Grundlagen der Praktischen Informatik

Basierend auf dem Skript von Henrik Brosenne

Stefan Siemer
Georg-August-Universität Göttingen
Institut für Informatik

Sommersemester 2025

Inhaltsverzeichnis

1 Organisation **1**

2 Haskell **7**
2.1 Einführung 7
2.2 Funktionen und Operatoren 9
2.3 Pattern Matching 16
2.4 Alternativen 19
2.5 Rekursion 21

3 Rechnermodelle **25**
3.1 Von-Neumann Architektur 25
3.2 Werke und Busse 25
3.2.1 Steuerwerk 29
3.2.2 Rechenwerk 30
3.2.3 Rechneraufbau 30
3.3 Befehlszyklus 31

4 Nachtrag: Haskell **33**
4.1 Boolesche Logik 33
4.2 Funktionen auf Listen und Tupeln 35

5 Betriebssysteme **41**
5.1 Einführung 41
5.1.1 Literatur 41
5.1.2 Was ist ein Betriebssystem? 41
5.1.3 Aufgaben eines Betriebssystems 43
5.1.4 Betriebsarten von Betriebssystemen 45
5.2 Prozessverwaltung 46
5.3 Scheduling 50
5.3.1 Scheduling in Batch-Systemen 51
5.3.2 Scheduling und Mehrprogrammbetrieb 52
5.4 Haskell 55
5.4.1 Typen 55
5.4.2 Typklassen 58
5.4.3 Eingeschränkte Typ-Parameter 60
5.4.4 Record Syntax 64
5.5 Prozess-Synchronisation 66
5.5.1 Mutex 69
5.6 Speicherverwaltung 71

5.6.1 Swapping 74

6 Automaten und Formale Sprachen 77
6.1 Einführung grundlegender Begriffe 77
6.2 Endliche Automaten 78
6.3 Reguläre Sprachen 89
6.3.1 Pumping-Lemma für reguläre Sprachen 90
6.3.2 Anwendung des Pumping-Lemmas 93
6.4 Grammatiken 94
6.5 Reguläre Grammatiken 95
6.6 Parser 99
6.6.1 Kontextfreie Grammatiken 99
6.6.2 Ableitung von kontextfreien Grammatiken 100
6.6.3 Syntaxanalyse 102
6.6.4 Rekursive LL-Parser 109
6.6.5 Nicht-rekursive LL-Parser 114

7 Logik 124
7.1 Aussagenlogik 124
7.1.1 Literatur 124
7.1.2 Grundlagen 125
7.1.3 Das Resolutionskalkül der Aussagenlogik 131
7.1.4 KNF/DNF Nachtrag 133
7.2 Prädikatenlogik 137
7.2.1 Einleitung 137
7.2.2 Formeln 138
7.2.3 Gültigkeitsbereich 142
7.2.4 Terme 144
7.2.5 Sprache 146
7.2.6 Zusammenfassung 149
7.2.7 Anwendung 151
7.2.8 Beispiele 151

8 Python 156
8.1 Einführung 156
8.2 Numerische Typen 166
8.2.1 Wahrheitswerte (`bool`) 166
8.2.2 Ganze Zahlen (`int`) 167
8.2.3 Gleitkommazahlen (`float`) 168
8.2.4 Operationen 169
8.3 Sequenz-Typen 170

8.3.1 Strings (`str`) . . . 170
8.3.2 Listen (`list`) . . . 172
8.3.3 Tupel (`tupel`) . . . 177
8.3.4 Dictionary (`dict`) . . . 178
8.4 Funktionen . . . 179
8.5 Kontrollstrukturen . . . 181
8.5.1 Konditionale (`if`) . . . 181
8.5.2 Schleifen (`for`, `while`) . . . 183
8.6 Klassen und Objekte . . . 187
8.7 Pandas . . . 191
8.7.1 Datenstrukturen . . . 192
8.7.2 Analyse Funktionen . . . 197
8.7.3 Einlesen und Auffüllen . . . 201
8.7.4 Kombinieren von Daten . . . 202
8.7.5 Group by: split-apply-combine . . . 206

9 Kryptographie **210**
9.1 Einführung . . . 210
9.2 Symmetrische Verschlüsselung . . . 213
9.3 Blockchiffren und Stromchiffren . . . 215
9.4 Pseudozufall . . . 222
9.5 Kryptosystem . . . 229
9.6 Kryptoanalyse . . . 233
9.7 Asymmetrische Verschlüsselung . . . 237

1 Organisation

Termine

Vorlesung, wöchentlich
ab erster Vorlesungswoche.

Dienstag	14:15 – 15:45 Uhr	MN 08
Online Videos	wöchentlich	Medien in StudIP
~~Freitag~~	~~14:15 – 15:45 Uhr~~	~~MN 08~~ bei Bedarf

Saalübung
ab zweiter Vorlesungswoche. Hier werden die Musterlösungen der Übungszettel besprochen.

Dienstag	etwa 16:00 – 17:30 Uhr	MN 08

Übungsgruppen IfI, wöchentlich
ab zweiter Vorlesungswoche. Hilfe, Nachfragen und gemeinsames Arbeiten in kleineren Gruppen.

Termine und Organisation der Übungen werden noch bekannt gegeben.

Lehrmanagement/Kommunikation

Die Kommunikation zur Vorlesung und den Übungen passiert hauptsächlich über:

- Das Lehrmanagementsystem *Stud.IP*.
- Die Kommunikationsplattform *Matrix/Element*.

Stud.IP/Element/Matrix

Tragen Sie sich in Stud.IP in die Veranstaltung *Grundlagen der Praktischen Informatik (Informatik II)* im SoSe 2025 ein.

- Die Folien und Videos von Vorlesung und Saalübung werden dort hinterlegt.
- Die Übungen werden dort ausgegeben und auch abgegeben.
- Die Klausur erfordert die Benutzung von ILIAS (zum Training hinterlegt).

Treten Sie der folgenden Gruppe in Element/Matrix bei um News zu bekommen und um Fragen zu stellen:

- gdpi:uni-goettingen.de

Übung - FlexNow

B.Inf.1102.Ue: Grundlagen der Praktischen Informatik - Übung

Die Anmeldung zur Übung in FlexNow ist **Voraussetzung** für die **Anmeldung zur Klausur**. Sie ist **nicht** mehr Voraussetzung für die Teilnahme am Übungsbetrieb.

Die **An- und Abmeldefrist** für die Übung in FlexNow ist bis zum **Ende der Vorlesungszeit** geöffnet. Am Besten melden Sie sich sofort an, damit Sie es nicht vergessen.

Es wird **keine** Nachmeldungen geben!

Wir können die Organisation von Übungen und Klausur für alle Teilnehmenden besser planen, sobald wir diese Daten bekommen haben.

Modulprüfung

B.Inf.1102.Mp: Grundlagen der Praktischen Informatik

E-Klausur

- 90 Minuten
- E-Prüfungsraum, MZG 1.116
- **08.08.2025, 08-12 Uhr**
- Die Anzahl der Kohorten richtet sich nach dem Bedarf.
- Sie können sich nur für die Klausur anmelden, nicht für eine der Kohorten.
- Nach dem Ende des Anmeldezeitraums werden Sie auf die Kohorten verteilt. Bei weniger als etwa $\sim$ 115 Teilnehmenden werden wir nur eine Kohorte anbieten (09-11 Uhr).
- Der zweite Termin wird am **14.10.2025** stattfinden (Details folgen).

Zulassungsbedingung zur Klausur 40% der Punkte auf jedem Übungszettel (bis auf zwei Ausnahmen) und damit erfolgreiche Absolvierung von B.Inf.1102.Ue. Falls Sie auf irgendeine Art ausfallen (Krankheit, Kinder, Pflege etc...), sprechen Sie bitte rechtzeitig mit uns, damit wir Probleme frühzeitig aus dem Weg räumen können.

oder

Erfolgreiche Absolvierung von B.Inf.1102.Ue in einem vorherigen Semester.

oder

Teilnahme an der Klausur zu *Informatik II / Grundlagen der Praktischen Informatik* in einem vorherigen Semester.

Probeklausur

E-Probe-Klausur

- Über das ILIAS System, damit Sie die Umgebung ausprobieren können.
- Verkürzte Altklausuraufgaben.
- Wir werden diese in der Saalübung besprechen.

Themenüberblick (vorläufig)

1. Funktionale Sprachen (Haskell)
2. Rechnermodelle (Von-Neumann-Rechner)
3. Betriebssysteme (Prozessverwaltung, Speicherverwaltung)
4. Automaten und Formale Sprachen (Reguläre/Kontextfreie Sprachen, Endliche Automaten, Pumping Lemma)
5. Logik (Logik und Logikprobleme, Prädikatenlogik)
6. Python als Skriptsprache (Grundlegendes, Anwendung von Python - Pandas zur Datenauswertung)
7. Kryptographie

Themenüberblick (Änderungen)

Änderung zum Vorjahr:

1. Keine großen Änderungen. Die Veranstaltung wird in den nächsten Jahren größere Änderungen bekommen.

Geplante Änderungen zum nächsten Jahr:

1. Python wird sehr wahrscheinlich zu Informatik I verlegt.
2. Kapitel Logik ausbauen
3. Prolog als logische Programmiersprache.
4. (evtl.) Shellskript passend zu Betriebssystemen.

2 Haskell

2.1 Einführung

Literatur

Haskell-Webseite
`https://www.haskell.org/`

Haskell-Wiki
`https://wiki.haskell.org/`

Hoogle
`https://hoogle.haskell.org/`

Graham Hutton
Programming in Haskell,
Cambridge University Press, 2016.

Richard Bird
Thinking Functional with Haskell,
Cambridge University Press, 2015.

Miran Lipovaca
Learn You a Haskell for Great Good!: A Beginner's Guide,
No Starch Press, 2011.
Online verfügbar: `http://learnyouahaskell.com/`

Funktionale Programmierung

- ist eine Methode Programme zu erstellen aus Funktionen und deren Anwendung und nicht aus Anweisungen und deren Ausführung.
- benutzt einfache mathematische Notationen, die es erlauben Probleme eindeutig, kurz und präzise zu beschreiben.
- hat eine einfache mathematische Basis, die die Anwendung von Methoden der Algebra auf die Eigenschaften von Programmen unterstützt.

(*frei übersetzt nach Richard Bird, Thinking Functional with Haskell*)

Es gibt verschiedene funktionale Programmiersprachen, eine davon ist **Haskell** (benannt nach dem amerikanischen Logiker Haskell B. Curry), die in dieser Veranstaltung eingesetzt wird.

GHC

Glasgow Haskell Compiler (GHC) ist der state-of-the-art, open-source Haskell-Compiler und bietet eine interaktive Umgebung für Entwicklung und Test.

Im Pool sind die Programmm **`ghc`** und **`ghci`** verfügbar.

GHCi ist eine interaktive Haskell-Umgebung. Haskell-Ausdrücke können direkt eingegeben werden, werden ausgewertet und das Ergebnis wird ausgegeben.

Außerdem ermöglicht GHCi das Kompilieren und Laden von Quelltext, um ihn zu testen, sowie das Einbinden von Modulen und das Ausgeben von Informationen über Funktionen, Typklassen, Datentypen und Module.

Startet man **`ghci`** erhält man folgende Ausgabe.

```
$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/
:? for help
Prelude>
```

Das Prompt **`Prelude`** gibt an, dass die Standardbibliothek mit grundlegenden Funktionen, sowie vordefinierten Typen und Werten geladen ist.

GHCi kann man als interaktive Testumgebung benutzen.

```
Prelude> 7+5
12
Prelude> sqrt 30.25
5.5
```

Beendet wird GHCi mit **`:quit`** oder **`:q`**.

```
Prelude> :q
Leaving GHCi.
```

2.2 Funktionen und Operatoren

Funktion

In der Mathematik ist eine **Funktion** eine Relation (Beziehung) zwischen zwei Mengen, die jedem Element (Funktionsargument) der einen Menge (Definitionsbereich) genau ein Element (Funktionswert) der anderen Menge (Wertebereich) zuordnet.

Ist X der Definitionsbereich und Y der Wertebereich, dann schreibt man

- $f : X \to Y$ für die Funktion (Deklaration)
- $f(x)$ für den Funktionswert aus Y, der dem Funktionsargument x aus X von der Funktion zugeordnet wird (Definition).

Die Funktion f bildet ein Argument x aus X auf einen Wert $f(x)$ aus Y ab.

Haskell Typen

In Haskell spricht man nicht von Mengen, sondern von **Typen**.

Typen sind Mengen von Elementen mit bestimmten Eigenschaften.

Beispiele

`Float`	Gleitkomma-Zahlen mit einfacher Genauigkeit
`Double`	Gleitkomma-Zahlen mit doppelter Genauigkeit
`Int`	beschränkte ganze Zahlen
`Integer`	unbeschränkte ganze Zahlen
`Bool`	Wahrheitswerte (`True`, `False`)
`Char`	Aufzählungstyp (nicht negative ganze Zahlen), dessen Werte Zeichen repräsentieren
`[`*Type*`]`	(beliebig lange) Listen mit Werten vom Typ *Type* z.B. `[Bool]`
`String`	Zeichenketten, ist Platzhalter für `[Char]`
`(`*TypeA*`,` *TypeB*`)`	Paare (2-Tupel) mit Typen *TypeA* und *TypeB* z.B. `(String, Int)`
TypeA `->` *TypeB*	Funktionen mit *TypeA* als Definitionsbereich und *TypeB* als Wertebereich z.B. `Float -> Float`

Bemerkung

- Der Typ **Int** umfasst mindestens das Intervall $[-2^{29}, 2^{29} - 1]$.
- Die Zahlenformate **Float** und **Double** entsprechen in Zahlenbereich und Genauigkeit mindestens den Gleitkommazahlen des IEEE-754-Standard.
- Die Werte des Aufzählungstyps **Char** repräsentieren Unicode-Zeichen (ISO/IEC 10646). Das ist eine Erweiterung der Latin-1 (ISO 8859-1) Zeichenmenge (die ersten 256 Zeichen), die wiederum eine Erweiterung der ASCII Zeichenmenge (die ersten 128 Zeichen) ist.

Haskell Funktion

Eine Funktion **f** bildet ein Argument vom Typ **X** auf einen Wert vom Typ **Y** ab.

In Haskell-Notation wird diese Deklaration wie folgt ausgedrückt.

```
f :: X -> Y
```

Beispiele

```
sqrt    :: Float -> Float
first   :: (String, Int) -> String
second  :: (String, Int) -> Int
not     :: Bool -> Bool
and     :: [Bool] -> Bool
logBase :: Float -> Float -> Float
```

In Haskell kann man

```
f x
```

für die Anwendung der Funktion **f** auf das Argument **x** schreiben.

Beispiele

```
sqrt 25.0
not True
and [True, False, True]
logBase 2 10
```

Bemerkung

- `logBase` bildet ein Argument vom Typ `Float` auf eine Funktion ab. Diese Funktion wiederum bildet ein `Float`-Argument auf einen `Float`-Funktionswert ab.
- Das entspricht dem, was man aus der Mathematik kennt, die Funktionen $\log_2$ und ln entsprechem dem Wert der Haskell Funktionsaufrufe `logBase 2` und `logBase e`.
- `e` ist in diesem Fall eine konstante Funktion.

```
e :: Float
```

Haskell Operatoren

Ein Spezialfall von Funktionen sind Operatoren.

Mathematisch ist ein Operator eine Funktion

$$op : X \to [Y \to Z]$$

mit

$$[Y \to Z] = \{f \mid f : Y \to Z\}$$

für dessen Anwendung sowohl die Präfix- als auch die Infix-Schreibweise

$$\underbrace{op(x)}_{\in [Y \to Z]}(y) \;=\; x \; op \; y \;\in Z \quad \text{für alle } x \in X, y \in Y$$

verwendet werden kann.

In Haskell wird aus einer Funktions- eine Operatordeklaration, indem der Bezeichner in rund Kammern eingeschlossen wird.

```
(op) :: X -> Y -> Z
```

Die Anwendung des Operators kann entweder präfix (Operator in runden Klammern) oder infix erfolgen.

```
(op) x y
x op y
```

Bemerkung

- In Haskell gelten die Zeichen `!#$%&*+./<=>?@^|-~:`) als Symbole, aber der Unterstrich _ ist kein Symbol.
- Bezeichner für Operatoren dürfen ausschließlich Symbole enthalten.
- Funktionsnamen dürfen keine Symbole enthalten, nur Zeichen, Ziffern und den Unterstrich.

Wichtige Operatoren:

`+`, `-`, `*`, `/`	Arithmetik
`==`, `/=`	Gleichheit/Ungleichheit
`<`, `<=`, `>`, `>=`	Vergleiche
`&&`, `||`	Verknüpfung von Wahrheitswerten

Beispiele

```
7+5
(*) 6 7
(3+4)*8
3 /= 5
3 >= 5
True || False
(&&) True False
```

Bemerkung

Durch runde Klammern in Ausdrücken kann man die Auswertungsreihenfolge beeinflussen.

Funktionen/Operatoren definieren

In der Mathematik besteht eine Funktionsdeklaration aus Angabe von Definitions- und Wertebereich.

Die Funktionsdefinition beschreibt wie jedes Element des Definitionsbereichs auf ein Element des Wertebereiches abgebildet wird.

In der Regel werden dazu bereits vorher definierte Operatoren und Funktionen verwendet.

$$f : \mathbb{Z} \to \mathbb{Z}$$
$$f(x) = x^2 + x$$

In Haskell macht man das genauso. Man kann z.B. in **prelude** definierte Operatoren und Funktionen verwenden.

```
f :: Int -> Int
f x = x * x + x
```

Bemerkung

Das Standard-Modul **prelude** wird per default in alle Haskell-Module importiert und enthält viele nützliche Operatoren, Funktionen und Definitionen.

Die Möglichkeit Funktionen direkt durch andere Funktionen zu definieren besteht in Haskell ebenfalls.

$$f : \mathbb{R} \to \mathbb{R}$$
$$g : \mathbb{R} \to \mathbb{R}$$
$$f = g$$

Beispiel

```
log2 :: Float -> Float
log2 = logBase 2
```

Erinnerung

logBase bildet ein Argument vom Typ **Float** auf eine Funktion **Float->Float** ab.

Die Deklaration eines Operators **op** erfolgt immer über die Deklaration der zugehörigen Funktion **(op)**.

Bei der Definition hat man die Wahl zwischen Infix- und Präfixdarstellung.

Beispiel

Implikation über dem booleschen Körper.

$$\Rightarrow : \ \mathbb{F}_2 \to [\mathbb{F}_2 \to \mathbb{F}_2]$$
$$a \Rightarrow b \ = \ \neg a \vee b$$

Präfix

```
(==>) :: Bool -> Bool -> Bool
(==>) a b = not a || b
```

Infix

```
(==>) :: Bool -> Bool -> Bool
a ==> b = not a || b
```

Konstante Funktionen

Konstante Funktionen werden durch explizite Angabe des Werts definiert.

```
-- euler number
e :: Float
e = 2.71828182845904523536028874
```

Oder mit Hilfe bereits vorher definierter Funktionen (z.B. aus `prelude`).

```
-- euler number
e :: Float
e = exp 1
```

Bemerkung

`--` leitet einen Zeilenkommentar ein, d.h. bis zum Ende der Zeile wird alles Nachfolgende ignoriert.

GHCi Kommandos

GHCi verarbeitet Eingaben zeilenweise, um mehrzeilige Eingaben zu verarbeiten, z.B. um Funktionen, Operatoren, etc. zu definieren, kann man mit dem Kommando `:{` einen *multiline block* öffnen und mit `:}` schließen.

```
Prelude> :{
Prelude| f :: Int -> Int
Prelude| f x = x * x + x
Prelude| :}
Prelude> f 5
30
```

Das Kommando **:type** oder **:t** gibt Auskunft über den Typ des nachfolgenden Ausdrucks.

```
Prelude> :t 'A'
'A' :: Char
Prelude> :t sqrt
sqrt :: Floating a => a -> a
```

'A' ist eine konstante Funktion, die eine Wert vom Typ **Char** zurückliefert.

sqrt ist nicht für einen bestimmten Typ definiert, sondern eine Funktion **a -> a**, wobei für den Typ **a** gelten muss, dass es sich um einen Gleitkomma-Typ handelt.

Tatsächlich ist es etwas komplizierter. **Floating** ist eine *Typklasse*. Typklassen werden später noch behandelt.

Eine Methode um selbst definierte Funktionen, Operatoren, etc. in GHCi zu benutzen ist, die Definitionen in einer Datei, üblicherweise mit der Endung **.hs**, zu speichern und in GHCi zu laden.

Eine Datei wird geladen, indem der Dateiname beim Starten von **ghci** auf der Kommandozeile übergeben wird.

Alternativ kann man im laufenden GHCi mit den Kommandos

```
:load [file]
:l [file]
```

die Datei *file* laden.

Die Kommandos

```
:reload
:r
```

laden die zuletzt geladene Datei, z.B. nach Änderungen, erneut.

```
> cat intro.hs
-- parabola
f :: Int -> Int
f x = x * x + x

-- logarithm to base 2
log2 :: Float -> Float
log2 = logBase 2
```

```
-- implication
(==>) :: Bool -> Bool -> Bool
a ==> b = not a || b

-- euler number
e :: Float
e = exp 1
```

```
> ghci intro.hs
...
ghci> f 5
30
ghci> log2 8
3.0
ghci> log2 0.25
-2.0
ghci> True ==> False
False
ghci> e
2.7182817
ghci> :q
```

```
> ghci
...
ghci> :l intro.hs
...
ghci> f 5
30
...
ghci> :q
```

2.3 Pattern Matching

In den Beispielen zu Funktions- und Operatordefinition wurde bereits *pattern matching* (Mustererkennung) verwendet.

Die Funktion

```
f :: Int -> Int
f x = x * x + x
```

benutzt das *pattern* **x** und der Operator

```
(==>) :: Bool -> Bool -> Bool
a ==> b = not a || b
```

die *pattern* `a` und `b` als Platzhalter für die Argumente, mit denen Funktion/-Operator aufgerufen werden.

Die verwendeten (allgemein gültigen) *pattern* können jeweils jeden beliebigen Wert des Definitionsbereichs repräsentieren. D.h. für jede gültige Anwendung der Funktion passen (*match*) diese *pattern* und die Funktion mit ihren Argumenten wird durch den Ausdruck, der dem passenden *pattern* zugeordnet ist, ersetzt.

Für die Auswertung eines Ausdrucks werden Funktionen und Argumente solange ersetzt, bis ein Ausdruck erreicht ist, der nur noch aus einem Element eines Typs (`Int`, `Float`, `(Int, Float)`, `[Char]` etc.) besteht.

Beispiel

```
f :: Int -> Int
f x = x * x + x
```

```
f 5
5 * 5 + 5
25 + 5
30
```

```
(==>) :: Bool -> Bool -> Bool
a ==> b = not a || b
```

```
True ==> False
not True || False
False || False
False
```

Bemerkungen

- Für Haskell ist alles Funktion/Operator, z.B. auch die arithmetischen Operatoren. Das Prinzip, die Argumente aus dem Definitionsbereich durch Funktionen/Operatoren auf einen Wert des Wertebereiches abzubilden, bleibt erhalten. Nur wird der Wert, durch den Funktion/-Operator und Argumente ersetzt werden, nicht durch *pattern matching* o.Ä. bestimmt, sondern durch Berechnungen des Prozessors.
- Haskell benutzt *lazy evaluation* um eine möglichst effiziente Auswertung zu erreichen, z.B. soll eine Funktion mit identischen Argumenten nur einmal ausgewertet werden.

Pattern können auch weniger allgemeingültig sein, bis hin zu *pattern*, die nur einen Wert des Definitionsbereich repräsentieren.

Eine Funktions-/Operatordefinition kann mehrere *pattern* enthalten. Jedem *pattern* muss eine Ausdruck zugeordnet werden, durch den Funktion/Operator und Argumente, bei passenden *pattern*, ersetzt werden. Im einfachsten Fall ist das ein Wert des Wertebereichs.

```
f :: ...
f [pattern_1] = [expr_1]
f [pattern_2] = [expr_2]
f ...
```

Die *pattern* werden von oben nach unten ausgewertet, d.h., dass erste passende *pattern* bestimmt den Ausdruck, durch den die Funktions-/Operatoranwendung ersetzt wird.

Beispiele

Negation der Aussagenlogik

```
neg :: Bool -> Bool
neg False = True
neg True  = False
```

Konjunktion der Aussagenlogik (Version 1)

```
(<&>) :: Bool -> Bool -> Bool
(<&>) False False = False
(<&>) False True  = False
(<&>) True  False = False
(<&>) True  True  = True
```

Konjunktion der Aussagenlogik (Version 2)

```
(<&>) :: Bool -> Bool -> Bool
False <&> False = False
False <&> True  = False
True  <&> False = False
True  <&> True  = True
```

Konjunktion der Aussagenlogik (Version 3)

```
(<&>) :: Bool -> Bool -> Bool
(<&>) True True  = True
(<&>) a b = False
```

Konjunktion der Aussagenlogik (Version 4)

```
(<&>) :: Bool -> Bool -> Bool
True <&> True  = True
_ <&> _ = False
```

Bemerkung

Den Unterstrich _ nennt man ein *wildcard* oder *don't care pattern*.

2.4 Alternativen

Alternativen

Betrachten wir folgende Definition des Betrags einer ganzen Zahl.

$$\text{abs}(x) = \begin{cases} -x & \text{wenn } x < 0 \\ x & \text{sonst} \end{cases}$$

Um diese Definition umzusetzen, bietet sich ein **bedingter Ausdruck** an.

if *[test]* **then** *[expr_if]* **else** *[expr_else]*

Abhängig vom *[test]* (ein Ausdruck der nach **Bool** ausgewertet wird) nimmt der Ausdruck für

- *[test]* **== True** den Wert von *[expr_if]* an.
- *[test]* **== False** den Wert von *[expr_else]* an.

```
absolute :: Int -> Int
absolute x = if x < 0 then -x else x
```

In Haskell sind **bewachte Gleichungen (guarded equations)** möglich, mit denen sich Alternativen, insbesondere mehr als zwei, sehr übersichtlich realisieren lassen.

```
f :: X -> Y -> Z
f x y
  | [guard_1] = [expr_1]
  | [guard_2] = [expr_2]
  | ...
  | [guard_n] = [expr_n]
  | otherwise = [expr_default]
```

Die Wächter *[guard_1]* ... *[guard_n]* sind Test (**Bool**-Ausdrücke), die von Oben nach Unten ausgewertet werden.

Der Wert der Funktion ist der erste Ausdruck *[expr_i]* für dessen Wächter *[guard_i]* **== True** gilt.

Es gilt **otherwise == True**.

Beispiel

Exclusives Oder der Aussagenlogik

```
-- XOR
(<+>) :: Bool -> Bool -> Bool
(<+>) a b
  | a == b    = False
  | otherwise = True
```

Vergleichsfunktion

$$\text{compare}(x, y) = \begin{cases} +1 & \text{wenn } x > y \\ -1 & \text{wenn } x < y \\ 0 & \text{sonst} \end{cases}$$

```
comp :: Int -> Int -> Int
comp x y
  | x > y     = 1
  | x < y     = -1
  | otherwise = 0
```

2.5 Rekursion

Rekursion

Definitionen von Folgen lassen sich häufig **rekursiv** sehr kompakt angeben.

