statement im Schleifenkörper muss geklammert werden
- **Achtung:** Der Schleifenkörper kann leer sein ➔ einzelnes Semikolon
 - Schleifenkörper im Schleifenkopf integriert
 - Schlechter Stil, außer bei sehr einfachen Schleifen

Initialisierung

```
for(int i=0; i<10; i=i+1) printf("i=%d \n", i);
```

Bedingung nach Schleifenkörper

Schleifenkörper

Beispiele für `for`-Schleife

```
1 /* for.c -- loop with for */
2 #include <stdio.h>
3
4 int main(int argc, char* argv[]){
5     for(int i=0; i<argc; i=i+1)
6         printf("%d. argument: %s\n", i+1, argv[i]);
7
8     return 0;
9 }
```

```
> ./for 23 foo Bar 42
1. argument: ./for
2. argument: 23
3. argument: foo
4. argument: Bar
5. argument: 42
```

- Es können einzelne Ausdrücke weggelassen werden...

```
1 /* for2.c -- loop with for, #2 */
2 #include<stdio.h>
3
4 int main(int argc, char* argv[]){
5     int i=0;
6     for(; i<argc; i=i+1)
7         printf("%d. argument: %s\n",
8             i+1,argv[i]);
9
10    return 0;
11 }
```

- ... oder auch alle Ausdrücke ➡ Endlosschleife

```
1 /* for-ever.c -- infinite loop */
2 #include<stdio.h>
3
4 int main(){
5
6     for(;;)
7         printf("The answer is 42.\n");
8
9     return 0; // never reached
10 }
```

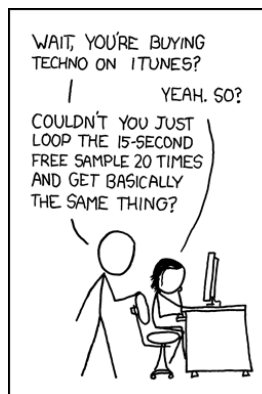
- Anmerkung: Vor C99 konnte in einer for-Schleife eine Laufvariable nur initialisiert, aber **nicht** deklariert werden
- Die Deklaration musste also vorher erfolgen
- Das obige Beispiel würde in ANSI-C nicht übersetzbar sein und müsste stattdessen so aussehen:



```

1  /* for.c -- loop with for, ANSI C */
2  #include <stdio.h>
3
4  int main(int argc, char* argv[]){
5      int i;
6      for(i=0; i<argc; i=i+1)
7          printf("%d. argument: %s\n", i+1, argv[i]);
8
9      return 0;
10 }

```



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/411/>

break **und** continue

- Schleifen werden solange durchlaufen, wie die Schleifenbedingung gegeben ist
- Allerdings gibt es zwei Möglichkeiten, **innerhalb des Schleifenkörpers** den Kontrollfluss zu ändern:

break: Beendet die Schleife unabhängig von der Schleifenbedingung

continue: Startet sofort die nächste Auswertung der Schleifenbedingung und ggf. den nächsten Schleifendurchlauf

Achtung!

Die Benutzung von `break` und `continue` ist eigentlich nicht notwendig und meist ein Indiz, dass die Programmlogik nicht hinreichend durchdacht wurde.

Daher sollten `break` und `continue` nur zur Behandlung von Notfällen (Ausnahmen) genutzt werden.

```

1  /* reciprocal.c -- calculate reciprocal value of array elements */
2  #include<stdio.h>
3
4  /* a negative value indicates end of list */
5  double f[]={1.0, .5, 3.1415, .33333, 0.0, 2.7182, 42.23, -1};
6
7  int main()
8  {
9      int i;
10     for(i=0; ; i=i+1){
11         if (f[i]<0.0) break;
12         if (f[i]<=0.0001) continue;
13         f[i]=1/f[i];
14     }
15
16     for(i=0; f[i]>=0.0; i=i+1)
17         printf("%d value: %f\n",i,f[i]);
18
19     return 0;
20 }

```

Geschachtelte Schleifen

- Schleifen können auch ineinander verschachtelt sein
- Dabei können sich die Schleifen auch beeinflussen

```

1  /* for3.c -- nested loops */
2  #include<stdio.h>
3
4  int main(int argc, char *argv[]){
5      int goal;
6
7      if (sscanf(argv[1],"%d",&goal)!=1) return -1;
8      for (int i=1; i<=goal; i=i+1)
9      {
10         for(int j=1; j<=i; j=j+1)
11             printf("%d ",i);
12         printf("\n");
13     }
14     return 0;
15 }

```

Schleifen in Python

- In Python gibt es zwei Arten von Schleifen:
 - while-Schleifen
 - for-Schleifen
- while-Schleifen ähneln ihren Pendanten in C:

```
1 i=10
2 while (i>0):
3     print("i=",i)
4     i=i-1
```

- Wie immer in Python können Blöcke durch Einrückungen markiert werden

Schleifensteuerung

- Auch Python kennt die Schlüsselwörter break und continue
- Die Semantik ist gleich zu C
- Zusätzlich gibt es bei Python-Schleifen einen optionalen else-Zweig
 - ⇒ Wird **nach der Schleife** ausgeführt, wenn diese **nicht** abgebrochen wurde

```
1
2 def prim(x):
3     y=int(x/2)
4     while y>=2:
5         if x % y == 0:
6             print(x, "has factor", y)
7             break
8         y=y-1
9     else:
10        print(x, "is prim")
```

- Mit Hilfe von break kann leicht die do-while-Schleife aus C nachempfunden werden

```
1 i=0
2 while True:
3     print("i=",i)
4     i=i+1
5     if not(i<10): break
6 print("Ende")
```


for-Schleife in Python

- Die Syntax for-Schleife sieht zunächst ähnlich wie die while-Schleife aus

```
1 for i in C:
2     do_something(i)
```

- Die Semantik ist jedoch anders:
 - C **muss** eine Sequenz oder Menge sein (d.h. String, Liste, Tupel, Menge, Dictionary)³
 - Es werden *i* nacheinander die Elemente von C zugewiesen

```
1 # for.py -- loops over sequence objects
2
3 for x in ['John', 'Paul', 'Georg', 'Pete']:
4     print(x)
```

- Dies funktioniert auch für gemischte Typen:

```
1 # for2.py -- mixed types
2
3 T=('Alpha',2,(1,2))
4 for x in T:
5     print(2*x)
```

```
python for2.py
AlphaAlpha
4
(1, 2, 1, 2)
```

- Um die Semantik der C-for-Schleife nachzuempfinden, kann die range-Funktion genutzt werden

```
range(start=0, end, step=1)
```

Von Bis
 (nicht einschliesslich) Schrittweite

³Genauer: Es muss ein *iterierbarer* Typ sein.

```
>>> range(5), range(2,5), range(2,10,2)
([0, 1, 2, 3, 4], [2, 3, 4], [2, 4, 6, 8])
>>>
```

```
1 # for3.py -- loops with range
2
3 for i in range(1,8):
4     for j in range(i):
5         print(i, end=' ')
6     print()
```

Iteratoren in C

- Umgekehrt kann man auch den Python-Ansatz in C „nachempfinden“

```
1 /* iter.c -- iterator with C */
2 #include<stdlib.h>
3 #include<stdio.h>
4
5 extern char *names[];
6
7 static int ndx=0;
8 char *first() { return names[ndx=0]; }
9 char *next(){
10     if (names[ndx]==NULL) ndx=0;
11     else ndx=ndx+1;
12     return names[ndx];
13 }
14
15 char *names[]={ "John","Paul","Georg",
16                 "Ringo", NULL};
17
18 int main(){
19     for(char *str= first(); str != NULL;
20         str=next())
21         printf("%s ",str);
22     return 0;
23 }
```

- Solche Iteratoren sind z.B. in C++ verbreitet
- ➡ Dort aber eleganter (auch in C geht es schöner 😊)

7.3 Iteration und Rekursion

- Bisher wurden in Schleifen immer (implizite) Zuweisungen benutzt ➔ Zustandsmodell
- Auch im funktionalen Modell sind Schleifen möglich
- Dort werden sie durch Rekursion realisiert
- Beispiel:

```
1 /* recursion.c -- loops by recursion */
2 #include <stdio.h>
3
4 void say_it(int i){
5     if (i==0) return;
6     else {
7         printf("Cthulhu!\n");
8         say_it(i-1);
9     }
10 }
11
12 int main(){
13     say_it(3);
14     return 0;
15 }
```

Schleifenfunktion

- Dabei kann der Schleifenkörper von der „Schleifenmechanik“ getrennt werden, so dass eine Schleifenfunktion möglich ist
- Wir nutzen hier einen **Zeiger auf Funktionen**



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1557/>

```

1  /* loop-func.c */
2  #include <stdio.h>
3
4  // Pointer to a void function with integer argument
5  typedef void (*VFP)(int);
6
7  void loop(VFP func, int start, int end,
8           unsigned int step){
9      if (start <= end) {
10         func(start);
11         loop(func, start+step, end, step);
12     }
13 }
14
15 void lbody(int param){
16     printf("%d^2=%d\n", param, param*param);
17 }
18
19 int main(){
20     loop(lbody, 0, 10, 1);
21     return 0;
22 }

```

- Dieses Beispiel hat einige Einschränkungen:
 - Der Typ der Schleifenkörper-Funktion ist festgelegt ➡ Der Schleifenkörper kann nur auf den Schleifenindex (und globale Variablen) zugreifen
 - Es muss `step > 0` gelten ➡ Die Schleife kann nur aufwärts zählen
- Man kann diese Einschränkungen überwinden, aber dies würde den Code „aufblasen“
- Durch die flexiblere Argumentensyntax kann dies in Python auch für Funktionen mit unterschiedlicher Parameteranzahl gelöst werden:

```

1  # loop-func.py -- loops by recursion
2  from sys import stdout
3  def loop(v,e,i,f,args):
4      f(*(v,)+args)
5      if v != e:
6          loop(v+i,e,i,f,args)
7
8  def printpow(i,x):
9      print(i,"**",x,"=",i**x)
10
11 def printi(i,b,a):
12     print(b+str(i)+a, end=" ")
13
14 loop(1,4,1, printpow,(2,))
15 loop(10,0,-1, printi, ("[" , "]" ))

```

```
> python loop-func.py
1 ** 2 = 1
2 ** 2 = 4
3 ** 2 = 9
4 ** 2 = 16
[10] [9] [8] [7] [6] [5] [4] [3] [2] [1] [0]
>
```

- Mit Hilfe von λ -Funktionen (namenlosen) sind noch allgemeinere und elegantere Varianten möglich, z.B. allgemeinere Abbruchbedingungen

Anmerkung:

Wenn Sie dieses Beispiel nicht verstehen, können Sie es ignorieren – λ -Funktionen werden in Kurs „Funktionale Programmierung“ behandelt

Äquivalenz

- Allgemein gilt:

Jede Schleife in einem Programm kann in eine rekursive Funktion umgewandelt werden.

und

Jede (trivial-)rekursive Funktion kann mit Hilfe von Schleifen berechnet werden.⁴

- Nicht trivialerekursiv ist z.B. die ACKERMANN-Funktion:

$$f(x, y) = \begin{cases} y + 1 & \Leftarrow x = 0 \\ f(x - 1, 1) & \Leftarrow (x \neq 0) \wedge (y = 0) \\ f(x - 1, f(x, y - 1)) & \text{sonst} \end{cases}$$

- Schleifen und Rekursion sind also für die meisten praktischen Anwendungsfälle **gleichmächtig**
- In der Regel wird eine Schleife als übersichtlicher empfunden
 - Es gibt aber auch Fälle (vgl. Kapitel 12), in denen Rekursion die handlichere Lösung bildet
- Meist kann die Korrektheit rekursiven Codes leichter überprüft werden
- Einige Programmiersprachen kennen keine Schleifen, sondern nur Rekursionen

⁴Genauer: Primitiv-rekursive Funktionen lassen sich stets mit *a priori* beschränkten Schleifen berechnen; bei sogenannten μ -rekursive Funktionen muss diese Einschränkung aufgegeben werden.

Knobelaufgaben

Aufgabe 7.1

Vor Caroline liegen 10 Karten: 

Sie soll sich in Gedanken eine der Karten aussuchen. Anschließend verrät sie Albert die Farbe der Karte, und Bernd den Wert.

Darauf hin sagt Albert: „Ich weiß nicht, welche Karte Du gewählt hast, aber ich weiß, dass es Bernd auch nicht weiß“. Darauf Bernd: „Naja, erst wußte ich es nicht, aber jetzt weiß ich es.“ Darauf Albert: „Dann weiß ich es auch.“

Welche Karte hat Caroline gewählt?

Aufgabe 7.2

In einer Schale und in einem Eimer seien eine Anzahl von weißen und orangen Tischtennisbällen, wobei soviel Bälle im Eimer sind, dass der folgende Algorithmus stets durchgeführt werden kann:

1. Nehme zwei beliebige Bälle aus der Schale.
 - Wenn sie die gleiche Farbe haben, werfe sie in den Eimer und nehme einen orangen Ball aus dem Eimer und lege ihn in die Schale.
 - Wenn sie unterschiedliche Farbe haben, lege den weißen in die Schale zurück und werfe den orangen Ball in den Eimer.
2. Wiederhole, bis nur noch ein Ball in der Schale ist.

Wird tatsächlich stets **ein** Ball übrig bleiben? Wenn ja, kann etwas über seine Farbe in Abhängigkeit zur ursprünglichen Anzahl der Bälle gesagt werden? Wenn ja, was?

Kapitel 8

Entwurf Algorithmen

*There is a pleasure in creating well-written, understandable programs.
There is a satisfaction in finding a program structure that tames the complexity of an application.
We enjoy seeing our algorithms expressed clearly and persuasively.
We also profit from our clearly written programs, for they are much more likely to be correct and maintainable than obscure ones.*

— Samuel P. Harbison

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

— John F. Woods

8.1 Idee

Warnung!

Es gibt **keinen** (exakten) Algorithmus, um Algorithmen zu entwickeln.

Abstraktionsvermögen, Kreativität und Erfahrung tragen aber dazu bei, (gute) Algorithmen zu entwickeln. Jede dieser Eigenschaften kann durch Training verbessert werden.

- Jedoch verlangt jeder Algorithmus **Problemverständnis** und (mindestens) eine **Idee**

Problemlösen

Man beachte:

Algorithmenentwurf ist Problemlösen.

Allgemeines Vorgehen beim Problemlösen:

1. Verstehen Sie das Problem!
2. Was ist das Ziel? Was ist gegeben?
3. Modellieren Sie das Problem!
4. Lösen Sie das Modellproblem!
5. Interpretieren Sie die Lösung!

Problemverstehen

Merke

Ich habe ein Problem **dann und erst dann** verstanden, wenn ich es anderen erklären kann!

• Interessante Fragen:

- Woraus genau besteht die Eingabe?
- Was genau ist das gewünschte Ergebnis oder die gewünschte Ausgabe?
- Kann ich ein Minimalbeispiel angeben, das von Hand zu lösen ist? Was passiert, wenn ich dies mache?
- Brauche ich eine optimale, eine nahezu optimale oder nur irgendeine Lösung? Was heißt in diesem Problem „optimal“?
- Wie groß ist eine typische Problemistanz? 10 Objekte? 1000 Objekte? 1000000 Objekte?
- Wie wesentlich ist Geschwindigkeit in meinem Problem? Muss es in Sekunden gelöst sein? Eine Minute? Eine Stunde? Ein Tag?
- Wie kann das Problem in einer Grafik beschrieben werden?
- Welche Gemeinsamkeiten hat das Problem mit anderen bekannten Problemen?
- Was unterscheidet dieses Problem von anderen ähnlichen Problemen?

Suchen der Idee

- Niemand kann Ihnen abnehmen, die Lösungsidee zu finden
- Jedoch gibt es mehrere typische Ansätze:
 - **Invarianten** ⇒ Herausfinden, was sich **nicht** ändert
 - **Induktion** ⇒ Übertragung vom (hoffentlich einfachen) Einzelfall auf alle Fälle
 - **Abstraktion und Nachnutzung** ⇒ Das Problem so **modellieren**, dass es mit anderen (bekannten) Problemen übereinstimmt
 - **Suchen** mit „brutaler Gewalt“ (*brute force*) ⇒ Alle infrage kommenden Fälle untersuchen, ob sie die Lösung bieten

- **Intelligentes Suchen** ➔ Wie „brute force“, aber vorher den Suchraum einschränken

Modellierung

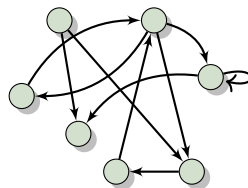
- Durch Modellierung kann ein Problem auf das Wesentliche beschränkt werden
- Als Modellierungsansätze sind vor allem grundlegende Konzepte aus der Mathematik geeignet:
 - **Mengen** (ungeordnete Elemente) und **Multimengen** (mehrfaches Vorkommen erlaubt)
 - **Permutationen** (geordnete Elemente)
 - **Hierarchien/Bäume**
 - **Graphen**
 - **Punkte** (Koordinaten in geometrischen Räumen)
 - **Polygone** (Regionen in geometrischen Räumen)
 - **Zeichenketten**

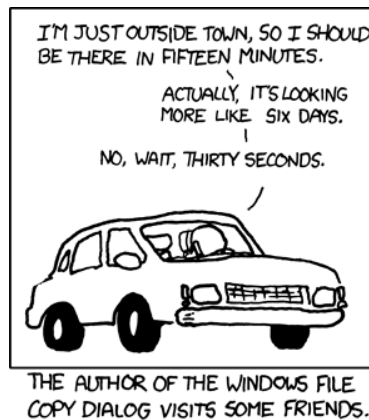
Exkurs: Graphen

Da Graphen eine wesentliche Methode zur Modellierung sind, wird hier eine (sehr) kurze Einführung in die Graphentheorie eingeschoben.

Falls Sie bereits mit Graphen vertraut sind, können Sie diese überspringen.

- **Graphen** sind ein häufig benutztes Mittel zur Modellierung von Sachverhalten
- Wir haben sie bereits ständig gebraucht, ohne sie zu benennen
- Vorteile von Graphen: gute grafische Representation
 - Knoten = Kreise, Punkte (engl.: *nodes*, *vertices*)
 - Kanten = Linien, Pfeile (engl.: *edges*, *arcs*)
- Ein ganzer Bereich der Mathematik beschäftigt sich mit Graphen ➔ **Graphentheorie**





Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/612/>

Definition 8.1: Graph

Ein **Graph** ist eine Menge von **Knoten** $V = \{v_1, v_2, \dots, v_n\}$, die durch eine (Multi-)Menge von **Kanten** $\{e_1, e_2, \dots, e_m\}$ in Beziehung stehen können. Kanten können **ungerichtet** oder **gerichtet** sein. Ungerichtete Kanten werden als Knotenmengen $\{v_i, v_j\}$ aufgefasst, gerichtete Kanten als geordnete Knotenpaare (Tupel) (v_i, v_j) .¹ Gerichtete Kanten werden in der Regel durch Pfeilspitzen repräsentiert.

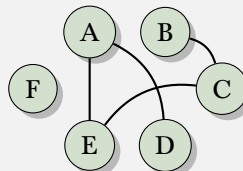
Kanten können Werte (Gewichte) zugewiesen werden, $w : E \rightarrow \mathbb{R}$. In diesem Fall spricht man von einem **gewichteten Graph**.

Beispiel 8.1

Anton, Bernd, Claudia, Dora, Ernst und Franziska seien bei dem Social-Web-Dienst „FaceVZ“ angemeldet. Es sind jeweils miteinander „befreundet“²:

- Anton und Dora
- Bernd und Claudia
- Anton und Ernst
- Claudia und Ernst

Dieser Sachverhalt lässt sich mit folgenden (ungerichteten) Graphen modellieren:



¹Häufig wird auch bei ungerichteten Graphen die Tupel-Schreibweise genutzt und $(v_i, v_j) \equiv (v_j, v_i)$ vorausgesetzt.

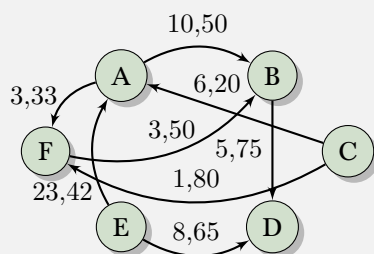
Beispiel 8.2

Anna, Brigitte, Charlotte, Denise, Eva und Friederike gehen ab und zu gemeinsam in eine Kneipe. Wenn, was häufiger vorkommt, die eine der Damen ihr Portemonnaie vergessen hat, zahlt eine andere für sie.

Am Ende des Jahres ergibt sich folgendes Schuldverhältnis:

- Anna schuldet Brigitte 10,50€ und Friederike 3,33€
- Brigitte schuldet Denise 5,75€
- Claudia schuldet Anna 6,20€ und Friederike 1,80€
- Eva schuldet Anna 23,42€ und Denise 8,65€
- Friederike schuldet Brigitte 3,50€

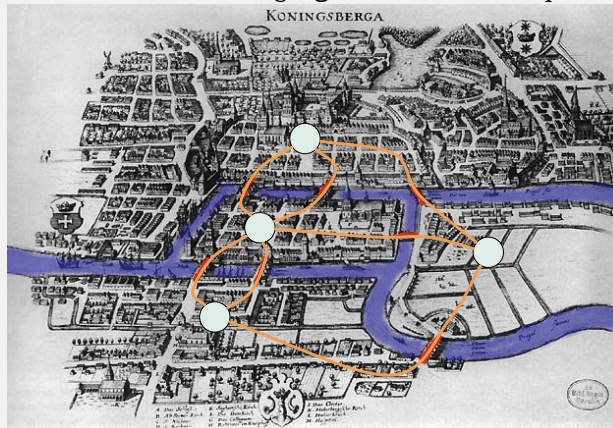
Die ist darstellbar in einem gerichteten und gewichteten Graphen:



- Durch die Reduzierung auf das (Graphen-)modell abstrahiert sich die Problemstellung

Beispiel 8.3: Königsberger Brückenproblem

Problem aus dem frühen 18. Jahrhundert: Gibt es in Königsberg einen Weg, bei dem man alle sieben Brücken über den Pregel **genau einmal** überquert?



Leonhard Euler löste dieses Problem 1736 mit Hilfe der Graphentheorie.

- Das **Modell** des Graphen ist von der **Datenstruktur** zu unterscheiden

²Was immer das auch bedeuten mag. 😊

- Beispielsweise kennt C keinen Datentyp für Graphen ➔ dieser muss ggf. selbst definiert werden

Achtung!

Nur weil ein Problem mit einem bestimmten Ansatz **modelliert** wurde, bedeutet dies **nicht** unbedingt, dass auch eine entsprechende Datenstruktur für diesen Ansatz gebraucht wird.

Falls tatsächlich eine solche Datenstruktur gebraucht wird, gibt es verschiedene Möglichkeiten, z.B.:

- **Adjazenzmatrix:** zweidimensionales Array, bei dem jedes Element $a_{i,j}$ beschreibt, ob vom Knoten v_i zum Knoten v_j eine Kante vorhanden ist oder welches Gewicht diese hat

```
1 enum { NumNodes=5};
2 bool a[NumNodes][NumNodes];
3 :
4 a[0][1]= true;
5 // = edge (v0,v1) exists
```

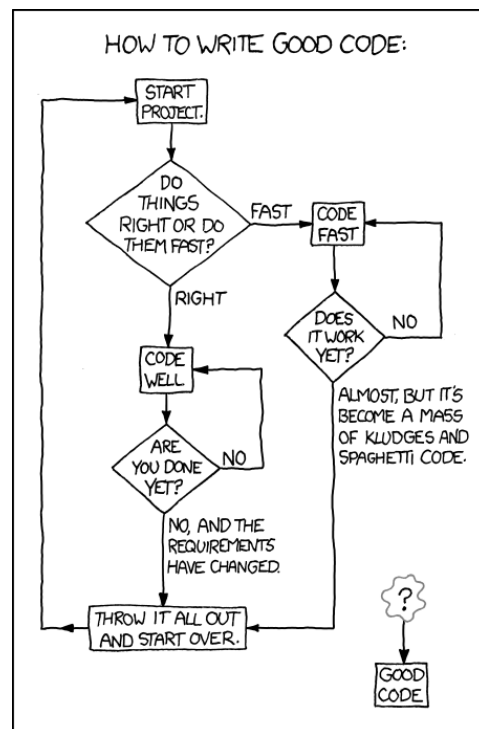
- Eignet sich besonders für gerichtete Graphen
- **Verbundstypen und Zeiger:** Knoten können durch `struct`-Typen und Kanten durch darin enthaltene Zeiger modelliert werden

```
1 enum{ MaxDegree = 6};
2 struct node;
3 typedef struct node{
4     int id;
5     struct node *in[MaxDegree];
6     struct node *out[MaxDegree];
7 } node_t;
```

- Eignet sich besonders für Graphen mit variabler Knotenanzahl

Beliebige andere Ansätze sind denkbar...

- Da die Graphentheorie ein ziemlich altes Gebiet der Mathematik ist, gibt es viele Sätze und Standardalgorithmen, die man direkt für die Lösung eines vorliegenden Problems nutzen kann
- Beispiele:
 - Satz über die Existenz von Euler-Wegen und -Kreisen
 - Kürzeste Wege in einem gewichteten Graphen



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/844/>

- Maximaler Fluss zwischen zwei Knoten in einen gewichteten Graph
- Effektivität von Suchen in Bäumen
- Eine kompakte Darstellung von Graphenalgorithmen bietet z.B. [Bra94]

Von der Idee zum Algorithmus

- Hat man eine Idee gefunden, muss der Algorithmus entwickelt werden
- Vorgehen der **schrittweisen Verfeinerung** (**Top-Down-Methode**)
 1. Beschreibe die Grobidee zur Lösung des Problems
 2. Zerlege das Problem in Unterprobleme
 3. Verfahre mit jedem Unterproblem entsprechend
- Wann beenden? ➔ Wenn **ausführbare Einzelschritte** gefunden sind
- Fähigkeiten des ausführenden Systems müssen bekannt sein

Korrektheit

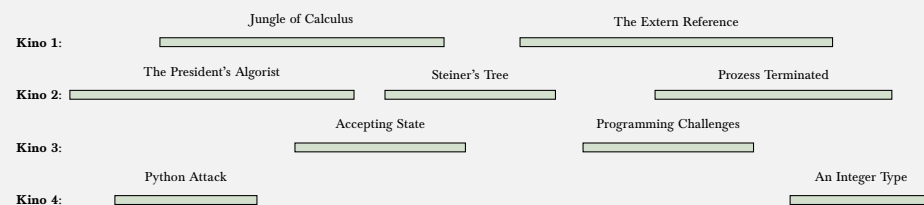
- Wenn ein Ansatz gefunden wurde, muss seine **Korrektheit** geprüft werden
- Zunächst sollte die Korrektheit der **Idee** geprüft werden, dann die anschließend die des konkreten Algorithmus
- Die Korrektheit einer Idee kann nicht formal geprüft werden
- Folgendes sollte überdacht werden:
 - Funktioniert die Idee allgemein oder nur in einem Spezialfall?
 - Gibt es Spezialfälle, in denen die Idee nicht funktioniert? Sind diese Fälle relevant?
 - Versuche, ein Gegenbeispiel zu finden
 - Extremfälle helfen
- Formalere Ansätze zur Nachweis der Programmkorrektheit werden im Kapitel ?? diskutiert

Beispiel

Beispiel 8.4

Sie sind Fan oder sogar Juror bei einem Filmfestival. Sie wollen soviel Filme wie möglich anschauen, die aber leider z.T. in verschiedenen Kinos laufen und sich zeitlich überschneiden.

Wie sollte Ihr allgemeines Vorgehen aussehen, damit Sie möglichst viele Filme sehen?



Probleme dieses Typs nennt man *Schedulingprobleme* (Planungsprobleme). Sie sind ein häufig auftretender Problemtyp.

- **Modellierung:**
 - **Gegeben** ist eine Menge I von Intervallen
 - **Gesucht** ist die größte Untermenge nichtüberschneidender Intervalle
- ~~1. Idee: EARLIEST/JOBS~~
 - Wähle den Film, der als erster anfängt
 - Streiche alle überlappenden Filme
 - Starte von vorn bis kein Film übrig

- ~~2. Idee: *SHORTESTJOBS*~~
 - Wähle den kürzesten Film
 - Streiche alle überlappenden Filme
 - Starte von vorn bis kein Film übrig
- 3. Idee: *EXHAUSTIVESHARCH* ⇒ funktioniert
 - Konstruiere alle Mengen von Filmen
 - Werfe alle Mengen weg, in denen sich mindestens zwei Filme überlappen
 - Wähle die Menge(n) mit den meisten Filmen aus

Bei n Filmen gibt es 2^n verschiedene Mengen, die betrachtet werden müssen.
- 4. Idee: *EARLIESTCOMPLETION* ⇒ funktioniert effizient
 - Wähle den Film, der zuerst beendet ist
 - Streiche alle überlappenden Filme
 - Starte von vorn bis kein Film übrig
- Betrachten weiteres Beispiel

Beispiel 8.5: Traveling Salesman

Ein Vertreter (Handlungsreisender) muss in verschiedene Städte fahren und anschließend nach Hause zurückkehren.

Wie muss er seine Route planen, damit die zurückzulegende Gesamtstrecke möglichst kurz ist?

- **Modellierung:**
 - Nutzen gewichteten Graphen
 - Knoten sind Städte
 - Kanten zwischen zwei Knoten (Städten) haben Entfernung als Kantengewicht
- ~~1. Idee: *NEARESTNEIGHBOR*~~
 - Wähle als erstes Ziel die nächstliegende Stadt aus
 - Wiederhole solange, bis keine unbesuchte Stadt übrig ist
 - Kehre nach Haus zurück
- ~~2. Idee: *CLOSESTPAIR*~~
 - Wähle die beiden Städte aus, die am nächsten sind
 - Streiche alle Städte, die bereits zwei Verbindungen haben
 - Wiederhole, bis alle Städte verbunden sind

- **3. Idee: *EXHAUSTIVSEARCH* ➔ funktioniert** (theoretisch)
 - Konstruiere alle möglichen Wege
 - Wähle den kürzesten aus
- Ein auf dieser Idee basierender Algorithmus ist **extrem** langsam
 - Er muss bei n Städten $\frac{n-1}{2}!$ Wege vergleichen
 - Bei $n = 30$ sind dies 4.420.880.996.869.850.977.271.808.000.000 ($\approx 4,4 \cdot 10^{30}$) verschiedene Wege
- Der derzeit³ schnellste Computer der Welt „Fugaku“ (RIKEN Center for Computational Science, Kōbe, Japan) würde für ein Problem mit 30 Städten mehrere Millionen(!) Jahre arbeiten!
- Ansatz funktioniert nur bei **sehr** kleinen Problemen

Merke

Für dieses Problem gibt es keine effektive⁴ Lösung.

Man begnügt sich daher in der Praxis mit Näherungslösungen. Ein solcher Algorithmus, der keine korrekte Lösung garantiert, heißt **Heuristik**.

8.2 Spezifikation

- Ein Algorithmus muss irgendwie beschrieben werden
- Eine Variante ist eine konkrete Implementation in einer konkreten Programmiersprache
- **Problem:** Eine Programmiersprache kennt nicht alle möglichen Konzepte

Beispiel 8.6

C kennt keine Datentypen und Operationen für mathematische Mengen. Mengenoperationen würden in C kompliziert aussehen und daher evtl. die **Idee** eines Algorithmus verschleiern.

- **Abhilfe:** Beschreibung des Algorithmus in natürlicher Sprache
- **Problem:** Mangel an Exaktheit
- Deshalb werden Algorithmen (wenn die algorithmische Idee betont werden soll) in **Spezifikationssprachen** beschrieben
- Es gibt sehr viele Spezifikationssprachen

⁴...aber effektivere 🙄

- Viele dienen nur dazu, die **Anforderungen** an den Algorithmus zu beschreiben ➔ für uns (hier) nicht interessant
- Es gibt **formale** und halbformale Spezifikationssprachen
 - Formale: **Automatisch** verarbeitbar, z.B. in Theorem Beweisern ➔ häufig schwer zu lesen⁵
 - Halbformale: Dient nur zur **Kommunikation** über Algorithmen
- Wir nutzen die wahrscheinlich gängigste halbformale Spezifikationssprache für imperative Algorithmen: **Pseudocode**

Pseudocode

- Jeder von Ihnen hat schon Pseudocode gesehen⁶
- Ähnlich PASCAL
- Legt einen Satz von Anweisungen fest
- Lässt beliebige mathematische Terme zu
- Lässt Sätze in natürlicher Sprache zu
 - Sollte möglichst wenig gebraucht werden
 - Müssen mit Verb beginnen und keine Nebensätze haben

Hinweis

Da Pseudocode niemals ausgeführt wird, gibt es eine Vielzahl verschiedener Formen.

In der Regel ist Pseudocode aber stets intuitiv lesbar, wenn man jemals mit einer imperativen Programmiersprache zu tun gehabt hat.

Pseudocode: Syntax und Layout

Anweisungen in Pseudocode:

- **Verzweigung**

```
if condition then
  statement(s)
end if
```

oder

⁵...für Menschen 😊

⁶Oder er/sie hat noch nicht hinreichend Selbsttestaufgaben bearbeitet.

```

if condition then
  statement(s)
else
  statement(s)
end if

```

- **While-Schleife**

```

while condition do
  statement(s)
end while

```

- **Repeat-Until-Schleife**⁷

```

repeat
  statement(s)
until condition

```

- **For-Schleife**

```

for expression do
  statement(s)
end for

```

- **Eingabe/Vorbedingung**

```

Require: requirement(s)

```

- **Ausgabe/Nachbedingung**

```

Ensure: postcondition(s)

```

- **Kommentare**

```

[statement] ▷ comment

```

oder

⁷...entspricht `do ...while` in C.

```
[statement]      { comment }
```

- **Unteralgorithmen**

```
procedure NAME(parameter)
    statement(s)
    [return expression]
end procedure
```

- **Aufruf**

```
NAME(parameter)
```

oder

```
call NAME(parameter)
```

Beispiel

- Beispiel „Euklidischer Algorithmus“

```
Require:  $A, B \in \mathbb{N}, A > 0 \wedge B > 0$ 
Ensure:  $a = b = \text{gcd}(A, B)$  ▷ gcd: greatest common divisor

 $a \leftarrow A; b \leftarrow B$ 
while  $a \neq b$  do
    if  $a < b$  then
         $b \leftarrow b - a$ 
    else
         $a \leftarrow a - b$ 
    end if
end while
```

- Künftig werden wir Algorithmen häufig in Pseudocode angeben, wenn wir von konkreten Datenstrukturen abstrahieren können

8.3 Beispiel: Ferienplanung

- Wir wollen den Algorithmenentwurf noch einmal im Zusammenhang von der Problemstellung bis zur Implementierung in einem kleinen Beispiel betrachten:

Ferienplanung

Die Studierenden eines Jahrgangs des Bachelor-Studiengangs „Crazy Computation“ wollen ein paar Jahre nach ihrem Abschluss zur Feier der gemeinsamen Erlebnisse einen gemeinsamen Urlaub machen.

Da die Kommilitoninnen und Kommilitonen inzwischen in unterschiedlichen Arbeits- und Familienverhältnissen und weit verstreut leben, wird schnell klar, dass kein Termin zu finden ist, an den alle können. Einige könnten mehrmals im Jahr, für einige kommt nur ein Termin in Frage. Auch die Dauer, für die einzelnen Kommilitoninnen und Kommilitonen verreisen können, ist sehr unterschiedlich.

Man einigt sich darauf, den (bzw. einen) Zeitraum zu finden, zu dem am meisten Kommilitoninnen und Kommilitonen die ganze Zeit mitmachen können.

Entwerfen Sie ein Programm, der einen solchen Zeitraum ermittelt!

- Auf den ersten Blick ist das Problem leicht und offensichtlich und ggf. durch „scharfes Draufsehen“ lösbar
- Allerdings braucht „scharfes Draufsehen“ Intuition und eine Gesamtansicht $\Rightarrow$ hat der Computer nicht
- **Modellierung:**
 - **Gegeben** ist eine Menge I von Intervallen⁸
 - **Gesucht** ist eine größte Untermenge von überschneidenden Intervallen
- Problem klingt ähnlich wie das Kino-Problem
- Offensichtlich würde hier auch der *EXHAUSTIVESEARCH*-Ansatz funktionieren

ExhaustiveSearch

Require: $I = \{I_1, I_2, \dots\}, |I| > 0$

Ensure: R ist nichtleerer Schnitt möglichst vieler Intervalle

```

P ← P(I); B ← ∅
for S ∈ P do
  C = ⋂Ii ∈ S Ii
  if (C ≠ ∅) ∧ (|S| > |B|) then
    B ← S
    R ← C
  end if
end for

```

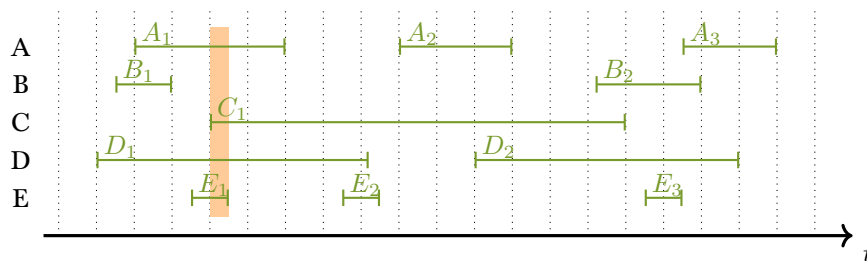
⁸Es wird angenommen, dass die Intervalle einer Person nicht überschneidend sind

- **Diskussion:**

- Ineffizient!
 - * Schleife wird $2^{|I|}$ mal durchlaufen
 - * $\mathcal{P}(I)$ und $\bigcap_{I_i \in S} I_i$ brauchen selbst bei der Implementation selbst Schleifen
 - * Verschachtelte Schleifen beeinträchtigen die Performance (► Kapitel 10)
- Implementation in C nicht offensichtlich
 - * Keine Intervall-Datentypen (und damit -Operationen)
 - * Keine Mengen-Datentypen (und damit -Operationen)

► Versuchen durch Modellbetrachtungen „besseren“ Algorithmus zu finden

Graphische Modellierung



- Lösung liegt im betrachteten Beispiel offensichtlich im mit markiertem Bereich
- **Beobachtungen**
 - Intervalle der gleichen Person (z.B. A_1 und A_2) können nicht gemeinsam Teil einer Lösung sein
 - Andere Paare von Intervallen können nur gemeinsam zu einer Lösung beitragen, wenn jeweils der Anfang des einen vor dem Ende des anderen Intervalls liegt
- Wie würde der Mensch vorgehen?
 - Wahrscheinlich entlang der Zeit (d.h. von links nach rechts) durchgehen und Intervall zählen
 - Änderung immer nur bei einem Intervallanfang oder -Ende

↑
Zeitpunkte, gut mit int darstellbar

Zählen der Intervalle

Require: $I = \{I_1, I_2, \dots\}, |I| > 0$

Ensure: R ist nichtleerer Schnitt möglichst vieler Intervalle

```

 $L \leftarrow []$  ▷ leere Liste
for  $(t_s, t_e) \in I$  do
    Füge Tupel  $(t_s, -1)$  zu  $L$ 
    Füge Tupel  $(t_e, 1)$  zu  $L$ 
▷ erster Tupelwert Zeit, zweiter Typ
end for
 $c \leftarrow 0$  ▷ # offener Intervalle
 $b \leftarrow 0$  ▷ bisher bester Wert
Sortiere  $L$  nach Zeitwert
for  $t_i \in L$  do
     $c \leftarrow c - \text{Typwert}(t_i)$ 
    if  $c > b$  then
         $b \leftarrow c$ 
         $R \leftarrow (\text{Zeitwert}(t_i), \text{Zeitwert}(t_{i+1}))$ 
    end if
end for

```

- Dies entspricht (etwas abgewandelt) dem MARZULLO-Algorithmus, der im Network Time Protocol (ntp) verwendet wird

Implementierung: Datentypen

- Dieser Pseudocode kann nun relativ einfach in C-Code umgesetzt werden
- Der Pseudocode enthält Datentypen, die es in C nicht gibt, diese können aber leicht nachgebaut werden
- Intervalle sind nur als Eingabe relevant und werden in Tupels gewandelt $\Rightarrow$ keine Intervall-Operationen notwendig
- Bei den Tupels sind die einzelnen (Integer-)Elemente relevant $\Rightarrow$ ebenfalls keine Tupel-Operatoren notwendig

$$\text{VOLUME}(R) = (4/\text{INT}(\text{PI})) * \text{PI} * R^{\text{INT}(\text{PI})}$$

PROGRAMMING TIP: THE NUMBER "3" IS CURSED. AVOID IT.

Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1275/>

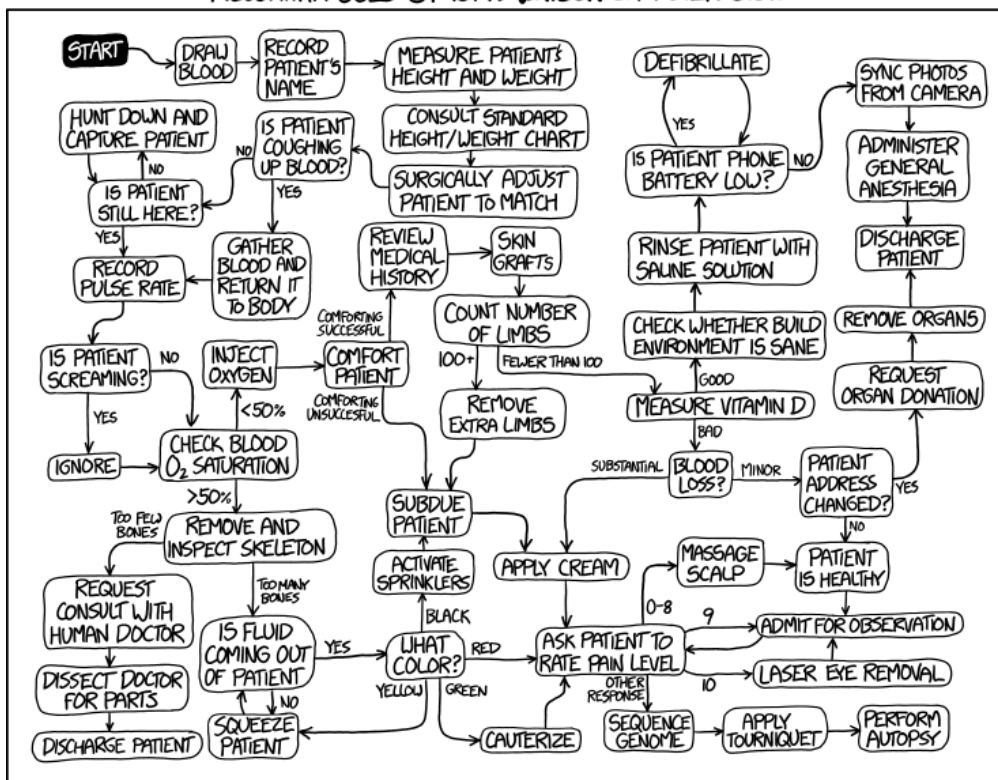
```

typedef struct interval{
    unsigned start;
    unsigned end;
} interval_t;

typedef struct tuple{
    unsigned time;
    int type;
} tuple_t;

```

A GUIDE TO THE MEDICAL DIAGNOSTIC AND TREATMENT
ALGORITHM USED BY IBM'S WATSON COMPUTER SYSTEM



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1619/>

Implementierung: Algorithmus

- Der eigentliche Algorithmus kann dann als Funktion relativ direkt implementiert werden:

```

interval_t marzullo(interval_t interval[], size_t len){
    tuple_t list[len*2], temp;
    interval_t r;
    size_t count, best;

    for(size_t i=0; i< len; i++){
        temp.time = interval[i].start;
        temp.type = -1;
        list[2*i] = temp;
        temp.time = interval[i].end;
        temp.type = 1;
        list[2*i+1] = temp;
    }
    // Sortieren. Bubble-Sort, da bekannt
    bool ch;
    do {
        ch = false;
        for(size_t i=0; i < 2*len-1; i++) {
            if (list[i].time > list[i+1].time) {
                temp = list[i+1];
                list[i+1] = list[i];
                list[i] = temp;
                ch = true;
            }
        }
    } while(ch);
    // der eigentliche Marzullo-Algorithmus
    count = best = 0;
    for(size_t i=0; i<len-1; i++){
        count -= list[i].type;
        if (count > best){
            best = count;
            r.start = list[i].time;
            r.end = list[i+1].time;
        }
    }
    return r;
}

```

Erkenntnisse

- Durch Modellierung und graphische Darstellung kann ein Problem leichter erfasst und Lösungsansätze gefunden werden.
- Halbformalen Beschreibungen wie Pseudocode helfen, algorithmische Lösungen losgelöst von den Nebenbedingungen der Programmiersprache darzustellen.

- Es dürfen Fähigkeiten der Programmiersprache (Typen, Bibliotheken, ...) berücksichtigt werden, aber **nur**, wenn es die Ausarbeitung der algorithmischen Idee nicht behindert
- Beim Übergang zur Implementation sollte man sich zuerst Gedanken über Datenstrukturen (d.h. Typen) machen

Merke

Es gilt eine Brücke zwischen dem Problem und den Fähigkeiten der Maschine (Computer) zu bauen, nicht den Fluss auf einmal zu überspringen.

- In den Kapiteln 12 und im Kapitel 13 werden anhand größerer Anwendungen nochmal Aspekte des Entwurfsprozesses besprochen



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1494/>

Knobelaufgaben

Aufgabe 8.1

Eine Anzahl von Wassergläsern stehen auf einem Tisch, manche davon verkehrt herum (Beispiel siehe Abbildung). Es sollen alle Gläser „richtig“ herum hingestellt werden, wobei stets **zwei** Gläser auf einmal gedreht werden müssen (nie ein einzelnes Glas).

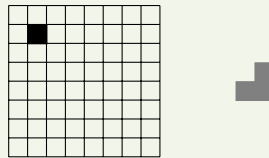


Unter welchen Startkonfigurationen gibt es eine Lösung?

Aufgabe 8.2: Trinominos

Ein Quadrat setzt sich aus $2^n \cdot 2^n - 1$ kleinen weißen Quadraten ($n \in \mathbb{N}$) und einem schwarzen Quadrat zusammen (Beispiel mit $n = 3$ siehe linke Abbildung). Ein Trinomino-Stein ist ein L-förmiger Stein, der drei kleine Quadrate abdeckt (siehe rechte Abbildung).

Beweisen Sie, dass die weißen Quadrate vollständig mit Trinomino-Steinen abgedeckt werden können, ohne dass sich Steine überlappen, Steine aus dem großen Quadrat ragen oder das schwarze Quadrat abgedeckt wird!



Kapitel 9

Logik und Automaten

Es ist aber der Zweck der Logik, nicht zu verwickeln, sondern aufzulösen, nicht verdeckt, sondern augenscheinlich etwas vorzutragen.

— Immanuel Kant

A computer is a state machine. Threads are for people who can't program state machines.

— Alan Cox

Das Binärsystem, die Grundlage der Funktion der (heutigen) Computer, benötigt bekanntlich nur zwei Ziffern: 0 und 1. Aber mit diesen beiden Ziffern kann eine eigene Algebra und – durch Interpretation der Wert als Wahrheitsaussagen – eine Logik aufbauen. Dieses Kapitel beschäftigt sich mit dieser Logik und der Algebra.

Außerdem betrachten wir Automaten, sowohl als theoretisches Konzept, als auch als praktischen ANsatz in der Programmierung.

9.1 Boolesche Algebra

Aussagelogik

- **Aussagelogik:** Teil der Logik, in dem Eigenschaften von Aussagen, die mittels Aussagenverknüpfungen aus anderen Aussagen entstehen, untersucht werden
- Begriff der Aussage wird **nicht** inhaltlich untersucht $\Rightarrow$ Prädikat
- Jede Aussage hat einen Wahrheitswert

Grundsätze

- Prinzip der **Zweiwertigkeit**: Jede Aussage hat entweder den Wert „wahr“ oder den Wert „falsch“
- Satz vom **ausgeschlossenen Dritten** (*tertium non datur*): Jede Aussage ist immer entweder wahr oder falsch
- Satz vom **ausgeschlossenen Widerspruch**: Keine Aussage ist zugleich wahr und falsch.

Prinzip der **Extensionalität**

Der Wahrheitswert einer Aussageverknüpfung hängt ausschließlich von den Wahrheitswerten ihrer Bestandteile ab.

• **Zuordnung**:

- wahre Aussagen $\Rightarrow t$ (*true*)
- falsche Aussagen $\Rightarrow f$ (*false*)

Es existieren auch mehrwertige Logiken:

- Zustand „unbestimmt“ (z.B. *open collector* in TTL)
- Zustandsänderungen als eigener Zustand
- Fuzzy-Logik

Boolesche Algebra

- GEORG BOOLE, 1815 - 1869
- Sei $\mathcal{B} = \{0, 1\}$
 - auch $\mathcal{B} = \{f, t\}$, $\mathcal{B} = \{\perp, \top\}$
 - entsprechen Wahrheitswerten
- Funktionen $f : \mathcal{B}^n \rightarrow \mathcal{B}^m$ mit $n, m \in \mathbb{N}, n, m \geq 1$ heißen **Boolesche Funktionen**
- Wenn $m = 1$: **echte Boolesche Funktion**
- Wichtige Verknüpfungsfunktionen $\Rightarrow n = 1$

Bezeichnung	$z = f(0)$	$z = f(1)$	Notation	alternative Bezeichnung
Identität	0	1	$z = x$	
Negation	1	0	$z = \neg x = \bar{x}$	Komplement
Einsfunktion	1	1	$z = \mathbf{1}(x)$	
Nullfunktion	0	0	$z = \mathbf{0}(x)$	

- Wichtige Verknüpfungsfunktionen $\Rightarrow n = 2$

Bezeichnung	$z = f(x, y)$				Notation	alternative Bezeichnung
	$x=0$ $y=0$	$x=0$ $y=1$	$x=1$ $y=0$	$x=1$ $y=1$		
Konjunktion	0	0	0	1	$z = x \wedge y = x \cdot y = xy$	Und
Disjunktion	0	1	1	1	$z = x \vee y$	Oder
Antivalenz	0	1	1	0	$z = x \oplus y$	Exklusiv-Oder
Äquivalenz	1	0	0	1	$z = x \equiv y = x \Leftrightarrow y$	Bisubjunktion
Implikation	1	1	0	1	$z = x \Rightarrow y$	Subjunktion

Andere Funktionen können zusammengesetzt werden, z.B.:

- Nicht-Und: $z = \overline{x \wedge y}$
- Nicht-Oder: $z = \overline{x \vee y}$

Interpretation der Implikation

$$x \Rightarrow y$$

x	y	$x \Rightarrow y$
<i>falsch</i>	<i>falsch</i>	<i>wahr</i>
<i>falsch</i>	<i>wahr</i>	<i>wahr</i>
<i>wahr</i>	<i>falsch</i>	<i>falsch</i>
<i>wahr</i>	<i>wahr</i>	<i>wahr</i>

Aussage: „Wenn das Wetter schön ist, gehe ich baden.“

- Aus „Wetter schön“ folgt „Baden“
- Jedoch ist **nicht** ausgeschlossen, dass auch bei schlechten Wetter gebadet wird
- Folglich wird die Aussage **nur falsch**, wenn bei schönen Wetter **nicht** gebadet wird

Logisch vollständige Mengen

- Gibt es eine Menge von Funktionen, mit denen alle anderen Funktionen dargestellt werden können?
- Ja, z.B.:
 - $\Omega = \{\vee, \neg\}$
 - $\Omega = \{\wedge, \neg\}$

- $\Omega = \{1, \oplus, \wedge\}$
- $\Omega = \{\text{Nicht-Und}\}$
- $\Omega = \{\text{Nicht-Oder}\}$

- Solche Mengen nennt man **logisch vollständig**

Boolesche Algebra

De Morgansche Regeln

- $\overline{x \wedge y} = \bar{x} \vee \bar{y}$
- $\overline{x \vee y} = \bar{x} \wedge \bar{y}$
- $x \Rightarrow y = \bar{y} \Rightarrow \bar{x}$

Assoziativgesetze

- $(x \wedge y) \wedge z = x \wedge (y \wedge z) = x \wedge y \wedge z$
- $(x \vee y) \vee z = x \vee (y \vee z) = x \vee y \vee z$
- $(x \oplus y) \oplus z = x \oplus (y \oplus z) = x \oplus y \oplus z$

Idempotenz

- $x \vee x = x$
- $x \wedge x = x$

Absorptionsregeln

- $x \vee \bar{x} = 1$
- $x \vee (x \wedge y) = x$
- $x \oplus x = 0$

Substitution von Konstanten

- $x \vee 0 = x$
- $x \wedge 0 = 0$
- $x \oplus 0 = x$

Kommutativgesetze

- $x \vee y = y \vee x$
- $x \wedge y = y \wedge x$
- $x \oplus y = y \oplus x$

Distributivgesetze

- $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
- $x \wedge (y \oplus z) = (x \wedge y) \oplus (x \wedge z)$

- $x \wedge \bar{x} = 0$
- $x \wedge (x \vee y) = x$
- $x \oplus \bar{x} = 1$

- $x \vee 1 = 1$
- $x \wedge 1 = x$
- $x \oplus 1 = \bar{x}$

Normalformen

- Boolesche Ausdrücke können sehr schnell unübersichtlich werden
- Abhilfe: Vereinheitlichung in Normalformen

Definition 9.1: Disjunktive Normalform

Eine boolesche Funktion ist in **disjunktiver Normalform** (DNF), wenn sie eine Disjunktion (Oder) von Konjunktionstermen (Und) ist, wobei die Konjunktionsterme nur (ggf. negierten) Funktionsparameter enthält.

$$y = \bigvee_i \left(\bigwedge_j [\neg] x_{i,j} \right)$$

Definition 9.2: Konjunktive Normalform

Eine boolesche Funktion ist in **konjunktiver Normalform** (KNF), wenn sie eine Konjunktion (Und) von Disjunktionstermen (Oder) ist, wobei die Disjunktionsterme nur (ggf. negierten) Funktionsparameter enthält.

$$y = \bigwedge_i \left(\bigvee_j [\neg] x_{i,j} \right)$$

- Normalformen können auf verschiedene Arten erstellt werden
 - Umformen mit Hilfe der booleschen Algebra
 - Nutzung von Wahrheitstabellen

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	0
1	1	1	1

DNF:

$$y = x_1 \bar{x}_2 \bar{x}_3 \vee x_1 x_2 \bar{x}_3 \vee \bar{x}_1 \bar{x}_2 x_3 \vee x_1 \bar{x}_2 x_3 \vee x_1 x_2 x_3$$

KNF:

$$y = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

Optimierung

- Mehrere Möglichkeiten, Ausdrücke zu reduzieren
- Beispiel **KV-Diagramm** (auch: *Karnaugh-Veitch-Diagramm* oder *Karnaugh-Diagramm*)

$$y = f(x_1, x_2, x_3) = x_1 \bar{x}_2 \bar{x}_3 \vee x_1 x_2 \bar{x}_3 \vee \bar{x}_1 \bar{x}_2 x_3 \vee x_1 \bar{x}_2 x_3 \vee x_1 x_2 x_3$$

	$x_1 x_2$			
	$\bar{x}_1 \bar{x}_2$	$\bar{x}_1 x_2$	$x_1 \bar{x}_2$	$x_1 x_2$
$\bar{x}_3$	0	0	1	1
x_3	1	0	1	1

- $y = x_1 \vee \bar{x}_2 x_3$
- Mehr in VL „Grundlagen der Technische Informatik“ oder „Einführung in die Funktion von Computern“

9.2 Logik in C

Typen

- C benutzt logische (boolesche) Ausdrücke und kennt auch einen entsprechenden Typ
- Da dieser Typ eng an Integer angelegt ist, wurde er vor C99 dem Programmierer nicht explizit zur Verfügung gestellt
- In C99 gibt es `_Bool`
- Durch Einbinden von `stdbool.h` steht Typ `bool` zur Verfügung
 - Werte: `false` und `true`
 - Bei C99 ist `bool` ein Synonym für `_Bool`



Merke:

Überall, wo ein Boolescher Typ verlangt wird, kann ein Integer stehen.

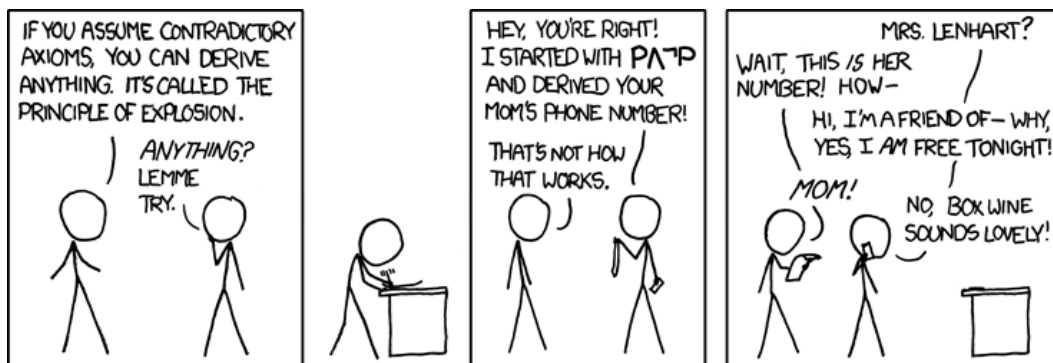
Dabei wird der Wert 0 als `false` interpretiert, jeder andere Wert als `true`.

Operatoren

- Wir haben boolesche Ausdrücke schon verwendet...
 - `if ( logischer Ausdruck ) ...`
 - `while ( logischer Ausdruck ) ...`
 - `do { ... } while ( logischer Ausdruck ) ...`
 - `for ( init, logischer Ausdruck, next ) ...`

- ... und Operatoren kennengelernt, die boolesche Werte liefern, z.B.
 - „`==`“ liefert `true`, wenn zwei Werte gleich sind
 - „`!=`“ liefert `true`, wenn zwei Werte ungleich sind
- Die folgende Tabelle listet alle in C eingebaute Operatoren, die einen booleschen Wert liefern:

Operator	Bezeichnung	Beispiel	Ergebnis
!	Negation	<code>!x</code>	$\bar{x}$
<	Kleiner-als	<code>x < y</code>	Wahr wenn x kleiner als y ist
>	Größer-als	<code>x > y</code>	Wahr wenn x größer als y ist
<=	Kleiner-gleich	<code>x <= y</code>	Wahr wenn x kleiner oder gleich y ist
>=	Größer-gleich	<code>x >= y</code>	Wahr wenn x größer oder gleich y ist
=	Gleich	<code>x == y</code>	Wahr wenn x und y den gleichen Wert haben
!=	Ungleich	<code>x != y</code>	Wahr wenn x und y nicht den gleichen Wert haben
&&	Und	<code>x && y</code>	Wahr wenn sowohl x als auch y wahr sind
	Oder	<code>x y</code>	Wahr wenn entweder x oder/und y wahr ist



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/407/>

Kurzschlussoperatoren

- Im Allgemeinen ist in C die Auswertungsreihenfolge in einem C-Ausdruck **nicht** festgelegt
- Die Logik-Operatoren `&&` und `||` bilden da eine Ausnahme

Kurzschlussoperatoren

Die Operatoren `&&` und `||` werden stets von links nach rechts ausgewertet und **zwar nur, bis das Ergebnis feststeht** ➔ **Kurzschlussoperatoren**

- Es ist eine gängige Programmiertechnik, links im Kurzschlussoperator eine Bedingung abzusichern, die rechts zu einem Laufzeitfehler führen würde

```

1  /* shortcut.c -- partial evaluation */
2  #include<stdio.h>
3
4  enum { arraysize=4 };
5  int z[arraysize] = { 2, 42, 0, 23 };
6  int r[arraysize];
7
8  int main()
9  {
10     for (int i=0; i<arraysize; ++i) {
11         (z[i] != 0) && (r[i] = 1000/z[i]);
12         printf("%d. value: %d\n",i,r[i]);
13     }
14     return 0;
15 }

```

Bedingungsoperator

- Wir haben im Kapitel 3 bereits das `if`-Statement kennengelernt, das die Ausführung eines Programms in Abhängigkeit von einer Bedingung steuert
- Wenn dagegen ein **Wert** von einer Bedingung abhängig sein soll, kann der **Bedingungsoperator** genutzt werden
 - Entspricht dem funktionalen Programmiermodell

Bedingungsoperator

Der Bedingungsoperator „?:“ ist der einzige ternäre (dreistellige) Operator in C. Er gibt in Abhängigkeit von einem logischen Ausdruck unterschiedliche Werte zurück.

cond ? trueval : falseval

Bedingung Rückgabewert wenn Bedingung wahr Rückgabewert wenn Bedingung falsch

- Die Funktion von „shortcut.c“ kann dann in etwa so erreicht werden:

```

1  /* cond-op.c -- condition operator */
2  #include<stdio.h>
3  #include<limits.h>
4
5  enum { arraysize=4 };
6  int z[arraysize] = { 2, 42, 0, 23 };
7  float r[arraysize];
8
9  int main()
10 {
11     for (int i=0; i<arraysize; ++i) {
12         r[i] = (z[i] != 0) ? 1000/z[i] : INT_MAX;
13         printf("%d. value: %d\n",i,r[i]);
14     }
15     return 0;
16 }

```

- Gegenüber der Lösung von Seite 179 kann hier im für den Fall der Division durch 0 ein Wert zugewiesen werden (z.B. INT_MAX, der in limits.h definiert ist)

Anmerkung

Für Gleitkommazahlen (float, double) ist die Division durch 0.0 übrigens meistens „sicher“: Bei vielen Plattformen ist das Ergebnis INFINITY, das in math.h definiert ist.

Ob die Gleitkomma-Division durch Null auf einer bestimmten Plattform eine Laufzeitausnahme generiert, kann getestet werden. Wenn der Ausdruck

math_errhandling & MATH_ERREXCEPT

einen Wert ungleich Null ergibt, wird eine Laufzeitausnahme erzeugt.



Mehrfachfallunterscheidung

- Häufig wird bei der Fallunterscheidung (Schlüsselwort „if“) durch den Gleichheitsoperator („=“) eine Ganzzahlvariable mit einem Literal verglichen
- Falls die Variable mehrere sinnvolle Werte annehmen kann, kommt es zum „Ketten-If“:

```

...
if (x == 23 ) {
...
} else if (x == 42 ) {
...
} else if (x == 111 ) {
...
} else {
...
}

```

- Dies ist nicht gut lesbar
- Es ist nicht gut zu erkennen, zu welchen if ein else gehört
- Für derartige Fälle hat C ein eigenes Konstrukt: die **Mehrfachfallunterscheidung**
- Neue Schlüsselwörter: switch, case und default

switch(Ausdruck) ← **muss von Type int sein**
 Wertet einen Ausdruck aus

case konstanter Ausdruck ← **ebenfalls int, muss zur Übersetzungszeit auswertbar sein**
 Markiert eine Sprungstelle im Switch-Block zu der gesprungen wird, wenn der Switch-Ausdruck mit dem Case-Ausdruck übereinstimmt

default:

Sprungstelle, wenn kein anderer Fall (case) zutrifft

- Nach dem switch **muss** stets ein Block beginnen, die Sprungstellen liegen **innerhalb** dieses Blocks
- Den Sprungstellen folgt stets ein Doppelpunkt („:“)



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1033/>

```
1 /* switch.c -- switch selection */
2 #include<stdio.h>
3 #include<stdlib.h>
4
5 int main(int argc, char *argv[])
6 {
7     if (argc !=2) return -1;
8     switch (atoi(argv[1])) {
9         default:
10            printf("Keine besondere Zahl\n");
11            break;
12        case 42:
13            printf("Antwort auf die Grosse Frage\n");
14            break;
15        case 23:
16            printf("Zahl der Illuminaten\n");
17            break;
18    }
19    return 0;
20 }
```

```
> ./switch 42
Antwort auf die Grosse Frage
> ./switch 5
Keine besondere Zahl.
```

Anmerkungen

- Vom `switch`-Schlüsselwort wird zum jeweilig zutreffenden Fall gesprungen
- Von dort wird solange weitergegangen, bis
 - zum Ende eines Blocks – oder
 - zu einem `break` – oder
 - zu einem `return`
- Bei Blockende wird „normal“ weitergearbeitet
- Bei `break` wird hinter das Blockende gesprungen
- Bei `return` wird die Funktion (und damit auch der `switch`-Block) beendet

Achtung

Ohne ein `break` oder `return` wird der nächste Fall abgearbeitet!

Falls dies tatsächlich beabsichtigt ist, sollte unbedingt ein Kommentar darauf hinweisen. In MISRA-C¹ ist das Auslassen eines `break` ganz verboten.

- Wenn kein Fall zutrifft, wird (so vorhanden) der `default`-Fall angesprungen
 - Der `default`-Fall kann weggelassen werden; dies wird aber **nicht empfohlen**
 - Es darf maximal **einen** `default`-Fall geben
 - Der `default`-Fall darf an beliebiger Stelle stehen; es ist jedoch günstig, ihn an den Anfang oder das Ende des `switch`-Blocks zu setzen

```

1 // punctuation.c -- switch fall through
2 #include<stdio.h>
3 #include<stdbool.h>
4
5 bool is_punctuation_mark(char c){
6     switch (c){
7         case '.': /* falls through */
8         case '!': /* falls through */
9         case '?': return true;
10        default : return false;
11    }
12 }
13
14 int main(){
15     char x;
16     printf("Input character: ");
17     scanf("%c",&x);
18     if (is_punctuation_mark(x))
19         printf("Punctuation mark\n");
20     else
21         printf("Not a punctuation mark\n");
22     return 0;
23 }

```

Bit-Operatoren

- Eng verwandt mit booleschen Operatoren sind die sogenannten **Bit-Operatoren**

¹Regelwerk der Motor Industry Software Reliability Association

- Dabei wird eine logische Operation **auf alle Bits** einer Ganzzahl angewendet

Operator	Entspricht
&	bit-weises Und
^	bit-weise Antivalenz
	bit-weises Oder
~	bit-weise Negation

Beispiel

Sei $x=42$ und $y=23$. Dann gilt:

$x \& y = 2$	$x \wedge y = 61$	$x y = 63$	$\sim y = -24$
00101010	00101010	00101010	
$\wedge$ 00010111	$\oplus$ 00010111	$\vee$ 00010111	$\neg$ 00010111
00000010	00111101	00111111	11101000

- Es gibt noch andere Bit-Operatoren, die aber eher mit arithmetischen als mit logischen Operationen verwandt sind

Operator	Entspricht
<<	bit-weises Linksschieben
>>	bit-weises Rechtsschieben

- Beide Operatoren haben zwei Argumente:
 - links steht der Ausgangswert
 - rechts die Anzahl von zu verschiebenden Bits
- Der Wert der „Auffüllbits“ ist unterschiedlich
 - Bei „<<“ wird mit 0 aufgefüllt
 - Bei „>>“ wird mit dem Wert des linken Bits aufgefüllt

Beispiel

$$\boxed{11001101} \gg 1 = \boxed{11100110}$$

Anmerkung

Linksschieben um n Bit entspricht (ohne Überlauf) einer Multiplikation mit 2^n .
 Rechtsschieben um n Bit entspricht einer Division durch 2^n (ggf. mit Abrunden).

```

1  /* shift.c -- demonstrates bit shift */
2  #include<stdio.h>
3
4  int main()
5  {
6      signed short int x = -42;
7      for (int i=0; i< (unsigned short)sizeof(short)*8; i=i+1)
8          printf("%3hd << %2hd = %6hd\t%3hd >> %3hd = %6hd\n",
9                 x, i, x << i, x, i, x >> i);
10
11     return 0;
12 }

```

```

> ./shift
-42 << 0 = -42      -42 >> 0 = -42
-42 << 1 = -84      -42 >> 1 = -21
-42 << 2 = -168     -42 >> 2 = -11
-42 << 3 = -336     -42 >> 3 = -6
-42 << 4 = -672     -42 >> 4 = -3
-42 << 5 = -1344    -42 >> 5 = -2
-42 << 6 = -2688    -42 >> 6 = -1
-42 << 7 = -5376    -42 >> 7 = -1
-42 << 8 = -10752   -42 >> 8 = -1
-42 << 9 = -21504   -42 >> 9 = -1
-42 << 10 = 22528    -42 >> 10 = -1
-42 << 11 = -20480   -42 >> 11 = -1
-42 << 12 = 24576    -42 >> 12 = -1
-42 << 13 = -16384   -42 >> 13 = -1
-42 << 14 = -32768   -42 >> 14 = -1
-42 << 15 = 0        -42 >> 15 = -1

```

Achtung!

Bit-Operatoren dürfen **nicht** mit den logischen Operatoren verwechselt werden!

- Bei Bit-Operatoren sind Argumente und Ergebnis Ganzzahlen
- Bei logischen Operatoren sind Argumente und Ergebnis Wahrheitswerte

Da Wahrheitswerte und Ganzzahlen in C nicht sauber unterschieden werden, geben die meisten Compiler auch keine Warnung aus!


```

1  /* bits.c -- logical vs. bit operators */
2  #include <stdio.h>
3
4  int main()
5  {
6      int x=42,y=85;
7      printf("First evaluation: ");
8      if (x && y) printf("true.\n");
9      else printf("false.\n");
10
11     printf("Second evaluation: ");
12     if (x & y) printf("true.\n");
13     else printf("false.\n");
14
15     return 0;
16 }

```

```

> ./bits
First evaluation: true.
Second evaluation: false.

```

Faulheitsoperatoren

- Da alle Ganzzahlausdrücke als logische Ausdrücke aufgefasst werden können, können praktisch alle Operatoren in logischen Ausdrücken auftreten
- Mit den Bit-Operatoren haben wir alle grundsätzlichen Operatoren kennengelernt
- Jedoch gibt es noch drei Operatorenklassen, die Kurzschreibweisen für andere Konstrukte bieten (⇒ **Faulheitsoperatoren** 😊)
- **Kommaoperator**: verbindet zwei Teilausdrücke
- **In-/Dekrementoperatoren**: Wertvergrößerung/-verkleinerung als Seiteneffekt
- **Selbstzuweisungsoperatoren**: Kurzschreibweisen für `var = var op arg`

Kommaoperator

- Der **Kommaoperator** „`,`“ lässt zwei Ausdrücke, die keinen Bezug zueinander haben (müssen), in **einem** Statement hintereinander stehen
- Der Wert eines Kommaoperatorausdrucks ist der Wert des zweiten Teilausdrucks
 - Der Wert des vorderen Teilausdrucks wird verworfen, nur evtl. Seiteneffekte sind relevant

- Der Kommaoperator garantiert (wie die Kurzschlussoperatoren) eine **Auswertungsreihenfolge** von links nach rechts
- Einsatz meist nur im Kopf von for-Schleifen
 - ➔ mehrere Initialisierungen oder Schleifenänderungen
- Der Kommaoperator hat eine **sehr** niedrige Priorität

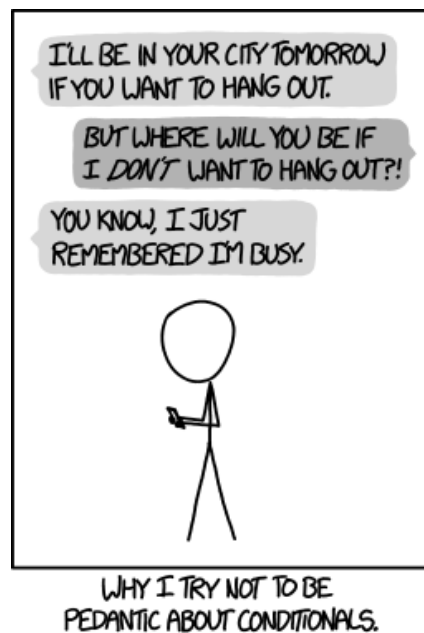
```

1  /* interval.c -- use of comma operator in for loops */
2  #include<stdio.h>
3  #include<stdlib.h>
4
5  int main(int argc, char *argv[]){
6      if (argc != 3) return -1;
7
8      for(int lower=atoi(argv[1]), upper=atoi(argv[2]);
9          lower<=upper; lower=lower+1, upper=upper-1) {
10         printf("[%d,%d] includes %d integer.\n",
11             lower, upper, upper - lower + 1);
12     }
13     return 0;
14 }

```

Inkrement- und Dekrementoperatoren

- Die Operatoren „++“ bzw. „--“ lassen sich auf alle **Skalartypen** (alle Ganzzahltypen, Gleitkommatypen, Zeiger) anwenden
- Operand **muss** ein L-Wert sein
- Operatoren **vergrößert** (++) bzw. **verkleinert** (--) den Operandenwert
 - um 1 bei Ganz- oder Gleitkommazahlen
 - um die Größe des Grundtyps bei Zeigern
- Die Operatoren gibt es als **Präfix-** und als **Postfixoperatoren**
 - Bei der Präfixvariante wird **erst** die Veränderung (Seiteneffekt) vorgenommen und **dann** der Wert des Gesamtausdrucks berechnet
 - Bei der Postfixvariante wird **erst** der Wert des Gesamtausdrucks berechnet und **dann** die Veränderung (Seiteneffekt) vorgenommen
- Operatoren werden häufig in Schleifen genutzt
- Unter Nutzung des Inkrement- und Dekrementoperators sieht das Intervallprogramm so aus:



Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/1652/>

```

1 // interval2.c -- use of increment/decrement
2 #include<stdio.h>
3 #include<stdlib.h>
4
5 int main(int argc, char *argv[]){
6     if (argc != 3) return -1;
7
8     for(int lower=atoi(argv[1]),
9         upper=atoi(argv[2]);
10        lower<=upper; ++lower, --upper){
11         printf("[%d,%d] includes %d integer.\n",
12             lower, upper, upper - lower + 1);
13     }
14     return 0;
15 }

```

Achtung!

Da i.d.R. die Auswertungsreihenfolge nicht festgelegt ist, müssen Seiteneffekte bei mehrmaligen Variablengebrauch in einem Ausdruck vermieden werden.

- Folgender Code demonstriert den Unterschied zwischen Prä- und Postfixoperator

```

1 #include<stdio.h>
2
3 int main()
4 {
5     int x=42;
6     printf("A: %d\n",x);
7     printf("B: %d\n",++x);
8     printf("C: %d\n",x++);
9     printf("D: %d\n",x);
10    return 0;
11 }

```

```

> ./prevspost
A: 42
B: 43
C: 43
D: 44

```

- Kommt es **nur** auf den Seiteneffekt an, ist in C egal, was genutzt wird

Selbstzuweisungsoperatoren

- Häufig gibt es Konstrukte wie: $x = x * 2$
- In diesem Fall gibt es einen Zuweisungsoperator, der den gleichen Effekt hat:

```

x *= 2;

```

- Analog gibt es:

+=	%=	=
-=	&=	>>=
/=	^=	<<=

- **Achtung:** Mitunter wird die Nutzung dieser Operatoren als unübersichtlich empfunden

Priorität und Assoziativität

- Wenn mehrere Operatoren in einem Ausdruck benutzt werden, hängt die Auswertungsreihenfolge von der Priorität und der Assoziativität ab
 - Priorität: Operatoren mit höherer Priorität werden zuerst ausgewertet
 - Assoziativität: Bestimmt die Richtung der Auswertung

Prio.	Operator	Assoz.
<i>hoch</i>	() [] -> .	⇒
	++ - ! ~ (type) sizeof + ¹ - ¹ * ² & ²	⇐
	* / %	⇒
	+ -	⇒
	<< >>	⇒
	< <= >= >	⇒
	= !=	⇒
	&	⇒
	^	⇒
		⇒
	&&	⇒
		⇒
	?:	⇐
	= += -= *= /= %= &= ^= = <= >=	⇐
<i>niedrig</i>	,	⇒

¹Vorzeichen ²Adressoperatoren

! Achtung!

Der Operatorenvorrang kann zu unerwarteten Ergebnissen führen. Im Zweifelsfall sollte immer geklammert werden.

```

1 // rank.c -- operators' rank
2 #include<stdio.h>
3 #include<stdbool.h>
4
5 int main()
6 {
7     int i;
8     int j=0xFF42;
9     bool b;
10    b = 0x42 == j & 0x00FF;
11    i= 23,42;
12    printf("i=%d, b=%d\n",i,(int)b);
13    return 0;
14 }

```

```

./rank
i=23, b=0

```

9.3 Automaten

- Betrachte folgende Aufgabenstellung:

Ein Programm soll die Wörter eines Satzes in umgekehrter Reihenfolge ausgeben.
Satzzeichen sollen in der Ausgabe nicht erscheinen.

- Satz liegt als Zeichenkette vor
 - Erste Aufgabe: Erkennung von Wörtern
 - Ansatz:
 - Wenn ein Buchstabe einem Leer- oder Satzzeichen **folgt**, beginnt hier ein Wort
- ➔ es muss also bekannt sein, was **vorher** geschah



Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/376/>

Satzinvertierung

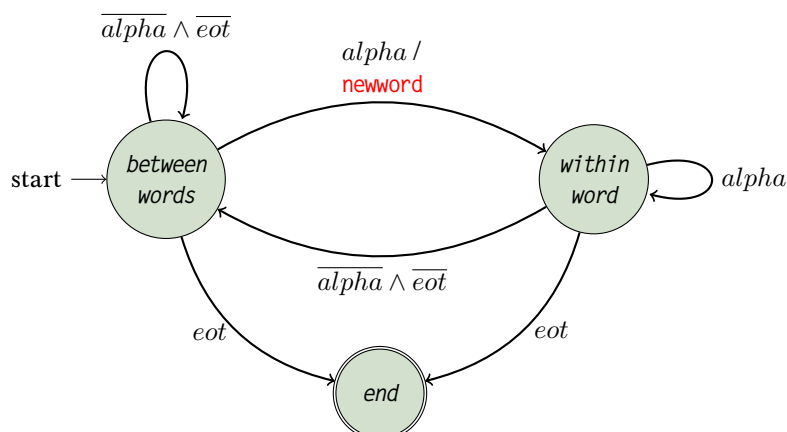
- Im folgendem Code wird in der Variable `skip_space` gespeichert, in welchem **Zustand** sich die Suche befindet
 - **True**: es wurden schon Leerzeichen gelesen
 - **False**: letztes Zeichen war kein Leerzeichen

```

1  /* reverse.c -- boolean states */
2  #include<stdio.h>
3  #include<stdbool.h> // true, false
4  #include<ctype.h>    // for: isalpha
5
6  enum{ maxwords=20};
7  char* words[maxwords];
8
9  int main(int argc, char* argv[]) {
10     int j=0;
11     _Bool skip_space;    /* continue as long as there are spaces */
12     if (argc != 2) return -1; /* None or more than one argument */
13
14     for(int i=0, skip_space=true; argv[1][i]!='\0'; ++i){
15         if ((skip_space==true) && (isalpha(argv[1][i])==true)) {
16             words[j++] = &argv[1][i];
17             skip_space = false;
18         } else if ((skip_space==false) && (isalpha(argv[1][i])==false)) {
19             argv[1][i]='\0';
20             skip_space = true;
21         }
22     }
23
24     for (int i=j-1; i>=0; --i)
25         printf("%s ",words[i]);
26     printf("\n");
27     return 0;
28 }

```

Zustand



```
> ./reverse "all you need is: love."
love is need you all
```

Definition

Definition 9.3: Endlicher deterministischer Automat

Ein **endlicher (deterministischer) Automat** (*EA*, *Zustandsmaschine*, *finite state machine* $\Rightarrow$ *FSM*, auch: *finite state automata*) ist ein Modell zur Beschreibung von Abläufen (z.B. in Computern).

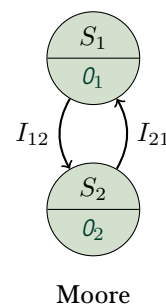
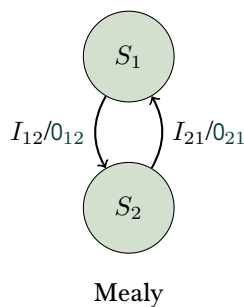
Ein EA besteht aus einer Menge von **Zuständen** S (*states*) und **Zustandsübergängen** $T : S \times \Gamma \rightarrow S$ (**Transitionen**, *transitions*).

Ein EA startet in einem **Startzustand**. Er „verarbeitet“ eine Sequenz von **Zeichen** oder **Ereignissen** Γ . Dabei bestimmt das nächste Zeichen/Ereignis, in welchen Zustand der EA wechselt.

Ein EA kann einen oder mehrere **Endzustände** (*accepting states*) besitzen. Wird ein solcher Zustand erreicht, ist die Abarbeitung beendet.

Aktionen

- Zustandsänderungen in einem endlichen Automaten können zu **Aktionen** oder **Ausgaben** führen
- Zwei Varianten:
 - Aktion/Ausgabe ist an einen bestimmten Übergang gebunden ($\Rightarrow$ **Mealy-Automat**)
 - Aktion/Ausgabe ist an die Ankunft in einem bestimmten Zustand gebunden ($\Rightarrow$ **Moore-Automat**)

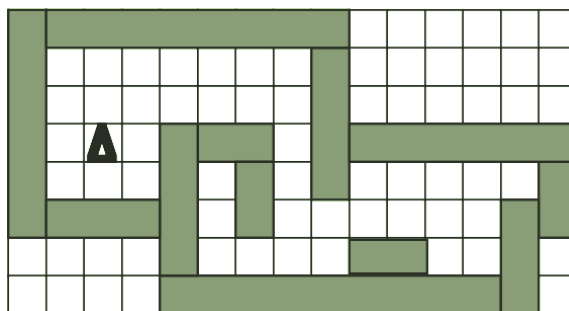


Programmentwurf mit Automaten

- Es ist häufig hilfreich, einen Ablauf zunächst als Automaten zu entwerfen
- Die Programmierung kann dann nach einem einheitlichen Muster erfolgen

Ein Roboter hat zwei Sensoren, die feststellen, ob unmittelbar vor ihm ($h = true$) oder unmittelbar rechts von ihm ($r = true$) ein Hindernis (Wand) ist. Er kann zwei Aktionen durchführen: Vorwärts gehen (F) und sich 90° nach rechts drehen (R).

Er hat die Aufgabe, in einem Labyrinth eine Wand zu finden und dann sich immer mit der rechten Seite an der Wand entlang bewegen.

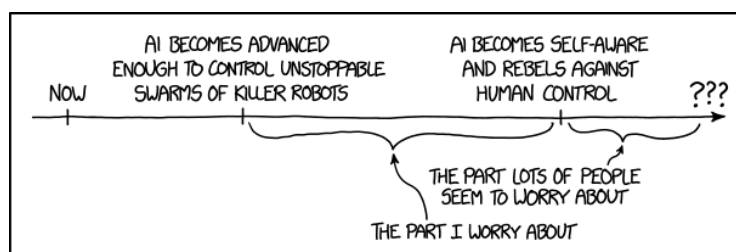
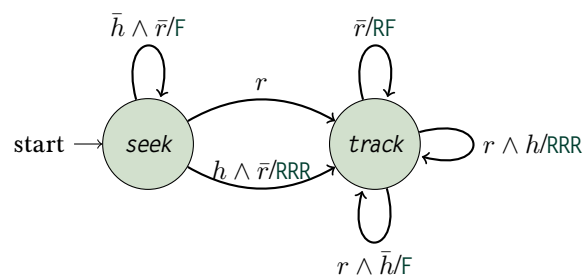


Aufgabe



$r = true$

- Automat:



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1968/>

```

○                                     :
○                                     :
○ 1  switch (state){
○ 2  case seek:
○ 3      if ((h==false) && (r==false))
○ 4          robot(forward);
○ 5      else if (r==true)
○ 6          state=track;
○ 7      else if ((h==true) && (r==false)) {
○ 8          robot(turn); robot(turn); robot(turn);
○ 9          state=track;
○10     }
○11     break;
○12 case track:
○13     if (r==false){
○14         robot(turn);
○15         robot(forward);
○16     }
○17     else if ((r==true) && (h==true)){
○18         robot(turn); robot(turn); robot(turn);
○19     }
○20     else if ((r==true) && (h==false))
○21         robot(forward);
○22     break;
○23 }
○24 }
○                                     :
○                                     :

```

Knobelaufgaben

Aufgabe 9.1

Entwickeln Sie einen Automaten, der ausgibt (terminiert), ob eine römische Zahl korrekt ist.

Beispiele:

- *XIX* ⇒ korrekt
- *XLIX* ⇒ korrekt
- *XVIX* ⇒ nicht korrekt
- *IL* ⇒ nicht korrekt

Aufgabe 9.2

Elf große Schachteln stehen auf einem Tisch. Eine unbekannte Anzahl von Schachteln davon wird ausgewählt und in jede acht kleinere Schachtel platziert. Eine unbekannte Anzahl dieser kleineren Schachteln wird ausgewählt und in jede acht winzige Schachtel platziert.

Zum Schluss gibt es 102 leere Schachteln. Wie viele Schachteln gibt es insgesamt?

Kapitel 10

Syntax

Without concrete symbols, a computer is merely a pile of junk

— Neil Postman

A computer program is a message from a man to a machine. The rigidly marshaled syntax and the scrupulous definitions all exist to make intention clear to the dumb engine.

— Fred Brooks

10.1 Einführung

- Woher weiß der Compiler, dass ein Programm nicht korrekt ist?
- Unterscheiden: **Grammatik** und **Semantik**

Grammatik (...wie wir es aus der Schule kennen.)

Formenlehre von Wörtern (**Morphologie**) und von Sätzen (**Syntax**)

- Bei Programmiersprachen spielt die Morphologie (praktisch) keine Rolle ➔ Grammatik und Syntax sind weitgehend synonym
- Die Bedeutung bzw. der Sinn (genannt: **Semantik**) des Satzes spielt keine Rolle

Syntax

Definition 10.1: Syntax, informal

Die Syntax einer Sprache ist eine Menge von Regeln, nach der Ausdrücke gebildet werden können.

- Die Aussage gilt für natürliche Sprachen (Deutsch, Englisch,...) wie für formale (C, Python, Java, aber auch mathematische Logik, Beschreibungssprachen, etc.)

- Die Syntax natürlicher Sprachen ist aber weitaus komplizierter als die (der meisten) formalen Sprachen
- Ausdrücke im Sinne der Definition können z.B. Sätze, Formeln oder Programmstatements sein

Beispiele



Quelle: Foto von Guillaume Blanchard, GNU FDL; aus TILL TANTAU: Beamer-Beispiel-Lecture

- Wir wissen nicht, was der Text bedeutet¹
- Es gibt aber offensichtlich Regeln, die eingehalten werden, z.B. „Hieroglyphen stehen in Zeilen“

In Sprachen wie Deutsch und Englisch kann man Sätze nach der S-P-O-Regel bilden

- Die Informatiker lieben die mathematische Logik. ✓
- Ein Tisch beißt die anaphylaktische Algebra. ✓
- Der Tiger jagt heute eine Antilope. ✓
- Dieser Satz kein Verb. ✗

Achtung

Je besser man die Semantik versteht, um so schwerer fällt es oft, die Syntax zu erkennen.

Auch mathematische Ausdrücke unterliegen syntaktischen Regeln:

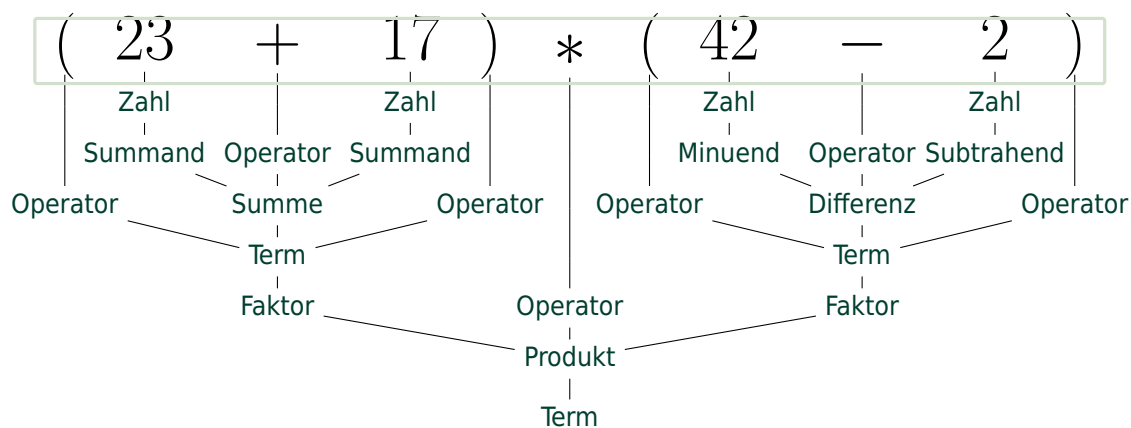
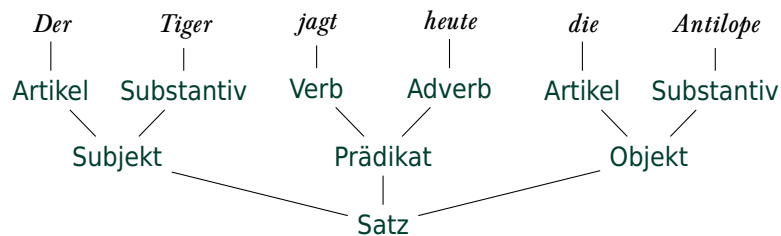
- $(a + b) \cdot (c - d)$ ✓
- $((a/b))$ ✗

¹Dies gilt zumindest für den Autor dieses Skripts. 😊

- $a \cdot b + (c \cdot \frac{a+d}{e})$



- $: -)$



10.2 Formalisierung

Alphabet

Zur Formalisierung definieren wir eine Reihe von Begriffen:

- Jede formale Sprache besteht aus **Symbolen** (Zeichen, *token*)
 - Ein Symbol ist die **kleinste** Einheit der Betrachtung **innerhalb der Sprache**
 - Symbole können z.B. einzelne Zeichen, Kombinationen davon oder auch ganze Wörter sein

Definition 10.2: Alphabet

Jede endliche, nichtleere Menge Σ von Symbolen wird **Alphabet** genannt.

- Aus den Elementen in Σ können **Ausdrücke** (auch: „Zeichenketten“, „Worte“, „Sätze“) gebildet werden

- Der leere Ausdruck wird mit ε bezeichnet
- **Beispiele:**
 - Die Menge $\Sigma_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ ist hinreichend, um alle natürlichen Zahlen in Dezimalsystem darzustellen
 - Aus dem Alphabet $\Sigma_2 = \{\bullet, \text{—}\}$ können alle **Buchstaben** des Morsealphabets gebildet werden
 - Aus dem Alphabet $\Sigma_3 = \{\bullet, \text{—}, \sqcup\}$ können alle **Morse-Nachrichten** gebildet werden
 - Aus dem Alphabet $\Sigma_4 = \left\{ \begin{array}{c} \text{♠}, \text{♠}, \text{♠}, \text{♠}, \text{♠}, \text{♠}, \text{♠}, \text{♠}, \text{♠}, \text{♠}, \\ \text{♥}, \text{♥}, \text{♥}, \text{♥}, \text{♥}, \text{♥}, \text{♥}, \text{♥}, \text{♥}, \text{♥}, \\ \text{♦}, \text{♦}, \text{♦}, \text{♦}, \text{♦}, \text{♦}, \text{♦}, \text{♦}, \text{♦}, \text{♦}, \\ \text{♣}, \text{♣}, \text{♣}, \text{♣}, \text{♣}, \text{♣}, \text{♣}, \text{♣}, \text{♣}, \text{♣} \end{array} \right\}$ können alle „Skat-Hände“ gebildet werden
 - Das Alphabet $\Sigma = \{A, T, G, C\}$ wird zur Beschreibung von Gensequenzen genutzt

Formale Sprachen

Definition 10.3: Kleenesche Hülle

Die Menge aller aus den Symbolen von Σ bildbaren endlichen Zeichenketten und der leeren Zeichenkette ε heißt **Kleenesche Hülle** Σ^* von Σ .

- Beispiel:

$$\Sigma = \{a, b\} \Rightarrow \Sigma^* = \{\varepsilon, a, b, aa, bb, ab, ba, aaa, aab, aba, baa, abb, bab, \dots\}$$
- Die Menge aller Zeichenketten ohne die leere Kette wird mit Σ^+ bezeichnet

Definition 10.4: Formale Sprache

Jede Teilmenge $L \subseteq \Sigma^*$ ist eine **formale Sprache**.

- Im Folgenden ist mit „Sprache“ immer „formale Sprache“ gemeint
- Man spricht von einer Sprache L **über** Σ
- **Beispiele für Sprachen:**
 - Die Menge $\{AAA, AAC, GAA\} \Rightarrow$ endliche Sprache
 - Die Menge aller Kartenpaare, die sich beim Skat im Skat befinden können $\Rightarrow$ endliche Sprache
 - Die Menge aller möglichen Schachspiele $\Rightarrow$ sehr groß, aber endlich
 - Menge aller C-Programmtexte $\Rightarrow$ unendliche Sprache
 - Die Menge aller Basensequenzen, die $GTAC$ enthalten $\Rightarrow$ unendlich

- Es stellt sich die Frage, wie man die (Syntax der) Sprache beschreiben soll
- **Anders ausgedrückt:** Welche Ausdrücke/Zeichenketten sind in einer Sprache zulässig und welche nicht?
- Aufzählung ist unpraktisch $\Rightarrow$ oft ist die Anzahl der Ausdrücke sehr groß oder unendlich
- **Lösung:** Es wird beschrieben, wie man legale Ausdrücke **generieren** kann
 - Man spricht von einer **generativen Grammatik**

Generative Grammatik

- Bei einer generativen Grammatik der Sprache L wird neben dem Alphabet Σ von L ein zweites Alphabet V von Variablen benutzt.
- Zur Unterscheidung nennt man
 - ... die Elemente aus $\Sigma \Rightarrow$ **Terminalsymbole**
 - ... die Elemente aus $V \Rightarrow$ **Nichtterminalsymbole**
 - Wenn man abstrakt von diesen Symbolen spricht, werden die i.d.R. die Elemente aus Σ mit Kleinbuchstaben ($a, b, c, \dots$) und die Elemente aus V mit Großbuchstaben bezeichnet
- Außerdem braucht man eine Menge von **Regeln** P , die beschreiben, wie man aus Elementen aus $(\Sigma \cup V)^+$ Elemente aus $(\Sigma \cup V)^*$ bildet:

$$P : (\Sigma \cup V)^+ \rightarrow (\Sigma \cup V)^*$$

- Diese Regeln werden auch **Produktionsregeln** genannt
- Um ein Ausdruck zu generieren, beginnt man mit einem **Startsymbol** $S \in V$
- Was bedeutet das praktisch?
 - Terminalsymbole sind Symbole, die man in der Sprache **tatsächlich** findet
 - In C wären das z.B. die einzelnen Schlüsselwörter, aber auch Operatoren, einzelne Zeichen der Identifier, etc.
 - Nichtterminalsymbole sind **Abstraktionen**, die noch untersetzt werden müssen (noch nicht endgültig = nicht *terminal* sind)
 - Beispielsweise ist in C das Konzept „Schleife“ noch nicht terminal, sondern kann in unterschiedlichen Ausprägungen vorkommen
- In einem konkreten Text der Sprache (der durch die Grammatik beschrieben wird) kommen **keine** Nichtterminalsymbole mehr vor
 - Wenn die Grammatikregeln P nicht die „Auflösung“ aller Nichtterminalsymbole in Terminalsymbole zulassen, ist die Grammatik (für unsere Einsatzfälle) unvollständig

Grammatiken

Nach der Art der Regeln P unterscheidet man verschiedene Arten von Grammatiken

Linksregulär: In jeder Regel aus P darf **links** nur ein Nichtterminalsymbol und **rechts** entweder ein Terminalsymbol oder ein Nichtterminalsymbol, gefolgt von einem Terminalsymbol stehen:

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow Ba \end{aligned}$$

Rechtsregulär: In jeder Regel aus P darf **links** nur ein Nichtterminalsymbol und **rechts** ein Terminalsymbol, das von maximal einem Nichtterminalsymbol gefolgt wird, stehen:

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow aB \end{aligned}$$

Regulär: Entweder **alle** Regeln aus P entsprechen einer linksregulären Grammatik oder **alle** Regeln aus P entsprechen einer rechtsregulären Grammatik

Kontextfrei: In jeder Regel aus P darf links nur ein Nichtterminalsymbol stehen:

$$A \rightarrow \dots$$

Kontextbehaftet: In jeder Regel aus P muss links ein Nichtterminalsymbol stehen, dass von (ggf. leeren) Zeichenketten eingerahmt wird; rechts müssen die gleichen Zeichenketten eine nichtleere Zeichenkette einrahmen:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

Allgemein: Beliebige Regeln; keine Einschränkung

- Alternativ nennt man ...
 - ... allgemeine Grammatiken auch **Typ-0-Grammatiken**
 - ... kontextbehaftete Grammatiken auch **Typ-1-Grammatiken**
 - ... kontextfreie Grammatiken auch **Typ-2-Grammatiken**
 - ... reguläre Grammatiken auch **Typ-3-Grammatiken**
- Die Grammatiken bilden die sogenannte **CHOMSKY²-Hierarchie**



Bildquelle: Wikipedia, Ministerio de Cultura de la[2ex] Nación Argentina

- Mit jeder Typ- $(n - 1)$ -Grammatik kann man alles (und mehr) „machen“, was man mit Typ- n kann
- Die Verarbeitung allgemeinerer Sprachen verlangen komplexere Automaten
 - Für Typ-0 ist eine **Turing-Maschine** (allgemeines Modell eines Computers) notwendig³
 - Es existieren auch formale Sprachen, die nicht auf diese Weise generiert werden können
- In der theoretischen Informatik relevant

Grammatik und Automaten

- Regulären Grammatiken sind eng verwandt mit endlichen Automaten (siehe Kapitel 9.3):

Äquivalenz von EA und regulären Grammatiken

Jede Folge von zulässigen Eingabesymbolen bzw. -ereignissen, die in einen Endzustand führen, entspricht einer formalen Sprache, die durch eine reguläre Grammatik beschrieben werden kann.

Oder anders formuliert:

Zu jeder regulären Grammatik existiert (mindestens) ein endlicher Automat, der genau alle Ausdrücke dieser Sprache akzeptiert.

- Für kontextfreie Sprachen ist ein endlicher Automat **nicht** hinreichend

²Noam Chomsky, *1928, US-amerikanischer Linguist und Bürgerrechtler

³... aber nicht in allen Beziehungen hinreichend

Beispiel

Allgemeine Klammerausdrücke (Regel: Klammern müssen in Paaren auftreten) sind nicht regulär und lassen sich nicht durch einen endlichen Automaten beschreiben.

- **Mögliche Abhilfen:**
 - Automat mit Kellerspeicher
 - Rekursiver Aufruf des Automaten

Syntaxdiagramme

Wie können die Regeln einer Grammatik beschrieben werden?

- Beschränken uns auf (maximal) kontextfreie Grammatiken
- Ansätze:
 - Syntaxdiagramme
 - (Erweiterte) Backus-Naur-Form

Syntaxdiagramme

- **Bestandteile**
 - abgerundete Kästchen $\Rightarrow$ Terminalsymbole
 - eckige Kästchen $\Rightarrow$ Nichtterminalsymbole
 - Verbindungen aus Linien und Pfeilen
- Jeder (in Pfeilrichtung) begehbare Weg ist ein valider Ausdruck

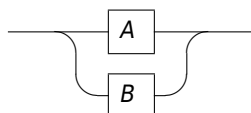
Terminalsymbol:



Nichtterminalsymbol:



Alternative:



Option:

Erweiterte Backus-Naur-Form

Erweiterte Backus-Naur-Form: EBNF

- Die BNF lässt sich direkt in kontextfreie Grammatik übersetzt
 - Benötigt aber (z.B. bei Wiederholungen) syntaktische Hilfsvariablen
- Erweiterung ➔ Erweiterte Backus-Naur-Form
- Wie BNF aber mit
 - ... Beschreibung **optionaler Teile** [...]
 - ... Beschreibung von **Wiederholungen** { ... }
- Syntaktische Unterschiede:
 - Beliebige Klammerung
 - Definitionszeichen ist „=“
 - Terminalzeichen werden in Anführungszeichen „“ oder ’“ gesetzt
 - Nichtterminalzeichen nicht besonders gekennzeichnet
 - Ausdrücke enden mit Semikolon „;“

Weitere Erweiterungen in der ISO/IEC 14977:1996(E), z.B.:

- Leerzeichen in Nichtterminalbezeichnern
 - ➔ Sequenz dann mit Komma „,“ zwischen Folgeausdrücken

signed integer = sign, integer;

- Bestimmte Anzahl von Wiederholungen, z.B. „4 * (...)“
- Kommentare: „(* Dies ist ein Kommentar *)“

Definition 10.5: Vollständigkeit einer (kontextfreien) Grammatik

Eine (E)BNF-Definition oder ein Syntaxdiagramm-Beschreibung ist **vollständig**, wenn es für jedes Nichtterminalsymbol, das auf einer rechten Regelseite auftaucht, auch eine Regel gibt, wo es auf linken Seite ist.

Als Beispiel sei noch mal die (vereinfachte) Grammatik der Funktionsdeklaration von Seite 207 in EBNF betrachtet:

```
function declaration = [storage class], type, identifier, "(", [ "void" | parameter ], ")", ";"
parameter = type, [ identifier ], { ",", type, [ identifier] }
```

Diese Beschreibung ist **nicht** vollständig, da `storage class`, `type` und `identifier` nicht auf der linken Seite auftauchen.

10.3 Anwendung

Reguläre Grammatik in Aktion

- Es ist eine häufige Aufgabe, Daten zu finden, die eine bestimmte **Struktur** haben
- Dazu gibt es Werkzeuge, die mit Hilfe von **regulären Ausdrücken** suchen können
- Reguläre Ausdrücke beschreiben formale Sprachen, die durch eine reguläre Grammatik generiert werden
- Beispiele für Tools: `grep`, `sed`, `awk`, `perl`, `Python`, `C#`, ...
- Syntax weicht bei den einzelnen Tools voneinander ab
- Standardisierung in POSIX
- Da nur ein begrenzter Vorrat an Zeichen zur Verfügung steht, wird zwischen „normalen“ (Terminal-)Zeichen und **Metazeichen** unterschieden, die eine besondere Bedeutung haben
- Typisch sind folgende Metazeichen
 - `.` (*Punkt*): steht für (nahezu) jedes andere Zeichen
 - `a | b` (*senkrechter Strich*): oder, d.h. es ist „a“ oder „b“ erlaubt
 - `[ abc ]` (eckige Klammern): steht für **eines** der Zeichen in der Klammer, also „a“ oder „b“ oder „c“
 - `*` (*Stern*): der vorherige Ausdruck kann beliebig oft (auch überhaupt nicht) auftreten
 - `+` (*Plus*): der vorherige Ausdruck kann beliebig oft auftreten, aber wenigstens einmal
 - `{n,m}` (*zwei Zahlen in geschweiften Klammern*): der vorstehende Ausdruck tritt mindestens *n*-mal und maximal *m*-mal auf
 - `\` (*Rückstrich*): Die Wirkung des folgenden Metazeichens wird aufgehoben

Zeichenklassen

- Um verschiedene Zeichen zusammenzufassen, gibt es sogenannte Zeichenklassen, z.B.:
 - `[:alpha:]` Steht für alle Buchstaben *a, b, ..., z, A, B, ... C*
 - `[:digit:]` Steht für die Ziffern 0 ... 9
 - `[:alnum:]` Vereinigung von `[:alpha:]` und `[:digit:]`
 - `[:blank:]` Leerzeichen und Tabulator
 - `[:print:]` Steht für jedes druckbare Zeichen

Beispiel `grep`

- `grep` wird u.a. genutzt, um Zeilen in Dateien zu finden, die bestimmte Inhalte haben.
- Beispielsweise findet


```
grep -E "typedef ([[:alpha:]]*[[:alnum:]]*[:blank:]]+)+time_t;" *.h
```

 alle Definitionen von `time_t` in Header-Files.

```
> grep -E "typedef ([[:alpha:]][[:alnum:]]*[[:blank:]]+)+time_t;" *.h
utime.h:typedef __darwin_time_t    time_t;
utmp.h:typedef __darwin_time_t    time_t;
```

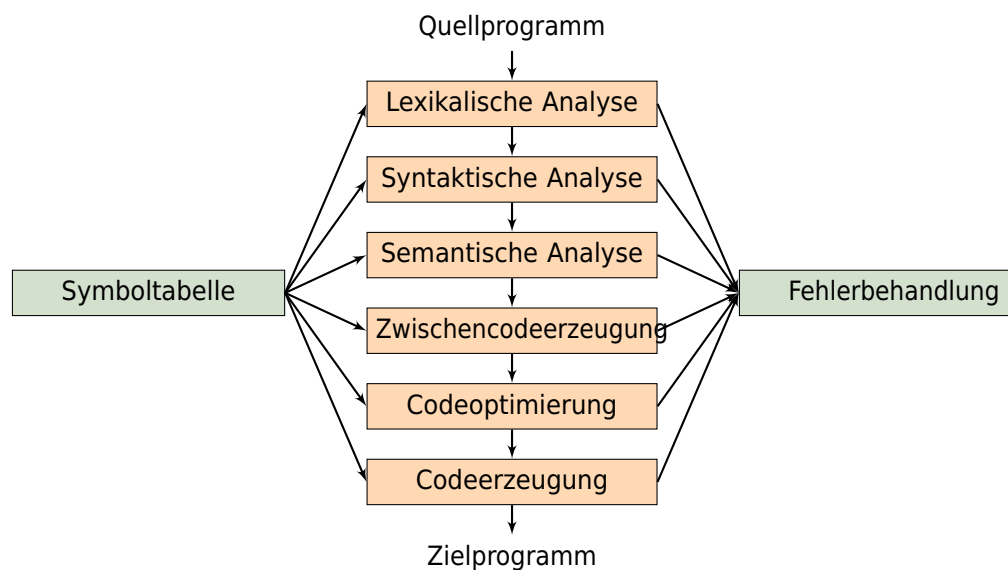
- Dagegen findet

```
grep -E "time_t[[:blank:]][[:alpha:]][[:alnum:]]*[[:blank:]]*\(\" *.h
```

alle Funktionsdeklarationen, die `time_t` als Rückgabebetyp haben

```
> grep -E "time_t[[:blank:]][[:alpha:]][[:alnum:]]*[[:blank:]]*\(\" *.h
time.h:time_t mktime(struct tm *) __DARWIN_ALIAS(mktime);
time.h:time_t time(time_t *);
time.h:time_t posix2time(time_t);
timeconv.h:time_t _long_to_time(long tlong);
timeconv.h:time_t _int_to_time(int tint);
```

Compiler

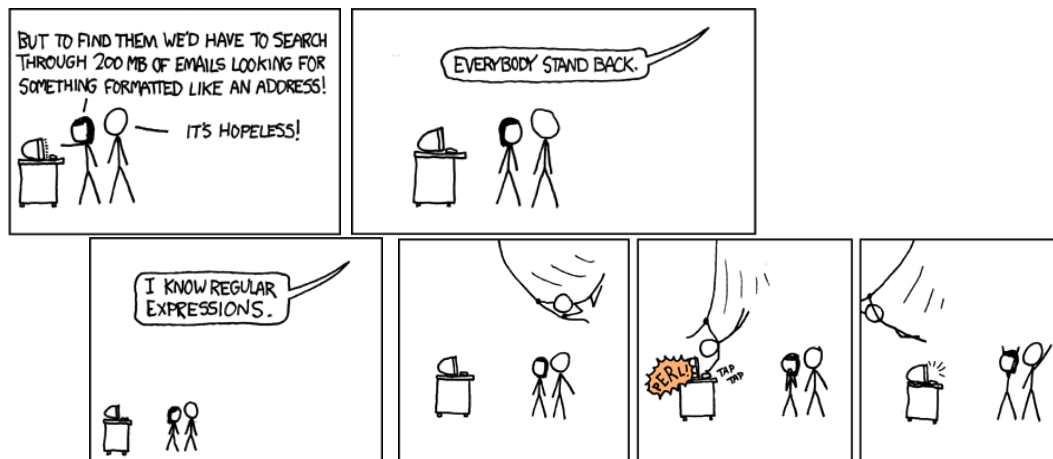


Formale Sprachen im Compiler

In einem modernen Compiler werden mehrmals formale Grammatiken verwendet:

- **Lexikalische Analyse** ➔ Identifizierung lexikalischer Elemente
 - Typisch: Reguläre Grammatik
- **Syntaktische Analyse**
 - Wurden die Syntaxregeln der Sprache eingehalten?

- Typisch: Kontextfreie Sprache
- Für die **Semantische Analyse** werden in der Regel kontextsensitive Sprachen verwendet
 - ➔ Bei Interesse mehr in VL „Compilerbau“ (Prof. Rüniger)



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/208/>

Token

- Die erste Stufe ist die Identifikation von lexikalischen Elementen
- Unterteilung des Eingabetexts in Schlüsselwörter, Bezeichner, Literale, ... ➔ **Token**
- Evtl. Verwerfen von Elementen
 - Z.B. werden in C Leerzeichen u.ä. ignoriert⁴ (➔ Whitespaces)
 - Dagegen gehören Leerzeichen in Python zur Syntax ➔ Einrückungen haben Bedeutung

```
/* token.c — greedy token parsing */

int main()
{
    int x=2,y=42, *px=&x;
    y=y/*px;
    /* use return value for result */;
    return y;
}
```

⁴Sie können aber zur Trennung anderer lexikalischer Elemente dienen.

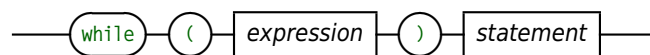
```
> gcc token.c -o token
> ./token
> echo $?
42
```

Syntax am Beispiel von Schleifen

Betrachten als Beispiel Schleifen in C (vgl. Kapitel 7)

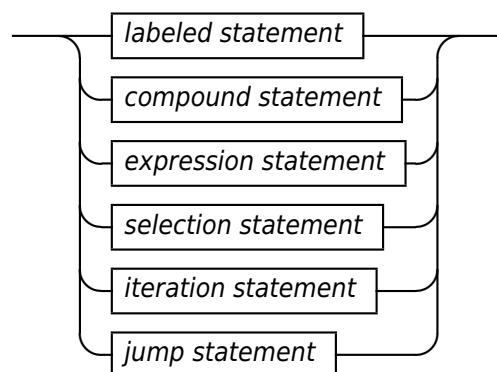
- Betrachten Regel für `while`-Schleife als Syntax-Diagramm:

`while` loop

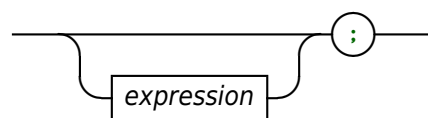


- Da „expression“ und „statement“ Nichtterminalsymbole sind, müssen sie (bis zur Vollständigkeit) definiert werden
- Verfolgen am Beispiel Regeln für „statement“

`statement`



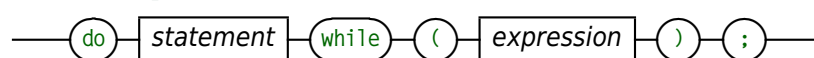
`expression statement`



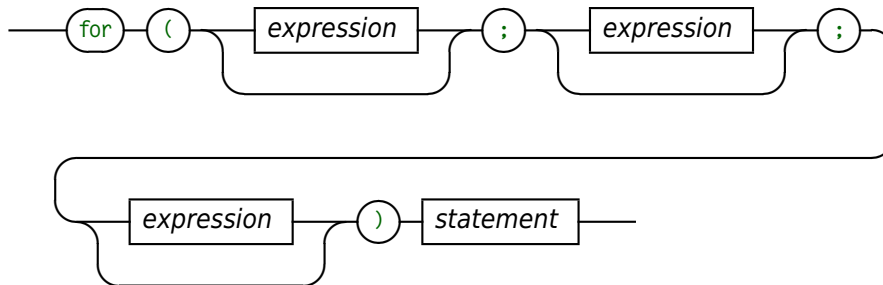
`compound statement`



`do while` loop



for loop

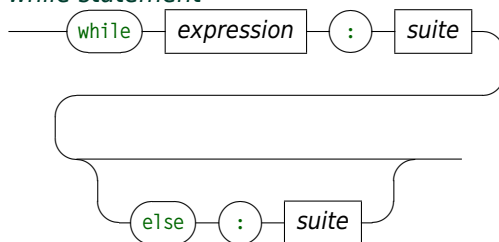


- Dieses Skript enthält im Anhang B.2 Regeln für den vollständigen Syntax von C in Syntaxdiagrammen

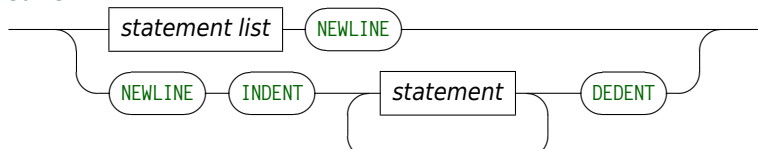
Beispiel Python-Schleifen

- Bei Python-Schleifen sieht man, dass für die lexikalische Analyse mitunter keine regulären Ausdrücke ausreichen
- In Python ist die Einrückung und das Zeilenende lexikalisches Elemente
- Abhilfe: Einführung von Token für Einrückungsänderung: „INDENT“ und „DEDENT“

while statement



suite



Formale Semantik

Merke

Während wir die Syntax (auch) formal beschrieben haben, geschah dies bei der Semantik stets informal.

Dies hat pragmatische Gründe: Während formale Syntax (Syntaxdiagramme, reguläre Ausdrücke) breite praktische Anwendungen haben, benötigt man formale Semantiken nur in engen Gebieten wie der (semi-)automatischen Programmverifikation.

Es gibt jedoch verschiedene Ansätze für formale Semantiken, z.B.

- **Denotationelle Semantik**
- **Axiomatische Semantik**
- **Operationelle Semantik**

Knobelaufgaben

Aufgabe 10.1

Fünf Häuser stehen an einer Straße nebeneinander. In ihnen wohnen Menschen von fünf unterschiedlichen Nationalitäten, die fünf unterschiedliche Getränke trinken, fünf unterschiedliche Zigarettenmarken rauchen und fünf unterschiedliche Haustiere haben.

1. Der Brite lebt im roten Haus.
2. Der Schwede hält sich einen Hund.
3. Der Däne trinkt gern Tee.
4. Das grüne Haus steht (direkt) links neben dem weißen Haus.
5. Der Besitzer des grünen Hauses trinkt Kaffee.
6. Die Person, die Pall Mall raucht, hat einen Vogel.
7. Der Mann im mittleren Haus trinkt Milch.
8. Der Bewohner des gelben Hauses raucht Dunhill.
9. Der Norweger lebt im ersten Haus.
10. Der Marlboro-Raucher wohnt neben der Person mit der Katze.
11. Der Mann mit dem Pferd lebt neben der Person, die Dunhill raucht.
12. Der Winfield-Raucher trinkt gern Bier.
13. Der Norweger wohnt neben dem blauen Haus.
14. Der Deutsche raucht Rothmanns.
15. Der Marlboro-Raucher hat einen Nachbarn, der Wasser trinkt.

Wem gehört der Fisch?

Aufgabe 10.2

Beweisen Sie, dass für alle $n > 1$ gilt:

$$\frac{1}{n+1} + \frac{1}{n+2} + \cdots + \frac{1}{2n} > \frac{13}{24}$$

Kapitel 11

Analyse von Algorithmen

Beware of bugs in the above code; I have only proved it correct, not tried it.

— Donald Knuth

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

— Alan J. Perlis

11.1 Einführung

- Die Beispiele aus Kapitel 8 (Kino-Problem, Traveling-Salesman-Problem) haben gezeigt, dass
 - man leicht einem inkorrekten algorithmischen Ansatz folgt
 - es unterschiedlich effiziente korrekte Algorithmen gibt
- Wir werden in diesem Kapitel Algorithmen **qualitativ** und **quantitativ** analysieren
 - **Qualitativ** ➔ Ist der Algorithmus effektiv (=korrekt)?
 - **Quantitativ** ➔ Wie Effizient ist der Algorithmus?
- Wir haben im Kapitel 8 bereits Adhoc-Betrachtungen zur Korrektheit der algorithmischen Idee gemacht
- Wiederholung:
 - Überlegen Sie, ob die algorithmische Idee allgemein oder nur in einem Spezialfall funktioniert!
 - Gibt es Spezialfälle, in denen die Idee nicht funktioniert? Sind diese Fälle relevant?
 - Versuchen Sie, ein Gegenbeispiel zu finden
 - Extremfälle helfen
- Bei Adhoc-Betrachtungen oder der Implementierung können Fehler gemacht werden ➔ fertiger Algorithmus bzw. fertiges Programm soll korrekt sein
- Prinzipiell gibt es **zwei** Möglichkeiten:

– **Erschöpfendes Testen**

Ausschluss fehlerhaften Verhaltens einer Implementation des Algorithmus durch Ausführung (Test) **mit jeder möglichen Instanz der Eingabe**

– **Korrektheitsbeweis**

Nachweis korrekten Verhaltens durch Nutzung von **mathematischen** Methoden

Probleme

Merke

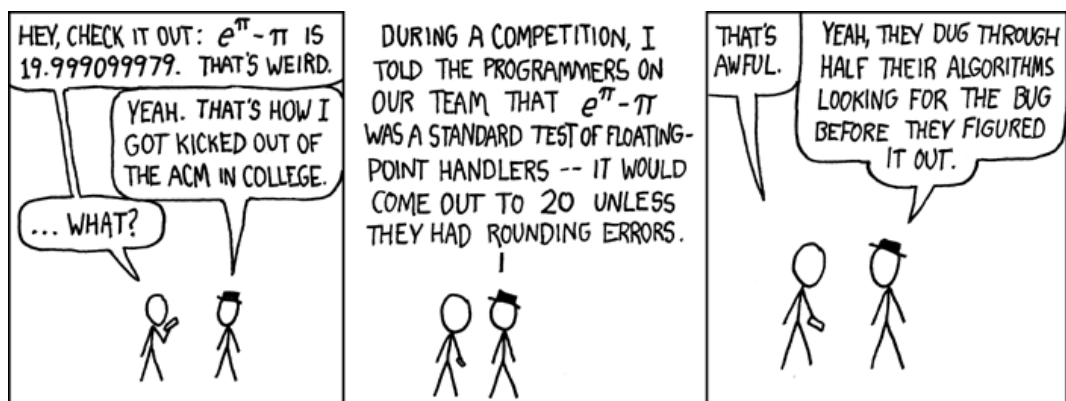
Erschöpfendes Testen ist in der Regel **unmöglich**, da der Eingaberaum meist sehr groß oder sogar unendlich ist.

- Durch **teilweises** Testen kann das Vertrauen in die Korrektheit eines Programmes gestärkt werden, es ersetzt aber keinen Nachweis
- Die gute Auswahl von Testfällen ist schwierig

Merke

Korrektheitsbeweise auf Implementationsebene sind i.d.R. wegen mangelnder Formalisierung/Formalisierbarkeit der Implementation **schwierig oder unmöglich**.

- Ersatzweise wird der Korrektheitsbeweis auf einer **abstrakteren Ebene** durchgeführt ➔ Fehler bei Umsetzung in konkrete Implementation möglich
- Abhilfe können (bis zu einem gewissen Maß) Tools, die selbst formal bewiesen sind, schaffen.
 - Allgemein ist dies aber nicht möglich ➔ **Gödelscher Unvollständigkeitssatz**
- Beschäftigen uns hier mit Beweisen (nicht Testen)



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/217/>

11.1.1 Beweise

- Was ist ein Beweis?

Beweis – Informale Definition

Ein Beweis ist eine Herleitung einer Aussage aus bereits bewiesenen Aussagen und/oder Grundannahmen (Axiomen).

- Wir kennen (aus der Mathematik¹ nur wenige (auch kombinierbare) Beweismethoden
 - Deduktion
 - Vollständige Fallunterscheidung (Aufzählung)
 - Vollständige/transfinite Induktion
 - Indirekte Beweise
- Betrachten diese Methoden kurz

Deduktion

- Deduktion ist der „klassische“ Beweis durch Kombination von Prämissen

Beispiel 11.1

- **Prämisse 1:** Alle Menschen sind sterblich
- **Prämisse 2:** Alle Könige sind Menschen
- **Folgerung:** Alle Könige sind sterblich

- Die Korrektheit der Prämissen muss entweder axiomatisch gegeben oder bereits bewiesen sein

Vollständige Fallunterscheidung

- Bei endlich vielen Varianten kann jede einzeln untersucht werden
- Ist eine Aussage für jede Variante wahr, ist sie insgesamt wahr

¹Klassenstufen 6+7, außer vollständige Induktion

Beispiel 11.2

- **Behauptung:** Alle ungeraden Zahlen im Intervall $[2^1, 2^3]$ sind Primzahlen.

$$\forall i = 2n + 1, n \in \mathbf{N}, i \in [2^1, 2^3], \text{prim}(i)$$

- **Fälle:** Im Intervall sind nur die Zahlen 3, 5 und 7 ungerade
 - 3 ist prim
 - 5 ist prim
 - 7 ist prim
- **Folgerung:** Behauptung stimmt

Anmerkung:

Die Operatoren $\forall$ und $\exists$ sind aussagelogische **Quantoren** (oder **Quantifikatoren**):

- $\forall$:
- **Bedeutung:** Aussage gilt für alle Elemente einer Menge
 - **Sprechweise:** „für alle“
 - **Beispiel:** $\forall x \in \{1, \dots, 4\}, \text{stat}(x) \Rightarrow$ „Für alle Werte x im Bereich von 1 bis 4 gilt Aussage $\text{stat}(x)$ “ $\forall x \in \{1, \dots, 4\}, \text{stat}(x)$ ist äquivalent zu $\text{stat}(1) \wedge \text{stat}(2) \wedge \text{stat}(3) \wedge \text{stat}(4)$
- $\exists$:
- **Bedeutung:** Es gibt ein Element in einer Menge, für Aussage gilt
 - **Sprechweise:** „es existiert“, „es gibt“
 - **Beispiel:** $\exists x \in \{1, \dots, 4\}, \text{stat}(x) \Rightarrow$ „Für mindestens ein x im Bereich von 1 bis 4 gilt Aussage $\text{stat}(x)$ “ $\exists x \in \{1, \dots, 4\}, \text{stat}(x)$ ist äquivalent zu $\text{stat}(1) \vee \text{stat}(2) \vee \text{stat}(3) \vee \text{stat}(4)$

Vollständige/transfinite Induktion

- **Induktionsanker** (auch: *Induktionsanfang*, *IA*) $\Rightarrow$ Zeigen, dass eine Aussage für einen speziellen (kleinsten) Fall (meist: $n = 0$ oder $n = 1$) gilt
- **Induktionsschritt** (*IS*)
 - **Induktionsvoraussetzung** (*IV*): Annahme, dass Aussage für ein $n = k$ gilt
 - **Induktionsbehauptung** (*IB*): Behauptung, dass Aussage für $n = k + 1$ gilt
 - Beweis, dass Induktionsbehauptung aus Induktionsvoraussetzung gilt
- **Induktionsschluss** $\Rightarrow$ Folgerung, dass Aussage für alle (unendlich viele) Fälle ab dem ersten Fall gilt

- Detaillierter in LV „Mathematik I“

Beispiel 11.3

- **Behauptung:** $\sum_{i=1}^n (2i - 1) = n^2 \Rightarrow$ Summe der ersten n ungeraden Zahlen ist n^2
- **IA:** $n = 1 \Rightarrow \sum_{i=1}^1 (2i - 1) = 2 - 1 = 1 \Rightarrow \text{w.A.}$
- **Induktionsschritt:** **IV:** $\sum_{i=1}^k (2i - 1) = k^2$, **IB:** $\sum_{i=1}^{k+1} (2i - 1) = (k + 1)^2$ **Beweis:**

$$\sum_{i=1}^{k+1} (2i - 1) = \sum_{i=1}^k (2i - 1) + (2(k + 1) - 1) = k^2 + 2k + 1 = (k + 1)^2$$
- **Induktionsschluss:** Behauptung gilt für alle $n \geq 1$

Indirekter Beweis

- Annahme des **Gegenteils** der Behauptung
- Unter Nutzung lediglich der Annahme, von Axiomen und bewiesenen Aussagen einen **Widerspruch** finden
- Folgerung dass Annahme falsch und folglich Behauptung wahr ist

Beispiel 11.4

- **Behauptung:** Es gibt keine größte Primzahl
- **Annahme A:** Es gibt eine größte Primzahl
- **Beweis:**
 - Aus A folgt, dass es nur endlich viele Primzahlen $p_1, \dots, p_n$ gibt (B)
 - Bilden $q = 1 + \prod_{i=1}^n p_i$
 - Fallunterscheidung:
 1. q ist prim $\Rightarrow q$ ist größer als $p_n \Rightarrow$ Widerspruch zu A
 2. q hat Primteiler $\Rightarrow$ diese sind nicht in $p_1, \dots, p_n \Rightarrow$ Widerspruch zu B und damit zu A
- **Schlussfolgerung:** Annahme A führt zum Widerspruch, folglich ist die Behauptung wahr

11.1.2 Beweis von Algorithmen

- Fragestellung ist ähnlich wie beim Entwurf
 - Was muss eigentlich gezeigt werden? $\Rightarrow$ verlangt Klarheit über Spezifikation
 - Was ist bekannt?
- **Beispiel Sortierung**
 - Was muss gezeigt werden:
 - $\Rightarrow$ Algorithmus sortiert **für jede gültige Eingabe** korrekt
 - Auf welchen Annahmen kann aufgebaut werden?
 - $\Rightarrow$ Jeder Einzelschritt (Pseudocode oder Programmiersprache) wird gemäß seiner Spezifikation ausgeführt
 - * Z.B. bewirkt $x \leftarrow x + 1$, dass Variable x um eins erhöht wird

Definition

- Man unterscheidet zwei Arten von „Korrektheit“

Definition 11.1: Partielle Korrektheit

Ein Algorithmus ist **partiell korrekt**, wenn er für eine spezifizierte erfüllte Vorbedingung (Q) bei einer **eventuellen Terminierung** eine spezifizierte Nachbedingung (R) erreicht; d.h. (R) ist nach Ausführung erfüllt.

Definition 11.2: Totale Korrektheit

Ein Algorithmus ist **total korrekt**, wenn er partiell korrekt ist und terminiert.

- Mit anderen Worten:
 - Ein partiell korrekter Algorithmus liefert das korrekte Ergebnis, *falls* er jemals fertig wird
 - Ein total korrekter Algorithmus liefert nach endlicher Zeit das korrekte Ergebnis

11.2 Partielle Korrektheit

Beispiel „Sortieren“

- In Kapitel 2 wurde im Beispiyalgorithmus 12 ein Sortieralgorithmus (Bubblesort) vorgestellt
- Die Korrektheit des Problems „Sortieren“ kann unabhängig vom Algorithmus ausgedrückt werden

Problem Sortieren

Eingabe: Folge von Elementen $(e_1, e_2, \dots, e_n)$ mit Ordnungsrelation („ $\leq$ “)

Ausgabe: Permutation $(e'_1, e'_2, \dots, e'_n)$ von $(e_1, e_2, \dots, e_n)$, so dass $e'_1 \leq e'_2 \leq \dots \leq e'_n$

- **Beispiel:** Sortieren einer Sequenz natürlicher Zahlen
 - **Eingabe:** 8, 15, 3, 14, 7, 6, 18, 19
 - **Ausgabe:** 3, 6, 7, 8, 14, 15, 18, 19
- Der Bubblesort-Algorithmus (BSORT) sieht in Pseudocode z.B. so aus:

Algorithm BSORT

Require: $e_1, \dots, e_n$

Ensure: $\forall i \in \{1, n-1\}, e_i \leq e_{i+1}$

```

1: repeat
2:   changed ← false;
3:   for i ← 1, ..., n-1 do
4:     if  $e_i > e_{i+1}$  then
5:       SWAP( $e_i, e_{i+1}$ )
6:       changed ← true
7:     end if
8:   end for
9: until changed = false
  
```

- Die partielle Korrektheit ist leicht zu beweisen:
 - Wenn BSORT terminiert, muss $changed = false$ gelten
 - $\leadsto$ für kein $i \in \{1, \dots, n-1\}$ darf $e_i > e_{i+1}$ gelten
 - $\leadsto \forall i \in \{1, \dots, n-1\}, e_i \leq e_{i+1}$ □
- Totale Korrektheit heben wir uns für später auf 😊

Alternative Problemlösung: Insertion Sort

- Bubblesort ist nur eine mögliche Lösung für das Sortierproblem
- Betrachten Alternative: „Insertion Sort“ (INSSORT)

Algorithm INSSORT

Require: $e_1, \dots, e_n$

Ensure: $\forall i \in \{1, n-1\}, e_i \leq e_{i+1}$

```

1: for  $j \leftarrow 2, \dots, n$  do
2:    $key \leftarrow e_j$ 
3:    $i \leftarrow j - 1$ 
4:   while  $(i > 0) \wedge (e_i > key)$  do
5:      $e_{i+1} \leftarrow e_i$ 
6:      $i \leftarrow i - 1$ 
7:   end while
8:    $e_{i+1} \leftarrow key$ 
9: end for
  
```

▷ Verschiebe alle Elemente $e_1, \dots, e_{j-1}$, die größer als key sind, nach rechts
 ▷ Fülle die Lücke mit key

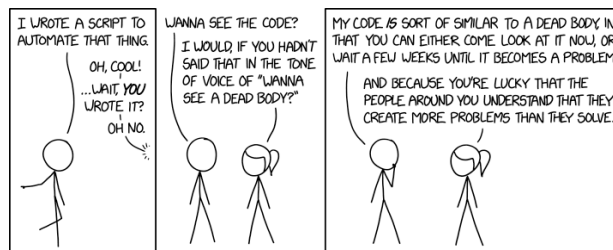
- Intuitiv ist der Algorithmus „Insertion Sort“ vermutlich leicht zu verstehen
- Versuchen uns an einem Beweis der Korrektheit
- Benutzen dazu einige Lemmata

Lemma 1. Wenn die Schleife 4 – 7, terminiert, ist entweder (a) $i = 0$ oder (b) $e_i \leq key$

• **Beweis:**

i startet mit einem Wert > 0 und wird schrittweise verkleinert, Rest folgt direkt aus der Schleifenbedingung

Lemma 2. Wenn die Elemente $e_1, \dots, e_{j-1}$ vor Eintritt in die Schleife 4 – 7 geordnet, so ist danach (a) die Sequenzen $e_1, \dots, e_i$ und (b) die Sequenz $e_{i+1}, \dots, e_j$ geordnet.



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/2138/>

- **Beweis:**

- (a) Die Sequenz $e_1, \dots, e_i$ wird nicht geändert

- (b) Fallunterscheidung:

- Die Schleife wird nicht betreten $\leadsto i = j - 1 \leadsto$ Sequenz $e_{i+1}, \dots, e_j$ besteht aus einem Element e_j , ist also geordnet
- Sonst: Entspricht der vorherigen Sequenz $e_i, \dots, e_{j-1}$, die ja geordnet war

Merke:

Ordnung ist eine **Invariante** der Schleife!

Lemma 3. Wenn vor dem Block 2 – 8 die Sequenz $e_1, \dots, e_{j-1}$ sortiert war, ist anschließend die Sequenz $e_1, \dots, e_j$ sortiert.

- **Beweis:** Fallunterscheidung, i nach Schleife:

- $i = 0$: Laut Schleifenbedingung $\forall k \in \{1, \dots, j-1\}, key < e_k$; außerdem laut Lemma 2 ist Sequenz $e_1, \dots, e_j$ sortiert
- $i > 0, i < j - 2$: Laut Lemma 1 ist $key \geq e_i$; außerdem laut Schleifenbedingung $\forall k \in \{i+2, j-1\}, key < e_k \leadsto e_1, \dots, e_i, e_{i+1} = key, \dots, e_j$ ist sortiert
- $i = j - 1$: Schleife nicht durchlaufen $\leadsto key \geq e_{i-1} \leadsto e_1, \dots, e_j$ ist sortiert

$key \geq e_i$ laut Lemma 1

Theorem 11.1

INSERTION SORT sortiert eine Sequenz.

- **Beweis:** Induktion über j :

- **IA:** $j = 1$ Einersequenz ist immer sortiert
- **IS:** Wenn $e_1, \dots, e_k$ sortiert ist, ist anschließend auch $e_1, \dots, e_{k+1}$ sortiert laut Lemma 3
- **Schlussfolgerung:** Theorem gilt für alle endlichen Sequenzen

Korrektheitskalküle

- Beweise, wie der angeführte Bubblesort-Beweis, sind *ad hoc*
- Es gibt spezielle Kalküle (**Kalkül** $\rightarrow$ formales System zum Ziehen logischer Schlüsse) für die Korrektheit von Programmen
- Beispiele
 - **Hoare-Kalkül** (auch: **Floyd-Hoare-Kalkül**)
 - **wp-Kalkül** (Edsger W.Dijkstra)

- Benutzung von Tripeln: $\{\text{Vorbedingung}\} \text{ Programmcode } \{\text{Nachbedingung}\}$
- **Axiomatische** Regeln: $\frac{\text{Prämisse}}{\text{Folgerung}}$
- Beispiel **Kompositionsregel**:
$$\frac{\{Q\} P1 \{R\} \quad \{R\} P2 \{S\}}{\{Q\} P1; P2 \{S\}}$$
- Mehr in LV „Verlässliche Systeme“

11.3 Terminierung

- **Partielle** Korrektheit wird bewiesen unter der Bedingung, dass der Code Terminiert
- Für **totale** Korrektheit muss also noch **Terminierung** bewiesen werden

Kritisch bei

- **Rekursion**
 - Die Rekursion muss nach einer endlichen Anzahl abbrechen, d.h. die Rekursionsbasis erreichen.
- **Schleifen**
 - Die Schleifenbedingung muss nach einer endlichen Anzahl von Iterationen *false* werden.
 - Es muss sichergestellt sein, dass auch der Schleifenrumpf terminiert (in jeder Iteration)

Terminierungsfunktion

- Zum Beweis der Terminierung einer Schleife muss eine **Terminierungsfunktion** τ angegeben werden:

$$\tau : V \rightarrow \mathbb{N}$$

- V ist eine Teilmenge der Ausdrücke über die Variablenwerte der Schleife
- Die Terminierungsfunktion muss folgende Eigenschaften besitzen:
 - Ihre Werte sind natürliche Zahlen (einschließlich 0)
 - Jede Ausführung des Schleifenrumpfs **verringert** ihren Wert (streng monoton fallend)
 - Die Schleifenbedingung ist *false*, wenn $\tau = 0$.
- ➔ τ ist obere Schranke für die noch ausstehende Anzahl von Schleifendurchläufen.

Terminierungsregel

- Ist eine Terminierungsfunktion bekannt, kann die **Terminierungsregel** genutzt werden

$$\frac{\{\tau = k\}P\{\tau < k\} \quad \{\tau = 0\} \Rightarrow \neg B \quad P \text{ terminiert}}{\text{while}(B) P; \quad \{\text{Terminierung}\}}$$

- Mit anderen Worten: **wenn** die Terminierungsfunktion streng monoton fallend ist, **und** (spätestens) ihr Wert 0 ein Ende der Schleife erzwingt, **und** der Schleifenkörper terminiert, **dann** terminiert eine Schleife

Es ist also zu zeigen:

- Streng monotonen Fallen von τ*
- Die Implikation der Nichterfüllung der Schleifenbedingung B bei niedrigsten τ*
- Die Terminierung des Rumpfs P*

Beispiel: Quadratzahl

```

1 /* Calculates the square of a nonnegative integer number a */
2
3 /* {Q: 0 ≤ a} */
4 y= 0;
5 x= 0;
6 while (y != a) {
7     x= x+2*y+1;
8     y= y+1;
9 }
10 /* {R: x = a²} */

```

- Wähle als Terminierungsfunktion $\tau = a - y$:
 - τ wird in jedem Durchgang dekrementiert, da y inkrementiert wird und a konstant bleibt.
 - Wenn $\tau = 0$ folgt $y = a$, d.h. die Schleifenbedingung $y \neq a$ ist falsch.
 - Schleifenrumpf enthält keine Schleifen/Rekursionen/Gotos... Terminierung trivial
- ➔ Die Schleife terminiert

Terminierung bei Rekursion

- Beim Beweis der Terminierung von Rekursionen kann genauso wie bei Schleifen vorgegangen werden
- Auch hier wird eine Terminierungsfunktion τ konstruiert, die mit zunehmender Rekursionstiefe kleiner wird.
- Es muss gelten:
 1. Die Werte von τ sind natürliche Zahlen (inkl. 0)
 2. Bei jedem Aufruf der Methode (rechte Seite der Rekursionsgleichung) wird der Wert von τ echt kleiner.
 3. Abbruch bei (spätestens) $\tau = 0$ erzwungen
- Beispiel: **Fibonacci-Zahlen**²

```

1 /* Computes Fibonacci n-th number */
2 int fib(int n)
3 {
4   if (n<3) return 1;
5   else return (fib(n-1)+fib(n-2));
6 }

```

- In diesem Fall können wir als Terminierungsfunktion $\tau = n$ wählen.

Probleme bei Terminierung

- Betrachten folgendes Beispiel:

```

Algorithm GOLDBACH


---


Require:  $x = 4$ 
Ensure:  $x > 4$ 

1: while  $x = \text{Summe zweier Primzahlen}$  do
2:    $x \leftarrow x + 2$ 
3: end while

```

- Terminiert diese Schleife?
- **Goldbachsche Vermutung:** Bis heute nicht allgemein bewiesen, aber bis 10^{18} geprüft

²Mehr zu Fibonacci-Zahlen im Kapitel 12.

Achtung!

Beweise für Terminierung sind **nicht** immer möglich!

- Diese Aussage ist wiederum bewiesen (Terminierungsproblem)

11.4 Komplexität

- Ein korrekter Algorithmus kann so ineffizient sein, dass er praktisch entweder überhaupt nicht, oder wenigstens für bestimmte Problemgrößen nicht durchführbar ist
- Beispiel: Traveling-Salesman-Problem (► Kapitel 8)
- **Effizienz** ► Ressourcenverbrauch
 - Rechenzeit
 - Speicherplatz
- Betrachten zunächst Rechenzeit

11.4.1 Messung der Rechenzeit

- **Ansatz:** Direktes Messen der Rechenzeit eines Programmes
 - Stoppuhr
 - Betriebssystemfunktionen (► `time`)
- **Problem:** Hier gibt es (zu) viele Einflüsse, die gemessen werden
 - Compiler
 - Rechnerkonfiguration
 - Rechnerlast
 - Betriebssystem

Merke:

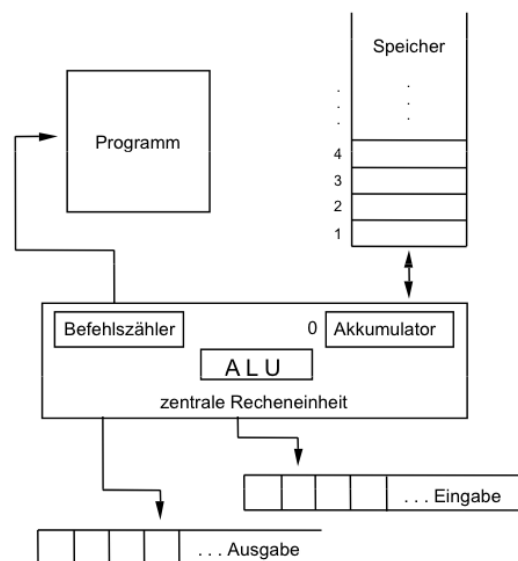
Zur Bestimmung der Qualität einer konkreten Implementation auf einem konkreten Rechner geeignet, aber nicht für allgemeine Aussagen über die Qualität von Algorithmen

- Rechner in Realität unterschiedlich schnell, **abhängig vom Instruktionsmix** eines Programmes
- **Beispiel:**
 - CPU1 führt alle Befehle in 2 Takten aus
 - CPU2 führt alle Integerbefehle in einem Takt aus, alle anderen in drei Takten

- Programm A bestehe aus 40% Integerbefehlen und 60% sonstigen Befehlen
- Programm B bestehe aus 90% Integerbefehlen und 10% sonstigen Befehlen
- Programm A ist schneller auf CPU1
- Programm B ist schneller auf CPU2
- **Erkenntnis:** Mit Laufzeitmessungen lassen sich Algorithmen nur schlecht vergleichen
- **Aber:**
 - Um die Qualität von Algorithmen (nicht Implementationen) zu beurteilen, ist eine genaue Zeit gar nicht nötig
 - Dazu nutzen wir **abstrakte Computermodelle**

11.4.2 Das RAM-Modell

- **Random Access Machine:** etwa Maschine mit freiem Zugriff
- Ähnlich von-Neumann-Modell, aber andere Motivation ➔ Abschätzung von Laufzeiten



Bestandteile der RAM

- **Programm:**
 - Nummerierte, endliche Folge von Befehlen
- **Speicher:**
 - abzählbar (unendlich) viele Speicherstellen (Register)
 - wahlfrei adressierbar

- jedes Register kann beliebige (ganze) Zahl speichern
- **Ein-/Ausgabe:**
 - Kontinuierliche Sequenzen (Bänder)
 - Jeweils nur Eingabe (Lesen) oder nur Ausgabe (Schreiben)
- **Zentrale Recheneinheit:**
 - Befehlszähler enthält Nummer des auszuführenden Befehls
 - Akkumulator: Zielregister von Berechnungen, Adresse 0
 - Arithmetic Logic Unit: Funktionseinheiten für die Ausführung von Operationen

Befehle und Kosten

- In einer RAM gibt es die „üblichen“ Befehle
 - Grundrechenarten: $+$, $-$, $\cdot$, $/$, mod
 - Vergleiche: $>$, $<$, $=$, $\geq$, $\leq$
 - Verzweigungen (IF)
 - Sprünge (GOTO) $\Rightarrow$ Schleifen sind Verzweigungen + Sprünge
 - Laden/Speichern (LOAD, STORE)
 - Ein-/Ausgabe (READ, WRITE)
- **Operanden:**
 - Register (wahlfrei), auch indirekt
 - Implizit Akkumulator
 - Eingabesequenz/Ausgabesequenz (nicht wahlfrei)

Zeitkosten

- Für die RAM gibt es zwei Zeitkostenmodelle:
 - **Uniforme Kostenmaß:** Jeder Befehl hat die Zeitkosten 1 Zeiteinheit³
 - **Logarithmische Zeitkosten:** Die Länge der bearbeiteten Zahlen bestimmen die Zeit
 - * Länge $l(x)$ von $x \in \mathbf{G}$: $l(0) = 1$, $l(x) = \lfloor \log_2 |x| \rfloor + 1$
 - * Die logarithmischen Zeitkosten eines Befehls sind gleich der Summe der Längen der bearbeiteten Zahlen

³Z.B. Takt oder Millisekunde

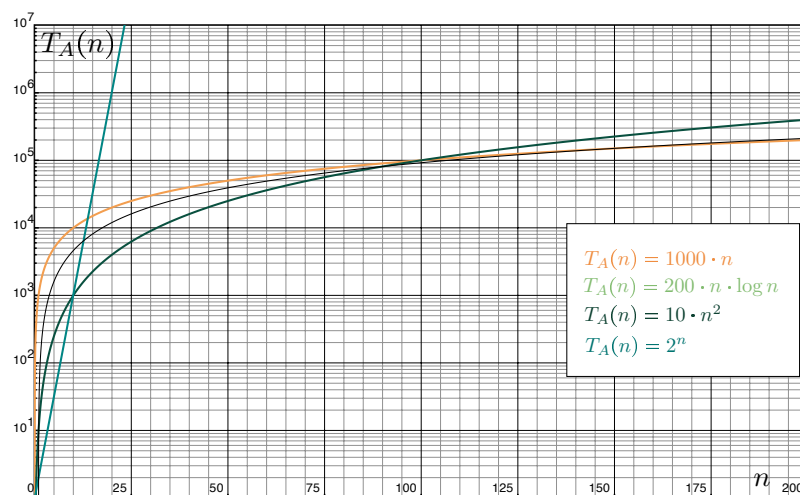
Anmerkung:

Das logarithmische Kostenmaß sollte immer dann verwendet werden, wenn die Größe der im Algorithmus vorkommenden Zahlen von entscheidender Bedeutung ist.

Beispiel: Primfaktorenzerlegung

- Da jeder Befehl (im uniformen Kostenmaß) die gleiche Länge hat, kommt es auf die Anzahl T_A von Befehls**ausführungen** an
- Im logarithmischen Modell muss zusätzlich die Größe der Operanden berücksichtigt werden
- Jeder Befehl, der in einer Schleife ausgeführt wird, zählt mehrmals
- Anzahl von Befehlsausführungen (und Schleifendurchläufen) häufig von der Eingabegröße n abhängig $\Rightarrow T_A(n)$
- Beispiel: Betrachte folgende Algorithmen zur Lösung eines Problems

Algorithmus	$T_A(n)$	optimal für
A_1	$1000 \cdot n$	$n \geq 101$
A_2	$200 \cdot n \cdot \log n$	nie
A_3	$10 \cdot n^2$	$10 \leq n \leq 100$
A_4	2^n	$1 \leq n \leq 9$



- Man beachte: $T_{A_4}(10) \approx T_{A_3}(10)$, aber $T_{A_4}(25) \approx 5400 \cdot T_{A_3}(25)$

Laufzeitanalyse

- Auch bei konstanten n ist T_A nicht immer gleich
- Betrachten:
- **Worst-Case-Analyse**
 - Für jedes n definiere Laufzeit $T(n) = \max(t(Input)), \forall |Input| = n$
 - Garantierte Schranke für jede Eingabe
 - Standard
- **Average-Case-Analyse**
 - Für jedes n definierte Laufzeit $T(n) = \bar{t}(Input), \forall |Input| = n$
 - Hängt von der Definition des Durchschnitts ab ➔ Verteilung der Eingaben
 - Seltener benutzt, trotz z.T. höherer praktischer Relevanz
- **Best-Case-Analyse**
 - Für jedes n definierte Laufzeit $T(n) = \min(t(Input)), \forall |Input| = n$
 - Minimale Laufzeit
 - Aufzeigen von Entwurfsfehlern

11.5 Die $\mathcal{O}$ -Notation

Merke

Da es relativ einfach ist, bei besten Fall zu mogeln, hat die Best-Case-Analyse praktisch keine Bedeutung.

Da es häufig sehr schwer ist, den Durchschnitt zu berechnen, wird es meist unterlassen.

- Allgemein ist die genaue Analyse von T_A sehr schwierig
- Deshalb: Konzept von **Größenordnungen**
- Bekannt aus dem täglichen Leben, z.B. für Geschwindigkeiten:

$$v_{Laufen} < v_{Rad} < v_{Auto} < v_{Flugzeug}$$

Die $\mathcal{A}$ -Notation

- Es ist interessant eine **maximale Schranke** anzugeben
- Definieren Menge $\mathcal{A}(x)$ von Funktionen/Werten, die **höchstens** den Wert x annehmen
- Beispiele:
 - $\pi = 3,14 + c, c \in \mathcal{A}(0,005)$ bedeutet, dass $\pi \approx 3,14$ ist
 - $2 \in \mathcal{A}(5)$, aber auch $4 \in \mathcal{A}(5)$
 - $\sin(x) \in \mathcal{A}(1)$
- Man schreibt häufig (etwas ungenau):
 - $\pi = 3,14 + \mathcal{A}(0,005)$
 - $2 = \mathcal{A}(5)$ bzw. $4 = \mathcal{A}(5)$
 - $\sin(x) = \mathcal{A}(1)$
- Das Gleichheitszeichen „ $=$ “ ist hier im Sinne von „ist“ zu verstehen und muss von links nach rechts gelesen werden
 - „Aristoteles ist ein Mensch“, aber ein Mensch ist nicht (notwendigerweise) Aristoteles
- Die $\mathcal{A}$ -Notation kann Konstanten wie Variablen oder Funktionen als Argument haben:

$$\begin{aligned}
 \sin(x) &= \mathcal{A}(1) \\
 x &= \mathcal{A}(1) \\
 \mathcal{A}(x) &= x \cdot \mathcal{A}(1) \\
 \mathcal{A}(x) + \mathcal{A}(y) &= \mathcal{A}(x + y) && \text{wenn } x \geq 0 \wedge y \geq 0 \\
 (1 + \mathcal{A}(t))^2 &= 1 + 3 \cdot \mathcal{A}(t) && \text{wenn } t = \mathcal{A}(1)
 \end{aligned}$$

Die $\mathcal{O}$ -Notation

- Häufig ist allerdings nicht interessant, wie groß etwas wird, sondern **wie schnell** es groß wird $\Rightarrow$ **Wachstum**

Definition 11.3: Wachstum

Zwei Funktionen $f(n)$ und $g(n)$ haben das gleiche Wachstumsverhalten, falls für genügend große n das Verhältnis der beiden nach oben und unten durch **Konstanten** beschränkt ist, d.h.

$$\exists c, d, n_0 \in \mathbf{N}, \forall n \geq n_0, \quad c < \frac{f(n)}{g(n)} < d, \quad c < \frac{g(n)}{f(n)} < d$$

- Warum die Forderung nur für „genügend große“ n ?

- ➔ Interessant sind Verfahren, die für große Probleminstanzen noch effizient sind
- Sprachregelung: „Sie skalieren gut.“

Beispiele

- $f_1(n) = n^2$ und $f_2(n) = 5 \cdot n^2 - 7 \cdot n$ haben das **gleiche** Wachstum:
 - Für alle $n > 2$ gilt: $\frac{1}{5} < \frac{(5n^2 - 7n)}{n^2} < 5$ und $\frac{1}{5} < \frac{n^2}{(5n^2 - 7n)} < 5$
- $f_1(n) = n^3$ und $f_2(n) = n^2$ haben **nicht** das gleiche Wachstum
 - Für alle hinreichend große n gilt: $\frac{n^3}{n^2} = n > c$

Mengen

- Ähnlich, wie die $\mathcal{A}$ -Notation eine Grenze angibt, definieren wir ein $\mathcal{O}$ -Notation zur Beschreibung des Wachstums
- Informal: $\mathcal{O}(x)$ entspricht $c \cdot \mathcal{A}(x)$ mit einer unbekannten Konstante c
- Formal:

Definition 11.4: $\mathcal{O}$ -Notation

$$\mathcal{O}(f(n)) = \{g(n) | \exists c > 0, \exists n_0 > 0, \forall n > n_0, g(n) \leq c \cdot f(n)\}$$

- Ähnlich wie bei $\mathcal{A}$ -Notation schreibt man
 - $5n^2 - 7n = \mathcal{O}(n^2)$ statt $5n^2 - 7n \in \mathcal{O}(n^2)$
- Die Schreibweise mit dem „großen O“ (ursprünglich war der griechische Buchstabe Omikron gemeint) wird als **Landau-Notation** bezeichnet

Dominierung

- Die $\mathcal{O}$ -Notation gibt an, welche Funktion das Wachstum **dominiert**

Definition 11.5: Dominierung

Bei zwei monotonen⁴ Funktionen $f(n)$ und $g(n)$ dominiert $f(n)$ die Funktion $g(n)$, wenn

$$g(n) \in \mathcal{O}(f(n))$$

gilt.