Beispiel

Die rekursive Näherung der Quadratwurzel einer positiven Zahl `a` nach Heron.

$$\begin{aligned} x_0 &= a \\ x_n &= (x_{n-1} + a/x_{n-1})/2 \qquad \text{für } n > 0 \end{aligned}$$

Daraus lässt sich eine rekursive Funktion zur Berechnung des n-ten Folgenglieds ableiten.

$$\begin{aligned} &\text{heron} :: \mathbb{N} \times \mathbb{R}^+ \to \mathbb{R}^+ \\ &\text{heron}(n, a) = \begin{cases} \Big(\text{heron}(n-1, a) + a/\text{heron}(n-1, a)\Big)/2 & \text{für } n > 0 \\ a & \text{sonst} \end{cases} \end{aligned}$$

Diese rekursive Definiton lässt sich in Haskell direkt umsetzen.

```
heronA :: (Int, Double) -> Double
heronA (n, a)
  | n > 0     = (heronA ((n-1), a) + a / heronA ((n-1), a))/2
  | otherwise = a
```

Häufig sinnvoller ist aber die Sichtweise, bei der die Funktion heron mit Argument n auf eine Funktion heron_n abbildet, die das n-te Folgenglied zum Argument a berechnet.

$$\begin{aligned} &\text{heron}_n :: \mathbb{R}^+ \to \mathbb{R}^+ \\ &\text{heron}_n(a) = \begin{cases} \Big(\text{heron}_{n-1}(a) + a/\text{heron}_{n-1}(a)\Big)/2 & \text{für } n > 0 \\ a & \text{sonst} \end{cases} \end{aligned}$$

```
heronB :: Int -> Double -> Double
heronB n a
  | n > 0     = (heronB (n-1) a + a / heronB (n-1) a)/2
  | otherwise = a
```

Haskell bietet nicht die Möglichkeit Speicherplätze zu reservieren, deshalb können in Funktionen auch keine Speicherplätze für lokale Variablen bereitgestellt werden, z.B. zur Speicherung von Zwischenergebnissen.

Mit der Klausel **where** können Platzhalter für lokale Definitionen, die nur im Kontext einer Funktionsdefinition gültig sind, angelegt werden.

Insbesondere zur Strukturierung und Verbesserung der Lesbarkeit von Definitionen ist **where** nützlich.

Beispiel

```
heronC :: Int -> Double -> Double
heronC n a
  | n > 0     = (x + a/x)/2
  | otherwise = a
  where x = heronC (n-1) a
```

Mit der Klausel **where** können mehrere lokale Definitionen vorgenommen werden.

```
f :: ...
f ...
  | [guard_1] = [expr_1]
  | [guard_2] = [expr_2]
  | ...
  | [guard_n] = [expr_n]
  | otherwise = [expr_default]
  where
    [expr_where_1]
    [expr_where_2]
    ...
    [expr_where_n]
```

Beispiel

Fibonacci-Folge

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \text{ für } n \geq 2$$

```
fibA :: Int -> Int
fibA n
  | n == 0    = 0
  | n == 1    = 1
  | otherwise = x + y
  where
    x = fibA (n-1)
    y = fibA (n-2)
```

Bei der Fibonacci-Folge bietet sich stattdessen die Verwendung passender *pattern* an.

```
fibB :: Int -> Int
fibB 0 = 0
fibB 1 = 1
fibB n = fibB (n-1) + fibB (n-2)
```

Bemerkung. Beide Versionen sind nicht korrekt definiert, denn es werden auch ungültige (negative) Argumente akzeptiert.

Fehler, z.B. ungültige Argumente, kann man mit der Funktion **`error`** aus **`Prelude`** behandeln.

Die Funktion **`error`** wird mit einem **`String`** (Zeichenfolge eingeschlossen in doppelte Hochkommata ") als Argument aufgerufen. Die Auswertung der aktuellen Funktion wird unterbrochen und eine Ausnahme (*exception*) ausgelöst, die den **`String`** ausgibt.

Beispiel

```
fibC :: Int -> Int
fibC 0 = 0
fibC 1 = 1
fibC n
  | n < 0     = error "illegal argument"
  | otherwise = fibC (n-1) + fibC (n-2)
```

Bemerkung.

Auf das Auslösen und Behandeln von Ausnahmen wird nicht weiter eingegangen.

Da Operatoren und Funktionen in Haskell im wesentlichen identisch sind, ist auch die rekursive Definition von Operatoren möglich.

Beispiel

Näherung der Quadratwurzel nach Heron als Operator.

```
(<##>) :: Int -> Double -> Double
0 <##> a = a
n <##> a
  | n > 0     = (x + a/x)/2
  | otherwise = error "illegal argument"
  where x = (n-1) <##> a
```

Wiederholungen

Viele Programmiersprachen bieten meist mehrere Konstrukte zur Wiederholung von Codeblöcken an, sogenannte Schleifen.

Schleifen gibt es in Haskell nicht.

Wiederholungen müssen durch Rekursion realisiert werden.

3 Rechnermodelle

3.1 Von-Neumann Architektur

Im Rahmen des Baus des *Electronic Discrete Variable Automatic Computers (EDVAC)* beschrieb John von Neumann 1945 ein revolutionäres Konzept.

Die entscheidende Neuerung bestand darin, die Befehle des Programms wie die zu verarbeitenden Daten zu behandeln, sie binär zu kodieren und im internen Speicher zu verarbeiten.

Dieses Konzept wird heute als **Von-Neumann Architektur** bezeichnet.

Konrad Zuse hatte viele Ideen der von-Neumann Architektur schon 1936 ausgearbeitet, 1937 patentiert und 1938 in der Z1 Maschine mechanisch realisiert. Allerdings wird allgemein angenommen, dass von Neumann Zuses Arbeiten nicht kannte.

Die meisten der heute gebräuchlichen Computer basieren auf dem Grundprinzip der Von-Neumann Architektur.

Grundprinzipien der Von-Neumann Architektur

- Programmsteuerung durch universelle Hardware
- Gemeinsamer Speicher für Daten und Programme
- Hauptspeicher besteht aus adressierbaren Zellen
- Programm besteht aus einer Folge von Befehlen
- Sprünge sind möglich (bedingte und unbedingte)
- Speicherung erfolgt binär

3.2 Werke und Busse

Ein Von-Neumann Rechner besteht aus folgenden Komponenten.

1. **Rechenwerk** (*Arithmetic Logic Unit, ALU*). Führt Rechenoperationen und logische Verknüpfungen durch.
2. **Steuerwerk** (*Control Unit*). Interpretiert die Anweisungen eines Programmes und steuert die Befehlsabfolge, auch Leitwerk genannt.

3. **Speicherwerk** (*Memory*). Speichert sowohl Programme als auch Daten, die für das Rechenwerk zugänglich sind.
4. **Eingabe-/Ausgabewerk** (*Input/Output Unit, I/O Unit*). Steuert die Ein- und Ausgabe von Daten, zum Anwender oder zu anderen Systemen.
5. **Bus-System.** Datenbus, Adressbus, Steuerbus. Verbindet die Komponenten des Rechners untereinander (nicht Teil des ursprünglichen Entwurfs).

Rechenwerk.

- Häufig synonym mit ALU (*Arithmetic Logic Unit*) gebraucht.

  Eigentlich ist die ALU eine zentrale **Komponente** des Rechenwerks und ein Rechenwerk kann auch mehrere ALUs enthalten.

- Das Rechenwerk besteht zusätzlich aus einer Reihe von **Registern**.

  Register sind Speicherbereiche im Rechenwerk, die unmittelbar z.B. Operanden oder Ergebnisse von Berechnungen aufnehmen.

  Die ALU selbst enthält keine Register und ist ein reines Schaltnetz.

Steuerwerk + Rechenwerk = Hauptprozessor (*Central Processing Unit, CPU*)

Der Arbeitsspeicher besitzt einen **eingeschränkten Adressumfang**.

Persistente Speicher wie z.B. Festplatten oder Caches, sowie Register zählt man logisch nicht zum Speicher.

Die Zugriffsgeschwindigkeit zum Arbeitsspeicher sollte der Arbeitsgeschwindigkeit des Hauptprozessors angepasst sein.

Für die verschiedenen Einsatzbereiche der Speicher werden unterschiedliche Speicherarten verwendet, die sich unterscheiden hinsichtlich

- Speichermedium und physikalischem Arbeitsprinzip,
- Organisationsform,
- Zugriffsart,
- Leistungsparameter,

- Preis.

Haupt- bzw. Arbeitsspeicher für Programme und Daten.

- Speicher mit wahlfreiem Zugriff *(random access memory, RAM)*.

  Jede Speicherzelle kann über ihre Speicheradresse direkt angesprochen werden (wahlfreier Zugriff).

  Der Begriff RAM wird heute im Sinne von Schreib-Lese-Speicher mit wahlfreiem Zugriff (*read-write-RAM*) verwendet.

- Nur-Lese-Speicher *(read only memory, ROM)* ist in der Regel auch Speicher mit wahlfreiem Zugriff.

- Löschbarer, programmierbarer Nur-Lese-Speicher (*Erasable Programmable Read-Only-Memory, EPROM)*.

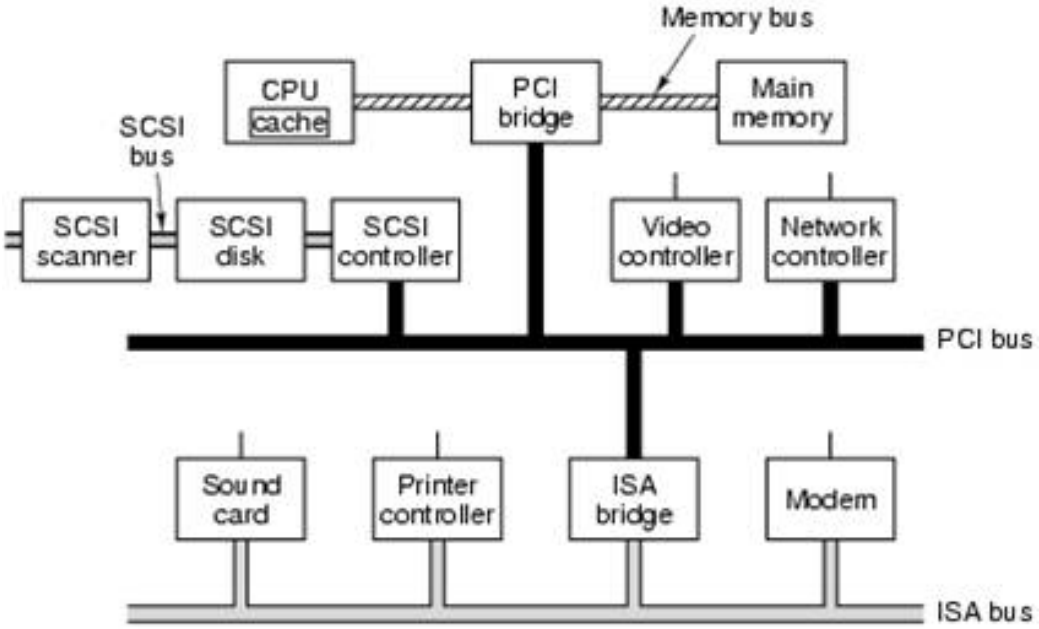
Das **Ein-/Ausgabewerk** steuert die Ein- und Ausgabe von Daten.

- zum Anwender.
  - Lochkarte
  - Tastatur
  - Maus
  - Scanner
  - Bildschirm
  - Drucker
- zu anderen Systemen (Schnittstellen).
  - Disketten
  - Festplatten
  - Magnetbänder
  - (Netzwerk)

Die Ein-/Ausgabe Geräte sind über das Bus-System mit Speicher und Hauptprozessor verbunden.

Bus-Systeme

- Ein gemeinsam genutztes Medium
- Die Anzahl gleichzeitig übertragbarer Bits ist die Busbreite.
- Busse können durch Brücken hierarchisch sein.

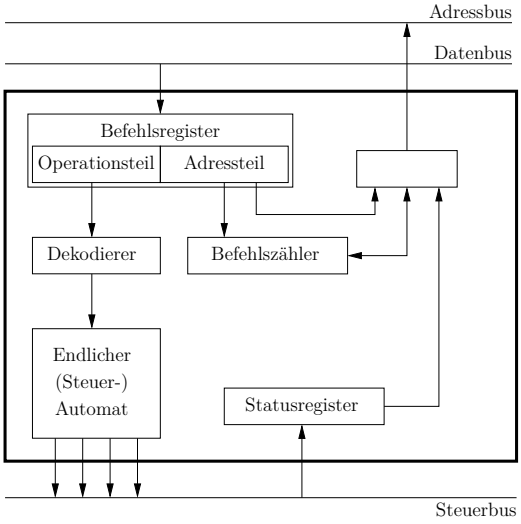


PCI (Peripheral Component Interconnect) Bus.

North Bridge (im Beispiel PCI bridge). Synchronisiert den Datentransfer von und zur CPU. Durch FSB (Front Side Bus) direkt mit der CPU verbunden

South Bridge (im Beispiel ISA bridge). Synchronisiert Datentransfer zwischen peripheren Geräten (Seriell, Audio, USB (Universal Serial Bus), Firewire). Durch PCI-Bus mit North Bridge verbunden.

3.2.1 Steuerwerk



Das Steuerwerk steuert die Arbeitsweise des Rechenwerks durch schrittweise Interpretation der Maschinenbefehle

Der Befehlszähler (*program counter, PC*) enthält die Adresse des nächsten auszuführenden Befehls. Das Steuerwerk verwaltet den Wert des Befehlszählers.

Das Befehlsregister (*instruction register, IR*) enthält den aktuellen Befehl.

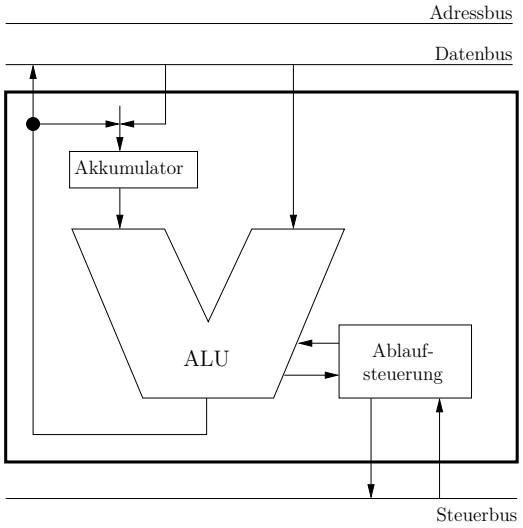
Das Statusregister (*status register, SR*) nimmt Rückmeldungen des Systems auf.

Das Steuerwerk decodiert den Befehl.

- Der Operationsteil (*operation code, opcode*) bestimmt dabei welche Operationen ausgeführt werden sollen.
- Operanden werden durch Angabe von Registern oder Speicheradressen bestimmt.
- Direktoperanden können durch Konstanten angegeben werden.
- Decodierung erfolgt in der Regel durch Mikroprogramme.

Das Steuerwerk erzeugt die nötigen Steuersignale für das Rechenwerk.

3.2.2 Rechenwerk

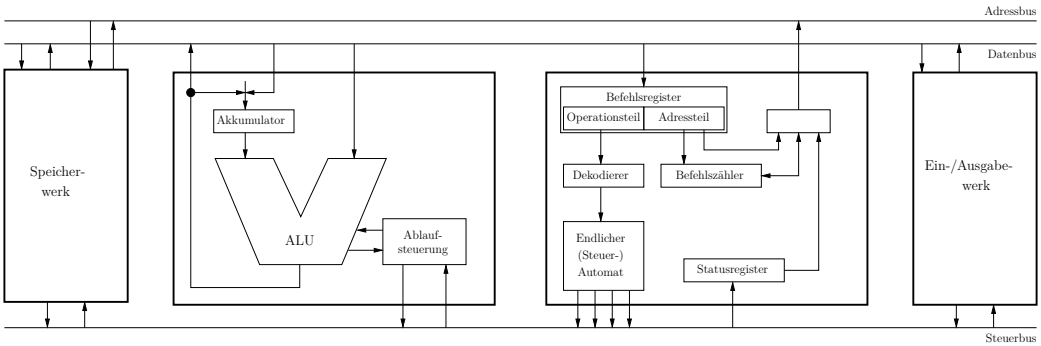


Das Rechenwerk bildet zusammen mit dem Steuerwerk den Hauptprozessor (CPU).

Es besteht aus

- einer (oder mehreren) ALU und Registern,
- arithmetische Operationen (Addition, Subtraktion, ...),
- logische Operationen (UND, ODER, NICHT, ...),
- Verschiebe-Operationen,
- Bitmanipulation (unter Umständen),
- Vergleichs- und Bit-Test-Operationen.

3.2.3 Rechneraufbau



3.3 Befehlszyklus

FETCH	Befehlsholphase
DECODE	Dekodierungsphase
FETCH OPERANDS	Operanden nachladen
EXECUTE	Befehl ausführen
UPDATE PC	Befehlszähler auf den nächsten Befehl zeigen lassen

Die Gemeinsame Arbeitsweise von Steuerwerk und Rechenwerk wird durch den Maschinenbefehlszyklus beschrieben.

Der Befehlszyklus wird von der CPU ständig durchlaufen.

Die Befehle stehen im Speicher.

Das Steuerwerk *weiß* jederzeit, welcher Befehl als nächster auszuführen ist.

Die Adresse (= Nummer der Speicherzelle) des nächsten auszuführenden Befehls steht in einem speziellen Register des Steuerwerks, dem Befehlszähler (PC).

Üblicherweise stehen aufeinanderfolgende Befehle in aufeinanderfolgenden Speicherzellen, der zuerst auszuführende Befehl hat die niedrigste Adresse.

Zu Beginn des Programms wird der Befehlszähler mit der Startadresse des ersten Befehls geladen.

Befehlsholphase.

- Speicherzugriff auf die vom Befehlszähler (PC) angezeigte Adresse.
- Der Befehl wird in das Befehlsregister des Steuerwerks geschrieben.
- Besteht ein Befehl aus mehreren Speicherworten, so setzt sich diese Phase auch aus mehreren Speicherzugriffen zusammen, bis der Befehl vollständig im Befehlsregister steht.
- Das Befehlsregister ist untergliedert in Operationsteil Register (OR) und Adressteil Register (AR).

Dekodierungsphase.

- Der Befehl im OR wird decodiert (Befehlsdecoder) und der Ablaufsteuerung zugeführt.
- Das Decodieren übernimmt ein Mikroprogramm oder ist hart verdrahtet.
- Die Ablaufsteuerung erzeugt die für die Befehlsausführung nötigen Steuersignale.
- Benötigt der Befehl Operanden, so wird deren Adresse aus dem Inhalt des AR ermittelt.

Operanden nachladen.

- Speicherzugriff auf die ermittelte Operandenadresse(n).

Befehl ausführen.

- Die durch den Operationsteil festgelegten Operationen werden ausgeführt.

Befehlszähler auf den nächsten Befehl zeigen lassen.

- Durch Sprungbefehle oder Prozeduraufrufe kann der Inhalt des PCs verändert werden.

4 Nachtrag: Haskell

4.1 Boolesche Logik

Einfache boolesche Operatoren

Anhand der Booleschen Logik und von Operationen auf Bit-Folgen werde weitere Beispiele für die Funktionalität von Haskell betrachtet.

Nachfolgend sind einige einfachen, d.h. unäre und binäre, boolesche Operatoren und deren Wahrheitswertetabellen angegeben.

Hinweis. 0 repräsentiert den Wahrheitswert *false*, 1 den Wahrheitswert *true*.

NOT

A	X
0	1
1	0

AND

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

NAND

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

XOR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

OR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

In `prelude` sind der Typ `Bool` und zu `NOT`, `AND` und `OR` passende Funktionen und Operatoren definiert.

```
ghci> :t not
not :: Bool -> Bool
ghci> :t (&&)
(&&) :: Bool -> Bool -> Bool
ghci> :t (||)
(||) :: Bool -> Bool -> Bool
```

Mehrstellige boolesche Funktionen

Es lassen sich, analog zu den einfachen booleschen Funktionen, auch mehrstellige boolesche Funktionen, d.h. Funktionen in mehreren Variablen, definieren.

$$AND\ (A_1, \ldots, A_n) = \begin{cases} 1 & \text{wenn alle } A_1, \ldots A_n \text{ gleich 1 sind} \\ 0 & \text{sonst} \end{cases}$$

$$OR\ (A_1, \ldots, A_n) = \begin{cases} 0 & \text{wenn alle } A_1, \ldots A_n \text{ gleich 0 sind} \\ 1 & \text{sonst} \end{cases}$$

$$NAND\ (A_1, \ldots, A_n) = \begin{cases} 0 & \text{wenn alle } A_1, \ldots A_n \text{ gleich 1 sind} \\ 1 & \text{sonst} \end{cases}$$

$$NOR\ (A_1, \ldots, A_n) = \begin{cases} 1 & \text{wenn alle } A_1, \ldots A_n \text{ gleich 0 sind} \\ 0 & \text{sonst} \end{cases}$$

Ebenfalls in **`prelude`** definiert sind Funktionen, die zu **`AND`** und **`OR`** in mehreren Variablen passen. Wobei die Haskell-Funktionen **`and`** und **`or`** auf Listen von Booleschen Werten angewendet werden.

Beispiele

```
ghci> and [True, False, False]
False
ghci> and [True, True, True]
True
ghci> or [True, False, False]
True
ghci> or [False, False False]
False
```

4.2 Funktionen auf Listen und Tupeln

Zum Testen von Funktionen und Operatoren ist es nützlich eine Testfunktion zu definieren.

Die Testfunktion soll als Funktionswert eine Zeichenkette (**String**) zurückliefert, die den Funktionswert der zu testenden Funktion, bei Anwendung auf konkrete Argumente, repräsentiert.

Den erste Schritt dazu realisiert die Funktion **show**, die für fast alle Typen definiert ist und als Funktionswert die Zeichenkettenrepräsentation des Arguments als **String** zurückliefert.

```
ghci> show( True )
"True"
ghci> show( and[False, False, False] )
"False"
```

Mehrere Strings kann man, wie Listen, mit dem Operator **(++)** verbinden (*concat*).

Der Zeilenumbruch wird über `"\n"` kodiert.

```
ghci> show( and[False, False, False] ) ++ "\n" ++
      show( and[True, True, True] )
"False\nTrue"
```

Eine formatierte Ausgabe erhält man mit der Funktion **putStrLn**.

```
ghci> putStrLn( show( and[False, False, False] ) ++ "\n" ++
      show( and[True, True, True] ) )
False
True
```

Beispiel

```
-- test_bool.hs
testAND :: String
testAND =
  show( and[False, False, False] ) ++ "\n" ++
  show( and[False, False, True ] ) ++ "\n" ++
  show( and[False, True , False] ) ++ "\n" ++
  show( and[False, True , True ] ) ++ "\n" ++
  show( and[True , False, False] ) ++ "\n" ++
  show( and[True , False, True ] ) ++ "\n" ++
  show( and[True , True , False] ) ++ "\n" ++
  show( and[True , True , True ] )
```

```
ghci> :l test_bool.hs
...
ghci> putStrLn testAND
False
False
False
False
False
False
False
True
```

Soll eine Wahrheitswertetabelle ausgegeben werden, geht das mit Haskell auch kompakter und insbesondere ohne Code-Vervielfachung.

Zuerst wird für eine beliebige Funktion (`[Bool] -> Bool`), die eine Liste von Wahrheitswerten auf genau einen Wahrheitswert abbildet, und eine Liste von Wahrheitswerten (`[Bool]`) eine Tabellenzeile (`String`) erzeugt.

```
table_row :: ([Bool] -> Bool) -> [Bool] -> String
table_row f xs = show xs ++ " : " ++ show (f xs)
```

```
ghci> table_row and [True, False, True]
"[True,False,True] : False"
```

Die ganze Tabelle wird erzeugt, indem eine Liste von Listen von Wahrheitswerten (`[[Bool]]`) rekursiv durchlaufen wird.

In jedem Schritt wird von `head` aus der umschließenden Liste die erste Liste (`[Bool]`) geliefert, diese wird mit `table_row` ausgegeben.

Die umschließenden Liste ohne die erste Liste wird von **tail** geliefert. Mit dieser kleineren Liste wir die Rekursion durchgeführt.

Der Basisfall, für den keine Rekursion mehr nötig ist, tritt ein, wenn die umschließende Liste leer ist (wird getestet mit **null**).

```
tableA :: ([Bool] -> Bool) -> [[Bool]] -> String
tableA f xs
  | null xs   = ""
  | otherwise = table_row f (head xs) ++ "\n" ++
                tableA f (tail xs)
```

Die Funktionen **head**, **tail** und **null** sind nicht die beste Option Listen zu bearbeiten. Vorzuziehen ist eine Lösung mit *pattern matching*.

```
table :: ([Bool] -> Bool) -> [[Bool]] -> String
table f [] = ""
table f (x:xs) = table_row f x ++ "\n" ++ table f xs
```

Das *pattern* **[]** beschreibt die leere Liste.

Das *pattern* **(x:xs)** passt **nur** zu einer **nicht leeren** Liste. Im folgende können **x**, das erste Element der Liste, und **xs**, die - möglicherweise leere - Liste ohne das erste Element, einzeln benutzt werden.

Ähnlich wie mit **where** lassen sich mit **let** Platzhalter für Ausdrücke definieren.

Haskell bietet die Möglichkeit Listen auf sehr viele verschiedene Arten zu definieren, eine wird im Folgenden verwendet.

Die sogenannte *list comprehension* definiert den Inhalt einer Liste analog zur mathematischen Notation von Mengen $\{x \mid x \in M, cond_1(x), cond_2(x) \ldots\}$.

```
ghci> let bool_tri =  [[a,b,c] | a <- [False, True],
                                 b <- [False, True],
                                 c <- [False, True]]
ghci> putStrLn ( table and bool_tri )
[False,False,False] : False
[False,False,True] : False
[False,True,False] : False
[False,True,True] : False
[True,False,False] : False
[True,False,True] : False
[True,True,False] : False
[True,True,True] : True
```

Haskell stellt (in jedem Modul) eine vordeklarierte Funktion **main** bereit, unter der sich, z.B. in einem **do**-Block, eine Folge von Ausdrücken zusammenfassen lässt.

Die Ausdrücke in der Funktion **main** erzeugen Ausgaben (z.B. **putStrLn**), erwarten Eingaben oder haben keinen Wert (z.B. **let**).

```
main = do
  let bool_tri =  [[a,b,c] | a <- [False, True],
                             b <- [False, True],
                             c <- [False, True]]
  putStrLn ( table and bool_tri )
```

```
ghci> main
[False,False,False] : False
[False,False,True] : False
[False,True,False] : False
[False,True,True] : False
[True,False,False] : False
[True,False,True] : False
[True,True,False] : False
[True,True,True] : True
```

Folgen von Bits

Ein Wahrheitswert (*false*/*true*) kann auch als **Bit** (0/1) interpretiert werden.