⁴Der allgemeine Fall ist etwas komplizierter

- Schreibweise: $\text{dom}(f(n), g(n))$ gibt die dominierende Funktion

- **Beispiele:**

- $\text{dom}(n^3, n^2) = n^3$
- $\text{dom}(2^n, n^k) = 2^n$ (für konstante $k > 1$)

Rechenregeln für $\mathcal{O}$ -Notation

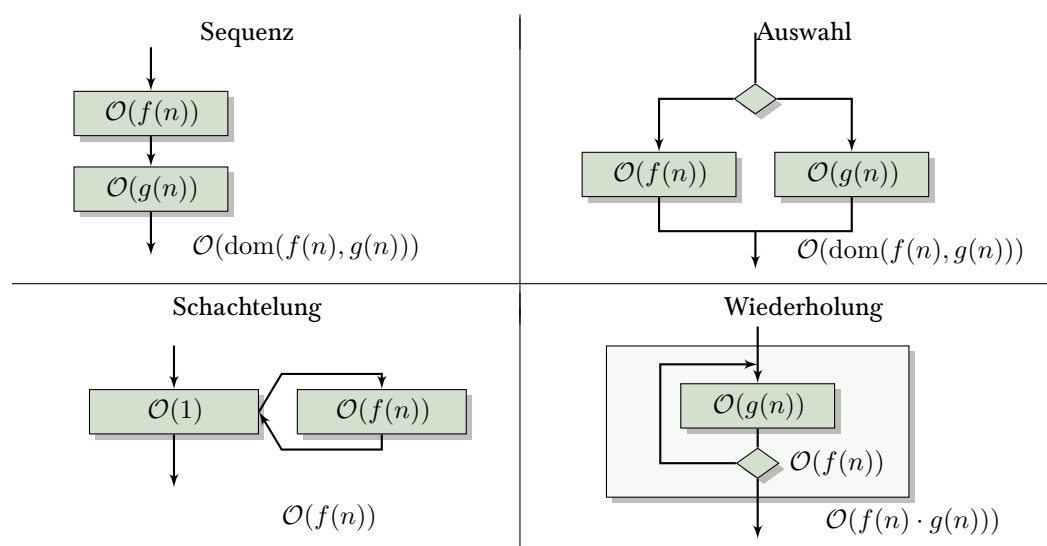
- Für Funktionen $f(n)$ (bzw. $g(n)$) mit der Eigenschaft $\exists n_0 > 0, \forall n > n_0, f(n) > 0$ gilt:

$$\begin{aligned}\mathcal{O}(f(n)) + \mathcal{O}(g(n)) &= \mathcal{O}(f(n) + g(n)) \\ \mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) &= \mathcal{O}(f(n) \cdot g(n)) \\ \mathcal{O}(f(n) + g(n)) &= \mathcal{O}(\text{dom}(f(n), g(n)))\end{aligned}$$

- Beispiel: $\mathcal{O}\left(\sum_{i=1}^n i\right) = \mathcal{O}\left(\frac{n^2+n}{2}\right) = \mathcal{O}(n^2)$

Achtung!

Bei Anwendung in Induktionen ist aufgrund des „unüblichen“ Gleichheitszeichen Vorsicht geboten!



Wichtige Klassen von Funktionen

	Sprechweise	Typische Algorithmen/Operationen
$\mathcal{O}(1)$	konstant	Addition, Vergleichsoperationen, rekursiver Aufruf, ...
$\mathcal{O}(\log n)$	logarithmisch	Suchen in einer sortierten Sequenz
$\mathcal{O}(n)$	linear	Bearbeiten jedes Elementes einer Menge
$\mathcal{O}(n \cdot \log n)$		gute Sortierverfahren
$\mathcal{O}(n \cdot \log^2 n)$		
$\vdots$		
$\mathcal{O}(n^2)$	quadratisch	primitive Sortierverfahren
$\mathcal{O}(n^k), k \geq 2$	polynomiell	
$\vdots$		
$\mathcal{O}(2^n)$		Ausprobieren von Kombinationen
$\mathcal{O}(k^n), k > 1$	exponentiell	

Eingabecodierung

- Die $\mathcal{O}(n)$ -Notation hängt von der Größe der Eingabe ab
- Was ist das?
- ➔ Speicherplatz in Bits oder Worten
- Vorsicht bei der Codierung!

Primfaktorenzerlegung

- **Gegeben:** $x \in \mathbb{N}$
- **Gesucht:** $\{p_i\}, \prod_i p_i = x$

Als schwieriges Problem bekannt ➔ Grundlage von RSA

- Trivialer Algorithmus für Primfaktorenzerlegung:

Require: $n \in \mathbb{N}$

Ensure: $P = \{p_i\}, x = \prod_i p_i$

```

1:  $P \leftarrow \emptyset$ 
2: for  $y \in [2, \lfloor \sqrt{x} \rfloor]$  do
3:   while  $x \bmod y = 0$  do
4:      $P \leftarrow P \cup \{y\}$ 
5:      $x \leftarrow \frac{x}{y}$ 
6:   end while
7: end for

```

▷ P is multiset

- **Binäre Kodierung** von x : Laufzeit exponentiell (bezüglich der Länge der Eingabe)
- **Unäre Kodierung** von x (x Einsen als Eingabe) Laufzeit linear

Merke:

Wird bei der $\mathcal{O}$ -Notation die Größe von Zahlen als Eingabegröße betrachtet, wird stets die „dichte“ binäre Kodierung angenommen.

Weitere asymptotische Maße

Neben der $\mathcal{O}$ -Notation, die angibt, wie eine Funktion „höchstens“ wächst, gibt es noch weitere Maße:

- $\Omega(f(n)) = \{g(n) | \exists c > 0, \exists n_0 > 0, \forall n > n_0, g(n) \geq c \cdot f(n)\}$
 $\Rightarrow$ „mindestens“
- $\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$
 $\Rightarrow$ „genau“
- $o(f(n)) = \{g(n) | \forall c > 0, \exists n_0 > 0, \forall n > n_0, g(n) \leq c \cdot f(n)\}$
 $\Rightarrow$ „weniger“
- $\omega(f(n)) = \{g(n) | \forall c > 0, \exists n_0 > 0, \forall n > n_0, g(n) \geq c \cdot f(n)\}$
 $\Rightarrow$ „mehr“
- **Beachte:** Es gibt kein $\theta(f(n))$, da $o(f(n)) \cap \omega(f(n)) = \emptyset$ gilt

Rechenzeiten

- Wie lange ist die minimale Rechenzeit, wenn ein einzelner Rechenschritt $1\mu\text{sec}$ dauert

Komplexität	10^1	10^2	10^3	10^4
$\Theta(\log_2 n)$	$4 \cdot 10^{-6} s$	$7 \cdot 10^{-6} s$	$10^{-5} s$	$1,4 \cdot 10^{-5} s$
$\Theta(n)$	$10^{-5} s$	$10^{-4} s$	$10^{-3} s$	$10^{-2} s$
$\Theta(n \log_2 n)$	$4 \cdot 10^{-5} s$	$7 \cdot 10^{-4} s$	$10^{-2} s$	$0,14 s$
$\Theta(n^2)$	$10^{-4} s$	$10^{-2} s$	$1 s$	$1,8 \text{ min}$
$\Theta(2^n)$	$10^{-3} s$	$0,4 \cdot 10^{17} \text{ Jahre}$	$> 10^{80} \text{ Jahre}$	$> 10^{80} \text{ Jahre}$
$\Theta(n!)$	$3 s$	$> 10^{80} \text{ Jahre}$	$> 10^{80} \text{ Jahre}$	$> 10^{80} \text{ Jahre}$

Beispiel: Minimumsuche**Algorithm** SEEKMINIMUM**Require:** Sequenz $a_1, \dots, a_n$ **Ensure:** $p = \min(a_1, \dots, a_n)$ 1: $p \leftarrow a_1$ 2: **for** $i \in \{2, \dots, n\}$ **do**3: **if** $a_i < p$ **then**4: $p \leftarrow a_i$ 5: **end if**6: **end for**▷ $\mathcal{O}(1)$ ▷ $\mathcal{O}(\sum_{i=2}^n (1 + T(L)))$ ▷ $\mathcal{O}(1)$ ▷ $\mathcal{O}(1)$

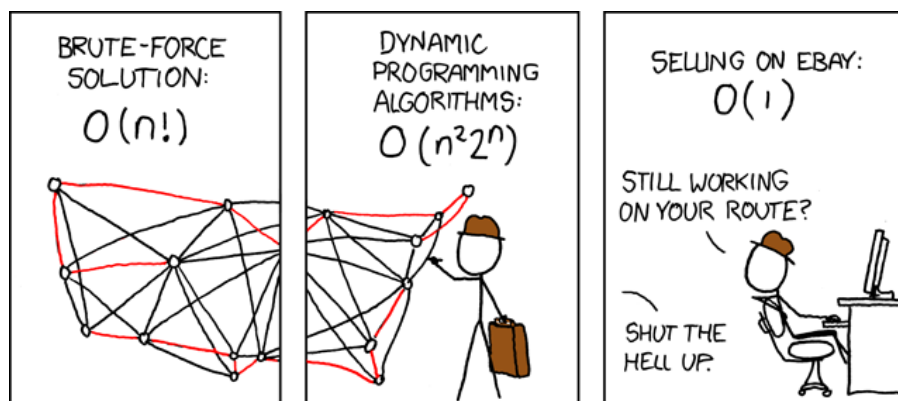
- Folglich gilt:

$$\mathcal{O}\left(1 + \left(\sum_{i=2}^n (1 + 1)\right)\right) = \mathcal{O}(n)$$

- Da Schleifendurchlauf fest, auch:

$$\Omega\left(1 + \left(\sum_{i=2}^n (1 + 1)\right)\right) = \Omega(n)$$

- ... und damit $\text{SEEKMINIMUM} \in \Theta(n)$



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/399/>

Beispiel: Bubblesort**Algorithm** BSORT**Require:** $e_1, \dots, e_n$ **Ensure:** $\forall i \in \{1, n-1\}, e_i \leq e_{i+1}$

```

1: repeat
2:    $changed \leftarrow \text{false};$ 
3:   for  $i \leftarrow 1, \dots, n-1$  do
4:     if  $e_i > e_{i+1}$  then
5:        $\text{SWAP}(e_i, e_{i+1})$ 
6:        $changed \leftarrow \text{true}$ 
7:     end if
8:   end for
9: until  $changed = \text{false}$ 

```

- Wieviel Schleifendurchläufe bei repeat-until-Schleife?
- Immer Worst-Case-Fall betrachten:
 - Nach einem for-Schleifen-Durchlauf ist **mindestens** ein Element mehr geordnet
 - Worst-Case: Es ist **nur** ein Element mehr geordnet
 - Folgerung: $\mathcal{O}(n)$
- **Gesamt:** $\mathcal{O}\left(\mathcal{O}(n) \cdot \sum_{i=1}^{n-1} (\mathcal{O}(1))\right) = \mathcal{O}(n^2)$

Rekursion

- Betrachten rekursiven Algorithmus: Fakultät

Require: $n \in \mathbb{N}$ **Ensure:** $n!$

```

1: procedure FAK( $n$ )
2:   if  $n = 1$  then                                ▷  $\mathcal{O}(1)$ 
3:     return 1                                       ▷  $\mathcal{O}(1)$ 
4:   else
5:     return  $n \cdot \text{FAK}(n-1)$                       ▷  $\mathcal{O}(1 + \dots?)$ 
6:   end if
7: end procedure

```

- $T(n)$: Laufzeit von Fakultät
- $T(1) = \mathcal{O}(1)$

- $T(n) = T(n-1) + \mathcal{O}(1)$
 $\Rightarrow T(n) = \mathcal{O}(n)$
- Mitunter ist die Komplexitätsbestimmung bei Rekursion wesentlich schwieriger $\Rightarrow$ Lösung von **Rekursionsgleichungen** notwendig

Durchschnittliche Laufzeit

- Laufzeiten müssen gewichtet werden
- Einfachster Ansatz: **Gleichverteilung** (uniforme Verteilung, alle Probleminstanzen $I \in \mathbf{I}_n$ gleichwahrscheinlich)

$$t(n) = \frac{1}{|\mathbf{I}_n|} \sum_{I \in \mathbf{I}_n} T(I)$$

- Tatsächliche Eingabeverteilung kann in der Praxis aber stark von uniformer Verteilung abweichen
- Dann Berücksichtigung der partiellen Wahrscheinlichkeiten

$$t(n) = \sum_{I \in \mathbf{I}_n} p_I \cdot T(I)$$

- Dabei muss $\sum_{I \in \mathbf{I}_n} p_I = 1$ gelten
- Probleme: Partielle Wahrscheinlichkeiten schwer zu bestimmen und zu berechnen

Erkenntnisse

- Die Komplexität kann ein Problem praktisch unlösbar machen
- Komplexität ist vor allem dann wichtig, wenn eine Lösung skalieren soll
- Durch geschickte Wahl eines Algorithmus kann unter Umständen die Komplexität reduziert werden
- Gleiches gilt für die Auswahl der benutzten Datenstrukturen
- (Viel) mehr davon in „**Datenstrukturen**“ und ggf. „**Theoretischer Informatik I**“

Knobelaufgaben**Aufgabe 11.1**

Beim Streichhölzchenspiel wird eine Anzahl von Streichhölzern auf den Tisch gelegt. Dann nehmen die beiden Spieler im Wechsel eine Anzahl von Streichhölzern weg, mindestens eines und maximal fünf. Der Spieler, der das letzte Steichholz nimmt (nehmen muss) hat verloren.

- Entwickeln Sie eine Strategie (Algorithmus), der Ihre Gewinnchance vergrößert oder sogar garantiert!
- Unter welchen Bedingungen (Anzahl Hölzer, erster oder zweiter Spieler) können Sie einen Sieg garantieren?
- Wie ändert sich der Algorithmus, wenn der Spieler mit dem letzten Hölzchen gewinnt?

Aufgabe 11.2

Betrachten Sie folgenden „Beweis“ und finden Sie den Fehler!

$$\begin{array}{ll}
 20 = 20 & | \cdot -1 \\
 -20 = -20 & \\
 16 - 36 = 25 - 45 & \\
 4 \cdot 4 - 4 \cdot 9 = 5 \cdot 5 - 5 \cdot 9 & \\
 4 \cdot 4 - 2 \cdot 4 \cdot \frac{9}{2} = 5 \cdot 5 - 2 \cdot 5 \cdot \frac{9}{2} & | x \stackrel{def}{=} \frac{9}{2} \\
 4^2 - 2 \cdot 4 \cdot x = 5^2 - 2 \cdot 5 \cdot x & | + x^2 \\
 4^2 - 2 \cdot 4 \cdot x + x^2 = 5^2 - 2 \cdot 5 \cdot x + x^2 & | (a-b)^2 = a^2 - 2ab + b^2 \\
 (4-x)^2 = (5-x)^2 & | \sqrt{} \\
 4-x = 5-x & | + x \\
 4 = 5 & \square
 \end{array}$$

Kapitel 12

Von Türmen und Damen - Rekursion und Backtracking

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

— Douglas R. Hofstadter

[The] ability to understand pointers and recursion is directly correlated with the ability to be a great programmer.

— Joel Spolsky

An dieser Stelle sei darauf hingewiesen, dass Sie nun das *prinzipielle* Handwerkzeug besitzen, um Algorithmen zu entwerfen, diese zu überprüfen, ihre Effizienz zu bewerten und sie mit Hilfe der Programmiersprache C in ausführbare Programme umzusetzen.

Achtung!

Damit diese Fähigkeiten sich bei Ihnen auch **aktiv** entwickeln, sollten Sie möglichst viel **praktisch** üben. Nutzen Sie die Computerpools oder ihre privaten Rechner so viel wie möglich, um eigene Programme zu programmieren.

Im Rest dieses Semesters werden diese Fähigkeiten vertieft und ergänzt, indem Beispiele und häufig genutzte Ansätze betrachtet werden.

Zwei dieser Ansätze, die zwar eng verwandt aber doch unabhängig sind, werden in diesem Kapitel besprochen: Die schon bekannte Rekursion, die erweitert und unter Einbeziehung technischer Aspekte diskutiert wird, sowie der Ansatz des Backtrackings (genauer: der Suche mit Backtracking), das in der Regel mit Hilfe von Rekursion implementiert wird.

12.1 Einführung

Wiederholung

- **rekursiv**: (*aus lat.*) selbstbezüglich

- **Rekursion:** wenn noch nicht verstanden, siehe ➔ **Rekursion**

- **Beispiele:**

- Rekursive Definition:
Liste := leere_Liste | Element, Liste
- Rekursive Funktionen:

```

1  int ggT(int x, int y)
2  {
3      if (y==0) return x;
4      else return ggT(y, x%y);
5  }

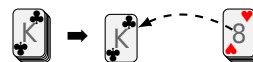
```

Stack

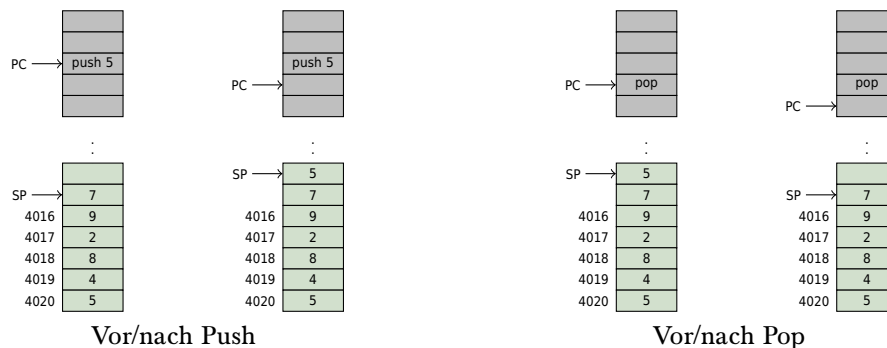
- Zur Diskussion über den Unterschied zwischen **Rekursion** und **Schleifen** betrachten wir die Datenstruktur **Stack**, die in vielen CPUs genutzt wird
- Der **Stack** ist ein Datencontainer, bei dem zwei Operationen erlaubt sind:
 - $S \leftarrow \text{Push}(D, S)$: Ein Datenelement D wird dem Stack hinzugefügt und verdeckt den Zugriff auf alle anderen Daten



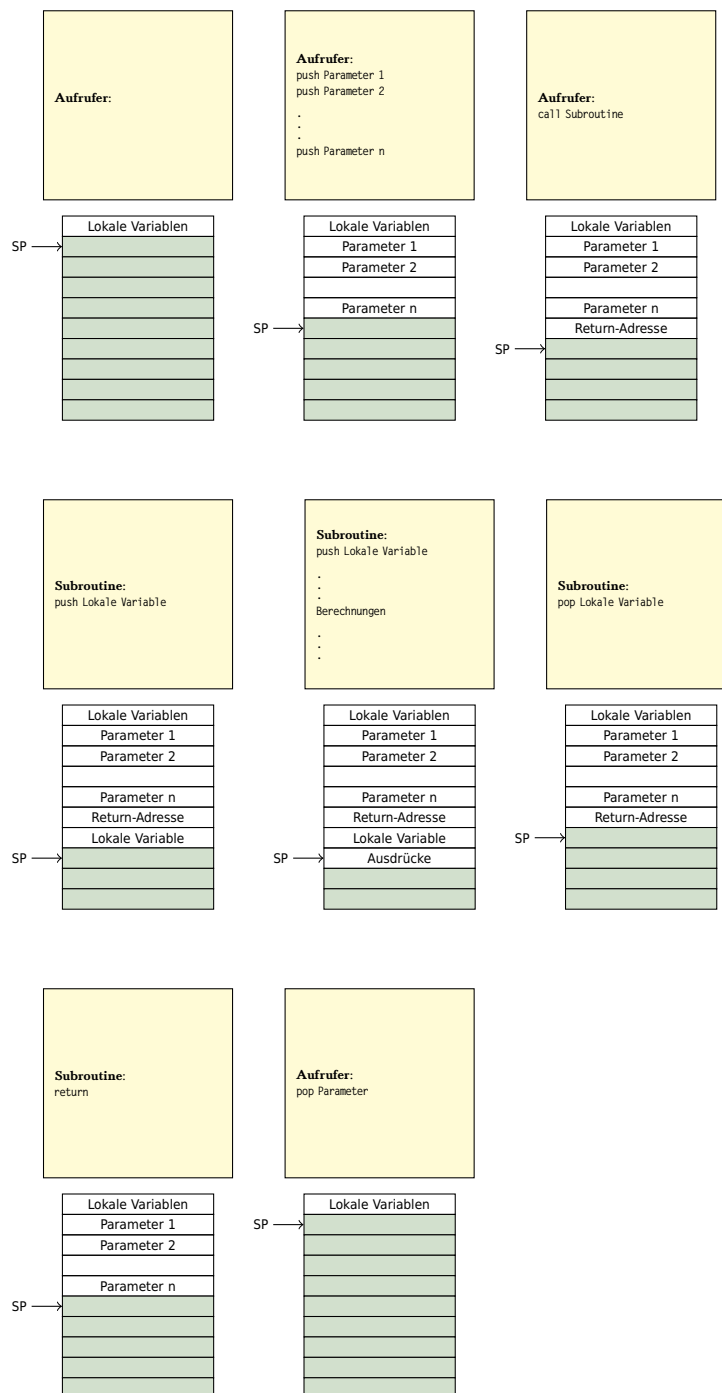
- $D \leftarrow \text{Pop}(S)$: Das (einzig sichtbare) Datenelement wird entfernt, eröffnet den Zugriff auf das zuletzt verdeckte



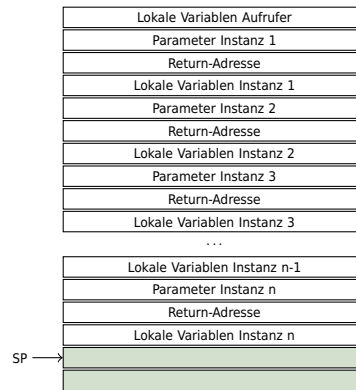
- Einfache Implementierung in Hardware ➔ **Stackpointer**



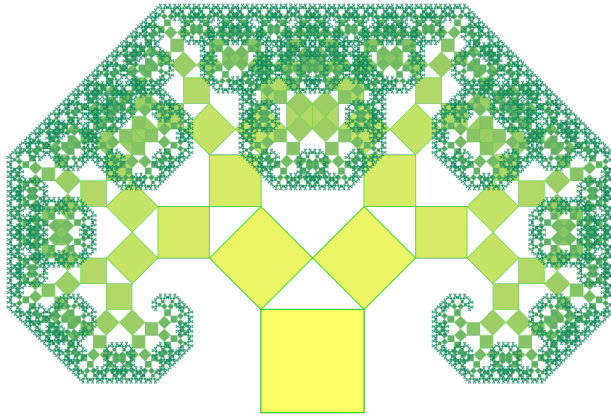
Unterprogrammaufruf



Stack bei Rekursion



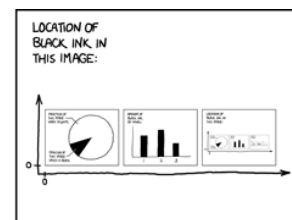
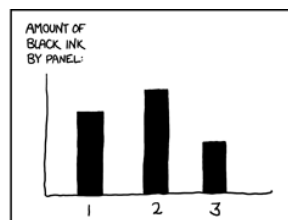
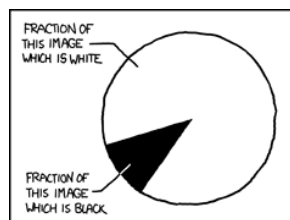
Rekursive Geometrie



Quelle: Guillaume Jacqueton Gjaqueton, Wikimedia Commons, CC-Lizenz

Pythagoräischer Baum

- Quadrat über zwei Punkten
- Oberseite des Quadrats bildet Grundseite eines Dreiecks, das mit einer gegebenen Höhe gezeichnet wird
- Eckpunkte der beiden Schenkel des Dreiecks bilden nun die Punkte für neue Quadrate
- **Terminierung** nach vorgegebener Rekursionstiefe



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/688/>

Rekursion vs. Iteration

Rekursion	Iteration
rekursiver Aufruf eines Unterprogramms (UP)	Durchführen der Anweisungsfolge eines Iterationsschrittes
Parameter des UPs werden im UP modifiziert und auf Erreichen einer Abbruchbedingung getestet	Steuerung der Anzahl Iterationsschritte meist mit Zählvariablen , einer Menge oder speziellen Iteratoren
Speicherung eines verarbeiteten Elements erfolgt implizit durch automatische Sicherung der UP-Parameter auf dem Stack	zu verarbeitendes Element bzw. eine Elementefolge muss explizit gespeichert werden (z.B. in einem Feld)
Zugriff auf Elemente einer Folge ist begrenzt auf die aktuelle Komponente der gerade aktiven UP-Instanz	Zugriff auf Elemente einer Folge kann wahlfrei erfolgen
entspricht Funktionenmodell	entspricht Zustandsmodell

Entwurf rekursiver Algorithmen

In jedem rekursiven Algorithmus müssen stets (mindestens) zwei Fälle behandelt werden:

1. Ende (Abbruch) der Rekursion
2. Rekursiver Aufruf

Prinzipiell gibt es zwei Ansätze, um rekursive Algorithmen zu entwerfen

1. Abstraktion:

- Schreiben von Speziallösungen für spezielle Probleminstanzen
- Schreiben weiterer Speziallösungen unter Nutzung vorhandener Lösungen
- Erkennen von Gemeinsamkeiten
- Zusammenfassung (Abstraktion) zu einer allgemeinen Lösung

2. Glaubenssprung (*Leap of faith*)

- Annahme, dass die Funktion/Prozedur bereits funktionieren würde
- Nutzen der Funktion für ein kleineres Teilproblem

12.2 Fibonacci-Zahlen

- Die **Fibonacci-Zahlen** sind eine unendliche Zahlenreihe mit einer Reihe interessanter Eigenschaften
- Eine Bildungsfunktion für die Reihe wird häufig als Beispiel für eine rekursive Funktion angegeben:

```

2  /* Computes Fibonacci n-th number */
3  long fib(long n)
4  {
5      if (n==0) return 0;
6      if (n<3) return 1;
7      else return (fib(n-1)+fib(n-2));
8  }

```

- Ursprüngliches Problem (aus Fibonacci Buch „liber abaci“):

Ein Mann setzt ein Kaninchenpaar in ein Gehege. Diese Kaninchen bekommen jeden Monat ein weiteres Paar. Ein eben geborenes Paar wird im zweiten Monat fruchtbar. Wie viele Kaninchenpaare gibt es nach einem Jahr in diesem Gehege?

- **Regelmäßigkeit**

- Wir erkennen, dass sich die Anzahl der Kaninchen eines Monats aus der Summe der Anzahl der Kaninchen der beiden vorherigen Monate ergibt

- **Fibonacci-Folge:**

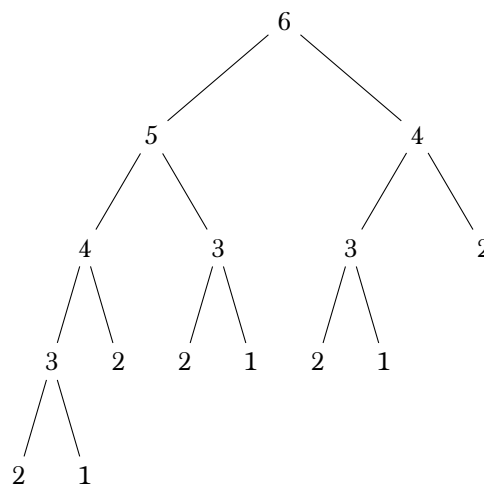
- Jede Folge, deren Glieder sich aus der Summe der beiden vorhergehenden Glieder ergibt, d.h. $a_{n+2} = a_n + a_{n+1}$

- **Standardfolge**

- Wir legen a_0 mit 0 und a_1 mit 1 fest und erhalten dann folgende Standardfolge:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

Aufrufgraph für $\text{fib}(6)$



Iterative Fibonacci-Funktion

- Die Fibonacci-Funktion ist ein Beispiel, dass Rekursion in der Implementation **nicht** eingesetzt werden sollte
- Iterative Version:

```

2 // Fibonacci numbers -- version 2
3 long fib(long n)
4 {
5     long *fnum = malloc((n+1)*sizeof(long));
6     long res;
7     if (n==0) return 0;
8     fnum[0]=0;
9     fnum[1]=1;
10    for(int i=2; i<=n; ++i)
11    {
12        fnum[i]=fnum[i-1]+fnum[i-2];
13    }
14    res = fnum[n];
15    free(fnum);
16    return res;
17 }

```

- Komplexität der Rekursion: $\Theta(c^n)$ ($c > 1$ 😊)
- Komplexität der Iteration: $\Theta(n)$

Explizite Berechnung

- Berechnung der Fibonacci-Zahlen ist sogar explizit möglich, d.h. ohne rekursive oder iterative Berechnung vorheriger Fibonacci-Zahlen
- Formel von Binet
- Nutzt Goldenen Schnitt $\Phi = \frac{\sqrt{5}+1}{2}$
- n -te Fibonacci-Zahl:

$$fib(n) = \frac{\Phi^n - \left(-\frac{1}{\Phi}\right)^n}{\sqrt{5}} = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right)$$

Achtung!

Bei Gleitkommaarithmetik kann es bei großen Werten zu Rechenungenauigkeiten kommen!

Hier treten solche (je nach Mathebibliothek) ab etwa $n = 70$ auf.

```

3  #include<math.h>
16 /* Fibonacci numbers -- version 3 */
17 long fib(long n)
18 {
19     const long double phi=(sqrtl(5)+1.0)*.5;
20     const long double iphi=(1.0-sqrtl(5))*0.5;
21     long double res;
22     res=(powl(phi,(double)n)-powl(iphi,n))/
23         sqrtl(5);
24     return lroundl(res);
25 }

```

```

> ./fib2 85
fib(85) = 259695496911122585
> ./fib3 85
fib(85) = 259695496911122586

```

12.3 Türme von Benares

- Rekursion kann häufig benutzt werden, um **prinzipielle** Lösungen zu demonstrieren
- Betrachten Problem, dessen Instanzen gut rekursiv abgebildet werden können
- Bekannt als „Türme von Hanoi“, ursprünglich aber „Türme von Benares“

Legende der Türme von Benares

- François Édouard Anatole Lucas erzählte 1883 folgende Geschichte:

Im großen Tempel von Benares¹, unter dem Dom, der die Mitte der Welt markiert, ruht eine Messingplatte, in der drei Diamantnadeln befestigt sind, jede eine Elle hoch und so dick wie der Körper einer Biene.

Brahma² steckte bei der Schöpfung auf eine der Nadeln einen Turm aus 64 Scheiben reinsten Goldes, wobei die größte auf der Messingplatte ruht, und die übrigen, immer kleiner werdend, eine auf der anderen.

Die Priester des Tempels haben nun die Aufgabe erhalten, den Turm unter Beachtung heiliger

Regeln auf eine andere der Nadeln zu bewegen:

1. Die Ringe dürfen stets nur von einer Nadel auf eine andere gelegt werden.
2. Es darf zur gleichen Zeit immer nur eine der Scheiben bewegt werden.
3. Es darf zu keiner Zeit eine Scheibe auf einer kleineren Scheibe liegen.

Wenn auf der anderen Nade der Turm aus 64 Scheiben errichtet ist, wird Shiva² die Nadeln mit den Scheiben und dem Tempel zu Staub zerfallen lassen und die gesamte Welt wird untergehen.

- Bekannt wurde das Spiel unter dem Namen „Türme von Hanoi“



Vereinfachte Version als Spiel

- **Problembeschreibung:**

- Ein Turm aus n beweglichen Teilen soll von Position A nach Position C bewegt werden

- **Bedingungen:**

- Die Teile des Turms werden nach oben hin immer kleiner und bei der Umschichtung darf kein größeres auf einem kleineren Teil liegen
- Es steht noch eine Zwischenablagemöglichkeit an Position B zur Verfügung

- **Nebenfrage:**

- Wie viele Schritte (Zeit) benötigt man für die Umschichtung?

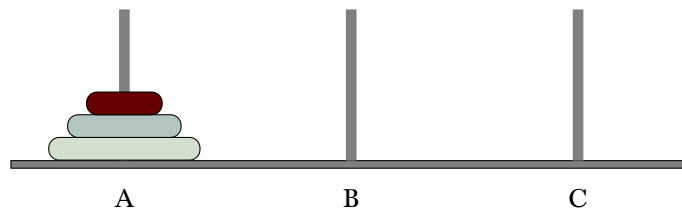
¹indische Stadt, früher Kashi, heute Varanasi

²hinduistischer Schöpfergott

³hinduistischer Gott der Zerstörung

Problem mit $n = 3$

- Zeigen Funktionsprinzip für $n = 3$ „per Hand“ 😊

**Induktion**

- Wir haben gezeigt, dass wir für $n = 3$ Scheiben den gesamten Turm von A (via B) nach C schieben können
- Sei nun $n = 4$
- Wir wissen, dass ein Turm der Höhe n umgeschichtet werden kann, deshalb verschieben wir
 1. Den (oberen) Teilturm der Höhe $n - 1$ von A nach B (via C)
 2. Die unterste Scheibe von A nach C
 3. Den Teilturm der Höhe $n - 2$ von B nach C (via A)
- Analog für $n = 5$, $n = 6$ usw.

Algorithmus**Algorithm** TOWERS OF HANOI

Require: Tower with height n at place A

Ensure: Tower at place C $\wedge$ rules always hold

MOVE-TOWER(n , C)

procedure MOVE-TOWER($height$, $goal$)

if $height = 0$ **then**

return

end if

 MOVE-TOWER($height - 1$, aux_place)

 move last_ring to $goal$

 MOVE-TOWER($height - 1$, $goal$)

end procedure

1. Verschiebe Teilturm der Höhe $n - 1$ vom Start zum Hilfsplatz (via Ziel)

2. Lege letzte Scheibe vom Start zum Ziel
3. Verschiebe Teilturm der Höhe $n - 1$ vom Hilfsplatz zum Ziel (via Start)

Anzahl der Umschichtungen

• Induktion

- Haben für $n = 3$ sieben Scheiben verschoben ($= 2^n - 1$)
 - Sei $n = m, m \in \mathbb{N}, n > 3$:
 1. Teilturm hat Höhe $n - 1$, also $2^{n-1} - 1$ Bewegungen
 2. Restscheibe wird direkt vom Ausgangs- zum Zielpunkt verschoben (1 Schritt)
 3. Teilturm hat Höhe $n - 1$, d.h. wieder $2^{n-1} - 1$
- Insgesamt also $2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$

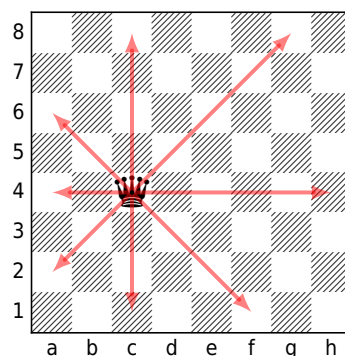
• Türme von Benares

- Da die Höhe des Turms $n = 64$ beträgt, müssen $2^{64} - 1$ mal Scheiben umgeschichtet werden
- Selbst bei einer Sekunde pro Scheibenzug wären das ca. 584.942.410.000 Jahre ➔ Ende der Welt

12.4 Backtracking

Damenproblem

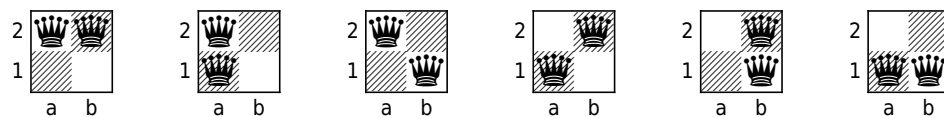
- Eine Dame kann auf einem Schachbrett alle Felder erreichen, die zur gleichen Reihe, Spalte oder Diagonalen gehören wie das Feld auf dem sie steht:



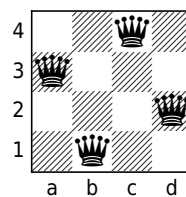
- Gesucht ist eine Anordnung von n Damen, so dass jeweils keine von ihnen eine andere Dame bedroht (schlagen könnte)
 - Farbe der Figuren werden ignoriert.

Problembeschreibung

- Zwei Parameter
 - n : Feldgröße $\Rightarrow n \times n$ -Feld
 - m : Anzahl der Damen
- Offensichtlich: Für $n < m$ nicht lösbar
- $n = 2$ und $m = 2 \Rightarrow$ unlösbar



- Beispiel 2: Das 4-4-Damenproblem ist lösbar



- Beschränken uns auf jeweils schwierigste Probleminstanz $m = n$
- Bekanntester Fall: 8-Damenproblem (auf „echtem“ Schachbrett)

Lösungsidee

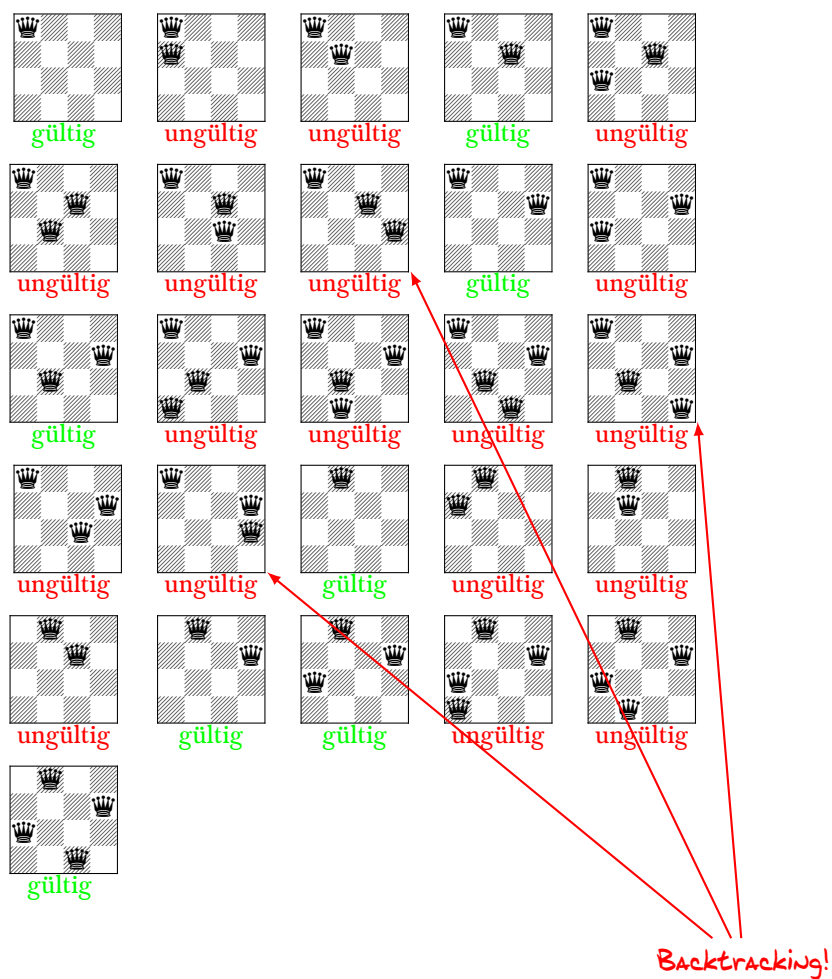
1. Starten mit einer Dame in der ersten Zeile und erste Spalte
2. Setze die nächste Dame in die nächste Zeile in die erste Spalte
3. Ziehe die zweite Dame so lange nach rechts, bis eine erlaubte Situation eintritt (kein gegenseitiger Angriff)
4. Wiederhole ab Schritt 2, bis
 - entweder alle acht Damen gesetzt sind $\Rightarrow$ Lösung gefunden
 - für eine Dame kein erlaubtes Feld in „ihrer“ Zeile gefunden werden kann

- In diesem Fall wird das letzte **gültig gesetzte** Dame als **ungültig behandelt** und sie weiter nach rechts gerückt
- Falls sie auf dem letzten Feld steht, wird mit der vorherigen Dame fortgefahren etc.

Backtracking

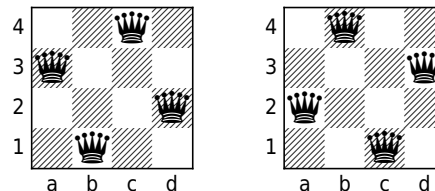
- Diesen Ansatz, eine bereits gefundene Teillösung zu verwerfen und zu bei einer vorherigen Teillösung weiter zu machen, bezeichnet man als **Backtracking**
- Im deutschen Sprachraum wird mitunter vom **Rücksetzverfahren** gesprochen
- Muss das Backtracking bis **vor** die erste Dame durchgeführt werden, gibt es **keine Lösung**
- Backtracking wird häufig zusammen mit Rekursion verwendet, ist aber ein davon unabhängiges Prinzip

Beispiel - Schritt für Schritt



Weitere Lösungen

- Neben der hier gefundenen Lösung gibt es noch (eine) andere (schon aus Symmetriegründen)



- Falls weitere Lösungen (oder alle) gefragt:
 - Gefundene Lösungen speichern
 - Backtracking auslösen
- Betrachten jetzt Algorithmus 8-Damen-Problem
 - Suche für einzelne Dame per Schleife
 - Rekursion für jede neue Dame

Algorithmus

Algorithm 8-QUEEN

Ensure: L : list of set queens

$L \leftarrow \text{PLACE-NEXT-QUEEN}([], 1)$

▷ empty list

procedure PLACE-NEXT-QUEEN(L, row)

if $\text{num} = 9$ **then**

return L

end if

$q.\text{row} \leftarrow \text{row}$

for $q.\text{col} \leftarrow \{1, \dots, 8\}$ **do**

if $\forall i = \{1, \dots, \text{length}(L)\}, \text{do\_not\_attac}(q, L_i)$ **then**

$R \leftarrow \text{PLACE-NEXT-QUEEN}(L + q, \text{num} + 1)$

▷ recursion

if $R \neq []$ **then**

return R

end if

end if

end for

return $[]$

end procedure

Implementation in C

```

1 #include <stdbool.h>
2 #include <stdio.h>
3
4
5 enum {CSIZE=8};
6 typedef struct { int row, col; } position;
7
8 bool place_queen(position *list, int already_placed);
9 bool is_valid(position *list, int count);
10 bool is_attacked(position q1, position q2);
11 void draw_board(position list[CSIZE]);
12 void print_solution(position list[CSIZE]);
13
14 int main()
15 {
16     position queenslist[CSIZE];
17     if (place_queen(queenslist,0)){
18         print_solution(queenslist);
19         draw_board(queenslist);
20     } else {
21         printf("No solution found.\n");
22     }
23     return 0;
24 }
25
26 bool place_queen(position *list, int already_placed)
27 {
28     position newqueen;
29     /* we've already placed all queens => fine */
30     if (already_placed==CSIZE) return true;
31     /* each queen has its own row */
32     newqueen.row=already_placed;
33     /* try columns */
34     for(newqueen.col=0; newqueen.col<CSIZE; ++newqueen.col){
35         list[already_placed]=newqueen;
36         if (is_valid(list, already_placed) && /* no conflict */
37             place_queen(list, already_placed+1)) /* recursion! */
38             return true;
39     }
40     return false;
41 }

```

Validität feststellen

- Wie kann die Validität einer Situation festgestellt werden?

- ~~1. Idee:~~

- Array für Schachfeld als Datenstruktur mit markierter „Einflusssphäre“
- Test für neue Damen, ob ihre Einflusssphäre die existierende überschneidet

- **Problem:** Rücknahme von Einflusssphären beim Backtracking

- **2. Idee:** Paarweises Testen

- Zeilen müssen ungleich sein (durch Algorithmus gewährleistet) $\Rightarrow y_i \neq y_j$
- Spalten müssen ungleich sein $\Rightarrow x_i \neq x_j$
- Diagonalen müssen ungleich sein

/-Diagonalen

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8	9
3	3	4	5	6	7	8	9	10
4	4	5	6	7	8	9	10	11
5	5	6	7	8	9	10	11	12
6	6	7	8	9	10	11	12	13
7	7	8	9	10	11	12	13	14

$$x_i + y_i \neq x_j + y_j$$

\-Diagonalen

	0	1	2	3	4	5	6	7
0	7	6	5	4	3	2	1	0
1	8	7	6	5	4	3	2	1
2	9	8	7	6	5	4	3	2
3	10	9	8	7	6	5	4	3
4	11	10	9	8	7	6	5	4
5	12	11	10	9	8	7	6	5
6	13	12	11	10	9	8	7	6
7	14	13	12	11	10	9	8	7

$$y_i - x_i + 7 \neq y_j - x_j + 7$$

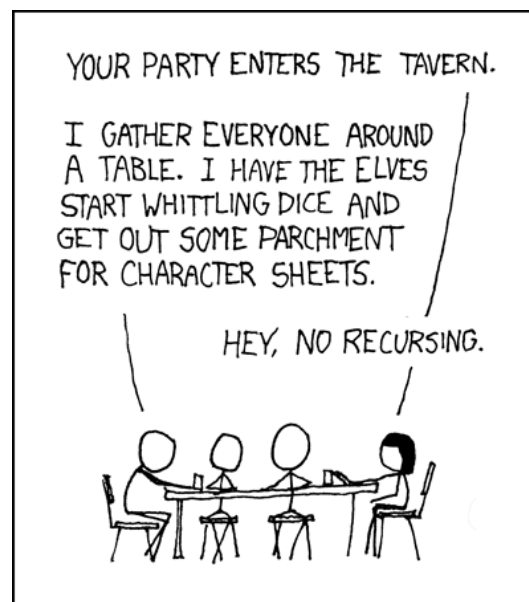
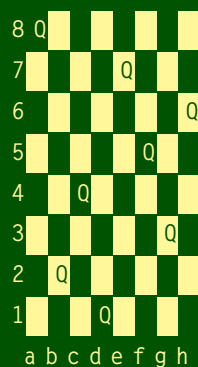
```

43 bool is_valid(position *list, int count)
44 {
45     int i;
46     for(i=0; i<count; ++i)
47         if (is_attacked(list[i], list[count]) == true) return false;
48     return true;
49 }
50
51 bool is_attacked(position q1, position q2)
52 {
53     return ( /* same row */
54             (q1.row == q2.row) ||
55             /* same column */
56             (q1.col == q2.col) ||
57             /* same / diagonal */
58             ((q1.row + q1.col) == (q2.row + q2.col)) ||
59             /* same \ diagonal */
60             ((q1.row - q1.col + CSIZE - 1) == (q2.row - q2.col + CSIZE - 1)));
61 }

```


Ergebnis

1. queen at a8.
2. queen at e7.
3. queen at h6.
4. queen at f5.
5. queen at c4.
6. queen at g3.
7. queen at b2.
8. queen at d1.



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/244/>

Knobelaufgaben**Aufgabe 12.1**

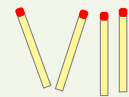
In 60 Minuten fährt ein Bus, den vier Wanderer ihn unbedingt erreichen wollen. Dazu müssen sie aber über eine wacklige Hängebrücke, die maximal zwei Personen trägt. Wegen der Dunkelheit können sie sich nur mit Taschenlampe auf die Brücke wagen – doch sie besitzen nur eine einzige, und die Brücke ist zu breit, um die Taschenlampe hinüberzuwerfen.

Außerdem sind die vier Männer unterschiedlich fit. Der erste schafft den Weg zur anderen Seite in 5 Minuten, der zweite in 10, der dritte in 20 und der vierte in 25.

Wie müssen die Wanderer vorgehen, um gemeinsam den Bus zu erreichen?

Aufgabe 12.2

Vier Streichhölzer sind wie auf dem Bild rechts angeordnet, so dass sie als römische Zahl aufgefasst den Wert 7 bilden. Durch Umlegen von nur einem Streichholz soll der Wert 1 entstehen.



Kapitel 13

Dateien und Textsuche

*Nur in seinem Suchen findet der Geist des
Menschen das Geheimnis, welches er sucht.*
— Friedrich Schlegel

*We have persistent objects, they're called
files.*
— Ken Thompson

13.1 Einführung

Sehr viele Computeranwendungen müssen in Datenbeständen etwas suchen. Diese Datenbestände sind häufig *persistent* gespeichert – d.h. sie bestehen weiter, wenn der Computer ausgeschaltet wird. In der Regel wird dies mit *Dateien* erreicht.

In diesem Kapitel werden wir uns daher mit (Text-)Suchalgorithmen und den Schnittstellen zur Dateiverwaltung.

- Betrachten Algorithmen zum Finden das erste Vorkommen einer bestimmten Zeichenkette (Muster) in einem längeren Text
- **Beispiel:** Gesucht wird „example“

*Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip **example** ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.*

13.2 Dateien

- Große Datenmengen (wie z.B. längere Texte) werden häufig nicht interaktiv eingegeben, sondern in **Dateien** gespeichert
- Beschäftigen uns daher zunächst mit Dateien

Definition 13.1: Datei

Eine **Datei** (*file*) ist eine Zusammenstellung von **logisch zusammengehörigen** Daten, die als eine Einheit behandelt werden.

Auf sie wird meist unter einem dem Betriebssystem bekannten **Dateinamen** (*file name*) zugegriffen. Dateien werden häufig auf **persistenten** Dateinträgern (z.B. Festplatten) gespeichert.

Dateien in C

- Dateien können unterschiedlich organisiert sein
- Im klassischen UNIX ist eine Datei stets ein einzelner, beliebig langer Datensatz von Bytes
- Dieser Datensatz wird als **Bytestrom** aufgefasst
 - Dadurch braucht nicht unterschieden werden, ob eine Eingabe aus einer Datei oder von einem Eingabegerät kommt
- Die Ein-/Ausgabe der C-Standardbibliothek ist diesem Konzept angepasst
- Die Standardbibliothek von C kennt zwei Klassen von Fileoperationen
 - **Low-Level-Fileoperationen**
 - ➔ Dateien/Datenströme werden über ein **Handle** identifiziert
 - ➔ spezifisch für jeweiliges Betriebssystem
 - **High-Level-Fileoperationen**
 - ➔ Dateien/Datenströme werden über einen **File-Pointer** identifiziert
 - ➔ unabhängig vom Betriebssystem
- Betrachten hier High-Level-Operationen
- Dateien müssen **geöffnet** werden
 - Dabei werden die Verwaltungsinformationen angelegt
 - Funktion der Standardbibliothek: `FILE* fopen(char* name, char* mode)`
 - Gibt einen Zeiger auf Dateiverwaltungsstruktur und im Fehlerfall NULL zurück
 - **Beispiel:**

```

#include<stdio.h>
/* ... */
FILE *fp;
fp = fopen("myfile.dat", "r");

```

- Wenn man die Datei nicht mehr braucht, sollte diese **geschlossen** werden
 - Freigabe der Verwaltungsressourcen
 - Bibliotheksfunktion `int fclose(FILE* stream)`
 - Gibt 0 bei fehlerfreier Ausführung zurück

Modi

Mode	Bedeutung
r	Öffnen einer Datei ausschließlich zum Lesen
w	Datei zum Schreiben erzeugen; existiert die Datei bereits, so wird sie überschrieben
a	Öffnen einer Datei zum Anfügen ; existiert die Datei bereits, so werden neue Daten an das Dateiende angefügt, ansonsten wird die Datei neu erzeugt
r+	Öffnen einer Datei zum Schreiben und Lesen ; die Datei muss bereits existieren
w+	Datei zum Schreiben und Lesen erzeugen; existiert die Datei bereits, so wird sie überschrieben
a+	Öffnen einer Datei zum Lesen und Schreiben ; existiert die Datei bereits, so werden neue Daten an das Ende angehängt, ansonsten wird die Datei neu erzeugt
b	<i>zusätzlich:</i> Öffnet Datei im Binär-Modus (sonst: Text-Modus)

Dateiein-/ausgabe

- Die Funktionen zur Ein- und Ausgabe von/in Dateien kann nach Interpretation der Daten unterschieden werden:
 - formatiert
 - einzelne Zeichen
 - Zeichenketten (Strings)
 - Binärdaten
- Wesentliche Funktionen sind:

	Eingabe	Ausgabe
formatiert	<code>int fscanf(FILE*, char*, ...)</code>	<code>int fprintf(FILE*, char*, ...)</code>
Zeichen	<code>int fgetc(FILE*)</code>	<code>int fputc(int, FILE*)</code>
Strings	<code>char* fgets(char*, int, FILE*)</code>	<code>int fputs(char*, FILE*)</code>
binär	<code>size_t fread(void*, size_t, size_t, FILE*)</code>	<code>size_t fwrite(void*, size_t, size_t, FILE*)</code>

Formatierte Dateiein-/ausgabe

- `fscanf()` und `fprintf()` funktionieren genauso wie `scanf()` und `printf()`, nur muss zusätzlich als erster Parameter ein Filepointer angegeben werden
 - Für `fscanf()` muss die Datei (mindestens auch) im Lesemodus geöffnet sein ➔ „r“, „r+“, „w+“, „a+“
 - Für `fprintf()` muss die Datei (mindestens auch) im Schreibmodus geöffnet sein ➔ „w“, „a“, „r+“, „w+“, „a+“

```

1  /* fprintf.c -- example for formatted file output */
2  #include<stdio.h>
3
4  int main()
5  {
6      FILE* file;
7      int n=42;
8      file = fopen("out.txt","w");
9      fprintf(file,"Hello world! The answer is %d\n",n);
10     fclose(file);
11     return 0;
12 }

```

Zeichenweise Dateiein-/ausgabe

- Die Funktion

```
int fgetc(FILE* stream)
```

gibt das nächste Zeichen einer Datei `stream` **als Integer**

- Wenn kein Zeichen (mehr) vorhanden ist, wird die Konstante `EOF` (definiert in `stdio.h`) zurückgegeben
- Gleiches gilt im Fehlerfall
- Die Funktion

```
int fputc(int c, FILE* stream)
```

schreibt das (ganzzahlcodierte) Zeichen `c` in die Datei `stream`

- Es wird die Anzahl der geschriebenen Zeichen (1) zurückgegeben
- Im Fehlerfall ist der Rückgabewert EOF (in `stdio.h` definiert)

Zeichenkettenein-/ausgabe aus/in Dateien

- Die Funktion

```
char* fgets(char restrict * str, int n, FILE* restrict stream)
```

liest maximal $n-1$ Zeichen aus der Datei `stream` in ein Zeichenarray, auf das `str` zeigt

- Das Lesen wird bei einem Zeilen- oder Dateiende oder bei einem Fehler beendet
 - Wenn kein Fehler auftrat, wird „\0“ an `str` angehängen
- Der Rückgabewert zeigt im Erfolgsfall auf `str`, sonst wird `NULL` zurück gegeben

Achtung

Es liegt in der Verantwortung des Programmierers, dass `str` auf ein Zeichenarray zeigt, dass (mindestens) n Zeichen groß ist!

Hinweis

Neben `fgets()` existierte bis C99 noch die ähnliche Funktion `gets()`. **Diese sollte nicht mehr verwendet werden, da sie ein Sicherheitsrisiko darstellt.** Entsprechend wurde `gets()` ab C11 aus der Standardbibliothek entfernt.



- Die Funktion

```
int fputs(char* str, FILE* stream)
```

gibt den (nullterminierten) String `str` in die Datei `stream` aus

- Funktion gibt im Erfolgsfall einen nichtnegativen Wert zurück, im Fehlerfall EOF
- **Achtung:** Alte Versionen gaben im Erfolgsfall immer 0 zurück



Binäre Dateiein-/ausgabe

- Die Funktion

```
size_t fread(void* ptr, size_t size, size_t nitems, FILE* stream)
```

liest `nitems` Einträge der jeweiligen Größe `size` aus der Datei `stream` und speichert sie unter der von `ptr` angegebenen Adresse

- Es wird die Anzahl der erfolgreich gelesenen Einträge (nicht der Bytes!) zurückgegeben
- Die Funktion

```
size_t fwrite(void* ptr, size_t size, size_t nitems, FILE* stream)
```

schreibt `nitems` ab der Adresse `ptr` angegebenen Einträge der jeweiligen Größe `size` in die Datei `stream`

- Es wird die Anzahl der erfolgreich geschriebenen Einträge (nicht der Bytes!) zurückgegeben

Standarddatenströme

- Beim Start eines C-Programmes werden automatisch drei Datenströme geöffnet
- Programmierer hat darauf Zugriff über in `stdio.h` Variablen von Typ `FILE*`

Variable	Funktion	Voreingestelltes Ziel
<code>stdin</code>	Standardeingabe	Tastatur
<code>stdout</code>	Standardausgabe	Bildschirm
<code>stderr</code>	Standardfehlerausgabe	Bildschirm

- `stdin`, `stdout` und `stderr` brauchen (und können) nicht geöffnet werden
- Die Unterscheidung von `stdout` und `stderr` kann zur Trennung von Nutz- und Statusdaten genutzt werden ➔ **Umlenkung** auf Kommandozeilenebene

```

1 /* fstd.c -- standard out vs. standard error */
2 #include<stdio.h>
3
4 int main()
5 {
6     fprintf(stdout,"Dies sind Nutzdaten.\n");
7     fprintf(stderr,"Dies ist eine Statusmeldung.\n");
8     return 0;
9 }

```

```

> ./fstd
Dies sind Nutzdaten.
Dies ist eine Statusmeldung.
> ./fstd 2> /dev/null
Dies sind Nutzdaten.
> ./fstd > /dev/null
Dies ist eine Statusmeldung.

```

Manipulation des Positionszeigers

- Allgemein werden Dateien als Datenströme aufgefasst, auf die **sequentiell** zugegriffen wird
- Bei „echten“ Dateien kann von der sequentiellen Zugriffsweise abgewichen werden

- Dazu stehen folgende Funktionen zur Verfügung:
 - `void rewind(FILE* stream)`
Setzt die Lese- oder Schreibposition der Datei `stream` wieder auf den Dateianfang
 - `int fseek(FILE* stream, long offset, int whence)`
Setzt die Lese- oder Schreibposition der Datei `stream` eine Position, die `offset` Bytes von `whence` entfernt ist
 - `whence` ist eine von drei in `stdio.h` definierten Konstanten:
 - `SEEK_SET`: offset ist relativ zum Dateianfang
 - `SEEK_CUR`: offset ist relativ zur aktuellen Position
 - `SEEK_END`: offset ist relativ zum Dateieinde
- Mit Hilfe von `long ftell(FILE* stream)` kann die aktuelle Position (relativ zum Dateianfang) abgefragt werden

Beispiel

```

1  /* fwrite.c -- write an array to a binary file */
2  #include <stdio.h>
3
4  int main () {
5      float array[] = {23.0f, 3.1415f, 42.0f, 47.11f};
6      const char* const filename = "example.bin";
7      FILE* file;
8      if ((file = fopen(filename, "w")) == NULL) {
9          fprintf(stderr, "error: can't open file\n");
10         return 1;
11     }
12     if (fwrite(array, sizeof(array), 1, file) == 1) {
13         fprintf(stderr, "success.\n");
14         return 0;
15     } else {
16         fprintf(stderr, "write error.\n");
17         return 1;
18     }
19 }

```



Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/1247/>

```

1  /* fread -- read an array from a binary file */
2  #include <stdio.h>
3
4  int main () {
5      float array[4];
6      const char* const filename="example.bin";
7      FILE* file;
8
9      if ((file = fopen(filename,"r")) == NULL) {
10         fprintf(stderr,"error: can't open file\n");
11         return 1;
12     }
13     if (fread(array, sizeof(array),1,file) == 1){
14         fprintf(stderr,"success\n");
15         for(int i=0; i<4; ++i) {
16             fprintf(stdout,"%f%s",array[i], (i==3)? "\n": ", ");
17         };
18         return 0;
19     } else {
20         fprintf(stderr,"read error.\n");
21         return 1;
22     }
23 }

```

```

> ./fwrite
success
> cat example.bin
?AVI@B?p<B
>
> ./fread
success
23.000000, 3.141500, 42.000000, 47.110001

```

Weitere Funktionen

- Die Standardbibliothek noch andere Funktionen zur Verfügung
- Interessant sind u.a.:
 - `int feof(FILE* stream)`
gibt bei Dateiende Wert ungleich 0 zurück
 - `int ferror(FILE* stream)`
gibt einen Wert ungleich 0 zurück, wenn vorher ein Dateifehler aufgetreten war
 - `int fflush(FILE* stream)`
Erzwingt das physische Schreiben (Cache wird geleert)

- `int remove(char* name)`
Löscht die Datei mit dem Namen `name`

Dateien in Python

- Eine Datei ist ein **Datentyp** in Python
- Eine Dateivariablen wird mit `f=open(filename[, mode[, bufsize]])` angelegt
- Modi sind eine Obermenge der C-Modi
- Mit `bufsize` kann die Cache-Größe für die Datei gesetzt werden
- Verschiedene Dateioperationen stehen zur Verfügung, siehe Tabelle

Operation	Bedeutung
<code>S=file.read()</code>	Liest gesamte Datei in einen einzigen String
<code>S=file.read(N)</code>	Liest N Bytes
<code>S=file.readline()</code>	Liest nächste Zeile (bis Zeilenendmarker)
<code>L=file.readlines()</code>	Liest gesamte Datei als Liste von Zeilen-Strings
<code>file.write(S)</code>	Schreibt String S in Datei
<code>file.writelines(L)</code>	Schreibt alle Strings einer Liste L in Datei
<code>file.close()</code>	Schliesst Datei

- Durch den Iterationsansatz (vgl. Kapitel 7) kann einfach eine komplette Datei zeilenweise bearbeitet werden

```

1 f= open("foo.txt","r")
2 for line in f:
3     print(line, end=' ')

```

- Die Python-Bibliothek stellt verschiedene Module für Filemanipulation auf verschiedenen Abstraktionsebenen zur Verfügung, z.B.:
 - Low-Level-Dateien ➔ Modul `os`
 - High-Level-Speicherung komplexer Objekte ➔ Module `shelve` und `pickle`
 - Datenbank-Schnittstellen ➔ Module `dbm` und `anydbm`

13.3 Einfache Suche

- Mit den Kenntnissen über Dateien können wir nun das Textsuchprogramm beginnen
- Das Programm soll folgende Aufrufparameter haben:

```
./search <Suchtext> <Dateiname>
```

- Suchtext ist der Text, nach dem gesucht wird
- Dateiname ist die Datei, in der gesucht wird
- Wenn der Suchtext gefunden wird, soll die „Umgebung“ ausgegeben werden, in der er vorkommt
- Dabei soll der Suchtext durch eckige Klammern hervorgehoben werden

```
./search "example" lorem.txt
ullamcorper suscipit lobortis nisl ut aliquip [example] ea commodo
```

Größe

- **Problem:** Es ist nicht bekannt, wie lang maximal eine Zeile oder die gesamte Datei ist
- **Lösung 1:** Einen Zeilenpuffer definieren, der „hinreichend“ groß für alle Fälle ist ➔ nicht sicher
- **Lösung 2:** Nicht immer ganze Zeilen einlesen ➔ erschwert Suche, wenn der Suchtext zwischen zwei Einleseblöcken liegt
- **Lösung 3:** Dateigröße ermitteln, dynamisch Platz reservieren und die gesamte Datei einlesen ➔ braucht evtl. sehr viel Speicher
- Wählen Lösung 3
 - ➔ Bietet Geschwindigkeitsvorteile
- Wie Filegröße ermitteln?
- Unter UNIX gibt es die Funktion

```
int stat(char* name, struct stat* buf)
```

die verschiedene Informationen über Datei `name` nach `buf` schreibt

- Größe = `buf.st_size` (in Bytes)
- Nicht kompatibel

- **Alternative:** Trickreiche Nutzung von Standardfunktionen

```

31 size_t filesize(FILE* file)
32 {
33     size_t ret;
34     fseek(file, 0L, SEEK_END);
35     ret = ftell(file);
36     rewind(file);
37     return ret;
38 }

```

main-Funktion

```

6 int main(int argc, char* argv[]){
7     FILE* file;
8     char* text;
9
10    if (argc != 3) {
11        return -2; // wrong number of parameters
12    }
13    if ((file = fopen(argv[2], "r")) == NULL) {
14        return -3; // can't open file
15    }
16    size_t size = filesize(file);
17    if ((text = malloc(size+1)) == NULL) {
18        return -4; // out of memory
19    }
20    if (fread(text, size, 1, file) != 1){
21        return -5; // can't read file
22    }
23    text[size] = '\0';
24    int found = search(argv[1], text, size); // yet to implement
25    if (found != -1)
26        presentresult(found, text, argv[1]); // dito
27    free(text);
28    return found;
29 }

```

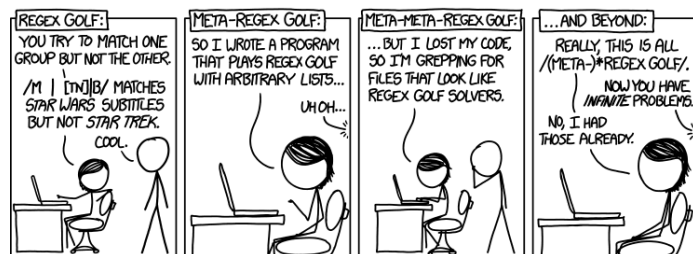
Ausgabe

- Die `main()`-Funktion besteht im Wesentlichen aus Fehlerprüfungen
- Außer der eigentlichen Suche brauchen wir noch eine Ausgabe
- Geben 20 Zeichen vor und nach Suchmuster aus

```

42 void presentresult(int pos, const char* str, const char* pattern)
43 {
44     int start, end, patlen, prelen;
45     start=pos>20? pos-20 : 0;
46     prelen=pos>20? 20 : pos;
47     patlen=length(pattern);
48     end=pos+patlen;
49     printf("%.s[s[%s]%.20s\n",prelen,&str[start],pattern,&str[end]);
50 }

```



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1313/>

Algorithmus

- Jetzt kann der eigentliche Algorithmus betrachtet werden
- **Idee:** Teste für jede Position des Textes *str*, ob der Suchstring *p* dort beginnt
 - Wenn ja, teste nächstes Zeichen etc.
- **Beispiel:** *str*="sein oder nicht sein!", *p*="nich"

sein oder nicht sein	
sein	X
oder	X
nicht	X
sein	X
oder	✓
nicht	✓X
sein	X
oder	X
nicht	X
sein	X
oder	X
nicht	X
sein	X
oder	✓
nicht	✓✓
sein	✓✓✓
oder	✓✓✓✓
nicht	✓✓✓✓✓ ➔ gefunden

Algorithm SIMPLE-SEARCH**Require:** str, p is text; $length(str) > length(p)$ **Ensure:** returns index of first appearance of p in str

```

procedure SIMPLE-SEARCH( $str, p$ )
     $pos \leftarrow 1$  ▷ first element
    while  $pos < length(str) - length(p)$  do
         $j \leftarrow 1$ 
        while  $(j \leq length(p)) \wedge (str[pos + j - 1] = p[j])$  do
            if  $j = length(p)$  then ▷ all characters matched
                return  $pos$ 
            end if
             $j \leftarrow j + 1$ 
        end while
         $pos \leftarrow pos + 1$ 
    end while
    return „not found“
end procedure

```

C-Implementierung

- Für den Algorithmus müssen wir die Länge eines Strings ermitteln
- Dafür gibt es eine Funktion in der Standardbibliothek
- Nutzen aber eigene Funktion:

```

21 int length(const char* str)
22 {
23     int len=0;
24     if (str==NULL) return 0;
25     while(str[len]!='\0') ++len;
26     return len;
27 }

```

- Der Algorithmus lässt sich relativ ähnlich zum Pseudocode in C implementieren
- Zu beachten:
 - Vorbedingungen sollten überprüft werden
 - Indizes beginnen bei 0

```

5 int search(const char* p, const char* str, size_t tlen)
6 {
7     size_t plen=length(p);
8     if (plen>tlen) return -1;
9     for(size_t pos=0; pos<tlen; ++pos){
10         size_t j=0;
11         while((j<plen) && (str[pos+j]==p[j])){
12             if (j==plen-1) return pos;
13             ++j;
14         }
15     }
16     return -1;
17 }

```

Bewertung des Algorithmus

- Wir haben zwei verschachtelte Schleifen
- Die erste durchläuft die Anzahl n der Elemente der Zeichenkette `str`
- Die zweite durchläuft die Anzahl m der Elemente des Suchmusters `p`
- ➔ Haben demzufolge eine Aufwandsobergrenze von $\mathcal{O}(m \cdot n)$
- Aufwandsuntergrenze liegt bei $\mathcal{O}(m)$
- Gibt es ein effizienteres Verfahren?

Boyer-Moore-Algorithmus

- Algorithmus von Robert S. Boyer und J. Strother Moore 📖[BM77]
- **Idee:**
 - Algorithmus vergleicht zunächst die **letzte** Stelle des Suchmusters `p` und verschiebt bei fehlender Übereinstimmung `p` soweit vor, bis die letzte Stelle wieder passt
 - Dabei wird in Abhängigkeit des Inhalts von `p` oft mit einem Schritt eine Verschiebung von `p` um mehrere Stellen erreicht
- **Beispiel**, `str="sein oder nicht sein!", p="nich"`

sein oder nicht sein	
nich	'h'≠'n', aber 'n'=p[1] ➡ Verschieben um 3
nich	'h'≠'d' und 'd' ∉ p ➡ Verschieben um 4
nich	'h'≠'n', aber 'n'=p[1] ➡ Verschieben um 3
nich	'h'='h'
nich	'c'='c'
nich	'i'='i'
nich	'n'='n' ➡ gefunden

• Berechnung der Verschiebung

- Entscheidend für die Verschiebung ist, wie weit sich das betrachtete Zeichen aus **str vom Ende** des Musters **p entfernt** befindet
 - * Ist es **nicht** in **p enthalten**, können wir das Suchmuster um die **komplette Suchmusterlänge** nach vorn schieben
 - * Ansonsten verschieben wir es um die Anzahl Stellen, die die **kürzeste Entfernung** des Zeichens zum Ende des Suchmusters ausmacht
- ➡ Zur Realisierung des Algorithmus brauchen wir also die Information, wie groß für alle in **p** vorkommenden Zeichen ihre **Distanz** zum Ende von **p** ist
 - Für alle anderen (nicht in **p** vorkommenden) Zeichen entspricht diese Distanz der Länge von **p**
 - Letztes Zeichen im Suchmuster (Distanz=0) muss nicht erfasst werden
 - Algorithmus ist besser für lange **p**

Aufbau der Distanztabelle

- Solange wir im ASCII/ANSI-Zeichensatz bleiben, können wir einfach eine Tabelle nutzen
- **Idee:** Tabelle mit **allen** Zeichen, nutzen vorkommendes Zeichen als **Index**

```

14 enum {MAXCHAR=248}; // covers all ISO 8859-1 umlauts
15 int disttable[MAXCHAR];
102
103 int init_table(const char* pattern) {
104     int len=length(pattern);
105     for(int i=0; i<MAXCHAR; ++i)
106         disttable[i]=len;
107     for(int i=0; i<len-1; ++i) {
108         if (((unsigned char)pattern[i])>=MAXCHAR)|| (pattern[i]<0))
109             return -1;
110         if (disttable[(int)pattern[i]]>len-i-1)
111             disttable[(int)pattern[i]]=len-i-1;
112     }
113     return 0;
114 }

```

- Uneffizient, aber (noch) praktikabel $\Rightarrow$ viel schlechter bei Unicode o.ä.
- Besser: Nutzung eines **Hashes** $\Rightarrow$ VL „Datenstrukturen“
- Bei Python kann Dictionary-Typ genutzt werden

Implementation der Suchfunktion

- Die Implementation der Suchfunktion ist ähnlich wie im simplen Algorithmus

```

56 int search(const char* p, const char* str, size_t tlen)
57 {
58     size_t plen=length(p);
59     if (plen>tlen) return -1;
60     size_t pos=plen-1;
61     while(pos<tlen){
62         size_t j=0;
63         while((j<plen) && (str[pos-j]==p[plen-j-1])){
64             if (j==plen-1)
65                 return pos-pos+1;
66             ++j;
67         }
68         pos=pos+disttable[(int)str[pos]];
69     }
70     return -1;
71 }

```

Bewertung des Algorithmus

- Der Algorithmus arbeitet am effizientesten, wenn die zu durchsuchende Zeichenkette *str* der Länge *n* (außer ggf des zu suchenden Worts) nur aus Zeichen besteht, die im Muster *p* der Länge *m* **nicht** vorkommen (bester Fall)
 - $\Rightarrow$ Dann ist Komplexität $\mathcal{O}(\frac{n}{m})$, also **sublinear**¹
- Im **ungünstigsten** Fall ist Komplexität jedoch ebenfalls $\mathcal{O}(m \cdot n)$ $\Rightarrow$ zwei verschachtelte Schleifen mit maximal *n* bzw. *m* Durchläufen
- Außerdem wird zusätzliche Zeit für Tabellenaufbau benötigt
- In der Praxis (häufig relativ lange Texte) wird der Boyer-Moore-Algorithmus in der Regel jedoch wesentlich **effizienter** ablaufen als der einfache Mustervergleich
- Weitere Verbesserungen möglich $\Rightarrow \mathcal{O}(n + m)$

¹...falls $m \ll n$, sonst irgendwann $\mathcal{O}(m + \frac{n}{m})$ aufgrund des Aufbaus der Tabelle

13.4 Wildcards

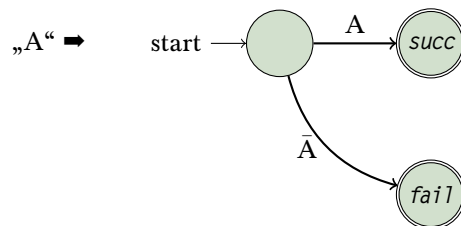
- Erweitern Problemstellung: Suchmuster soll **Jokerzeichen** (Wildcards) enthalten können
- Nutzen drei Wildcards:
 - „?“ Passt auf **genau** einen Buchstaben
 - „/“ Passt auf **einen** oder **keinen** Buchstaben
 - „*“ Passt auf eine **beliebige Anzahl** von Buchstaben (auch 0)
- **Beispiele:**
 - „A?C“ wird in „abcABC“ und „aaAcCcc“ gefunden, aber nicht in „aACc“
 - „A!C“ wird in „abcABC“ und „aACc“ gefunden, aber nicht in „aAbbCc“
 - „A*C“ wird in „aAbbCc“ und „aACc“ gefunden, aber nicht in „cCbbAa“

Idee

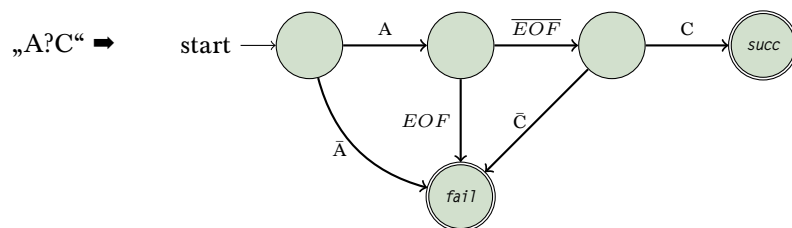
- Nutzung von Automaten (siehe Kapitel 10)
 - Ausdrücke mit Wildcards sind **regulär**
 - Äquivalenter Automaten existiert zum Test auf Übereinstimmung
 - * Der Suchstring definiert eine reguläre Grammatik
 - * Automat kommt in einen **Akzeptiere**-Zustand an, wenn ein der Grammatik entsprechender Ausdruck im zu durchsuchenden Text gefunden wird
- Jeder Suchstring ist anders ⇒ Automat muss (zur Laufzeit!) **generiert** werden
 - Automat besteht aus generischen Teilautomaten
 - Vom Start kommt man in einen der End-Zustände:
 - * (Teil-)**Erfolg** (s, „gefunden“) = *accept*
 - * **Misserfolg** (f, „nicht gefunden“)
 - Ohne Jokerzeichen ⇒ Kette ⇒ *accept*-Zustand ist *start*-Zustand des nächsten Teilautomat

Automaten

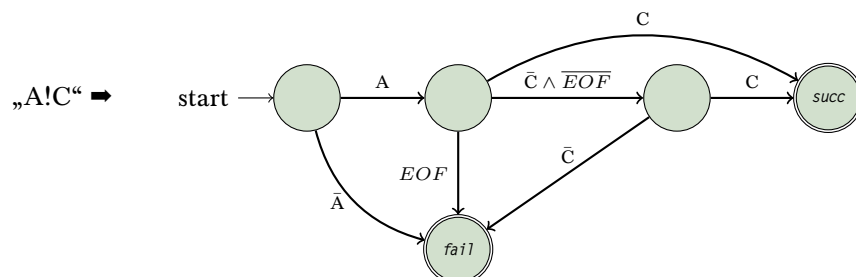
- Automat für Nicht-Joker-Zeichen (z.B. „A“)
- Ausgangskante für Eingabe von $A \Rightarrow s$ = nächster Zustand
- Ausgangskante für alles andere $\Rightarrow f$



- Automat für Joker-Zeichen „?“ (z.B. „A?C“)
- Ausgangskante für End-of-Text (EOF) $\Rightarrow f$
- Ausgangskante für alles andere $\Rightarrow s$ = nächster Zustand

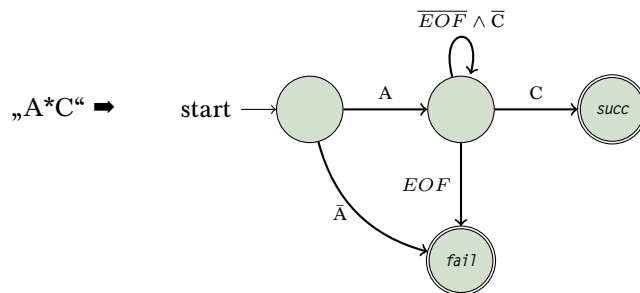


- Automat für Joker-Zeichen „!“ (z.B. „A!C“) $\Rightarrow$ nutzt Zeichen der Ausgangskante des **nächsten** Zustandes
- Ausgangskante für Eingabe von $C \Rightarrow$ **ü**bernächster Zustand
- Ausgangskante für End-of-Text $\Rightarrow f$
- Ausgangskante für alles andere $\Rightarrow$ nächster Nicht-„!“-Zustand



- Automat für Joker-Zeichen „*“ (z.B. „A*C“)
- Ausgangskante für Eingabe von $c \Rightarrow$ nächster Zustand

- Ausgangskante für End-of-Text $\Rightarrow f$
- Für alles andere $\Rightarrow$ Schleife zu sich selbst



Problem

- **Problem:** Was ist, wenn „!“ oder „*“ **kein** normales Zeichen folgt?
- **Beobachtungen:**
 1. Eine Reihe aus ausschließlich „!“ sind nicht kritisch
 2. Beginnt das Suchmuster mit „*“ oder einem „!“, so kann das Wildcard ignoriert (übergangen) werden
 3. Folgt einem „*“ oder einem „!“ End-of-Text, so kann das Wildcard ignoriert (übergangen) werden
 4. Folgt einem „!“ oder einem „*“ ein „?“, so können beide Zeichen getauscht werden
 5. Folgt einem „!“ ein „*“, so kann das „!“ ignoriert (übergangen) werden
 6. Folgt einem „*“ ein „!“ oder „*“, kann das Folgezeichen ignoriert werden
- **Folgerung:** Durch Transformation kann immer erreicht werden, dass die Generierungsregeln hinreichen sind
- Entsprechend der Beobachtungen 2–6 können Umformregeln in einem Algorithmus formuliert werden



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1171/>

Algorithmus

Algorithm SANITIZE

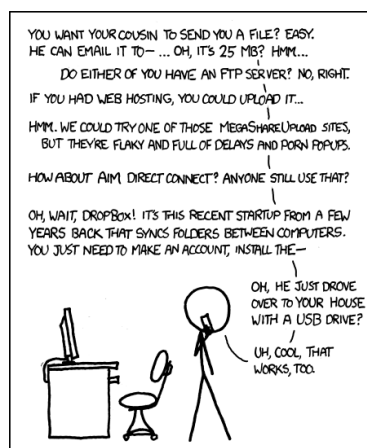
Require: *str*, possibly with wildcards

Ensure: returns sanitized *str*

```

repeat
  while  $str[first] = '*' \vee str[first] = '!'$  do
    remove first character from str                                ▷ #2
  end while
  while  $str[last] = '*' \vee str[last] = '!'$  do
    remove last character from str                                ▷ #3
  end while
  changed  $\leftarrow$  true
  for  $\forall sub \in str, length(sub) = 2$  do
    if  $sub = '!?'$  then
      replace in str sub with '?'                                ▷ #4
    else if  $sub = '*?'$  then
      replace in str sub with '?'                                ▷ #4
    else if  $(sub = '!*') \vee (sub = '*!') \vee (sub = '**')$  then
      replace in str sub with '*'                                ▷ #5,#6
    else
      changed  $\leftarrow$  false
    end if
  end for
until (changed = false)

```



I LIKE HOW WE'VE HAD THE INTERNET FOR DECADES, YET "SENDING FILES" IS SOMETHING EARLY ADOPTERS ARE STILL FIGURING OUT HOW TO DO.

Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/949/>

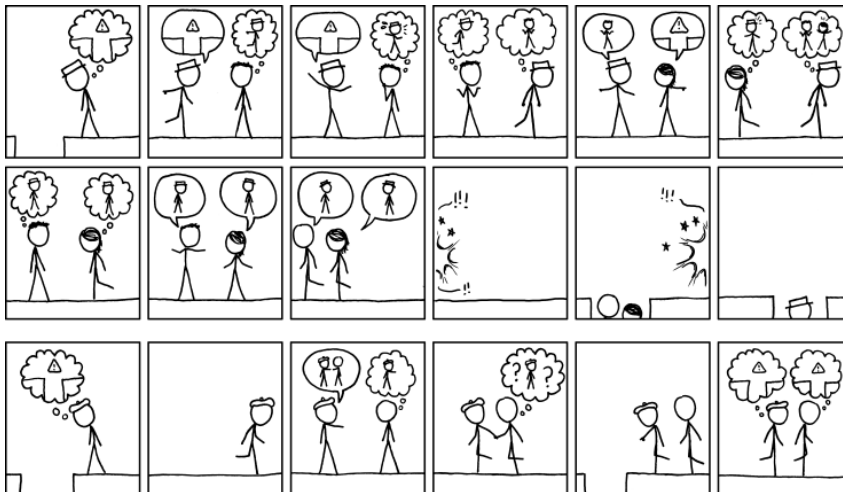
Implementation der Transformationsregeln

- Da in Python Stringmanipulationen einfacher sind, betrachten wir zunächst eine Implementation in Python

```

2 def sanitize(str):
3     changed=True
4     while (changed==True):                # repeat as often as needed
5         changed=False
6         while (str[0]=='*') or (str[0]=='!'): # delete leading * or !
7             str=str[1:]
8         while (str[-1]=='*') or (str[-1]=='!'): # delete trailing * or !
9             str=str[:-1]
10        for i in range(0,len(str)-2):
11            if (str[i:i+2]=='*?') or (str[i:i+2]=='!?'):
12                changed=True
13                str=str[0:i]+'?'+str[i]+str[i+2:] # *? => ?* and !? => ?!
14            if ((str[i:i+2]=='!*') or (str[i:i+2]=='*!') or
15                (str[i:i+2]=='**')):
16                changed=True
17                str=str[0:i]+'*'+str[i+2:]        # !* or *! or ** => *
18    return str

```



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1028/>

C-Implementierung

- Die C-Implementierung ist etwas länger, macht aber genau das Gleiche

```

147 char* sanitize(char* str) {
148     char* nstr=str;
149     bool changed;
150     int len=length(nstr);
151     if ((nstr==NULL) || (len==0)) return NULL;
152     do {
153         changed=false;
154         while((nstr[len-1]=='*') || (nstr[len-1]=='!')){
155             nstr[--len]='\0'; /* shorten tail by 1 */
156         }
157         while((nstr[0]=='*') || (nstr[0]=='!')) {
158             nstr=&nstr[1]; /* move start 1 to the right */
159             --len;
160         }
161         for(int i=0; i<len; ) {
162             if (((nstr[i]=='*') || (nstr[i]=='!')) && (nstr[i+1]=='?')) {
163                 nstr[i+1]=nstr[i]; /* !? => ?! resp. *? => ?* */
164                 nstr[i]='?';
165                 changed=true;
166             } else ++i;
167         }
168         for(int i=1; i<len-1; ){
169             if (((nstr[i]=='*') && (nstr[i+1]=='!')) ||
170                 ((nstr[i]=='!') && (nstr[i+1]=='*')) ||
171                 ((nstr[i]=='*') && (nstr[i+1]=='*'))){
172                 --len;
173                 nstr[i]='*';
174                 int j=i+1;
175                 do{ /* move remains one to the left */
176                     nstr[j]=nstr[j+1];
177                     ++j;
178                 } while(nstr[j]!='\0');
179                 changed=true;
180             }
181             else ++i;
182         }
183     } while(changed);
184     return nstr;
185 }
186

```

Datenstrukturen

- Der Automat muss zur Laufzeit generiert werden ➡ Darstellung in Daten, nicht im Code
- Was ist eine geeignete Datenstruktur?

- Automat ist ein Graph (vergleiche Exkurs im Kapitel 8), d.h., er besteht aus Knoten und Kanten
- Knoten sind die Zustände im Automaten
- Kanten sind die Übergänge
- Da sowohl Knoten als auch Kanten weitere Informationen „tragen“, ist eine Adjazenzmatrix ungeeignet ➔ wir nehmen `structs` mit Verweisen, vergleiche Kapitel 8, Seite 158
- Was ist nötig zur Beschreibung eines **Übergangs**?
 1. Ausgangs- und Endzustand
 2. Bedingung
- Wir werden die Ausgangskanten den entsprechenden Zuständen zuordnen, deshalb brauchen wir nur das Ziel
- Es kommen im Automaten nur fünf verschiedene Bedingungen vor:²
 1. Das Zeichen wird gefunden
 2. Das Zeichen wird nicht gefunden
 3. Der Text ist zu Ende
 4. Der Text ist nicht zu Ende
 5. Es wurde weder das Zeichen noch das Textende gefunden
- Entsprechend können wir Konstanten und eine Datenstruktur vereinbaren:

```

17 typedef enum { MatchChar, MatchNotChar, MatchEOT, MatchNotEOT,
    MatchNotCharNotEOT, NoEdge } match_t;

19 typedef struct[
20     match_t condition;
21     int next;
22 ] edge_t;

```

- Mit der Konstante `NoEdge` sollen Kanten markiert werden, die im Automaten nicht existieren
- Wir Beschreibung einen Zustand mit:
 1. Markierung ob Endzustand (`s` oder `f`) oder anderer
 2. Das Zeichen, auf den sich der Zustand bezieht
 3. Die Übergänge von diesem Zustand
- Da es immer nur maximal drei Übergänge zu einem anderen Zustand gibt, erlauben wir uns, ggf. etwas Speicherplatz zu verschwenden und ein Array für genau drei Kanten anzulegen

```

25 typedef enum { StateDefault, StateSuccess, StateFail} statetype_t;
27 typedef struct{
28     statetype_t type;
29     char        ch;
30     edge_t       edge[3];
31 } state_t;

```

- Wir wissen erst zur Laufzeit, wie groß unser Automat wird, d.h., wie viele Zustände er hat
- Wir legen anonymes Array an ➔ brauchen Zeiger
- Zusätzlich Index des Startzustands

```

35 typedef struct{
36     state_t* state;
37     int      initial;
38 } automata_t;

```

Konstruktion des Automaten

- Nun kann der Automat zur Laufzeit aus dem Suchmuster erzeugt werden

```

48 automata_t init_automata(const char*);

```

- Wir brauchen...
 - einen Success-Zustand
 - einen Fail-Zustand
 - einen Zustand pro Nicht-*Zeichen im Suchmuster
- Dafür wird Speicher reserviert

```

188 automata_t init_automata(const char* p) {
189     int snr, len;
190     len=snr=length(p);
191     for(int i=len-1; i>=0; --i)
192         if(p[i]=='*') --snr; // decrease for each "*"
193     automata_t a;
194     a.state = calloc(snr+2, sizeof(state_t));

```

- In das anonyme Array werden die Elemente (Automatenzustände) eingetragen
- Der Success- und der Fail-Zustand sind immer gleich, wir vereinbaren dafür Indexkonstanten:

```

33 enum { S_FAIL=0, S_SUCC=1 };
195 /* Fail */
196 a.state[S_FAIL].type=StateFail;
197 a.state[S_FAIL].edge[0].condition=NoEdge;
198 a.state[S_FAIL].edge[1].condition=NoEdge;
199 a.state[S_FAIL].edge[2].condition=NoEdge;
200 /* Success */
201 a.state[S_SUCC].type=StateSuccess;
202 a.state[S_SUCC].edge[0].condition=NoEdge;
203 a.state[S_SUCC].edge[1].condition=NoEdge;
204 a.state[S_SUCC].edge[2].condition=NoEdge;

```

- Theoretisch könnten sie ausgelassen werden
- Jedoch vereinfacht ihre Existenz die allgemeine Behandlung
- Die anderen Zustände werden durch Fallunterscheidung aus dem Suchmusterstring generiert
- Fangen vom Ende an ➔ haben Folgezustände stets zur Verfügung

```

206 snr=1;
207 for(int pos=len-1; pos>=0; --pos){
208     switch (p[pos]) {
209         :
210     }
245 }
246 }

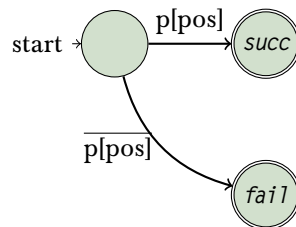
```

- Zunächst der Fall für normale Zeichen:
 - Zwei Ausgangskanten
 - Eine hat das Zeichen aus dem Suchmuster zur Bedingung, die andere die Negation
 - Bedingungen mit der Markierung `NoEdge` werden später nicht ausgewertet

```

208 switch (p[pos]) {
209     default:
210         ++snr;
211         a.state[snr].ch=p[pos];
212         a.state[snr].type=StateDefault;
213         a.state[snr].edge[0].condition=MatchChar;
214         a.state[snr].edge[0].next=snr-1;
215         a.state[snr].edge[1].condition=MatchNotChar;
216         a.state[snr].edge[1].next=S_FAIL;
217         a.state[snr].edge[2].condition=NoEdge;
218         break;

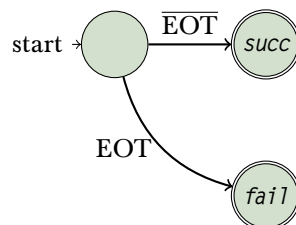
```



- Der Fall für das Wildcard-Zeichen „?“ ist ähnlich wie der für ein normales Zeichen, nur dass die Bedingung *EOT* ist:

```

219 case '?':
220     ++snr;
221     a.state[snr].type=StateDefault;
222     a.state[snr].edge[0].condition=MatchNotEOT;
223     a.state[snr].edge[0].next=snr-1;
224     a.state[snr].edge[1].condition=MatchEOT;
225     a.state[snr].edge[1].next=S_FAIL;
226     a.state[snr].edge[2].condition=NoEdge;
227     break;
  
```



- Der Zustand für das Wildcard-Zeichen „!“ braucht drei Ausgangskanten
 - Eine Kante führt zum **Folgezustand des Folgezustands** mit der Bedingung, dass das **Zeichen des Folgezustandes** gefunden wird
 - Eine Kante führt zum Folgezustand, wenn weder das Zeichen des Folgezustandes noch *EOT* (d.h. $\neq \backslash 0$) gefunden wird
 - Bei *EOT* landet der Automat in *Fail*

```

228 case '!':
229     ++snr;
230     a.state[snr].ch=a.state[snr-1].ch; // get character from next
        state
231     a.state[snr].type=StateDefault;
232     a.state[snr].edge[0].condition=MatchChar;
233     a.state[snr].edge[0].next=a.state[snr-1].edge[0].next;
234     a.state[snr].edge[1].condition=MatchEOT;
235     a.state[snr].edge[1].next=S_FAIL;
236     a.state[snr].edge[2].condition=MatchNotCharNotEOT;
237     a.state[snr].edge[2].next=snr-1;
238     break;

```

- Das Wildcard-Zeichen „!“ generiert **keinen** neuen Zustand
- Vielmehr wird der Zustand des Folgezeichens **modifiziert**
 - Die Bedingung der Kante zu *Fail* wird auf *EOT* abgeschwächt
 - Eine Schleife zu sich selbst wird hinzugefügt, wenn weder das ursprüngliche Zeichen noch *EOT* eintritt

```

239 case '*':
240     // no ++snr !
241     a.state[snr].edge[1].condition=MatchEOT;
242     a.state[snr].edge[2].condition=MatchNotCharNotEOT;
243     a.state[snr].edge[2].next=snr;
244     break;

```

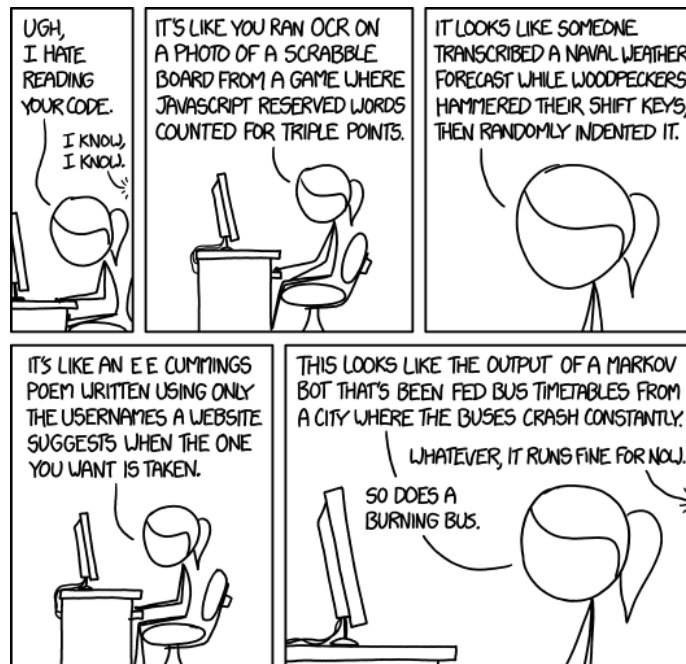
- Zum Schluss: Bestimmung des Initialzustands des Automaten
- Da wir das Suchmuster „von hinten“ bearbeitet haben, ist der Initialzustand der letzte generierte Zustand

```

206 snr=1;
207 for(int pos=len-1; pos>=0; --pos){
208     switch (p[pos]) {
209         :
210     }
245 }
246 }
247 a.initial=snr;
248 return a;
249 }

```

- Damit ist der Automat konstruiert und kann eingesetzt werden



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1695/>

Ausführung des Automats

- Nachdem wird den Automat für die Suche initialisiert haben, wird er bei der Suche ausgeführt
- Da durch die Wildcards die Länge des gefundenen Textstücks nicht vorher klar ist, muss die Suche zwei Werte zurückgeben

```

40 typedef struct {
41     int start;
42     int end;
43 } searchresult_t;

45 searchresult_t search(automata_t, const char*, size_t);

```

- Bei der Ausführung muss im Wesentlichen abgeprüft werden, ob eine Bedingung zutrifft, und dann in den entsprechenden Folgezustand gewechselt werden
- Kommt der Automat dabei in `Fail` oder `Success` wird der Automat beendet

```

88 searchresult_t search(automata_t a, const char* text, size_t tlen) {
89     searchresult_t res;
90     for(size_t pos=0; pos<tlen; ++pos){
91         int snr=a.initial;
92         size_t j=pos;
93         while(a.state[snr].type==StateDefault){
94             for(int i=0; i<3; i++){
95                 if (((a.state[snr].edge[i].condition==MatchChar) &&
96                     (text[j]==a.state[snr].ch)) ||
97                     ((a.state[snr].edge[i].condition==MatchNotChar) &&
98                     (text[j]!=a.state[snr].ch)) ||
99                     ((a.state[snr].edge[i].condition==MatchEOT) &&
100                    (text[j]=='\0')) ||
101                    ((a.state[snr].edge[i].condition==MatchNotEOT) &&
102                    (text[j]!='\0')) ||
103                    ((a.state[snr].edge[i].condition==MatchNotCharNotEOT) &&
104                    (text[j]!='\0') &&
105                    (text[j]!=a.state[snr].ch))))
106             {
107                 snr = a.state[snr].edge[i].next;
108                 ++j; /* next character */
109                 break;
110             }
111         } /* end for over edges */
112     }
113     if (a.state[snr].type==StateSuccess) {
114         res.start=pos;
115         res.end=j;
116         return res;
117     }
118 } /* end for over text */
119 res.start=-1;
120 res.end=-1;
121 return res;
122 }

```

\	BACKSLASH
\\	REAL BACKSLASH
\\\	REAL REAL BACKSLASH
\\\\	ACTUAL BACKSLASH, FOR REAL THIS TIME
\\\\\\	ELDER BACKSLASH
\\\\\\\\	BACKSLASH WHICH ESCAPES THE SCREEN AND ENTERS YOUR BRAIN
\\\\\\\\\\	BACKSLASH SO REAL IT TRANSCENDS TIME AND SPACE
\\\\\\\\\\\\	BACKSLASH TO END ALL OTHER TEXT
\\\\\\\\\\\\\\...	THE TRUE NAME OF BAYAL, THE SOUL-EATER

Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1638/>

Ausgabe

- Durch den neuen Typ des Suchergebnisses durch die unbekannte Länge des gefundenen Ausdrucks gibt es eine explizite Start- und Endmarkierung muss auch die Ausgabe modifiziert werden

```

139 void presentresult(searchresult_t res, const char* str) {
140     int start, prelen;
141     start=res.start>20? res.start-20 : 0; /* start where? */
142     prelen=res.start>20? 20 : res.start; /* length of prefix */
143     printf("%.s[%.s]%.20s\n",prelen, &str[start],
144           res.end-res.start, &str[res.start],&str[res.end]);
145 }

```

main()-Funktion

- Die main()-Funktion ist schließlich ähnlich der der bisherigen Suchprogramm-Varianten
- Das komplette Listing ist im Anhang D.1.2 zu finden

```

55 int main(int argc, char* argv[]) {
56     FILE* file;
57     char* text;
58     char* pattern;
59     automata_t automata;
60     searchresult_t found;
61     size_t size;
62
63     /* ... error checks and file/memory
64      * handling as in simple search ... */
65
66     text[size]='\0';
67     pattern = sanitize(argv[1]);
68     automata= init_automata(pattern);
69
70     found= search(automata,text,size);
71     if (found.start != -1)
72         presentresult(found,text);
73     free(text);
74     free(automata.state);
75     return found.start;
76 }

```


Diskussion

- Der Algorithmus hat – wie die einfache Suche – eine Komplexität $\mathcal{O}(m \cdot n)$, wobei n die Größe des zu durchsuchenden Textes und m die **maximale** (expandierte) Länge des Suchmusters ist
 - Da im schlechtesten Fall bei Vorkommen von „*“ im Suchmuster die expandierte Länge die des zu durchsuchenden Textes ist, ist die Komplexität $\mathcal{O}(n^2)$
 - Daher wird in der Praxis meist vereinbart, dass Suchen nur zeilenweise durchgeführt werden, ein Zeilenende nicht „gematcht“ wird oder es eine maximale Expansionslänge gibt
- Wildcard-Suchalgorithmen werden häufig nicht über den hier dargestellten Automatenansatz realisiert, sondern mit Hilfe **rekursiver** Funktionen

Knobelaufgaben

Aufgabe 13.1

Gegeben sei eine sequentielle Datei mit maximal 4 Milliarden 32-Bit-Integerzahlen in zufälliger Reihenfolge. Sie sollen einen effizienten Algorithmus entwickeln, der eine Zahl findet, die **nicht** in dieser Datei vorkommt. Sie dürfen zwar davon ausgehen, dass Sie viel RAM-Speicher zur Verfügung haben, aber auf jeden Fall weniger als 3 GByte.

Bedenken Sie, dass ein Zugriff auf eine Datei 3 – 6 Zehnerpotenzen langsamer ist, als ein Zugriff auf den Speicher. Versuchen Sie deshalb, die Anzahl der Filezugriffe zu minimieren.

Aufgabe 13.2

Eine Kachelwand ist 4×6 Kacheln groß. Es gibt zwei Sorten von Kachel, gelbe und blaue. Kacheln Sie die Wand so, dass jedes Rechteck aus Kacheln, bei dem jede Seite mindestens zwei Kacheln lang ist, keine vier gleichfarbigen Ecken hat!

Kapitel 14

Let's play

The only legitimate use of a computer is to play games.

— Eugene Jarvis

A strange game. The only winning move is not to play.

— WOPR im Film „Wargames“

14.1 Einführung

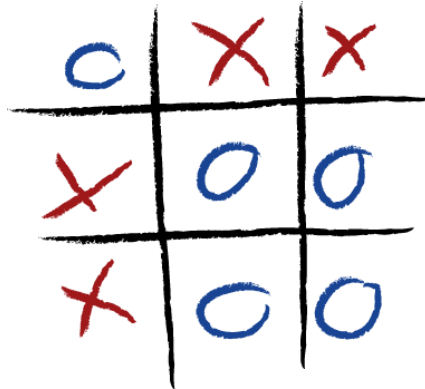
- Um die bisher gelernten Fähigkeiten im Zusammenhang anzuwenden, soll in diesem Kapitel ein größeres Projekt betrachtet werden
- Wir entwickeln ein **Spiel**

Achtung

In diesem Kapitel soll noch einmal der Entwicklungsvorgang illustriert werden. Deshalb werden auch vorläufige und falsche Lösungen präsentiert.

TicTacToe

- Spiel **Tic Tac Toe**
- **Regeln** (falls die jemand nicht kennt 🤔):
 - Spielfeld ist ein Gitter mit 3×3 Feldern
 - Zwei Spieler setzen im Wechsel eigenes Symbol (meist Kreuz und Kreis) in ein Feld
 - Wer zuerst drei Symbole in einer Reihe oder Diagonalen hat, hat gewonnen
- Der Computer soll ein „Spieler“ sein



Grobstruktur

- TicTacToe ist ein rundenbasiertes Spiel
- Egal wer dran ist, es sollte immer der aktuelle Spielstand angezeigt werden
- Es wird bis zum Sieg oder Unentschieden gespielt
- Dies ergibt folgenden Grobalgorithmus:

Algorithm TTT-GENERAL

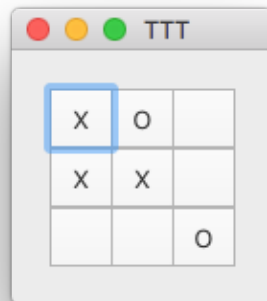
```

Input who will start → turn                                ▷ turn ∈ {player, computer}
repeat
  if turn = computer then
    Calculate move
    turn ← player
  else                                ▷ turn = player
    Input move
    turn ← computer
  end if
  Display move
until (somebody won) ∨ (draw)

```

Nutzerinterface

- Moderne Programme besitzen in der Regel eine graphische Nutzerschnittstelle (**GUI** - **graphical user interface**)



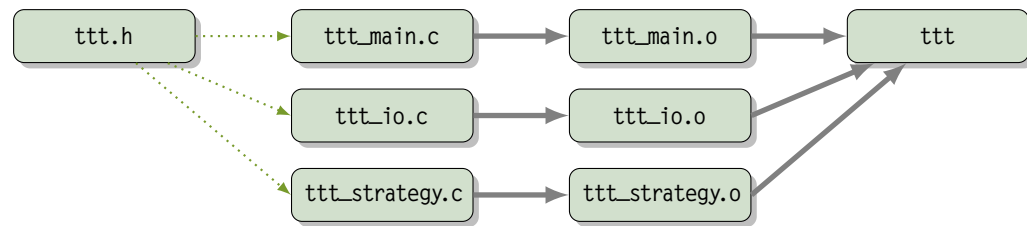
- Wir wollen aber ein Konsolenprogramm (Terminal) schreiben
- Wenn wir uns an den POSIX-Standard halten, läuft das Programm auf allen gängigen Betriebssystemen (sogar MS Windows 😊) ➔ **Portabilität**
- Auf dem Terminal könnte das Spiel etwa so aussehen:

```
x|o|x
-+-
x|o|
-+-
o| |x
```

- Das ist nicht ganz so schön, aber erst einmal etwas einfacher und portabler
- Um eine spätere GUI nicht zu „verbauen“, sollte das Nutzerinterface so weit wie möglich von der anderen Programmlogik getrennt sein

Modularisierung

- Es liegt nahe, das Projekt in drei Module zu zerlegen:
 - „`ttt_main.c`“, das als Überbau fungiert und auch die `main`-Methode bereitstellt
 - „`ttt_io.c`“, das sich um die Ein- und Ausgabe kümmert
 - „`ttt_strategie.c`“, das für die Berechnung des Computerzuges verantwortlich ist
- Damit alle Module auf gemeinsame Typen und Interfaces zugreifen können, soll es eine gemeinsame Header-Datei „`ttt.h`“ geben



Exkurs: Make

- Wenn ein Projekt aus mehreren Modulen besteht, muss jedes einzeln zu einem Objektfile übersetzt (z.B. „gcc -c <file.c>“) und anschließend alle Objektdateien (mit den Bibliotheken) verlinkt werden
 - „Händische Ausführung“ ist umständlich
 - ⇒ Batch-Job (Shellskript)
- Wird ein **einzelnes** Quellfile geändert, ist **nur** seine Übersetzung und das Linken zu wiederholen
 - ⇒ Shellskript **ungeeignet**
- **Lösung: Buildsysteme**, die Abhängigkeiten berücksichtigen
- Betrachten das weitverbreitetste System: **make**
- **make** ist ein kleines Hilfsprogramm, das andere Programme **in Abhängigkeit von bestimmten Bedingungen** ausführt
- Dazu interpretiert **make** eine (einfache und namenlose) Programmiersprache, die in einem **Makefile** beschrieben wird
 - Die Make-Sprache folgt (in erster Linie) dem **Konditionalmodell** (siehe Kapitel 2)
 - Entsprechend besteht ein Makefile aus **Regeln**
 - Regeln beschreiben **Abhängigkeiten**
 - Außerdem gibt es **Variablen** und **Funktionen**

Variablen

<name>:= <wert> oder
 <name>= <wert>

• Beispiel:

```
CC:= cc
```

- Zugriff mit \$(CC) oder \${CC}
- Variante ohne Doppelpunkt („:=“) lässt Rekursion bei der Namensauflösung zu

Funktionen

```
$(<funktionsname> <arg 1>, <arg 2>, ...) oder
${<funktionsname> <arg 1>, <arg 2>, ...}
```

• Beispiele:

```
SOURCEFILES= $(wildcard ttt*.c)
OBJECTFILES= ${subst .c,.o,${SOURCEFILES}}
```

• Meist zur Textmanipulation

• Beispiele für Funktionen:

```
$(subst <from>,<to>, <text>)
```

Ersetzt in *<text>* alle Vorkommen von *<from>* durch *<to>*

```
$(addprefix <prefix>,<list>) / $(addsuffix <suffix>,<list>)
```

Fügt jedem Wort der Liste *<list>* den Prä- bzw. Suffix hinzu

```
$(join <list1>,<list2>)
```

Fügt die Listen wortweise zu einer gemeinsamen Liste zusammen

```
$(foreach <var>,<list>,<text>)
```

Kreiert für jeden Wert in *<list>* eine neue Instanz von *<text>*, in der jedes Vorkommen von *<var>* durch den Listenwert ersetzt wird

```
$(shell <command>)
```

Führt den Befehl *<command>* in einer Shell aus

• Mehr in der Dokumentation von „make“

Regeln

```
<zieldatei>: <quelldateien>↵
↳ <Befehle zur Generierung der Zieldatei>
```

• Beispiel:

```
euclid.o: euclid.c
$(CC) -c euclid.c -o euclid
```

• Regeln können auch für **Klassen** von Dateien geschrieben werden ➡ „**implizite Regeln**“ oder „**Musterregeln**“

- Prozentzeichen („%“) dient dabei als „Joker“
- Spezielle (**automatische**) Variablen wie „\$*“ oder „\$@“ beziehen sich auf (Teile von) Zielen oder Vorbedingungen

- **Beispiel:**

```
%.o: %.c
$(CC) -c $< -o $@
```

- Einige automatische Variablen für Regeln:

- `$@`: Ziel einer Regel
- `$<`: Name der ersten Vorbedingung
- `$?`: Liste aller Vorbedingungen, die sich geändert haben
- `$^`: Liste aller Vorbedingungen
- `${@D}` bzw. `${<D}`: Verzeichnis des Ziels bzw. der ersten Vorbedingung
- `${@F}` bzw. `${<F}`: Dateiname (ohne Verzeichnis) des Ziels bzw. der ersten Vorbedingung

- Bei Aufruf von `make`

```
> make <ziel>
```

startet `make` alle Aktivitäten, die entsprechend der Datei „`Makefile`“ zur Aktualisierung von `<ziel>` notwendig sind

- Ob eine Datei aktuell ist, wird anhand des **Zeitstempels** der Datei entschieden
- Wenn `<ziel>` weggelassen wird, wird das Ziel der **ersten** Regel in „`Makefile`“ benutzt
- Regeln werden entsprechend ihrer Abhängigkeiten abgearbeitet

```
# Example Makefile
a: b
c: e f
   generates c from e and f
b: c d
   generates b from c and d
```

- In diesem Fall bei `make a`:

- „d“, „e“ und „f“ müssen vorhanden sein
- Zuerst wird „c“ aus „e“ und „f“ generiert/aktualisiert
- Dann wird „b“ aus „c“ und „d“ generiert/aktualisiert
- Da es für „a“ keinen Ausführungsteil gibt, ist hier die Abarbeitung beendet

- Beispielsweise könnte die Datei „`Makefile`“ für unser TicTacToe-Projekt so aussehen:

PAGE 3

DEPARTMENT	COURSE	DESCRIPTION	PREREQS
COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432

Quelle: *xkcd* - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/754/>

```

1 PROG=ttt
2 CC=cc
3 CFLAGS= -std=c99 -Wpedantic -Wall -Wextra
4 LDFLAGS=
5
6 HEADER= $(wildcard ttt*.h)
7 SOURCEFILES= $(wildcard ttt_*.c)
8 OBJECTFILES= $(subst .c,.o,$(SOURCEFILES))
9
10 $(PROG): $(OBJECTFILES)
11         $(CC) $(LDFLAGS) $(OBJECTFILES) -o $(PROG)
12
13 %.o: %.c
14         $(CC) -c $(CFLAGS) $<
15
16 .PHONY: clean
17 clean:
18         rm -f $(PROG) *.o

```

14.2 Die Hauptschleife und Ein-/Ausgabe

- Betrachten Hauptschleife
- Bevor wir uns an den ersten C-Code machen, überlegen wir, ob noch mehr Eigenschaften gefragt sind
 - Erkennung, ob Gewinn oder Unentschieden ➔ vermutlich eng mit Strategie verwandt, sollte dort bereit gestellt werden
 - Abbruch auch während des Spiels ➔ muss in Hauptschleife ausgewertet werden
 - Mehrere Spiele in Serie ➔ Hauptschleife
- Verfeinern Grobstruktur

Hauptschleife

Algorithm TTT-GENERAL (2. VERSION)

```

repeat
  Input who will start  $\rightarrow turn$   $\triangleright turn \in \{player, computer\}$ 
  repeat
    if  $turn = computer$  then
      Calculate move
       $turn \leftarrow player$ 
    else  $\triangleright turn = player$ 
      Input move  $\triangleright$  Abort is special move
       $turn \leftarrow computer$ 
    end if
    Display move
  until (somebody won)  $\vee$  (draw)  $\vee$  (abort)
  Input if another game?
until not another game

```

Hauptmodul

- Die Hauptschleife in C könnte dann etwa so aussehen

```

8  do {      /* main loop */
9      int player = ttt_x_or_o();           // select symbol (x or o)
10     bool computer_turn= (player == 'o');
11     ttt_init_board(board);               // erase playground
12     int move, assessment;
13     do { /* main loop for a game instance */
14         ttt_update_display(board);        // display board
15         move = computer_turn?             // computer's move?
16             ttt_calculate_move(board, ttt_opponent(player)): // calculate
17             ttt_input_move(board);        // player's move
18         if (move != TTT_ABORT)            // continue?
19             board[move] = computer_turn?  // computer's move?
20             ttt_opponent(player) : player; // apply move
21         computer_turn=!computer_turn;     // change the turn
22         assessment = ttt_won_or_draw(board,player); // game over?
23         if (assessment != TTT_UNDECIDED) {
24             ttt_update_display(board);    // display final board
25             ttt_output_result(assessment); // print result
26         }
27     } while ((move != TTT_ABORT) && (assessment == TTT_UNDECIDED));
28 } while(ttt_another_game() == true);

```

- Man beachte, dass wir Funktionen nutzen, die bisher unbekannt sind
- Diese sollen z.T. von anderen Modulen zur Verfügung gestellt werden

Best Practice

Namen, die global sichtbar sind (z.B. nicht-statische Funktionen) sollten einen Präfix bekommen, der das Projekt und möglicherweise auch das Modul beschreibt.

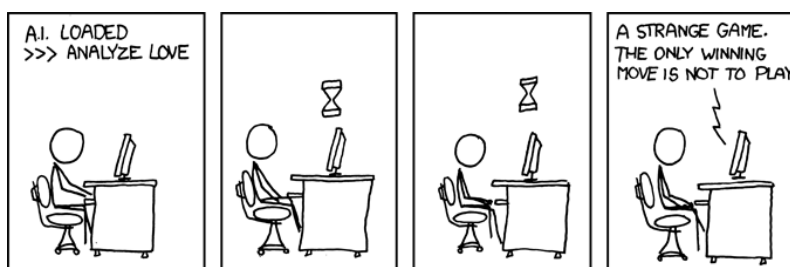
Auf diese Weise wird ein Namenskonflikt unwahrscheinlicher.

- In unserem Fall: „ttt_“ als Präfix

Datenstruktur

- Wie sollen die Daten in C dargestellt werden?
- Naheliegend für Spielsituation: multidimensionales Array (⇒ `int field[3][3];`)
- **Aber:**
 - Stets zwei Indizes notwendig
 - ⇒ Suche braucht dann stets zwei Schleifen
 - **Außerdem:** Situation soll vermutlich an Funktion übergeben werden ⇒ Array-Zerfall bei geschachteltem Array noch unübersichtlicher
- ⇒ Nutzen „normales“ Array (`int board[9]`, bzw. `typedef int ttt_board_t[NUMBER_OF_FIELDS];`)
 - Feldzuordnung:

0	1	2
3	4	5
6	7	8



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/601/>

Interface

- Deklarationen von Typen und Konstanten kommen in das gemeinsame Header-File

```

2  #include<stdbool.h>
3
4  /* general constants */
5  typedef enum {TTT_ABORT=-1, NUMBER_OF_FIELDS=9} ttt_constant_t;
6
7  /* game outcomes */
8  typedef enum {TTT_PLAYER_WINS, TTT_COMPUTER_WINS, TTT_DRAW,
9               TTT_UNDECIDED} ttt_result_t;
10
11 typedef int ttt_board_t[NUMBER_OF_FIELDS];

```

- ... genauso wie die Prototypen der (noch zu schreibenden) C-Funktion, die aus anderen Modulen aufgerufen werden

Hauptmodul

- Die beiden verbleibenden Funktionen im Hauptmodul sind relativ simpel:

```

33 void ttt_init_board(ttt_board_t f)
34 {
35     /* empties every field */
36     int i;
37     for(i=0; i<NUMBER_OF_FIELDS; i++)
38         f[i]=' ';
39 }
40
41 char ttt_opponent(char symbol)
42 {
43     switch(symbol)
44     {
45         case 'x': return 'o';
46         case 'o': return 'x';
47         default: return ' ';
48     }
49 }

```

Ein- und Ausgabe

- Es ist häufig so, dass das Nutzerinterface am aufwändigsten ist
- In diesem Fall nicht ➔ Textinterface

- **Problem:** Trotzdem etwas Bequemlichkeit erzielen
- **Beispiel:** Bildschirm löschen
 - Rausscrollen ➡ Fenstergröße muss bekannt sein
 - Unix-Befehl „clear“ ➡ **sehr** langsam und nicht portabel
 - ANSI/VT100-Steuerzeichen ➡ auch nicht 100% portabel
 - Beste Lösung: Nutzung von portablen Bibliotheken
 - * Hier z.B.: ncurses (für Windows auch: pdcurses)

```

1  /* Use VT100 ESC code to clean terminal */
2  static void ttt_clean_terminal(void)
3  {
4      printf( "%c[2J", 27 );
5  }

```

- Wir nutzen für TTT der Einfachheit halber aber die Lösung mit den Steuerzeichen 😊
- Die Bibliotheksfunktion „getchar()“ liest einen Tastendruck ein und gibt den Integer(!)wert zurück
- **Problem:**
 - Das Terminal gibt eine Eingabe erst weiter, wenn <RETURN> gedrückt wurde
 - Das Zeichen „↵“ ist Teil des Eingabestroms
- **Lösung:**
 - Die Funktion wird zweimal aufgerufen und das zweite Ergebnis verworfen
- Beispiel:

```

75 bool ttt_another_game()
76 {
77     int input;
78     printf("Do you want to play another game [y/n] ->");
79     input=getchar(); getchar();
80     if ((input=='Y') || (input=='y'))
81         return true;
82     else
83         return false;
84 }

```

Anmerkung

Diese Lösung ist nicht sehr elegant und auch nicht portable. Bei einem anderen Terminalmodus (siehe man `termios`) funktioniert die Lösung **nicht** mehr!

- Eine andere Alternative wäre die Änderung des Terminal-Modes (nicht portabel) oder wieder die Nutzung einer Bibliothek (z.B. wieder `ncurses`)
- Bei der Zueingabe überprüfen wir gleich die Korrektheit der Eingabe:

```

45 int ttt_input_move(const ttt_board_t board)
46 {
47     int input;
48     printf("\nPlease enter the number of the field you want to occupy
49           (0 for abort)\n");
50     ttt_display_board(numberfield);
51     printf("Your move ->");
52     do{
53         do {
54             input=getchar(); getchar();
55         } while ((input<'0') || (input>'9'));
56         if (input=='0') return TTT_ABORT;
57         input=input-'1'; // character => integer index
58     } while (board[input]!=' ');
59     return input;
60 }

```

- Für die Ausgabe wird ein wenig „ASCII-Art“ gezeichnet:

```

14 void ttt_display_board(const ttt_board_t board)
15 {
16     for(int i=0;i<3;i++){
17         if(i) // not for first row
18             printf("\n  +-+-");
19         printf("\n  ");
20         for(int j=0;j<3;j++){
21             if(j) printf("|"); // not for first column
22             printf("%c",board[3*i+j]);
23         }
24     }
25     printf("\n");
26 }

```

- Das sieht dann z.B. so aus:


```

1  /* NOT in final version */
2  int winning_move(const ttt_board_t f, char s)
3  {
4      /* triplet (0,1,2) */
5      if((f[0]==s)&&(f[1]==s)&&(f[2]==' ')) return 2;
6      if((f[1]==s)&&(f[2]==s)&&(f[0]==' ')) return 0;
7      if((f[0]==s)&&(f[2]==s)&&(f[1]==' ')) return 1;
8
9      /* triplet (3,4,5) */
10     if((f[3]==s)&&(f[4]==s)&&(f[5]==' ')) return 5;
11     if((f[4]==s)&&(f[5]==s)&&(f[3]==' ')) return 3;
12     if((f[3]==s)&&(f[5]==s)&&(f[4]==' ')) return 4;
13
14     :
15
44    /* triplet (2,4,6) */
45    if((f[2]==s)&&(f[4]==s)&&(f[6]==' ')) return 6;
46    if((f[4]==s)&&(f[6]==s)&&(f[2]==' ')) return 2;
47    if((f[2]==s)&&(f[6]==s)&&(f[4]==' ')) return 4;
48    return -1;
49 }

```

- Eine solche Funktion (obgleich korrekt) ist unnötig groß und somit fehleranfällig
- **Idee:** Nutzung von Daten und Indirektionen

```

1  /* NOT in final version */
2  enum {NUMBER_TRIPLES=8, NONE=-1};
3  typedef int ttt_triple_t[3];
4  const ttt_triple_t triples[NUMBER_TRIPLES]={
5      {0,1,2},{3,4,5},{6,7,8},{0,3,6},{1,4,7},{2,5,8},{0,4,8},{2,4,6}
6  };
7
8  int winning_move(const ttt_board_t f, char s) {
9      for(int i=0; i<NUMBER_TRIPLES;++i)
10         for(int j=0; j<3; ++j) {
11             int idx2=(j+1)%3; /* 3 => 0 */
12             int idx3=(j+2)%3; /* 3 => 0; 4 => 1 */
13             if ((f[triples[i][j]]==' ') &&
14                 (f[triples[i][idx2]]==s) && (f[triples[i][idx3]]==s))
15                 return triples[i];
16         }
17     return NONE;
18 }

```

- Für den Schritt „Can opponent win?“ könnte die Funktion noch einmal genutzt werden:


```

1 /* NOT in final version */
2 int block_opponent_winning_move(const ttt_board_t f, char s)
3 {
4     return winning_move(f, ttt_opponent(s));
5 }

```

- Damit zeichnet sich eine Struktur ab: Es werden mehrere Funktionen benötigt, die die Stellung bewerten, ähnlich zu `winning_move()`
- **Idee:** Allgemeinere Bewertungsfunktion

Allgemeine Bewertung

- Was könnte von Interesse sein?

- Triple gewonnen ⇒ Feststellung des Spielendes:

X	.	.
.	X	.
.	.	X

- Triple kann zum Sieg führen ⇒ `winning_move()`

.	.	.
.	X	.
.	.	X

- Triple „bisher meins“ ⇒ könnte Gegner zwingen

X	.	.
.	.	.
.	.	.

- Triple ist nutzlos

.	.	.
.	O	.
.	.	X

oder

O	.	.
.	X	.
.	.	X

oder

O	.	.
.	X	.
.	.	O

- Triple leer

.	.	.
.	.	.
.	.	.

- Die ersten drei Bewertungen entsprechend für gegnerisches Symbol
- Wie können verschiedene Bewertungen nach einem gemeinsamen Schema durchgeführt werden?
- **Idee:** Nutzung von Primzahlen ($\neq 3$)
 - Eigenes Symbol zählt 2
 - Gegnersymbol zählt 5
 - Leeres Feld zählt 0
- Felder eines Triples werden addiert ⇒
 - Eigener Sieg ⇒ $\sum = 6$
 - Sieg Gegner ⇒ $\sum = 15$
 - Eigener Sieg möglich ⇒ $\sum = 4$
 - Gegnerischer Sieg möglich ⇒ $\sum = 10$

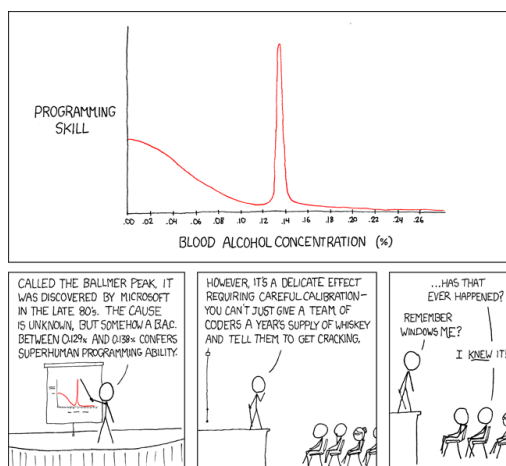
- Mein Triple $\Rightarrow \Sigma = 2$
- Gegner-Triple $\Rightarrow \Sigma = 5$
- Leeres Triple $\Rightarrow \Sigma = 0$
- Nutzloses Triple $\Rightarrow \Sigma = 7 \vee \Sigma = 9 \vee \Sigma = 12$

- Bei einem solchen Ansatz muss jedes Triple nur einmal evaluiert werden
- **Nachteil:** Es muss anschließend **im** Triple nach dem leeren Feld gesucht werden ➡ nehmen wir in Kauf, da nur einmal ausgeführt

```

1 enum {NUMBER_TRIPLES=8, I_KEY=2, OPP_KEY=5}; // more, if needed...
2
3 typedef int ttt_triple_t[3];
4 const ttt_triple_t triples[NUMBER_TRIPLES]={
5     {0,1,2},{3,4,5},{6,7,8},{0,3,6},{1,4,7},{2,5,8},{0,4,8},{2,4,6}
6 };
7
8 int ttt_evaluate(const ttt_board_t field, int tnr, int my_symbol) {
9     int val=0;
10    int opp_symbol=ttt_opponent(my_symbol);
11    for (int i=0; i<3; ++i) {
12        if (field[triples[tnr][i]]==my_symbol)
13            val+=I_KEY;
14        else if (field[triples[tnr][i]]==opp_symbol)
15            val+=OPP_KEY;
16    }
17    return val;
18 }

```



Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/323/>

- Die Computerstrategie sähe unter Nutzung von `ttt_evaluate()` bis jetzt so aus:

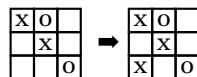
```

1 typedef int ttt_triplev_t[NUMBER_TRIPLES];
2
3 static int winning_move(const ttt_board_t board,
4                        const ttt_triplev_t eval, int who) {
5     for (int i=0; i<NUMBER_TRIPLES; i++)
6         if (eval[i]==who) // find winning triple
7             for(int j=0;j<3;j++) // find the empty place
8                 if (board[triples[i][j]]==' ') return triples[i][j];
9     return NONE;
10 }
11
12 int ttt_calculate_move(const ttt_board_t board, char symbol) {
13     ttt_triplev_t eval;
14     for (int i=0; i< NUMBER_TRIPLES; ++i)
15         eval[i]=ttt_evaluate(field,i,symbol);
16     int move=winning_move(board,eval,I_CANWIN);
17     if (move!=NONE) return move;
18     move=winning_move(board,eval,OPP_CANWIN);
19     if (move!=NONE) return move;
20     /* yet to implement */
21     return NONE;
22 }

```

Gewinn vorbereiten

- Betrachten wir noch einmal die Grobstrategie
- Wie kann ein Computer-Gewinn im **nächsten** Zug vorbereitet werden?
- **Idee: Zwei** gewinnbringende Triples vorbereiten ➡ **Gabelung**
- **Beispiel** (Computer hat x):



- In diesem Fall kann der o-Spieler nur **ein** Triple blockieren
- Das gemeinsame Feld soll **Achsfeld** (engl.: pivot element) genannt werden
- Um ein Achsfeld zu finden, müssen folgende Bedingungen erfüllt sein
 - Es muss zwei Triples geben, die bisher ausschließlich dem Computer gehören ➡ Bewertung: 2
 - Diese müssen ein **gemeinsames leeres** Element besitzen

- **Vorgehen:**

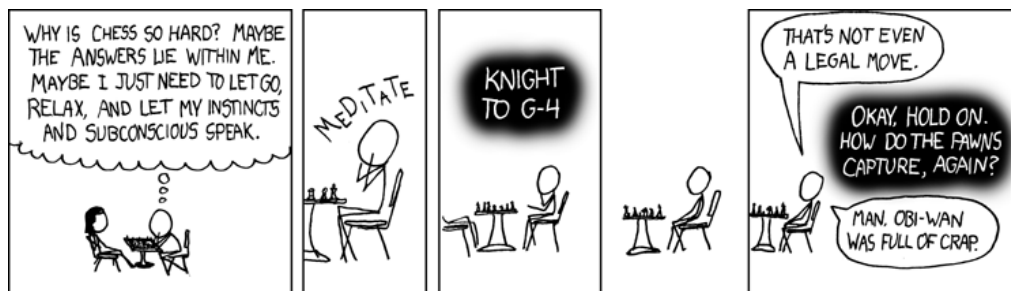
- Wir betrachten alle acht Gewinntripel und untersuchen, ob sie jeweils mit 2 bewertet wurde
- Jedes Spielfeld, das in jeweils einem dieser Tripel **leer** vorkommt, erhält einen Punkt
- Ein Spielfeld, das so mehr als einen Punkt erhalten hat, ist ein Achselement

- Im Array `singles[NUMBER_OF_FIELDS]` werden die „Punkte“ gezählt

```

1 int forking_move(const ttt_playground_t field, const ttt_triplev_t
    eval)
2 {
3     int i,j=0;
4     int singles[NUMBER_OF_FIELDS]={0,0,0,0,0,0,0,0,0,0};
5     for (i=0; i<NUMBER_TRIPLES; ++i)
6     {
7         if (eval[i]==I_KEY)
8         {
9             for(j=0;j<3;j++)
10                if (field[triples[i][j]]==' ')
11                    ++singles[triples[i][j]];
12        }
13    }
14    for(i=0; i< NUMBER_OF_FIELDS; ++i)
15        if (singles[i]>=2) return i;
16    return NONE;
17 }

```



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/232/>

Offensive

- Angenommen, unsere Strategie hat bisher keinen Zug ergeben
- ~~Idée:~~ Wie schon beim Sieges-Zug: Strategie spiegeln und verhindern

- **Problem:** Gegner könnte zwei Achselemente besitzen

Algorithm TTTCHOOSEMOVE (2ND VERSION)

```

if Can I win? then
    Choose winning move
else
    if Can opponent win? then
        Block winning move
    else
        if Can I fork? then
            Choose pivot element
        else
            if Can opponent fork? then
                Block opponent's pivot element
            else
                Whatever
            end if
        end if
    end if
end if

```

- **Alternative:** In die Offensive gehen
 - ➔ Zwingen, kommenden Sieg zu blockieren
- **Problem:** Gegner darf nicht in eine Gabel gezwungen werden
- **Beispiel** (aus Sicht von „x“):

–

O	X	
	O	
		X

- Es darf nicht auf Feld 8 gesetzt werden, weil dies zu gegnerischer Gabel führt

➔

O	X	
	O	
O	X	X

```

int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
              // guaranteed to be random.
}

```

Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/221/>

- Die Strategie sieht jetzt so aus:

Algorithm TTTCHOOSEMOVE (3RD VERSION)

```

if Can I win? then
    Choose winning move
else
    if Can opponent win? then
        Block winning move
    else
        if Can I fork? then
            Choose pivot element
        else
            if Can I force without opponent's fork? then
                Choose forcing move
            else
                Move to best available place
            end if
        end if
    end if
end if

```



MY HOBBY: BUILDING PLASTIC POKÉMON WITH SUBTLE UNDERLIGHTING
AND A GYROSCOPE TO MAKE THEM DRIFT BACK AND FORTH, THEN
LEAVING THEM SITTING AROUND TO MESS WITH POKÉMON GO PLAYERS.

Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/1705/>

- Um den Offensiv-Zug zu finden, untersuchen wir zunächst, wohin wir **nicht** ziehen dürfen
➔ Verbleibendes Feld ist Achselemente des Gegners
- Dann suchen wir Triples, in denen **genau ein** eigenes (und **kein** gegnerisches) Zeichen ist ➔ Bewertung: 2
- Wenn dieses Triple **kein** gegnerisches Achselement enthält, wähle ein leeres Feld als Offensivzug

- Sonst wähle ein Achselement des Gegners

```

1 int forcing_move(const ttt_board_t board, const ttt_triplev_t
  eval) {
2     int opp_singles[NUMBER_OF_FIELDS];
3     for (int i=0; i<NUMBER_TRIPLES; ++i) {
4         if (eval[i]==OPP_KEY) {
5             for(int j=0;j<3;j++)
6                 if (board[triples[i][j]]==' ')
7                     ++opp_singles[triples[i][j]];
8         }
9     }
10    for (int i=0; i<NUMBER_TRIPLES; ++i) {
11        if (eval[i]==I_KEY) {
12            if((opp_singles[triples[i][0]]<=1) &&
13                (opp_singles[triples[i][1]]<=1) &&
14                (opp_singles[triples[i][2]]<=1))
15                for(j=0;j<3;j++)
16                    if (board[triples[i][j]]==' ')
17                        return triples[i][j];
18            for(j=0;j<3;j++)
19                if(opp_singles[triples[i][j]]>1)
20                    return triples[i][j];
21        }
22    }
23    return NONE;
24 }

```

Was bleibt

- Wenn alles andere fehlschlägt, wählen wir irgendein Feld
- Unterschiedliche Wertigkeiten: Mitte ➡ Ecke ➡ Rand

```

1 int best_remaining_move(const ttt_board_t board)
2 {
3     const int best[]={4,0,2,6,8,1,3,5,7};
4
5     for(int i=0; i<NUMBER_OF_FIELDS; ++i)
6         if(board[best[i]]==' ') return best[i];
7     return NONE; /* Should never happen */
8 }

```

Gesamtstrategie

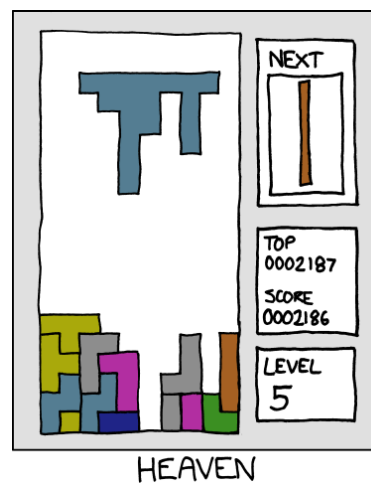
- Mit Hilfe der beschriebenen Funktionen der Zug relativ berechnet werden

```

122 int ttt_calculate_move(const ttt_board_t field, char symbol)
123 {
124     ttt_tripleval_t eval;
125     for (int i=0; i< NUMBER_TRIPLES; ++i)
126         eval[i]=evaluate(field,i,symbol);
127     int move=winning_move(field,eval,I_CANWIN);
128     if (move!=NONE) return move;
129     move=winning_move(field,eval,OPP_CANWIN);
130     if (move!=NONE) return move;
131     move=forking_move(field,eval);
132     if (move!=NONE) return move;
133     move=forcing_move(field,eval);
134     if (move!=NONE) return move;
135     return best_remaining_move(field);
136 }

```

- Nach „außen“ muss nur `ttt_calculate_move` (und die noch nicht betrachtete Funktion `ttt_won_or_draw`) sichtbar sein
- Deshalb deklarieren wir alle „internen“ Funktionen als `static`



Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/888/>

Spielende

- Die Entscheidung über das Spielende funktioniert ähnlich zur Zugentscheidung


```

138 ttt_result_t ttt_won_or_draw(const ttt_board_t board, char symbol)
139 {
140     bool undecided=false;
141     for(int i=0; i<NUMBER_TRIPLES; i++) {
142         int state=evaluate(board,i,'x');
143         switch (state) {
144             case I_WIN:
145                 if (symbol=='x') return TTT_PLAYER_WINS;
146                 else return TTT_COMPUTER_WINS;
147             case OPP_WIN:
148                 if (symbol=='o') return TTT_PLAYER_WINS;
149                 else return TTT_COMPUTER_WINS;
150             case DRAW_I: /* trough */
151             case DRAW_OPP: /* trough */
152                 break;
153             case DRAW: /* trough */
154             default:
155                 undecided=true;
156         }
157     }
158     if (undecided) return TTT_UNDECIDED;
159     else return TTT_DRAW;
160 }

```

Fertig!

- Damit ist das Tic-Tac-Toe-Programm im Wesentlichen fertiggestellt
- Im Anhang D.2 ist der Code des kompletten Programms dokumentiert

Alternativen

- Die hier dargestellte Strategie stellt **eine** Möglichkeit dar
- **Alternativen:**
 - Da es „nur“ 255168 verschiedene Spielverläufe gibt, könnte man alle vorher berechnen und jeweils einen Zug aussuchen, der zu einen günstigen Ausgang führt
 - Durch Transformation (Spiegelung und/oder Drehung) können die Endstellungen auf 138 verschiedene Möglichkeiten reduziert werden, so dass man den Zug so wählt, dass er zu einer günstigen Endstellung passt

Beispiel

Folgende Situationen sind äquivalent:

X	O	

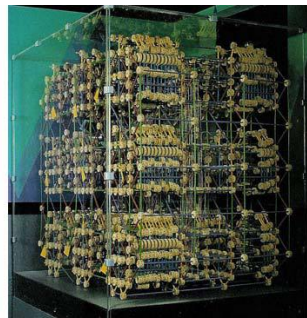
X		
O		

	O	X

O	X	

 etc.

TicTacToe-Computer



- Der TicTacToe-Computer am M.I.T., ausschließlich aus Tinkertoy[™]-Bausteinen gebaut
- Er arbeitet mit der zweiten Alternativstrategie

Knobelaufgaben

Aufgabe 14.1

12 Leute spielen Skat an vier Tischen, also jeweils drei Leute pro Tisch. Nach jeweils 12 Spielen werden die Gruppen neu gemischt. Insgesamt werden 60 Spiele gespielt. Ist es möglich, dass jeder mindestens einmal mit jedem anderen an einem Tisch saß?

Aufgabe 14.2

Auf einem Klassentreffen erzählt Anton Berta, dass er drei Kinder hat, alles Mädchen. Berta fragt wie alt diese sind. Anton antwortet: „Das Produkt der Alter ist 72, die Summe entspricht der Hausnummer, in der wir uns treffen.“

Nach kurzem überlegen antwortet Berta, dass sie noch nicht genug Information hat. Darauf Anton: „Ach ja, die Älteste hat als einzige im Sommer Geburtstag.“

Was ist jeweils das Alter der Mädchen?

Kapitel 15

Linker und Bibliotheken

Ich habe mir das Paradies immer als eine Art Bibliothek vorgestellt

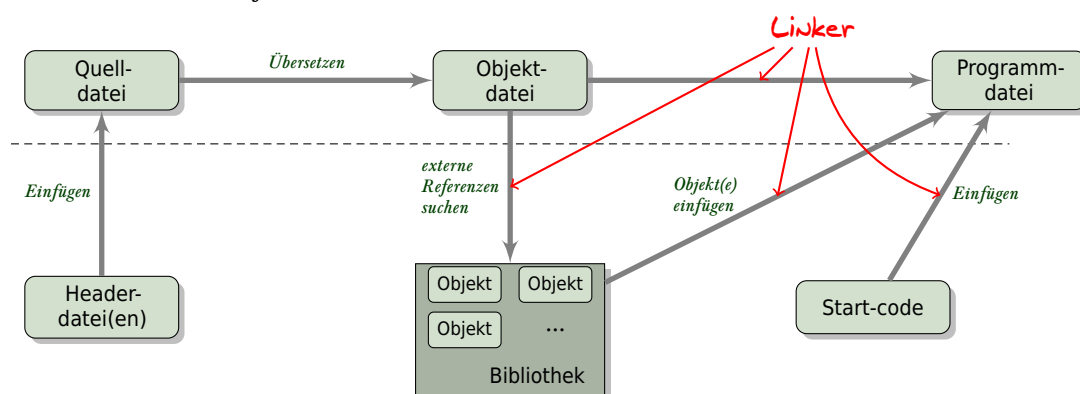
— Jorge Luis Borges

Every program is a part of some other program and rarely fits.

— Alan J. Perlis

15.1 Wiederholung: Module und Linker

- Im letzten Kapitel haben wir eine Anwendung in verschiedene Module zerlegt, die jeweils ein eigenes C-Quellfile hatten
- Erst der Linker hat die einzelnen Module zusammengefügt
- Wir wollen uns jetzt den Linker näher betrachten



- Als Beispiel betrachten wir folgenden Code einer allgemeinen Tausch-Funktion:

```

1 #include <stddef.h>           // for size_t
2 extern void* malloc(size_t);
3 extern void free(void*);
4
5 static void bytecopy(char *dest, char *src, size_t size){
6     while(size > 0){
7         *dest++ = *src++;
8         size--;
9     }
10 }
11
12 int swap(void *first, void *second, size_t size){
13     void *tmp = malloc(size);
14     if (!tmp) return 0;
15     bytecopy(tmp,first,size);
16     bytecopy(first,second,size);
17     bytecopy(second,tmp,size);
18     free(tmp);
19     return 1;
20 }

```

Die Funktion `memcpy(dest,src,size)` gehört zur Standardbibliothek und kopiert `size` Byte im Speicher von `src` nach `dest`.

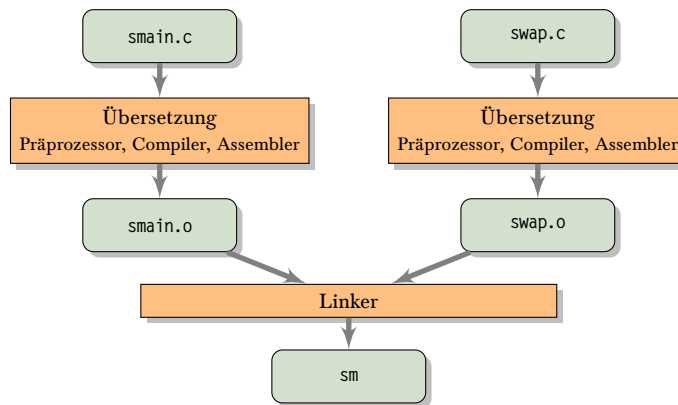
- Der Code kann mit verschiedenen Typen genutzt werden, wie folgendes Programm demonstriert:

```

1 #include <stdio.h>
2 int swap(void *first, void *second, size_t size);
3
4 char* str1="Hello";
5 char* str2="World";
6 int a,b;
7
8 int main(){
9     a=42;
10    b=23;
11    printf("1st string: %s, 2nd string: %s\n",str1,str2);
12    swap(&str1,&str2,sizeof(str1));
13    printf("1st string: %s, 2nd string: %s\n",str1,str2);
14    printf("1st integer: %d, 2nd integer: %d\n",a,b);
15    swap(&a,&b,sizeof(a));
16    printf("1st integer: %d, 2nd integer: %d\n",a,b);
17    return 0;
18 }

```

```
> cc -std=c99 -Wall euclid.c swap.c smain.c -o sm
```



```

1 > ./sm
2 1st string: Hello, 2nd string: World
3 1st string: World, 2nd string: Hello
4 1st integer: 42, 2nd integer: 23
5 1st integer: 23, 2nd integer: 42
6 >

```

Anmerkung

Der hier benutzte Code ist nicht unbedingt optimal. Z.B. entspricht die statische Funktion `bytecopy()` der Funktion `memcpy()` aus der Standardbibliothek. Mit Hilfe von Präprozessor-Makros und ggf. von Generics (Schlüsselwort `_Generic` ab C11) könnte man eine solche Funktion noch eleganter (ohne Übergabe der Größe) schreiben. In C++ werden dafür **Templates** benutzt.



Warum Linker?

- **Modularität**

- Programme können als Ansammlung kleinerer Quelldateien geschrieben werden, anstatt als eine monolithische Datei
- Es können Bibliotheken von häufig genutzten Funktionen geschrieben werden (mehr in Abschnitt 15.3), z.B. Standardbibliothek, Mathematik-Bibliothek, etc.

- **Effizienz**

- Zeit: Es müssen nicht immer der komplette Quellcode übersetzt werden
- Platz: Bibliotheken können eine Sammlung unterschiedlicher Funktionen enthalten, das fertige Programm enthält aber nur die Funktionen, die es benötigt

15.2 Linker

Was macht der Linker?

- **1. Auflösung von Symbolen**

- Programme definieren und referenzieren **Symbole**, d.h. Namen von Variablen und Funktionen

<code>int swap() {...}</code>	definiert Symbol <code>swap</code>
<code>swap(&a,&b,sizeof(a));</code>	referenziert Symbole <code>swap</code> , <code>a</code> und <code>b</code>
<code>int a;</code>	definiert Symbol <code>a</code>
- Symboldefinitionen werden vom Compiler in einer **Symboltabelle** gespeichert
 - * Die Symboltabelle ist ein Array einer speziellen Datenstruktur (`struct`), die für jedes Symbol den Namen, den Typ, die Größe und den Ort enthält
- Der Linker verknüpft jede Symbolreferenz mit genau einer Symboldefinition

- **2. Verschiebung (*Relocation*)**

- Fügt separate Code- und Datenabschnitte in gemeinsamen Abschnitten (*Sections*) zusammen
- Verschiebe Symbole von ihrer relativen Position im Object-File (.o) auf ihre finale absolute Speicherposition im ausführbaren File
- Korrigiert alle Referenzen auf diese Symbole, so dass diese auf die neuen Positionen zeigen

Linkersymbole

Es gibt drei Typen von Linkersymbolen

- **Globale Symbole**

- Symbole, die in einem Modul definiert sind und durch andere Module referenziert werden können (z.B. nicht-statische C-Funktionen oder nicht-statische globale Variablen)

- **Externe Symbole**

- Globale Symbole die in einem Modul referenziert werden, aber in einem anderen Modul definiert werden

- **Lokale Symbole**

- Symbole, die in einem Modul definiert und ausschließlich auch dort referenziert werden, z.B. Funktionen und globale Variablen die das `static`-Attribut haben

Merke

Lokale Variablen erzeugen **keine** Linkersymbole, auch keine lokalen.

```

1 #include <stddef.h>           // for size_t
2 extern void* malloc(size_t);
3 extern void free(void*);      extern
4
5 static void bytecopy(char *dest, char *src, size_t size){
6     while(size > 0){          lokal
7         *dest++ = *src++;
8         size--;
9     }
10 }                             global
11
12 int swap(void *first, void *second, size_t size){
13     void *tmp = malloc(size);
14     if (!tmp) return 0;
15     bytecopy(tmp, first, size);
16     bytecopy(first, second, size);
17     bytecopy(second, tmp, size);
18     free(tmp);
19     return 1;
20 }

```

Der Linker weiß nichts von tmp

- Mit Hilfe von Programmen wie `readelf`, `objdump` oder `nm` können die Symboltabellen ausgelesen werden

```

> nm -fs swap.o
Symbols from swap.o:

```

Name	Value	Class	Type	Size	Line	Section
bytecopy	0000000000000000	t	FUNC	000000000000003d		.text
free		U	NOTYPE			*UND*
malloc		U	NOTYPE			*UND*
swap	000000000000003d	T	FUNC	000000000000008a		.text

```

> nm -fs smain.o
Symbols from smain2.o:

```

Name	Value	Class	Type	Size	Line	Section
a	0000000000000004	C	OBJECT	0000000000000004		*COM*
b	0000000000000004	C	OBJECT	0000000000000004		*COM*
main	0000000000000000	T	FUNC	00000000000000c1		.text
		U	NOTYPE			*UND*
str1	0000000000000000	D	OBJECT	0000000000000008		.data
str2	0000000000000008	D	OBJECT	0000000000000008		.data
swap		U	NOTYPE			*UND*

```

>

```

Executable and Linkable Format (ELF)

- Es gibt verschiedene Datenformate für Objektdateien
 - Z.B.: a.out, COFF, Mach-O, PE
 - Betrachten hier ELF
- **ELF** – *Executable and Linkable Format*
- Ursprünglich von AT&T für System V entwickelt
- Heute bei vielen Betriebssystemen verwendet, u.a. BSD und Linux
- ELF wird für verschiedene Arten von Binärdateien eingesetzt:
 - Verschiebbare Objectfiles (.o)
 - Ausführbare Programmfiles
 - Geteilte Objectfiles (.so)

ELF header
Segment header table
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
section header table

- **ELF Header:** Wortgröße, Byteordnung, Dateityp, Plattform, etc.
- **Segment-Header-Tabelle:**
 - nötig für ausführbare Dateien
 - Seitengröße, Sections, Segmentgröße
- **.text:** (Maschinen-)Code
- **.rodata:** Nur-les-Daten (Sprungtabellen, ...)
- **.data:** initialisierte globale Variablen
- **.bss:** uninitialisierte globale Variablen¹
 - wird im Speicher mit 0 initialisiert
- **.symtab:** Symboltabelle
- **.rel.text:** Verschiebeinformation für Code

- Adressen von Befehlen, die beim Verschieben im Speicher modifiziert werden müssen
- Instruktionen für die Verschiebung
- **.rel.data**: Verschiebeinformation für Variablen
- **.debug**: Information für symbolisches Debugging (cc -g)
- **Section header table**: Offset und Größe für jeden Abschnitt

Verschiebung von Code und Daten

Verschiebbarer Objectcode

Startupcode	.text
Startupdaten	.data
nichtinitialisierte Startupdaten	.bss
main()	.text
char* str1="Hello"	.data
char* str2="World"	.data
int a,b;	.bss
bytecopy() swap()	.text

Ausführbarer Code

Header	
Startupcode	.text
main()	
bytecopy() swap()	
Weiterer Code	
Startupdaten	.data
char* str1="Hello"	
char* str2="World"	
nichtinitialisierte Startupdaten	.bss
int a,b	
Symboltabelle	
Debugging-Daten	

Das Problem globaler Variablen

- Beim Linken werden nur die Symbole betrachtet, es wird **keine** Typüberprüfung durchgeführt
- Betrachten folgenden Code:

¹.bss steht für *block started by symbol*, ein Pseudobefehl im Assembler für IBM 704

```

1 #include <stdio.h>
2
3 int a;
4 int b;
5
6 void printab1(){
7     printf("1: a=%d, b=%d\n",
8           a,b);
9 }
10
11 void setab1(){
12     a=42;
13     b=23;
14 }

```

```

1 #include <stdio.h>
2
3 double a;
4 int b;
5
6 void printab2(){
7     printf("2: a=%0.0f, b=%d\n",
8           a,b);
9 }
10
11 void setab2(){
12     a=42.0;
13     b=23;
14 }

```

- Die gemeinsame Verwendung führt jedoch zu Problemen:

```

1 void printab1();
2 void printab2();
3 void setab1();
4 void setab2();
5
6 int main(){
7     setab1();
8     printab1();
9     setab2();
10    printab2();
11    printab1();
12    return 0;
13 }

```

```

> cc -Wall -Wextra -c -o global1.o global1.c
> cc -Wall -Wextra -c -o global2.o global2.c
> cc -Wall -Wextra -c -o global.o global.c
> cc global1.o global2.o global.o -o global
>
> ./global
1: a=42, b=23
2: a=42, b=23
1: a=0, b=23
>

```

Oops!

- Daher sollten folgenden Regeln beachtet werden:

Wenn es möglich ist, **vermeiden** Sie globale Variablen!

- Andernfalls:
 - Nutzen Sie wenn möglich `static`!
 - Initialisieren Sie globale Variablen bei der Definition!
 - Nutzen Sie `extern` für externe Variablen!