Mit Hilfe der Booleschen Logik sind dann auch Operationen auf Folgen von Bits (z.B. 8 Bit = 1 **Byte**, 4 Bit = 1 **Nibble**) möglich.

Der Test auf Gleichheit für zwei gleich lange Folgen von Bits kann wie folgt realisiert werden.

$$\begin{aligned} &equals(x_{n-1}, \ldots, x_0, y_{n-1}, \ldots y_0) \\ &= NOR\left((x_{n-1}\ XOR\ y_{n-1}), \ldots, (x_0\ XOR\ y_0)\right) \end{aligned}$$

Nicht definiert in **prelude** ist ein Funktion oder ein Operator für **XOR**.

Man kann selbst einen passenden Operator für **XOR** definieren.

```
(<+>) :: Bool -> Bool -> Bool
(<+>) a b = (a || b) && (not (a && b))
```

Zum Testen des Operators wird eine Testfunktion (**table**), mit Hilfsfunktion (**table_row**), definiert.

```
table_rowA :: (Bool -> Bool -> Bool) -> (Bool, Bool) -> String
table_rowA f xt = show(xt) ++ " : " ++ show(f (fst xt) (snd xt))
```

Hinweis

- Das erste Element eines Paars (2-Tupels) wird von **fst**, das zweite von **snd** geliefert.

Das lässt sich auch mit *pattern matching* umsetzen.

```
table_row :: (Bool -> Bool -> Bool) -> (Bool, Bool) -> String
table_row f (x,y) = show (x,y) ++ " : " ++ show (f x y)
```

Hinweis

- Das *pattern* **(x,y)** steht für ein Paar, die Elemente **x** und **y** können nachfolgend auch einzeln verwendet werden.

Testfunktion mit **main**.

```
table :: (Bool -> Bool -> Bool) -> [(Bool, Bool)] -> String
table f []     = ""
table f (x:xs) = table_row f x ++ "\n" ++ table f xs

-- test XOR operator
main = do
  let bool_duo =  [(a,b) | a <- [False, True],
                           b <- [False, True]]
  putStrLn ( table (<+>) bool_duo )
```

Der Test ist erfolgreich.

```
ghci> main
(False,False) : False
(False,True) : True
(True,False) : True
(True,True) : False
```

Für den Test auf Gleichheit von zwei Bit-Folgen wird noch ein mehrstelliges NOR benötigt.

Man kann die aus der Mathematik bekannte **Komposition** (Hintereinanderausführung) von Funktionen verwenden.

Seien X, Y, Z beliebige Mengen und $f : X \to Y$ und $g : Y \to Z$ Funktionen, dann ist die Funktion $g \circ f : X \to Z$ die Komposition von f und g und es gilt folgendes.

$$\Big(g \circ f\Big)(x) = g\Big(f(x)\Big)$$

In Haskell wird die Komposition mit dem Punkt-Operator (`.`) realisiert.

```
nor :: [Bool] -> Bool
nor = not.or
```

Für eine kompaktere Darstellung wird mit Hilfe des Statements **`type`** ein Synonym **`Nibble`**, für ein 4-Tupel von Wahrheitswerten, eingeführt.

```
type Nibble = (Bool, Bool, Bool, Bool)
```

Jetzt kann eine Funktion zum Test auf Gleichheit von zwei **`Nibble`** definiert werden.

```
equals :: Nibble -> Nibble -> Bool
equals (a3, a2, a1, a0) (b3, b2, b1, b0)
  = nor [a3 <+> b3, a2 <+> b2, a1 <+> b1, a0 <+> b0]
```

Die Funktion kann wie üblich getestet werden.

```
ghci> equals (True, False, False, False)
             (True, False, False, False)
True
ghci> equals (True, False, False, False)
             (False, False, False, False)
False
```

5 Betriebssysteme

5.1 Einführung

5.1.1 Literatur

Andrew S. Tanenbaum, **Moderne Betriebssysteme**, 2te Auflage, Pearson Studium, 2002.

Carsten Vogt, **Betriebssysteme,** Spektrum Akademischer Verlag, 2001.

Abraham Silberschatz, Peter B. Galvin, **Operating System Concepts (5th Edition),** John Wiley and Sons, 1999.

William Stallings, **Betriebssysteme - Prinzipien und Umsetzung (4te Auflage),** Prentice Hall, 2002.

5.1.2 Was ist ein Betriebssystem?

Computersoftware gliedert sich in zwei Gruppen.

- **Systemprogramme** ermöglichen den Betrieb des Computers.
- **Anwenderprogramme** erfüllen die Anforderungen der Anwender.

Das **Betriebssystem** ist das wichtigste Systemprogramm.

Definition eines Betriebssystems nach DIN 44300

- Betriebssystem: Die Programme eines digitalen Rechnersystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.

Man kann bei Rechnersystemen zwei Sichten einnehmen.

Hardwaresicht. Ein Rechnersystem besteht aus einer Menge kooperierender Hardwarekomponenten.

- Prozessor (CPU). Ausführung von Maschinenprogrammen
- Hauptspeicher. *Kurzfristige* Speicherung einer begrenzte Menge von Daten.

- Hintergrundspeicher (Festplatte, Diskette, CD, DVD). *Langfristige* Speicherung größerer Datenmengen.
- Eingabegeräte (Tastatur, Maus).
- Ausgabegeräte (Bildschirm, Drucker).
- Netzwerkkarte. Verbindung an ein Kommunikationsnetz.
- ...

Anwendersicht. Ein Rechnersystem stellt (benutzerfreundliche) Konzepte bereit, mit denen Daten und Informationen verarbeitet werden können.

- Dateisystem. Klar strukturiert mit Dienstprogrammen.
- Programmierumgebung. Schreiben und übersetzen von Programmen.
- Ein- und Ausgabedienste. Zugriff auf Daten, Internet Angebote, etc.
- Systemverwaltung.
- Multi-User-Fähigkeit. Unterstützung mehrere Benutzer.
- ...

Position des Betriebssystems

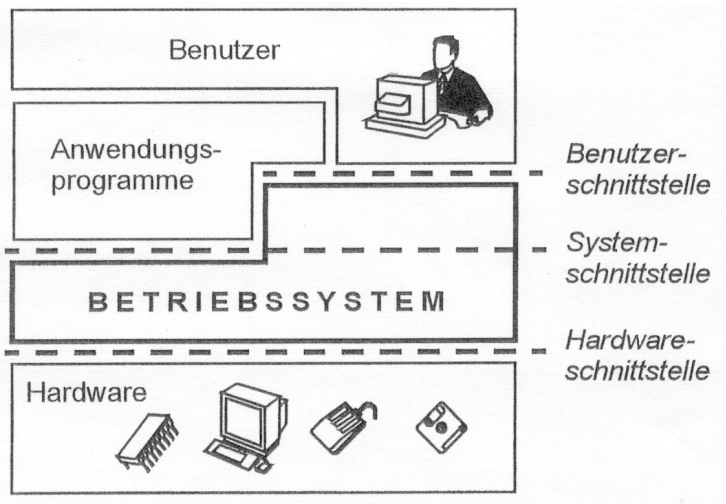


Abbildung 1: Position des Betriebssystems (Quelle: Carsten Vogt, Betriebssysteme, Spektrum Akademischer Verlag, 2001)

Das Betriebssystem (engl. *operating system*) ist die Softwarekomponente, die die abstrakte Anwendersicht auf der Grundlage der realen Hardwaresicht umsetzt.

Das Betriebssystem

- macht die Hardware für den Anwender benutzbar.
- setzt auf der Hardware auf, steuert diese und bietet *nach oben* benutzer- und programmierfreundliche Dienste an.
- enthält interne Programme und Datenstrukturen.

Ressourcen (Betriebsmittel). Die Ressourcen (Betriebsmittel) eines Betriebssystems sind alle Hard- und Softwarekomponenten, die für die Programmausführung relevant sind. z.B. Prozessor, Hauptspeicher, I/O-Geräte, Hintergrundspeicher, etc.

Betriebssystem als Ressourcenverwalter. Ein Betriebssystem bezeichnet alle Programme eines Rechensystems, die die Ausführung der Benutzerprogramme, die Verteilung der Ressourcen auf die Benutzerprogramme und die Aufrechterhaltung der Betriebsart steuern und Überwachen.

5.1.3 Aufgaben eines Betriebssystems

Hauptaufgaben eines Betriebssystems

- Prozessverwaltung
- Speicherverwaltung
- Verwaltung des Dateisystems
- Geräteverwaltung

Prozessverwaltung (Ein Prozess oder auch Task ist ein in Ausführung befindliches Programm)

- Erzeugen und Löschen von Prozessen.
- Prozessorzuteilung (Scheduling).
- Prozesskommunikation.
- Synchronisation nebenläufiger Prozesse, die gemeinsame Daten benutzen.

Speicherverwaltung

- Zuteilung des verfügbaren physikalischen Speichers an Prozesse.
  - Segementierung (= Unterteilung des benutzten Speicheradressraums in einzelne Segmente).
  - ...
- Einbeziehen des Hintergrundspeichers (z.B. Festplatte).
  - Paging (= Bereitstellung von virtuellem Speicher).
  - Swapping (= Ein-/Auslagern von Prozessen).
  - ...

Verwaltung des Dateisystems

- Logische Sicht auf Speichereinheiten (Dateien).
  - Benutzer arbeitet mit Dateinamen. Wie und wo die Dateien gespeichert werden, ist ihm egal.
- Systemaufrufe für Dateioperationen.
  - Erzeugen, Löschen, Öffnen, Lesen, Schreiben, Kopieren, etc.
- Strukturierung mittels Verzeichnissen (engl. directories).
- Schutz von Dateien und Verzeichnissen vor unberechtigtem Zugriff

Geräteverwaltung

- Auswahl und Bereitstellung von I/O-Geräten.
- Anpassung an physikalische Eigenschaften der Geräte.
- Überwachung der Datenübertragung.

Weitere wichtige Konzepte

Fehlertoleranz.

- Graceful Degradation. Beim Ausfall einzelner Komponenten läuft das System mit vollem Funktionsumfang mit verminderter Leistung weiter.
- Fehlertoleranz wird durch Redundanz erkauft.

Realzeitbetrieb.

- Betriebssystem muss den Realzeit-kritischen Prozessen die Betriebsmittel so zuteilen, dass die angeforderten Zeitanforderungen eingehalten werden.
- Für zeitkritische Systeme. Messsysteme, Anlagensteuerungen, etc.

Benutzeroberflächen.

- Betriebssystem kann eine Benutzerschnittstelle für die eigene Bedienung enthalten.
- Betriebssystem kann Funktionen bereitstellen, mit denen aus Anwendungsprogrammen heraus auf die Benutzerschnittstelle zugegriffen werden kann.

5.1.4 Betriebsarten von Betriebssystemen

Die **Betriebsart** ist ein wichtiges Kennzeichen für die Leistungsfähigkeit und den Anwendungsbereich eines Betriebssystems.

Rechensysteme habe im Allgemeinem mehrere Aufträge gleichzeitig zu bearbeiten. Die Betriebsart bestimmt (im wesentlichen) den zeitlichen Ablauf der Ausführung dieser Aufträge.

Die Spanne möglicher Abläufe reicht von **streng sequentiell** (= hintereinander) bis **voll nebenläufig** (= gleichzeitig).

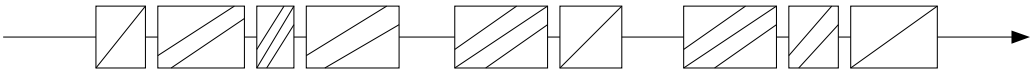
Einzelbenutzerbetrieb

> **Ein Benutzer** belegt das **gesamte Rechensystem** und erteilt Aufträge, die **streng sequentiell** abgearbeitet werden.

Batchbetrieb

Ein Auftrag (engl. *job*) wird durch eine Reihe von **Steuerkommandos an das Betriebssystem**, formuliert in einer *job control language*, eingeleitet und abgeschlossen. Zwischen den Steuerkommandos befindet sich das eigentliche Programm.

Mehrprogrammbetrieb



 = Prozessor arbeitet

- Mehrer Aufträge werden **nebenläufig** (engl. *concurrent*) bearbeit, wobei der Prozessor (im schnellen Wechsel) umgeschaltet wird.
- Flexible Prozessorzuteilung. Der Prozessor kann auch während des Abarbeiten eines Auftrag zu einem anderen wechseln, weil
  - der gerade ausgeführte Auftrag wartet.
  - ein dringenderer Auftrag bearbeitet werden soll.
  - gleichberechtigte Aufträge gleichmäßig (fair) bearbeiten werden sollen.
- Ermöglicht **Timesharing**. Mehrere Benutzer können gleichzeitig mit dem Rechensystem arbeiten.

5.2 Prozessverwaltung

Auf modernen Rechensystemen können mehrere Programme nebenläufig bearbeitet werden.

- Benutzerprogramme, Lesen/Schreiben von/auf Platten, Drucken von Dateien, etc.
- Ermöglicht bessere Nutzung der Ressourcen.

Ein **Prozess** (engl. *process*, ***task***) ist die Abstraktion eines laufenden Programms.

Ein Prozess benötigt **Betriebsmittel** (Prozessorzeit, Speicher, Dateien, etc.) und ist selbst ein Betriebsmittel.

Ein Prozessor führt in jeder Zeiteinheit maximal einen Prozess aus. Laufen mehrere Prozesse, dann finden Prozesswechsel statt.

Prozesse werden vom Betriebssystem verwaltet.

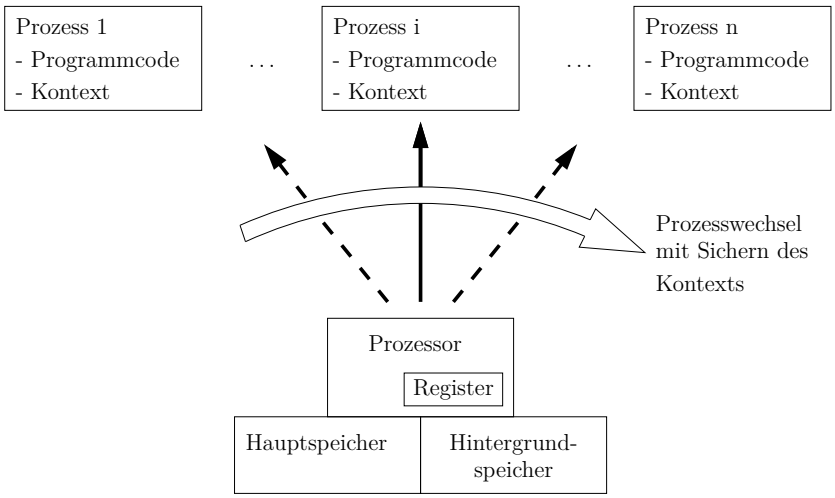
Eine Aufgabe des Betriebssystems besteht darin, den verschiedenen Prozessen Prozessorzeit zuzuteilen (engl. ***scheduling)***.

Um das Scheduling zu ermöglichen, besteht ein Prozess aus dem **auszuführenden Programmcode** (inkl. Daten) und einem **Kontext**.

Zum **Kontext** eines Prozesses gehören

- die **Registerinhalte des Prozessors**,
- dem Prozess zugeordnete **Bereiche des direkt zugreifbaren Speichers**,
- durch den Prozess **geöffnete Dateien**,
- dem Prozess **zugeordnete Peripheriegeräte**,
- **Verwaltungsinformationen** über den Prozess.

Prozesswechsel



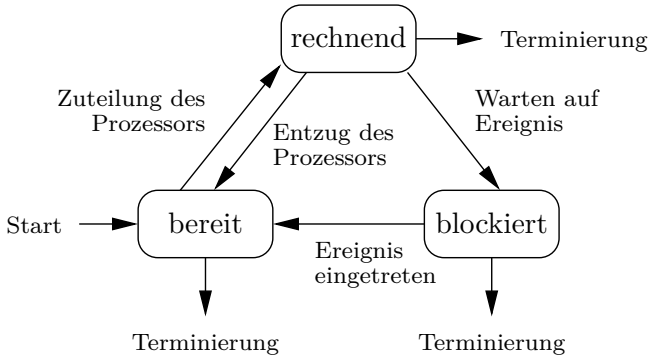
Ein Prozess befindet sich zu jedem Zeitpunkt in einem bestimmten **Zustand**. Es finden **dynamisch Zustandsübergänge**, also Veränderungen des Prozesszustands statt.

Prozesszustände

- **rechnend** (*running*). Prozess wird momentan ausgeführt.
- **bereit** (*ready*). Prozess ist ausführbar und wartet auf die Zuteilung des Prozessors.
- **blockiert** (*blocked*). Prozess kann momentan nicht ausgeführt werden und wartet auf das Eintreten eines Ereignisses (z.B. Nachricht von einem E/A-Prozess).

I.d.R. existieren noch weitere Zustände, z.B. *new* (Prozess wird gerade erzeugt) oder *exit* (Prozess wird gerade beendet) sowie evtl. weitere Verfeinerungen der obigen Zustände.

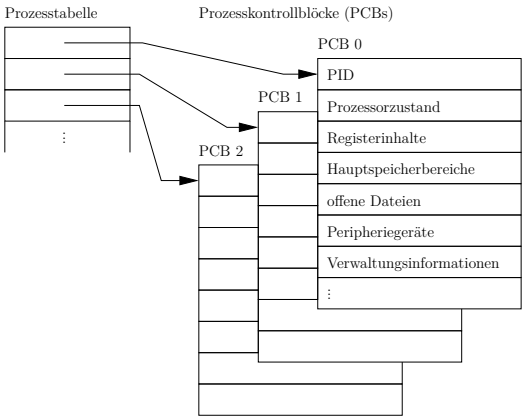
Zustandsübergänge



Das Betriebssystem verwaltet Prozesse mit Hilfe einer **Prozesstabelle**, die **Prozesskontrollblöcke** (engl. *process control blocks*, **PCBs**), bzw. Verweise auf PCBs, für alle existierenden Prozesse enthält.

Ein PCB enthält

- den Kontext des Prozesses,
- für das Scheduling benötigte Informationen,
- weitere Verwaltungsinformationen.



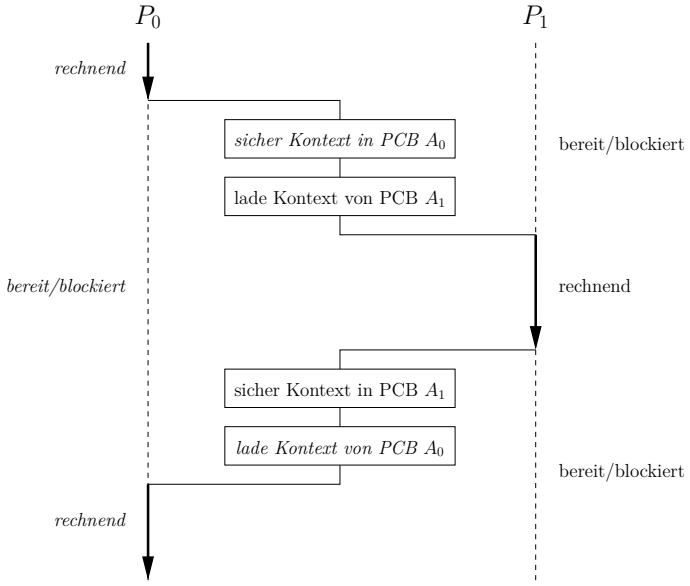
Prozesswechsel (Dispatching)

Prozesswechsel

- Der aktuelle Kontext eines Prozesses P_0 wird im einem PCB gesichert.
- Der Kontext eines anderen Prozesses P_1 wird aus einem PCB geladen.
- Der Zustand beider PCBs muss aktualisiert werden.
- **Prozesswechsel = Kontextwechsel**

Prozesswechsel sind relativ teuer (benötigen viel Zeit).

Prozesswechsel wird häufig von spezieller Hardware unterstützt.



5.3 Scheduling

Scheduling ist die Zuteilung von Prozessorzeit an die Prozesse.

Komponenten des Scheduling.

- **Prozesswechselkosten.** Prozesswechsel sind relativ teuer, weil der Kontextes der Prozesse gesichert/gelanden werden muss.
- **Warteschlangenmodell.** Wartende Prozesse werden in internen Warteschlangen gehalten, die Auswahlstrategie der Warteschlangen haben wesentlichen Einfluss auf das Systemverhalten.
- **Scheduling-Verfahren.**

Fragen

- Wann erfolgt der Kontextwechsel?
- Nach welchen Kriterien wird der Prozess ausgewählt, der als nächstes bearbeitet wird?

Alle Systeme

- *Fairness.* Jeder Prozess bekommt Rechenzeit der CPU.
- *Policy Enforcement.* Durchsetzung der Verfahrensweisen, keine Ausnahmen.
- *Balance.* Alle Teile des Systems sind (gleichmäßig) ausgelastet.
- *Data Protection.* Keine Daten oder Prozesse gehen verloren.
- *Scalability.* Mittlere Leistung wird bei wachsender Last (Anzahl von Prozessen) beibehalten. D.h. es gibt keine Schwelle, ab der das Scheduling nur noch sehr langsam oder gar nicht mehr funktioniert.

Batch-Systeme (Stapelverarbeitungssysteme)

- *Throughput* (Durchsatz). Maximiere nach Prozessen pro Zeiteinheit.
- *Turnaround Time.* Minimiere die Zeit vom Start bis zur Beendigung eines Prozesses.
- *Processor Load.* Belege die CPU konstant mit Jobs.

5.3.1 Scheduling in Batch-Systemen

First Come First Served (FCFS)

Prinzip

- Prozesse bekommen den Prozessor entsprechend ihrer Ankunftsreihenfolge zugeteilt.
- Keine Abhängigkeiten zwischen den Prozessen.
- Laufende Prozesse werden nicht unterbrochen.

Eigenschaften

- Fair (jeder Prozess kommt dran).
- Einfache Implementierung.

Bemerkungen

- Die mittlere Wartezeit kann unter Umständen sehr hoch werden.

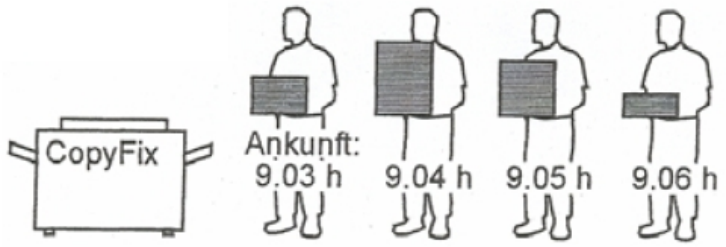


Abbildung 2: First Come First Served (Quelle: Carsten Vogt, Betriebssysteme, Spektrum Akademischer Verlag, 2001)

Shortest Job First (SJF)

Prinzip

- Es wird jeweils der Prozess mit der kürzesten Rechenzeit als nächstes gerechnet.
- Keine Abhängigkeiten zwischen den Prozessen.
- Laufende Prozesse werden nicht unterbrochen.

Eigenschaften

- Nicht fair (kurze Prozesse können lange Prozesse überholen).

Problem

- Wie wird die Rechenzeit eines Prozesses ermittelt?

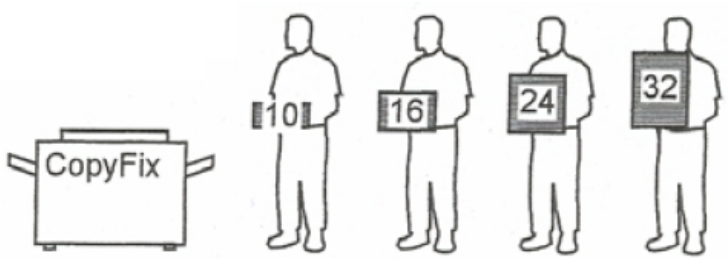


Abbildung 3: Shortest Job First (Quelle: Carsten Vogt, Betriebssysteme, Spektrum Akademischer Verlag, 2001)

5.3.2 Scheduling und Mehrprogrammbetrieb

Auf Sytemen im Mehrprogrammbetrieb sind normalerweise immer mehrere Prozesse zu einem Zeitpunkt rechenbereit.

Verfahren

- Man zerlegt die Rechenzeit in Zeitscheiben (gleicher oder variabler Länge) und ordnet diese nach bestimmten Kriterien (z.B. Fairness, Prioritäten, Rechenzeit, etc.) den rechenbereiten Prozessen zu.
- Hat ein Prozess seine Zeitscheibe verbraucht wird er unterbrochen und muss auf eine neue Zuteilung warten.

Round-Robin-Scheduling

Prinzip

- Die Rechenzeit wird in gleichlange Zeitscheiben/-schlitze (*time slices*) aufgeteilt.
- Prozesse werden in einer Warteschlange eingereiht und in FIFO-Ordnung (*first in, first out*) ausgewählt.
- Ein rechnender Prozess wird nach Ablauf einer Zeitscheibe unterbrochen und wieder hinten in die Warteschlange eingestellt (Rundlauf, *round robin*).

Bemerkung

Wird ein Prozess blockiert oder beendet er sich bevor dessen Zeitscheibe komplett aufgebraucht ist, wird sofort der nächste Prozess ausgewählt und kann eine Zeitscheiben lang rechnen.

Eigenschaften

- Die Prozessorzeit wird nahezu gleichmässig auf die vorhandenen Prozesse aufgeteilt.

Abbildung 4: Round-Robin (Quelle: Carsten Vogt, Betriebssysteme, Spektrum Akademischer Verlag, 2001)

Ankunfts-/Rechenzeit

Die **Ankunftszeit** eines Prozesses ist der Zeitpunkt ab dem der Prozess vom Scheduling berücksichtigt wird. Der Prozess ist rechenbereit und wenn zu diesem Zeitpunkt der Prozessor nicht belegt ist, bekommt der Prozess sofort Rechenzeit zugeteilt.

Die **Rechenzeit** eines Prozesses ist die Anzahl an Zeiteinheiten, für die der Prozess Rechenzeit zugeteilt bekommt muss, um vollständig abzulaufen, d.h. sich zu beenden.

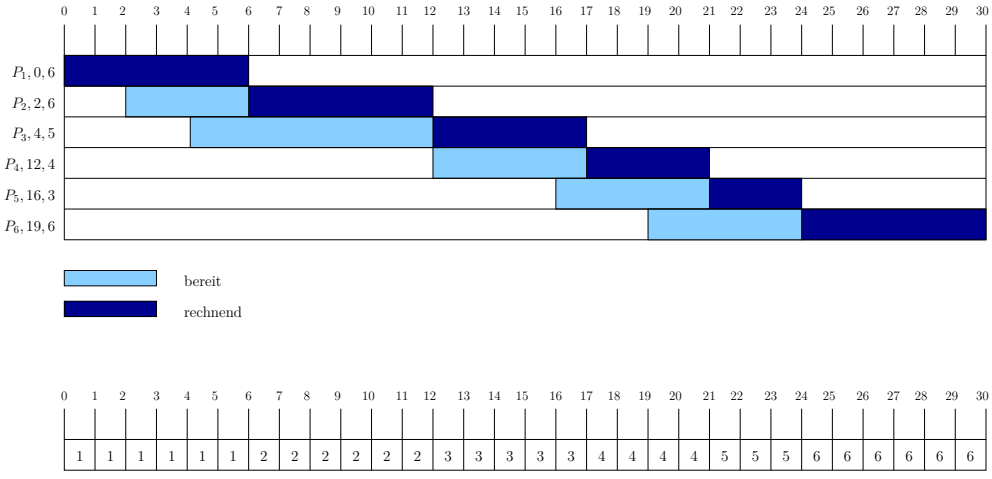
Beispiel

Gegeben seien die Prozesse P_1 bis P_6 mit folgenden *Ankunftszeiten* a_i und *Rechenzeiten* t_i.