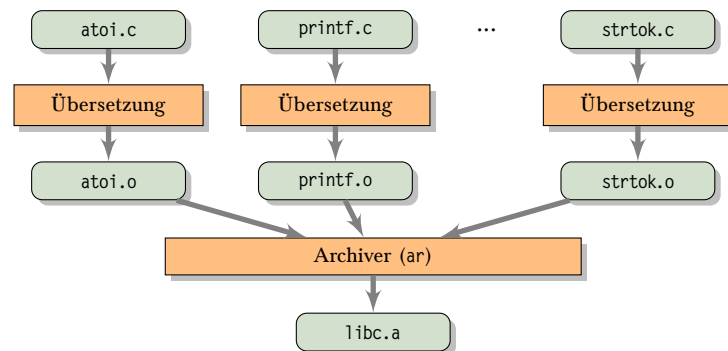
15.3 Bibliotheken

- Wie kann nachnutzbarer Code „paketierte“ werden?
 - Ein- und Ausgabe, Mathe-Funktionen, ...
- Mit den bisherigen Tool gibt es zwei Möglichkeiten:
 1. Alle Funktionen kommen in ein großes Quellfile und landen in einem großen Objectfile
 - ➔ Programmierer linken dieses große Objectfile zu ihren Programmen
 - ➔ Zeit- und platzineffizient
 2. Jede Funktion kommt in ein eigenes Quellfile
 - ➔ Programmierer linken explizit ausgesuchte Objectfiles zu ihren Programmen
 - ➔ Ist zwar effizienter, aber sehr aufwendig für den Programmierer

Lösung

- **Lösung:** Nutzung von **Archiven**, auch bekannt als (statische) **Bibliotheken**
- Archive ist eine einzelne Datei, die eine Sammlung von mehreren Objectfiles enthält, zusammen mit einem Index
- Linker wird so erweitert, dass er bei der Auflösung von unaufgelösten Referenzen in einem oder mehreren Archiven nachschaut
- Wenn ein Symbol in einem Archive gefunden wird, kopiert der Linker das entsprechende Objectfile und linkt es zum Programm
- Zum Erzeugen und Managen von Archiven gibt es ein eigenes Tool: `ar`

Erzeugung eines Archivs



```
> ar rs libc.a atoi.o printf.o ... strtok.o
```

- Der Archiver erlaubt inkrementelle Updates
- Das geänderte Quellfile wird compiliert und im Archiv ersetzt

Gemeinsam genutzte Bibliotheken

- Jedes System enthält eine Reihe Bibliotheken
- Der C-Standard verlangt mindestens zwei:
 - `libc.a` oder `libgcc.a`²: Standardbibliothek, enthält z.B. Ein- und Ausgabe, Speicher-verwaltung, Zeichenkettenoperationen, Signalbehandlung, ...
 - `libm.a`: Gleitkomma-Mathematik
- Auf dem OpenSubmit-Server besteht...
 - `libgcc.a` aus 221 Objectdateien, die 3 MByte einnehmen
 - `libm.a` aus 460 Objectdateien, die 2,1 MByte einnehmen
- Die Standardbibliothek (und Startup-Code) werden (solange nicht die Optionen „-nostartfiles“ bzw. „-nostartfiles“ angegeben werden) automatisch gelinkt

²GNU Compiler-Suite

Nutzung von (statischen) Bibliotheken

- Der Linker kann beliebige Archive (auch eigene) nutzen
- Optionen:
 - `-Lpfad`: Linker sucht in *pfad* nach Bibliotheken; die Option kann mehrmals angegeben werden
 - `-lname`: Linker durchsucht die Archiv-Datei *libname.a* bei der Symbolauflösung
- **Beispiel**: der folgende Aufruf sagt dem Linker, dass die Gleitkomma-Mathebibliothek eingebunden werden soll:

```
> cc -o myprog mycode.c -lm
```

- Eigene Bibliotheken machen werden genauso behandelt:

```
> cc -c swap.c
> ar rs libsw.a swap.o
> ar: creating libsw.a
> cc -o sm smain.c -L. -lsw
```

- Bei einigen Compilern (z.B. gcc) spielt die Reihenfolge der Optionen eine Rolle, da die Symbolauflösung strikt von links nach rechts erfolgt:

```
> gcc -o sm smain.c -L. -lsw
/tmp/cc3DPr0G.o: In function 'main':
smain2.c:(.text+0x48): undefined reference to 'swap'
smain2.c:(.text+0x99): undefined reference to 'swap'
collect2: error: ld returned 1 exit status
```

Dynamische Bibliotheken

- Statische Bibliotheken haben einige Nachteile:
 - Vervielfachung des Codes im Massenspeicher (die Standardbibliothek wird von [nahezu] jedem Programm genutzt)
 - Vervielfachung des Codes im Hauptspeicher
 - Kleine Fehlerbehebungen im Bibliothekscode erfordern ein explizites Neulinken **jedes** Programmes
- **Lösung: Dynamische Bibliotheken** (shared libraries)
 - Objectfiles werden erst zur Ladezeit oder Laufzeit geladen und gelinkt
 - Code wird nur einmal in Speicher geladen und mehrmals genutzt (shared/virtueller Speicher ➡ Vorlesung Betriebssysteme)

- Standard in modernen Systemen, u.a für C-Standardbibliothek

* Linux: `libc.so`, Windows: `mscrt.dll`, macOS: `libSystem.dylib` bzw. `libc.dylib`

- Erzeugung einer dynamischen Bibliothek

```
> gcc -c -Wall -Werror -fpic swap.c
> gcc -shared -o libsw.so swap.o
```

- Bedeutung der Optionen:

- `-fpic`: Erzeugung von positionsunabhängigen Code (*position independent code*) ➔ keine „Umrechnung“ beim Laden/Linken nötig
- `-shared`: Erzeugt eine unabhängige Bibliothek

- Bibliothek einbinden:

```
> gcc smain2.c -o sm -L. -lsw
```

- *Achtung!* Ein dynamisch gelinktes Programm kann nicht „einfach so“ ausgeführt werden:

```
> ./sm
./sm: error while loading shared libraries: libsw.so: cannot open shared object file: No such file or directory
```

- Der Lader sucht nach dynamischen Bibliotheken in vordefinierten Verzeichnissen, u.a. die in der Systemvariable `LD_LIBRARY_PATH` Pfaden

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
> ./sm
1st string: Hello, 2nd string: World
1st string: World, 2nd string: Hello
1st integer: 42, 2nd integer: 23
1st integer: 23, 2nd integer: 42
```

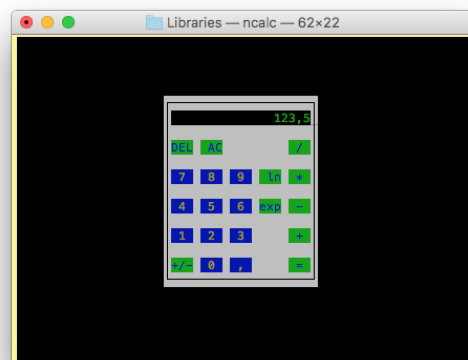
Einsatz von Bibliotheken

- Wenn Funktionen aus fremden Bibliotheken genutzt werden sollen, sollte man immer genau wissen, was der **Effekt** der Funktion ist
 - ➔ Man kann meist nicht „mal schnell“ im Quellcode nachsehen
- Manche Bibliotheken sind lediglich eine Sammlung thematisch zusammengehöriger Funktionen
 - Beispiel: `libmath`
- Andere Bibliotheken stellen ein komplettes **Framework** dar, bei denen die Funktionen nur in einem Zusammenspiel sinnvoll verwendet werden können

- Beispiel: viele GUI-Frameworks, z.B. GTK+ oder Qt
- In diesem Fall hilft mitunter die Dokumentationen zu den einzelnen Funktionen nur wenig
- Häufig gibt es dann Dokumentationen für die gesamte Bibliothek oder Tutorials, HowTos etc.

15.4 Fallstudie: Taschenrechner

- Wir betrachten als Beispiel eine Anwendung, die `ncurses`- und die `math`-Bibliothek nutzt: einen „Taschenrechner“
- Der Rechner beherrscht die Grundrechenarten, zusätzlich noch logarithmieren die Exponentialfunktion
- Die Anwendung läuft zwar im Terminal, aber hat die Anmutung eines „richtigen“ Taschenrechners
- Es wird nicht die komplette Programmentwicklung diskutiert, sondern nur spezielle Aspekte



Ncurses

- Es wird die Bibliothek `ncurses` genutzt
- Die `ncurses`-Bibliothek nutzt die Möglichkeiten der ANSI-Terminals
 - Weiterentwicklung der `curses`-Bibliothek
 - existiert auf für die meisten UNIXe, macOS, Windows, DOS
- `ncurses` stellt (je nach Version) zwischen ca. 800-1000(!) Funktionen zur Verfügung

- Viele Funktionen existieren in verschiedenen Versionen, Namensprefixe geben Auskunft über verwendete Parameter
 - Z.B. hat eine Funktion, die den Prefix „w“, immer ein Parameter von Typ `WINDOW*`, ein Prefix „mv“ bewegt den Cursor, etc.

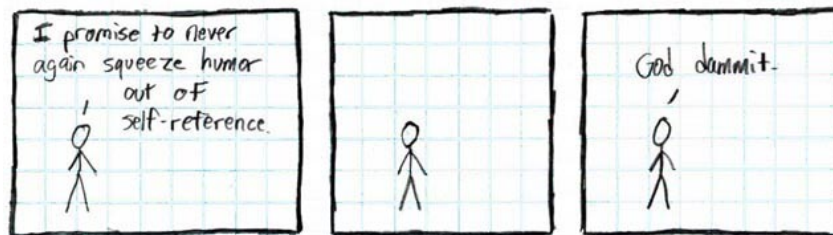
Ncurses - Konzepte

- Kennt ein Konzept von **Fenstern**, die jedoch nicht überlappend sein dürfen ➔ Erweiterung mit `panel`-Bibliothek
- Ausgaben werden erst wirksam, wenn `wrefresh()` für das Fenster aufgerufen hat – häufige Quelle von Fehlern
- Benötigt umfangreiche **Initialisierung**
 - Echo bei Eingabe, Terminalmode, Nutzung von Funktionstasten, etc. ➔ ebenfalls Quelle von Fehlern
- **Farben**/Attribute
 - Farbattribute müssen mit Kombination von Vordergrund- und Hintergrundfarbe initialisiert werden
 - Nutzung über (fenster-)globale Umschaltung **vor** der eigentlichen Ausgabe

```

1  const int mycolor=1;
2  init_pair(mycolor,COLOR_GREEN,COLOR_BLACK);
3  wattrset(win, mycolor);

```



Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/33/>

Ausgabe

- Das Layout wird programmiert
- Zur besseren späteren Änderbarkeit sollten Layoutdaten in einer Datenstruktur zusammen gefasst werden


```

24 static struct button_layout_t layout[]={
25     { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=
        COLOR_PAIR(NUMBER_BT_COLOR) | A_STANDOUT, .x=5, .y=11, .text="
        0", .key=NCK_0},
26     { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=
        COLOR_PAIR(NUMBER_BT_COLOR) | A_STANDOUT, .x=1, .y=9, .text="1
        ", .key=NCK_1},
27     { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=
        COLOR_PAIR(NUMBER_BT_COLOR) | A_STANDOUT, .x=5, .y=9, .text="2
        ", .key=NCK_2},
28     { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=
        COLOR_PAIR(NUMBER_BT_COLOR) | A_STANDOUT, .x=9, .y=9, .text="3
        ", .key=NCK_3},

29     { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=
        COLOR_PAIR(OPERATION_BT_COLOR) | A_STANDOUT, .x=17, .y=3, .text
        ="/", .key=NCK_DIV},
30     { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=
        COLOR_PAIR(OPERATION_BT_COLOR) | A_STANDOUT, .x=17, .y=5, .text
        ="*", .key=NCK_MUL},
31     { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=
        COLOR_PAIR(OPERATION_BT_COLOR) | A_STANDOUT, .x=17, .y=7, .text
        = "-", .key=NCK_SUB},
32     { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=
        COLOR_PAIR(OPERATION_BT_COLOR) | A_STANDOUT, .x=17, .y=9, .text
        = "+", .key=NCK_ADD},
33     { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=
        COLOR_PAIR(OPERATION_BT_COLOR) | A_STANDOUT, .x=17, .y=11, .
        text="=", .key=NCK_RESULT},
34     { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=
        COLOR_PAIR(OPERATION_BT_COLOR) | A_STANDOUT, .x=13, .y=5, .text
        ="ln", .key=NCK_LOG},
35     { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=
        COLOR_PAIR(OPERATION_BT_COLOR) | A_STANDOUT, .x=13, .y=7, .text
        ="exp", .key=NCK_EXP},
36 };

```

(AN UNMATCHED LEFT PARENTHESIS
CREATES AN UNRESOLVED TENSION
THAT WILL STAY WITH YOU ALL DAY.

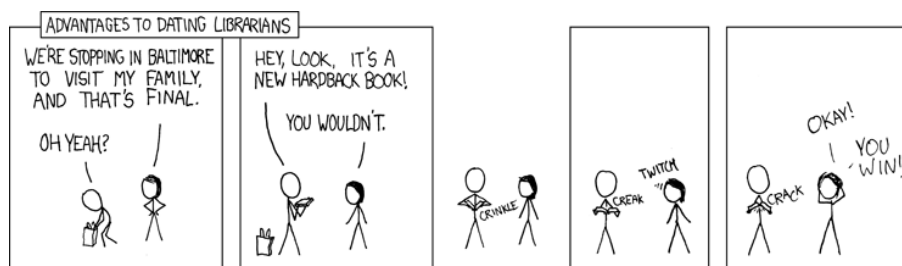
Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/859/>

- Damit kann der eigentliche Code relativ kompakt gehalten werden

```

95 static void ncDrawButton(struct button_layout_t button, bool high){
96     wattrset(win,high? button.click_attr: button.default_attr);
97     if (button.text[1]==0){
98         mvwprintw(win,button.y,button.x," %s ",button.text);
99     } else {
100         mvwprintw(win,button.y,button.x,"%3s",button.text);
101     }
102 }
103
104 static void ncDrawWindow(){
105     // Mitte des Terminals
106     int height=13;
107     int width=21;
108     int x0 = (COLS-width)/2;
109     int y0 = (LINES-height)/2;
110     win = subwin(stdscr,height, width, y0 ,x0 );
111     // Loesche Hintergrund
112     wbkgd(win,COLOR_PAIR(BLACK_WHITE));
113     box(win,0,0);
114     // Zeichne Tasten
115     for (unsigned i=0; i< sizeof(layout)/sizeof(struct button_layout_t
116         ); i++){
117         ncDrawButton(layout[i],false);
118     }
119 }

```

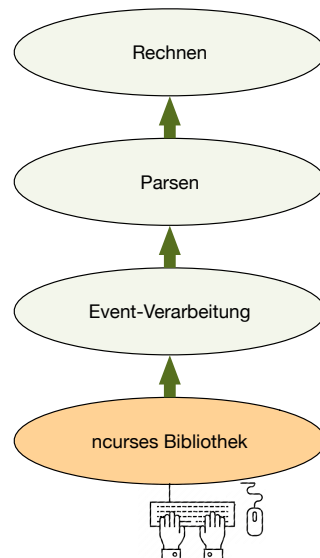


Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/280/>

Eingabe

- Die ncurses-Bibliothek gibt einzelne Ereignisse an die Anwendung...
 - Tastenbetätigung, d.h. Ziffern (keine Zahlen!), einschliesslich Komma
 - Mouse-Aktionen

- Offensichtlich mehrere Schichten der Verarbeitung:
 - Interaktion mit ncurses (Events)
 - Events zu Werten
 - Das eigentliche Rechnen



Tastatur- und Mouse-Ereignisse

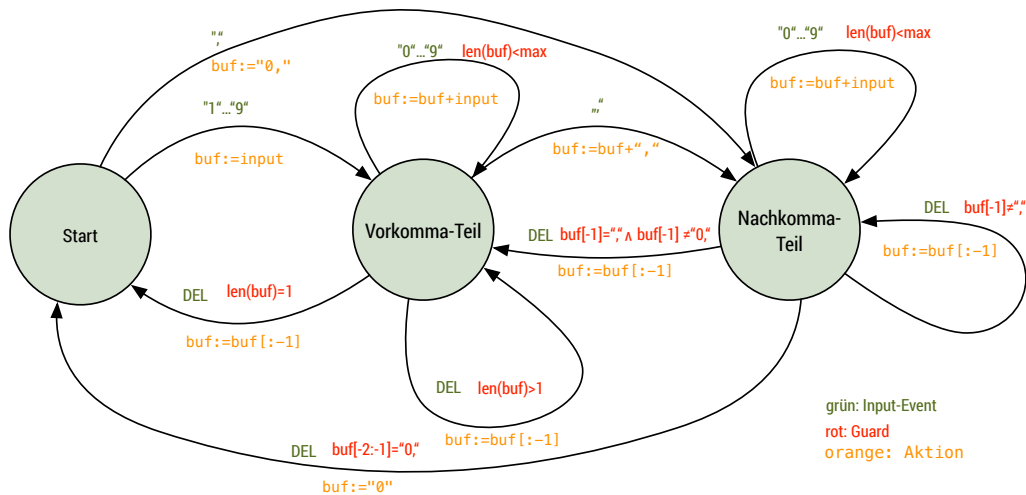
```

120 nc_key_t ncInput(){
121     int c=getch();
122     if ((c>='0') && (c<='9'))
123         return ncGuiAck(c-'0');
124     switch (c) {
125         case ',': // fall through
126         case '.':
127             return ncGuiAck(NCK_DPOINT);
128         case '+':
129             return ncGuiAck(NCK_ADD);
130         case '-':
131             return ncGuiAck(NCK_SUB);
132         case '/':
133             return ncGuiAck(NCK_DIV);
134         case '*':
135             return ncGuiAck(NCK_MUL);
136         case '#': // fall through
137         case KEY_F(1):
138             return ncGuiAck(NCK_SIGN);
139         case '=': // fall through
  
```

```
140     case KEY_ENTER:    // fall through
141     case '\n':
142         return ncGuiAck(NCK_RESULT);
143     case KEY_BACKSPACE:
144         return ncGuiAck(NCK_DEL);
145     case KEY_DC:
146         return ncGuiAck(NCK_RESET);
147     case 'l':
148         return ncGuiAck(NCK_LOG);
149     case 'e':
150         return ncGuiAck(NCK_EXP);
151     case KEY_MOUSE:
152         return ncGuiMouse();
153     case 'q':           // fall through
154     case 'Q':
155         return NCK_QUIT;
156     default:
157         return NCK_NONE;
158 }
159 }
160
161 nc_key_t ncGuiMouse(void){
162     MEVENT event;
163     int ret=getmouse(&event);
164     if (ret != OK || !wenclose(stdscr,event.y,event.x))
165         return NCK_NONE;
166     int x=event.x-getbegx(win);
167     int y=event.y-getbegy(win);
168
169     for (unsigned i=0;
170          i< sizeof(layout)/sizeof(struct button_layout_t);
171          i++){
172         if ((layout[i].y==y) && (x>=layout[i].x)
173             && (x<=layout[i].x+2)) {
174             // Optischer Feedback: Taste blinkt
175             ncDrawButton(layout[i],true);
176             wrefresh(win);
177             usleep(100000);
178             ncDrawButton(layout[i],false);
179             wrefresh(win);
180             return layout[i].key;
181         }
182     }
183     return NCK_NONE;
184 }
```

Parsen

- Die Verarbeitung der Eingabe ist relativ komplex
- Darstellung als guarded Automat(vereinfacht):



Punktrechnung vor Strichrechnung

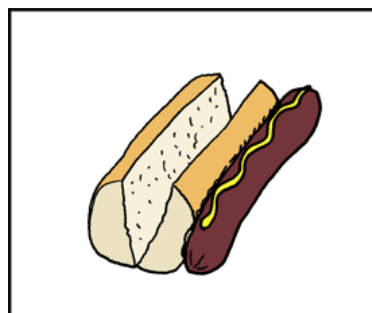
- Auswertung eines Ausdrucks wie z.B.

$$a + b \cdot c - d$$

ist nicht trivial korrekt auszuwerten

- Einfacher in **umgekehrter polnischer Notation** (UPN, *Postfixnotation*):

$$a \ b \ c \ \cdot \ + \ d \ -$$



REVERSE POLISH SAUSAGE

Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/645/>

- Nutzung des **Rangierbahnhof-Algorithmus** (shunting-yard algorithm) von E. DIJKSTRA

Algorithm SHUNTING-YARD ALGORITHM*

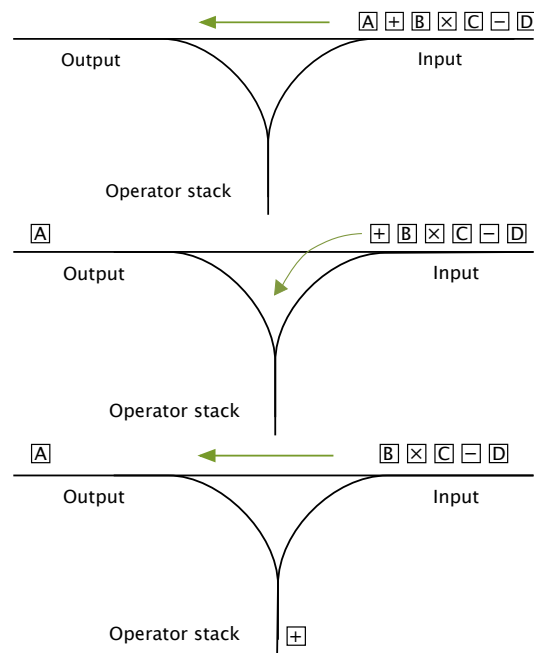
```

repeat
  Lese token
  if token ist Zahl then
    token in Output-Queue
  else
    while auf Operator-Stack liegt operator mit  $\geq$  Vorrang do
      operator in Output-Queue
    end while
    token auf Operator-Stack
  end if
until keine Token übrig
while auf Operator-Stack liegt operator do
  operator in Output-Queue
end while

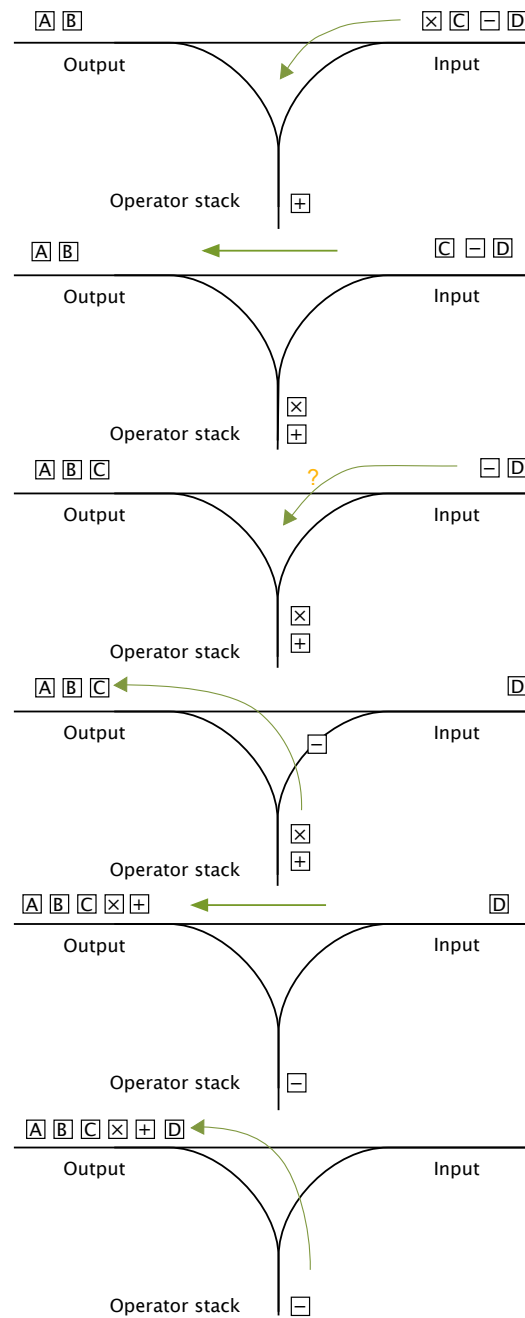
```

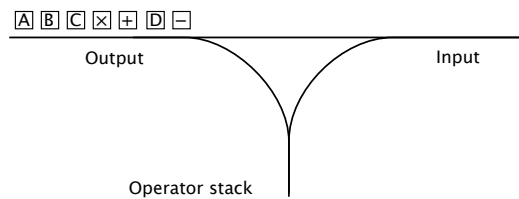
* vereinfacht, Original-Algorithmus behandelt auch Klammern, Funktionen und linksassoziative Operatoren, siehe [Dij63]

Beispiel Shunting-Yard-Algorithmus



15. Linker und Bibliotheken ► 15.4 Fallstudie: Taschenrechner





Implementierung des Shunting-Yard-Algorithmus

- Unsere Funktionen ($\log(x)$, e^x) werden als Postfix-Operatoren mit höchstem Rang behandelt
- Da es beim Taschenrechner kein *a-priori* Ende der Eingabe gibt, wird der Operator-Stack nur bei Tastendruck „=“ explizit geleert

```

26 static void ncShuntingYard() {
27     while (!ncIsEmptyQueue(inputQueue)) {
28         nc_token_t token = ncPopBackQueue(inputQueue);
29         if (ncIsNumber(token)) {
30             ncPushQueue(outputQueue, token);
31         }
32         if (ncIsFunction(token)) {
33             ncPushQueue(outputQueue, token);
34             ncEvaluate();
35         }
36         if (ncIsOperator(token)) {
37             while ((!ncIsEmptyQueue(operatorStack)) &&
38                 (ncPeekTOS(operatorStack).preference >= token.
39                  preference)) {
40                 nc_token_t opToken = ncPopQueue(operatorStack);
41                 ncPushQueue(outputQueue, opToken);
42                 ncEvaluate();
43             }
44             ncPushQueue(operatorStack, token);
45             ncEvaluate();
46         }
47         if (ncIsFlush(token)) {
48             while (!ncIsEmptyQueue(operatorStack)) {
49                 ncPushQueue(outputQueue, ncPopQueue(operatorStack));
50             }
51             ncEvaluate();
52         }
53     }
54 }

```

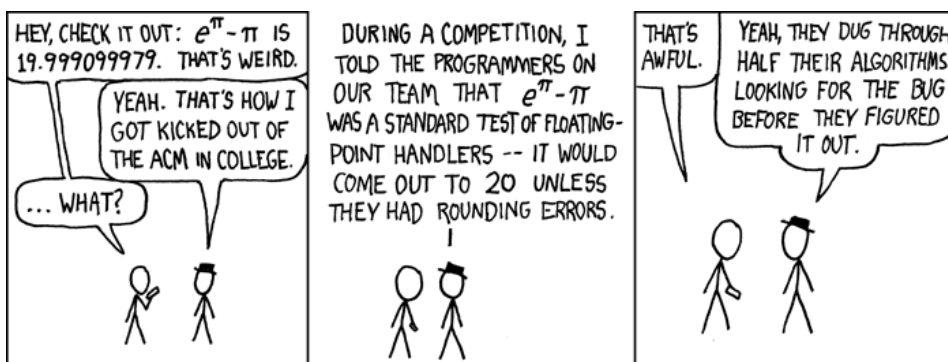

Erkenntnisse

- In dieser Anwendung werden für die eigentliche Rechenlogik weniger als 20% der Gesamtzeilenzahl gebraucht (gesamtes Programm im Anhang des Skripts)
 - Dies ist für GUI-Anwendungen nicht untypisch
- Bei Frameworks spielen viele Komponenten zusammen, so dass mitunter das Erlernen der Nutzung dem einer (manchmal nicht so) kleinen Programmiersprache nahekommmt
- Häufig werden bestimmte Programmieransätze und -stile diktiert (z.B. Ereignisschleifen)

```

1 >wc -l nc-*. [hc]
2     109 nc-calc.c
3         3 nc-calc.h
4     198 nc-gui.c
5         20 nc-gui.h
6         14 nc-main.c
7     147 nc-parseloop.c
8         2 nc-parseloop.h
9     159 nc-token.c
10        44 nc-token.h
11     696 total
12 >

```



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/217/>

Knobelaufgaben

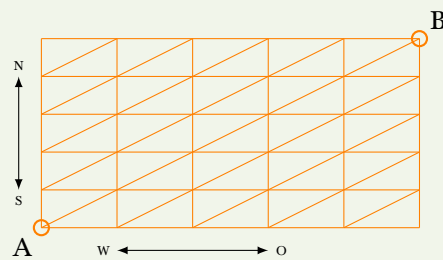
Aufgabe 15.1

Ein Kryptogramm oder Alphametrik ist ein Rätsel, das eine mathematische Gleichung oder ein Gleichungssystem unbekannter Zahlen bildet, deren Ziffern durch Buchstaben ersetzt wurden.

Lösen Sie folgende Kryptogramme:

BLAU	CROSS	PLANET
+LILA	+ROADS	+ EARTH
-----	-----	-----
BRAUN	DANGER	ROTATES

Aufgabe 15.2



Wieviele verschiedene Wege gibt es von A nach B, wenn man niemals „rückwärts“ (also in südliche/westliche Richtung) läuft?

Kapitel 16

Besserer Code

Any fool can write code that a computer can understand. Good Programmers write code that humans can understand.

— Martin Fowler

Ugly programs are like ugly suspension bridges: they're much more liable to collapse than pretty ones, because the way humans (especially engineer-humans) perceive beauty is intimately related to our ability to process and understand complexity.

— Eric S. Raymond

Besserer Code

- Das Erlernen einer Programmiersprache ermöglicht Ihnen (hoffentlich) das Schreiben von (mehr oder weniger) korrekten Code
- **Das ist jedoch erst der Anfang!**
- Ihr nächsten Schritte sollten darin bestehen, Ihre Fähigkeiten und damit Ihren Code zu **verbessern**
- Besserer Code betrifft vielerlei Aspekte, z.B.:
 - Effektivität
 - Wartbarkeit
 - Portierbarkeit
 - Dokumentation
 - Eleganz
- In Bereich **Software-Engineering** (Modul Softwaretechnologie) werden Sie diese Bereiche adressieren
- Wir werden hier aber einiges kurz ansprechen

16.1 Programmier-Konventionen

- Nicht alles, was zulässig ist, sollte genutzt werden
- Nutzung von **Programmier-Konventionen** (*coding conventions*) **Selbstgesetzte** Einschränkungen beim Programmieren
- Wichtig kooperierender Programmerstellung, aber auch für Einzelprogrammierer
- Kann sich auf verschiedene Aspekte beziehen
 - Welches Layout soll benutzt werden?
 - Welche Sprachmittel dürfen benutzt werden?
 - Wie sollen bestimmte Sprachmittel benutzt werden?
 - Wie soll dokumentiert werden?

Layout

- Ein einheitliches Layout trägt maßgeblich zur Lesbarkeit bei
- Sprachen wie Python oder Haskell **erzwingen** bestimmte Layout-Elemente, da diese dort syntaktische Bedeutung haben
- Anders in C
- Wichtige Layout-Elemente in C
 - Reihenfolge von Sprachelementen
 - Einrückung und Klammersetzung der geschweiften Klammern

Reihenfolge von Sprachelementen

- Für C-Programme hat sich eine typische Reihenfolge als Konvention durchgesetzt:

C-Files

- Kommentar-Prolog
- `#includes` für System-Header-Dateien
- `#includes` für eigene Header-Dateien
- Typen/Konstanten-Definitionen
- (Datei-)globale Variablen
- Funktionsdefinitionen

Ggf. werden die letzten drei Punkte wiederholt

H-Files

- Typen/Konstanten-Definitionen
- externe Variablendefinitionen
- externe Funktionsdefinitionen

Auch diese Ordnung kann ggf. mehrfach wiederholt werden

Einrückung und Klammersetzung

- Es gibt einige „typische“ Stile
- Jeder Stil hat Vor- und Nachteile

```

2 int func(int x, int y, int z)
3 {
4     if (x < foo(y, z)) {
5         haha = bar[4] + 5;
6     } else {
7         while (z) {
8             haha += foo(z, z);
9             z--;
10        }
11        return ++x + bar();
12    }
13 }

```

- K&R-Stil (auch: 1TBS)

```

15 int func(int x, int y, int z)
16 {
17     if (x < foo(y, z))
18     {
19         haha = bar[4] + 5;
20     }
21     else
22     {
23         while (z)
24         {
25             haha += foo(z, z);
26             z--;
27         }
28         return ++x + bar();
29     }
30 }

```

- BSD-Stil (auch: Allman-Stil)

```

32 int func (int x, int y, int z)
33 {
34     if (x < foo (y, z))
35         haha = bar[4] + 5;
36     else
37     {
38         while (z)
39         {
40             haha += foo (z, z);
41             z--;
42         }
43         return ++x + bar ();
44     }
45 }

```

- GNU-Stil

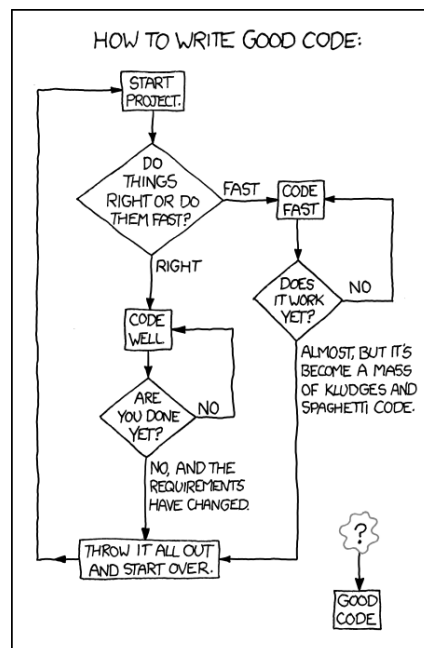
Best Practices

- Es gibt einige Regel, die sich als „gut“ bewährt haben
 - Abweichungen davon sollten nur erfolgen, wenn man „bessere“ Gründe hat
1. Keine Code-Zeile sollte länger als 120 Zeichen sein
 2. Eine einzelne Funktion sollte nie mehr als maximal 200 logische Code-Zeilen enthalten
 3. Tabs sollten vermieden werden; auf keinen Fall dürfen Tabs- und Leerzeicheneinrückungen gemischt werden
 4. Einrücken sollten mindestens 2 Zeichen breit und konsistent durch den gesamten Quelltext sein
 5. Der Long-Suffix für Literale sollte stets groß geschrieben werden
 6. Token der gleichen Art sollten stets durch ein Leerzeichen getrennt werden

Bezeichnernamen

Wie sollen Bezeichner benannt werden?

- **Ungarische Notation**
 - <Präfix><Datentyp><Bezeichner>
 - Präfix ist eine Art „Nutzungstyp“
 - **Vorteil:** Bei typenschwachen Sprachen wie C wichtige Zusatzinformation



Quelle: *xkcd - A webcomic of romance, sarcasm, math, and language*
<http://xkcd.com/844/>

- **Nachteil:** Information kann veralten
- **Binnenmajuskel (Camel-Case)**
 - AussagekraeftigeZusammensetzungMehrereWorte
 - **Vorteil:** Semantische Information
 - **Nachteil:** Tendiert zu sehr langen, schlecht lesbaren Bezeichnern
- **Grundtypen in Schreibweise**
 - z.B. Konstanten in GROSSBUCHSTABEN, Klassen/Variablen als Substantive, Methodennamen als verben
 - **Vorteil:** Gewisse Zusatzinformation, leicht wartbar
 - **Nachteil:** eingeschränkte Aussagekraft

Ungarische Notation

- Der Präfix sagt etwas über die Semantik aus
- Sie werden z.T. kombiniert
- Wichtige Präfixe:

Präfix	Ableitung	Bedeutung
p	p ointer	Zeiger zu einer Adresse
h	h andle	Pointer auf Pointer (entspricht <code>pp</code>), meist in Zusammenhang mit Betriebssystemrufen
rg	r ange	Integerindiziertes Array
mp	m ap	Allgemeiner Sammlungstyp (z.B. Array, Hash, etc.), verlangt beim Typ zusätzlich den Typ des Indexes
i	i ndex	Index (z.B. eines Arrays)
c	c ount	Zähler, Anzahl von Elementen
d	d ifference	Unterschied (Differenz) zwischen zwei Werten, typischerweise Indizes
gr	g roup	Verbund von Variablen (typischerweise <code>struct</code>)

- Der Typ ist ein erweiterter Datentyp
- Wichtige Typenbezeichner

Präfix	Ableitung	Bedeutung
ch	ch aracter	Ein-Byte-Zeichen ➔ <code>char</code>
st	s tring	Pascal-ähnlicher String (mit Längeninformation)
sz	s tring, zero terminated	Nullterminierte Zeichenkette (C-String)
w	w ord	Maschinenwort, etwa <code>int</code>
b	b yte	8-Bit-Integer
l	l ong	4-Byte-Integer (nicht unbedingt C- <code>long</code>)
r	r real	Gleitkommazahl
d	d ouble	Genauere Gleitkommazahl

- Oft wird es bei Präfix und Typ belassen und nichts mehr angehängen
- Wird eine Bezeichner-Suffix angehängt, sollte er Zusatzinformationen beinhalten
- Es gibt Standard-Suffixe, z.B. `Min` für das kleinste Element (eines Arrays)
- Die Ungarische Notation wurde von Microsofts Windows-Gruppe „abgewandelt“ und ist in dieser modifizierten Form bekannt geworden
 - Dabei wurde auf die Semantikinformation verzichtet und dafür Sichtbarkeitsinformationen eingeführt
 - Diese Modifikationen führten zu starker Kritik an der Ungarischen Notation

Best Practices

- Wieder gibt es einige Regeln, die sich für C als „gut“ bewährt haben
- Abweichungen davon sollten nur erfolgen, wenn man „bessere“ Gründe hat

1. Zwei Bezeichner sollten sich nicht unterscheiden durch...

- ausschließlich Groß-/Kleinschreibung
- dem Tausch des Buchstabens „O“ und der Ziffer „0“ oder des Buchstabens „D“
- dem Tausch des Buchstabens „I“ mit der Ziffer „1“ oder dem Buchstaben „l“

Ähnliches (aber etwas weniger kritisch) gilt für die Paare „S“/„5“, „Z“/„2“ und „n“/„h“

2. Bezeichner sollten nicht mit „_“ (Unterstrich) beginnen

3. Bezeichner sollten sich in den ersten 31 Zeichen unterscheiden

16.2 Kommentare & Dokumentation

- Neben einem klaren Programmierstil ist eine gute Kommentierung und Dokumentation des Programmes wichtig
- **Kommentare** sind im Programmcode und dienen zur Erleichterung der **Wartung** des Codes
- **Dokumentation** sind externe Dokumente und können sowohl zur Erleichterung (oder erst Ermöglichung) der **Nutzung** des Programmes dienen, als auch der Wartung des Codes

Kommentare

Merke

Kommentare sollen stets die **Anwendungssicht** unterstützen, nicht die Implementations-sicht.

- Anders gesagt: Dokumentiere in Kommentaren nicht **was** getan wird, sondern **warum**
- Insbesondere sollte **nie** etwas Offensichtliches kommentiert werden:

```
2 int callCount = 0; /* declares an integer variable */
```

- Besser:

```
4 int callCount = 0; /* number of calls to [function] */
```

- Schlecht:

```
34 // assign defer values and increment both pointers until a zero is found
35 while(*to++ = *from++);
```

- Besser:

```

42 // copy C-string <from> to <to>
43 while(*to++ = *from++);

```

- Noch besser, da der Code lesbarer ist:

```

50 // copy C-string <from> to <to>
51 while((*to++ = *from++) != '\0');

```

- Niemals sollten „**magische Zahlen**“ unkommentiert im Programmtext stehen
- Schlecht:

```

2 for (i=0; i<80; i++){
3     ...

```

- Besser:

```

5 for (i=0; i<80; i++){ // 80: number of columns
6     ...

```

- Noch besser:

```

8 enum {tCol=80, tRow=24}; // terminal size
9     ...
10 for (i=0; i<tCol; i++){
11     ...

```

- Wenn eine Variable eine physikalische Größe beschreibt, sollte unbedingt die Einheit angegeben werden:

```

6 double weight; /* weight of [something] in kilogramm */

```

Empfehlung

Wenn Sie eine Sprache mit einem geeigneten Typsystem haben, nutzen Sie es, um den richtigen Gebrauch von physikalischen Einheiten durchzusetzen!

- Es sollte **immer** ein Kommentar gegeben werden, wenn **potentiell fehlerhafte** Konstrukte tatsächlich so gemeint sind
- Insbesondere gilt dies für
- fehlendes `break/return` in `switch`-Statements

```

2 switch (a){
3   case 0: // fall through
4   case 1:
5     bar(x);
6     break;
7   default:
8     foo(y);
9     break;
10 }

```

- Zuweisungen in if-Statements (sollte aber stets vermieden werden!)

```

11 if (x = y) { // assignment
12   ...

```

- Einige Compiler geben bei doppelter Klammerung keine Warnung aus:

```

11 if ((x = y)) {
12   ...

```

- Aussagekräftige Bezeichnernamen stellen auch eine Kommentierung dar
- **Aber:** Vorsicht bei Indexvariablen
- **Vergleich:**

```

2 for(i=0 to MAX)
3   array[i]=0

```

und

```

5 for(elementnumber=0 to MAX)
6   array[elementnumber]=0;

```

Prolog

- Jede Quellcode-Datei sollte mit einem längeren Kommentarblock anfangen, der wesentliche Informationen enthält, wie
 - Namen des Projekts, in dem der Code entstanden ist
 - Autor
 - Entstehungsdatum

- Was macht der Code
- Wie nutzt man den Code/das Programm
- Wesentliche (globale) Informationen können auch sein:
 - Referenzen bei Nutzung fremden Codes
 - Verwendete Dateiformate
 - Fehlerbehandlung
 - Änderungsgeschichte
- Auch jede Funktion sollte einen Prolog besitzen
- Beispiel aus dem Linux-Sourcecode

```

1  /*
2  * Calculate task priority from the waiter list priority
3  *
4  * Return task->normal_prio when the waiter list is empty or when
5  * the waiter is not allowed to do priority boosting
6  */
7  int rt_mutex_getprio(struct task_struct *task)
8  {
9      if (likely(!task_has_pi_waiters(task)))
10         return task->normal_prio;
11
12     return min(task_top_pi_waiter(task)->pi_list_entry.prio,
13               task->normal_prio);
14 }

```

Konventionen

In vielen Projekten ist der Funktionsprolog genau festgelegt.

Mitunter muss er bestimmten formalen Standards entsprechen, um die Nutzung von Dokumentationstools (➡ nächster Abschnitt) zu unterstützen

• Beispiel:

```

1  /*
2  Function: Multiply
3
4  Multiplies two integers.
5
6  Parameters:
7     x - The first integer.
8     y - The second integer.

```

```

9
10 Returns:
11     The two integers multiplied together.
12
13 See Also:
14     <Divide>
15 */
16 int Multiply (int x, int y)
17 { return x * y; };

```

- Beim Prolog sollten Header- und Source-Dateien unterschiedlich behandelt werden (.h vs. .c)
- Sie haben unterschiedliche Funktionen:
 - Source-Dateien sind reine Implementationsdateien
 - Header-Dateien beschreiben (häufig) Schnittstellen zu den implementierten Code, der genutzt wird, **ohne dass die Implementation bekannt** ist
 - * Dies gilt insbesondere für Bibliotheken
- ➔ Kommentare in Header-Dateien sollen die **Nutzung** der dort deklarierten Funktionen ermöglichen, **ohne** die (Kommentare der) Implementierung zu kennen
- Falls die Funktionen **gut dokumentiert** sind, können auf entsprechende Kommentare (Prologe) in Header-Dateien verzichtet werden

Interface

- Mitunter will man bei Bibliotheken gar nicht alle Datenstrukturen offenlegen
- Dann funktioniert der Trick der unvollständigen Deklaration und die Unterteilung in öffentliche und private Headerfiles

– Öffentlicher Header

```

4 typedef struct foo foo_t;
5
6 /* processes <var> */
7 void bar(foo_t* var);

```

– Privater Header

```

11 struct foo{
12     int x;
13     float y;
14 };

```

Beispiele

- ... wie man es nicht unbedingt machen sollte¹

```
1 /* I don't understand how the following bit works, but it
2    worked in the program I stole it from.
3 */
```

- Obwohl das immer noch besser ist, als kein Kommentar – der Maintainer weiß, woran er ist

```
1 int main(void) /* Program starts here */
```

```
1 /* as you can see: I comment the code! */
```

```
1 /* If you cannot figure it out, you should not be reading this */
```

```
1 /*
2  * Dear maintainer:
3  *
4  * Once you are done trying to 'optimize' this routine,
5  * and have realized what a terrible mistake that was,
6  * please increment the following counter as a warning
7  * to the next guy:
8  *
9  * total_hours_wasted_here = 39
10 */
```

Gesetz von Oualline

Gesetz von Oualline

In 90 von Hundert Fällen, in denen man die Dokumentation braucht, ist diese verloren gegangen.

In den restlichen 10 Fällen beschreibt in 9 davon die Dokumentation eine andere Version des Produkts beschreiben und ist damit nutzlos.

In dem verbleibenden Fall, in dem man eine Dokumentation zur Verfügung hat und diese die richtige Version des Produkts beschreibt, ist die Dokumentation in Chinesisch.² 😊

¹Beispiele aus realen Code, gefunden bei <http://stackoverflow.com>.

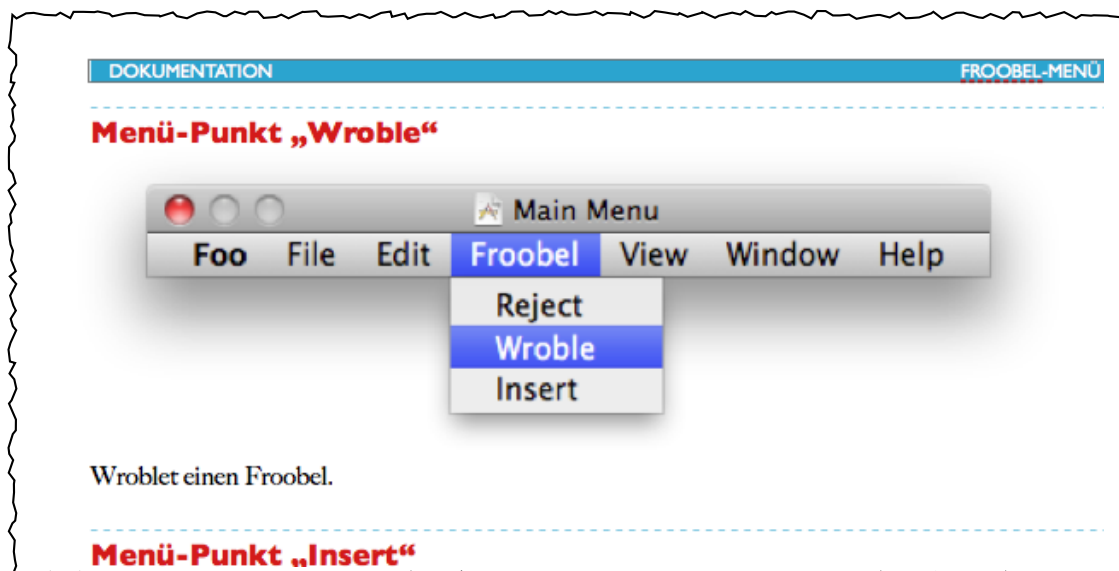
²Ausnahmen gelten hier für chinesisch sprechende Menschen: In diesem Fall ist die Dokumentation in Suaheli o.ä.

Zielgruppe

- Eine Dokumentation ist eine Veröffentlichung³
- Man sollte sich daher stets darüber klar sein
 - **wer** das Dokument liest (insbesondere welches Vorwissen vorhanden ist)
 - **was** man damit erreichen will
- Folglich gibt es verschiedene Arten von Dokumentationen, z.B.:
 - Nutzerhandbuch
 - Installationsanleitung
 - Bibliotheksmanual
 - Implementationsbeschreibung
 - Dokumentation der zugrundeliegenden Konzepte (z.B. Programmiersprache bei einem Compiler/Interpreter)
 - ...

Inhalt

- Dokumentationen sind technische Dokumente ➔ Sie sollten knapp, präzise und **leicht lesbar** sein
- *Achtung:* Immer Lesersicht beachten!
- Typisches (schlechtes) Beispiel:

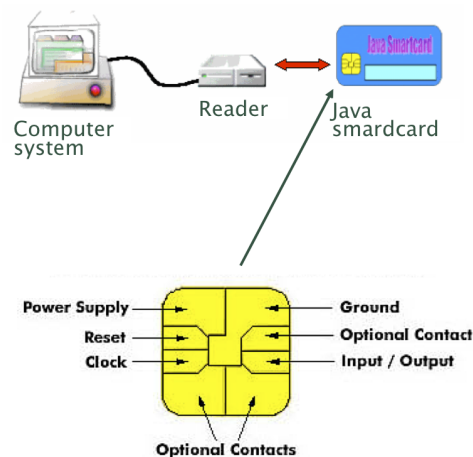


³Genau genommen gilt dies auch schon für den Programmcode.

- Was ist ein(e) „Froobel“?
- Was ist „Wroblen“?
- **Warum** sollte man ein(e) Froobel überhaupt wroblen?
- **Begriffe, Konzepte und Vorgehensweisen** sollten erklärt werden!

Bilder

- Ein Bild sagt mehr als 1000 Worte
- **Vergleich:**
- Java-card wird mit einem speziellen Lesegerät gelesen
- Kontakte
 - Stromversorgung
 - Reset
 - Clock
 - Ground
 - Input/Output
 - Optionale Kontakte



- **Achtung:** Bilder sollten präzise und **zielgerichtet** eingesetzt werden
 - Ansonsten lenken sie ab

Format

- Dokumentationen können entweder in gedruckter Form oder in computer-lesbarer Form vorliegen
- Bei computer-lesbarer Form sollte darauf geachtet werden, dass sie (auf der Plattform) allgemein lesbar sind
- (Relativ) universell sind folgend Formate
 - ASCII-Text
 - PDF
 - HTML
 - evtl. PostScript (stirbt derzeit aus)
- Für spezifische Plattformen:
 - Windows: CHM
 - UNIX: man-Pages (groff), info
- Wenn Dokumentationen in Papierformat gegeben werden, hat man bei der Wahl der Werkzeuge ein weites Spektrum an Möglichkeiten

Achtung!

Der Prozess der Dokumentation sollte eng mit dem des Programmentwurfs und -entwicklung verbunden sein. Der gewählte Arbeitsablauf sollte daher eine einfache Datenmigration in die Dokumentation zu ermöglichen.

- **Geeignet:** Tools, die automatisierte Verknüpfungen unterstützen ➡ z.B. $\text{\LaTeX}$ mit Unterstützung von Texttools
- **Schlechter geeignet:** Maus-orientierte Programme ➡ z.B. MS Word

UNIX-Manuals

- In der „UNIX-Welt“ hat sich ein bestimmter Stil für Dokumentationen herausgebildet ➡ man-Pages
- Sie bestehen i.d.R. aus folgenden Abschnitten
 - **Name**
Welche(s) Programm(e), Funktion(en) etc. werden beschrieben
 - **Synopsis**
Synopsis/Synopsis = Zusammenfassung: das Wichtigste auf einen Blick, z.B. die Aufrufparameter oder Funktionssignatur

- **Description**
Hier erfolgt die ausführliche Beschreibung (der Funktion); ggf. kann dieser Abschnitt weiter unterteilt werden
 - **Options**
Wenn Optionen vorhanden sind (und nicht im Description-Abschnitt diskutiert wurden) wird hier die Wirkung **aller** Optionen beschrieben
 - **Exit Status/ Return Value**
optional; Bedeutung von Rückgabewerten von Programmen/Funktionen
 - **Errors**
Was bedeuten zurückgegebene Fehlercodes
 - **Files**
Welche Dateien werden genutzt, z.B. zur Konfiguration oder für Logging
 - **Notes**
Anmerkungen, die in keine der anderen Kategorien passen
 - **Bugs**
Wenn Fehler vorhanden sind, sollten sie dokumentiert werden (oder noch besser: behoben werden)
 - **Example**
Nutzungsbeispiele
 - **See also**
Welche anderen Dokumente sollte man sich ggf. noch ansehen
- `man`-Pages sind Textdateien mit speziellen Formatierungsbefehlen, jeweils am Zeilenanfang:
 - `.TH [Titel]` ➔ setzt den Titel der `man`-Page
 - `.SH [Abschnittsname]` ➔ Erzeugt neuen Abschnitt mit dem Namen [Abschnittsname]
 - `.PP` ➔ Startet neuen Absatz
 - `.B [Text]` ➔ Setzt [Text] fett
 - `.I [Text]` ➔ Setzt [Text] kursiv, in Terminals häufig unterstrichen
 - Für weitere Befehle ➔


```
man groff_man
```
 - Beispiel:

```

1 .TH BABY 1 "2000-1-1" "" "Funny Man Pages"
2 .SH NAME
3 .B baby
4 -- create new process from two parents
5 .SH SYNOPSIS
6 .I baby -sex [m|f] [-name name]
7 .SH DESCRIPTION
8 .I baby
9 is initiated when one parent process polls another
  server
10 process through a socket connection in the BSD
  version or
11 through pipes in the System V implementation.
12 .I baby
13 runs at low priority for approximately forty weeks
  and then
14 terminates with a heavy system load. Most systems
  require
15 constant monitoring when
16 .I baby
17 reaches its final stages of execution.
18 .PP
19 Older implementations of
20 .I baby

```

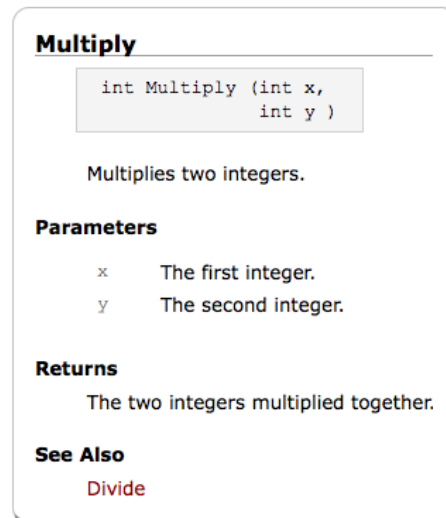
- Man-Pages eines Systems sind in verschiedene Abschnitte (*sections*) unterteilt
- Der Abschnitt bestimmt Namen und Pfad der Man-Page-Datei

Abschnitt	Beschreibung
1	Von der Shell aus ausführbare Programme
2	Systemrufe (Funktionen, die der Kernel bereit stellt)
3	Bibliotheksfunktionen
4	Spezialdateien (typischerw. Treiber)
5	Dateiformate und Konventionen (z.B. /etc/passwd)
6	Spiele
7	Verschiedenes (inkl. Macro-Pakete), z.B. man(7), groff(7)
8	Systemadministration
9	Kernelroutinen und Programmierstil (Kein Standard)
n	Tcl/Tk-Befehle
x	Befehle und Dateien des X-Systems

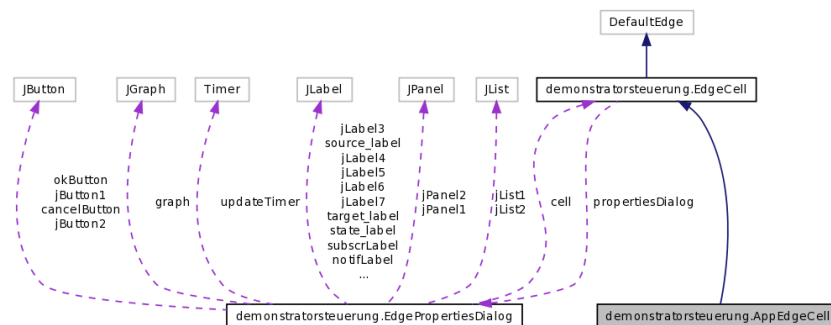
Automatische Generierung

- Für Standardaufgaben der Programmdokumentation stehen Tools zur Verfügung
- Typische Aufgabe: Programmcode dokumentieren

- **Idee:** Information aus dem Quelltext generieren, z.B. aus standardisierten Prologen
- **Bekannte Tools:** Doxygen, Natural Docs oder HeaderDoc



- Nützlich sind solche Tools insbesondere, wenn **Beziehungen** zwischen Datenstrukturen dokumentiert werden sollen
- Insbesondere bei **Objektorientierung** ⇒ Vorlesung „Algorithmen und Datenstrukturen“
- **Beispiel** aus „echtem“ Projekt:



Literate Programming

- **Idee:** Die Dokumentation ist das Wichtige, der Code Ergänzung ⇒ **Literate Programming**
- Literate-Programming-System:
 1. Quelldokument ist Mischung von Quellcode und Kommentaren
 2. Quellcode-Abschnitte können in beliebiger Reihenfolge erscheinen

3. Aus Quelldokument(en) wird sowohl Dokumentation (mit Inhaltsverzeichnis, Register, etc.) und ausführbares Programm generiert

- **Bekannte Systeme:**

- WEB/CWEB ➔ ursprünglich für Pascal, heute auch für C
- Haskel ➔ schon im Sprachentwurf enthalten

16.3 Portabilität

- Guter Code zeichnet sich durch **Portabilität** aus
- Portabilität hat zwei **Dimensionen**
 - **räumlich:** Der Code kann auf einer anderen Plattform ausgeführt werden
 - **zeitlich:** Der Code wird auf einer anderen Version der gleichen Plattform ausgeführt *oder* mit einem älteren/jüngeren Compiler übersetzt
- Zwei Ziele:
 - Nicht-portablen Code **vermeiden**
 - Wenn dies nicht möglich: **Isolieren** und **gezielt behandeln**

Typische Portabilitätsprobleme

- Jedes Sprach-/Bibliotheksfeature, das als „implementation defined“ gekennzeichnet ist, gilt als nicht portabel ➔ **Beispiel:** Alignment
- Es sollten keine Annahmen über die genaue Größe von Ganzzahltypen gemacht werden (oder `stdint.h` genutzt werden)
- `int*` und `char*` können unterschiedliche Darstellungen haben
- „Magische“ Adressen (außer `NULL`) sind nicht portabel
- Maschinennahe Datenstrukturen (insbesondere Register etc.) sind nicht portabel
- Ein-/und Ausgabe (außer `stdio`) sind nicht portabel
- Strukturen können Löcher enthalten (oder auch nicht)
- Auswertungsreihenfolge von Ausdrücken und Parametern ist i.d.R. nicht festgelegt
- Dateipfade (insbesondere absolute) sind **im hohen Maße** nicht-portabel



Ein- und Ausgabe

- Ein- und Ausgabe ist ein Hauptquell von Nichtportabilität
- **Achtung:** Viele „Wizards“ generieren nicht-portablen Code
- Abhilfe:
 1. E/A isolieren (Schichtenmodell) ➔ Beispiel Tic Tac Toe aus Kapitel 14
 2. Plattformunabhängig grafische E/A nutzen, z.B.
 - Tcl/Tk
 - GTK+
 - Qt

Präprozessor

- Wenn versions- oder plattformabhängiger Code eingesetzt werden muss, kann der Präprozessor helfen
- Der Präprozessor ermöglicht **bedingte Übersetzung** und **Macros**

```

1 #ifdef SYMBOL
2 /* code will be translated if SYMBOL is defined */
3 #define MYINT int
4 #else
5 /* code will be translated if SYMBOL is *not* defined */
6 #define MYINT long
7 #endif

```

- Einfache Macros: `#define LINT long int`
 - Ersetzt überall im Quelltext LINT durch long int
 - Der Compiler „sieht“ den Bezeichner LINT **nicht!**
 - Einsatz als **symbolische Konstante** ➔ nicht empfohlen
- Macros mit Parametern: `#define STR(name,x) char name[x+1]`
 - Ersetzt überall im Quelltext z.B. STR(foo,30) durch char foo[30+1]
 - Vorsicht bei Seiteneffekten!

Der Präprozessor hat Direktiven zur bedingte Übersetzung:

- `#if <cond>` ➔ Der folgende Code wird übersetzt, wenn Bedingung <cond> wahr ist
- `#ifdef <sym>` oder `#if defined(<sym>)` ➔ Der folgende Code wird übersetzt, wenn (Präprozessor-)Symbol <sym> definiert ist

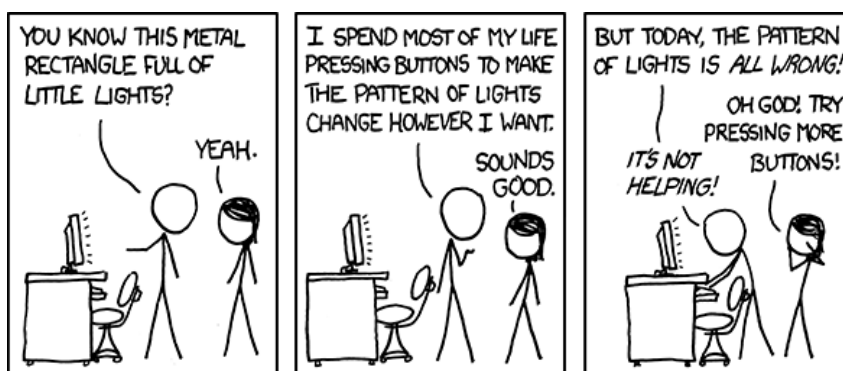
- `#ifndef <sym>` oder `#if !defined(<sym>)` ⇒ Der folgende Code wird übersetzt, wenn (Präprozessor-)Symbol `<sym>` undefiniert ist
- `#else` ⇒ Alternativzweig bei bedingter Übersetzung
- `#elif` ⇒ Fallunterscheidung (⇒ `#else` + `#if`)
- `#endif` ⇒ Ende der Fallunterscheidung
- Beim Compiler sind eine Reihe von Präprozessorsymbolen sind je nach Umgebung vordefiniert
 - Können mit `gcc -E -dM <file>` angezeigt werden
 - Eigene Symbole können mit „-D“-Optionen an den Compiler übergeben werden
- Beispiel Portabilität durch bedingten Code:

```

1 #if defined(__gnu_linux__)
2   #define SYS_HDR "linux.h"
3 #elif defined(_WIN64) || defined(_WIN32)
4   #define SYS_HDR "windows.h"
5 #elif defined(__APPLE__)
6   #define SYS_HDR "macosx.h"
7 #else
8   #define SYS_HDR "default.h"
9 #endif
10
11 #include SYS_HDR /* system dependent */

```

- Parameterisierte Makros können genutzt werden, um Code lesbarer zu machen
- Z.B. stellt(te) Windows die Funktion `cprintf()` zur Verfügung, die wie `printf()` funktioniert, aber deren Ausgabe nicht umleitbar ist



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/722/>

```

1 #if defined(_WIN32)
2   #include<conio.h>
3   #define xprintf(x) cprintf(x)
4 #else
5   #include<stdio.h>
6   #define xprintf(x) \
7       {FILE* _con = fopen("/dev/tty","w"); \
8         fprintf(_con,x); \
9         close(_con);}
10 #endif
11
12 int main(){
13     xprintf("Hello world!\n");
14     return 0;
15 }

```

Anmerkung

Dieses Beispiel hat einige Probleme und dient nur zur Illustration; vom Einsatz in „echten“ Code wird abgeraten.

U.a. kann nur Text und keine weiteren Parameter übergeben werden. Dies könnte man z.B. durch **varadische** Makros lösen (ab C99).



- Der Präprozessor kann noch einiges mehr, was hier nicht besprochen wird
- **Warnung:** Der Präprozessor unternimmt reine Textersetzungen, kennt keine Typen, und „versteh“ kein C
- Typischer Fehler:

```

1 #include<stdio.h>
2 #define max(x,y) (x > y)? x : y
3
4 int main(){
5     int x,y,z;
6     x=42; y=43;
7     z=max(x,y++);
8     printf("x,y,z=(%d,%d,%d)\n",x,y,z);
9     return 0;
10 }

```