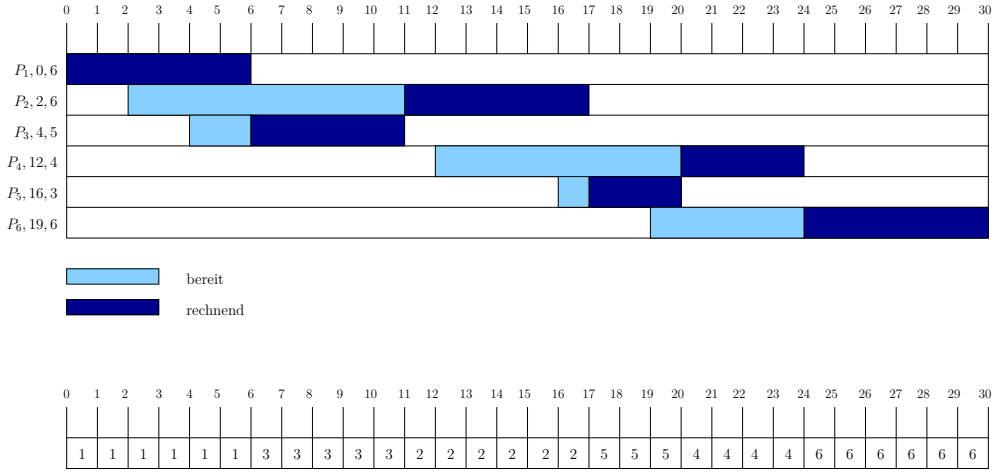
Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit a_i	0	2	4	12	16	19
Rechenzeit t_i	6	6	5	4	3	6

Beispiel, FCFS

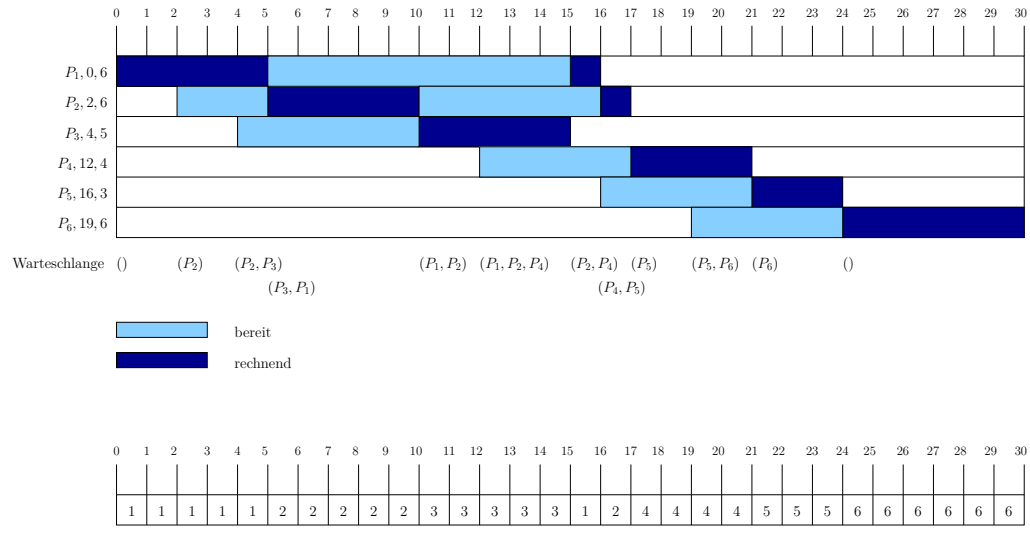
Darstellung des Schedules als *Gantt Chart* (nach Henry L. Gantt 1861-1919) oder *Balkenplan*.



Beispiel, SJF



Beispiel, Round-Robin (Zeitscheibe 5)



5.4 Haskell

5.4.1 Typen

Ein Synonym (*type synonym*) ist ein neuer Name für einen existierenden Typ, man definiert ein Synonym mit **`type`**.

Ausdrücke verschiedener Synonyme desselben Typs sind kompatibel.

Beispiel

Strings bilden in Haskell keinen eigenen Datentypen, sondern sind ein Synonym für Listen von Zeichen.

```
type String = [Char]
```

Durch die Verwendung von Synonymen kann man die Lesbarkeit von Programmen verbessern, z.B. können komplizierte Typen abgekürzt oder sprechende Namen verwendet werden.

Beispiel

Synonyme **Nibble** und **Byte**.

```
type Nibble = (Bool, Bool, Bool, Bool)
type Byte = (Nibble,Nibble)

lastBitNibble :: Nibble -> Bool
lastBitNibble (a, b, c, d) = d

lastBitByte :: Byte -> Bool
lastBitByte (a, b) = lastBitNibble b
```

Test in **ghci**.

```
> lastBitNibble (True, True, True, False)
False
> lastBitByte ((True, True, True, False), (False, True, True, True))
True
```

Neue Typen können mit **data** definiert werden.

Beispiel

```
data Signal = X | O
```

Damit wird der Typ **Signal** definiert, der genau die zwei Werte **X** und **O** hat.

X und **O** sind (parameterlose) Konstruktoren, die jeweiligen Ausdrücke erzeugen einen Wert des Typs **Signal**.

Beispiel

```
> let z = X
> :t z
z :: Signal
```

Hinweis

Im laufenden GHCi kann man mit den Kommandos

```
:info [name]
:i [name]
```

die verfügbaren Informationen über den vergebenen Namen *name* bekommen, insbesondere auch, ob der Name *name* bereits vergeben ist.

Selbstdefinierten Typen fehlen einige Eigenschaften, die die in **Prelude** definierten Typen mitbringen.

Z.B. fehlt die Funktion **show**, die für jeden Wert der Typs eine Zeichenkettenrepräsentation (**String**) zurückliefert oder der Test auf Gleichheit.

Beispiel

```
> let z = X
> :t z
z :: Signal
> show z
No instance for (Show Signal) arising from a use of 'show'
...
> X == 0
No instance for (Eq Signal) arising from a use of '=='
...
```

Man kann (selbstdefinierten) Typen Eigenschaften verleihen, indem man den Typ einer oder mehrerer Typklasse (*typeclasses*) zuordnet.

5.4.2 Typklassen

Die Deklaration einer Typklasse beginnt mit dem Schlüsselwort **`class`**, gefolgt vom Namen der Typklasse und einem Platzhalter für einen Typ (in den folgenden Beispielen **`a`**), der Instanz der Typklasse werden soll.

Dem Schlüsselwort **`where`** folgen die Deklarationen der Funktionen, die Teil der Typklasse sind. In den Deklarationen kann der Platzhalter für den Typ verwendet werden.

Weiterhin können Definitionen der deklarierten Funktionen enthalten sein.

Beispiel

Vereinfachte Versionen der Typklassen **`Show`** und **`Eq`** (*equality*) aus **`Prelude`**.

```
class Show a where
    show :: a -> String
```

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

Ein neuer Typ wird Mitglied eine Typklasse, indem eine Instanz der Typklasse über diesem Typ gebildet wird.

Dazu dient das Schlüsselwort **`instance`** gefolgt von der Typklasse und dem Typ.

Dem Schlüsselwort **where** folgen Definitionen von Funktion, die in der Typklasse deklariert wurden.

Zur Bildung einer Instanz müssen alle in der Typklasse deklarierten Funktionen gültig Definitionen haben. Welche Funktionen dazu definiert werden müssen, hängt von den in der Typklasse bereits angegebenen Definitionen ab.

Die Bildung einer Instanz führt dazu, dass die Funktionen der Typklasse, die mit dem Typ-Platzhalter deklariert wurden, jetzt für den Typ, über dem die Instanz gebildet wurde, verfügbar sind.

In Haskell führt die Bildung einer Instanz zum Überladen der Funktionen (*overloading*), die in der Typklasse deklarierten sind. D.h. von einer Funktion stehen verschiedene Definition zur Verfügung und beim Aufruf der Funktion wird anhand der Typen der Argumente entschieden, welche Definition der Funktion Anwendung findet.

Polymorphismus (Vielgestaltigkeit) nennt man das Konzept, das durch den Kontext bestimmt wird, welche Definition eines Sprachkonstrukts, z.B. einer Funktion, verwendet wird.

Um Mitglied der Typklasse **Show** zu werden, muss eine Instanz von **Show** gebildet werden. Dabei ist die Definiton der Funktion **show** ausreichend.

Beispiel

```
instance Show Signal where
  show X = "X"
  show O = "O"
```

Jetzt gibt es eine Funktion **show :: Signal -> String**, die auf Argumente vom Typ **Signal** angewendet wird.

```
> let z = X
> :t z
z :: Signal
> show z
"X"
> print z
X
```

Zum Bilden einer Instanz der Typklasse **Eq** ist die Definition des Operators **(==)** ausreichend, denn die Definition von **(/=)**, basierend auf **(==)**, ist bereits in **Eq** enthalten.

Beispiel

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

```
instance Eq Signal where
  O == O    = True
  O == X    = False
  X == O    = False
  X == X    = True
```

Durch das Bilden einer Instanz von **Eq** über **Signal** sind die Operatoren **(==)** und **(/=)** für Werte vom Typ **Signal** definiert.

```
> X == O
False
> O /= O
False
```

5.4.3 Eingeschränkte Typ-Parameter

Bei der Verwendung von Typ-Parametern, z.B. bei der Definition von Funktionen oder Typklassen, kann festgelegt werden, dass nur Typen, die Mitglieder einer oder mehrerer Typklassen sind, als Agumente für diese Parameter benutzt werden können, dann spricht man von eingeschränkte Typ-Parametern.

Bei der Vereinbarung des eingeschränkte Parameters wird diesem die Typklasse, auf die er eigeschränkt wird, vorangestellt (kann in runde Klammern eingeschlossen werden). Wird ein Parameter mehrfach eingeschränkt oder werden mehrere eingeschränkt Parameter vereinbart, werden diese durch Kommata getrennt und in runde Klammern eingeschlossen.

Der Vereinbarung von eingeschränkten Parametern folgt **=>** und die Verwendung der Typ-Parameter.

Beispiele

```
class (Eq a) => X a where
  foo :: a -> a -> Bool

f :: (Eq a, Show a) => a -> String

g :: (Eq a, Show b) => a -> b -> String
```

Beispiel

Die Funktion **contains** ermittelt, ob ein Wert vom Typ **a** in einer Liste vom gleichen Typ **[a]** enthalten ist, mit der Einschränkung, dass der Typ **a** Mitglied der Typklasse **Eq** sein muss.

```
contains :: (Eq a) => a -> [a] -> Bool
contains _ [] = False
contains z (x:xs)
  | z == x    = True
  | otherwise = contains z xs
```

Hinweis. Vergleiche Funktion **elem** aus **Prelude**.

In der Definition der Funktion **contains** sind *pattern matching* und *guarded equations* kombiniert.

- **_** ist eine *wildcard*, zu diesem Pattern passt jeder Wert, des zugehörigen Definitionsbereichs. Dieses Pattern wird verwendet, wenn der konkrete Wert nicht benötigt wird.
- **[]** ist die leere Liste.
- **z** ist ein Pattern, zu dem ebenfalls jeder Wert des zugehörigen Definitionsbereichs passt, der konkrete Wert wird an **z** gebunden und kann nachfolgend verwendet werden.
- **(x:xs)** passt zu jeder nicht leeren Liste. An **x** wird das erste Element (*head*) und an **xs** die Liste der nachfolgenden Elemente (*tail*) gebunden.

Bei *pattern matching* und *guarded equations* gilt *fall through*, d.h. die *pattern*/die *guards* werden von oben nach unten abgearbeitet, bis ein passendes/passender gefunden ist. Passt das aktuelle *pattern*/der aktuelle *guard* nicht, wird zum nächsten weitergegangen.

Bei *guarded equations* geht *fall through* noch weiter. Passt keiner der *guards*, wird das zu den *guarded equations* gehörige *pattern* als nicht passend behandelt, d.h. es wird mit dem nächsten *pattern* fortgefahren (falls vorhanden).

Beispiel

```
contains :: (Eq a) => a -> [a] -> Bool
contains _ [] = False
contains z (x:xs)
  | z == x    = True
contains z (x:xs) = contains z xs
```

Beispiel

Typklasse **Ord**

```
class (Eq a) => Ord a where
  compare              :: a -> a -> Ordering
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min             :: a -> a -> a

  -- Minimal complete definition (<=) or compare
  compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT

  x <= y = compare x y /= GT
  x <  y = compare x y == LT
  x >= y = compare x y /= LT
  x >  y = compare x y == GT

  max x y
    | x >= y    = x
    | otherwise = y
  min x y
    | x <  y    = x
    | otherwise = y
```

```
data Ordering = LT | EQ | GT
```

Damit ein Typ einen Instanz der Typklasse **Ord** werden kann, muss dieser Typ bereits Instanz der Typklasse **Eq** sein.

Dann reicht die Definition des Operators **(<=)**, um die Funktion **compare** vollständig zu definieren, denn eine Definition des Operator **(==)** ist bereits vorhanden.

Mit der Definition von **compare** sind alle anderen Operatoren und Funktion der Typklasse **Ord** definiert.

Hinweis

Alternativ können alle anderen Operatoren und Funktionen auch durch die Angabe einer Definition für **compare** vollständig definiert werden.

5.4.4 Record Syntax

Wenn der Konstruktor eines Datentyps mehrere Argumente hat, ist es meistens ohne Kommentare nicht ersichtlich, was die einzelnen Komponenten darstellen.

Beispiel

```
data ProzessA = ProzessA
                  String  -- pid
                  Int     -- arrival
                  Int     -- computing
                  deriving (Show)
```

Weiterhin müssen neue Funktionen definiert werden, um Zugriff auf die einzelnen Komponenten zu bekommen.

Bemerkung

Durch **deriving (Show)** erbt ein Datentyp die Standarddarstellung, die für einen Wert des Datentyps eine **String**-Darstellung des Ausdrucks zurückliefert, mit dem dieser Wert erzeugt werden kann.

```
> show (ProzessA "P1" 5 10)
"ProzessA \"P1\" 5 10"
```

Record Syntax erlaubt es, die Argumente eines Konstruktors zu benennen. Benannte Argumente werden als Felder bezeichnet.

Wenn Record Syntax benutzt wird, werden Funktionen zum Zugriff auf die Felder (*accessor functions*) automatisch erzeugt.

Beispiel

```
data Prozess = Prozess { pid       :: String
                       ,arrival   :: Int
                       ,computing :: Int } deriving (Show)
```

```
> let p = Prozess { pid = "P1", arrival = 5, computing = 10 }
> :t pid
pid :: Prozess -> String
> pid p
"P1"
> arrival p
5
> computing p
10
> print p
Prozess {pid = "P1", arrival = 5, computing = 10}
```

Pattern Matching ist mit Record Syntax ebenfalls möglich, dabei ist die Reihenfolge der Felder beliebig und es müssen nicht alle Felder berücksichtigt werden.

Beispiel

```
arrivedBefore :: Int -> Prozess -> Bool
arrivedBefore t Prozess { arrival = a } = a < t
```

```
> let p = Prozess { pid = "P1", arrival = 5, computing = 10 }
> arrivedBefore 2 p
False
> arrivedBefore 6 p
True
```

5.5 Prozess-Synchronisation

Nebenläufig ablaufende Prozesse (Mehrprogrammbetrieb) haben häufig **gemeinsame Ressourcen**.

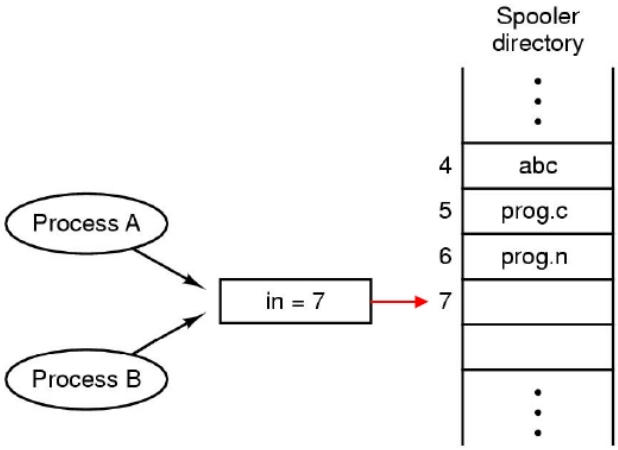
- **Geräte.** Drucker, Platten, usw.
- **Daten.** Dateien, Shared Memory, usw.

Zugriffe auf gemeinsame Ressourcen müssen geordnet erfolgen. Um zu vermeiden, dass die Ergebnisse abhängig sind von der **Reihenfolge** der Abarbeitungsschritte der einzelnen Prozesse (**Race Condition**).

Beispiel, Race Condition

Zwei Prozesse A und B schreiben Druckaufträge in einen Druckerspooler.

Die Prozesse verwenden hierzu die Kontrollvariable **`in`** (nächster freier Slot im Spoolerdirectory) des Druckerspoolers.




```
1 // main process
2
3 start_concurrent(A, B);
```

```
1 // A
2
3 in = spooler->in;
4 spool(spooler, in, jobA);
5 in = in + 1;
6 spooler.in = in;
```

```
1 // B
2
3 in = spooler->in;
4 spool(spooler, in, jobB);
5 in = in + 1;
6 spooler->in = in;
```

Prozess A

- Liest die Variable `in`.
- Schreibt den Auftrag `jobA` in den durch `in` angegebenen Slot (= 7) des Spoolers.
- Berechnet den Wert (= 8), mit dem `in` aktualisiert werden soll.
- Wird vom Scheduler unterbrochen und von Prozess B aus dem Prozessor verdrängt.

Prozess B

- Liest Variable `in`.
- Schreibt den Auftrag `jobB` den durch `in` angegebenen Slot (= 7) des Spoolers überschriebt damit den Auftrag `jobA`.
- Aktualisiert Variable `in` (neuer Wert = 8) und terminiert.

Prozess A

- Nimmt die Bearbeitung wieder auf.
- Aktualisiert Variable `in` (neuer Wert = 8). In der (falschen) Annahme das niemand zwischenzeitlich auf den Spooler zugegriffen hat und terminiert.

Der Auftrag von Prozess A wird nie bearbeitet.

Problem

Ein Prozess befindet sich in seinem **kritischen Abschnitt**, wenn er auf gemeinsame Ressourcen zugreift.

Vermeidung von Race Conditions.

- **Wechselseitiger Ausschluss** (*mutual exclusion*). Keine zwei Prozesse dürfen sich gleichzeitig in ihren kritischen Abschnitten befinden.
- Kein Prozess, der außerhalb seines kritischen Abschnitts läuft, darf andere Prozesse blockieren.
- Es dürfen keine Annahmen über Hardware (z.B. Geschwindigkeit und Anzahl der CPUs) und Betriebssystem (z.B. Scheduling-Algorithmus) gemacht werden.

- Kein Prozess sollte ewig darauf warten müssen, in seinen kritischen Abschnitt einzutreten.

5.5.1 Mutex

Ein **Mutex** (von *mutual exclusion*) kann zur Synchronisation von nebenläufigen Prozessen benutzt werden.

Ein Mutex **ist** eine neue (geschützte) Variablenart auf der nur **unteilbaren (atomaren) Operationen** ausgeführt werden können.

- Datenstruktur

```
mutex {
  boolean          free;
  prozess_queue    queue;
}
```

- Deklaration und Initialisierung

```
1 mutex m;
2 mutex m = true;
3 mutex m = false;
```

  Der Mutex `m` wird deklariert und wie folgt initialisiert

  - 1: `m.free=true` 2: `m.free=true` 3: `m.free=false`
  - `m.queue` ist eine (leere) Datenstruktur, die Prozesse (Verweise auf Prozesskontrollblöcke) aufnehmen kann.

Operationen auf einem Mutex.

- **down** (wait, P)

```
down(m) {
  if (m.free)
    m.free = false;
  else {
    block(this_process);
    insert(m.queue, this_process);
  }
}
```

- **up** (signal, V)

```
up(m) {
  if (is_empty(m.queue))
    m.free = true;
  else {
    process = remove(m.queue);
    wake_up(process);
  }
}
```

Beispiel, ohne Race Condition

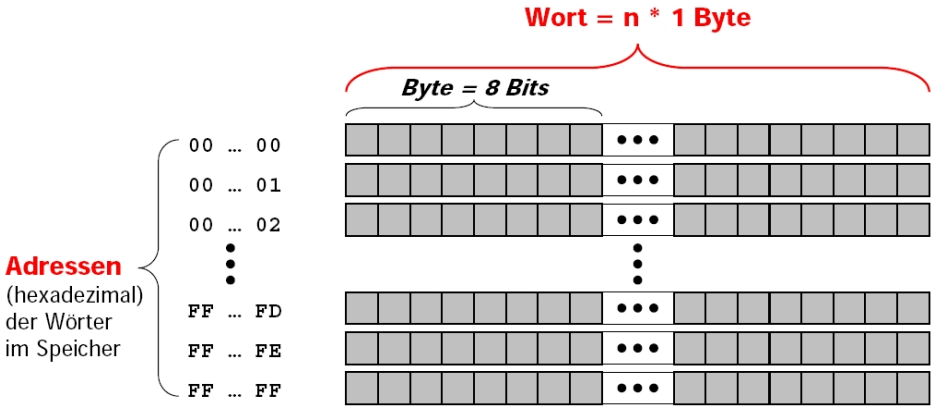
```
1 // main process
2 global mutex m;
3 start_concurrent(A, B);
```

```
1 // A
2 down(m);
3 in = spooler->in;
4 spool(spooler, in, jobA);
5 in = in + 1;
6 spooler->in = in;
7 up(m);
```

```
1 // B
2 down(m);
3 in = spooler->in;
4 spool(spooler, in, jobB);
5 in = in + 1;
6 spooler->in = in;
7 up(m);
```

5.6 Speicherverwaltung

Speicherorganisation



Die einfachste Speicherverwaltungsstrategie ist den Speicher zwischen Prozessen und Betriebssystem aufzuteilen.

Dann wird entweder einem Prozess der gesamten, für Prozesse reservierten, Speicher zur Verfügung gestellt oder der Speicher wird unter mehreren Prozessen aufgeteilt.

Dabei wird jeder Prozess immer **vollständig** im Hauptspeicher gehalten.

Adressraum

Wird der Speicher zwischen Betriebssystem und einem oder mehreren Prozessen aufgeteilt, ergibt sich ein wesentliches Problem.

Prozesse sprechen grundsätzlich direkt physikalische Adressen an.

- **Relokation.** Ein Programm enthält absolute Adressen, diese müssen relativ zur Lage des Prozesses im Speicher umgesetzt werden.
- **Schutz.** Jeder Prozess soll nur die Adressen des Speicherbereichs ansprechen, der im zugeteilt ist.

Diese Anforderungen werden durch die Einführung des **Adressraums** (*adress space*), einer naheliegenden Speicherabstraktion, erfüllt.

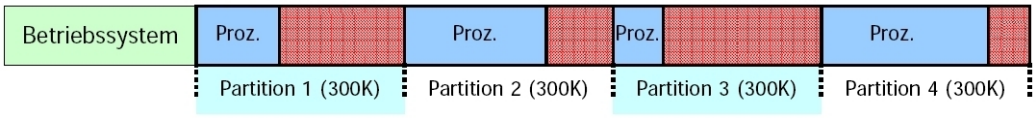
Ein Adressraum ist eine Menge von Adressen, die ein Prozess zur Adressierung des Speichers nutzen kann.

Physikalisch ist der Adressraum, im einfachsten Fall, ein zusammenhängender Speicherbereich (Partition) fester Größe.

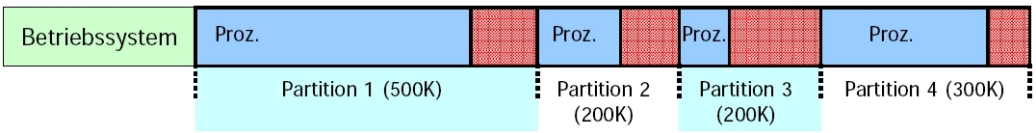
Aus Sicht des Prozesses ist der Adressraum der ganze zur Verfügung stehende Speicher, der in der Regel mit der Adresse 0 beginnt. Die Endadresse ist abhängig von der Größe des zugeteilten Speicherbereichs.

Partition

Feste Partition/Adressräume gleicher Größe.



Feste Partition/Adressräume unterschiedlicher Größen.



Dynamische Relokation

Der Adressraum jedes Prozeses wird auf einen feste Partition des Speichers abgebildet.

Der Prozessor hat zwei zusätzliche Register, das **Basisregister** (`base`) und das **Limitregister** (`limit`). Diese Register gehören zum Kontext des Prozesses.

Beim Erzeugen eines Prozesses wird die physikalische Anfangsadresse der dem Prozess zugeteilten Partition in das Basisregister geladen und die Größe der Partition wird in das Limitregister geladen.

Wenn ein Prozess Speicher referenziert, um eine Befehl zu holen, einen Datenwort zu lesen oder zu schreiben, etc. benutzt der Prozessor automatisch Limit- und Basisregister.

- **Schutz**. Für jede Adresse prüft der Prozessor ob der Wert kleiner oder gleich dem Wert im Limitregister ist, wenn nicht wird der Zugriff verweigert.

- **Relokation**. Zu jeder Adresse addiert der Prozessor automatisch den Wert im Basisregister und schreibt die berechnete Adresse auf den Adressbus.

Bemerkung

Dynamische Relokation wurde noch im Intel 8088 eingesetzt.

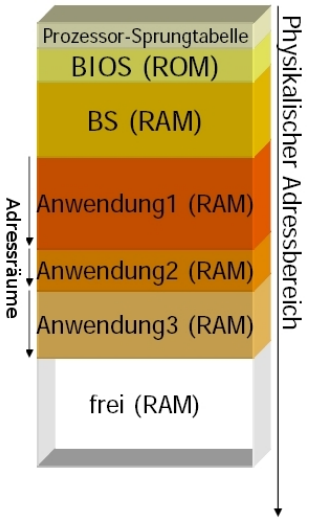
Beispiel UNIX mit dynamischer Relokation

Mehrere Prozesse, jeder hat **eigenen Adressraum**.

Bei jedem Kontextwechsel werden Basis- und Limitregister mitgeführt, sodass jeweils der Adressraum des aktuellen Prozesses gestellt wird.

- **Relokation**. Alle Programme werden für den gleichen Adressraum kompiliert.

- **Schutz**. Fremder Speicher ist gar nicht sichtbar.



5.6.1 Swapping

Die Anforderungen an die Verwaltung des Speichers ergeben sich aus den Design modernen Rechensysteme.

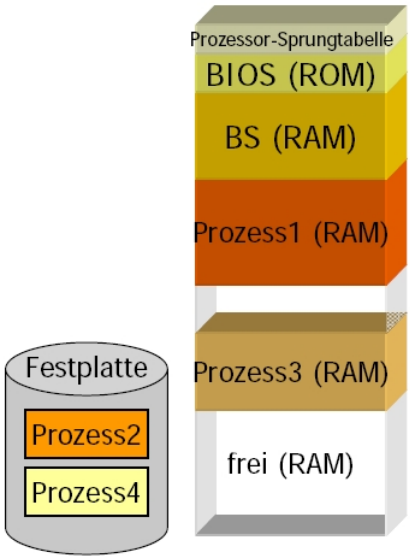
Es laufen soviele Prozesse, dass der physikalische Speicher nicht groß genug ist um alle gleichzeitig aufzunehmen.

Swapping ist ein grundlegender Ansatz, wie man der überlastung des Speichers begegnen kann, wobei der Adressraum eines Prozesses entweder vollständig im Speicher gehalten wird oder vollständig auf den Hintergrundspeicher (z.B. Festplatte) ausgelagert wird.

Swapping beschreibt das Ein-/Auslagern des **vollständigen** Adressraumes der Prozesses.

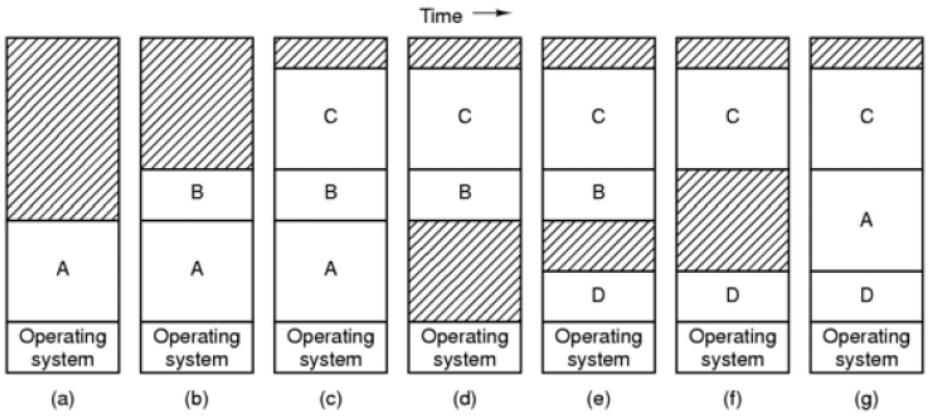
- Auslagern von (z.B. blockierten) Prozessen auf die Festplatte.
- Einlagern von (z.B. bereiten) Prozesses in den Speicher.

Es ist keine spezielle Hardware notwendig. Der Scheduler hat überblick über den rechnenden, die bereiten und blockierte Prozesse und kann die Ein-/Auslagerung veranlassen.



Beispiel

Speicherzuteilung für einzelne Prozesse ändert sich bei der Ein- und Auslagerung (freier Speicher = schraffierte Bereiche).

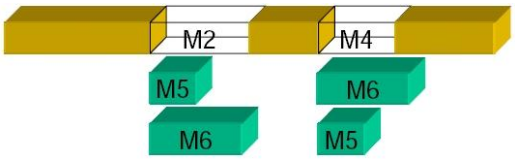


Beim Einlagern kann sich der physikalische Adressraum (die Partition) des Prozesses ändern. Das heißt bei dynamischer Relokation müssen beim Einlagern eines Prozesses u.U. die Werte im Basis- und Limitregister angepasst werden.

Probleme

Positionierung

M2, M4 werden freigegeben, wo M5, M6 platzieren?



Fragmentierung

Anforderung hat nur selten genau die Größe eines freien Speicherbereichs. Nach einiger Zeit entstehen viele kleine *Löcher* im Speicher.

Fazit.

Verschiedene **Speicherbelegungsstrategien** sind möglich.

Speicherbelegungsstrategien

First Fit

Der erste ausreichend große freie Speicherbereich wird belegt. Die Suche beginnt am Anfang des Speichers.

Next Fit

Wie First Fit, aber die Suche beginnt an der Stelle, wo zuletzt ein passendes Speicherbereich gefunden wurde. D.h. wenn beim letzten Mal das Loch nicht vollständig geschlossen, sondern lediglich verkleinert wurde, beginnt die Suche bei diesem Loch.

Best Fit

Es wird der kleinste freie Speicherbereich belegt, der die Anforderung noch erfüllen kann. Werden mehrere passende Speicherbereiche gefunden, wird einer davon ausgewählt.

Worst Fit

Es wird der größte freie Speicherbereich belegt. Werden mehrere passende Speicherbereiche gefunden, wird einer davon ausgewählt.

6 Automaten und Formale Sprachen

6.1 Einführung grundlegender Begriffe

Alphabet, Zeichen, Wort

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

Die Elemente eines Alphabets heißen **Zeichen**.

Eine Folge von Zeichen aus einem Alphabet ist ein **Wort** über dem Alphabet.

Für ein Alphabet Σ beschreibt $\mathbf{\Sigma^*}$ die **Menge aller Wörter** über dem Alphabet Σ.

Die Menge Σ^* umfasst auch das **leere Wort** ε.

Die **Länge** eines Wortes $w \in \Sigma^*$ entspricht der Anzahl der Zeichen des Wortes und wird mit $|\mathbf{w}|$ bezeichnet.

Mehrfaches Hintereinanderstellen eines Wortes wird beschrieben durch $\mathbf{w^n} = \underbrace{\mathbf{ww...w}}_{\mathbf{n-mal}}$.

Es gilt $\mathbf{w^0} = \varepsilon$.

Formale Sprache

Eine **formale Sprache** über dem Alphabet Σ ist eine beliebige Teilmenge von Σ^*.

Formale Sprachen sind im Allgemeinen unendlich. Damit man mit Ihnen umgehen kann benötigt man **endliche Beschreibungenmöglichkeiten** für formale Sprachen. Das sind zum Beispiel **Grammatiken** und **Automaten**.

Beispiel

Beispiel 6.1. Sei $\Sigma_{\text{expr}} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}, +, -, *, /\}$ ein Alphabet.

Die Worte der Sprache $L_{\text{expr}} \subseteq \Sigma^*_{\text{expr}}$, der einfachen arithmetischen Ausdrücke über den natürlichen Zahlen, bestehen aus Zahlen und Operatoren.

- Zahlen sind Ziffernfolgen ohne führende Nullen.
- Operatoren sind die Zeichen $+$, $-$, $*$, $/$.

Weiterhin gelten folgende Regeln.

- Ein Wort beginnt mit einer Zahl.
- Jeder in einem Wort enthaltene Operator steht zwischen zwei Zahlen.

$$\begin{aligned} 1234567890 &\in L_{\text{expr}} \\ 1024 * 2 + 128/64 &\in L_{\text{expr}} \\ 1 + 2 + +012 &\notin L_{\text{expr}} \end{aligned}$$

6.2 Endliche Automaten

Automaten sind eine **endliche Beschreibungenmöglichkeiten** für formale Sprachen.

Ein Automat arbeitet ein Wort über einem Alphabet (die Eingabe) zeichenweise ab und akzeptiert es schließlich oder nicht. Die Menge der akzeptierten Wörter bildet die durch den Automaten beschriebene Sprache.

Ein Automat kann deterministisch oder nichtdeterministisch arbeiten.

Ein **deterministischer Automat** befindet sich beim Abarbeiten der Eingabe zu jedem Zeitpunkt in genau einem Zustand. Eine Eingabe wird akzeptiert, wenn der Automat sich nach dem Abarbeiten der gesamten Eingabe in einem akzeptierenden Zustand befindet.

Ein **nichtdeterministischer Automat** kann sich während des Abarbeitens der Eingabe in mehreren Zuständen gleichzeitig befinden. Eine Eingabe wird akzeptiert, wenn einer der Zustände, in dem sich der Automat nach dem Abarbeiten der gesamten Eingabe befindet, ein akzeptierender Zustand ist.

Das Zulassen von Nichtdeterminismus kann folgende Gründe haben.

Beschreibungsmächtigkeit.

> Es gibt einen nichtdeterministische Automaten, zu denen kein äquivalenter deterministischer Automat existiert.

Laufzeit.

Es gibt nichtdeterministische Automaten, die ihre Eingaben in sehr viel weniger Schritten akzeptieren als jeder äquivalente deterministische Automat.

Beschreibungskomplexität.

Ein nichtdeterministische Automaten kann unter Umständen sehr viel kleiner sein als jeder äquivalente deterministische Automat.

Für die im Folgenden betrachteten **endliche Automaten** erhöht der Nichtdeterminismus nur die Beschreibungskomplexität.

Definition 6.2. Ein **deterministischer endlicher Automat** (DEA) ist gegeben durch ein 5-Tupel $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$. Dabei ist

- Σ das **Alphabet** der Eingabewörter,
- Q die endliche Menge der **Zustände**,
- q_0 der **Startzustand**, $q_0 \in Q$,
- F die Menge der **akzeptierenden Zustände**, $F \subseteq Q$,
- δ die **Überführungsfunktion**, $\delta : Q \times \Sigma \to Q$.

Sei $w = w_1 \dots w_i w_{i+1} \dots w_n \in \Sigma^*$ eine Eingabe.

Der Automat befindet sich, nach dem Abarbeiten der Eingabe bis zum Zeichen w_i (inclusive), im Zustand $q \in Q$.

Dann **überführt** die Funktion δ den Automaten in den Zustand $\delta(q, w_{i+1}) \in Q$.

Beispiel

Beispiel 6.3.

$\mathcal{A}_{\text{expr}} = (\Sigma_{\text{expr}}, \{q_0, q_{\text{expr}}, q_{\text{op}}, q_{\text{error}}\}, q_0, \{q_{\text{expr}}\}, \delta_{\text{expr}})$

$\Sigma_{\text{expr}} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}, +, -, *, /\}$

Die Überführungsfunktion bildet wie folgt ab.

$$\begin{aligned}
\delta_{\text{expr}}(q_0, a) &= q_{\text{expr}} \text{ für alle } a \in \{1,2,3,4,5,6,7,8,9\} \\
\delta_{\text{expr}}(q_0, a) &= q_{\text{error}} \text{ für alle } a \in \{0,+,-,*,/\} \\
\delta_{\text{expr}}(q_{\text{expr}}, a) &= q_{\text{expr}} \text{ für alle } a \in \{0,1,2,3,4,5,6,7,8,9\} \\
\delta_{\text{expr}}(q_{\text{expr}}, a) &= q_{\text{op}} \text{ für alle } a \in \{+,-,*,/\} \\
\delta_{\text{expr}}(q_{\text{op}}, a) &= q_{\text{expr}} \text{ für alle } a \in \{1,2,3,4,5,6,7,8,9\} \\
\delta_{\text{expr}}(q_{\text{op}}, a) &= q_{\text{error}} \text{ für alle } a \in \{0,+,-,*,/\} \\
\delta_{\text{expr}}(q_{\text{error}}, a) &= q_{\text{error}} \text{ für alle } a \in \Sigma_{\text{expr}}
\end{aligned}$$

Beispiel 6.4.

$\mathcal{A}_{\text{expr}} = (\Sigma_{\text{expr}}, Q_{\text{expr}}, q_0, \{q_{\text{expr}}\}, \delta_{\text{expr}})$

$\Sigma_{\text{expr}} = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, +, -, *, /\}$

$Q_{\text{expr}} = \{q_0, q_{\text{expr}}, q_{\text{op}}, q_{\text{error}}\}$

Die Überführungsfunktion δ_{expr} bildet wie folgt ab.

$p \in Q_{\text{expr}}$	$\delta_{\text{expr}}(p, 0)$	$\delta_{\text{expr}}(p, z)$ $z \in \{1, \ldots, 9\}$	$\delta_{\text{expr}}(p, \text{op})$ $\text{op} \in \{+, -, *, /\}$
q_0	q_{error}	q_{expr}	q_{error}
q_{expr}	q_{expr}	q_{expr}	q_{op}
q_{op}	q_{error}	q_{expr}	q_{error}
q_{error}	q_{error}	q_{error}	q_{error}

Zustandsgraph

Deterministische endliche Automaten können durch ihren **Zustandsgraphen** veranschaulicht werden.

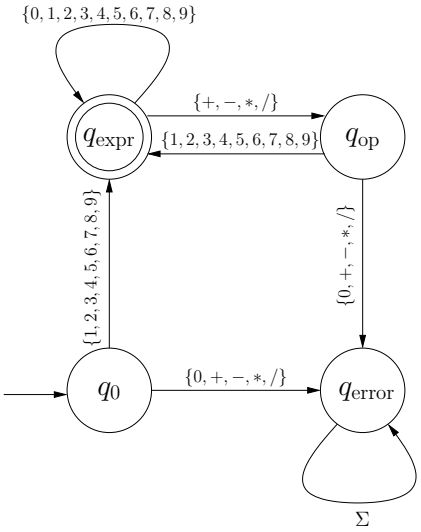
Ein Zustandsgraphen ist ein gerichteter Graph.

Für jeden Zustand enthält der Graph einen Knoten (dargestellt durch Kreise).

Der Anfangszustand ist durch eine unmarkierte eingehende Kante gekennzeichnet.

Akzeptierende Zustände sind durch Doppelkreise gekennzeichnet.

Jeder Knoten hat für jedes Zeichen aus dem Alphabet eine ausgehende Kante.



Folgezustand

Sei $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ ein endlicher Automat.

- Für den Zustand $q \in Q$ bezeichne $qa = \delta(q, a)$ den **Folgezustand** von q nach dem Lesen des Zeichens $a \in \Sigma$.
- Für den Zustand $q \in Q$ und das Wort $w = w_1 w_2 \dots w_n \in \Sigma^*$ bezeichne $qw = (\cdots((qw_1)w_2), \dots, w_n)$ den **Folgezustand** von q nach dem Lesen des Wortes w.

Beispiel 6.5. $\mathcal{A}_{\text{expr}} = (\Sigma_{\text{expr}}, \{q_0, q_{\text{expr}}, q_{\text{op}}, q_{\text{error}}\}, q_0, \{q_{\text{expr}}\}, \delta_{\text{expr}})$

$$\begin{aligned} q_{\text{expr}}\ + &= q_{\text{op}} \\ q_0\ 1024 * 2 + 128/64 &= q_{\text{expr}} \\ q_{\text{expr}}\ + 012 &= q_{\text{error}} \end{aligned}$$

Ein Wort wird **akzeptiert**, wenn das Lesen seiner Zeichen den deterministischen Automaten von seinem Startzustand in einen akzeptierenden Zustand überführt.

Bemerkung

Das leere Wort ε wird genau dann akzeptiert, wenn der Startzustand ein akzeptierender Zustand ist.

Die Menge aller Wörter, die von dem Automaten akzeptiert werden können, ist die durch den Automaten akzeptierte Sprache.

Definition 6.6. Die durch den deterministischen endlichen Automaten $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ **akzeptierte Sprache** $L_\mathcal{A} = \{w \in \Sigma^* \mid q_0 w \in F\}$.

Beispiel 6.7. Die von $\mathcal{A}_{\text{expr}}$ akzeptierte Sprache, ist die Sprache der einfachen arithmetischen Ausdrücke über den natürlichen Zahlen, $L_{\text{expr}} = L_{\mathcal{A}_{\text{expr}}}$.

Nichtdeterministische endliche Automaten

Definition 6.8. Ein **nichtdeterministischer endlicher Automat** (NEA) ist gegeben durch ein 5-Tupel $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$. Dabei ist

- Σ das **Alphabet** der Eingabewörter,
- Q die endliche Menge der **Zustände**,
- q_0 der **Startzustand**, $q_0 \in Q$,
- F die Menge der **akzeptierenden Zustände**, $F \subseteq Q$,
- Δ die **Überführungsfunktion**, $\Delta : Q \times \Sigma \to \{U \mid U \subseteq Q\}$.

Sei $w = w_1 \dots w_i w_{i+1} \dots w_n \in \Sigma^*$ eine Eingabe.

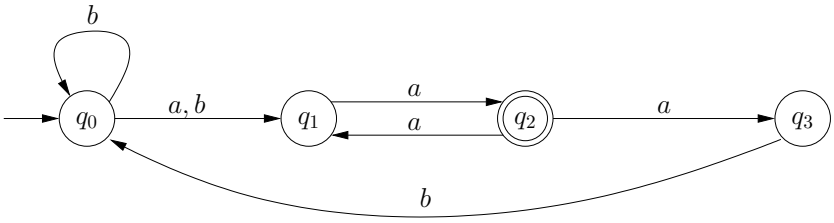
Der Automat befinde sich, nach dem Abarbeiten der Eingabe bis zum Zeichen w_i (inclusive), gleichzeitig in den Zuständen $U \subseteq Q$.

Dann **Überführt** die Funktion Δ den Automaten in folgende Zustandsmenge.

$$\bigcup_{q \in U} \Delta(q, w_{i+1}) \subseteq Q$$

Auch Nichtdeterministische endliche Automaten können durch einen Zustandsgraphen veranschaulicht werden.

Im Unterschied zu deterministischen endlichen Automaten kann ein Knoten für jedes Zeichen aus dem Alphabet **beliebig viele ausgehenden Kanten** (auch keine) haben.



Ein Wort wird akzeptiert, wenn das Lesen seiner Zeichen den nichtdeterministischen Automaten von seinem Startzustand in eine Menge von Zuständen überführt, die einen akzeptierenden Zustand enthält.

Bemerkung

Das leere Wort ε wird genau dann akzeptiert, wenn der Startzustand selbst ein akzeptierender Zustand ist.

Sei $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$ ein nichtdeterministischer endlicher Automat und $q \in Q$ ein beliebiger Zustand.

Für $w = w_1 w_2 \dots w_n$ bezeichne qw die Menge der Folgezustände. Es gilt

$$\begin{aligned} M_1 &:= \Delta(q, w_1) \\ M_i &:= \bigcup_{p \in M_{i-1}} \Delta(p, w_i) \quad \text{für } i = 2, \dots, n \\ qw &:= M_n \end{aligned}$$

Definition 6.9. Die durch den nichtdeterministischen endlichen Automaten $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$ **akzeptierte Sprache** $L_{\mathcal{A}} = \{w \in \Sigma^* \mid q_0 w \cap F \neq \emptyset\}$.

Satz 6.10. *Jeder nichtdeterministischen endlichen Automaten kann mit einem äquivalenten deterministicher Automat simuliert werden, der dieselbe formale Sprache akzeptiert.*

Beweis.

Durch Konstruktion.

Simulation eines NEA mit einem DEA über den Zustandsgraphen.

Schritt 1. Gegeben ist ein NEA mit n Zuständen.

- Erstelle einen Graphen, dessen Knoten markiert werden können und jeweils mehrere Zustände des NEA repräsentieren können.
- Der Graph enthält nur einen Knoten, der den Startzustand des NEA repräsentiert und nicht markiert ist.

Schritt 2. Betrachte einen nicht markierten Knoten v des Graphen.

- Sei Q_v die Menge der von v repräsentierten Zuständen des NEA.
- Für jedes $a \in \Sigma$ führe Folgendes aus.
  - Ermittle die Menge $Q_v a$ der Zustände, die unter a im NEA von den Zuständen in Q_v erreichbar sind, $Q_v a = \bigcup_{q \in Q_v} \Delta(q, a)$.
  - Erzeuge einen nicht markierten Knoten, der die Menge $Q_v a$ repräsentiert, wenn er noch nicht vorhanden ist (auch für $Q_v a = \emptyset$).

    Bemerkung. Die $Q_v a$ sind nicht notwendigerweise alle verschieden.
  - Füge eine mit a markierte Kante vom Knoten v zu dem $Q_v a$ repräsentierenden Knoten ein.
- Der abgearbeitete Knoten v wird markiert.

Wiederhole Schritt 2 bis keine nicht markierten Knoten mehr übrig sind.

Schritt 3. Graph zu DEA.

- Die Knoten sind die Zustände des DEA.

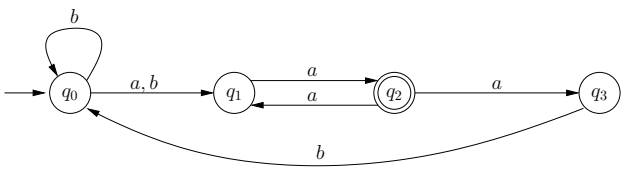
  Bemerkung. Es gibt nur noch markierte Knoten. Deshalb hat jeder Knoten für jedes $a \in \Sigma$ genau eine ausgehende Kante.
- Der Startzustand des DEA ist der Knoten, der den Startzustand des NEA repräsentiert.

- Die akzeptierenden Zustände des DEA sind die Konten, die mindestens einen akzeptierenden Zustand des NEA repräsentieren.

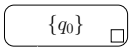
Beispiel

Schritt 1

NEA

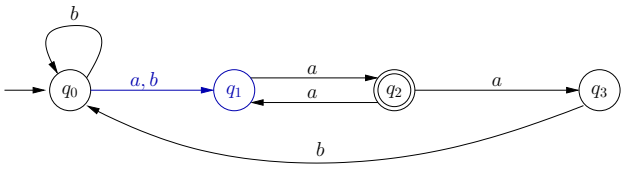


Graph

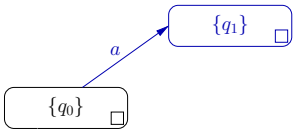


Schritt 2 (Auswahl)

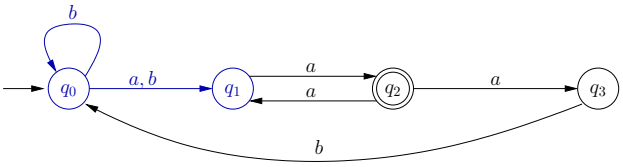
NEA



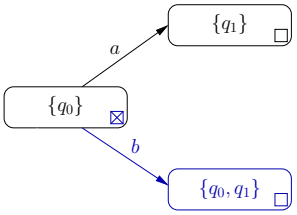
Graph



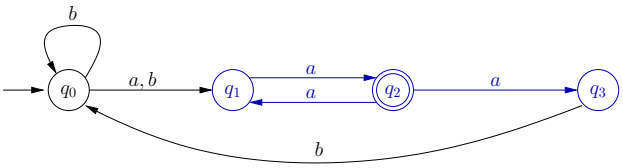
NEA



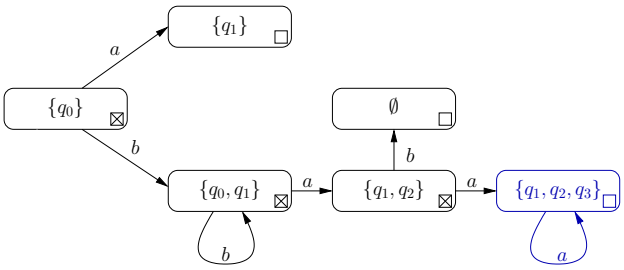
Graph



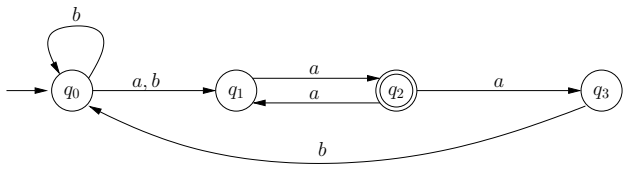
NEA



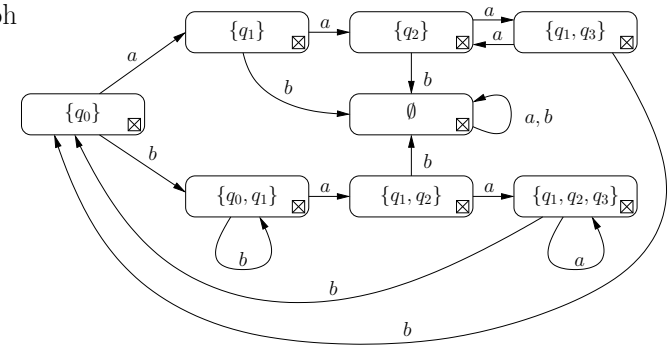
Graph



NEA

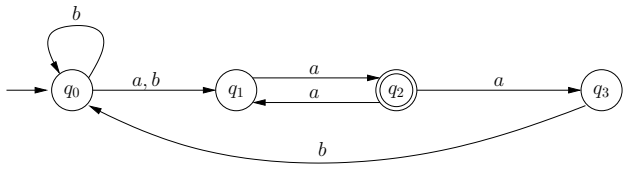


Graph

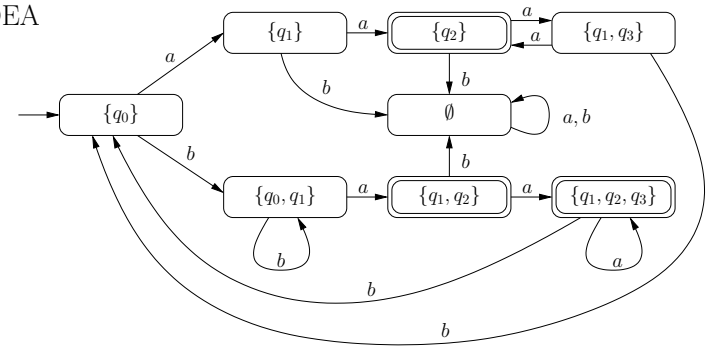


Schritt 3

NEA



DEA



Beschreibungsmächtigkeit.

Zu jedem NEA existiert ein äquivalenter DEA.

Laufzeit.

Ein Eingabe wird sowohl vom NEA als auch von DEA nach Abarbeitung aller Zeichen akzeptiert.

Beschreibungskomplexität.

Die Mächtigkeit der Zustandmenge eines DEA, der einen NEA $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$ simuliert, ist im schlechtesten Fall

$$|\{U \mid U \subseteq Q\}| = 2^{|Q|} \ .$$

6.3 Reguläre Sprachen

Eine von einem endlichen Automaten akzeptierte Sprache ist eine formale Sprache. Aber nicht für jede formale Sprache kann ein endlicher Automat angegeben werden, der diese akzeptiert.

Definition 6.11. Eine formale Sprache $L \subseteq \Sigma^*$ heißt **reguläre Sprache**, wenn es einen endlichen Automaten $\mathcal{A}$ gibt der L akzeptiert, $L = L_{\mathcal{A}}$.

Regeln für reguläre Sprachen

Ist $L \subseteq \Sigma^*$ regulär, dann auch ist auch das *Komplement* regulär.

$$\overline{L} := \{w \in \Sigma^* | w \notin L\}$$

Sind L_1, L_2 regulär, dann ist auch der *Durchschnitt* regulär.

$$L_1 \cap L_2 := \{w | w \in L_1 \text{ und } w \in L_2\}$$

Sind L_1, L_2 regulär, dann ist auch die *Verkettung* regulär.

$$L_1 L_2 := \{vw | v \in L_1, w \in L_2\}$$

Sind L_1, L_2 regulär, dann ist auch die *Vereinigung* regulär.

$$L_1 \cup L_2 := \{w | w \in L_1 \text{ oder } w \in L_2\}$$

Die Kleenesche Hülle

Seien $L_1, L_2 \subseteq \Sigma^*$ Sprachen über Σ. Dann ist die Sprache

$$L_1 L_2 := \{vw \mid v \in L_1, w \in L_2\} \ .$$

Sei $L \subseteq \Sigma^*$, dann gilt

$$\begin{aligned} L^0 &:= \{\varepsilon\} \\ L^1 &:= L \\ L^i &:= LL^{i-1} \quad \text{für alle } i \in \mathbb{N} \\ L^* &:= \bigcup_{i \in \mathbb{N}} L^i \end{aligned}$$

Die Sprache L^* ist die **Kleenesche Hülle** der Sprache L. Beispiel: $L = \{a, bb\}$ und $L^* = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, \ldots\}$

Ist L regulär, dann ist auch L^* regulär.