```

> ./max
>x,y,z=(42,45,44)

```


16.4 Über den Tellerrand

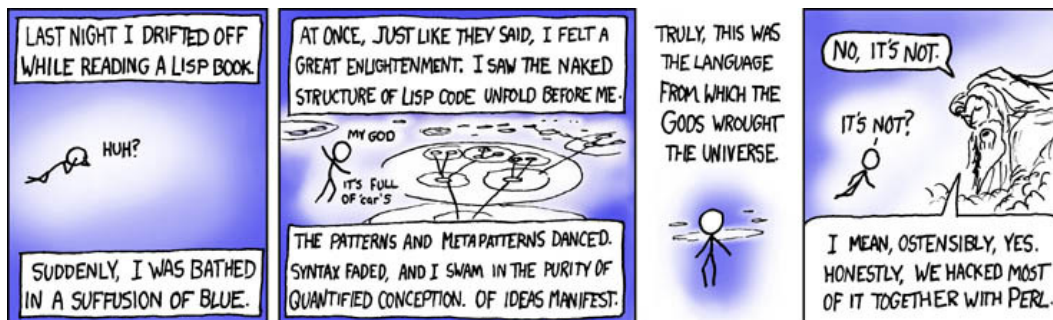
- Eine Voraussetzung für besseren Code ist Anwendung

Nur wenn Sie ständig programmieren, kann sich Ihr Programmierstil verbessern!

- **Wichtig:** Eine Programmiersprache „richtig“ beherrschen
- **Nützlich:** Konzepte aus anderen Programmiersprachen und Systemen kennenlernen
 - ...auch wenn Sie nicht dafür programmieren (müssen)

Empfehlungen

- Im Folgenden einige Empfehlungen für Programmiersprachen, die Sie sich „mal angucken“ sollten
- (Irgendein) **Assembler**
 - ➔ Verständnis für die Programmausführung auf der tatsächlichen Hardware
- **Lisp** (oder **Scheme**, **Logo**, etc.)
 - ➔ Erkenntnis, dass Ganzzahlen nicht der Anfang aller Weisheit sind ☹️
 - ➔ Denkansatz, Daten und Code als austauschbar zu betrachten
 - ➔ Fähigkeit, EMACS zu erweitern



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/224/>

- **Smalltalk** (oder **Squeak**, **S#**, etc.)
 - ➔ Erkenntnis, dass Objektorientierung und Typensystem orthogonale Konzepte sind
- **Haskell** (oder **ML**, **(O)Caml**, etc.)
 - ➔ Schönheit und Eleganz des funktionalen Modells
 - ➔ Kraft starker Typenkonzepte
- **Occam** (oder **Go**, **Limbo** etc.)

- ➔ Erkenntnis, dass die Welt nicht linear ist (➔ **Nebenläufigkeit**)
- **LabView** (oder **Simulink**, **Quartz Composer**, etc.)
 - ➔ Eleganz des Datenflussmodells
 - ➔ Erkenntnis, dass Programmierung nicht immer in Textform erfolgen muss
- Vielleicht auch interessant: **Oz**, **Prolog**, **Lua**, etc.

Esoterische Programmiersprachen

- **Esoterische Programmiersprache**
 - nicht für den tatsächliche Einsatz geeignet
 - demonstriert „ungewöhnliche“ Sprachkonzepte
 - häufig auch als Witz
- Beispiel **Brainfuck**
 - Minimale Sprache
 - Beruht auf Modell der Turingmaschine
- Beispiel **Piet**
 - Quell„texte“ sind Bilder
- Beispiel **Q-BAL**
 - Benutzt Queues (Warteschlangen) als Datenmodell (anstatt Stacks)



Aufgaben

Aufgabe 16.1

Entwerfen und installieren Sie eine UNIX-Man-Page für ein Programm Ihrer Wahl!

- Für die nächsten Aufgaben benötigen Sie **Chef**
 - Chef ist eine esoterische Programmiersprache, deren Quelltexte wie Kochrezepte aussehen
 - Ausführungsmodell basiert auf (mehreren) Stacks
 - Es gibt einen Chef-Interpreter (der in Perl und Java programmiert ist)
 - Mehr unter <http://www.dangermouse.net/esoteric/chef.html>

Aufgabe 16.2

Finden Sie heraus, was folgendes Chef-Programm macht:

Banana bread.

Delicious banana bread with walnuts. Ingredients.

5 cups butter
2 cups sugar
2 eggs
2 teaspoons water
2 cups mashed bananas
1 teaspoon vanilla extract
3 cups flour
1 teaspoon salt
2 teaspoons baking soda
1 cup chopped walnut

Cooking time: 1 hour.

Put flour into the 1st mixing bowl. Add salt into the 1st mixing bowl. Combine baking soda into the 1st mixing bowl. Liquify butter. Combine butter into the 1st mixing bowl. Add vanilla extract. Put sugar into the 1st mixing bowl. Add eggs into the 1st mixing bowl. Combine water into the 1st mixing bowl. Combine mashed bananas into the 1st mixing bowl. Fold chopped walnuts into the 1st mixing bowl. Put chopped walnuts into the 1st mixing bowl. Combine flour into the 1st mixing bowl. Add butter. Add butter. Liquify contents of the 1st mixing bowl. Pour contents of the 1st mixing bowl into the baking dish.

Serves 1.

Aufgabe 16.3

Schreiben Sie ein sinnvolles Programm in Chef, das gleichzeitig ein sinnvoll ausführbaren Rezept^a darstellt! (*Achtung, das ist schwerer als es klingt.*) 😊

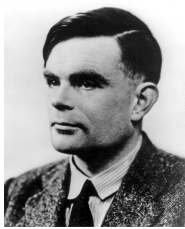
^aD.h. eines, bei dem Sie das Ergebnis der Durchführung auch tatsächlich essen möchten.

Anhang

Anhang A.

Programmiersprachen allgemein

A.1. Galerie von Computerpionieren



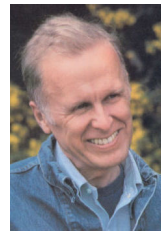
Alan M. Turing



Konrad E.O.
Zuse



Grace B.M.
Hopper



John W. Backus



John McCarthy



Ole-Johan Dahl



Kristen Nygaard



C.A.R. Hoare



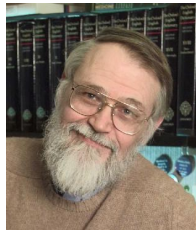
Edsger W.
Dijkstra



Alan C. Kay



Alain M.A.
Colmerauer



Brian W.
Kernighan



Kenneth L.
Thompson



Dennis Ritchie



Guido van
Rossum

A.2. Wenn Programmiersprachen Filmcharaktere wären...



C

Idee nach [Tat10]



Java



Python



Lisp



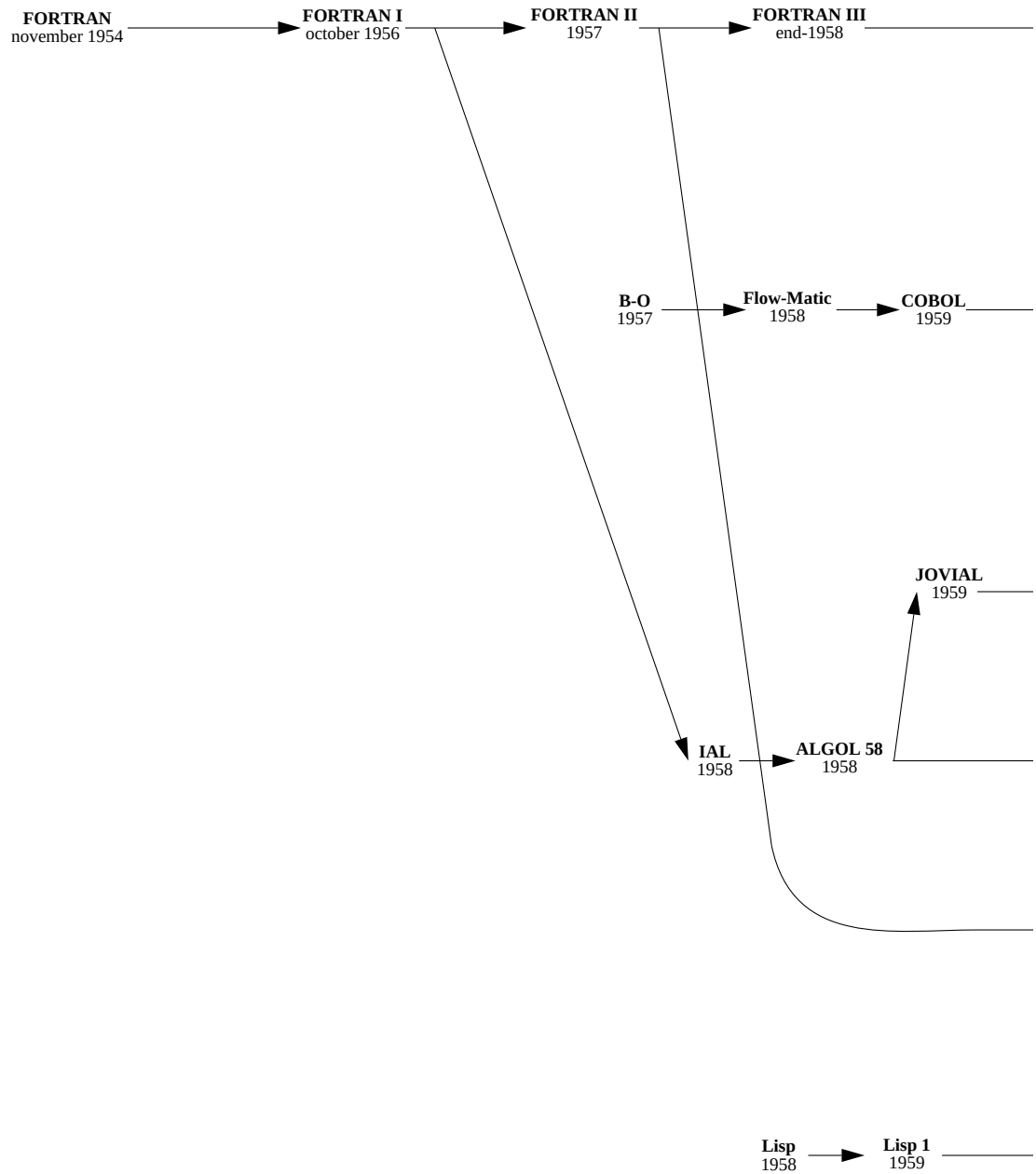
Haskell

A.3. Zeittafel

- Auf den folgenden Seiten ist für die 50 wichtigsten Programmiersprachen eine Zeittafel dargestellt.
- Die Quelle der Übersicht ist <http://www.levenez.com/lang/>.

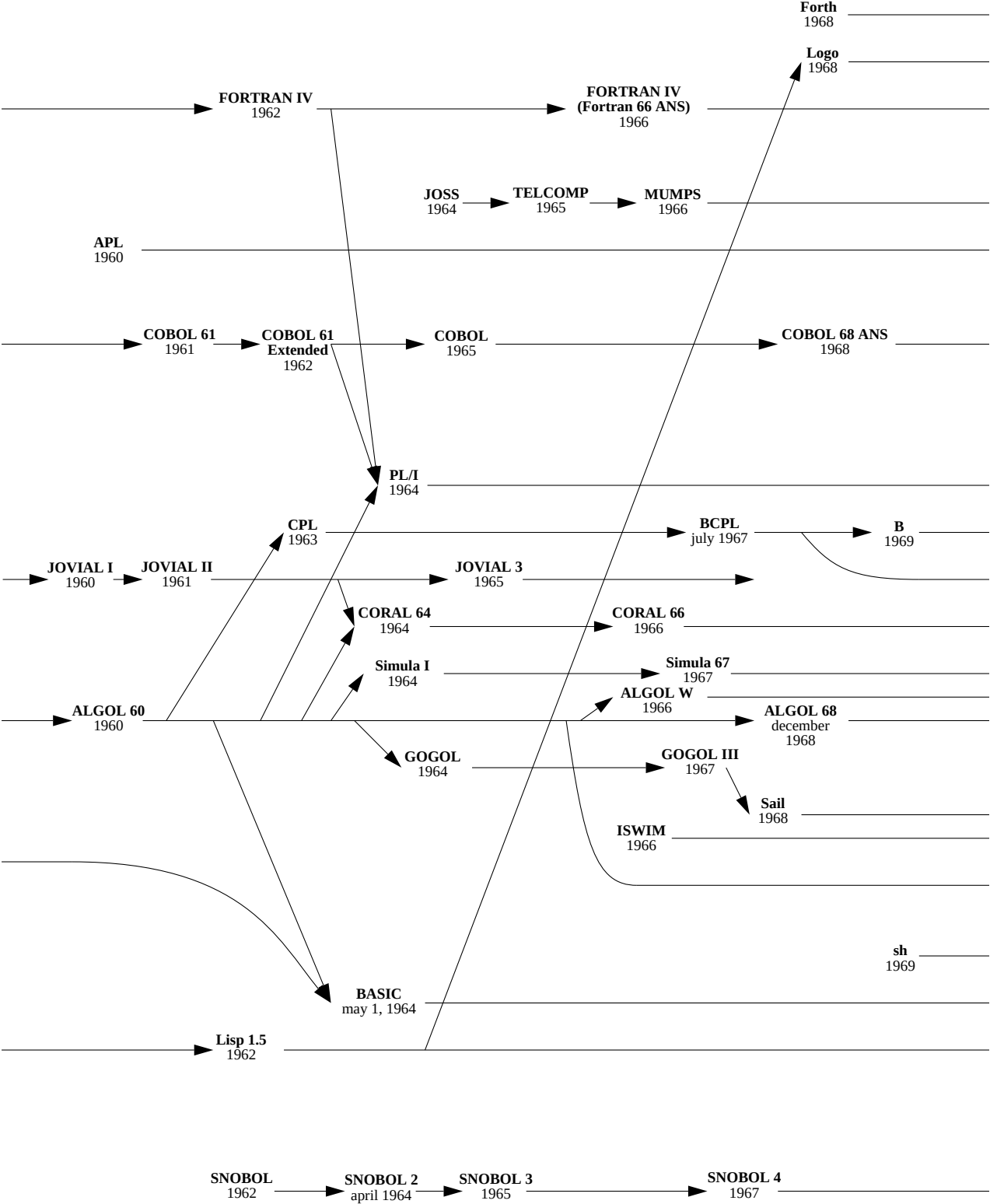
1954

1957



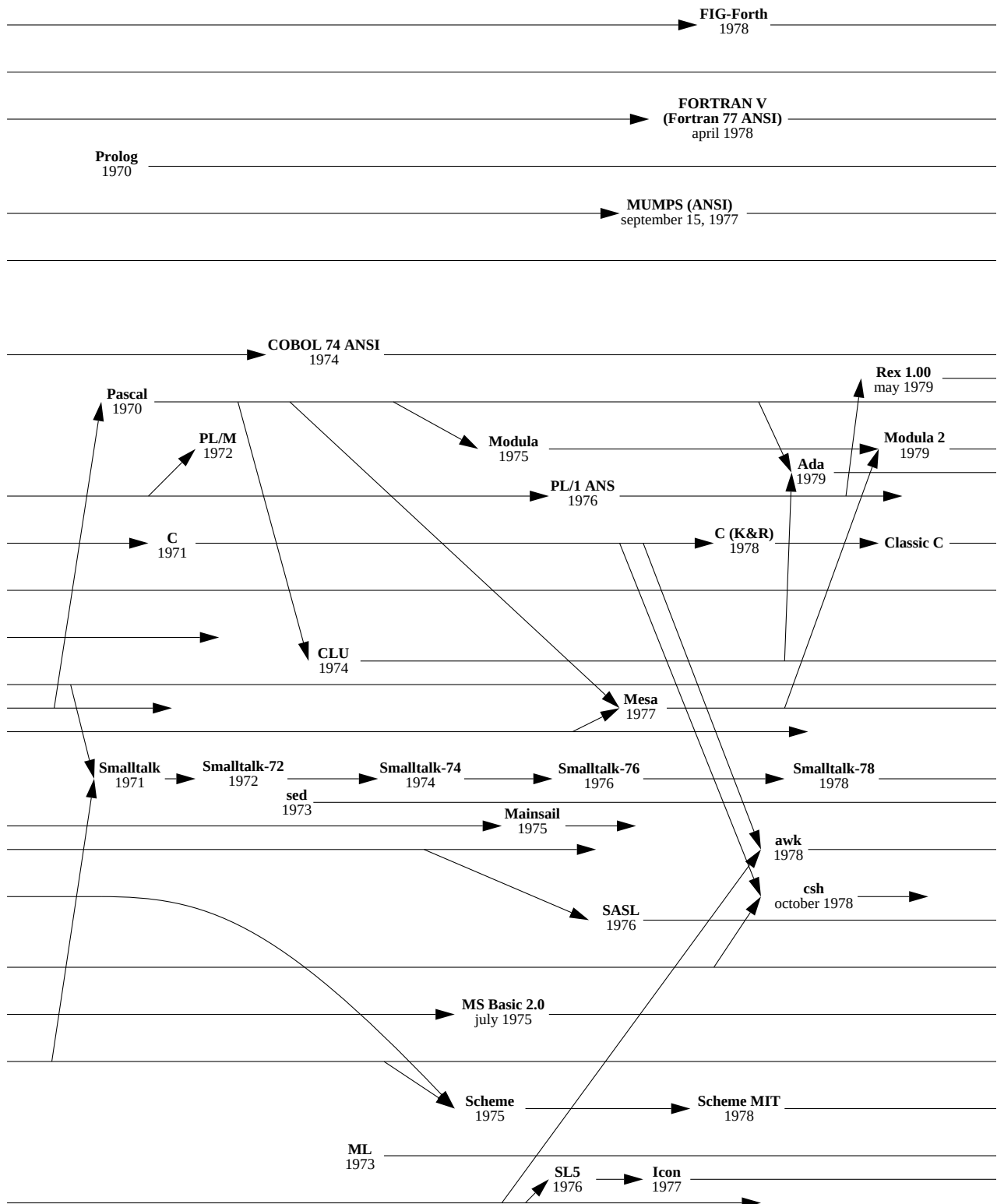
1960

1965



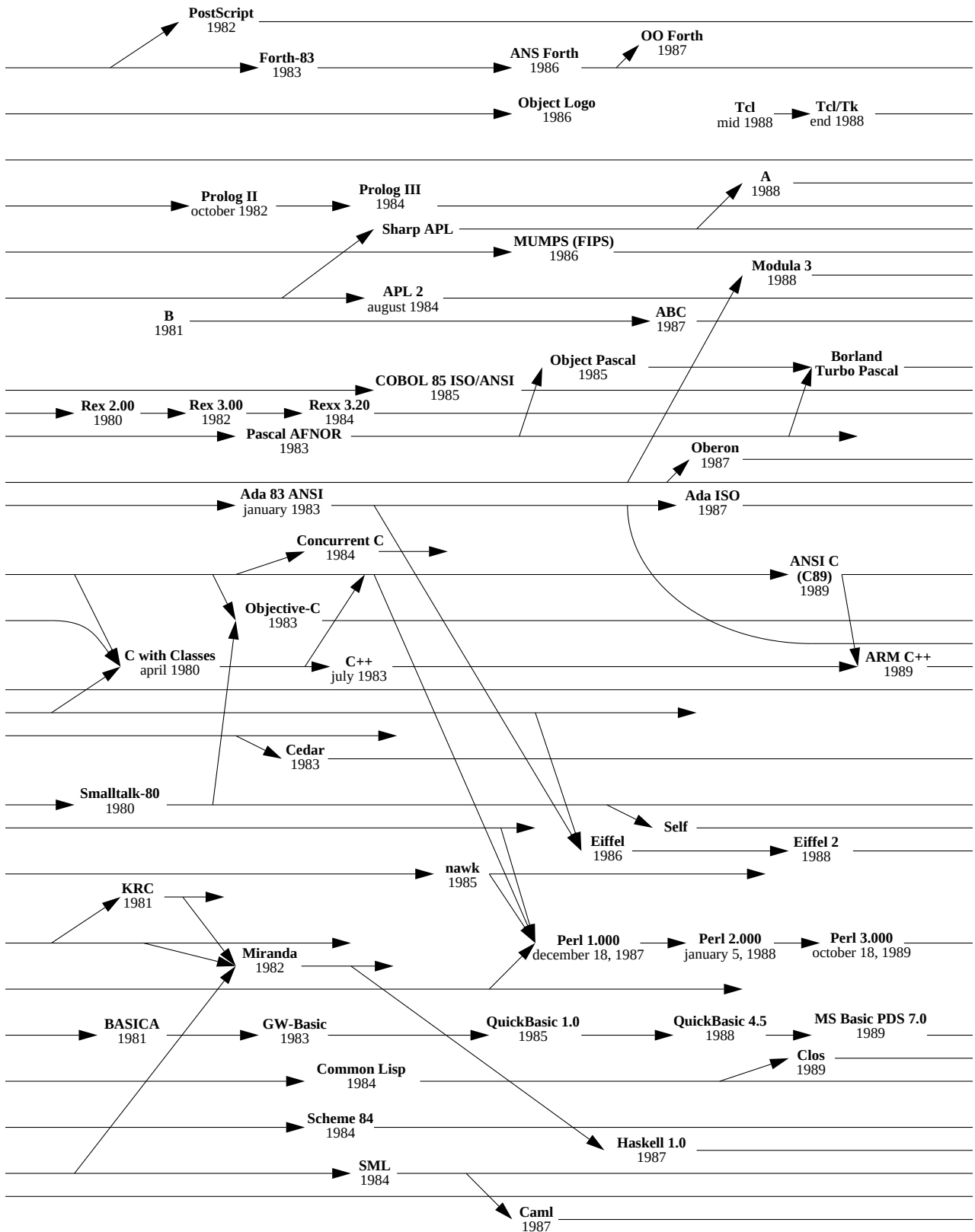
1970

1975

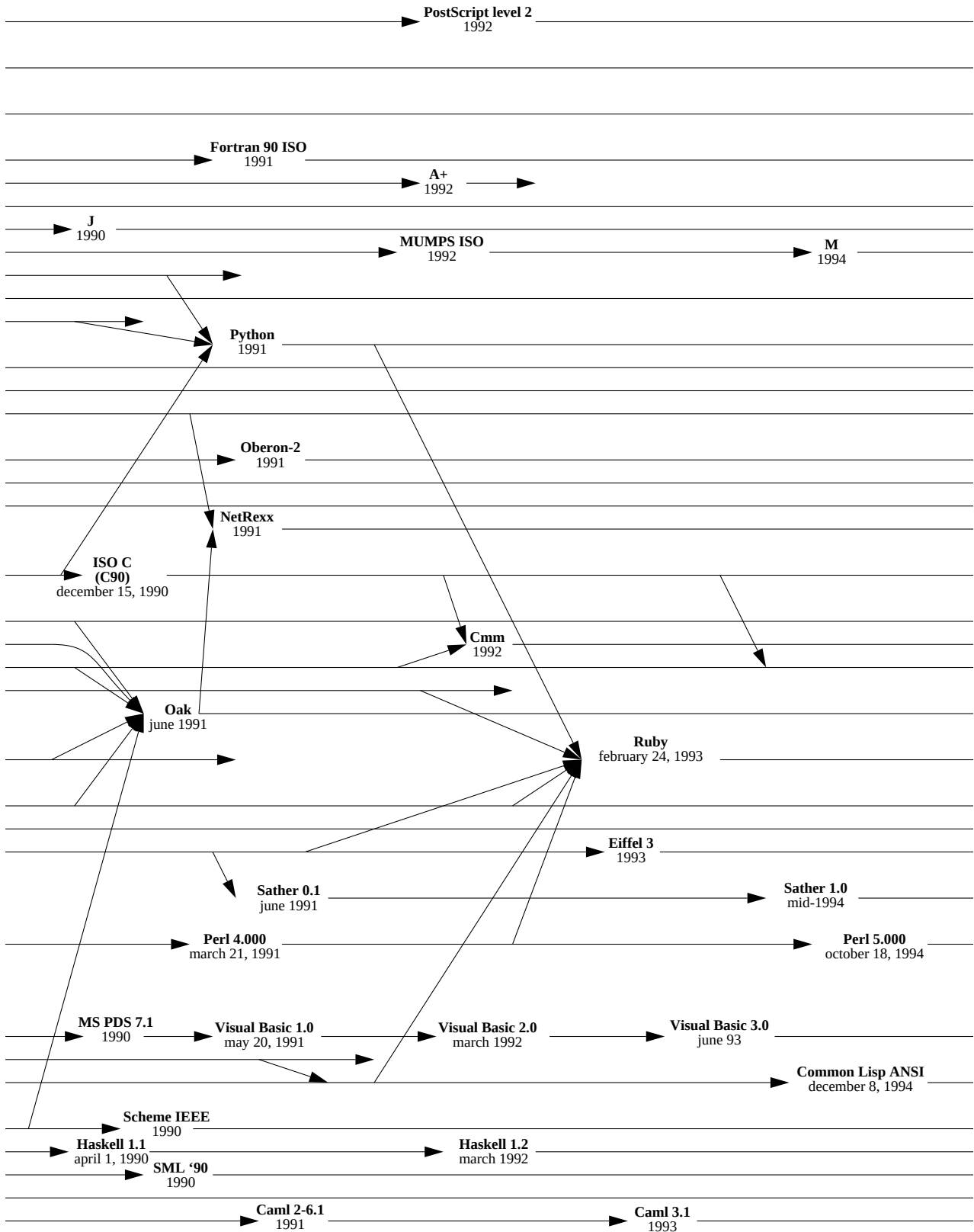


1980

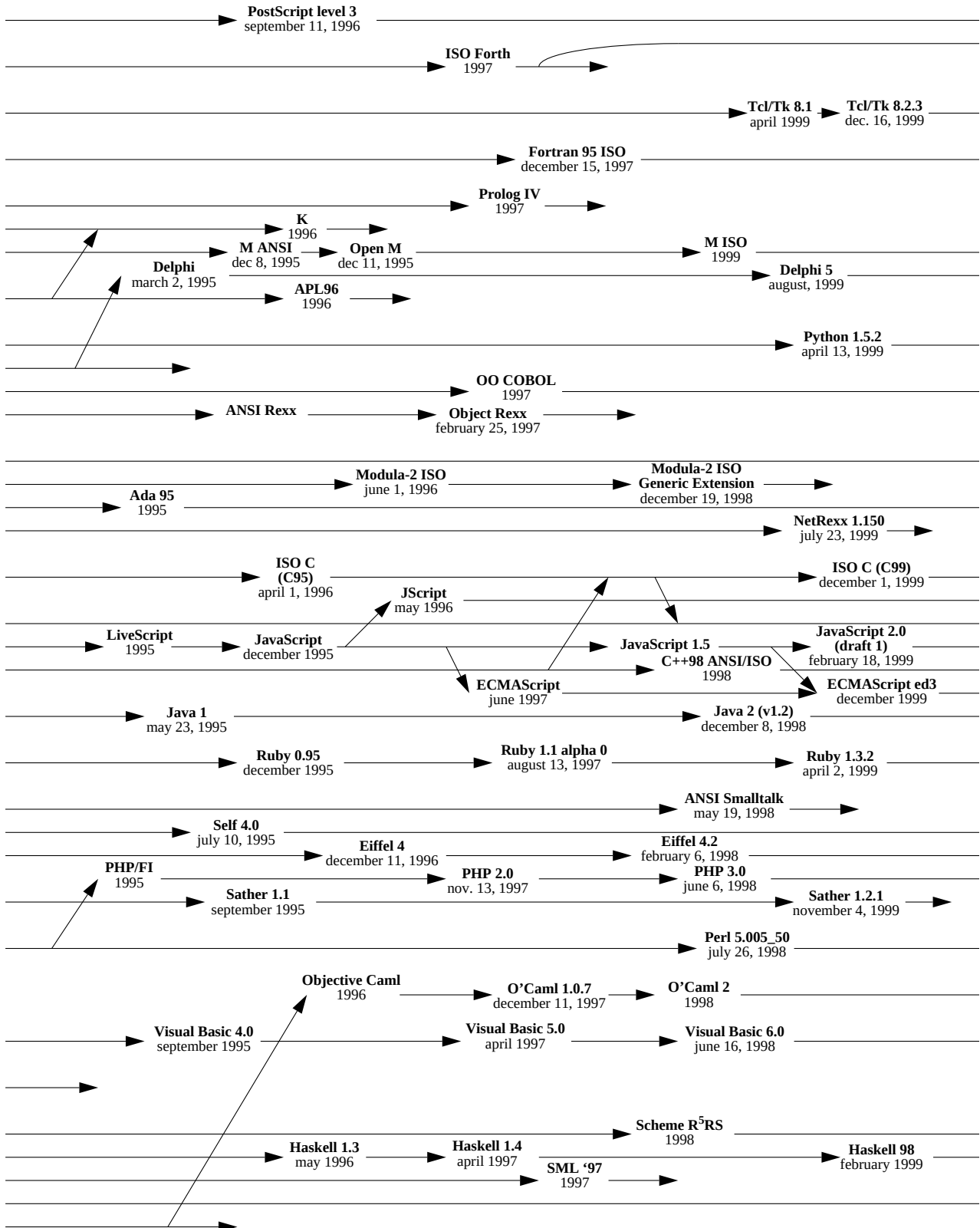
1985



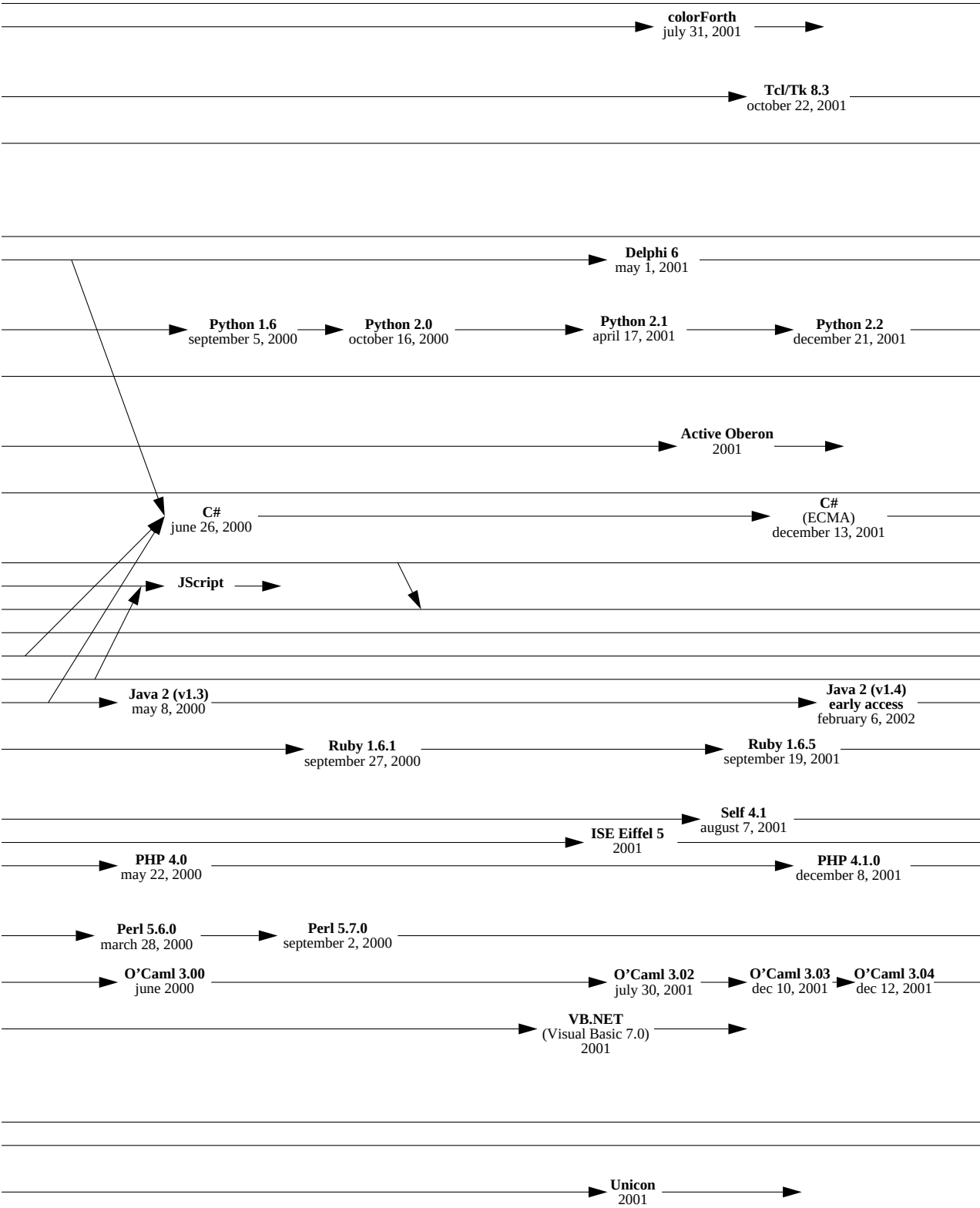
1990



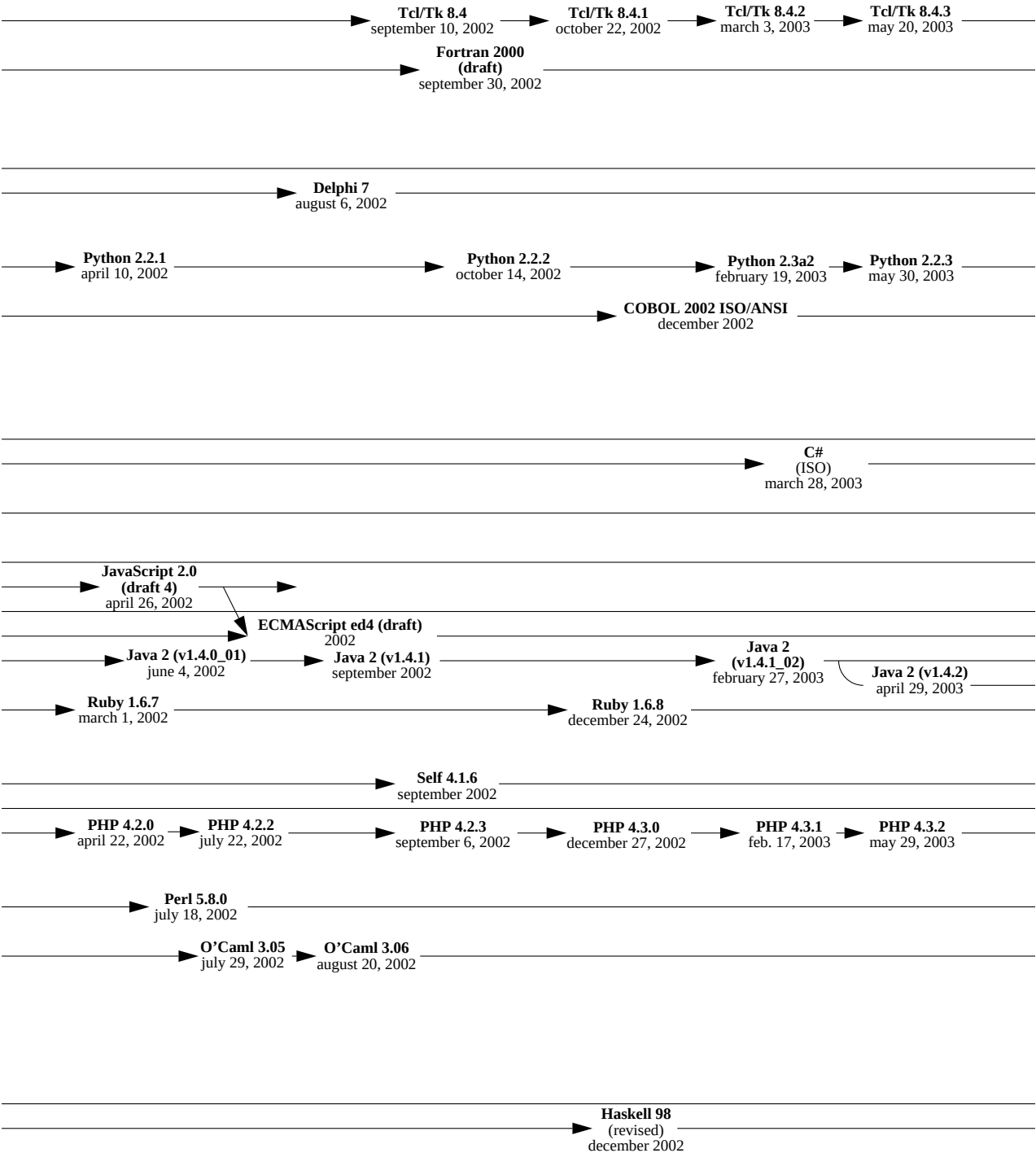
1995



2000

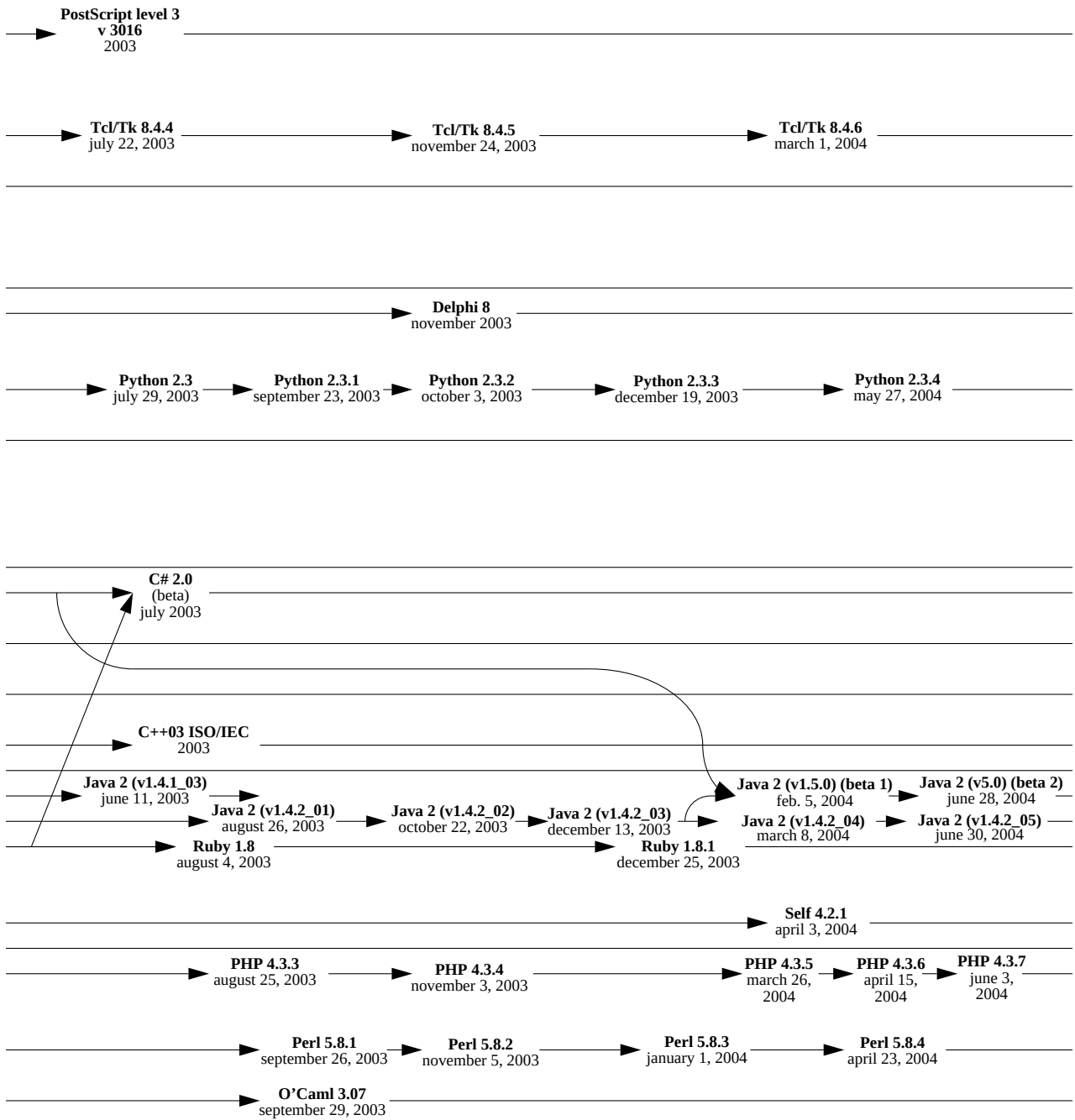


2002

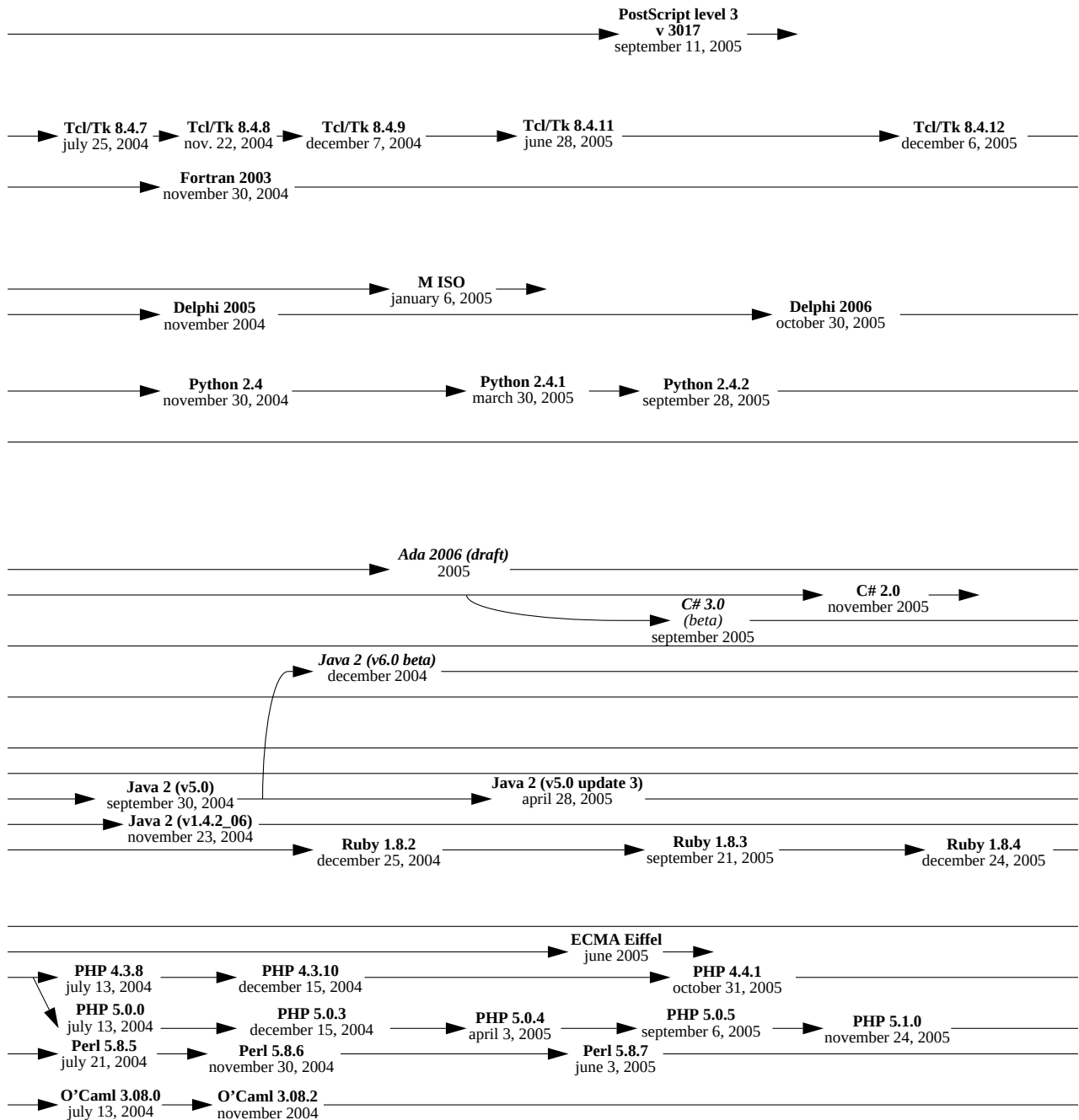


2003

2004

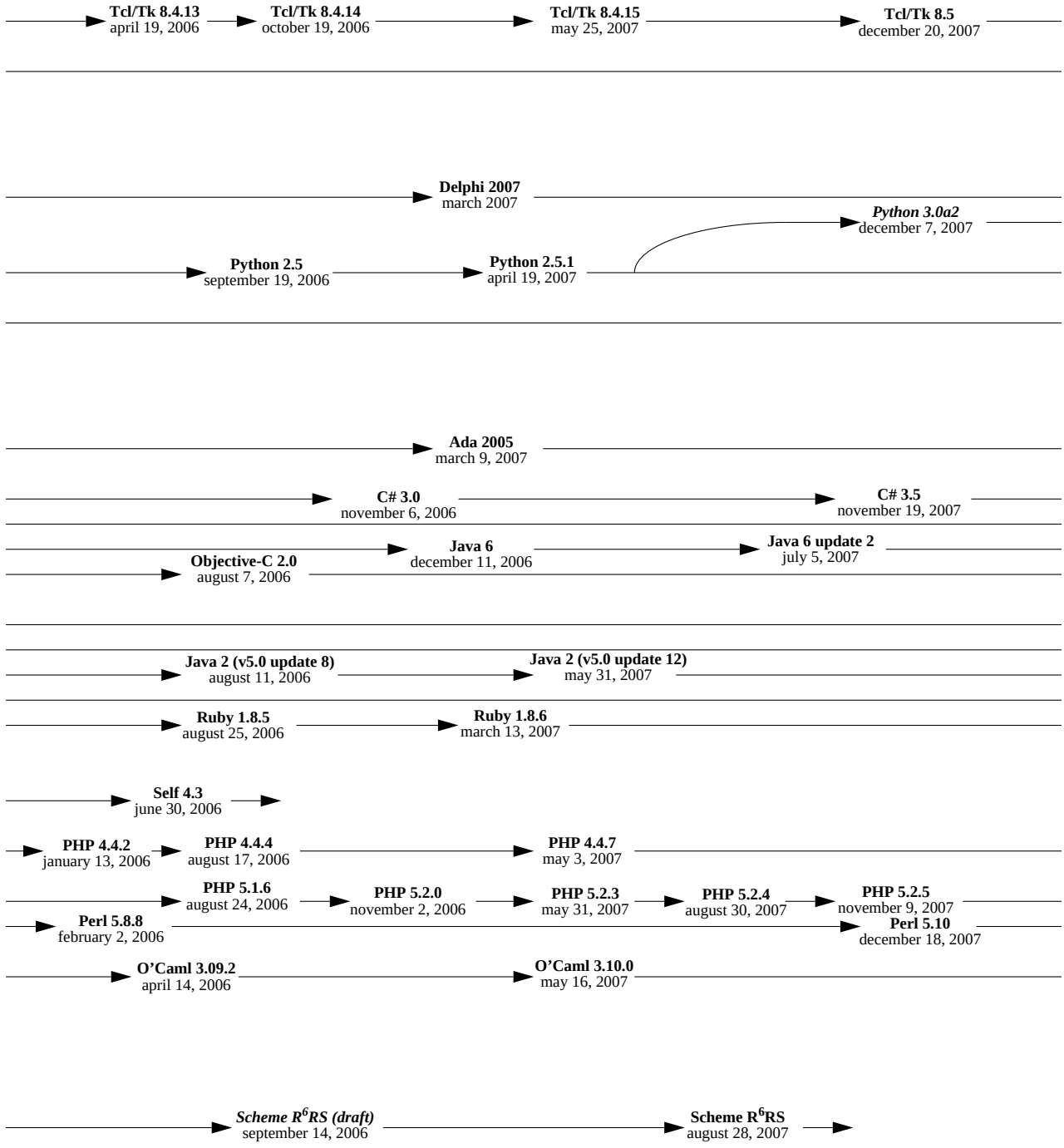


2005



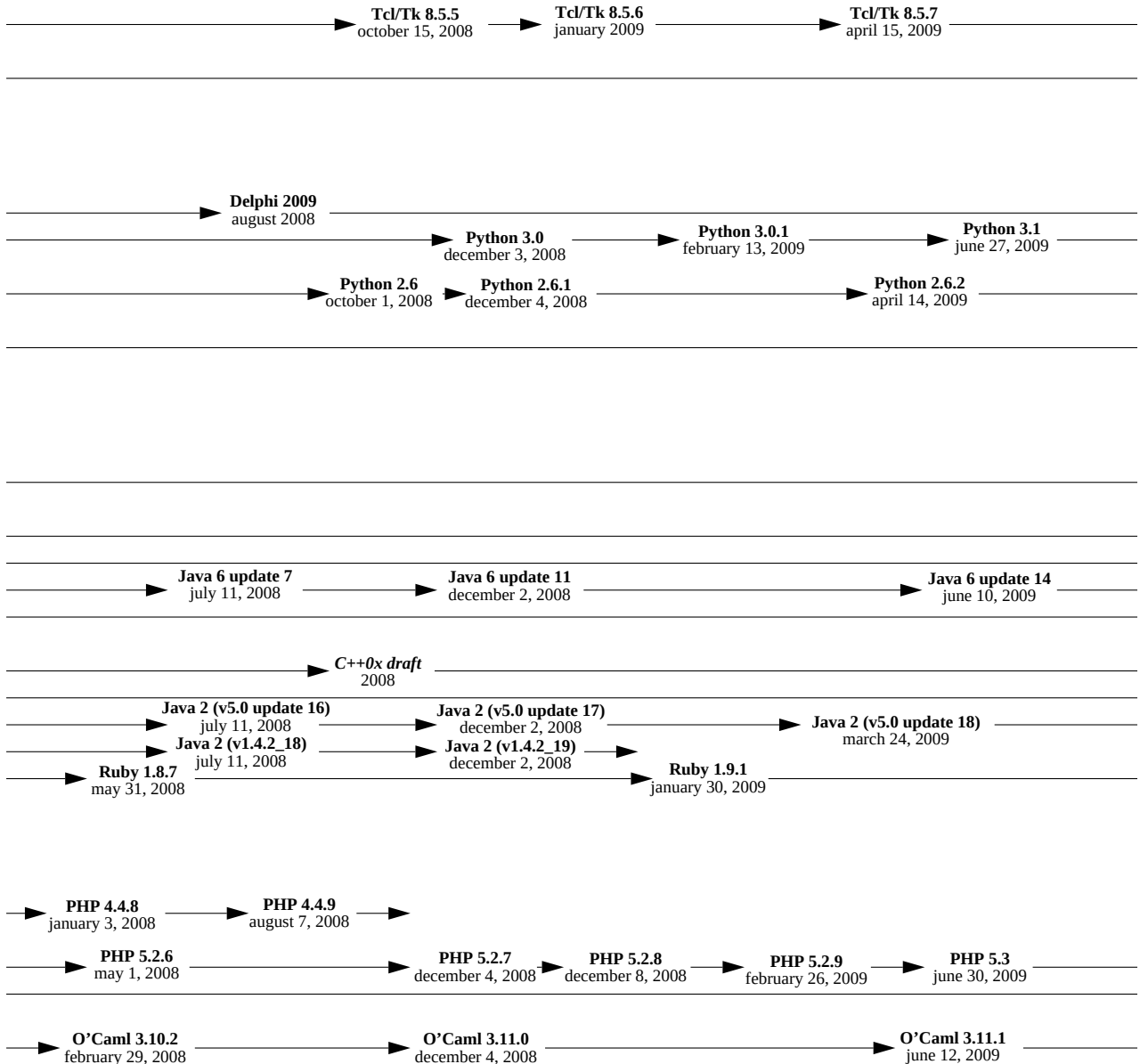
2006

2007



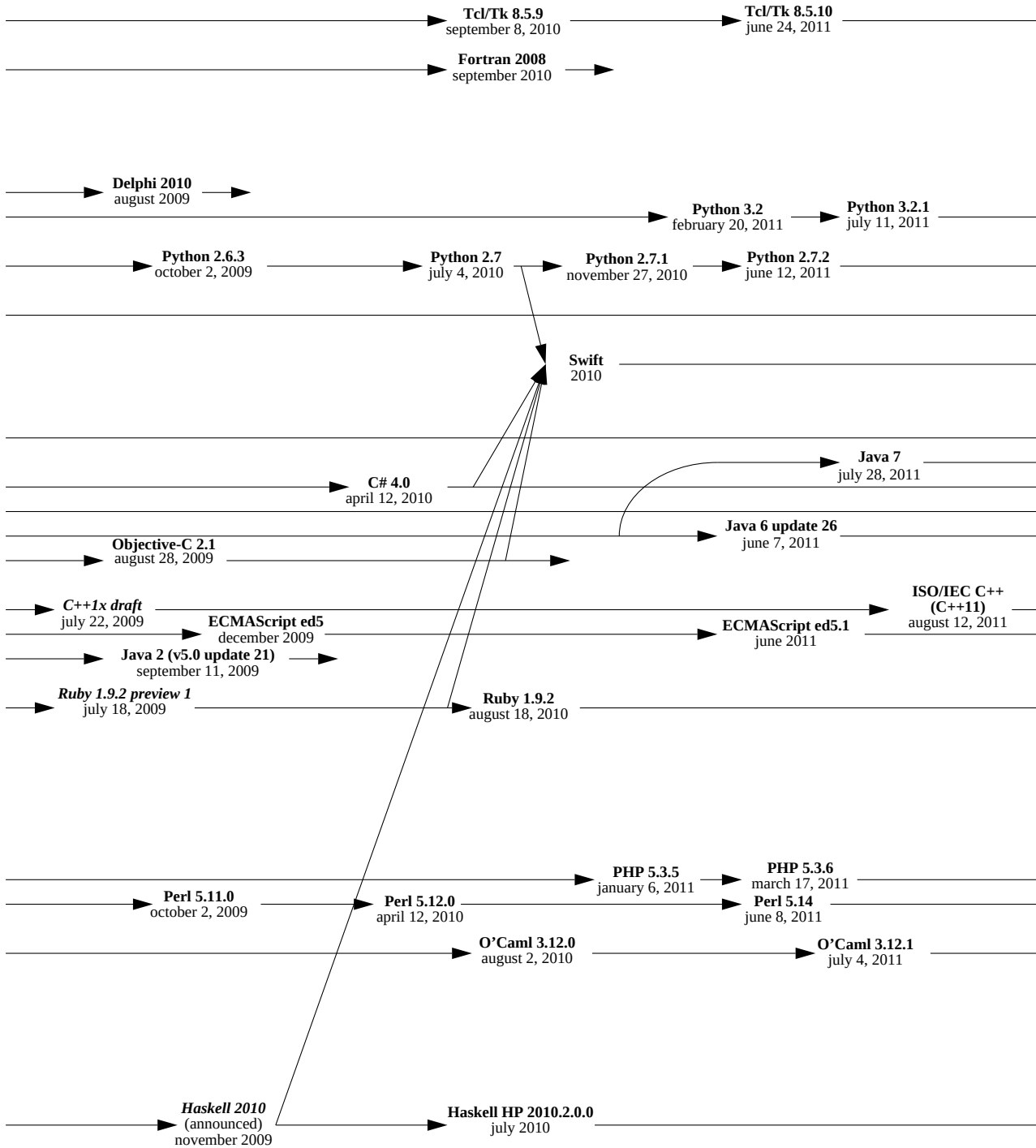
2008

2009



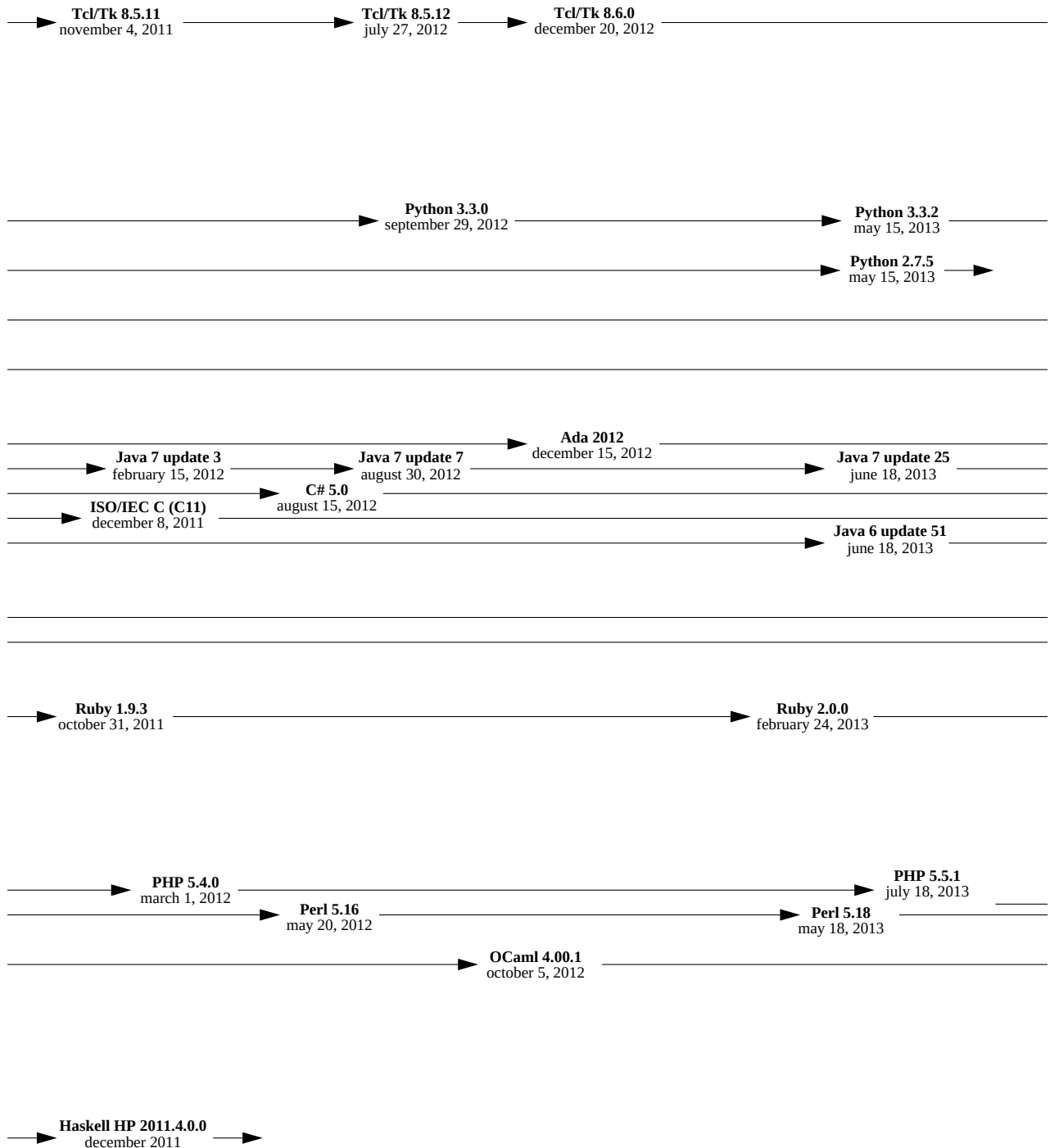
2010

2011

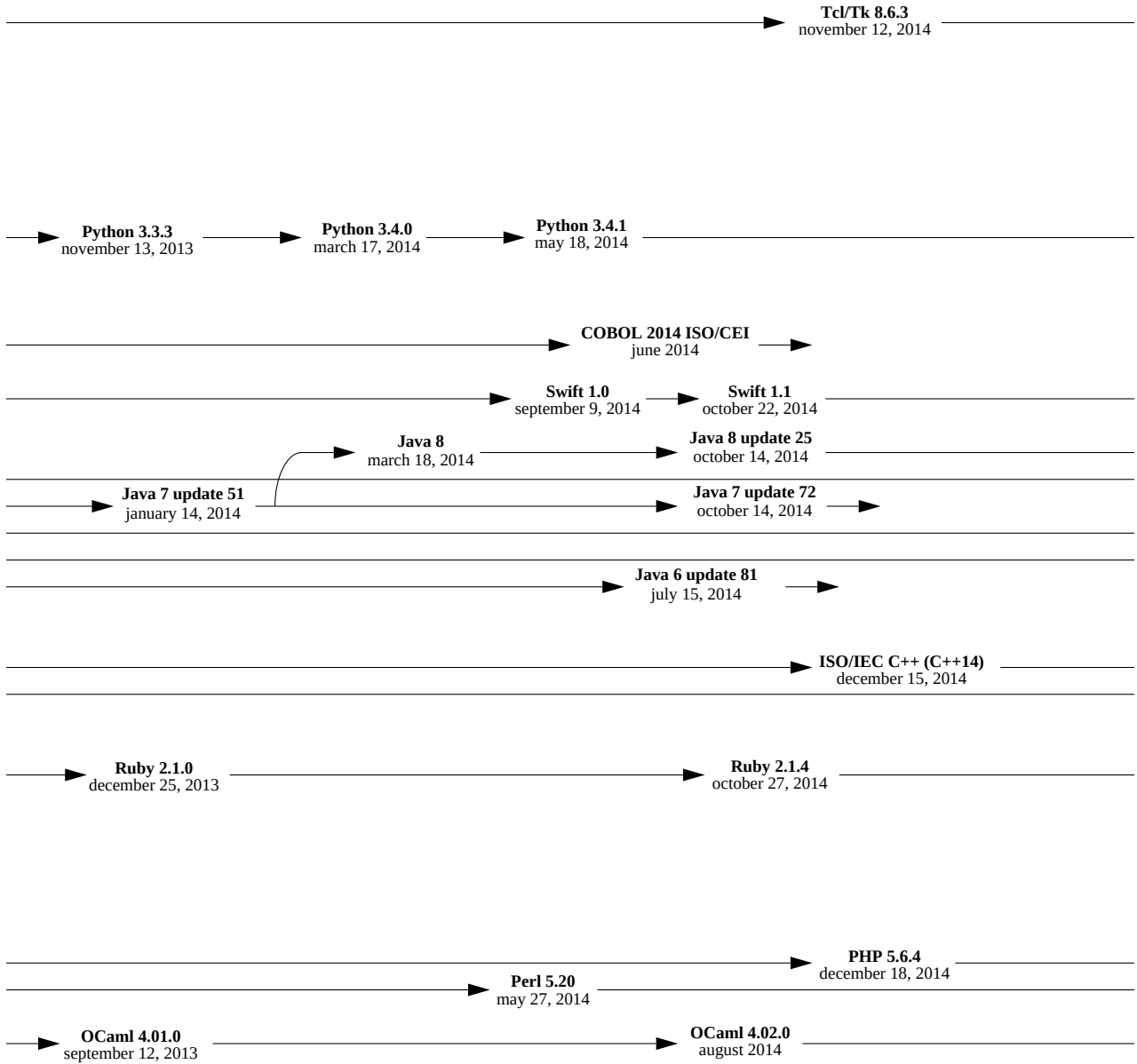


2012

2013

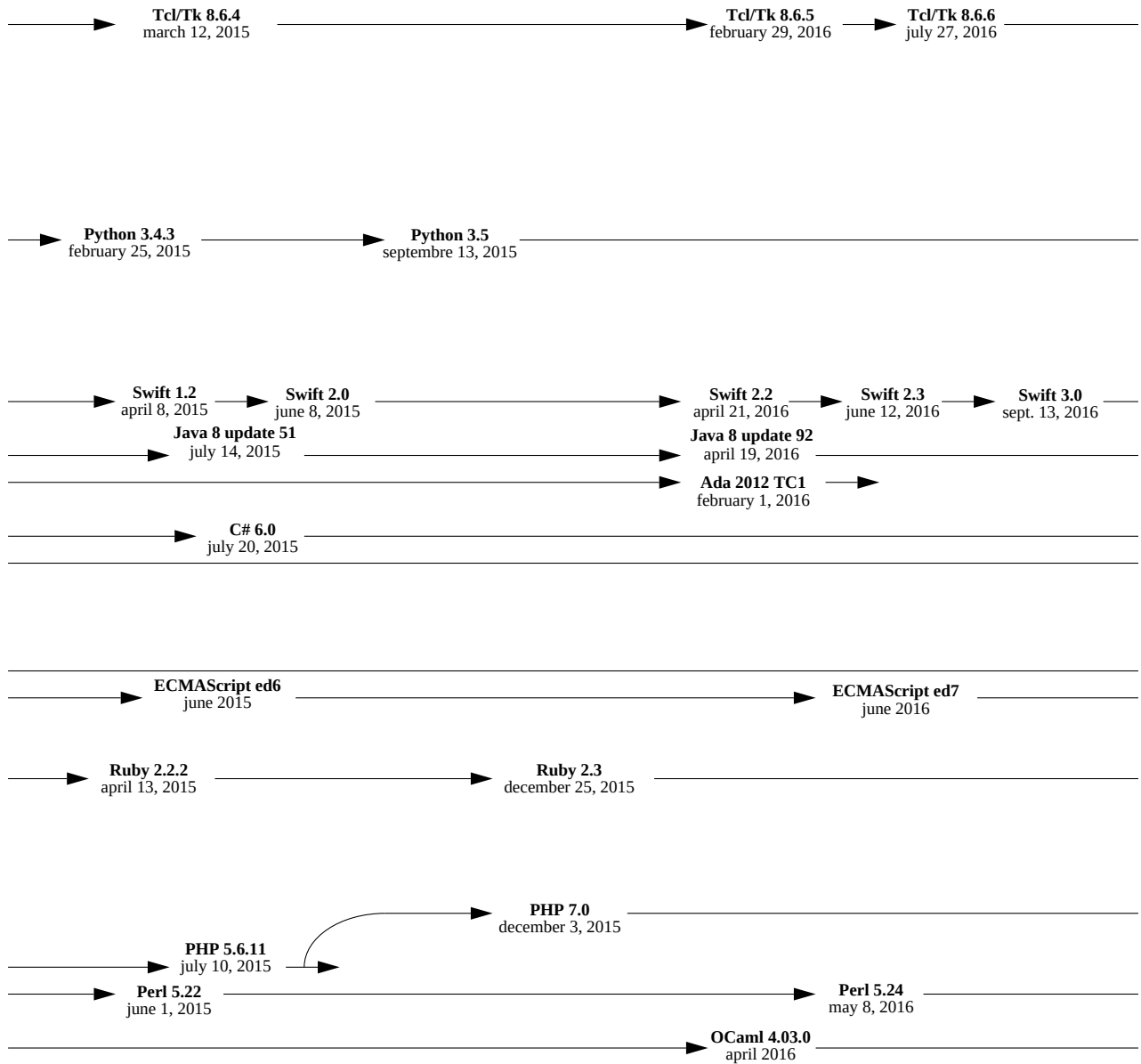


2014



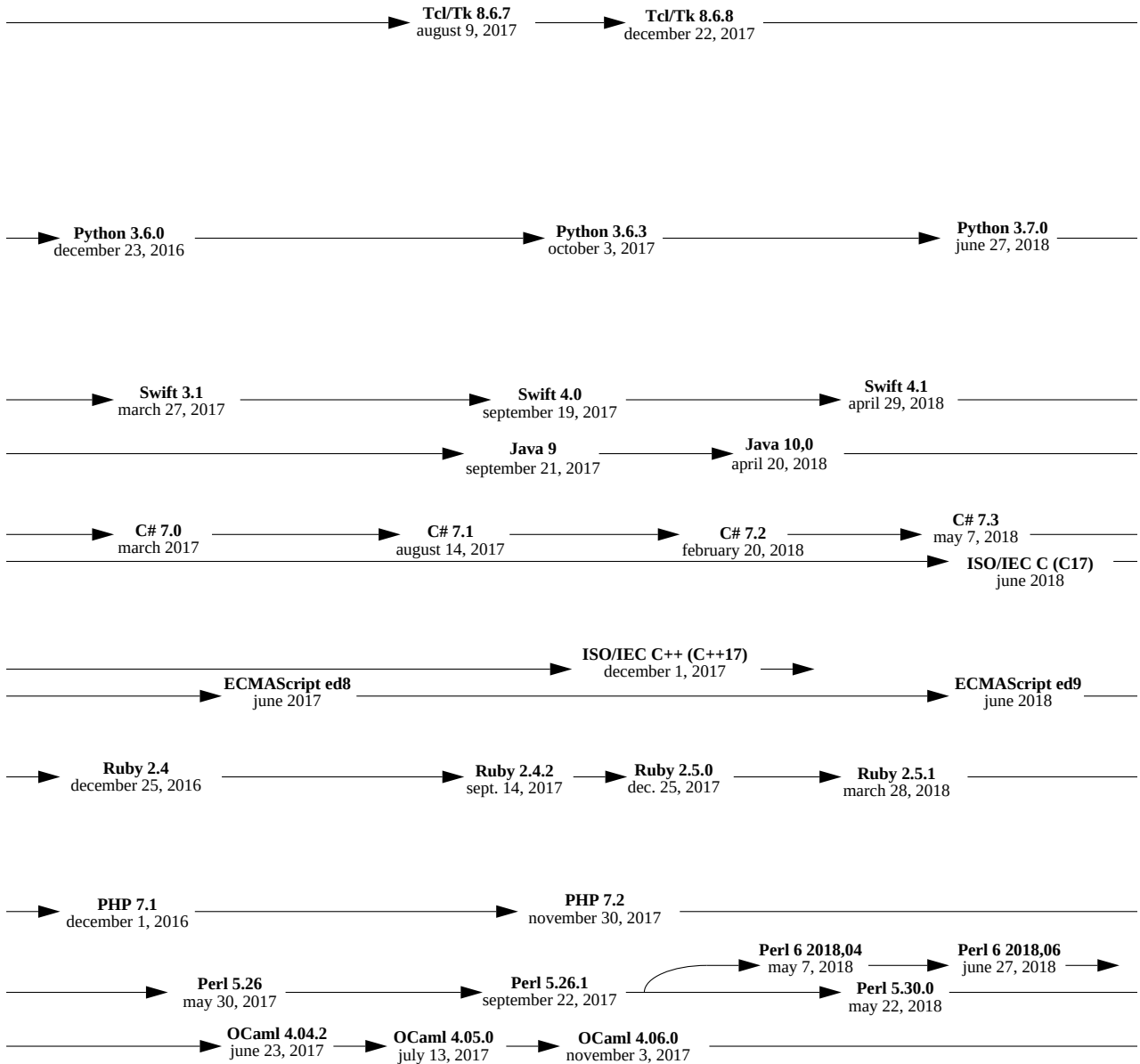
2015

2016

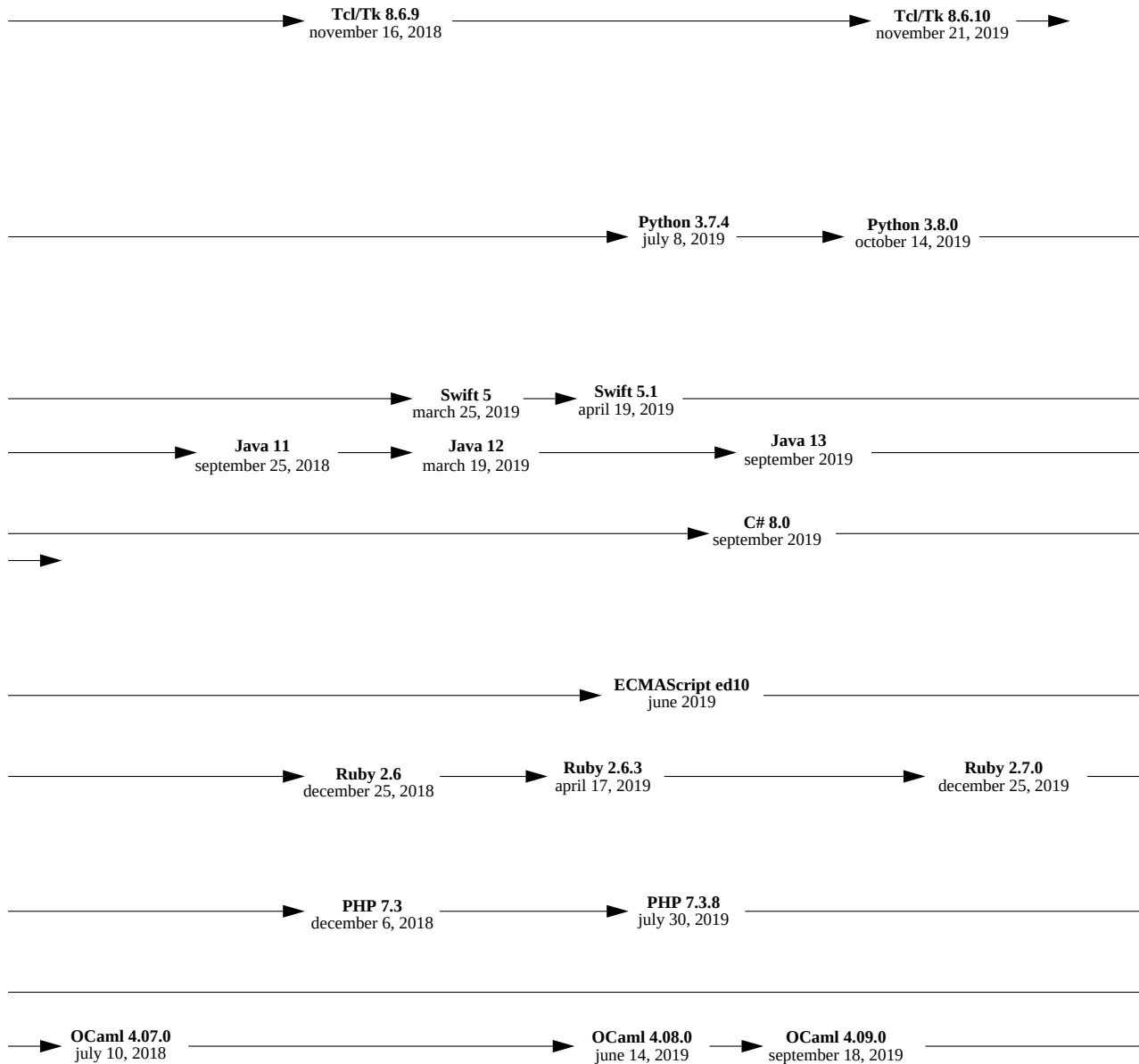


2017

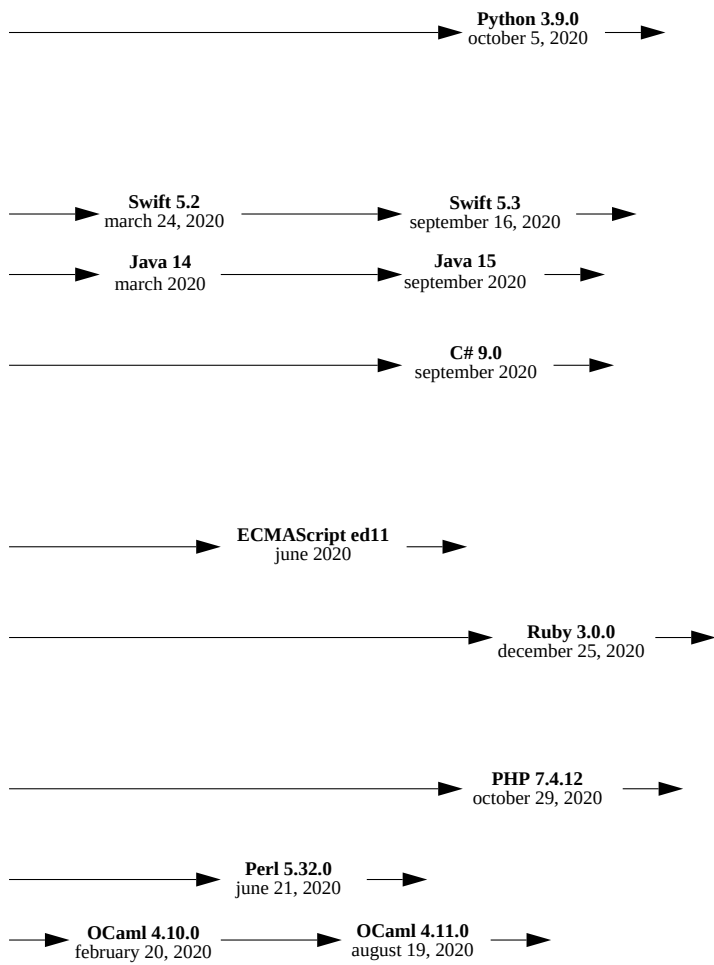
2018



2019



2020



Anhang B.

Die Sprache C

B.1. Schlüsselwörter

C99 kennt folgende Schlüsselwörter:

auto	enum	restrict ¹	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while
const	goto	sizeof	_Bool ¹
continue	if	static	_Complex ¹
default	inline	struct	_Imaginary ¹
do	int	switch	
double	long	typedef	
else	register	union	



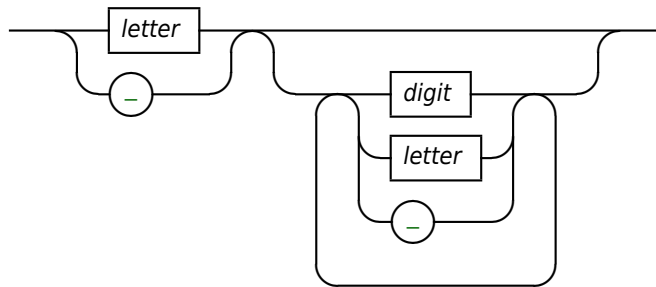
In C11 kommen noch folgende Schlüsselwörter dazu:

alignof	_Atomic	_Noreturn	_Thread_local
_Alignas	_Generic	_Static_assert	

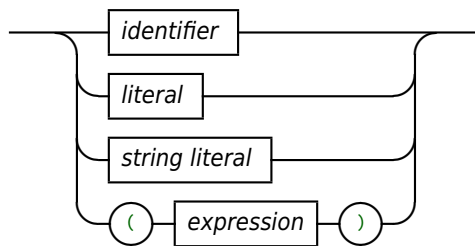
¹Nicht in C90

B.2. Syntax

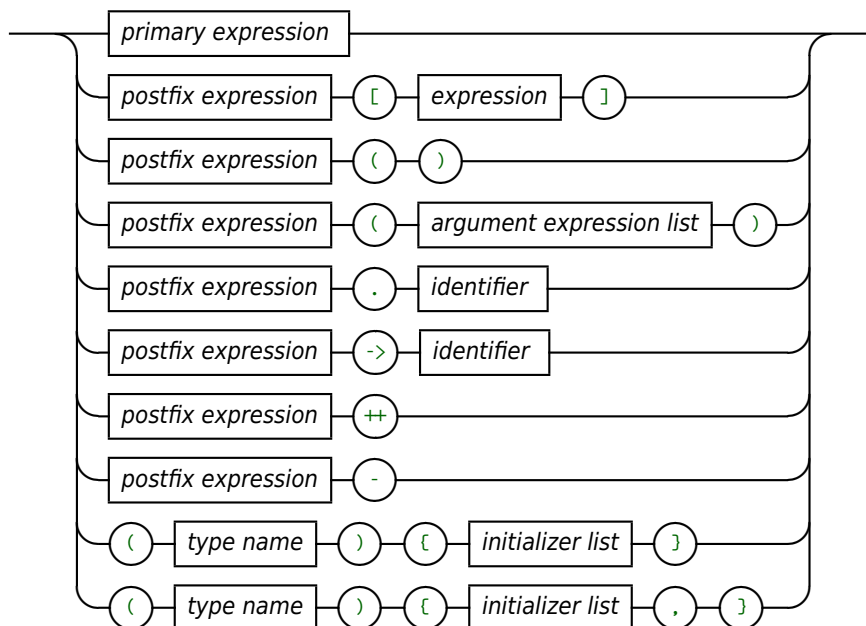
identifier

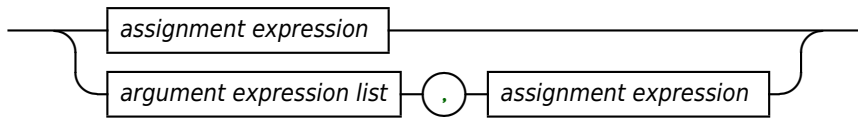
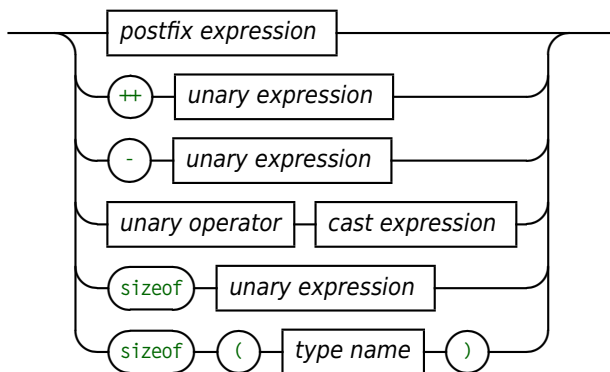
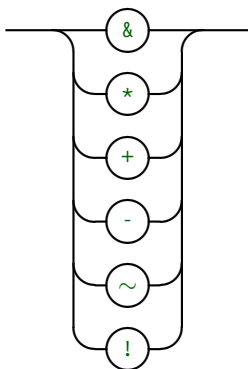
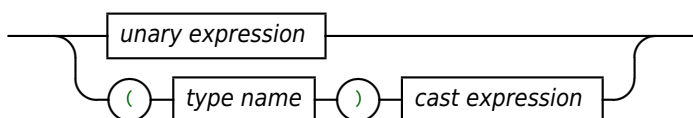


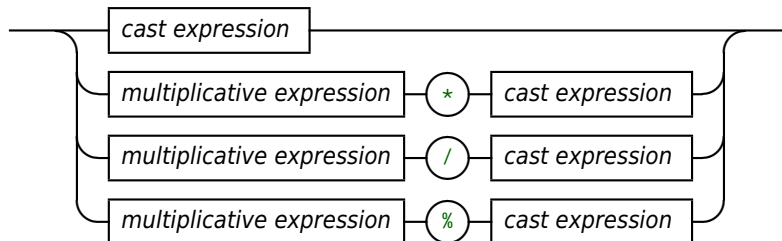
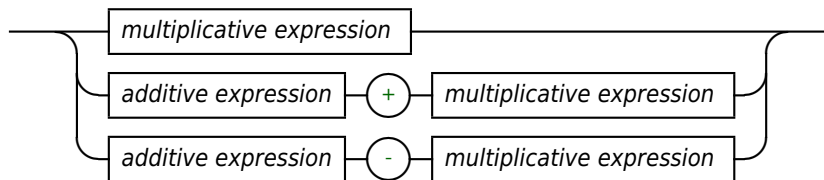
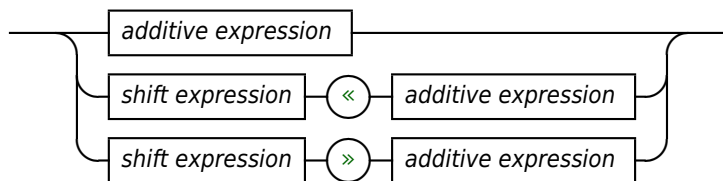
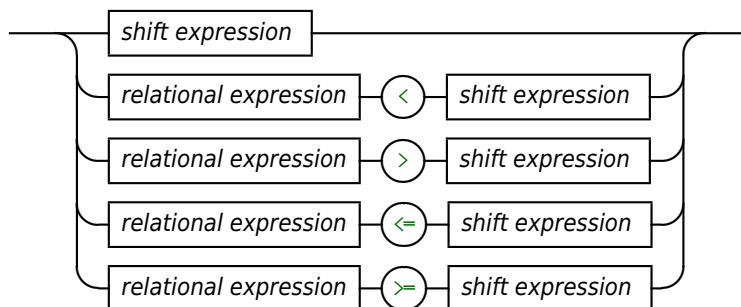
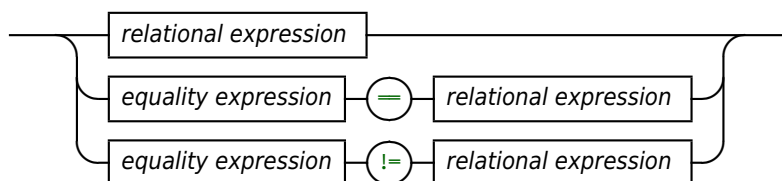
primary expression



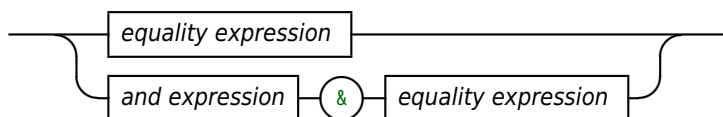
postfix expression



argument expression list*unary expression**unary operator**cast expression*

multiplicative expression*additive expression**shift expression**relational expression**equality expression*

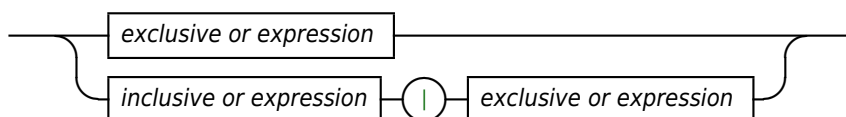
and expression



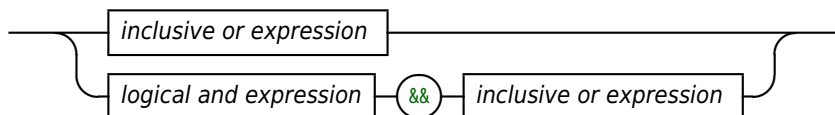
exclusive or expression



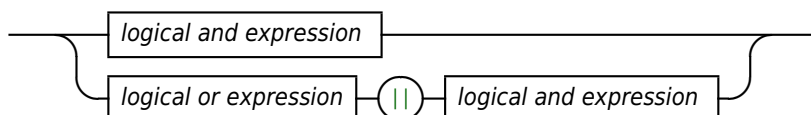
inclusive or expression



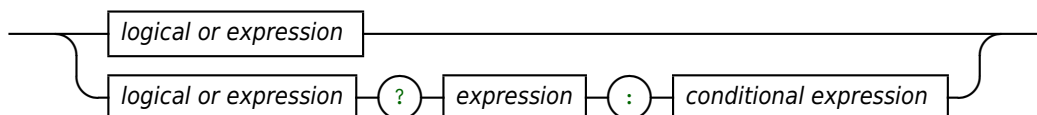
logical and expression



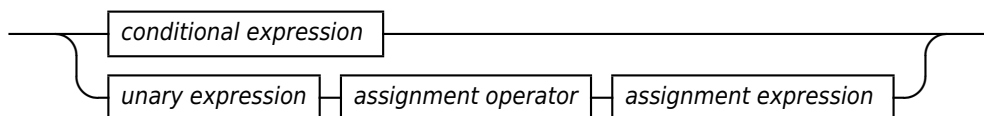
logical or expression



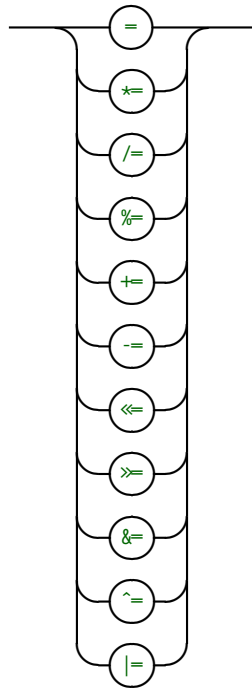
conditional expression



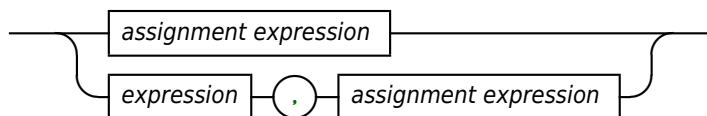
assignment expression



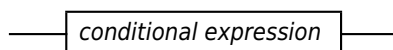
assignment operator



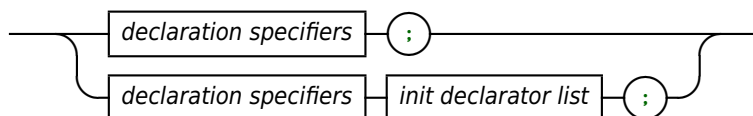
expression

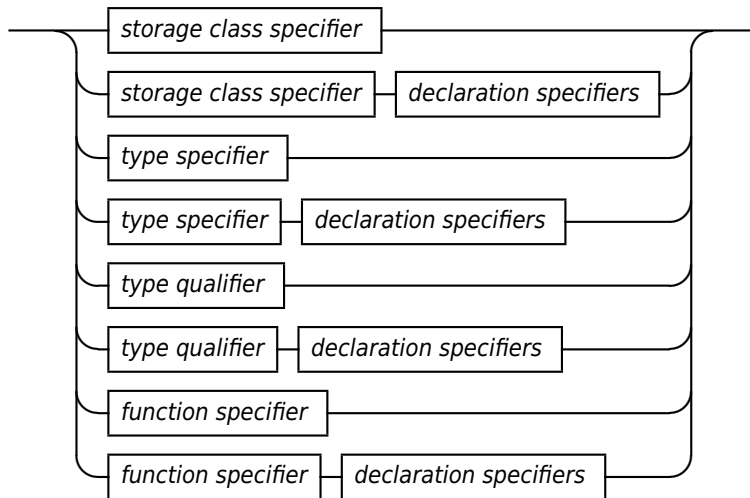
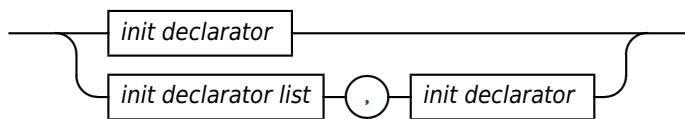
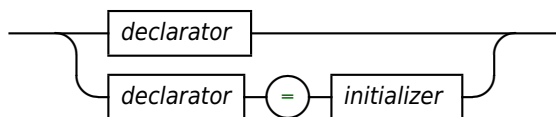
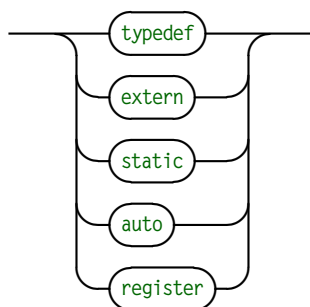


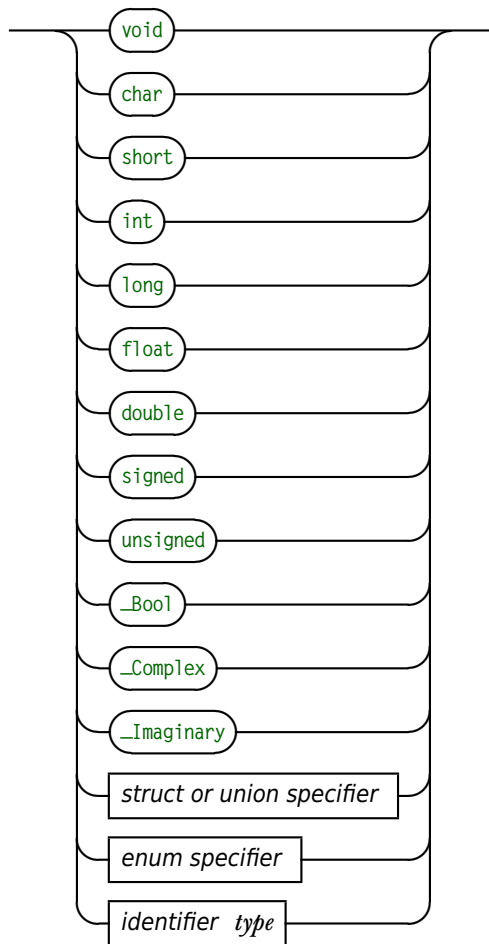
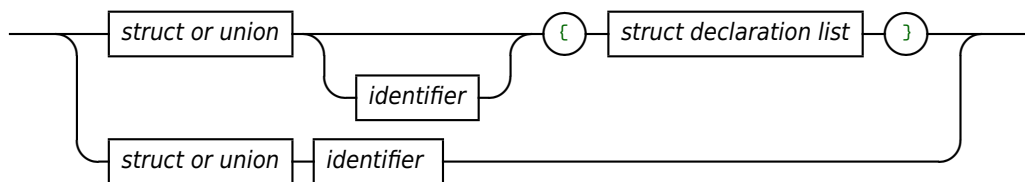
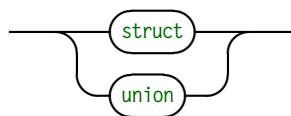
constant expression

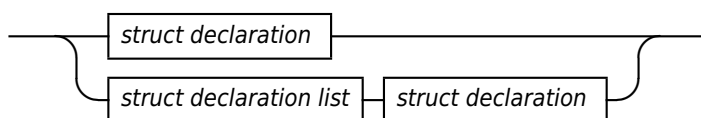
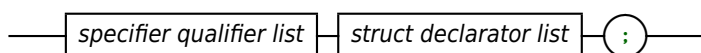
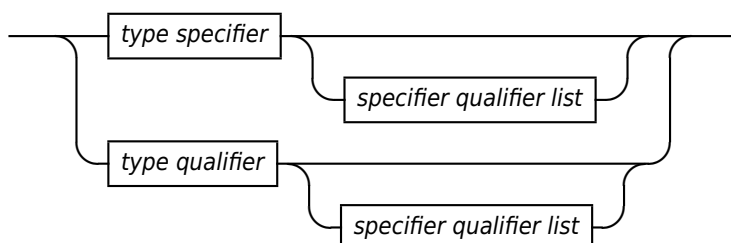
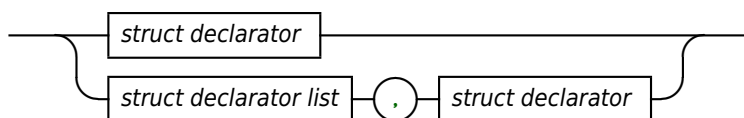
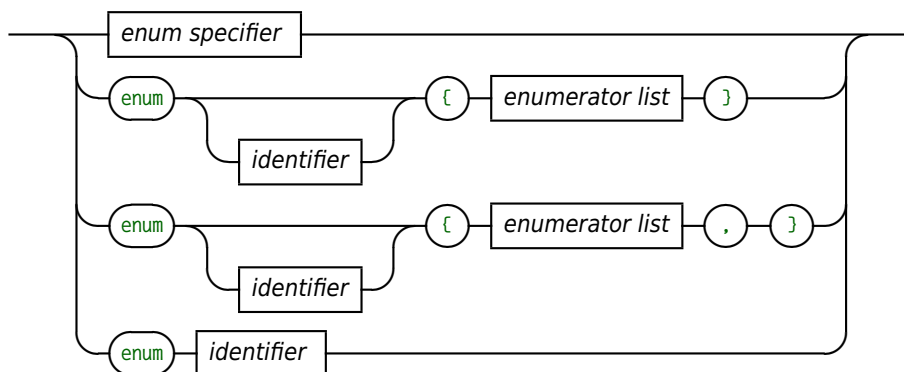
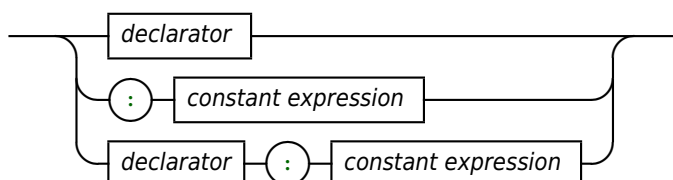


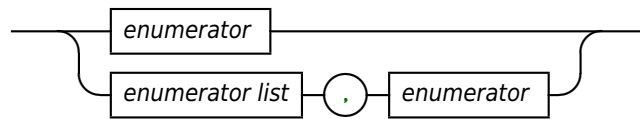
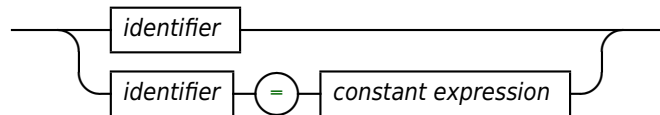
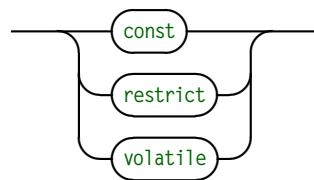
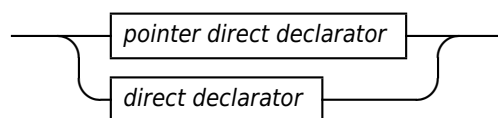
declaration

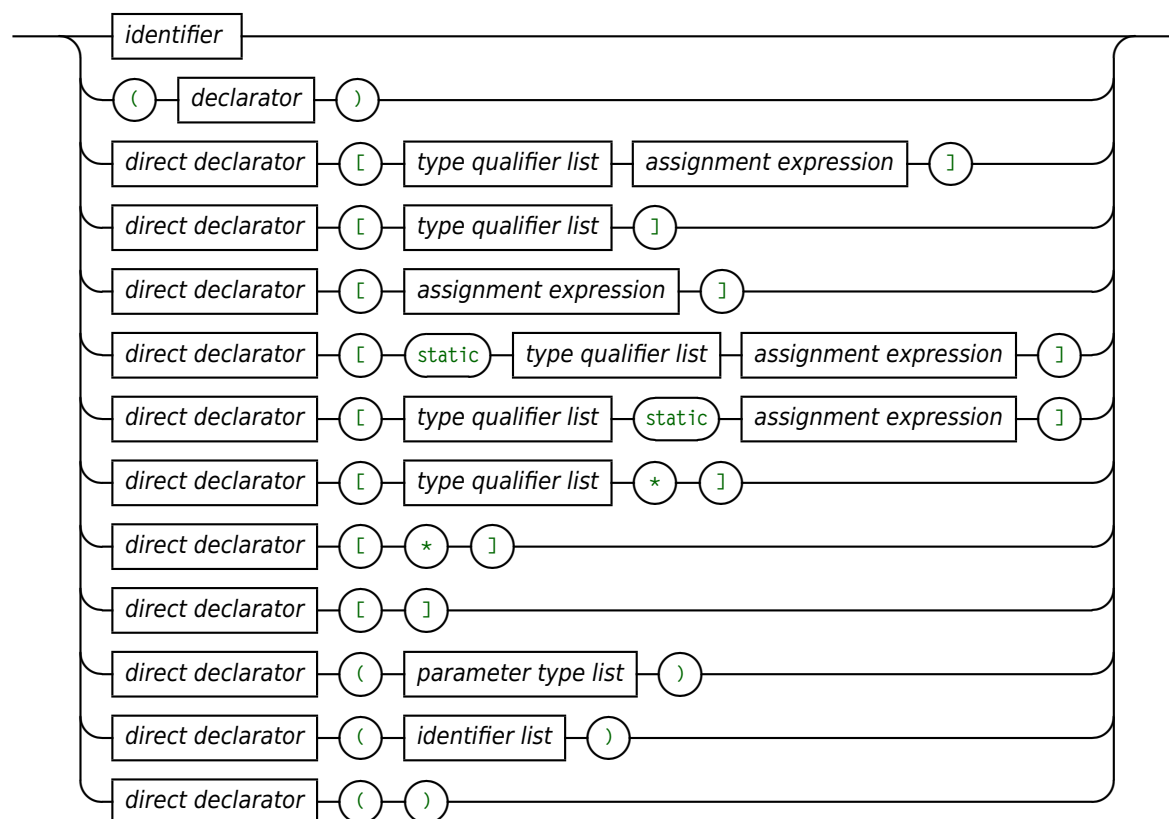
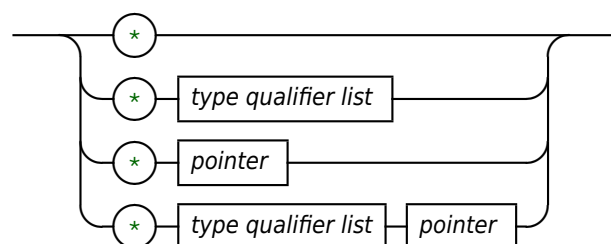
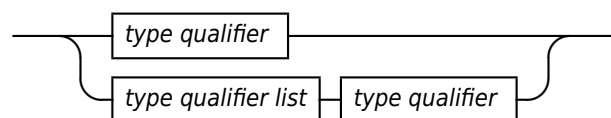


declaration specifiers*init declarator list**init declarator**storage class specifier*

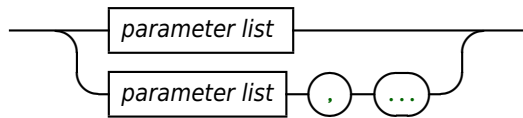
type specifier*struct or union specifier**struct or union*

struct declaration list*struct declaration**specifier qualifier list**struct declarator list**struct declarator*

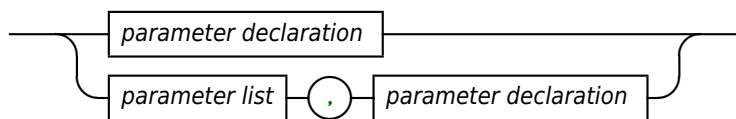
enumerator list*enumerator**type qualifier**function specifier**declarator*

direct declarator*pointer**type qualifier list*

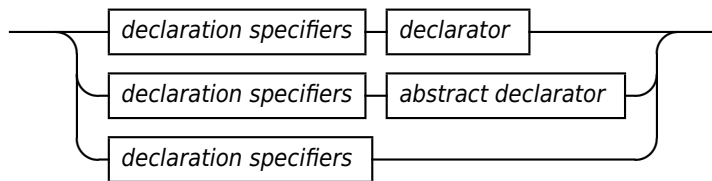
parameter type list



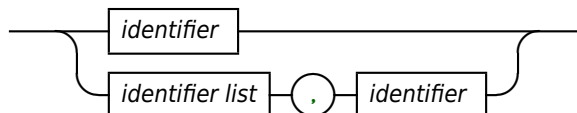
parameter list



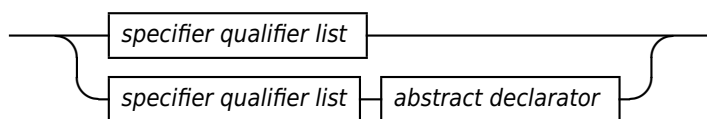
parameter declaration



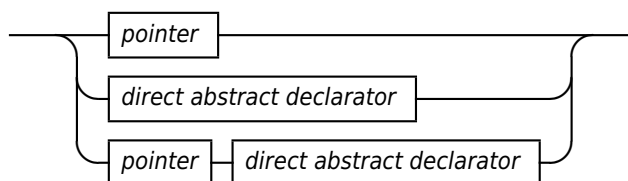
identifier list

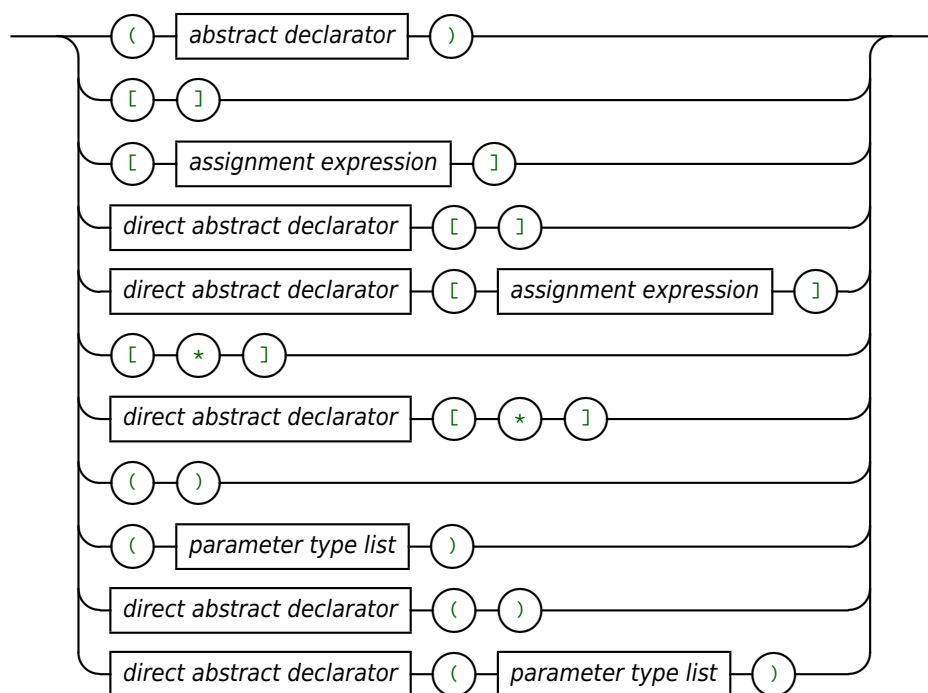
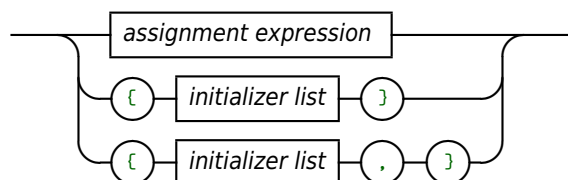
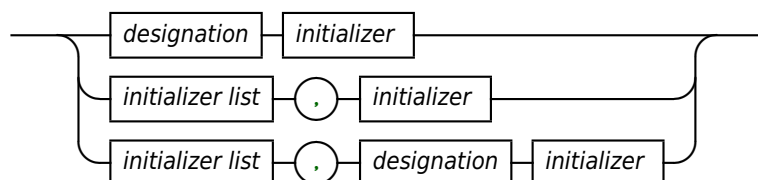
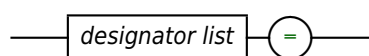


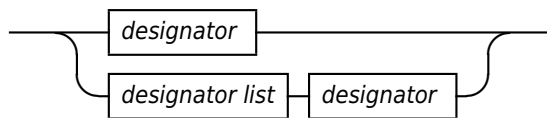
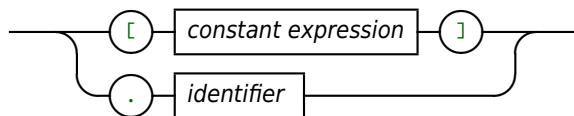
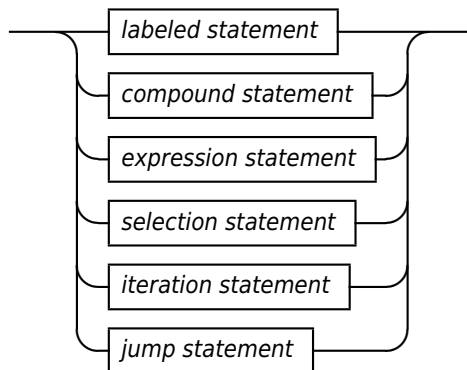
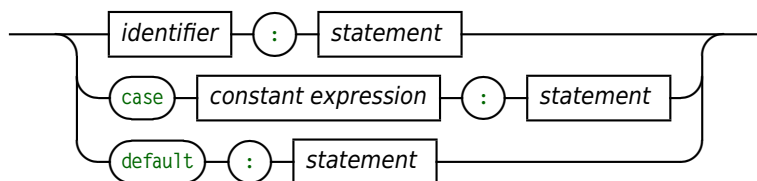
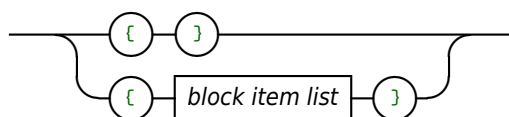
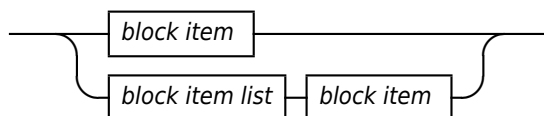
type name

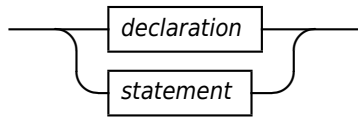
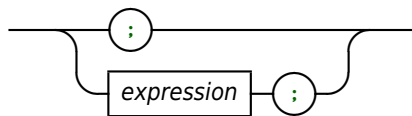
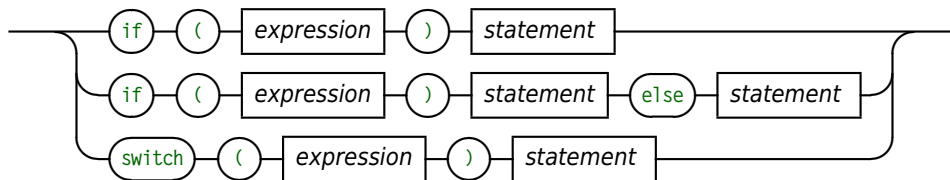
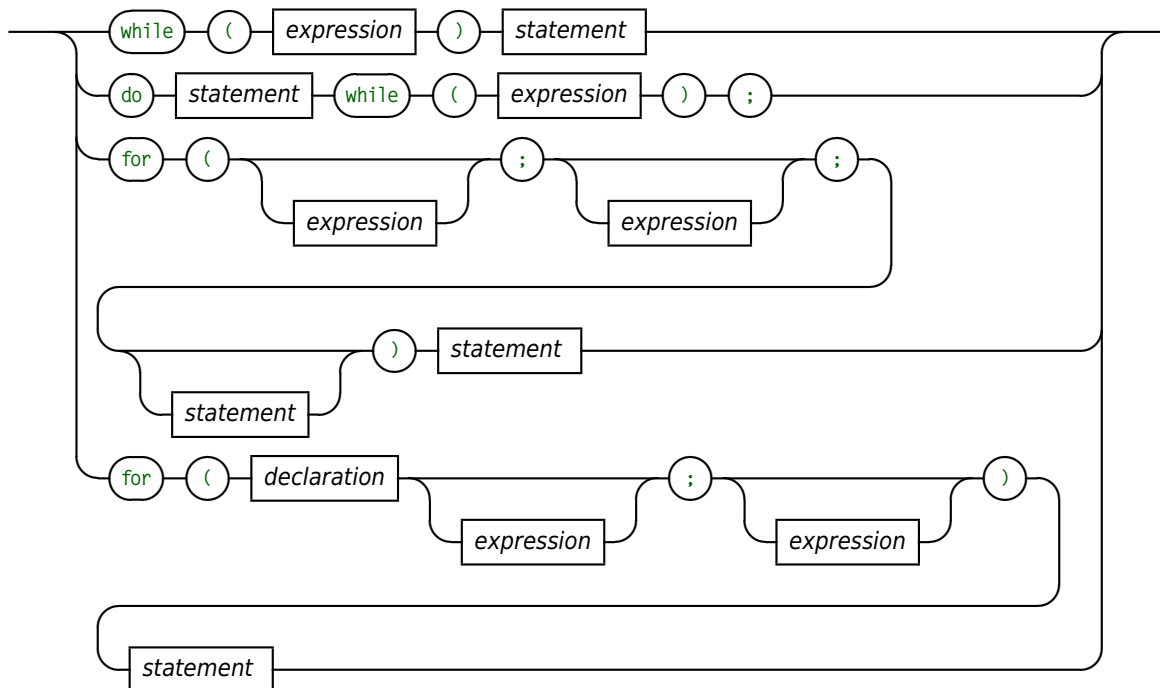


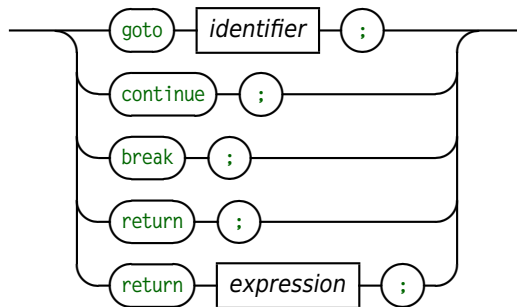
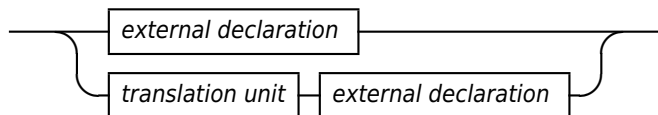
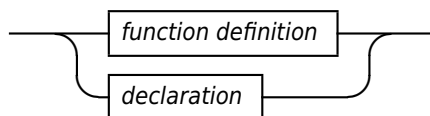
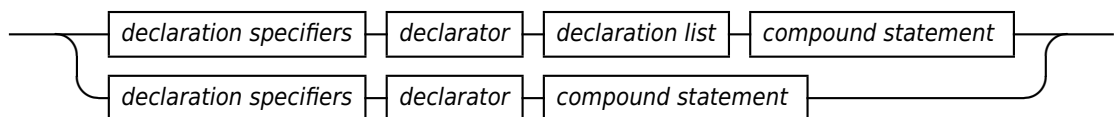
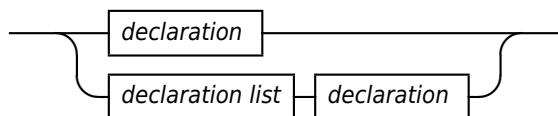
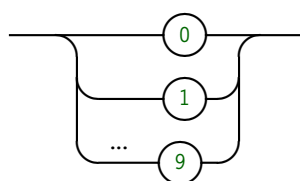
abstract declarator



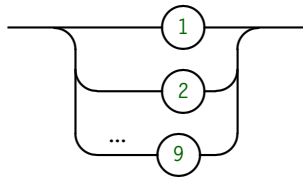
direct abstract declarator*initializer**initializer list**designation*

designator list*designator**statement**labeled statement**compound statement**block item list*

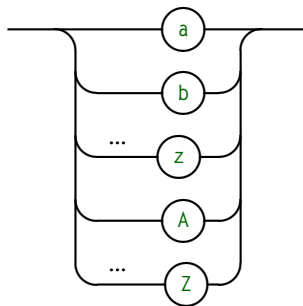
block item*expression statement**selection statement**iteration statement*

jump statement*translation unit**external declaration**function definition**declaration list**digit*

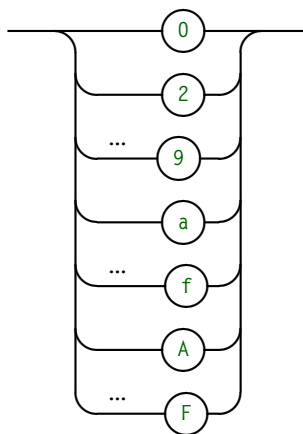
nonzero digit



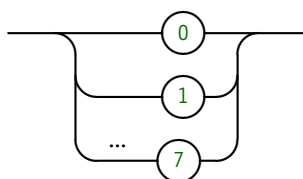
letter



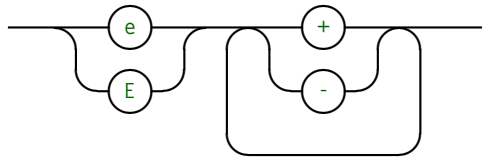
hexadecimal digit



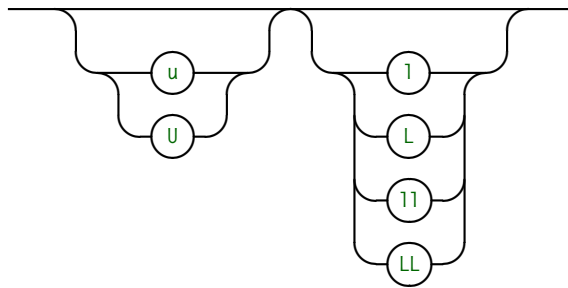
octal digit



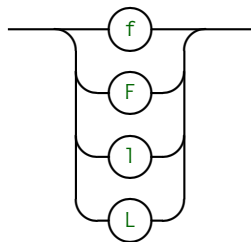
exponent prefix



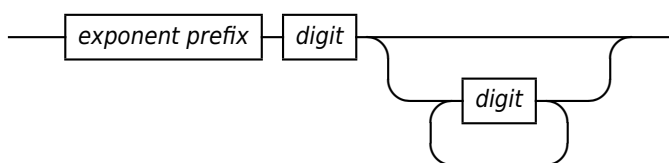
integer suffix

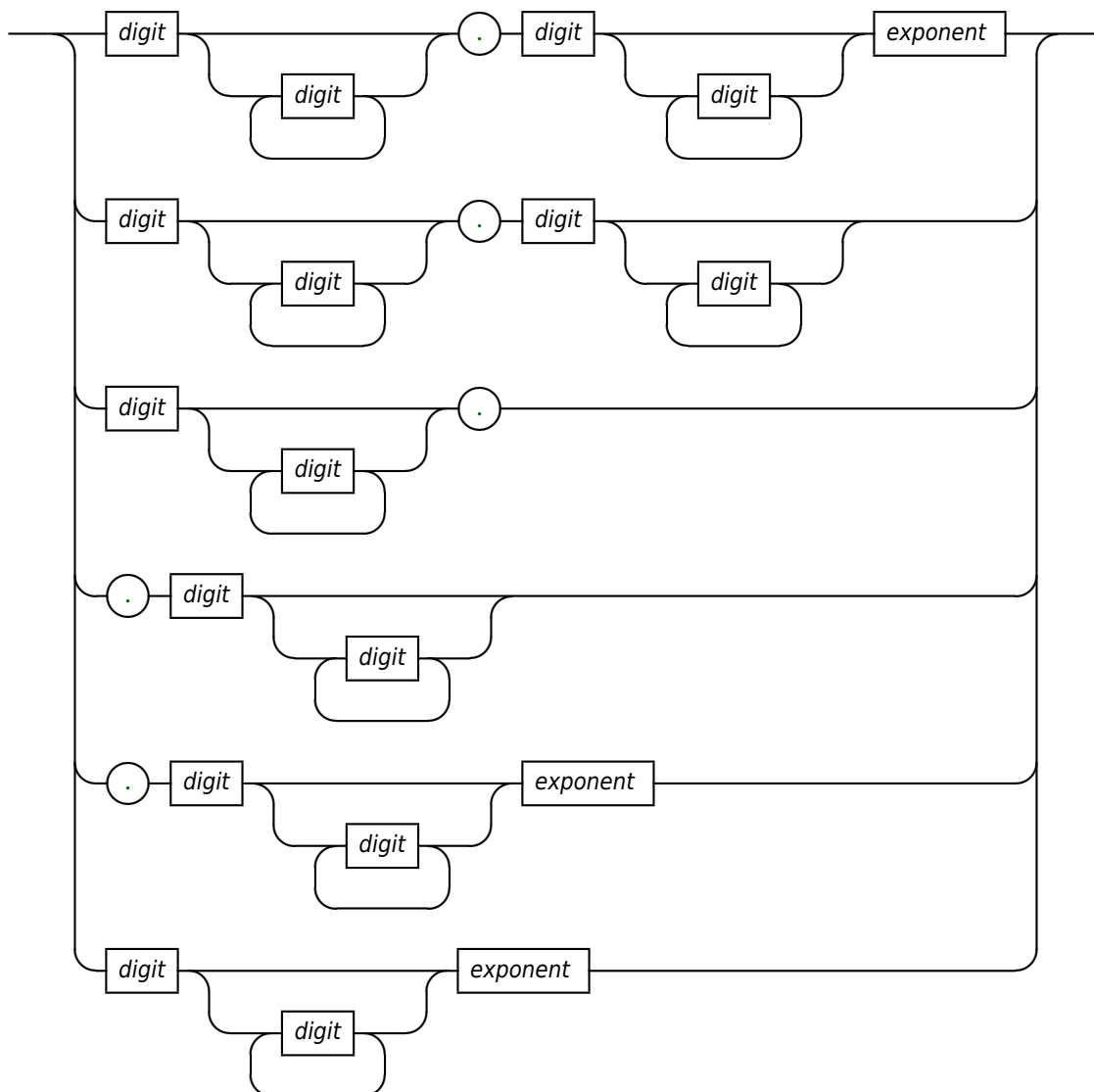
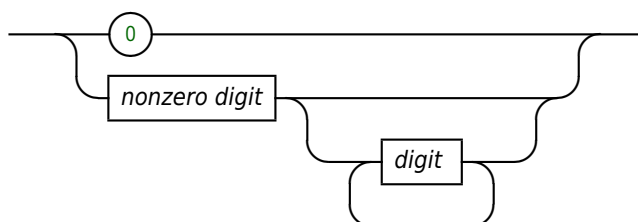


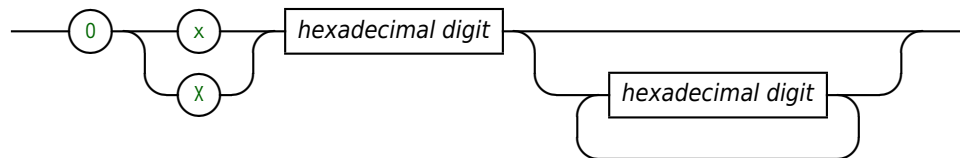
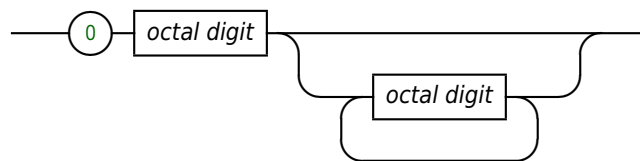
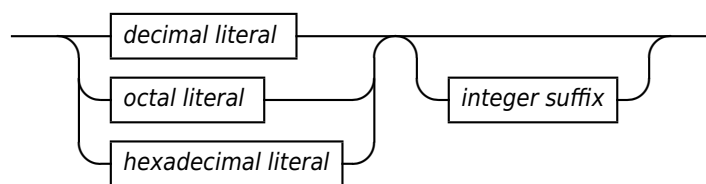
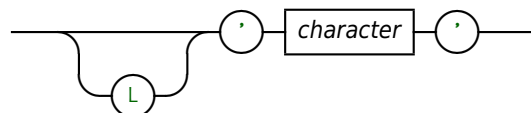
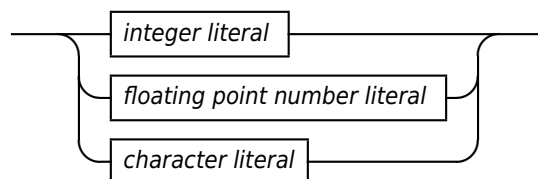
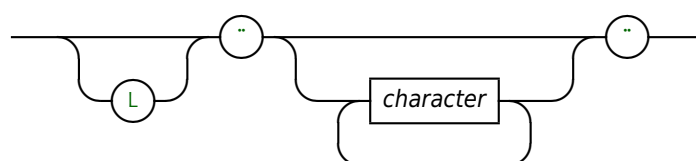
floating point number suffix



exponent



floating point number literal*decimal literal*

hexadecimal literal*octal literal**integer literal**character literal**literal**string literal*

B.3. Operatorenvorrang und Assoziativität

Prio.	Operator	Assoz.
<i>hoch</i>	() [] -> .	⇒
	++ - ! ~ (type) sizeof + ¹ - ¹ * ² & ²	⇐
	* / %	⇒
	+ -	⇒
	<< >>	⇒
	< <= > >=	⇒
	= !=	⇒
	&	⇒
	^	⇒
		⇒
	&&	⇒
		⇒
	?:	⇐
	= += -= *= /= %= &= ^= = <= >=	⇐
<i>niedrig</i>	,	⇒

¹Vorzeichen ²Adressoperatoren

B.4. Aus- und Eingabeformatierung

Deklaration:

```
1 int printf(char* format,...);
2 int scanf(char* format,...);
```

Im folgenden wird die Interpretation des „char* format“-Arguments beschrieben. Der Standard sieht noch weitere Möglichkeiten als hier beschrieben vor, die wiederum durch die jeweilige Implementation noch ergänzt werden können.

Im Zweifel ist die Man-Page („man 3 printf“) zu konsultieren.

B.4.1. printf

Der Formatstring wird ausgegeben, wobei Literale wie „\n“ beachtet werden. Ein Prozentzeichen („%“) kündigt die Ausgabe eines Wertes an; die Ausgabe eines Prozentzeichens erfolgt durch Wiederholung („%%“).

Nach dem Prozentzeichen können folgende Elemente folgen:

Flags (optional) beeinflusst die Darstellung des Wertes. Folgende Zeichen (auch mehrere) können verwendet werden:

„-“ Die Darstellung erfolgt linksbündig

„+“ (nur bei numerischen Argumenten) Es wird stets ein Vorzeichen ausgegeben, also nicht nur bei negativen Werten

„#“ schaltet alternative Darstellung ein

„0“ führende Null(en)

Feldbreite (optional) Dezimalwert, steuert die Darstellung führender Nullen oder Leerzeichen.

Präzision (optional) Punkt, gefolgt von einem Dezimalwert („.n“) steuert die Anzahl der Nachkommastellen oder der Zeichen im Ausgabestring

Typgrößenmodifikation (optional) steuert Typencasting vor Ausgabe:



„hh“ interpretiert Wert als char (nicht in C90)

„h“ interpretiert Wert als int

„l“ interpretiert Wert als long int



„ll“ interpretiert Wert als long long int (nicht in C90)

„L“ interpretiert Wert als long double

Typ (verpflichtend) welche Art von Wert soll ausgegeben werden:

„i“, „d“ Vorzeichenbehaftete Dezimalzahl

„u“ Ausgabe eines Ganzzahlwertes als vorzeichenlose Dezimalzahl

„o“ Ausgabe eines Ganzzahlwertes als vorzeichenlose Oktalzahl

„x“, „X“ Ausgabe eines Ganzzahlwertes als vorzeichenlose Hexadezimalzahl, entweder mit Klein- („x“) oder Großbuchstaben („X“) für die Werte größer 9

„c“ Ausgabe eines char-Ganzzahlwertes als Zeichen mit dem entsprechenden Code

„f“, „F“ Ausgabe einer Gleitkommazahl (double) in Standarddarstellung

„e“, „E“ Ausgabe einer Gleitkommazahl (double) in Exponentialdarstellung

„s“ Ausgabe eines C-Strings (bis zu einem „\0“), auf den das Zeigerargument zeigt

„p“ Ausgabe des *Wertes* eines Zeigers auf void

B.4.2. scanf

Als Eingabe wird jedes Zeichen des Formatstrings erwartet, es sei denn, ein Prozentzeichen („%“) ist im String. In diesem Fall wird die Eingabe interpretiert und den *dereferenzierten* Argumenten (Argumente sind ausschließlich Zeiger!) zugewiesen.

Nach dem Prozentzeichen können folgende Elemente folgen:

„*“ (optional) Input wird gelesen, aber nicht zugewiesen

Feldbreite (optional) Dezimalwert, steuert maximale Länge der Eingabe

Typgrößenmodifikation (optional) steuert Typencasting vor Ausgabe:

„hh“ interpretiert Wert als Zeiger auf char (für Ganzzahlen, nicht in C90)

„h“ interpretiert Wert als Zeiger auf int (für Ganzzahlen)

„l“ interpretiert Wert als Zeiger auf long int (für Ganzzahlen)

„ll“ interpretiert Wert als Zeiger auf long long int (für Ganzzahlen, nicht in C90)

„L“ interpretiert Wert als Zeiger auf long double (für Gleitkommazahlen)

„z“ interpretiert Wert als Zeiger auf size_t

Typ (verpflichtend) welche Art von Wert soll eingegeben werden:

„d“ Ggf. vorzeichenbehaftete dezimale Ganzzahl

„i“ Ggf. vorzeichenbehafteter Ganzzahlwert, Basis wird nach den Präfixregeln für Literale ermittelt

„o“ Vorzeichenlose oktale Ganzzahl

„x“, „X“ Vorzeichenlose hexadezimale Ganzzahl

„c“ Ganzzahlwerte als Zeichen mit dem entsprechenden Code

„f“ dezimale Gleitkommazahl in Festkommadarstellung

„e“ dezimale Gleitkommazahl in Exponentialdarstellung

„g“ Gleitkommazahl, wahlweise in Festkomma- oder Exponentialdarstellung

Gleitkommazahlen sind mindestens `double`, es ist eine Eingabe für `float` vorgesehen.



Anhang C.

Python

C.1. Schlüsselwörter

In Python 2.7 gibt es folgende Schlüsselwörter

and	elif	if	print
as	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	with
def	from	or	yield
del	global	pass	

In Python 3.0 entfällt das Schlüsselwort „print“. Dafür kommen noch folgende Schlüsselwörter hinzu:

False None True

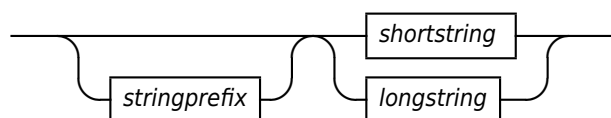
C.2. Syntax

Anmerkung

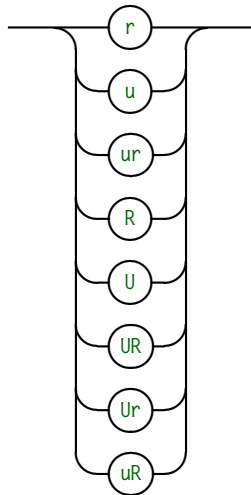
Um Komplexität zu vermeiden, werden im Folgenden die Token „INDENT“, „DEDENT“, „NEWLINE“ und „ENDMARKER“ verwendet.

Die Regeln zur Bildung von Bezeichnern (Identifiern) ist identisch zu C.

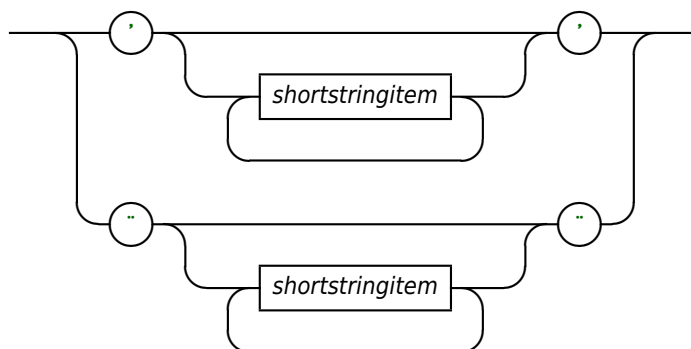
stringliteral



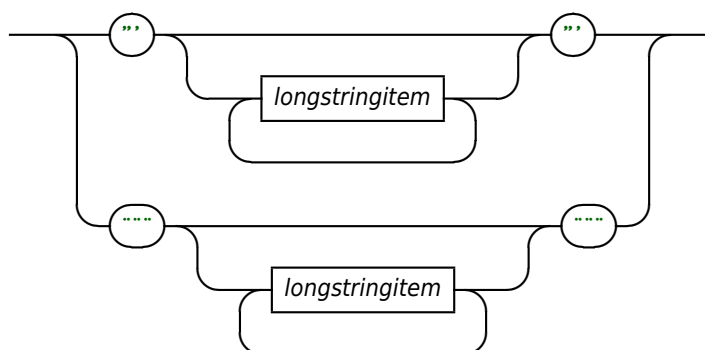
stringprefix



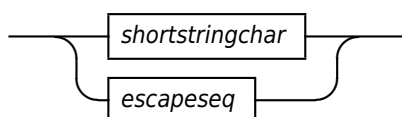
shortstring



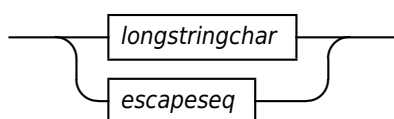
longstring



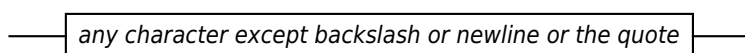
shortstringitem



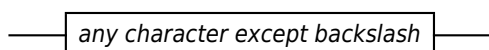
longstringitem



shortstringchar



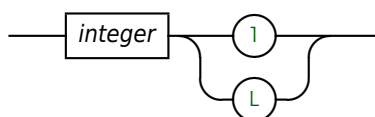
longstringchar



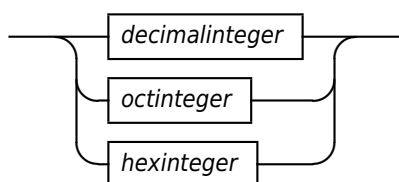
escapeseq



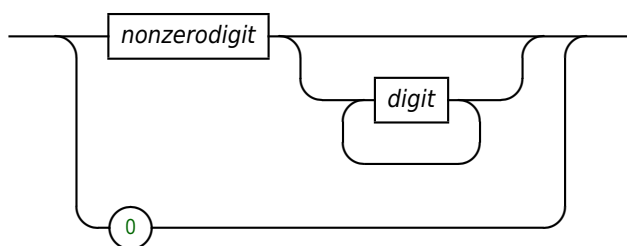
longinteger



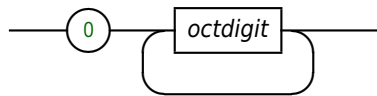
integer



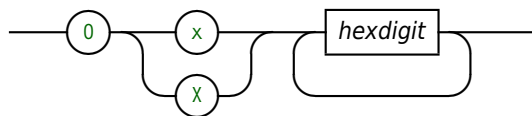
decimalinteger



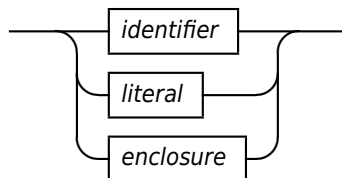
octinteger



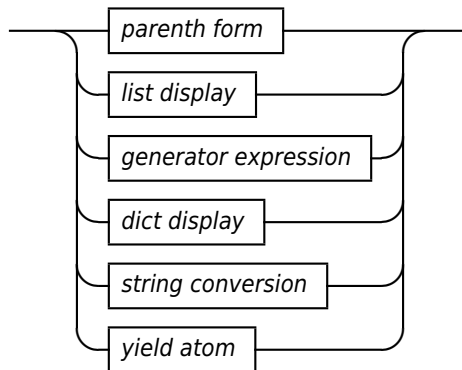
hexinteger



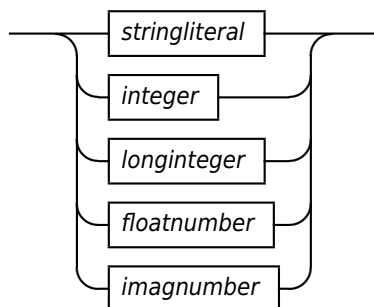
atom



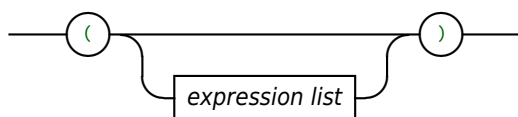
enclosure



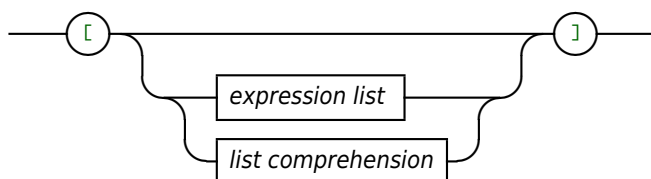
literal



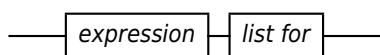
parenth form



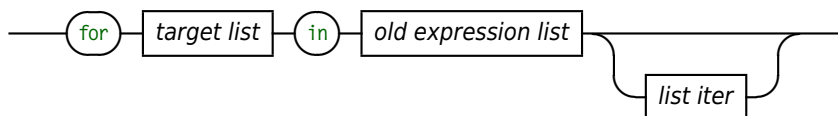
list display



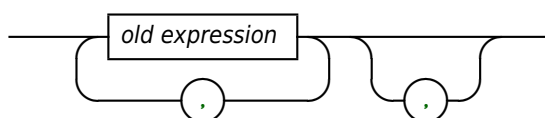
list comprehension



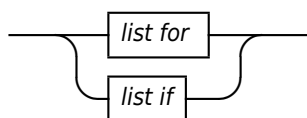
list for



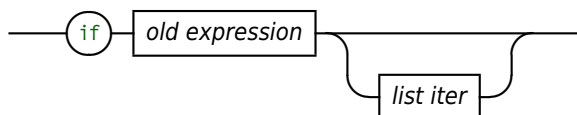
old expression list



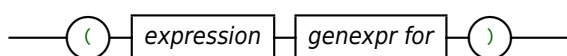
list iter



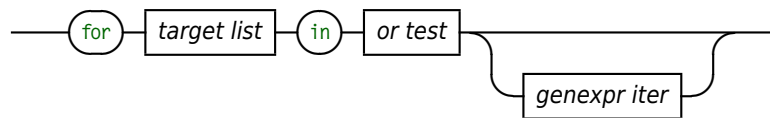
list if



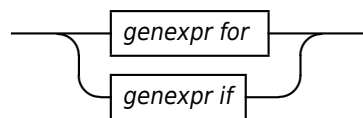
generator expression



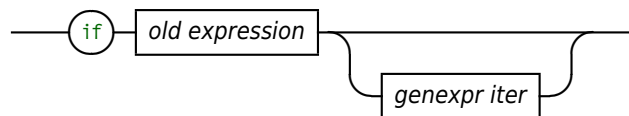
genexpr for



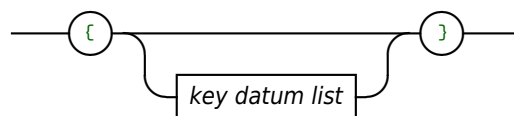
genexpr iter



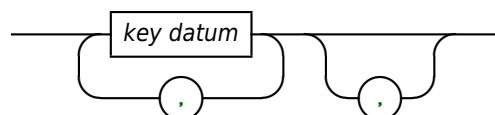
genexpr if



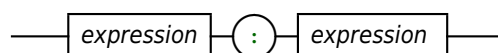
dict display



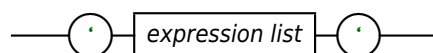
key datum list



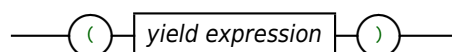
key datum



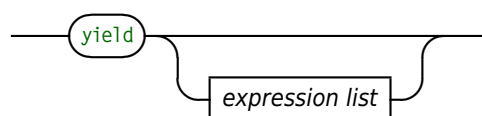
string conversion



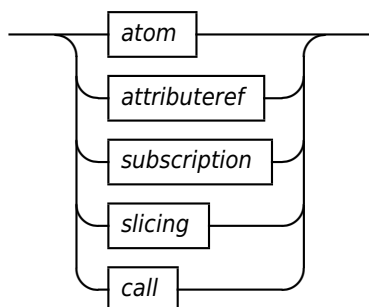
yield atom



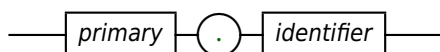
yield expression



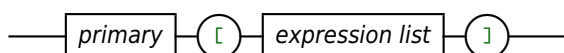
primary



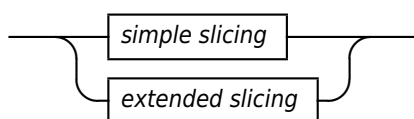
attributeref



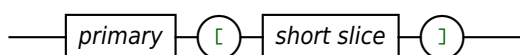
subscription



slicing



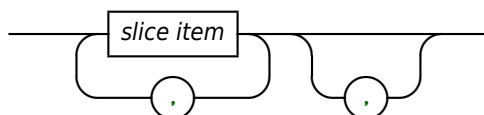
simple slicing



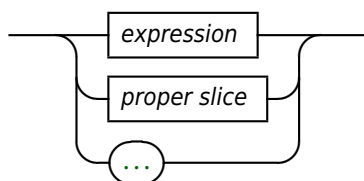
extended slicing



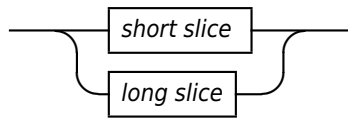
slice list



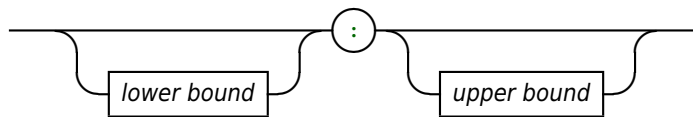
slice item



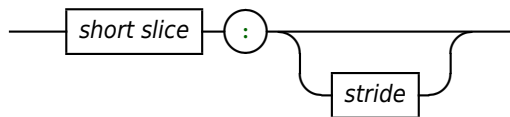
proper slice



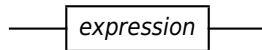
short slice



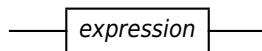
long slice



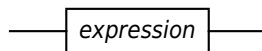
lower bound



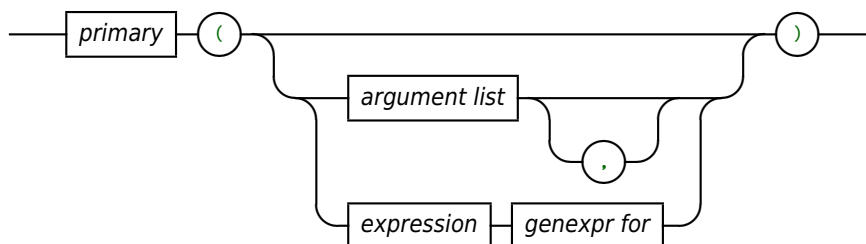
upper bound

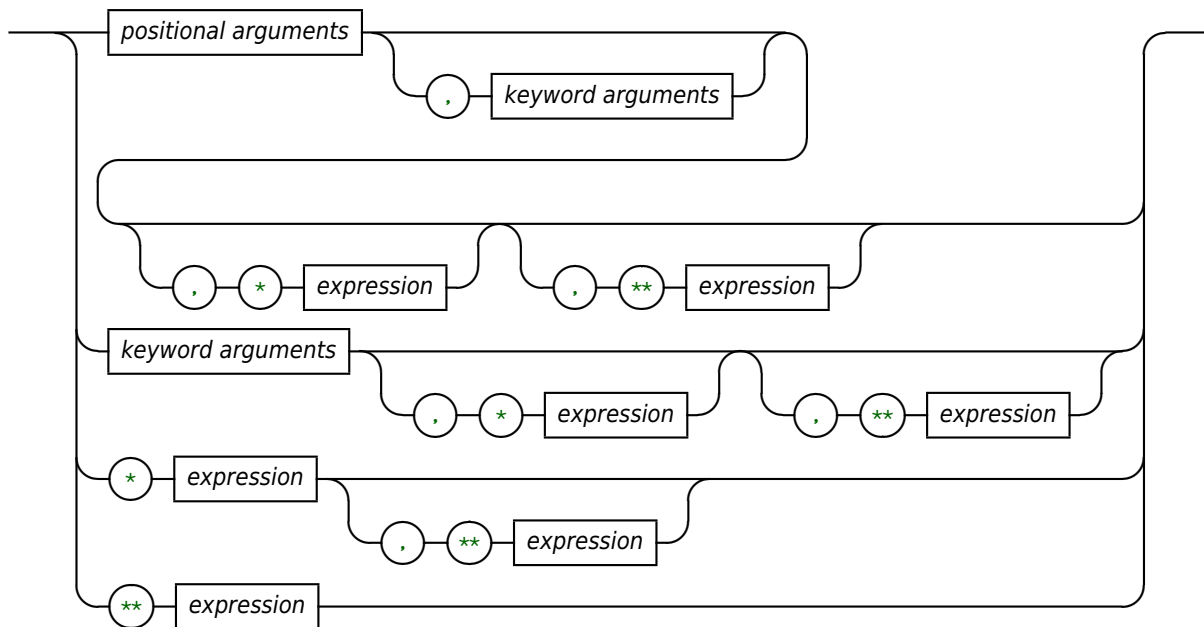
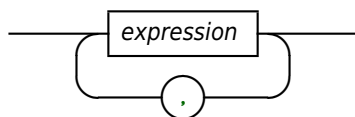
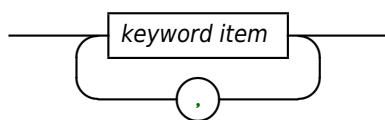
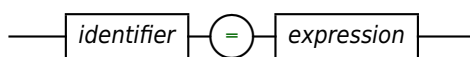
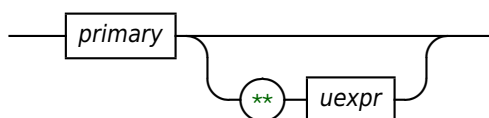


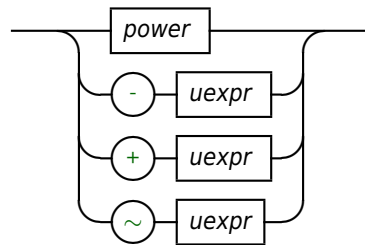
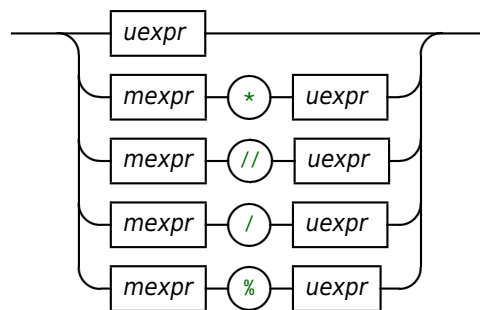
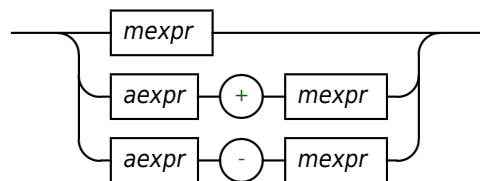
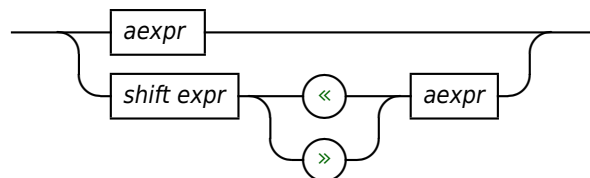
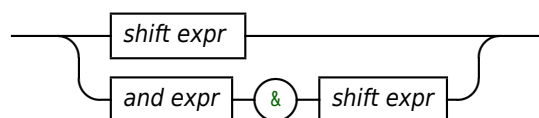
stride



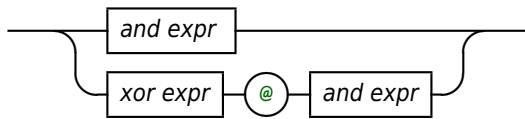
call



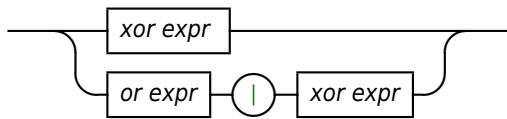
argument list*positional arguments**keyword arguments**keyword item**power*

uexpr*mexpr**aexpr**shift expr**and expr*

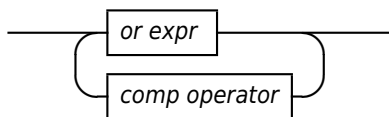
xor expr



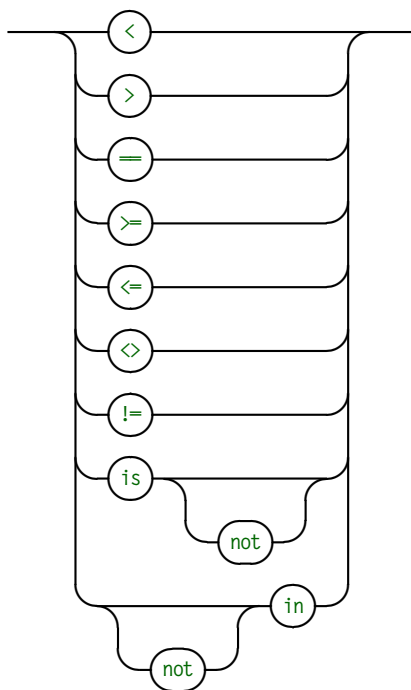
or expr



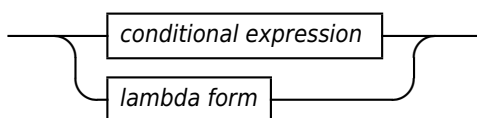
comparison



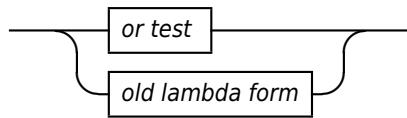
comp operator



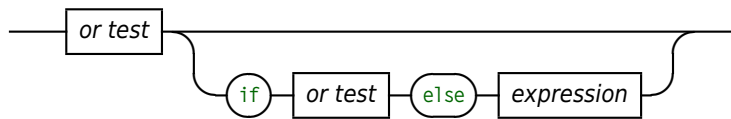
expression



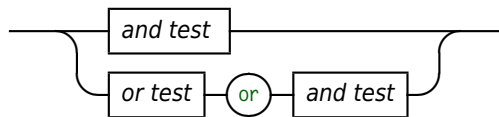
old expression



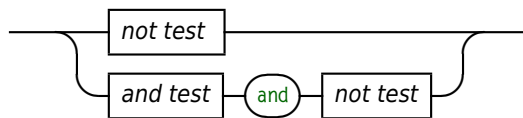
conditional expression



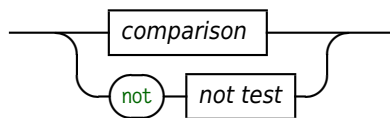
or test



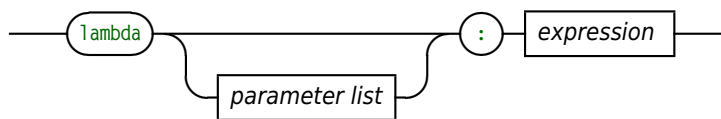
and test



not test



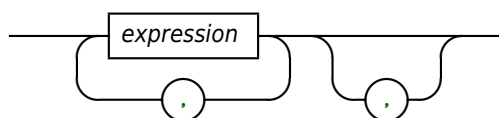
lambda form

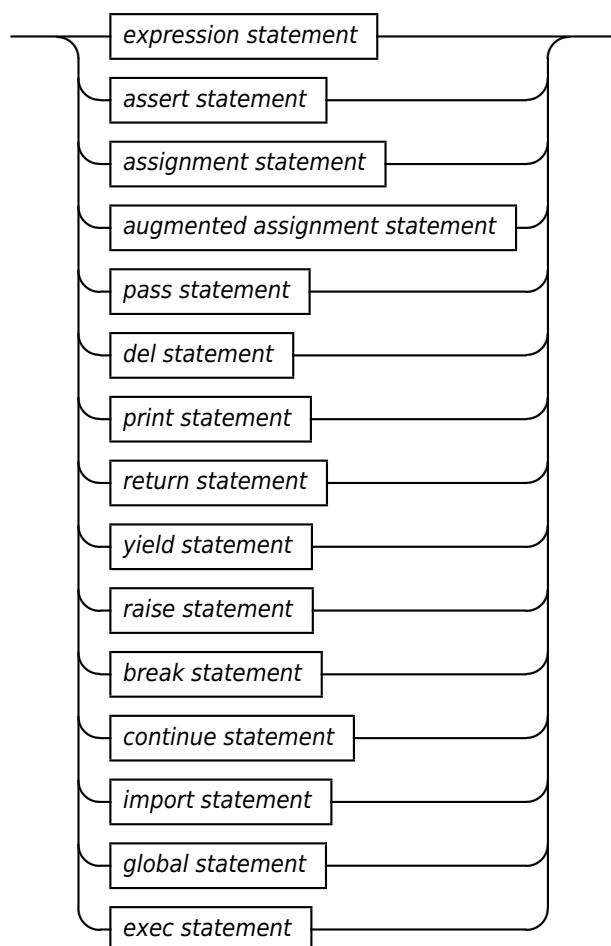
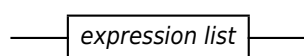
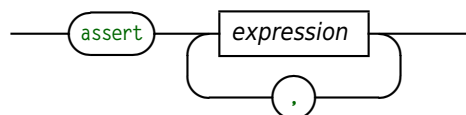
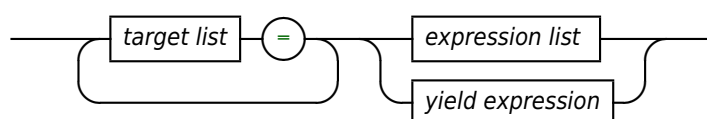


old lambda form

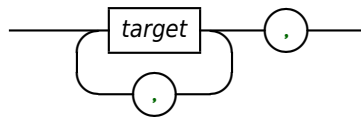


expression list

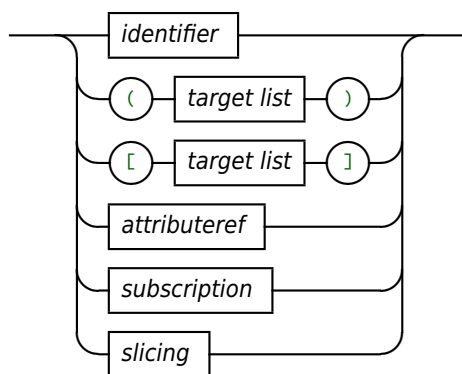


simple statement*expression statement**assert statement**assignment statement*

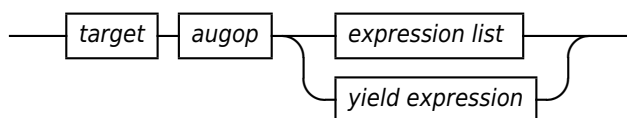
target list



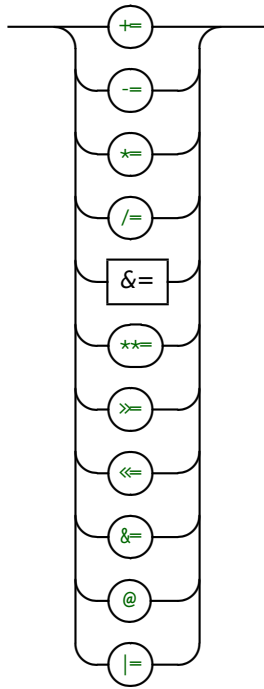
target



augmented assignment statement



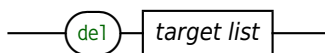
augop



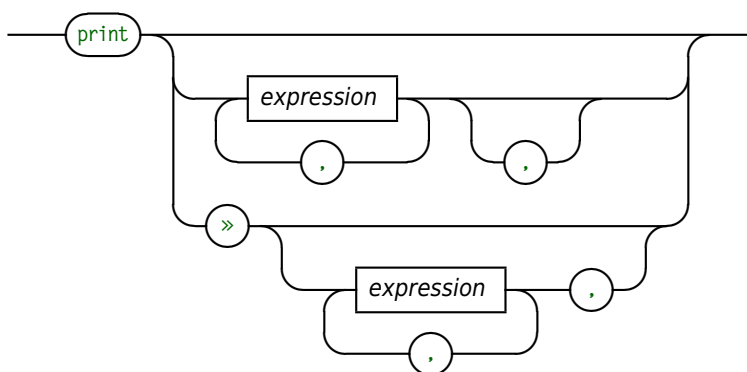
pass statement



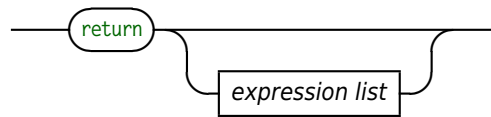
del statement



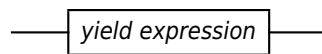
print statement



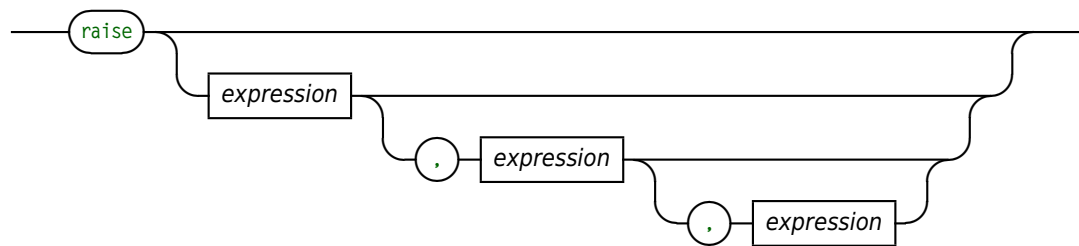
return statement



yield statement



raise statement

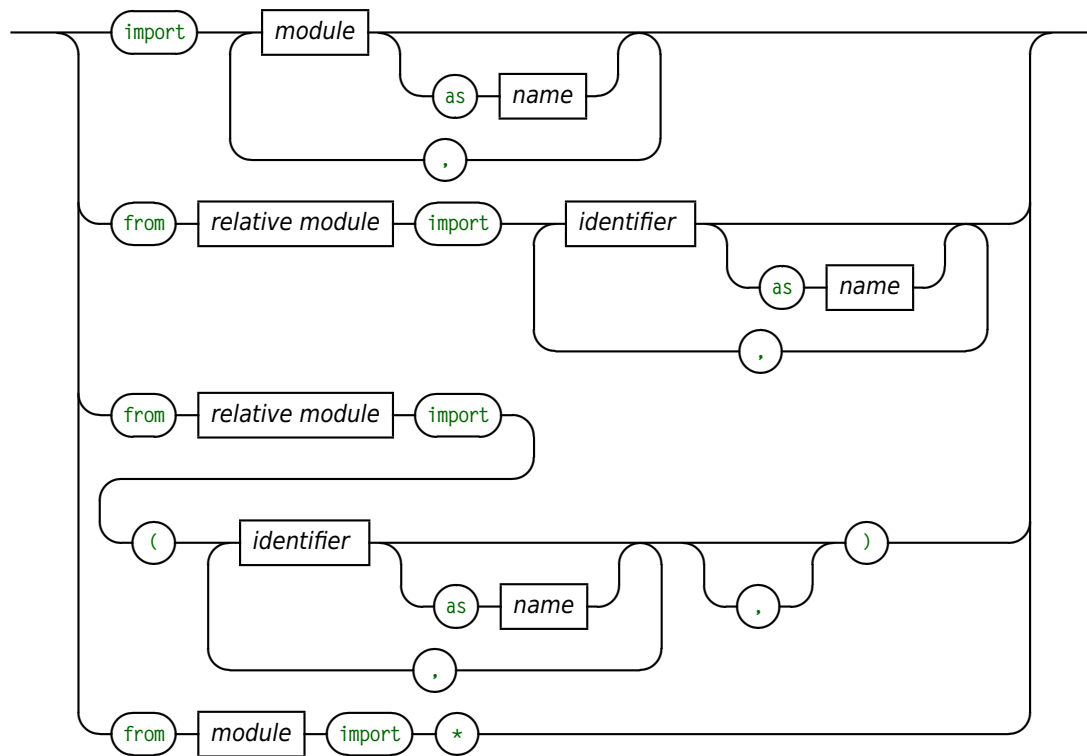
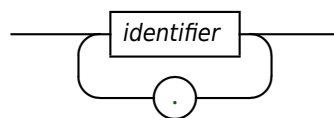
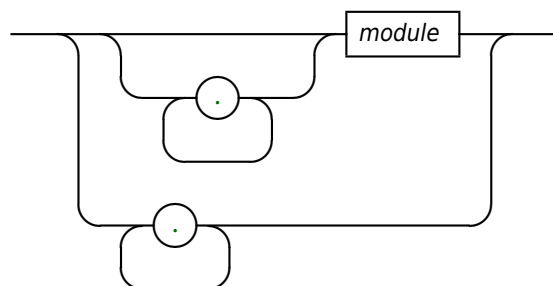
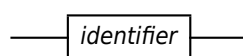


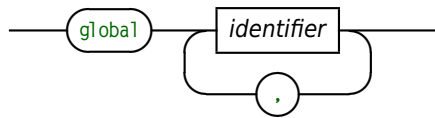
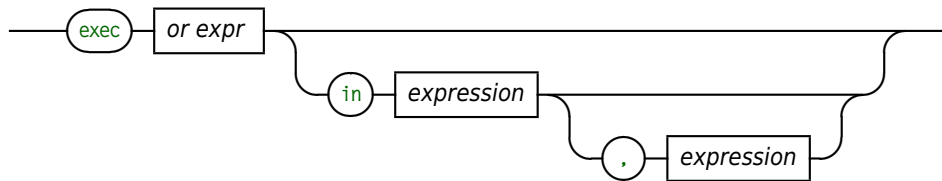
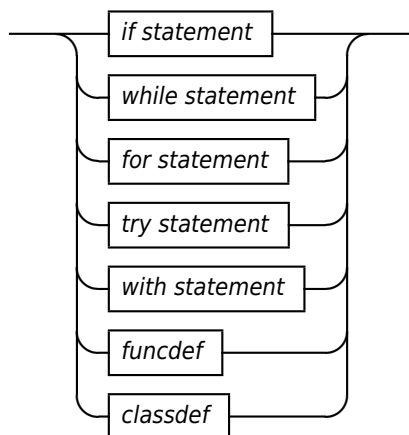
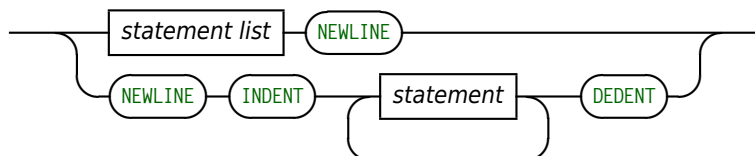
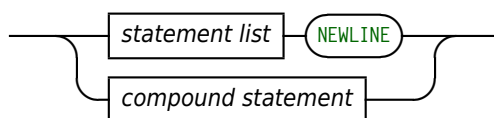
break statement

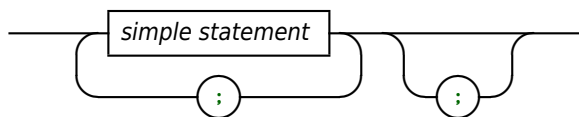
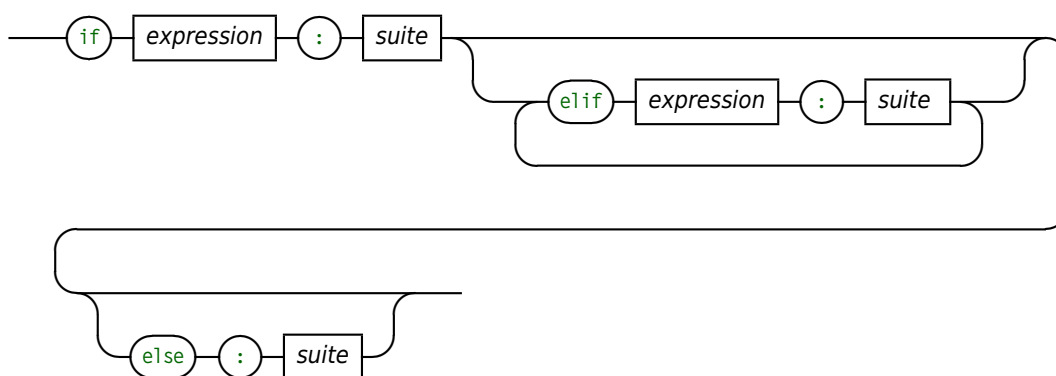
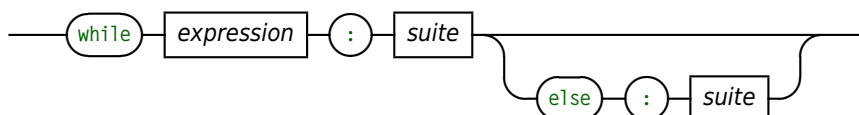
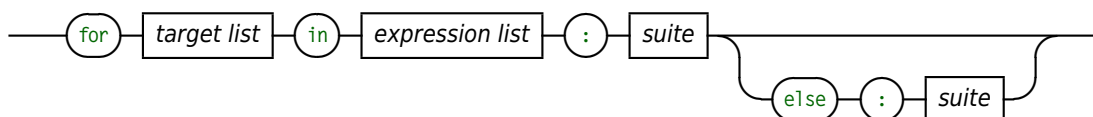
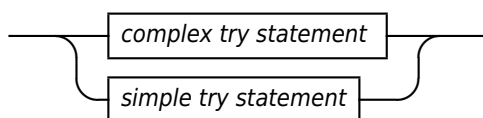


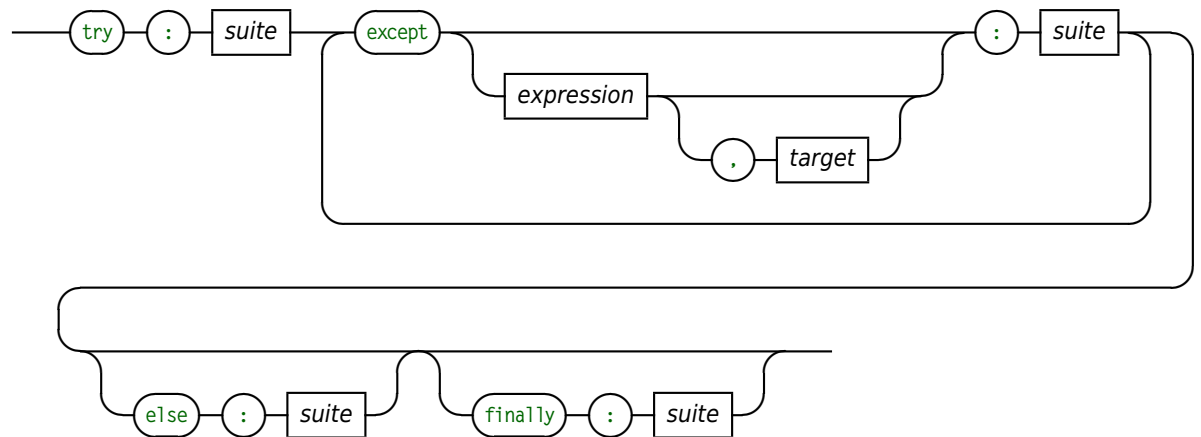
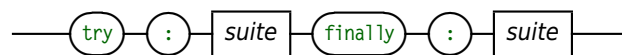
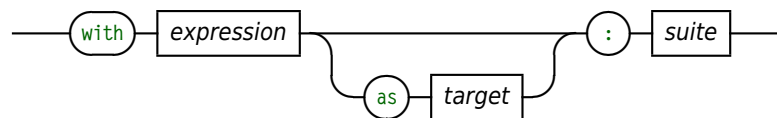
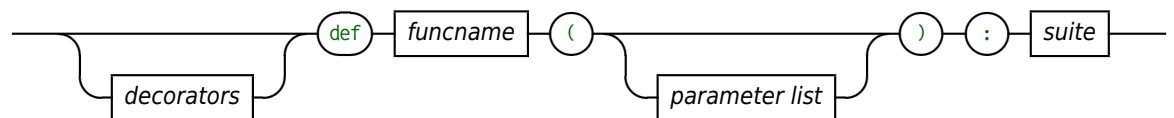
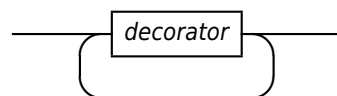
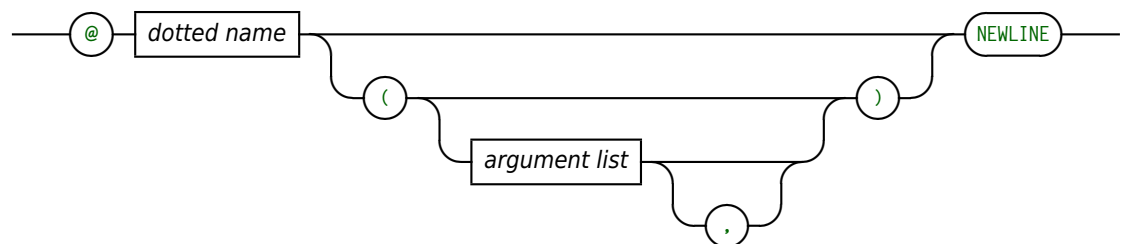
continue statement



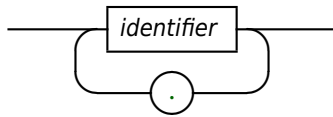
import statement*module**relative module**name*

global statement*exec statement**compound statement**suite**statement*

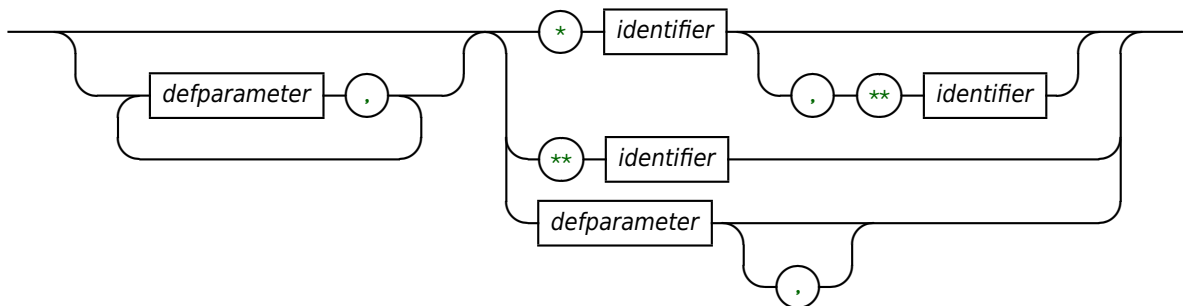
statement list*if statement**while statement**for statement**try statement*

complex try statement*simple try statement**with statement**funcdef**decorators**decorator*

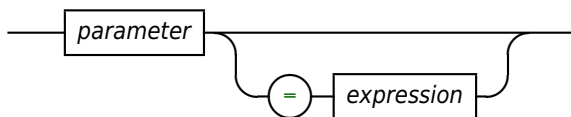
dotted name



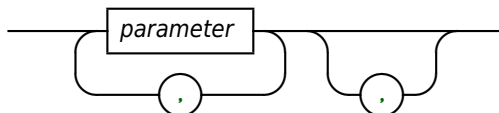
parameter list



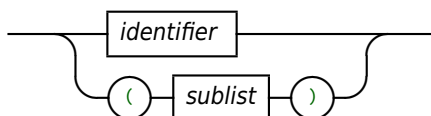
defparameter



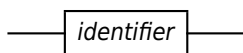
sublist



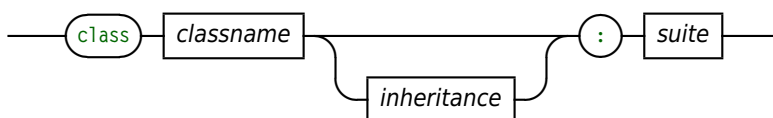
parameter



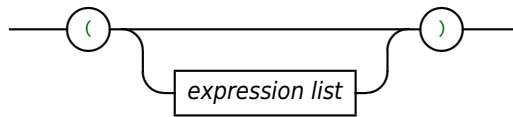
funcname



classdef



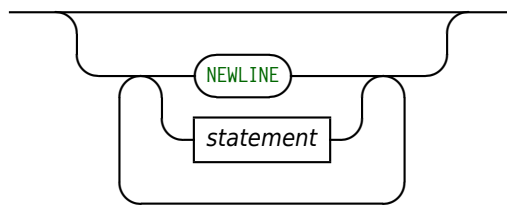
inheritance



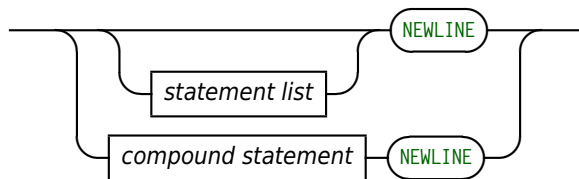
classname



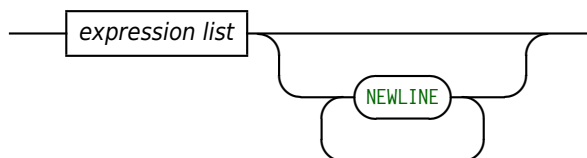
file input



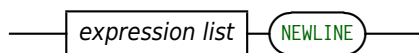
interactive input



eval input



input input



Anhang D.

Code

Der Übersicht halber sind in diesem Anhang verschiedene Codebeispiele aus dem Skript in geschlossener Form zu finden.

D.1. Textsuche

Der vollständigen Programmcode für einige Codebeispiele aus Kapitel 13.

D.1.1. Suchen mit dem Boyer-Moore-Algorithmus