Definition 6.12. Die **regulären Basissprachen** eine Alphabets Σ sind

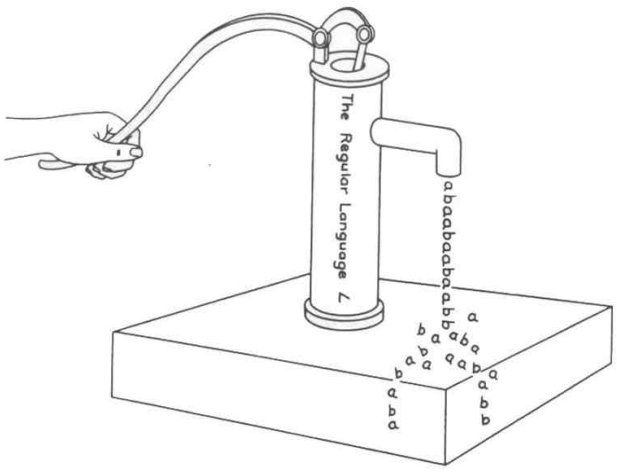
- $\emptyset$,
- $\{\varepsilon\}$,
- $\{a\}$ für jedes $a \in \Sigma$.

Satz 6.13. *Eine Sprache $L \subseteq \Sigma^*$ ist regulär genau dann, wenn sie durch eine endliche Folge der Operationen Vereinigung, Verkettung und Kleenesche Hülle aus den regulären Basissprachen von Σ erzeugt werden kann.*

Reguläre Sprachen sind *einfach gebaut.*

6.3.1 Pumping-Lemma für reguläre Sprachen

Das **Pumping-Lemma** liefert ein Kriterium um zu zeigen, dass eine Sprache **nicht regulär** ist.



Satz 6.14. *Sei L eine reguläre Sprache. Dann gibt es eine natürliche Zahl (Pumping-Konstante) $k \in \mathbb{N}$, sodass man jedes Wort $w \in L$ mit $|w| \geq k$ zerlegen kann in Wörter $xyz \in \Sigma^*$, für die gilt*

- $w = xyz$,
- $|y| \geq 1$, *das heißt* $y \neq \varepsilon$,
- $|xy| \leq k$,
- *für alle* $i \in \mathbb{N} \cup \{0\}$ *gilt,* $xy^iz \in L$.

Anhand des Pumping-Lemmas kann bewiesen werden, dass eine Sprache **nicht regulär** ist. Trifft das Pumping-Lemma nicht zu, dann ist die Sprache auch nicht regulär.

Satz 1
Werden k Objekte in ℓ Fächer gegeben ($\ell < k$), dann enthält mindestens eins der Fächer mehr als ein Objekt.

Beweis. Seien $F := \{F_1, \ldots, F_\ell\}$ die Menge der Fächer. Sei $|F_i|$ die Anzahl der Objekte im Fach F_i. Es gilt

$$\sum_{i=1}^{\ell} |F_i| = k$$

Annahme. Keins der Fächer enthält mehr als ein Objekt. Das heißt für alle F_i gilt $|F_i| \leq 1$, daraus folgt

$$\sum_{i=1}^{\ell} |F_i| \leq \ell < k$$

Das ist ein Widerspruch. □

Beweis

Da L regulär ist gibt es eine deterministischen endlichen Automaten $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$, der L akzeptiert.

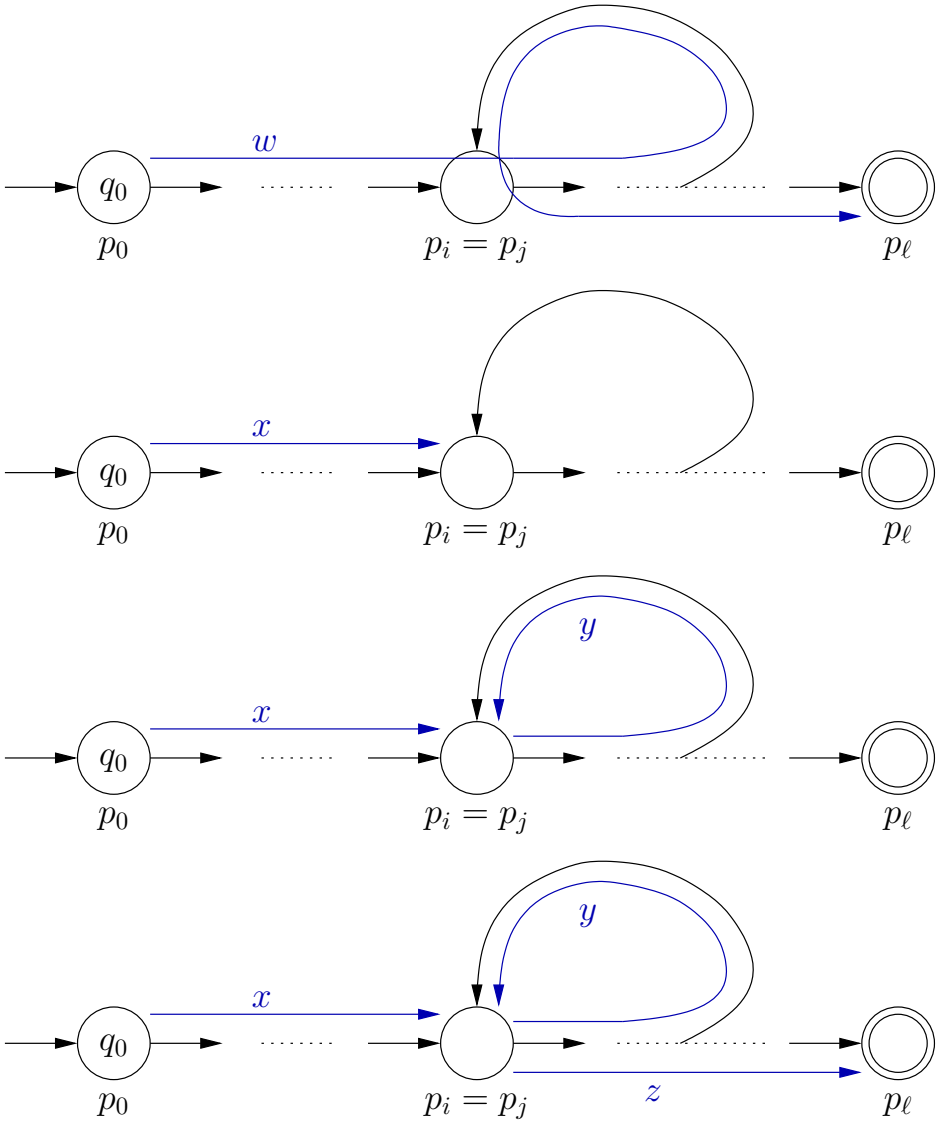
Wähle die Pumping-Konstante $k = |Q|$, wobei $|Q|$ die Anzahl der Zustände von $\mathcal{A}$ ist.

Sei $w \in L$ ein beliebiges Wort der Länge $|w| = \ell \geq k$.

Das Wort w wird vom Automaten $\mathcal{A}$ akzeptiert. Beim Akzeptieren von w durchläuft der Automat $\ell + 1$ Zustände $p_0, \ldots, p_\ell \in Q$, mit Startzustand $p_0 = q_0$ und akzeptierenden Zustand $p_\ell \in F$.

Da $k+1 > |Q|$ folgt mit dem Schubfachprinzip, dass in der Teilfolge $p_0, \ldots, p_k$ ein Zustand doppelt auftreten muss.

Das heißt, es gibt i, j mit $0 \leq i < j \leq k$ für die gilt $p_i = p_j$.



Das Wort w wird nun in $x, y, z \in \Sigma^*$ zerlegt, sodass Folgendes gilt.

$$p_0 x = p_i \qquad p_i y = p_j \qquad p_j z = p_\ell$$

Es gilt

- $xyz = w$ nach Konstruktion,
- $|y| \geq 1$, weil $i < j$,
- $|xy| \leq k$, weil $j \leq k$,
- für alle $i \in \mathbb{N} \cup \{0\}$ gilt $xy^i z \in L$ und mit $\hat{p} := p_i = p_j$ folgt

$$\begin{aligned} p_0 x &= \hat{p} \\ \hat{p} y &= \hat{p} \quad \text{also} \quad \hat{p} y y = \hat{p} y = \hat{p} \quad \text{und} \quad \hat{p} y^i = \hat{p} \\ \hat{p} z &= p_\ell \end{aligned}$$

  das heißt $p_0 x y^i z = p_\ell$.

□

6.3.2 Anwendung des Pumping-Lemmas

Behauptung.

Die Sprache L ist nicht regulär.

Beweis.

Durch Widerspruch.

Annahme. L ist regulär, daraus folgt das Pumping-Lemma gilt mit Pumping-Konstante $k \in \mathbb{N}$.

Betrachte **ein** Wort $w \in L$ mit $|w| \geq k$ und zeige **für jede** Zerlegung $xyz = w$ mit $|y| \geq 1$ und $|xy| \leq k$ **gibt es** ein $i \in \mathbb{N} \cup \{0\}$ mit $xy^i z \notin L$.

Das ist ein Widerspruch. □

Behauptung.

Die Sprache $L = \{a^n b^n \mid n \in \mathbb{N}\}$ ist nicht regulär.

Beweis. Durch Widerspruch.

Annahme. L ist regulär, daraus folgt das Pumping-Lemma gilt mit Pumping-Konstante $k \in \mathbb{N}$.

Betrachte das Wort $a^k b^k \in L$. Sei $xyz = a^k b^k$ eine beliebige Zerlegung.

$|xy| \leq k$, daraus folgt xy besteht nur aus a's, das heißt $xy = a^r$ mit $r \leq k$.

$|y| \geq 1$, daraus folgt y besteht aus mindestens einem a, das heißt $xy = a^{r-s}a^s$ mit $1 \leq s \leq r$.

Es gilt $a^k b^k = a^{r-s}a^s a^{k-r} b^k = xyz$.

Abpumpen von y. Es gilt $xy^0 z = a^{k-s} b^k \notin L$, weil mit $s \geq 1$ gilt $k - s \neq k$. Das ist ein Widerspruch.

Aufpumpen von y. Sei $i \geq 2$, dann ist $xy^i z = a^{k-s} a^{si} b^k \notin L$, weil mit $s \geq 1$ gilt $k + s(i-1) \neq k$. Das ist ein Widerspruch. □

6.4 Grammatiken

Ergänzung.

Σ^+ ist die Menge der nichtleeren Zeichenketten über dem Alphabet Σ.

Definition 6.15. Eine **Grammatik** über einem Alphabet Σ ist ein 4-Tupel $G = (N, T, P, S)$ bestehend aus:

- einer Menge N von *Nichtterminalsymbolen*
- einer Menge $T \subseteq \Sigma$ von *Terminalsymbolen* mit $N \cap T = \emptyset$
- einer nichtleeren Menge $P \subseteq (N \cup T)^+ \times (N \cup T)^*$ von *Produktionen*

  [Produktion (p, q) wird mit $p \to q$ notiert.]
- einem Startsymbol $S \in N$.

Beispiel

Beispiel 6.16. $G_{\mathtt{a}^n\mathtt{b}^n} = (\{\mathtt{S}\}, \{\mathtt{a}, \mathtt{b}\}, \{S \to \mathtt{a}S\mathtt{b}, S \to \varepsilon\}, S)$

Abkürzung für *Produktionsgruppen* mit gleicher linker Seite.

Beispiel: $S \to \mathtt{a}S\mathtt{b}|\varepsilon$

Ableitung

Sei $x = vpw$ mit $v, w \in (N \cup T)^*$, $p \in (N \cup T)^+$ und $p \to q$ eine Produktion.

Durch Ersetzen von p durch q entsteht $y = vqw \in (N \cup T)^*$.

Notation. $x \Longrightarrow y$ ($x \overset{p \to q}{\Longrightarrow} y$).

Allgemein.

Seien $x \in (N \cup T)^+$ und $y \in (N \cup T)^*$. Dann heißt y mit G **ableitbar** aus x, falls

- $x = y$ oder
- es gibt $v_1, v_2, \ldots, v_n$ mit $x = v_1$, $y = v_n$ und $v_i \Longrightarrow v_{i+1}$ für $1 \leq i < n$

Notation. $x \overset{G}{\Longrightarrow} y$

Definition 6.17. Die von der Grammatik G **erzeugte Sprache** $L(G)$ ist die Menge aller aus dem Startsymbol S ableitbaren Worte aus T^*.

$$L(G) = \{w \mid S \overset{G}{\Longrightarrow} w \text{ und } w \in T^*\}$$

Beispiel 6.18. Die Grammatik $G_{\mathtt{a}^n\mathtt{b}^n}$ erzeugt (oder generiert) die Sprache $L_{\mathtt{a}^n\mathtt{b}^n} := \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$.

6.5 Reguläre Grammatiken

Definition 6.19. Die Grammatik (N, T, P, S) heißt **rechtslinear**, wenn jede Produktion die Form $A \to aB$ oder $A \to a$ oder $A \to \varepsilon$ hat, mit $A, B \in N$ und $a \in T$.

Definition 6.20. Die Grammatik (N, T, P, S) heißt **linkslinear**, wenn jede Produktion die Form $A \to Ba$ oder $A \to a$ oder $A \to \varepsilon$ hat, mit $A, B \in N$ und $a \in T$.

Definition 6.21. Eine **reguläre Grammatik** ist eine rechtslineare oder linkslineare Grammatik.

Satz 6.22. *Zu jedem endlichen Automaten $\mathcal{A}$ kann man eine reguläre Grammatik $G_{\mathcal{A}}$ konstruieren mit $L(G_{\mathcal{A}}) = L_{\mathcal{A}}$ und umgekehrt.*

Beweis durch Konstruktion.

- Erzeuge aus einer rechtslinearen Grammatik einen äquivalenten nichtdeterministischen endlichen Automaten.
- Es gilt, nichtdeterministischen und deterministische endlichen Automaten sind äquivalent.
- Erzeuge aus einem deterministischen endlichen Automaten eine äquivalente rechtslinearen Grammatik.
- Bleibt zu zeigen, rechtslineare und linkslineare Grammatiken sind äquivalent.

Rechtslineare Grammatik → NEA

Sei $G = (N, T, P, S)$ einen rechtslineare Grammatik. Erzeuge einen nichtdeterministischen endlichen Automat $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$, sodass $L(G) = L_{\mathcal{A}}$ gilt.

- Setze Alphabet gleich Terminale, $\Sigma := T$.
- Setze Zustände gleich Nichterminale, $Q := N$.
- Setze Startzustand gleich Startsymbol, $q_0 := S$.
- Noch keine akzeptierenden Zustände, $F := \emptyset$.
- Initialisiere die Übergangsfunktion $\Delta(A, a) = \emptyset$ für alle $A \in Q$, $a \in \Sigma$.
- Gibt es eine Produktion $A \to a$, erzeuge neuen akzeptierenden Zustand E, setze $F := F \cup \{E\}$ und $Q := Q \cup \{E\}$.
- Für alle Produktionen $A \to \varepsilon$ setze $F := F \cup \{A\}$.
- Für alle Produktionen $A \to a$ erweitere Δ, sodass gilt $E \in \Delta(A, a)$.
- Für alle Produktionen $A \to aB$ erweitere Δ, sodass gilt $B \in \Delta(A, a)$.

DEA → rechtslineare Grammatik

Sei $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ ein deterministischer endlicher Automat. Erzeuge eine rechtslineare Grammatik $G = (N, T, P, S)$, sodass $L_{\mathcal{A}} = L(G)$ gilt.

- Setze Terminale gleich Alphabet, $T := \Sigma$.
- Setze Nichterminale gleich Zustände $N := Q$.
- Setze Startsymbol gleich Startzustand, $S := q_0$.
- Initialisiere die Menge der Produktionen, $P := \emptyset$.
- Für alle $A \in F$ erweitere die Menge der Produktionen $P := P \cup \{A \to \varepsilon\}$.
- Für alle $A \in Q$, $a \in \Sigma$ erweitere die Menge der Produktionen $P := P \cup \{A \to aB\}$ mit $B = \delta(A, a)$.

  Bemerkung.

  Es müssen auch die akzeptierenden Zustände ($A \in F \subset Q$) berücksichtigt werden.

Links- und rechtslineare Grammatiken

Satz 6.23. *Für jede linkslineare Grammatik G_ℓ kann eine rechtslineare Grammatik G_r konstruiert werden mit $L(G_\ell) = L(G_r)$ und umgekehrt.*

Beweis durch Konstruktion.

Linkslineare Grammatik → rechtslineare Grammatik

Sei $G_\ell = (N_\ell, T_\ell, P_\ell, S_\ell)$ einen linkslineare Grammatik und $G_r = (N_r, T_r, P_r, S_r)$ die gesuchte rechtslineare Grammatik.

Die Terminale sind gleich.

- $T_r := T_\ell$

Die Nichterminale werden erweitert um eine neues Startsymbol S_r, das alte Startsymbol S_ℓ wird ein einfaches Nichterminal.

- $N_r := N_\ell \cup \{S_r\}$

Initalisiere die Produktionen. G_ℓ erzeugt die Wörter von rechts nach links, d.h. in G_r ist S_ℓ das Ende eines Wortes.

- $P_r := \{S_\ell \to \varepsilon\}$

Der Abschluss eines Wortes in G_ℓ ist der Anfang eines Wortes in G_r.

- Für alle $(A \to \varepsilon) \in P_\ell$ setze $P_r := P_r \cup \{S_r \to A\}$.
- Für alle $(A \to a) \in P_\ell$ setze $P_r := P_r \cup \{S_r \to aA\}$.

Alle Produktionen mit zwei Nichtterminalen ändern die Erzeugungsrichtung.

- Für alle $(A \to Ba) \in P_\ell$, setze $P_r := P_r \cup \{B \to aA\}$.

Entferne aus P_r alle Produktionen $S_r \to A$ durch Einsetzen.

- $P_r := P_r \backslash \{S_r \to A\}$.
- Für alle $(A \to \varepsilon) \in P_r$ setze $P_r := P_r \cup \{S_r \to \varepsilon\}$
- Für alle $(A \to a) \in P_r$ setze $P_r := P_r \cup \{S_r \to a\}$
- Für alle $(A \to aB) \in P_r$ setze $P_r := P_r \cup \{S_r \to aB\}$

Entferne aus P_r alle Produktionen mit linker Seite $A \neq S_r$, wenn A auf keiner rechten Seite einer Produktion vorkommt, deren linke Seite nicht A ist.

Bemerkung

Die Konstruktion einer äquivalenten linkslinearen Grammatik zu einer gegebenen rechtslinearen Grammatik kann analog durchgeführt werden.

Bemerkung

Wenn rechts- und linkslineare Regeln in der Menge der Produktionen gemischt werden, darf man nicht erwarten, dass das Ergebnis regulär ist.

Beispiel 6.24. Die Grammatik $G = \{\{S, A, B\}, \{a, b\}, \{S \to Ba, S \to aA, A \to Sa, B \to aS, S \to b\}, S\}$ erzeugt die Sprache $\{a^i b a^i\}$, die nicht regulär ist.

6.6 Parser

6.6.1 Kontextfreie Grammatiken

Definition 6.25. Eine Grammatik (N, T, P, S) heißt **kontextsensitive Grammatik**, wenn für alle Produktionen $p \to q$ aus P gilt $|p| \leq |q|$. [Keine rechte Seite einer Produktionen ist kürzer als die zugehörigen linke Seite.]

Definition 6.26. Eine formale Sprache ist genau dann ein **kontextsensitive Sprache** (*context sensitive language, CSL*), wenn es eine kontextsensitive Grammatik gibt, die diese Sprache erzeugt.

Definition 6.27. Eine Grammatik (N, T, P, S) heißt **kontextfreie Grammatik**, wenn für alle Produktionen $p \to q$ aus P gilt $p \in N$. [$p \to q$ ist unabhängig von umgebenden Symbolen anwendbar oder nicht.]

Definition 6.28. Eine formale Sprache ist genau dann ein **kontextfreie Sprache** (*context free language, CFL*), wenn es eine kontextfreie Grammatik gibt, die diese Sprache erzeugt.

Beispiel 6.29. Die kontextfreie Grammatik

$$G_{\mathtt{a}^n\mathtt{b}^n} = (\{\mathtt{S}\}, \{\mathtt{a}, \mathtt{b}\}, \{S \to \mathtt{a}S\mathtt{b} \mid \varepsilon\}, S)$$

erzeugt die Sprache

$$L_{\mathtt{a}^n\mathtt{b}^n} := \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\} \ .$$

Beispiel 6.30. Grammatik $G := \{N, T, P, S\}$

- Terminale $T := \{+,\ (,\ ),\ \text{int},\ \$\$\}$.
- Nichtterminale $N := \{\text{START},\ \text{SUM},\ \text{EXPR}\}$.
- Startsymbol $S := \text{START}$.

- Produktionen P

  START → EXPR $$
  EXPR → (SUM)
  EXPR → int
  SUM → EXPR + EXPR

6.6.2 Ableitung von kontextfreien Grammatiken

Wenn eine kontextfreie Grammatik G zur Analyse von Quelltexten einer Programmiersprache verwendet wird, dann gilt Folgendes.

- $L(G)$ ist die Menge der Tokenfolgen, die zu gültigen Quellprogrammen gehören.
- Der Scanner (lexikalischen Analyse) liefert ein Folge von Terminalsymbolen der Grammatik (genannt *Token*) zurück.

  Das $$ Token markiert das Ende der Eingabe (z.B. *EOF*). Es ist hilfreich, denn damit kann das Ereignis *Ende der Eingabe erreicht* in die Grammatik eingearbeitet werden.

  Beispiel

  Quellprogramm `(77+(5+10))` wird zur Tokenfolge `(int+(int+int))`$$.
- Der **Parser** (Syntaxanalyse) versucht die Tokenfolge entsprechend der Grammatik abzuleiten.

  Gelingt die Ableitung, ist die Tokenfolge und damit das Quellprogrammen gültig.

Der Prozess zur Erzeugung eines Wortes aus einer Grammatik ist die **Ableitung** (*derivation*).

Ein durch den Ableitungsprozess entstehende Zeichenkette, insbesondere wenn diese noch Nichtterminale enthält, wird **Satzform** genannt.

Das die abschließende Satzform (das Wort der Sprache), die nur Terminalsymbole enthält, wird auch **Satz** (*sentence*) der Grammatik oder **Ergebnis** (*yield*) des Ableitungsprozesses genannt.

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik.

- **Start.** Beginne mit einer Produktion $S \to u \in (N \cup T)^*$. Setze die Satzform sf gleich der rechten Seite dieser Produktion $sf := u$.
- **Abbruchbedingung.** Enthält die Satzform sf nur Terminale ist sf das Ergebnis des Ableitungsprozesses.
- **Einsetzen.** Wähle ein Nichtterminal A aus der Satzform $sf = vAw$ und eine Produktion $A \to q \in (N \cup T)^*$. Ersetze A durch q, das heißt $sf = vqw$.
- **Schleife.** Beginne wieder mit dem Prüfen der Abbruchbedingung.

Der durch den Ableitungsprozess erzeugte Satz sf ist ein Wort der von G erzeugten Sprache $L(G)$, das heißt $sf \in L(G)$.

Rechtsseitige Ableitungen (*rightmost derivations*)

- Ersetze jeweils das **erste rechte** Nichtterminalsymbol in jedem Ableitungsschritt.
- Manchmal auch **kanonische** Ableitung genannt.

Linksseitige Ableitungen (*leftmost derivations*)

- Ersetze jeweils das **erste linke** Nichtterminalsymbol in jedem Ableitungsschritt.

Andere Ableitungsreihenfolgen sind möglich.

Bemerkung

Die meisten Parser suchen entweder nach einer rechtsseitigen oder linksseitigen Ableitung

Beispiel 6.31. Grammatik $G := \{N, T, P, S\}$

- Terminale $T := \{+,\ (,\ ),\ \text{int},\ \$\$\}$.
- Nichtterminale $N := \{\text{START},\ \text{SUM},\ \text{EXPR}\}$.
- Startsymbol $S := \text{START}$.
- Produktionen

  START $\to$ EXPR \$\$
  EXPR $\to$ (SUM)
  EXPR $\to$ int
  SUM $\to$ EXPR + EXPR

Ist folgender Satz gültig?

(int + (int + int)) $$

Eine mögliche linksseitige Ableitung des Satzes.

START → EXPR $$
(SUM) $$
(EXPR + EXPR) $$
(int + EXPR) $$
(int + (SUM)) $$
(int + (EXPR + EXPR))$$
(int + (int + EXPR)) $$
(int + (int + int)) $$

6.6.3 Syntaxanalyse

Ableitungsbaum

Ein **Ableitungsbaum** (auch Parsebaum genannt) ist eine graphische Repräsentation des Ableitungsprozesses.

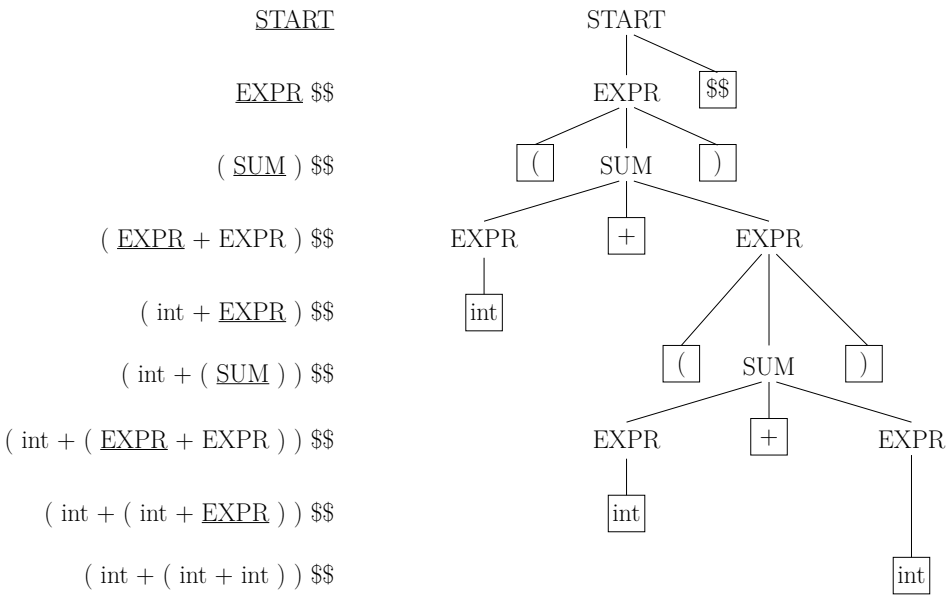
Die **Blätter** eines Ableitungsbaumes entsprechen den Terminalsymbolen der Grammatik.

Innere Knoten eines Ableitungsbaumes entsprechen den Nichtterminalsymbolen der Grammatik (linke Seite der Produktionen).

Bemerkung

Die meisten Parser konstruieren einen Ableitungsbaum während des Ableitungsprozesses für eine spätere Analyse.

Beispiel



Eindeutig/Mehrdeutig

Eindeutige Grammatiken haben genau einen Ableitungsbaum, wenn nur links- oder rechtsseitige Ableitungen verwendet werden.

Eine Grammatik ist **mehrdeutig**, wenn ein Satz mit (mindestens zwei) verschiedenen Ableitungsbäumen abgeleitet werden kann, obwohl nur links- oder rechtsseitige Ableitungen verwendet werden.

Es keinen allgemeingültigen Algorithmus zur Entdeckung von Mehrdeutigkeiten und deren Auflösung.

Vermeidung von Mehrdeutigkeiten durch Festlegung der Auswertungsreihenfolge.

- Bei verschiedenen Terminalen Prioritäten setzen.
- Bei gleichartigen Terminalen Gruppierung festlegen.

Beispiel

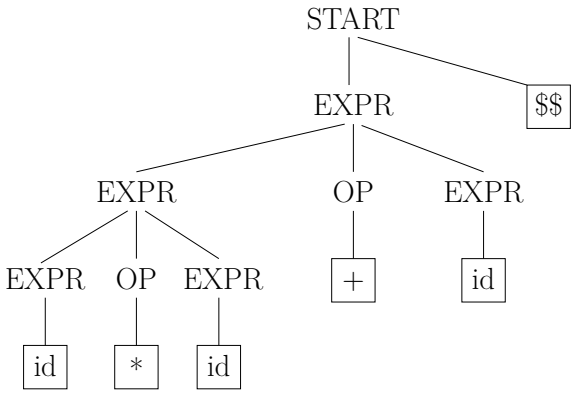
Beispiel 6.32.

- Terminale $T := \{+,\ -,\ *,\ /,\ \text{id},\ \$\$\}$.
- Nichtterminale $N := \{\text{START},\ \text{EXPR},\ \text{OP}\}$.
- Startsymbol $S :=$ START.
- Produktionen

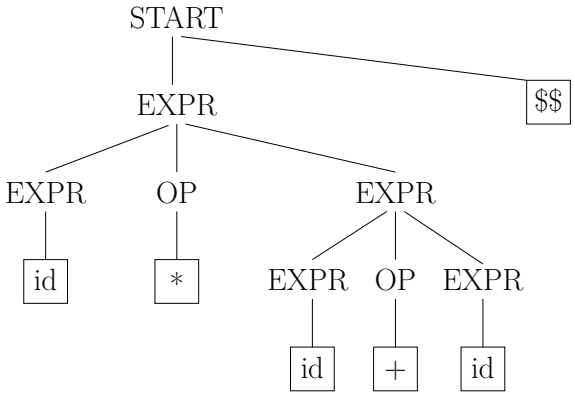
  START → EXPR $$
  EXPR → id | EXPR OP EXPR
  OP → + | - | * | /

Satz
`id * id + id $$`

Ableitungsbaum 1



Ableitungsbaum 2



Kontextfreie Grammatiken **erzeugen** durch den Ableitungsprozess Sätze.

Ein Parser für eine kontextfreie Grammatik **erkennt** Sätze in der von einer kontextfreien Grammatiken erzeugten Sprache.

Parser können automatisch aus einer kontextfreien Grammatik generiert werden.

Parser zum Erkennen von allgemeinen kontextfreien Sprachen können langsam sein.

Klassen von Grammatiken und Parsern

LL(k) Parser

- Eingabe wird von **links-nach-rechts** (erstes L) abgearbeitet.
- **Linksseitige Ableitung** (zweites L).
- Genannt *top-down*, *prädiktive* oder *voraussagende* Parser.

LR(k) Parser

- Eingabe wird von **links-nach-rechts** (L) abgearbeitet.
- **Rechtsseitige Ableitung** (R).
- Genannt *bottom up*, *shift-reduce* oder *schiebe-reduziere* Parser.

Der Wert **k** steht für die Anzahl von Token für die in der Eingabe **vorausgeschaut** werden muss um eine Entscheidung treffen zu können.

- LL(k). Welche nächste Produktion (rechten Seite) ist bei der linksseitigen Ableitung zu wählen.

Syntaxanalyse

Top-down oder LL-Syntaxanalyse.

- Baue den Ableitungsbaum von der Wurzel aus bis hinunter zu den Blättern auf.
- Berechne in jedem Schritt voraus welche Produktion zu verwenden ist um den aktuellen nichtterminalen Knoten des Ableitungsbaumes aufzuweiten (*expand*), indem die nächsten k Eingabesymbole betrachtet werden.

Beispiel

Grammatik
G = { {ID_LIST, ID_LIST_TAIL},
{`id`, `,`, `;`},
P,
ID_LIST }

Produktionen P
ID_LIST → `id` ID_LIST_TAIL
ID_LIST_TAIL → `,id` ID_LIST_TAIL
ID_LIST_TAIL → `;`$$

Quelltext
```
A, B, C;
```

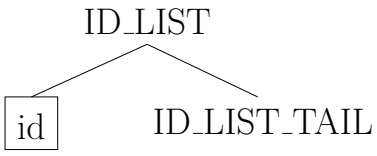
Satz (Tokenfolge)
`id,id,id;`$$

Ableitungsbaum

ID_LIST

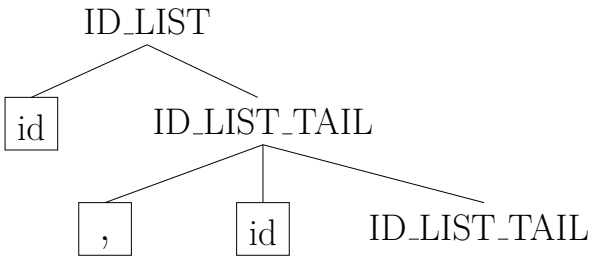
Satz (Tokenfolge)
`id`,id,id;$$

Ableitungsbaum



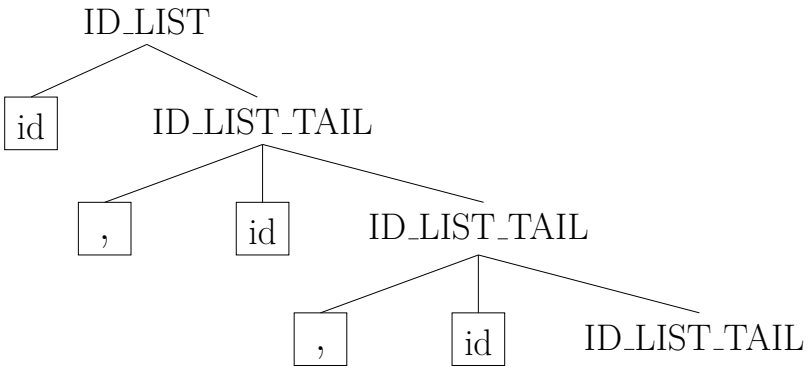
Satz (Tokenfolge)
~~id~~ `,` id,id;$$

Ableitungsbaum



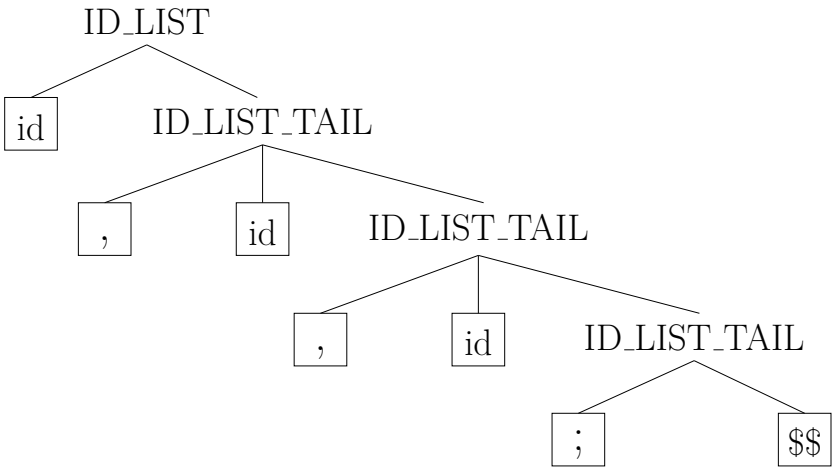
Satz (Tokenfolge)
~~id,id~~ `,` id;$$

Ableitungsbaum



Satz (Tokenfolge)
id , id , id ; $$

Ableitungsbaum



Ergebnis
id , id , id ; $$

Der Satz ist ein Wort der Sprache $L(G)$.

6.6.4 Rekursive LL-Parser

Rekursiver Abstieg

Rekursiver Abstieg ist ein Weg um LL (top-down) Parser zu implementieren

- Es ist einfach von Hand zu schreiben.
- Es wird kein Parsergenerator benötigt.

Jedes nichtterminale Symbol in der Grammatik hat einen Prozeduraufruf.

- Die nachste anzuwendende linksseitige Ableitung wird bestimmt (*predict*), indem nur die nächsten k Symbole angeschaut werden.

Beispiel

Grammatik $G := \{N, T, P, S\}$ einfacher arithmetischen Ausdrücke.

$T = \{+, *, \text{const}, \$\$\}$
$N = \{\text{PROG, EXPR, TERM, TTAIL, FACTOR, FTAIL}\}$
$S = \text{PROG}$

$$
\begin{array}{lll}
\text{P} := \{ & \text{PROG} & \rightarrow \text{EXPR \$\$} \\
 & \text{EXPR} & \rightarrow \text{TERM TTAIL} \\
 & \text{TERM} & \rightarrow \text{FACTOR FTAIL} \\
 & \text{TTAIL} & \rightarrow + \text{ TERM TTAIL} \mid \varepsilon \\
 & \text{FACTOR} & \rightarrow \text{const} \\
 & \text{FTAIL} & \rightarrow * \text{ FACTOR FTAIL} \mid \varepsilon \quad \}
\end{array}
$$

Programm

```
void error() {
  // no derivation for input
}

token current_token() {
  // return current token of input
}

void next_token() {
  // goto next token of input
}

void match(token expected) {
```

```
    if (expected == current_tocken())
      next_token();
    else
      error();
  }

  // start symbol ----------------------------------
  void PROG() {
    EXPR();
    match($$);
  }

  void EXPR() {
    TERM();
    TTAIL();
  }

  void TTAIL() {
    if (current_token() == +) {
      match(+);
      TERM();
      TTAIL();
    }
  }

  void TERM() {
    FACTOR();
    FTAIL();
  }

  void FTAIL() {
    if (current_token() == *) {
      match(*);
      FACTOR();
      FTAIL();
    }
  }

  void FACTOR() {
    match(const);
  }
```

LL(k)-Syntaxanalyse

Finde zu einer Eingabe von Terminalsymbolen (Tokens) passende Produktionen in einer Grammatik durch Herstellung von linksseitigen Ableitungen.

Für eine gegebene Menge von Produktionen für ein Nichtterminal,

$$X \to \alpha_1 | \ldots | \alpha_n$$

und einen gegebenen, linksseitigen Ableitungsschritt

$$\gamma X \delta \Rightarrow \gamma \alpha_i \delta$$

muss bestimmt werden welches α_i zu wählen ist, indem nur die nächsten k Eingabesymbole betrachtet werden.

Bemerkung.

Für eine gegebene Menge von linksseitigen Ableitungsschritten, ausgehend vom Startsymbol $S \Rightarrow \gamma X \delta$, wird die Satzform γ nur aus Terminalen/Tokens bestehen und repräsentiert den passenden Eingabeabschnitt zu den bisherigen Grammatikproduktionen.

LL-Syntaxanalyse, Linksrekursion

Linksrekursion. Folgender Grammatikausschnitt enthält eine linksrekursive Produktionen.

$$A \to A\alpha$$
$$A \to \beta$$

Wenn eine Grammatik linksrekursive Produktionen enthält, dann kann es für diese Grammatik keinen LL Parser geben.

- LL Parser können in eine Endlosschleife eintreten, wenn versucht wird eine linksseitige Ableitung mit so einer Grammatik vorzunehmen.

Linksrekursion kann durch Umschreiben der Grammatik vermieden werden.

$$A \to \beta A'$$
$$A' \to \alpha A' \,|\, \varepsilon$$

Nicht formale Rechtfertigung.

Linkrekursion.

$$A \to A\alpha$$
$$A \to \beta$$

Ableitungen

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \Rightarrow \ldots$$
$$\beta\alpha\alpha\alpha\ldots$$

Auflösung.

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \varepsilon$$

Ableitungen

$$A \Rightarrow \beta A' \Rightarrow \beta\alpha A' \Rightarrow \beta\alpha\alpha A' \Rightarrow \ldots$$
$$\beta\alpha\alpha\alpha\ldots$$

Beispiel

Produktionen mit Linksrekursion.

ID_LIST → ID_LIST_PREFIX `;`$$
ID_LIST_PREFIX → ID_LIST_PREFIX `,id`
ID_LIST_PREFIX → `id`

Umschreiben, neues Nichterminal ID_LIST_TAIL.

ID_LIST → ID_LIST_PREFIX `;`$$
ID_LIST_PREFIX → `id` ID_LIST_TAIL
ID_LIST_TAIL → `,id` ID_LIST_TAIL | ε

Vereinfachen, Nichterminal ID_LIST_PREFIX fällt weg.

ID_LIST → `id` ID_LIST_TAIL `;`$$
ID_LIST_TAIL → `,id` ID_LIST_TAIL | ε

LL-Syntaxanalyse, gemeinsame Präfixe

Gemeinsame Präfixe. Folgender Grammatikausschnitt enthält Produktionen mit gemeinsamen Präfixen.

$$A \to \alpha X$$
$$A \to \alpha Y$$

Wenn eine Grammatik Produktionen mit gemeinsamen Präfixen der Länge k enthält, dann kann es für diese Grammatik keinen LL(k) Parser geben. Denn der Parser kann nicht entscheiden mit welcher Produktion der nächste Ableitungsschritt vorgenommen werden soll.

Gemeinsame Präfixe können durch Umschreiben der Grammatik beseitigt werden.

$$A \to \alpha A'$$
$$A' \to X \mid Y$$

Beispiel

Produktionen mit gemeinsamen Präfixen.

STMT → `id :=` EXPR
STMT → `id (` ARGUMENT_LIST `)`

Umschreiben, neues Nichterminal STMT_LIST_TAIL.

STMT → `id` STMT_LIST_TAIL
STMT_LIST_TAIL → `:=` EXPR | `(` ARGUMENT_LIST `)`

Probleme mit der LL-Syntaxanalyse

Der Ausschluss von Linksrekursion und gemeinsamen Präfixen garantiert nicht, dass es für diese Grammatik einen LL(1)-Parser gibt.

Es gibt Algorithmen mit denen man für eine gegebene Grammatik ermitteln kann, ob sich ein LL(1)-Parser finden lässt.

Wenn man keinen LL(1)-Parser für eine Grammatik finden kann, dann muss man eine mächtigere Technik verwenden, z.B. LL(k)-Parser mit $k \geq 2$.

6.6.5 Nicht-rekursive LL-Parser

Literatur

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman.

Compilers - Principles, Techniques and Tools (Dragon Book),

Addison-Wesley.

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman.

Compiler, Prinzipien, Techniken und Werkzeuge

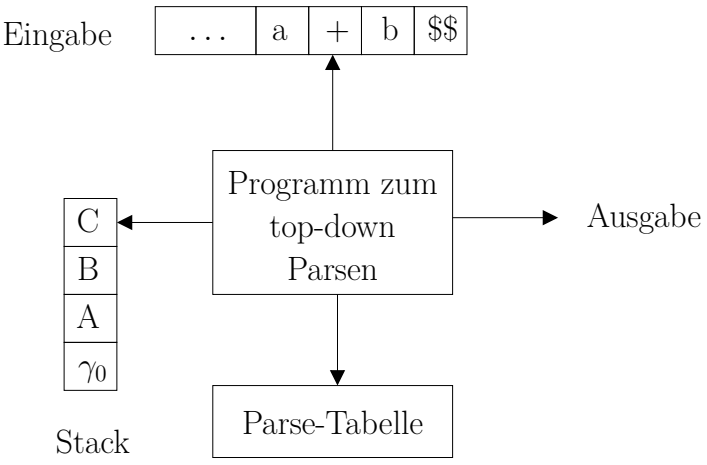
Pearson Studium.

Modell

Es ist möglich LL-Parser **nicht-rekursiv** zu implementieren.

Für die nicht-rekursive Implementation wird ein Stapel explizit verwaltet, anstatt die implizite Stapelverwaltung der rekursiven Aufrufe zu benutzen.

Für die Auswahl einer Produktion für ein zu ersetzendes Nichtterminal inspizieren nicht-rekursive Parser eine **Parse-Tabelle**.



Ein **tabellen-gesteuerter LL-Parser** besteht aus einem Eingabepuffer, einem Stapel, einer Parse-Tabelle und einem Ausgabestrom.

Der Eingabenpuffer enthält den Satz, der analysiert werden soll, abgeschlossen mit dem Symbol \$\$, das bei diesem Verfahren **nicht** zu den Terminalen der Grammatik gehört.

Der Stapel enthält eine Folge von Grammatiksymbolen (Terminale und Nichterminale) und eine Kennzeichnung γ_0 für das unterste Element, die nicht zu den Grammatiksymbolen gehört.

Zu Beginn enthält der Stapel γ_0 und darüber das Startsymbol der Grammatik.

Die Parse-Tabelle ist ein zweidimensionales Feld $M[A, a]$, wobei A ein Nichterminal und a ein Terminal der Grammatik oder \$\$ ist. Ein Eintrag der Parse-Tabelle enthält entweder eine Produktion $A \to \alpha$ oder ist leer.

Steuerung

Ein Programm mit folgenden Verhalten übernimmt die Steuerung des Parsers.

Das oberste Stapelsysmbol X und das aktuelle Eingabezeichen a werden inspiziert.

- $X = \gamma_0$, $a =$ \$\$. Der Parser stoppt und meldet Erfolg.
- $X = a$ ($X \neq \gamma_0$, $a \neq$ \$\$). Das oberste Element X wird vom Stapel entfernt und das nächste Zeichen der Eingabe wird zum aktuellen Eingabezeichen.
- X ist eine Nichterminal und der Eintrag $M[X, a]$ der Parse-Tabelle enthält genau eine Produktion. Das oberste Stapelelement X wird entfernt und die rechte Seite der Produktion wird Symbol für Symbol von rechts-nach-links auf den Stapel gelegt.

  Bemerkung. An dieser Stelle könnte beliebiger Code, z.B. zum Aufbauen des Parsenbaums, ausgeführt werden.
- In allen anderen Fällen stoppt der Parser und meldet einen Fehler.

Beispiel

Grammatik $G = (N, T, P, S)$.

Nichtterminale $N = \{E, E', F, T, T'\}$,

Terminal $T = \{\mathbf{id}, +, *, (,)\}$,

Startsymbol $S = E$,

Produktionen P wie folgt.

$$
\begin{array}{ll}
E & \to TE' \\
E' & \to +TE' \,|\, \varepsilon \\
T & \to FT' \\
T' & \to *FT' \,|\, \varepsilon \\
F & \to (\,E\,) \,|\, \mathbf{id}
\end{array}
$$

Parse-Tabelle

	id	+	*	(	)	$$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \mathbf{id}$			$F \to (\,E\,)$		

Ablauf für Eingabe $\mathbf{id} + \mathbf{id} * \mathbf{id}$.

Eingabe (erzeugt)	Stapel	Eingabe (Rest)	Produktion
	$E\ \gamma_0$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\ \$\$$	$E \to TE'$
	$TE'\ \gamma_0$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\ \$\$$	$T \to FT'$
	$FT'E'\ \gamma_0$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\ \$\$$	$F \to \mathbf{id}$
	$\mathbf{id}T'E'\ \gamma_0$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\ \$\$$	
$\mathbf{id}$	$T'E'\ \gamma_0$	$+\,\mathbf{id} * \mathbf{id}\ \$\$$	$T' \to \varepsilon$
$\mathbf{id}$	$E'\ \gamma_0$	$+\,\mathbf{id} * \mathbf{id}\ \$\$$	$E' \to +TE'$
$\mathbf{id}$	$+TE'\ \gamma_0$	$+\,\mathbf{id} * \mathbf{id}\ \$\$$	
$\mathbf{id} +$	$TE'\ \gamma_0$	$\mathbf{id} * \mathbf{id}\ \$\$$	$T \to FT'$
$\mathbf{id} +$	$FT'E'\ \gamma_0$	$\mathbf{id} * \mathbf{id}\ \$\$$	$F \to \mathbf{id}$
$\mathbf{id} +$	$\mathbf{id}T'E'\ \gamma_0$	$\mathbf{id} * \mathbf{id}\ \$\$$	
$\mathbf{id} + \mathbf{id}$	$T'E'\ \gamma_0$	$*\,\mathbf{id}\ \$\$$	$T' \to *FT'$
$\mathbf{id} + \mathbf{id}$	$*FT'E'\ \gamma_0$	$*\,\mathbf{id}\ \$\$$	
$\mathbf{id} + \mathbf{id} *$	$FT'E'\ \gamma_0$	$\mathbf{id}\ \$\$$	$F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}T'E'\ \gamma_0$	$\mathbf{id}\ \$\$$	
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\ \gamma_0$	$\$\$$	$T' \to \varepsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\ \gamma_0$	$\$\$$	$E' \to \varepsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	γ_0	$\$\$$	

FIRST und FOLLOW

Mit Hilfe einer Grammatik kann man Funktionen FIRST und FOLLOW definieren, die die Einträge für die Parse-Tabellen liefern.

Bemerkung

Die von FOLLOW gelieferten Symbole lassen sich als synchronisierende Symbole z.B. bei der Fehlersuche verwenden.

FIRST

Sei $G = (N, T, P, S)$ eine Grammatik, es gilt $\$\$ \notin T$.

FIRST(α) ist definiert für eine beliebige Folge von Grammatiksymbolen $\alpha \in (N \cup T)^*$ und liefert eine Teilmenge der Terminale vereinigt mit dem leeren Wort, $\text{FIRST}(\alpha) \subseteq \Big(T \cup \{\varepsilon\}\Big)$.

FIRST(α) liefert die Menge aller Terminale mit denen ein aus α abgeleiteter

Satz beginnen kann.

$$\text{FIRST}(\alpha) := \left\{ a \in T \mid \alpha \overset{G}{\Rightarrow} a\beta \text{ mit } \beta \in T^* \right\}$$

FIRST(α) enthält zusätzlich ε, wenn ε aus α abgeleitet werden kann, $\alpha \overset{G}{\Rightarrow} \varepsilon$.

FOLLOW

Sei $G = (N, T, P, S)$ eine Grammatik, es gilt \$\$ $\notin T$.

FOLLOW(A) ist definiert für jedes Nichtterminal $A \in N$ und liefert eine Teilmenge der Terminale vereinigt mit dem Endsymbol der Eingabe, $\text{FOLLOW}(A) \subseteq \left(T \cup \{\$\$\}\right)$.

FOLLOW(A) liefert die Menge aller Terminale, die in einer Satzform direkt rechts neben A stehen können.

$$\text{FOLLOW}(A) := \left\{ a \in T \mid S \overset{G}{\Rightarrow} \alpha A a \beta \text{ mit } \alpha, \beta \in (N \cup T)^* \} \right\}$$

Bemerkung

Zwischen A und a können während der Ableitung Grammatiksymbole gestanden haben, die aber verschwunden sind, weil ε aus ihnen abgeleitet wurde.

FOLLOW(A) enthält zusätzlich \$\$, wenn es eine Satzform gibt in der A das am weitesten rechts stehende Grammatiksymbol ist.

Berechnung von FIRST

Für alle Terminale $t \in T$ gilt $\text{FIRST}(t) := \{t\}$.

Für das leere Wort ε gilt $\text{FIRST}(\varepsilon) := \{\varepsilon\}$.

Für ein Nichtterminal $A \in N$ wird für jede Produktion $A \to \alpha$ die Menge FIRST(α) zu FIRST(A) hinzugefügt.

Bemerkung

Enthalten ist folgender Spezialfall.

Gibt es eine Produktion $A \to \varepsilon$, dann füge ε zu FIRST(A) hinzu.

FIRST(α) wird für $\alpha = \alpha_1\alpha_2 \dots \alpha_n$ mit $\alpha_i \in (N \cup T)$ wie folgt bestimmt.

- Zu FIRST(α) wird FIRST(α_1)\{ε} hinzugefügt.
- Zu FIRST(α) wird für $i = 2, \dots n$ die Menge FIRST(α_i)\{ε} hinzugefügt, wenn $\varepsilon \in$ FIRST(α_j) für alle $j = 1, \dots, i-1$, denn d.h. $\alpha_1 \dots \alpha_{i-1} \overset{G}{\Rightarrow} \varepsilon$.
- Zu FIRST(α) wird ε hinzugefügt, wenn ε in allen FIRST(α_1), ..., FIRST(α_n) enthalten ist.

Berechnung von FOLLOW

Für alle Nichterminale $A \in N$ werden die folgenden Regeln solange angewandt, bis keine FOLLOW-Menge mehr vergrößert werden kann.

- In FOLLOW(S) wird \$\$ aufgenommen, wobei S das Startsymbol und \$\$ die Endemarkierung der Eingabe ist.
- Wenn es eine Produktion $A \to \alpha B \beta$ gibt, wird jedes Element von FIRST(β) mit Ausnahme von ε in FOLLOW(B) aufgenommen.
- Wenn es Produktionen $A \to \alpha B$ gibt, dann wird jedes Element von FOLLOW(A) zu FOLLOW(B) hinzugefügt.
- Wenn es eine Produktion $A \to \alpha B \beta$ gibt und $\varepsilon \in$ FIRST(β) enthalten (d.h. $\beta \overset{G}{\Rightarrow} \varepsilon$), dann wird jedes Element von FOLLOW(A) zu FOLLOW(B) hinzugefügt.

Konstruktion der Parse-Tabelle

Angenommen $A \in N$ ist das zu ersetzende Nichterminal und $t \in T \cup \{\$\$\}$ das aktuelle Eingabezeichen.

Der Parser sucht eine Produktion $A \to \alpha$ durch deren rechte Seite A ersetzt werden kann.

- Das geht wenn $t \in$ FIRST(α) gilt.
- Das geht wenn $\alpha = \varepsilon$ oder $\alpha \overset{G}{\Rightarrow} \varepsilon$ und $t \in$ FOLLOW(A) gilt.

Zur Konstruktion der Parse-Tabelle führe für jede Produktion $A \to \alpha$ folgende Schritte durch.

- Trage für jedes Terminal $t \in \text{FIRST}(\alpha)$ die Produktion $A \to \alpha$ an der Stelle $M[A, t]$ ein.
- Gilt $\varepsilon \in \text{FIRST}(\alpha)$ trage $A \to \alpha$ für jedes Terminal $t \in \text{FOLLOW}(A)$ an der Stelle $M[A, t]$ ein.
- Gilt $\varepsilon \in \text{FIRST}(\alpha)$ und \$\$ $\in \text{FOLLOW}(A)$ trage $A \to \alpha$ an der Stelle $M[A, \$\$]$ ein.