```
1 /* search-bm.c -- An implementation of the Boyer/Moore algorithm */
2 #include<stdlib.h>
3 #include<assert.h>
4 #include<stdio.h>
5 #include <ctype.h>
6
7 int search(const char*, const char*, size_t);
8 int length(const char*);
9 int init_table(const char*);
10 void presentresult(int, const char*, const char*);
11 size_t filesize(FILE*);
12
13
14 enum {MAXCHAR=248}; // covers all ISO 8859-1 umlauts
15 int disttable[MAXCHAR];
16
17
18
19 int main(int argc, char* argv[]){
20     FILE* file;
21     char* text;
22     int found;
23     size_t size;
24
25     if (argc!=3) {
```

```

26     fprintf(stderr,"wrong number of parameters.\n");
27     return 1;
28 }
29 if (init_table(argv[1])) {
30     fprintf(stderr,"initialization failed.\n");
31     return 1;
32 }
33 if ((file=fopen(argv[2],"r"))==NULL) {
34     fprintf(stderr,"can't open file \"%s\".\n",argv[2]);
35     return 1;
36 }
37 size = filesize(file);
38 if ((text= malloc(size+1))==NULL) {
39     fprintf(stderr,"out of memory.\n");
40     return 1;
41 }
42 if (fread(text,size,1,file)!=1){
43     fprintf(stderr,"can't read file \"%s\".\n",argv[2]);
44     return 1;
45 }
46 text[size]='\0';
47 found=search(argv[1],text,size);          /* yet to implement */
48 if (found != -1) presentresult(found,text,argv[1]); /* dito */
49 free(text);
50 if (found != -1) return 0;
51 else return -1;
52 }
53
54
55
56 int search(const char* p, const char* str, size_t tlen)
57 {
58     size_t plen=length(p);
59     if (plen>tlen) return -1;
60     size_t pos=plen-1;
61     while(pos<tlen){
62         size_t j=0;
63         while((j<plen) && (str[pos-j]==p[plen-j-1])){
64             if (j==plen-1)
65                 return pos-plen+1;
66             ++j;
67         }
68         pos=pos+disttable[(int)str[pos]];
69     }
70     return -1;
71 }
72
73

```

```

74 int length(const char* str)
75 {
76     int len=0;
77     if (str==NULL) return 0;
78     while(str[len]!='\0') ++len;
79     return len;
80 }
81
82
83 size_t filesize(FILE* file) {
84     size_t ret;
85     fseek(file,0L,SEEK_END);
86     ret=ftell(file);
87     rewind(file);
88     return ret;
89 }
90
91
92 void presentresult(int pos, const char* str, const char* pattern) {
93     int start, end, patlen, prelen;
94     start=pos>20? pos-20 : 0;
95     prelen=pos>20? 20 : pos; /* length of prefix */
96     patlen=length(pattern);
97     end=pos+patlen;
98     printf("%.s[%s]%.20s\n",prelen,&str[start],pattern,&str[end]);
99 }
100
101
102
103 int init_table(const char* pattern) {
104     int len=length(pattern);
105     for(int i=0; i<MAXCHAR; ++i)
106         disttable[i]=len;
107     for(int i=0; i<len-1; ++i) {
108         if (((unsigned char)pattern[i])>=MAXCHAR)|| (pattern[i]<0))
109             return -1;
110         if (disttable[(int)pattern[i]]>len-i-1)
111             disttable[(int)pattern[i]]=len-i-1;
112     }
113     return 0;
114 }
115

```

D.1.2. Suchen mit Wildcards

```

1 /* search-wc.c -- search with wildcards
2 *
3 * Usage: search-wc <pattern> <file>

```

```

4  *
5  * You may use the following wildcards within the search pattern:
6  * '?' matches exactly one character
7  * '!' matches one or none character
8  * '*' matches an arbitrary number (i.e. also 0) of characters
9  */
10 #include<stdlib.h>
11 #include<stdio.h>
12 #include<ctype.h>
13 #include<stdbool.h>
14
15
16
17 typedef enum { MatchChar, MatchNotChar, MatchEOT, MatchNotEOT,
18               MatchNotCharNotEOT, NoEdge} match_t;
19
20 typedef struct{
21     match_t condition;
22     int     next;
23 } edge_t;
24
25 typedef enum { StateDefault, StateSuccess, StateFail} statetype_t;
26
27 typedef struct{
28     statetype_t type;
29     char        ch;
30     edge_t      edge[3];
31 } state_t;
32
33 enum { S_FAIL=0, S_SUCC=1 };
34
35 typedef struct{
36     state_t* state;
37     int      initial;
38 } automata_t;
39
40 typedef struct {
41     int start;
42     int end;
43 } searchresult_t;
44
45 searchresult_t search(automata_t, const char*, size_t);
46 int length(const char*);
47
48 automata_t init_automata(const char*);
49
50 char* sanitize(char* str);

```

```

51 void presentresult(searchresult_t, const char*);
52 size_t filesize(FILE*);
53
54
55 int main(int argc, char* argv[]) {
56     FILE* file;
57     char* text;
58     char* pattern;
59     automata_t automata;
60     searchresult_t found;
61     size_t size;
62
63     if (argc!=3) {
64         return -2; // wrong number of parameters
65     }
66     if ((file=fopen(argv[2],"r"))==NULL) {
67         return -3; // can't open file
68     }
69     size = filesize(file);
70     if ((text= malloc(size+1))==NULL) {
71         return -4; // out of memory
72     }
73     if (fread(text,size,1,file)!=1){
74         return -5; // can't read file
75     }
76     text[size]='\0';
77     pattern = sanitize(argv[1]);
78     automata= init_automata(pattern);
79
80     found= search(automata,text,size);
81     if (found.start != -1)
82         presentresult(found,text);
83     free(text);
84     free(automata.state);
85     return found.start;
86 }
87
88 searchresult_t search(automata_t a, const char* text, size_t tlen) {
89     searchresult_t res;
90     for(size_t pos=0; pos<tlen; ++pos){
91         int snr=a.initial;
92         size_t j=pos;
93         while(a.state[snr].type==StateDefault){
94             for(int i=0; i<3; i++){
95                 if (((a.state[snr].edge[i].condition==MatchChar) &&
96                     (text[j]==a.state[snr].ch)) ||
97                     ((a.state[snr].edge[i].condition==MatchNotChar) &&
98                     (text[j]!=a.state[snr].ch)) ||

```

```

99         ((a.state[snr].edge[i].condition==MatchEOT) &&
100          (text[j]=='\0')) ||
101         ((a.state[snr].edge[i].condition==MatchNotEOT) &&
102          (text[j]!='\0')) ||
103         ((a.state[snr].edge[i].condition==MatchNotCharNotEOT) &&
104          (text[j]!='\0') &&
105          (text[j]!=a.state[snr].ch)))
106     {
107         snr = a.state[snr].edge[i].next;
108         ++j; /* next character */
109         break;
110     }
111 } /* end for over edges */
112 }
113 if (a.state[snr].type==StateSuccess) {
114     res.start=pos;
115     res.end=j;
116     return res;
117 }
118 } /* end for over text */
119 res.start=-1;
120 res.end=-1;
121 return res;
122 }
123
124 int length(const char* str) {
125     int len=0;
126     if (str==NULL) return 0;
127     while(str[len]!='\0') ++len;
128     return len;
129 }
130
131 size_t filesize(FILE* file) {
132     size_t ret;
133     fseek(file,0L,SEEK_END);
134     ret=ftell(file);
135     rewind(file);
136     return ret;
137 }
138
139 void presentresult(searchresult_t res, const char* str) {
140     int start, prelen;
141     start=res.start>20? res.start-20 : 0; /* start where? */
142     prelen=res.start>20? 20 : res.start; /* length of prefix */
143     printf("%.*s[%.*s]%.20s\n",prelen, &str[start],
144           res.end-res.start, &str[res.start],&str[res.end]);
145 }
146

```



```

147 char* sanitize(char* str) {
148     char* nstr=str;
149     bool changed;
150     int len=length(nstr);
151     if ((nstr==NULL) || (len==0)) return NULL;
152     do {
153         changed=false;
154         while((nstr[len-1]=='*') || (nstr[len-1]=='!')){
155             nstr[--len]='\0'; /* shorten tail by 1 */
156         }
157         while((nstr[0]=='*') || (nstr[0]=='!')) {
158             nstr=&nstr[1]; /* move start 1 to the right */
159             --len;
160         }
161         for(int i=0; i<len; ) {
162             if (((nstr[i]=='*') || (nstr[i]=='!')) && (nstr[i+1]=='?')) {
163                 nstr[i+1]=nstr[i]; /* !? => ?! resp. *? => ?* */
164                 nstr[i]='?';
165                 changed=true;
166             } else ++i;
167         }
168
169         for(int i=1; i<len-1; ){
170             if (((nstr[i]=='*') && (nstr[i+1]=='!')) ||
171                 ((nstr[i]=='!') && (nstr[i+1]=='*')) ||
172                 ((nstr[i]=='*') && (nstr[i+1]=='*'))){
173                 --len;
174                 nstr[i]='*';
175                 int j=i+1;
176                 do{ /* move remains one to the left */
177                     nstr[j]=nstr[j+1];
178                     ++j;
179                 } while(nstr[j]!='\0');
180                 changed=true;
181             }
182             else ++i;
183         }
184     } while(changed);
185     return nstr;
186 }
187
188 automata_t init_automata(const char* p) {
189     int snr, len;
190     len=snr=length(p);
191     for(int i=len-1; i>=0; --i)
192         if(p[i]=='*') --snr; // decrease for each "*"
193     automata_t a;
194     a.state = calloc(snr+2, sizeof(state_t));

```

```

195  /* Fail */
196  a.state[S_FAIL].type=StateFail;
197  a.state[S_FAIL].edge[0].condition=NoEdge;
198  a.state[S_FAIL].edge[1].condition=NoEdge;
199  a.state[S_FAIL].edge[2].condition=NoEdge;
200  /* Success */
201  a.state[S_SUCC].type=StateSuccess;
202  a.state[S_SUCC].edge[0].condition=NoEdge;
203  a.state[S_SUCC].edge[1].condition=NoEdge;
204  a.state[S_SUCC].edge[2].condition=NoEdge;
205
206  snr=1;
207  for(int pos=len-1; pos>=0; --pos){
208      switch (p[pos]) {
209          default:
210              ++snr;
211              a.state[snr].ch=p[pos];
212              a.state[snr].type=StateDefault;
213              a.state[snr].edge[0].condition=MatchChar;
214              a.state[snr].edge[0].next=snr-1;
215              a.state[snr].edge[1].condition=MatchNotChar;
216              a.state[snr].edge[1].next=S_FAIL;
217              a.state[snr].edge[2].condition=NoEdge;
218              break;
219          case '?':
220              ++snr;
221              a.state[snr].type=StateDefault;
222              a.state[snr].edge[0].condition=MatchNotEOT;
223              a.state[snr].edge[0].next=snr-1;
224              a.state[snr].edge[1].condition=MatchEOT;
225              a.state[snr].edge[1].next=S_FAIL;
226              a.state[snr].edge[2].condition=NoEdge;
227              break;
228          case '!':
229              ++snr;
230              a.state[snr].ch=a.state[snr-1].ch; // get character from next state
231              a.state[snr].type=StateDefault;
232              a.state[snr].edge[0].condition=MatchChar;
233              a.state[snr].edge[0].next=a.state[snr-1].edge[0].next;
234              a.state[snr].edge[1].condition=MatchEOT;
235              a.state[snr].edge[1].next=S_FAIL;
236              a.state[snr].edge[2].condition=MatchNotCharNotEOT;
237              a.state[snr].edge[2].next=snr-1;
238              break;
239          case '*':
240              // no ++snr !
241              a.state[snr].edge[1].condition=MatchEOT;
242              a.state[snr].edge[2].condition=MatchNotCharNotEOT;

```

```

243     a.state[snr].edge[2].next=snr;
244     break;
245 }
246 }
247 a.initial=snr;
248 return a;
249 }
250

```

D.2. Tic Tac Toe

Der komplette Code des Tic-Tac-Toe-Programms aus Kapitel 14.

D.2.1. Makefile

```

1  PROG=ttt
2  CC=cc
3  CFLAGS= -std=c99 -Wpedantic -Wall -Wextra
4  LDFLAGS=
5
6  HEADER= $(wildcard ttt*.h)
7  SOURCEFILES= $(wildcard ttt*.c)
8  OBJECTFILES= $(subst .c,.o,$(SOURCEFILES))
9
10 $(PROG): $(OBJECTFILES)
11          $(CC) $(LDFLAGS) $(OBJECTFILES) -o $(PROG)
12
13 %.o: %.c
14          $(CC) -c $(CFLAGS) $<
15
16 .PHONY: clean
17 clean:
18          rm -f $(PROG) *.o

```

D.2.2. ttt.h

```

1  /* ttt.h -- common header file for "tic tac toe" */
2  #include<stdbool.h>
3
4  /* general constants */
5  typedef enum {TTT_ABORT=-1, NUMBER_OF_FIELDS=9} ttt_constant_t;
6
7  /* game outcomes */
8  typedef enum {TTT_PLAYER_WINS, TTT_COMPUTER_WINS, TTT_DRAW, TTT_UNDECIDED}
9          ttt_result_t;
10
11 typedef int ttt_board_t[NUMBER_OF_FIELDS];

```

```

11
12
13 /* general functions */
14 void ttt_init_board(ttt_board_t);
15 char ttt_opponent(char);
16
17 /* strategy functions */
18 int ttt_calculate_move(const ttt_board_t, char);
19 ttt_result_t ttt_won_or_draw(const ttt_board_t, char);
20
21 /* input/output functions */
22 void ttt_update_display(const ttt_board_t);
23 int ttt_input_move(const ttt_board_t);
24 void ttt_output_result(const ttt_result_t);
25 int ttt_x_or_o();
26 bool ttt_another_game();

```

D.2.3. ttt_main.c

```

1 /* ttt_main.c -- main module for "tic tac toe" */
2
3 #include "ttt.h"
4
5
6 int main() {
7     ttt_board_t board;
8     do { /* main loop */
9         int player = ttt_x_or_o(); // select symbol (x or o)
10        bool computer_turn = (player == 'o');
11        ttt_init_board(board); // erase playground
12        int move, assessment;
13        do { /* main loop for a game instance */
14            ttt_update_display(board); // display board
15            move = computer_turn ? // computer's move?
16                ttt_calculate_move(board, ttt_opponent(player)) : // calculate
17                ttt_input_move(board); // player's move
18            if (move != TTT_ABORT) // continue?
19                board[move] = computer_turn ? // computer's move?
20                ttt_opponent(player) : player; // apply move
21            computer_turn = !computer_turn; // change the turn
22            assessment = ttt_won_or_draw(board, player); // game over?
23            if (assessment != TTT_UNDECIDED) {
24                ttt_update_display(board); // display final board
25                ttt_output_result(assessment); // print result
26            }
27        } while ((move != TTT_ABORT) && (assessment == TTT_UNDECIDED));
28    } while (ttt_another_game() == true);
29    return 0;

```

```

30 }
31
32
33 void ttt_init_board(ttt_board_t f)
34 {
35     /* empties every field */
36     int i;
37     for(i=0;i<NUMBER_OF_FIELDS;i++)
38         f[i]=' ';
39 }
40
41 char ttt_opponent(char symbol)
42 {
43     switch(symbol)
44     {
45         case 'x': return 'o';
46         case 'o': return 'x';
47         default: return ' ';
48     }
49 }

```

D.2.4. ttt_io.c

```

1 #include<stdio.h>
2 #include"ttt.h"
3
4 static ttt_board_t numberfield={'1','2','3','4','5','6','7','8','9'};
5
6 /* Use VT100 ESC code to clean terminal */
7 static void ttt_clean_terminal(void)
8 {
9     return;
10    /* The sequence ESC[2J clears the screen */
11    printf( "%c[2J", 27 );
12 }
13
14 void ttt_display_board(const ttt_board_t board)
15 {
16     for(int i=0;i<3;i++){
17         if(i) // not for first row
18             printf("\n  -+-");
19         printf("\n  ");
20         for(int j=0;j<3;j++){
21             if(j) printf("|"); // not for first column
22             printf("%c",board[3*i+j]);
23         }
24     }
25     printf("\n");

```

```

26 }
27
28 void ttt_update_display(const ttt_board_t board)
29 {
30     ttt_clean_terminal();
31     ttt_display_board(board);
32 }
33
34 int ttt_x_or_o()
35 {
36     int input;
37     printf("Do you want to play 'x' (first) or 'o' (second)? ->");
38     do {
39         input=getchar(); getchar();
40     } while ((input!='x') && (input!='o'));
41     return input;
42 }
43
44
45 int ttt_input_move(const ttt_board_t board)
46 {
47     int input;
48     printf("\nPlease enter the number of the field you want to occupy (0 for
         abort)\n");
49     ttt_display_board(numberfield);
50     printf("Your move ->");
51     do{
52         do {
53             input=getchar(); getchar();
54         } while ((input<'0') || (input>'9'));
55         if (input=='0') return TTT_ABORT;
56         input=input-'1'; // character => integer index
57     } while (board[input]!=' ');
58     return input;
59 }
60
61 void ttt_output_result(const ttt_result_t result)
62 {
63     printf("\nGAME OVER\n");
64     if (result==TTT_DRAW)
65     {
66         printf("Nobody won.\n");
67         return;
68     }
69     if (result==TTT_PLAYER_WINS) {
70         printf("You won!\n");
71     } else {
72         printf("You lost!\n"); }

```

```

73 }
74
75 bool ttt_another_game()
76 {
77     int input;
78     printf("Do you want to play another game [y/n] ->");
79     input=getchar(); getchar();
80     if ((input=='Y') || (input=='y'))
81         return true;
82     else
83         return false;
84 }
85

```

D.2.5. ttt_strategy.c

```

1  #include "ttt.h"
2  #include <stdio.h>
3
4  enum {NUMBER_TRIPLES=8,
5        I_KEY=2,          /* computer symbol's value      */
6        OPP_KEY=5,        /* player symbol's value       */
7        I_WIN=6,          /* computer won (=2+2+2)       */
8        OPP_WIN=15,       /* player won (=5+5+5)         */
9        I_CANWIN=4,       /* computer can win (=2+2+0)   */
10       OPP_CANWIN=10,     /* player can win (=5+5+0)     */
11       DRAW=7,           /* draw, one 'x', one 'o' (=5+2+0) */
12       DRAW_I=9,         /* draw, two computer symbols and
13                          one player symbol (=2+2+5) */
14       DRAW_OPP=12,      /* draw, two player symbols and
15                          one computer symbol (=5+5+2) */
16       NONE=-1           /* no move found */
17 };
18 typedef int ttt_triplev_t[NUMBER_TRIPLES];
19 typedef int ttt_triple_t[3];
20 static const ttt_triple_t triples[NUMBER_TRIPLES]=
21     {{0,1,2},{3,4,5},{6,7,8},{0,3,6},{1,4,7},{2,5,8},{0,4,8},{2,4,6}};
22
23 static int evaluate(const ttt_board_t, int, int);
24 static int winning_move(const ttt_board_t, const ttt_triplev_t, int);
25 static int forking_move(const ttt_board_t, const ttt_triplev_t);
26 static int forcing_move(const ttt_board_t, const ttt_triplev_t);
27 static int best_remaining_move(const ttt_board_t);
28
29
30 static int evaluate(const ttt_board_t field, int tnr, int my_symbol)
31 {

```

```

32  int val=0;
33  int opp_symbol=ttt_opponent(my_symbol);
34  for (int i=0; i<3; ++i){
35      if (field[triples[tnr][i]]==my_symbol)
36          val+=I_KEY;
37      else if(field[triples[tnr][i]]==opp_symbol)
38          val+=OPP_KEY;
39  }
40  return val;
41 }
42
43 static int winning_move(const ttt_board_t field, const ttt_tripleval_t eval
    , int who)
44 {
45     for (int i=0; i<NUMBER_TRIPLES; i++)
46     {
47         if (eval[i]==who) /* Find winning triple */
48         {
49             /* Find the empty place */
50             for(int j=0;j<3;j++)
51                 if (field[triples[i][j]]==' ') return triples[i][j];
52         }
53     }
54     return NONE;
55 }
56
57 static int forking_move(const ttt_board_t field, const ttt_tripleval_t eval
    )
58 {
59     int singles[NUMBER_OF_FIELDS]={0,0,0,0,0,0,0,0,0,0};
60     /* Count number of empty places in triples I own */
61     for (int i=0; i<NUMBER_TRIPLES; ++i)
62     {
63         if (eval[i]==I_KEY)
64         {
65             for(int j=0;j<3;j++)
66                 if (field[triples[i][j]]==' ')
67                     ++singles[triples[i][j]];
68         }
69     }
70     /* Is there a pivot? */
71     for(int i=0; i< NUMBER_OF_FIELDS; ++i)
72         if (singles[i]>=2) return i;
73     return NONE;
74 }
75
76 static int forcing_move(const ttt_board_t field, const ttt_tripleval_t eval
    )

```



```

77 {
78     int opp_singles[NUMBER_OF_FIELDS]={0,0,0,0,0,0,0,0,0};
79
80     /* Count number of empty places in triples opponent owns */
81     for (int i=0; i<NUMBER_TRIPLES; ++i)
82     {
83         if (eval[i]==OPP_KEY)
84         {
85             for(int j=0;j<3;j++)
86                 if (field[triples[i][j]]==' ')
87                     ++opp_singles[triples[i][j]];
88         }
89     }
90
91     /* Find a triple I own */
92     for (int i=0; i<NUMBER_TRIPLES; ++i)
93     {
94         if (eval[i]==I_KEY)
95         {
96             /* If there is no opponent's pivot element... */
97             if((opp_singles[triples[i][0]]<=1) &&
98                 (opp_singles[triples[i][1]]<=1) &&
99                 (opp_singles[triples[i][2]]<=1))
100                 for(int j=0;j<3;j++)
101                     /* ...select an empty field */
102                     if (field[triples[i][j]]==' ')
103                         return triples[i][j];
104                 for(int j=0;j<3;j++)
105                     /* Select opponent's pivot element */
106                     if(opp_singles[triples[i][j]]>1)
107                         return triples[i][j];
108         }
109     }
110     return NONE;
111 }
112
113 static int best_remaining_move(const ttt_board_t field)
114 {
115     const int best[]={4,0,2,6,8,1,3,5,7};
116
117     for(int i=0;i<NUMBER_OF_FIELDS;i++)
118         if(field[best[i]]==' ') return best[i];
119     return NONE; // Should never happen
120 }
121
122 int ttt_calculate_move(const ttt_board_t field, char symbol)
123 {
124     ttt_tripleval_t eval;

```

```
125     for (int i=0; i< NUMBER_TRIPLES; ++i)
126         eval[i]=evaluate(field,i,symbol);
127     int move=winning_move(field,eval,I_CANWIN);
128     if (move!=NONE) return move;
129     move=winning_move(field,eval,OPP_CANWIN);
130     if (move!=NONE) return move;
131     move=forking_move(field,eval);
132     if (move!=NONE) return move;
133     move=forcing_move(field,eval);
134     if (move!=NONE) return move;
135     return best_remaining_move(field);
136 }
137
138 ttt_result_t ttt_won_or_draw(const ttt_board_t board, char symbol)
139 {
140     bool undecided=false;
141     for(int i=0; i<NUMBER_TRIPLES; i++) {
142         int state=evaluate(board,i,'x');
143         switch (state) {
144             case I_WIN:
145                 if (symbol=='x') return TTT_PLAYER_WINS;
146                 else return TTT_COMPUTER_WINS;
147             case OPP_WIN:
148                 if (symbol=='o') return TTT_PLAYER_WINS;
149                 else return TTT_COMPUTER_WINS;
150             case DRAW_I: /* trough */
151             case DRAW_OPP: /* trough */
152                 break;
153             case DRAW: /* trough */
154             default:
155                 undecided=true;
156         }
157     }
158     if (undecided) return TTT_UNDECIDED;
159     else return TTT_DRAW;
160 }
```

D.3. Taschenrechner

Der Code für den „Taschenrechner“ aus Kapitel 15]

D.3.1. Makefile

```

1 CFLAGS= -std=c99 -Wall -Wextra -Werror
2 LDFLAGS=-lncurses -lm
3
4 MODULES= nc-main.o nc-gui.o nc-parseloop.o nc-token.o nc-calc.o
5
6 .PHONY: clean
7
8 ncalc: $(MODULES)
9         $(CC) $(LDFLAGS) $(MODULES) -o ncalc
10
11 %.o:    %.c
12         $(CC) $(CFLAGS) -c $<
13
14 clean:
15         rm -f ncalc *.o

```

D.3.2. nc-main.c

```

1 #include "nc-gui.h"
2 #include "nc-token.h"
3 #include "nc-calc.h"
4 #include "nc-parseloop.h"
5
6 int main(){
7     char display[ncMaxNumberDigits+2];
8     display[0]='0';
9     display[1]=0;
10    ncResetCalc();
11    ncGuiInit();
12    ncParseLoop(display);
13    ncGuiClean();
14    return 0;
15 }

```

D.3.3. nc-gui.h

```

1 // nc-gui.h
2 typedef enum { NCK_NONE=-3, NCK_QUIT=-2,
3     // NCK_0 bis NCK_9 sind so gewählt, dass sie die Werte 0 bis 9 haben
4     NCK_0=0, NCK_1, NCK_2, NCK_3, NCK_4, NCK_5, NCK_6, NCK_7, NCK_8, NCK_9,
5     NCK_SIGN,    // Vorzeichen
6     NCK_DPOINT, // Dezimalzeichen

```

```

7     NCK_SUB, NCK_ADD,
8     NCK_DIV, NCK_MUL,
9     NCK_LOG, NCK_EXP,
10    NCK_RESULT, NCK_DEL, NCK_RESET
11 } nc_key_t;
12
13 extern const unsigned int ncMaxNumberDigits;
14 extern const char* ncConvertSting;
15
16 void ncGuiInit(void);
17 void ncGuiClean(void);
18 void ncGuiUpdate(char* num);
19 nc_key_t ncInput(void);

```

D.3.4. nc-gui.c

```

1  /*
2   Hauptschnittstelle zur curses-Bibliothek
3   */
4  #include <locale.h>
5  #include <ncurses.h>
6  #include <stdlib.h> // für exit
7  #include <unistd.h> // für usleep
8  #include "nc-gui.h"
9
10 struct button_layout_t {
11     int      x;
12     int      y;
13     char*    *text;
14     nc_key_t key;
15     attr_t   default_attr;
16     attr_t   click_attr;
17 };
18
19 enum { BLACK_WHITE=1, DISPLAY_COLOR, NUMBER_BT_COLOR, OPERATION_BT_COLOR };
20 const int unsigned ncMaxNumberDigits=20;
21 const char* ncConvertSting="% .19g";
22
23 static WINDOW *win;
24 static struct button_layout_t layout[]={
25     { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
26         NUMBER_BT_COLOR) | A_STANDOUT, .x=5, .y=11, .text="0", .key=NCK_0},
27     { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
28         NUMBER_BT_COLOR) | A_STANDOUT, .x=1, .y=9, .text="1", .key=NCK_1},
29     { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
30         NUMBER_BT_COLOR) | A_STANDOUT, .x=5, .y=9, .text="2", .key=NCK_2},
31     { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
32         NUMBER_BT_COLOR) | A_STANDOUT, .x=9, .y=9, .text="3", .key=NCK_3},

```

```

29 { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
    NUMBER_BT_COLOR) | A_STANDOUT, .x=1, .y=7, .text="4", .key=NCK_4},
30 { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
    NUMBER_BT_COLOR) | A_STANDOUT, .x=5, .y=7, .text="5", .key=NCK_5},
31 { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
    NUMBER_BT_COLOR) | A_STANDOUT, .x=9, .y=7, .text="6", .key=NCK_6},
32 { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
    NUMBER_BT_COLOR) | A_STANDOUT, .x=1, .y=5, .text="7", .key=NCK_7},
33 { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
    NUMBER_BT_COLOR) | A_STANDOUT, .x=5, .y=5, .text="8", .key=NCK_8},
34 { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
    NUMBER_BT_COLOR) | A_STANDOUT, .x=9, .y=5, .text="9", .key=NCK_9},
35 { .default_attr=COLOR_PAIR(NUMBER_BT_COLOR), .click_attr=COLOR_PAIR(
    NUMBER_BT_COLOR) | A_STANDOUT, .x=9, .y=11, .text="," , .key=
    NCK_DPOINT},
36 { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=COLOR_PAIR(
    OPERATION_BT_COLOR) | A_STANDOUT, .x=1, .y=11, .text="+/-", .key=
    NCK_SIGN},
37 { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=COLOR_PAIR(
    OPERATION_BT_COLOR) | A_STANDOUT, .x=1, .y=3, .text="DEL", .key=
    NCK_DEL},
38 { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=COLOR_PAIR(
    OPERATION_BT_COLOR) | A_STANDOUT, .x=5, .y=3, .text="AC", .key=
    NCK_RESET},
39 { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=COLOR_PAIR(
    OPERATION_BT_COLOR) | A_STANDOUT, .x=17, .y=3, .text="/", .key=
    NCK_DIV},
40 { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=COLOR_PAIR(
    OPERATION_BT_COLOR) | A_STANDOUT, .x=17, .y=5, .text="*", .key=
    NCK_MUL},
41 { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=COLOR_PAIR(
    OPERATION_BT_COLOR) | A_STANDOUT, .x=17, .y=7, .text="-", .key=
    NCK_SUB},
42 { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=COLOR_PAIR(
    OPERATION_BT_COLOR) | A_STANDOUT, .x=17, .y=9, .text="+", .key=
    NCK_ADD},
43 { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=COLOR_PAIR(
    OPERATION_BT_COLOR) | A_STANDOUT, .x=17, .y=11, .text "=", .key=
    NCK_RESULT},
44 { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=COLOR_PAIR(
    OPERATION_BT_COLOR) | A_STANDOUT, .x=13, .y=5, .text="ln", .key=
    NCK_LOG},
45 { .default_attr=COLOR_PAIR(OPERATION_BT_COLOR), .click_attr=COLOR_PAIR(
    OPERATION_BT_COLOR) | A_STANDOUT, .x=13, .y=7, .text="exp", .key=
    NCK_EXP},
46 };
47
48 static void ncDrawWindow(void);

```

```
49 static void ncDrawButton(struct button_layout_t button, bool high);
50 static nc_key_t ncGuiMouse(void);
51 static nc_key_t ncGuiAck(nc_key_t key);
52
53 void ncGuiInit(){
54     setlocale(LC_ALL, "");
55     initscr();
56     clear();
57     noecho();
58     cbreak();
59     keypad(stdscr, TRUE);
60     if(has_colors() == FALSE)
61     {
62         endwin();
63         printf("Your terminal does not support color\n");
64         exit(1);
65     }
66     start_color();
67     init_pair(BLACK_WHITE, COLOR_BLACK, COLOR_WHITE);
68     init_pair(DISPLAY_COLOR, COLOR_GREEN, COLOR_BLACK);
69     init_pair(NUMBER_BT_COLOR, COLOR_YELLOW, COLOR_BLUE);
70     init_pair(OPERATION_BT_COLOR, COLOR_BLUE, COLOR_GREEN);
71     mmask_t mask=mousemask(BUTTON1_CLICKED, NULL);
72     if (mask==0){
73         endwin();
74         printf("Your terminal does not report mouse events\n");
75         exit(1);
76     }
77     ncDrawWindow();
78     wrefresh(win);
79 }
80
81 void ncGuiUpdate(char* num){
82     wattrset(win, COLOR_PAIR(DISPLAY_COLOR));
83     if (num==0) {
84         mvwprintw(win, 1, 1, "Error");
85     } else {
86         mvwprintw(win, 1, 1, "%19s", num);
87     }
88     wrefresh(win);
89 }
90
91 void ncGuiClean(){
92     endwin();
93 }
94
95 static void ncDrawButton(struct button_layout_t button, bool high){
96     wattrset(win, high? button.click_attr: button.default_attr);
```

```

97     if (button.text[1]==0){
98         mvwprintw(win,button.y,button.x," %s ",button.text);
99     } else {
100         mvwprintw(win,button.y,button.x,"%3s",button.text);
101     }
102 }
103
104 static void ncDrawWindow(){
105     // Mitte des Terminals
106     int height=13;
107     int width=21;
108     int x0 = (COLS-width)/2;
109     int y0 = (LINES-height)/2;
110     win = subwin(stdscr,height, width, y0 ,x0 );
111     // Loesche Hintergrund
112     wbkgd(win,COLOR_PAIR(BLACK_WHITE));
113     box(win,0,0);
114     // Zeichne Tasten
115     for (unsigned i=0; i< sizeof(layout)/sizeof(struct button_layout_t); i
        ++){
116         ncDrawButton(layout[i],false);
117     }
118 }
119
120 nc_key_t ncInput(){
121     int c=getch();
122     if ((c>='0') && (c<='9'))
123         return ncGuiAck(c-'0');
124     switch (c) {
125         case ',': // fall through
126         case '.':
127             return ncGuiAck(NCK_DPOINT);
128         case '+':
129             return ncGuiAck(NCK_ADD);
130         case '-':
131             return ncGuiAck(NCK_SUB);
132         case '/':
133             return ncGuiAck(NCK_DIV);
134         case '*':
135             return ncGuiAck(NCK_MUL);
136         case '#': // fall through
137         case KEY_F(1):
138             return ncGuiAck(NCK_SIGN);
139         case '=': // fall through
140         case KEY_ENTER: // fall through
141         case '\n':
142             return ncGuiAck(NCK_RESULT);
143         case KEY_BACKSPACE:

```

```

144         return ncGuiAck(NCK_DEL);
145     case KEY_DC:
146         return ncGuiAck(NCK_RESET);
147     case 'l':
148         return ncGuiAck(NCK_LOG);
149     case 'e':
150         return ncGuiAck(NCK_EXP);
151     case KEY_MOUSE:
152         return ncGuiMouse();
153     case 'q':           // fall through
154     case 'Q':
155         return NCK_QUIT;
156     default:
157         return NCK_NONE;
158 }
159 }
160
161 nc_key_t ncGuiMouse(void){
162     MEVENT event;
163     int ret=getmouse(&event);
164     if (ret != OK || !wenclose(stdscr,event.y,event.x))
165         return NCK_NONE;
166     int x=event.x-getbegx(win);
167     int y=event.y-getbegy(win);
168
169     for (unsigned i=0;
170          i< sizeof(layout)/sizeof(struct button_layout_t);
171          i++){
172         if ((layout[i].y==y) && (x>=layout[i].x)
173             && (x<=layout[i].x+2)) {
174             // Optischer Feedback: Taste blinkt
175             ncDrawButton(layout[i],true);
176             wrefresh(win);
177             usleep(100000);
178             ncDrawButton(layout[i],false);
179             wrefresh(win);
180             return layout[i].key;
181         }
182     }
183     return NCK_NONE;
184 }
185
186 static nc_key_t ncGuiAck(nc_key_t key){
187     // Optischer Feedback: Taste blinkt
188     for (unsigned i=0; i< sizeof(layout)/sizeof(struct button_layout_t); i
189          ++){
189         if (key == layout[i].key) {
190             ncDrawButton(layout[i],true);

```



```

191         wrefresh(win);
192         usleep(100000);
193         ncDrawButton(layout[i],false);
194         wrefresh(win);
195     }
196 }
197 return key;
198 }

```

D.3.5. nc-parseloop.h

```

1 // nc-parseloop.h
2 void ncParseLoop(char numbuf[]);

```

D.3.6. nc-parseloop.c

```

1 /*
2  In der Parse-Schleife werden aus den Eingaben Zahlen- und Operations-Token
3  geformt.
4  */
5 #include <locale.h> // für localeconv()
6 #include "stdio.h" // für sprintf()
7 #include "nc-gui.h"
8 #include "nc-token.h"
9 #include "nc-calc.h"
10
11 typedef enum {NCPS_START, NCPS_INT, NCPS_DEC} nc_parse_state_t;
12
13 void ncParseLoop(char numbuf[]){
14     nc_key_t keyev;
15     nc_parse_state_t parsestate = NCPS_START;
16     unsigned len=0;
17     const char decimal_point= *(localeconv()->decimal_point);
18     ncGuiUpdate(numbuf);
19
20     while((keyev = ncInput())!=NCK_QUIT){
21         if (keyev == NCK_RESET) {
22             ncResetCalc();
23             parsestate = NCPS_START;
24         }
25         switch(parsestate){
26             case NCPS_START:
27                 numbuf[0]=' '; // Vorzeichen
28                 numbuf[1]='0';
29                 numbuf[2]=0;
30                 len = 0;
31                 switch(keyev){
32                     case 1: case 2: case 3: case 4: // alle Durchläufer
33                     case 5: case 6: case 7: case 8: case 9:

```

```

33         numbuf[1] = '0'+keyev;
34         numbuf[2] = 0;
35         len = 1;
36         parsestate=NCPS_INT;
37         break;
38     case NCK_DPOINT:
39         numbuf[1] = '0';
40         numbuf[2] = decimal_point;
41         numbuf[3] = 0;
42         len = 2;
43         parsestate=NCPS_DEC;
44         break;
45     case NCK_SIGN:
46         if (numbuf[0] == '-') {
47             numbuf[0] = ' ';
48         } else {
49             numbuf[0] = '-';
50         }
51         break;
52     case NCK_SUB: case NCK_ADD: case NCK_DIV: // alle Durchlä
53         ufer
54     case NCK_MUL: case NCK_LOG: case NCK_EXP: case NCK_RESULT:
55         ncInputToken(ncMakeNumTokenFromString(numbuf));
56         double val = ncInputToken(ncMakeOpToken(keyev));
57         sprintf(numbuf,ncConvertSting,val);
58         parsestate = NCPS_START;
59         break;
60     default:
61         break;
62 }
63 break;
64 case NCPS_INT:
65     switch(keyev){
66     case 0: case 1: case 2: case 3: case 4: // alle Durchläufer
67     case 5: case 6: case 7: case 8: case 9:
68         if (len < ncMaxNumberDigits) {
69             numbuf[++len] = '0'+keyev;
70             numbuf[len+1] = 0;
71         }
72         break;
73     case NCK_DPOINT:
74         if (len < ncMaxNumberDigits) {
75             numbuf[++len] = decimal_point;
76             numbuf[len+1] = 0;
77             parsestate=NCPS_DEC;
78         }
79         break;
80     case NCK_DEL:

```

```

80         if (len == 1) {
81             parsestate=NCPS_START;
82         } else {
83             numbuf[len--] = 0;
84         }
85         break;
86     case NCK_SIGN:
87         if (numbuf[0] == '-') {
88             numbuf[0] = ' ';
89         } else {
90             numbuf[0] = '-';
91         }
92         break;
93     case NCK_SUB: case NCK_ADD: case NCK_DIV: // alle Durchlä
94         ufer
95     case NCK_MUL: case NCK_LOG: case NCK_EXP: case NCK_RESULT:
96         ncInputToken(ncMakeNumTokenFromString(numbuf));
97         double val = ncInputToken(ncMakeOpToken(keyev));
98         sprintf(numbuf,ncConvertSting,val);
99         parsestate = NCPS_START;
100        break;
101    default:
102        break;
103    }
104    break;
105    case NCPS_DEC:
106        switch(keyev){
107            case 0: case 1: case 2: case 3: case 4: // alle Durchläufer
108            case 5: case 6: case 7: case 8: case 9:
109                if (len < ncMaxNumberDigits) {
110                    numbuf[++len] = '0'+keyev;
111                    numbuf[len+1] = 0;
112                }
113                break;
114            case NCK_DPOINT:
115                break;
116            case NCK_DEL:
117                if (numbuf[len]==decimal_point){
118                    if ((numbuf[1] == '0') && (numbuf[2] == decimal_point)
119                    ) {
120                        parsestate=NCPS_START;
121                    } else {
122                        numbuf[len--] = 0;
123                        parsestate=NCPS_INT;
124                    }
125                } else {
126                    numbuf[len--] = 0;
127                }

```

```

126         break;
127     case NCK_SIGN:
128         if (numbuf[0] == '-') {
129             numbuf[0] = ' ';
130         } else {
131             numbuf[0] = '-';
132         }
133         break;
134     case NCK_SUB: case NCK_ADD: case NCK_DIV: // alle Durchlä
135         ufer
136     case NCK_MUL: case NCK_LOG: case NCK_EXP: case NCK_RESULT:
137         ncInputToken(ncMakeNumTokenFromString(numbuf));
138         double val = ncInputToken(ncMakeOpToken(keyev));
139         sprintf(numbuf, ncConvertSting, val);
140         parsestate = NCPS_START;
141         break;
142     default:
143         break;
144 }
145 break;
146 }
147 ncGuiUpdate(numbuf);
148 }

```

D.3.7. nc-token.h

```

1 // nc-token.h
2 #include <stdbool.h>
3 typedef enum {NCT_INVALID, NCT_NONE, NCT_FLUSH, NCT_NUMBER, NCT_FUNCTION,
4             NCT_OPERATOR} nc_token_type_t;
5
6 typedef struct {
7     nc_token_type_t type;
8     union {
9         double value;
10        struct {
11            nc_key_t operation;
12            unsigned short preference;
13        };
14    };
15 } nc_token_t;
16
17 nc_token_t ncMakeNumToken(double val);
18 nc_token_t ncMakeNumTokenFromString(const char *buf);
19 nc_token_t ncMakeOpToken(nc_key_t key);
20
21 bool ncIsNumber(nc_token_t token);

```

```

21 bool ncIsFunction(nc_token_t token);
22 bool ncIsOperator(nc_token_t token);
23 bool ncIsFlush(nc_token_t token);
24
25 // Für Warteschlangen und Stack wird eine gemeinsame Datenstruktur genutzt
26 typedef struct {
27     int sp;
28     int max;
29     nc_token_t *data;
30 } nc_tokenqueue_t;
31
32 void ncInitQueue(nc_tokenqueue_t **q, int max);
33 int ncInsertIntoQueue(nc_tokenqueue_t *q, nc_token_t token, int pos);
34 nc_token_t ncRemoveFromQueue(nc_tokenqueue_t *q, int pos);
35 nc_token_t ncPeekQueue(nc_tokenqueue_t *q, int pos);
36 int ncCountQueue(nc_tokenqueue_t *q);
37
38 bool ncIsEmptyQueue(nc_tokenqueue_t *q);
39 bool ncOnlyNumbers(nc_tokenqueue_t *q);
40
41 int ncPushQueue(nc_tokenqueue_t *q, nc_token_t token);
42 nc_token_t ncPopQueue(nc_tokenqueue_t *q);
43 nc_token_t ncPopBackQueue(nc_tokenqueue_t *q);
44 nc_token_t ncPeekTOS(nc_tokenqueue_t *q);

```

D.3.8. nc-token.c

```

1  /*
2   Token stellen Zahlen und Operationen dar.
3   Sie werden in den Warteschlangen und Stacks im Shunting-Yard-Algorithmus
   genutzt.
4  */
5  #include <stdbool.h>
6  #include <stdlib.h>
7  #include <assert.h>
8  #include "nc-gui.h"
9  #include "nc-token.h"
10
11 nc_token_t ncMakeNumToken(double val){
12     nc_token_t token;
13     token.value = val;
14     token.type=NCT_NUMBER;
15     return token;
16 }
17
18 nc_token_t ncMakeNumTokenFromString(const char *buf){
19     nc_token_t token;
20     char *endp;

```

```

21 token.value = strtod(buf,&endp);
22 if (*endp == '\0'){
23     token.type=NCT_NUMBER;
24 } else {
25     token.type=NCT_INVALID;
26 }
27 assert(token.type != NCT_INVALID);
28 return token;
29 }
30
31 nc_token_t ncMakeOpToken(nc_key_t key){
32     nc_token_t token;
33     switch(key){
34         case NCK_EXP: // fall through
35         case NCK_LOG:
36             token.type = NCT_FUNCTION;
37             token.operation = key;
38             token.preference = 3;
39             break;
40         case NCK_SUB: case NCK_ADD: // fall through
41             token.type = NCT_OPERATOR;
42             token.operation = key;
43             token.preference = 1;
44             break;
45         case NCK_DIV: case NCK_MUL: // fall through
46             token.type = NCT_OPERATOR;
47             token.operation = key;
48             token.preference = 2;
49             break;
50         case NCK_RESULT:
51             token.type = NCT_FLUSH;
52             token.operation = key;
53             token.preference = 0;
54             break;
55         default:
56             token.type = NCT_INVALID;
57     }
58     assert(token.type != NCT_INVALID);
59     return token;
60 }
61
62 bool ncIsNumber(nc_token_t token){
63     return (token.type == NCT_NUMBER);
64 }
65
66 bool ncIsFunction(nc_token_t token){
67     return (token.type == NCT_FUNCTION);
68 }

```

```

69
70 bool ncIsOperator(nc_token_t token){
71     return (token.type == NCT_OPERATOR);
72 }
73
74 bool ncIsFlush(nc_token_t token){
75     return (token.type == NCT_FLUSH);
76 }
77 void ncInitQueue(nc_tokenqueue_t **q, int max){
78     if (*q == NULL) {
79         *q = malloc(sizeof(nc_tokenqueue_t));
80     } else {
81         free((*q)->data);
82     }
83     (*q)->max = max;
84     (*q)->data = calloc(max, sizeof(nc_token_t));
85     (*q)->sp=0;
86 }
87
88 int ncInsertIntoQueue(nc_tokenqueue_t *q, nc_token_t token, int pos){
89     if ((q->sp == q->max) || (pos < -1) || (pos > q->sp))
90         return -1;
91     if (pos == -1) {
92         q->data[q->sp] = token;
93     } else {
94         for(int i= q->sp; i > pos; i--){
95             q->data[i] = q->data[i-1];
96         }
97         q->data[pos] = token;
98     }
99     q->sp++;
100     return 0;
101 }
102
103 nc_token_t ncRemoveFromQueue(nc_tokenqueue_t *q, int pos){
104     nc_token_t token = { .type = NCT_NONE};
105     if ((pos < -1) || (pos > q->sp-1))
106         return token;
107     if (pos == -1) {
108         pos = q->sp-1;
109     }
110     token = q->data[pos];
111     for(int i=pos; i < q->sp-1; i++) {
112         q->data[i] = q->data[i+1];
113     }
114     q->sp--;
115     return token;
116 }

```

```

117
118 nc_token_t ncPeekQueue(nc_tokenqueue_t *q, int pos){
119     nc_token_t token;
120     if ((pos<-1) || (pos>=q->sp)) {
121         token.type = NCT_NONE;
122     } else {
123         if (pos == -1)
124             pos = q->sp-1;
125         token = q->data[pos];
126     }
127     return token;
128 }
129
130 int ncCountQueue(nc_tokenqueue_t *q){
131     return q->sp;
132 }
133
134 int ncPushQueue(nc_tokenqueue_t *q, nc_token_t token){
135     return ncInsertIntoQueue(q, token, -1);
136 }
137
138 nc_token_t ncPopQueue(nc_tokenqueue_t *q){
139     return ncRemoveFromQueue(q, -1);
140 }
141
142 nc_token_t ncPopBackQueue(nc_tokenqueue_t *q){
143     return ncRemoveFromQueue(q, 0);
144 }
145
146 nc_token_t ncPeekTOS(nc_tokenqueue_t *q){
147     return q->data[q->sp-1];
148 }
149
150 bool ncIsEmptyQueue(nc_tokenqueue_t *q){
151     return (q->sp == 0);
152 }
153
154 bool ncOnlyNumbers(nc_tokenqueue_t *q){
155     for(int i=0; i<q->sp; i++) {
156         if (!ncIsNumber(ncPeekQueue(q,i))) return false;
157     }
158     return true;
159 }

```

D.3.9. nc-calc.h

```

1 // nc-calc.h
2 void ncResetCalc(void);

```



```
3 double ncInputToken(nc_token_t token);
```

D.3.10. nc-calc.c

```
1  /*
2  Hier finden die eigentlichen Berechnungen statt.
3  ncShuntingYard() überführt die Token in eine UPN-Form, die von ncEvaluate
   () ausgewertet wird.
4  */
5  #include <stdlib.h>
6  #include <math.h>
7  #include <stdbool.h>
8  #include <assert.h>
9  #include "nc-gui.h"
10 #include "nc-token.h"
11 #include "nc-calc.h"
12
13 static nc_tokenqueue_t *inputQueue=NULL;
14 static nc_tokenqueue_t *outputQueue=NULL;
15 static nc_tokenqueue_t *operatorStack=NULL;
16
17 static void ncEvaluate(void);
18 static void ncShuntingYard(void);
19
20 void ncResetCalc(){
21     ncInitQueue(&inputQueue,2);
22     ncInitQueue(&operatorStack,3);
23     ncInitQueue(&outputQueue,5);
24 }
25
26 static void ncShuntingYard() {
27     while (!ncIsEmptyQueue(inputQueue)) {
28         nc_token_t token = ncPopBackQueue(inputQueue);
29         if (ncIsNumber(token)) {
30             ncPushQueue(outputQueue,token);
31         }
32         if (ncIsFunction(token)){
33             ncPushQueue(outputQueue,token);
34             ncEvaluate();
35         }
36         if (ncIsOperator(token)){
37             while((!ncIsEmptyQueue(operatorStack)) &&
38                 (ncPeekTOS(operatorStack).preference >= token.preference)) {
39                 nc_token_t opToken = ncPopQueue(operatorStack);
40                 ncPushQueue(outputQueue,opToken);
41                 ncEvaluate();
42             }
43             ncPushQueue(operatorStack,token);
```

```

44         ncEvaluate();
45     }
46     if (ncIsFlush(token)) {
47         while (!ncIsEmptyQueue(operatorStack)){
48             ncPushQueue(outputQueue, ncPopQueue(operatorStack));
49         }
50         ncEvaluate();
51     }
52 }
53 }
54
55 static void ncEvaluate(){
56     while (!ncOnlyNumbers(outputQueue)) {
57         for(int i=0; i < ncCountQueue(outputQueue); i++) {
58             bool startOver = false;
59             nc_token_t token = ncPeekQueue(outputQueue, i);
60             if (ncIsOperator(token) || ncIsFunction(token)){
61                 startOver = true;
62                 nc_token_t arg1, arg2;
63                 if (ncIsOperator(token)) {
64                     arg1 = ncRemoveFromQueue(outputQueue, i-2);
65                     arg2 = ncRemoveFromQueue(outputQueue, i-2);
66                     ncRemoveFromQueue(outputQueue, i-2); // lösche Operation
67                 }
68                 if (ncIsFunction(token)) {
69                     arg1 = ncRemoveFromQueue(outputQueue, i-1);
70                     ncRemoveFromQueue(outputQueue, i-1); // lösche Operation
71                 }
72                 switch(token.operation){ // hole Argumente
73                     case NCK_ADD:
74                         ncInsertIntoQueue(outputQueue, ncMakeNumToken(arg1.value +
75                             arg2.value), i-2);
76                         break;
77                     case NCK_SUB:
78                         ncInsertIntoQueue(outputQueue, ncMakeNumToken(arg1.value -
79                             arg2.value), i-2);
80                         break;
81                     case NCK_MUL:
82                         ncInsertIntoQueue(outputQueue, ncMakeNumToken(arg1.value *
83                             arg2.value), i-2);
84                         break;
85                     case NCK_DIV:
86                         ncInsertIntoQueue(outputQueue, ncMakeNumToken(arg1.value /
87                             arg2.value), i-2);
88                         break;
89                     case NCK_LOG:
90                         ncInsertIntoQueue(outputQueue, ncMakeNumToken(log(arg1.
91                             value)), -1);

```

```
87         break;
88     case NCK_EXP:
89         ncInsertIntoQueue(outputQueue, ncMakeNumToken(exp(arg1.
90             value)), i-1);
91         break;
92     default: // sollte nie passieren
93         break;
94     }
95     if (startOver) continue;
96 }
97 }
98 }
99
100 double ncInputToken(nc_token_t token){
101     ncPushQueue(inputQueue, token);
102     ncShuntingYard();
103     nc_token_t outToken = ncPeekTOS(outputQueue);
104     if (ncIsNumber(outToken)) {
105         return outToken.value;
106     } else {
107         return 0.0;
108     }
109 }
```


Bibliography

- [Ada79] Douglas Noel Adams. *The Hitchhiker's Guide to the Galaxy*. Pan Books, 1979. ISBN: 0-330-25864-8.
- [AHU75] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Design and Analysis of Computer Algorithms*. Addison-Wesley, 1975.
- [AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [Are+15] Tilo Arens et al. *Mathematik*. 3rd ed. Spektrum Akademischer Verlag, 2015.
- [Ben00] Jon Bentley. *Programming Pearls*. 2nd ed. Addison-Wesley, 2000.
- [Ben06] Mordechai Ben-Ari. *Understanding Programming Languages*. For personal use free available: <http://www.weizmann.ac.il/sci-tea/benari/books/upl.zip>. John Wiley & Sons, 2006.
- [Ben88] Jon Bentley. *More Programming Pearls. Confessions of a Coder*. 1st ed. Addison-Wesley, 1988.
- [Bia09] Federico Biancuzzi. *Masterminds of Programming. Conversations with the Creators of Major Programming Languages*. O'Reilly, 2009.
- [BM77] Robert S. Boyer and J. Strother Moore. „A fast string searching algorithm“. In: *Communications of the ACM* 20.10 (1977), pp. 762–772.
- [Bra94] Andreas Brandstädt. *Graphen und Algorithmen*. Teubner, 1994.
- [Chu06] Wesley J. Chun. *Core Python Programming*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006. ISBN: 0132269937.
- [Cor+01] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd ed. MIT Press, 2001.
- [Dij63] Edgar W. Dijkstra. „An ALGOL 60 Translator for the X1“. In: *Annual Review in Automatic Programming* 3 (1963), pp. 329–345.
- [Edm08] Jeff Edmonds. *How to Think About Algorithms*. Cambridge University Press, 2008. ISBN: 978-0521614108.
- [Goo04] Dan Gookin. *C For Dummies*. 2nd ed. Wiley Publishing, 2004.
- [Har92] David Harel. *Algorithmics. The Spirit of Computing*. Addison-Wesley, 1992.
- [Har97a] Brian Harvey. *Advanced Techniques*. 2nd ed. Vol. 2. Computer Science Logo Style. MIT Press, 1997.
- [Har97b] Brian Harvey. *Beyond Programming*. 2nd ed. Vol. 3. Computer Science Logo Style. MIT Press, 1997.
- [Har97c] Brian Harvey. *Symbolic Computing*. 2nd ed. Vol. 1. Computer Science Logo Style. MIT Press, 1997.

- [HLW06] Helmut Herold, Bruno Lurz, and Jürgen Wohlrab. *Grundlagen der Informatik*. Pearson Studium, 2006.
- [HW94] Brian Harvey and Matthew Wright. *Simply Scheme: Introducing Computer Science*. MIT Press, 1994.
- [Knu69] Donald E. Knuth. *Fundamental Algorithms*. Vol. 1. The Art of Computer Programming. Addison-Wesley, 1969.
- [Knu81] Donald E. Knuth. *Seminumerical Algorithms*. Vol. 2. The Art of Computer Programming. Addison-Wesley, 1981.
- [Knu98] Donald E. Knuth. *Sorting and Searching*. Vol. 3. The Art of Computer Programming. Addison-Wesley, 1998.
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Prentice Hall, 1988.
- [Lut13] Mark Lutz. *Learning Python*. 5th. O'Reilly, 2013. ISBN: 978-1-4493-5573-9.
- [Man89] Udi Manber. *Introduction to Algorithms. A Creative Approach*. Addison-Wesley, 1989.
- [Mit02] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2002.
- [MS00] Brian McNamara and Yannis Smaragdakis. „Functional Programming in C++“. In: *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2000, pp. 118–129. ISBN: 1-58113-202-6. DOI: <http://doi.acm.org/10.1145/351240.351251>.
- [Oua97] Steve Oualline. *Practical C*. 3rd ed. O'Reilly, 1997.
- [Reg03] Regionales Rechenzentrum Niedersachsen, ed. *Die Programmiersprache C. Ein Nachschlagewerk*. 18th ed. Universität Hannover, 2003.
- [Sch01] Axel-Tobias Schreiner. *Object-Oriented Programming in ANSI C*. <http://www.cs.rit.edu/~ats/oop-2001-2/pdf/ooc.pdf>. Rochester Institute of Technology, 2001.
- [Sch14] Helmut O.B. Schellong. *Moderne C-Programmierung*. 3rd ed. Springer, 2014. ISBN: 978-3-642-54436-1.
- [Sed03] Robert Sedgewick. *Algorithms in Java. Part I – IV*. 3rd ed. Addison-Wesley, 2003.
- [Sed83] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983.
- [Sed90] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.
- [Ski08] Steven S. Skiena. *The Algorithm Design Manual*. 2nd ed. Springer, 2008.
- [Tat10] Bruce Tate. *Seven Languages in Seven Weeks*. Pragmatic Bookshelf, Inc., 2010.
- [Wir86] Niklaus Wirth. *Algorithms & Data Structures*. Prentice-Hall, 1986.
- [Wol16] Jürgen Wolf. *Grundkurs C: C-Programmierung verständlich erklärt*. 2nd ed. Rheinwerk Computing, 2016.
- [Zel03] John M. Zelle. *Python Programming: An Introduction to Computer Science*. Franklin Beedle & Associates, 2003.

Index

C-Schlüsselworte, Standardfunktionen und UNIX-Befehle sind in Terminalschrift dargestellt.
Python-Schlüsselworte wurden nicht in den Index aufgenommen.

- MARZULLO-Algorithmus, 168
- 42, 47
- $\mathcal{A}$ -Notation, 233
- Abstrahierung (Eigenschaft von Algorithmen), 24, 25
- Ada, 37
- AL-CHWĀRIZMĪ, ABU JAFAR MUHAMMAD IBN MŪSĀ, 21
- ALGOL, 39
- Algorithmus
 - Begriff, 21
 - Boyer-Moore-, *siehe* Boyer-Moore-Algorithmus
 - Definition, 22
 - Eigenschaften, 22–28
 - Funktionsmodell, 25, 28–32
 - Konditionalmodell, 30, 31
 - randomisiert, 27
 - von Euklid, 14–15, 57–58, 63–65
 - Zustandsmodell, 29–31
- Alphabet, 201
- Analytical Engine, 38
- ANSI C, 40
- Array, 119–129
 - Initialisierung, 121, 122
 - von char, 126
 - Zerfall, 123
- Assembler, 36, 66, 67, 363
- assert.h, 72
- atoi(), 127
- Aufzählungstyp, 118
- Ausdruck, 50
- Ausführbarkeit (Eigenschaft von Algorithmen), 23
- Ausführungsumgebung, 53
- Aussagelogik, 173–174
- auto, 134
- Automat, 192–206, 277–291
 - endlicher deterministischer, 194
 - Mealy, 194
 - Moore, 194
- BABBAGE, CHARLES, 38
- Backtracking, 254–259
- BACKUS, JOHN W., 38, 369
- Backus-Naur-Form, 207
 - erweiterte, 207
- Bezeichner, 94
- Bibliothek, 53, 317, 325–328
 - dynamische, 327–328
 - statische, 325–327
- Block, 56
- BNF, *siehe* Backus-Naur-Form
- _Bool, 178
- Boolesche Algebra, 174–178
- Boyer-Moore-Algorithmus, 274–276
- Brainfuck, 364
- break, 144
- Bubblesort, 17–21, 223, 240
- BYRON, AUGUSTA ADA KING, 38
- C
 - Dialekte, 40
 - Geschichte, 39–41
- C++, 37
- C11, 41, 319
- C18, 41
- C89, 40
- C90, 40
- C95, 40
- C99, 40, 413
- C#, 37
- Call by name*, *siehe* Namensersetzung

- Call by reference, siehe* Referenzübergabe
- Call by value, siehe* Wertübergabe
- `calloc()`, 102
- Caml, 363
- `cc`, 49
- `char`, 79
- `chmod`, 63
- `clang`, 46
- Coding convention, siehe* Programmier-Konvention
- COLMEAUER, ALAIN, 39, 369
- Compiler, 35, 36, 45, 67, 210
- Compilerfehler, *siehe* Syntaxfehler
- Compileroption
 - E, 66
 - S, 66
 - Wall, 49
 - Werror, 73
 - c, 66
 - std=, 49
 - v, 66
- `_Complex`, 86
- `const`, 93, 135
- `continue`, 144

- DAHL, OLE-JOHAN, 39, 369
- Damenproblem, 253–259
- Dangling Pointer*, 105
- Datei, 262
 - in Python, 269
 - Modus, 263
- Datenstrom, 262
- Definiertheit (Eigenschaft von Algorithmen), 23
- Deklaration, 51, 57
- Dereferenzierung, 101
- Determiniertheit (Eigenschaft von Algorithmen), 24, 27
- Determinismus (Eigenschaft von Algorithmen), 24, 27
- Dictionary (Typ in Python), 133
- DIJKSTRA, EDSGER W., 369
- `do ... while`, 141
- `double`, 85
- Dynamische Finitheit (Eigenschaft von Algorithmen), 24, 27

- EBNF, *siehe* Backus-Naur-Form
- Editor, 45

- Effizienz (Eigenschaft von Algorithmen), 26
- Eiffel, 37
- ELF, *siehe* Executable and Linkable Format
- `else`, 56
- ENIAC, 38
- `enum`, 118
- EUKLID VON ALEXANDRIA, 15
- Executable and Linkable Format, 321
- expression, *siehe* Ausdruck
- `extern`, 93, 134

- `fgetc()`, 264
- `fgets()`, 264
- Fibonacci-Zahlen, 228, 247
- File, siehe* Datei
- Finitheit (Eigenschaft von Algorithmen), 22
- `float`, 85
- `for`, 141
- Formale Sprache, 202
- FORTRAN, 38
- `fprintf()`, 264
- `fputc()`, 264
- `fputs()`, 264
- `fread()`, 264
- `fscanf()`, 264
- Funktion
 - Definition, 47, 55, 58
 - Deklaration, 76
 - Parameter, 47
 - Signatur, 76
- Funktionskörper, 47
- Funktionsrumpf, *siehe* Funktionskörper
- `fwrite()`, 264

- Ganzzahl, 79
- `gcc`, 46
- `_Generic`, 319
- Gleitkommazahl, 84
- `Go`, 363
- GOLDBERG, ADELE, 39
- Grammatik, 199
 - generative, 203
 - kontextbehaftet, 204
 - kontextfrei, 204
 - linksregulär, 204
 - rechtsregulär, 204
 - regulär, 204, 205, 277
- Graph, 155–159

- .h-Datei, *siehe* Headerdatei
- Haskell, 37, 63, 363
- Headerdatei, 53, 343, 351
- HOARE, CHARLES, 369
- HOPPER, GRACE, 38, 369
- if, 56
- INGALLS, DAN, 39
- Insertion Sort, 224
- int, 79
- International Obfuscated C Code Contest, 44
- Interpreter, 35, 36
- IOCCC, *siehe* International Obfuscated C Code Contest
- Iteration, 32
- Java, 37
- K&R C, 40
- Karnaugh-Diagramm, *siehe* Karnaugh-Veitch-Diagramm
- Karnaugh-Veitch-Diagramm, 177
- KAY, ALAN, 39, 369
- KERNIGHAN, BRIAN W., 40, 369
- Kleenesche Hülle, 202
- Kommentar, 47, 347–352
- Komplexität, 26, 229–241
- Korrektheit, 217
 - partielle, 222
 - totale, 222
- Kurzschlussoperator, 179
- KV-Diagramm, *siehe* Karnaugh-Veitch-Diagramm
- L-Value, 101
- LabView, 364
- λ -Kalkül, 38
- Landau-Kalkül, *siehe* $\mathcal{O}$ -Notation
- L^AT_EX, 355
- Laufzeitfehler, 70, 71
- Limbo, 363
- Linker, 53, 66, 67, 317–328
- Linkerfehler, 68
- Linkersymbol, 320
- Linkswert, *siehe* L-Value
- Lisp, 37, 38, 363
- Liste (Typ in Python), 130
- Literal, 81
- Logikfehler, 70, 72
- Logo, 37, 363
- long, 79
- Lua, 364
- main()
 - Parameter, 128
- main(), 46, 47
- make, 296–299
- Makro, 319
- malloc(), 102
- Maschinencode, 35, 36
- MCCARTHY, JOHN, 38, 369
- Memory Leak*, 105
- Menge (Typ in Python), 132
- Miranda, 37
- ML, 37, 363
- Modell der kleinen Menschen, 58
- Modellierung, 155–159
- Modula 2, 37
- Namensersetzung, 107
- Namensparameter, 106
- Normalform
 - disjunktive, 177
 - konjunktive, 177
- ntp, 168
- NULL, 105
- NYGAARD, KRISTEN, 39, 369
- .o-Datei, *siehe* Objektdatei
- $\mathcal{O}$ -Notation, 234
 - Rechenregeln, 236
- Objektdatei, 53, 66
- OCaml, 363
- Occam, 363
- Operator
 - & (Adressoperator), 98
 - & (Bitoperator), 185
 - &&, 179
 - !, 179
 - !=, 179
 - ^, 185
 - /, 51
 - ? ... :, 180
 - =, 94
 - ==, 56, 179
 - >>, 185

- >, 56, 179
- >=, 56, 179
- [], 120, 124
- <, 56, 179
- <<, 185
- <=, 56, 179
- ., 187
- *, 51
- , 51
- , 188
- +, 51
- ++, 188
- >, 116
- %, 51
- ., 114
- * (Dereferenzierung), 101
- |, 185
- ||, 179
- ~, 185
- Vorrang, 191, 411
- Oz, 364
- Parameter, 106–111
 - formaler, 106
 - tatsächlicher, 106
- Pascal, 37
- Piet, 364
- Plankalkül, 38
- Pointer, *siehe* Zeiger
- Portabilität, 359–362
- Positionsparameter, 106
- printf(), 127
- Programmier-Konvention, 342
- Programmiersprache
 - deklarativ, 37
 - funktional, 37
 - nicht-strikt, 37
 - objektorientiert, 37
 - reflexiv, 37
 - strikt, 37
- Programmiersprache, imperative, 37
- Programmierung
 - deklarativ, 37
 - funktional, 37
 - modular, 37
- Programmierung, imperative, 37
- Prolog, 37, 39, 364
- Präprozessor, 54, 66, 360
- bedingte Übersetzung, 360
- Direktive
 - #define, 360, 361
 - #elif, 360
 - #else, 360
 - #if, 360
 - #ifdef, 360
 - #ifndef, 360
 - #include, 53
- Macro, 360
- Pseudocode, 163–165
- Python, 61–65, 415–436
 - Geschichte, 42
- Q-BAL, 364
- Quartz Composer, 364
- RAM-Modell, 230–232
- Rangierbahnhof-Algorithmus, *siehe* Shunting-Yard-Algorithmus
- RAYMOND, ERIC S., 42
- Referenzübergabe, 107
- register, 93, 134
- Regulärer Ausdruck, 209
- Rekursion, 32, 58–59, 148, 243–259
- Relocation, 320
- Ressourcenbeschränkung, *siehe* Dynamische Finitheit (Eigenschaft von Algorithmen)
- restrict, 93, 136
- return, 47
- RITCHIE, DENNIS, 40, 42, 369
- VAN ROSSUM, GUIDO, 42, 369
- S#, 363
- scanf(), 108
- Scheme, 37, 363
- Schleife, 137–148
- Schlüsselwörter, 49
- Seiteneffekt, 108
- Semantik, 199
 - formale, 214
- Semantikfehler, *siehe* Logikfehler
- Shell, 45
- short, 79
- Shunting-Yard-Algorithmus, 336
- signed, 79
- Simula, 37, 39

- Simulink, 364
- sizeof, 80, 103
- Smalltalk, 37, 39, 363
- Speicherattribut, 93, 134
- Speicherklasse, 93, 134
- SQL, 37
- Squeak, 363
- Standardausgabe, 266
- Standardbibliothek, 51, 66
- Standardeingabe, 266
- Standardfehlerausgabe, 266
- Start-Objekt, 66
- static, 93, 95, 134
- stderr, *siehe* Standardfehlerausgabe
- stdin, *siehe* Standardeingabe
- stdout, *siehe* Standardausgabe
- String
 - Python, 130
- struct, 114
- Symbol, Linker, *siehe* Linkersymbol
- Symbole, 320
- Syntax, 199
- Syntaxdiagramm, 206
- Syntaxfehler, 67, 72

- Telefontest, 33
- Terminierung
 - Beweis, 226
 - Eigenschaft von Algorithmen, 24, 25
- THOMPSON, KENNETH L., 42, 369
- Tupel (Typ in Python), 131
- TURING, ALAN M., 38, 369
- Turing-Maschine, 38
- Typ, 75–134
 - abgeleiteter, 113–127
 - einer Funktion, 46
 - Inferenz, 88
- typedef, 118
- Typenalias, 118
- Türme von Benares, 250–253
- Türme von Hanoi, *siehe* Türme von Benares

- Ungarische Notation, 345–346
- union, 118
- unsigned, 79

- Variable, 91
 - anonym, 102
 - Lebensdauer, 103
 - Sichtbarkeit, 103
- benannt, 92
 - Adresse, 98
 - Definition, 93
 - Deklaration, 93
 - Lebensdauer, 95
 - Sichtbarkeit, 95
- Verbundtyp, 114
- Vergleichoperator, 56
- Virtuelle Maschine, 36
- VM, *siehe* Virtuelle Maschine
- void, 86
- volatile, 93, 135

- Wertübergabe, 107
- while, 139

- Z3, 38
- Zeiger, 4, 99
- ZUSE, KONRAD, 38, 369
- Zuweisung, 30, 94
- Zwischencode, 36
 - Python, 64