Parse-Tabellen

Mit Hilfe von FIRST und FOLLOW läßt sich grundsätzlich für jede Grammatik eine Parse-Tabelle erstellen.

Es gibt Grammatiken, bei denen die Parse-Tabelle mehrere Einträge pro Zelle enthält. Z.B. wenn die Grammatik linksrekursiv ist oder es gemeinsame Präfixe gibt.

Eine **LL(1)-Grammatik** ist eine Grammatik deren Parse-Tabelle keine Mehrfacheinträge enthält, d.h. es gibt einen LL(1)-Parser, der genau die Sätze dieser Grammatik erkennt. Z.B. den nicht-rekursiven Parser mit der aus FIRST und FOLLOW erzeugten Parse-Tabelle.

Beispiel

Grammatik $G = (N, T, P, S)$

$N = \{E, E', F, T, T'\}, \quad T = \{\mathbf{id}, +, *, (,)\}, \quad S = E$

$P = \{$

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \mid \varepsilon \\ T &\to FT' \\ T' &\to *FT' \mid \varepsilon \\ F &\to (\,E\,) \mid \mathbf{id} \end{aligned}$$

$\}$

Ziel. Für jede Produktion Bestimmung der FIRST-Menge der rechten Seite.

$\text{FIRST}(\varepsilon) = \{\varepsilon\}$
$\text{FIRST}(a) = a$ für alle $a \in T$

$\text{FIRST}(\,(\,E\,)\,) = \{\,(\,\}$
$\text{FIRST}(*FT') = \{*\}$
$\text{FIRST}(+TE') = \{+\}$
$\text{FIRST}(*FT') = \{*\}$

$\text{FIRST}(FT')$?
- $\text{FIRST}(FT') \supset \text{FIRST}(F)$
- $\text{FIRST}(F)$?
  - $\text{FIRST}(F) \supset \text{FIRST}(\,(\,E\,)\,) = \{\,(\,\}$
  - $\text{FIRST}(F) \supset \text{FIRST}(\mathbf{id}) = \{\mathbf{id}\}$
  - es gibt keine weitere Produktion $F \to \ldots$
  - d.h. $\text{FIRST}(F) = \{\,(\,, \mathbf{id}\,\}$
- $\text{FIRST}(FT') \supset \text{FIRST}(F) = \{\,(\,, \mathbf{id}\,\}$
- $\varepsilon \notin \text{FIRST}(F)$
- d.h. $\text{FIRST}(FT') = \{\,(\,, \mathbf{id}\,\}$

$\text{FIRST}(TE')$?
- $\text{FIRST}(TE') \supset \text{FIRST}(T)$
- $\text{FIRST}(T)$?
  - $\text{FIRST}(T) \supset \text{FIRST}(FT') = \{\,(\,, \mathbf{id}\,\}$
  - es gibt keine weitere Produktion $T \to \ldots$
  - d.h. $\text{FIRST}(T) = \{\,(\,, \mathbf{id}\,\}$
- $\text{FIRST}(TE') \supset \text{FIRST}(T) = \{\,(\,, \mathbf{id}\,\}$
- $\varepsilon \notin \text{FIRST}(T)$
- d.h. $\text{FIRST}(TE') = \{\,(\,, \mathbf{id}\,\}$

Ziel. Für jede Produktion $A \to \alpha$ mit $\varepsilon \in \text{FIRST}(\alpha)$ Bestimmung von FOLLOW(A), d.h. gesucht ist FOLLOW(E') und FOLLOW(T').

FOLLOW(E')?
- $E' \to +TE'$
- d.h. FOLLOW(E') $\supset$ FOLLOW(E')
- $E \to TE'$
- d.h. FOLLOW(E') $\supset$ FOLLOW(E)
- FOLLOW(E)?
  - $S = E$
  - d.h. FOLLOW(E) $\ni$ \$\$
  - $F \to (\ E\ )$
  - d.h. FOLLOW(E) $\supset$ FIRST( ))$\backslash\{\varepsilon\}$ = {) }
  - keine weitere rechte Seite $\ldots E \ldots$
  - d.h. FOLLOW(E) = {) , \$\$ }
- FOLLOW(E') $\supset$ FOLLOW(E) = {) , \$\$ }
- keine weitere rechte Seite $\ldots E' \ldots$
- d.h. FOLLOW(E') = {) , \$\$ }

FOLLOW(T')?
- $T' \to *FT'$
  - d.h. FOLLOW(T') $\supset$ FOLLOW(T')
- $T \to FT'$
- d.h. FOLLOW(T') $\supset$ FOLLOW(T)
- FOLLOW(T)?
  - $E \to TE'$
  - d.h. FOLLOW(T) $\supset$ FIRST(E')$\backslash\{\varepsilon\}$
  - FIRST(E')?
    - FIRST(E') = $\{+, \varepsilon\}$
  - FOLLOW(T) $\supset$ FIRST(E')$\backslash\{\varepsilon\}$ = $\{+\}$
  - $E \to TE'$ und $\varepsilon \in$ FIRST(E')
  - d.h. FOLLOW(T) $\supset$ FOLLOW(E) = {) , \$\$ }
  - $E' \to +TE'$
  - genauso, weil FOLLOW(E') = FOLLOW(E)
  - d.h. FOLLOW(T) = {+,) , \$\$ }
- FOLLOW(T') $\supset$ FOLLOW(T) = {+,) , \$\$ }
- keine weitere rechte Seite $\ldots T' \ldots$
- d.h. FOLLOW(T') = {+,) , \$\$ }

Ziel. Konstruiere die Parse-Tabelle wie folgt.

- Für $t \in \text{FIRST}(\alpha)$ setze $M[A, t] := A \to \alpha$.
- Gilt $\varepsilon \in \text{FIRST}(\alpha)$ setze $M[A, t] := A \to \alpha$ für jedes $t \in \text{FOLLOW}(A)$.

	id	$+$	$*$	$($	$)$	\$\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \mathbf{id}$			$F \to (\,E\,)$		

7 Logik

7.1 Aussagenlogik

7.1.1 Literatur

Literatur

Martin Kreuzer, Stefan Kühling.

Logik für Informatiker,

Pearson Studium.

Die folgenden Abschnitte wurden großteils aus dem Buch *Logik für Informatiker* übernommen.

7.1.2 Grundlagen

Syntax der Aussagenlogik

Definition 7.1. Formel

1. Eine **atomare Formel** ist von der Form A_i mit $i \in \mathbb{N}$, d.h. atomare Formeln sind nur die einfachen Aussagen.
2. Eine beliebige Formel entsteht induktiv aus atomaren Formeln, wobei die folgenden Schritte erlaubt sind:
   - Jede atomare Formel ist eine Formel.
   - Sind F, G zwei Formeln, so sind auch $F \wedge G$ sowie $F \vee G$ Formeln.
   - Für jede Formel F ist auch $\neg F$ eine Formel.

$F \wedge G$ nennt man **Konjunktion** von F und G. $F \vee G$ nennt man **Disjunktion** von F und G. $\neg F$ nennt man **Negation** von F.

Notationen

Für Aussagen verwenden wir statt $A_0, A_1, A_2, \ldots$ auch $A, B, C, \ldots$. Seien nun Formeln $F_1, F_2, F_3, \ldots$ gegeben:

1. Für $(\neg F_1 \vee F_2)$ schreiben wir auch $(F_1 \Rightarrow F_2)$. Wir nennen $F_1 \Rightarrow F_2$ auch eine **Folgerung** oder **Implikation**.
2. Für $(F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2)$ schreiben wir auch $(F_1 \Leftrightarrow F_2)$. Wir nennen $F_1 \Leftrightarrow F_2$ eine **Äquivalenz**.
3. Für $(\cdots((F_1 \vee F_2) \vee F_3) \vee \cdots \vee F_n)$ schreiben wir auch $\bigvee_{i=1}^{n} F_i$.
4. Für $(\cdots((F_1 \wedge F_2) \wedge F_3) \wedge \cdots \wedge F_n)$ schreiben wir auch $\bigwedge_{i=1}^{n} F_i$.

Semantik der Aussagenlogik (1/2)

Definition 7.2. Belegungen

1. Die Elemente der Menge $\{wahr, falsch\}$ heißen die **Wahrheitswerte**. Wir schreiben auch 1 statt $wahr$ und 0 statt $falsch$.

2. Sei M eine Menge von atomaren Formeln. Eine **Belegung** von M ist eine Abbildung
$$\alpha : M \to \{0, 1\}$$

3. Sei $\widehat{M}$ die Menge aller Formeln, die mit Hilfe der atomaren Formeln in M gebildet werden können, und sei $\alpha : M \to \{0, 1\}$ eine Belegung. Dann erweitern wir α zu einer Abbildung $\widehat{\alpha} : \widehat{M} \to \{0, 1\}$ gemäß den folgenden Vorschriften.

Semantik der Aussagenlogik (2/2)

Für $\widehat{\alpha} : \widehat{M} \to \{0, 1\}$ gilt:

1. Für atomare Formeln $A \in M$ gilt $\widehat{\alpha}(A) = \alpha(A)$.
2. Für Formeln $F, G \in \widehat{M}$ gilt:
$$\widehat{\alpha}(F \wedge G) = \begin{cases} 1, & \text{falls } \widehat{\alpha}(F) = 1 \text{ und } \widehat{\alpha}(G) = 1, \\ 0, & \text{sonst.} \end{cases}$$
3. Für Formeln $F, G \in \widehat{M}$ gilt:
$$\widehat{\alpha}(F \vee G) = \begin{cases} 1, & \text{falls } \widehat{\alpha}(F) = 1 \text{ oder } \widehat{\alpha}(G) = 1, \\ 0, & \text{sonst.} \end{cases}$$
4. Für Formel $F \in \widehat{M}$ gilt:
$$\widehat{\alpha}(\neg F) = \begin{cases} 1, & \text{falls } \widehat{\alpha}(F) = 0 \\ 0, & \text{sonst.} \end{cases}$$

Im Folgenden schreiben wir der Einfachheit halber α statt $\widehat{\alpha}$. Ist eine Belegung der in einer Formel vorkommenden Aussagensymbole gegeben, so ist der Wahrheitswert der Formel gemäß dieser Definitionen leicht zu ermitteln.

Modelle und Erfüllbarkeit

Definition 7.3. Sei F eine aussagenlogische Formel und sei $\alpha : M \to \{0, 1\}$ eine Belegung, dann:

1. Sind alle in F vorkommenden atomaren Formeln in M enthalten, so heißt α zu F **passend**.

2. Ist α zu F passend und gilt $\alpha(F) = 1$, so schreiben wir $\alpha \models F$. Wir sagen, dass F unter der Belegung α gilt und nennen α ein **Modell** für F.

3. Ist $\mathcal{F}$ eine Menge aussagenlogischer Formeln, so heißt α ein **Modell** für $\mathcal{F}$, wenn für alle $F \in \mathcal{F}$ gilt: $\alpha \models F$. In diesem Fall schreiben wir $\alpha \models \mathcal{F}$.

4. Eine Menge $\mathcal{F}$ von Formeln heißt **erfüllbar**, falls $\mathcal{F}$ mindestens ein Modell besitzt. Ansonsten heißt $\mathcal{F}$ **unerfüllbar**.

5. Eine Formel F heißt **allgemein gültig** oder auch **Tautologie**, wenn jede zu F passende Belegung ein Modell für F ist.

Äquivalenzen der Aussagenlogik

Theorem 7.4. *Die fundamentalen Äquivalenzen der Aussagenlogik Für aussagenlogische Formeln F, G, H gelten die folgenden Äquivalenzen:*

1. $(F \vee F) \equiv F$, *sowie* $(F \wedge F) \equiv F$ ***(Idempotenz)***
2. $(F \vee G) \equiv (G \vee F)$, *sowie* $(F \wedge G) \equiv (G \wedge F)$ ***(Kommutativität)***
3. $((F \vee G) \vee H) \equiv (F \vee (G \vee H))$, *sowie* $((F \wedge G) \wedge H) \equiv (F \wedge (G \wedge H))$ ***(Assoziativität)***
4. $(F \vee (F \wedge G)) \equiv F$, *sowie* $(F \wedge (F \vee G)) \equiv F$ ***(Absorption)***
5. $(F \vee (G \wedge H)) \equiv ((F \vee G) \wedge (F \vee H))$ ***(Distributivität)***
6. $(F \wedge (G \vee H)) \equiv ((F \wedge G) \vee (F \wedge H))$ ***(Distributivität)***
7. $\neg\neg F \equiv F$ ***(Doppelnegation)***
8. $\neg(F \wedge G) \equiv (\neg F \vee \neg G)$, *sowie* $\neg(F \vee G) \equiv (\neg F \wedge \neg G)$ ***(de Morgansche Regeln)***
9. *Ist F Tautologie, so gilt* $(F \vee G) \equiv F$, *sowie* $(F \wedge G) \equiv G$ ***(Tautologieregeln)***
10. *Ist F unerfüllbar, so gilt* $(F \vee G) \equiv G$, *sowie* $(F \wedge G) \equiv F$ ***(Unerfüllbarkeitsregeln)***

Normalformen

Definition 7.5. Ein **Literal** ist eine atomare Formel oder die Negation einer atomaren Formel. Im ersten Fall sprechen wir von einem **positiven Literal**. im zweiten Fall von einem **negativem Literal**.

1. Eine Formel F ist in **konjunktiver Normalform (KNF)**, falls sie eine Konjunktion von Disjunktionen von Literalen ist. Mit anderen Worten, es muss Literale L_{ij} geben, so dass F von folgender Form ist:

$$F = (L_{11} \vee \cdots \vee L_{1m_1}) \wedge \cdots \ldots (L_{n1} \vee \cdots \vee L_{nm_n})$$

2. Eine Formel F ist in **disjunktiver Normalform (DNF)**, falls sie eine Disjunktion von Konjunktionen von Literalen ist, d.h. falls F von folgender Form ist:

$$F = (L_{11} \wedge \cdots \wedge L_{1m_1}) \vee \cdots \vee (L_{n1} \wedge \cdots \wedge L_{nm_n})$$

Algorithmus Erzeugung KNF

Definition 7.6. Gegeben sei eine Formel F. Führe die folgenden Schritte durch:

1. Eliminiere $\Rightarrow$ und $\Leftrightarrow$ mittels ihrer Definitionen.
2. Ersetze in F jede Teilformel der Form $\neg\neg G$ durch G.
3. Ersetze in F jede Teilformel der Form $\neg(G \wedge H)$ durch $(\neg G \vee \neg H)$. Entsteht hierdurch eine Teilformel der Form $\neg\neg K$, so wende Schritt 2) an.
4. Ersetze in F jede Teilformel der Form $\neg(G \vee H)$ durch $(\neg G \wedge \neg H)$. Entsteht hierdurch eine Teilformel der Form $\neg\neg K$, so wende Schritt 2) an.
5. Wiederhole die Schritte 3) und 4) so oft wie möglich.
6. Ersetze in F jede Teilformel der Form $(G \vee (H \wedge I))$ durch $((G \vee H) \wedge (G \vee I))$.
7. Ersetze in F jede Teilformel der Form $((G \wedge H) \vee I)$ durch $((G \vee I) \wedge (H \vee I))$.
8. Wiederhole die Schritte 6) und 7) so oft wie möglich.

Die resultierende Formel ist dann in KNF.

KNF zu 3-KNF

Definition 7.7. Gegeben sei eine Formel F in KNF $F = \bigwedge_i (L_{i1} \vee L_{i2} \vee \ldots \vee L_{i,m_i})$. Führe die folgenden Schritte durch um eine erfüllbarkeitsäquivalente Formel zu bekommen, in der in jeder Disjunktion maximal drei Literale stehen.

1. Sei $F = K_1 \wedge K_2 \wedge \ldots \wedge K_n$ eine Formel mit n Disjunktionen.
2. Betrachte nacheinander alle Disjunktionen K_i mit $1 \leq i \leq n$.
3. Solange die Disjunktion noch mehr als drei Literale enthält:
   - Generiere neue Variable X.
   - Erstelle neue Disjunktion $K_i' = (L_{i1} \vee L_{i2} \vee X)$.
   - Ersetze $K_i = (\neg X \vee L_{i3} \vee \ldots \vee L_{im_i})$.

Die resultierende Formel ist dann in 3-KNF.

Definition 7.8. Gegeben sei eine aussagenlogische Formel F in KNF mit Literalen L_{ij}.

$$F = (L_{11} \vee L_{12} \vee \ldots \vee L_{1\,m_1}) \wedge \ldots \wedge (L_{n1} \vee L_{n2} \vee \ldots \vee L_{n\,m_n})$$

Die **Klausel** $K_1, K_2, \ldots, K_n$ von F sind die Mengen der Literale der einzelnen Disjunktionen.

$$\begin{aligned} K_1 &= \{L_{11}, L_{12}, \ldots, L_{1\,m_1}\} \\ K_2 &= \{L_{21}, L_{22}, \ldots, L_{2\,m_2}\} \\ &\ldots \\ K_n &= \{L_{n1}, L_{n2}, \ldots, L_{n\,m_n}\} \end{aligned}$$

Die F zugeordnete **Klauselmenge** $\mathcal{K}(F)$ ist die Menge der Klauseln von F.

$$\mathcal{K}(F) = \{K_1, K_2, \ldots, K_n\}$$

Observation 7.9. *Verschiedene Formeln in KNF können dieselbe Klauselmenge haben.*

Aufgrund von Idempotenz und Kommutativität sind Formel mit derselben Klauselmenge äquivalent.

Example 7.10. Die folgenden Formeln haben die Klauselmenge $\{\{A\}, \{B, \neg C\}\}$ und sind nicht identisch, aber äquivalent.

$$F = (A \vee A) \wedge (B \vee \neg C)$$
$$G = A \wedge (\neg C \vee B)$$
$$H = A \wedge (B \vee \neg C \vee B)$$

7.1.3 Das Resolutionskalkül der Aussagenlogik

Lemma 7.11. *Sei F eine aussagenlogische Formel und $\mathcal{K}(F)$ ist die zugeordnete Klauselmenge.*

Dann ist F äquivalent zu jeder Formel G, für die gilt $\mathcal{K}(G) = \mathcal{K}(F) \cup \{R\}$, wobei R eine Resolvente zweier Klausel $K_1, K_2 \in \mathcal{K}(F)$ ist.

Beweis. Sei $\alpha : M \mapsto \{0, 1\}$ eine zu F passende Belegung (d.h. alle atomaren Formeln erhalten einen Wahrheitswert), dann ist α auch eine passende Belegung für G.

Fall 1

Es gilt $\alpha(F) = 0$, dann gibt es eine Klausel $K = \{L_1, \ldots, L_m\}$ in $\mathcal{K}(F)$, für die gilt $\alpha(L_i) = 0$ für alle $i \in \{1, \ldots m\}$.

Da $\mathcal{K}(F) \subset \mathcal{K}(G)$ ist die Klausel K, aus der $\alpha(F) = 0$ folgt, auch eine Klausel von G und daraus folgt $\alpha(G) = 0$.

Fall 2

Es gilt $\alpha(F) = 1$, dann gibt es in jeder Klausel $K_1, \ldots, K_n$ mindestens ein $L_i \in K_i$ mit $\alpha(L_i) = 1$.

Angenommen $L \in K_1$ und $\neg L \in K_2$. Es gilt $\alpha(L) \neq \alpha(\neg L)$, daraus folgt entweder in K_1 oder in K_2 gibt es ein Literal $L' \neq L$ mit $\alpha(L') = 1$.

$R = K_1 \backslash \{L\} \cup K_2 \backslash \{\neg L\}$, d.h. $L' \in R$ mit $\alpha(L') = 1$.

Aus $\mathcal{K}(G) = \mathcal{K}(F) \cup \{R\}$ folgt $\alpha(G) = 1$. □

Example 7.12. $F \equiv (P \vee S) \wedge (\neg P \vee \neg S) \wedge (\neg S \vee A) \wedge (\neg A \vee P) \wedge P \wedge S$

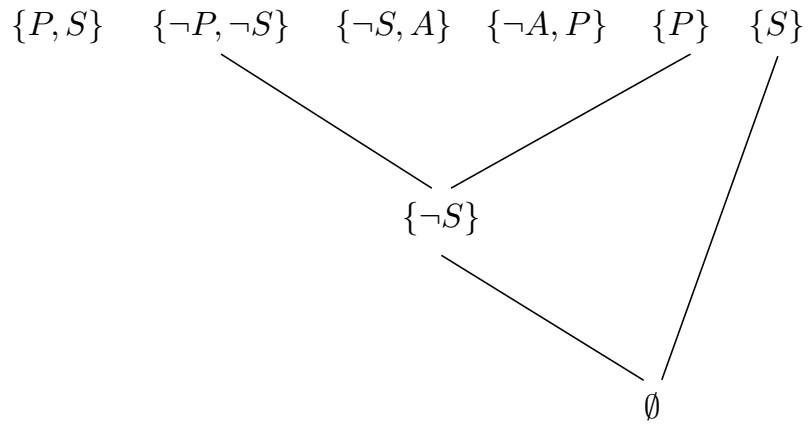
Klauselmenge

$$\mathcal{K}(F) = \Big\{\{P, S\}, \{\neg P, \neg S\}, \{\neg S, A\}, \{\neg A, P\}, \{P\}, \{S\}\Big\}$$

Resolventenmengen

$$
\begin{aligned}
\mathrm{Res}^0(\mathcal{K}(F)) =& \mathcal{K}(F) \\
\mathrm{Res}^1(\mathcal{K}(F)) =& \mathrm{Res}^0(\mathcal{K}(F)) \cup \\
& \Big\{\{P, \neg P\}, \{P, A\}, \{\neg A, \neg S\}, \{S, \neg S\}, \{\neg S\}, \{\neg P\}, \{P, \neg S\}, \{A\}\Big\} \\
\mathrm{Res}^2(\mathcal{K}(F)) =& \mathrm{Res}^1(\mathcal{K}(F)) \ \cup \Big\{\{\neg A\}, \emptyset, \ldots\Big\}
\end{aligned}
$$

Example 7.13. Das Beispiel grafisch.



7.1.4 KNF/DNF Nachtrag

Sei F eine aussagenlogische Formel in **KNF** mit Literalen L_{ij}.

$$F = (L_{11} \vee L_{12} \vee \ldots \vee L_{1\,m_1}) \wedge (L_{21} \vee L_{22} \vee \ldots \vee L_{2\,m_2}) \wedge \ldots \wedge (L_{n1} \vee L_{n2} \vee \ldots \vee L_{n\,m_n})$$

F ist eine **Tautologie**, wenn es in **jeder** Disjunktion (Klausel) Literale J, K gibt, für die gilt $J = \neg K$.

Sei G eine aussagenlogische Formel in **DNF** mit Literalen L_{ij}.

$$G = (L_{11} \wedge L_{12} \wedge \ldots \wedge L_{1\,m_1}) \vee (L_{21} \wedge L_{22} \wedge \ldots \wedge L_{2\,m_2}) \vee \ldots \vee (L_{n1} \wedge L_{n2} \wedge \ldots \wedge L_{n\,m_n})$$

G ist genau dann **unerfüllbar**, wenn es in **jeder** Konjuktion Literale J, K gibt, für die gilt $J = \neg K$.

Sowohl DNF als auch KNF werden im schlechtesten Fall (*worst case*) exponentiell groß, d.h. es gibt in der Anzahl der atomaren Formeln exponentiell viele Konjunktionen/Disjunktionen.

Beispiel

Seien $A_1, \ldots, A_n$ atomare Formeln, dann ist die n-äre **Paritätsfunktion** wie folgt definiert.

$$\bigoplus_{i=1}^{n} A_i = (\ldots (A_1 \oplus A_2) \oplus A_3) \ldots) \oplus A_n)$$

Seien α eine passende Belegung für $\{A_1, \ldots, A_n\}$, dann gilt Folgendes.

$$\alpha\left(\bigoplus_{i=1}^{n} A_i\right) = \begin{cases} 1 & \text{Anzahl } i \in \{1, \ldots, n\} \text{ mit } \alpha(A_i) = 1 \text{ ist ungerade} \\ 0 & \text{sonst} \end{cases}$$

Die Paritätsfunktion $\bigoplus_{i=1}^{n} A_i$ kann mit einer Formel, die linear in n ist (mit höchstens $2n$ Literalen), dargestellt werden, denn es gilt

$$A \oplus B = (A \vee B) \wedge (\neg A \vee \neg B) \ .$$

Die Paritätsfunktion hat dargestellt in DNF/KNF jeweils 2^{n-1} viele Konjunktionen/Disjunktionen (Klauseln).

A	B	C	$A \oplus B \oplus C$	A	B	C	$A \oplus B \oplus C$
0	0	0	0	1	0	0	1
0	0	1	1	1	0	1	0
0	1	0	1	1	1	0	0
0	1	1	0	1	1	1	1

DNF, Konstruktion aus den Zeilen für die $A \oplus B \oplus C = 1$ gilt.

$$A \oplus B \oplus C = (\neg A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge \neg C) \vee (A \wedge B \wedge C)$$

KNF, Konstruktion aus den Zeilen für die $A \oplus B \oplus C = 0$ gilt.

$$\begin{aligned} A \oplus B \oplus C =& \neg\Big(\neg A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge \neg C)\Big) \\ =& (A \vee B \vee C) \wedge (A \vee \neg B \vee \neg C) \wedge (\neg A \vee B \vee \neg C) \wedge (\neg A \vee \neg B \vee C) \end{aligned}$$

In DNF/KNF können keine Konjuktionen/Disjuktionen verschmolzen werden, weil für die Bestimmung der Parität immer alle Literale ausgewertet werden müssen.

Definition 7.14. Eine Formel F in KNF ist eine Hornformel, falls jede Disjunktion/Klausel maximal ein positives Literal enthält. Jede Klausel K kann dadurch als eine Implikation $T \to H$ gesehen werden. Dabei ist T eine Konjunktion von Atomen oder $T = wahr$ und H ist ein Atom oder $H = \bot$:

$$\begin{aligned} (\neg A \vee \neg B \vee C) &\equiv (A \wedge B \to C) \\ (\neg A \vee \neg B) &\equiv (A \wedge B) \to \bot \\ A &\equiv (wahr \to A) \end{aligned}$$

Definition 7.15. Gegeben eine Hornformel F. Wir prüfen die Erfüllbarkeit mit folgendem Algorithmus, welcher gleichzeitig ein Modell konstruiert:

1. Setze für alle Atome A_i in F die Belegung initial auf $\alpha(A_i) = 0$.
2. Solange in F Klauseln der Art $(T \to H)$ exisiteren mit $\alpha(T) = wahr$ und $\alpha(H) = 0$:
   - Falls $H = \bot$ gebe **unerfüllbar** aus.
   - Sonst setze $\alpha(H) = 1$

3. Gebe **erfüllbar** aus und $\alpha \models F$.

$F = (A_1) \wedge (\neg A_1 \vee A_2) \wedge (\neg A_1 \vee \neg A_3 \vee A_4) \wedge (\neg A_1 \vee \neg A_4 \vee A_5) \wedge (\neg A_2 \vee \neg A_4)$

Example 7.16.

$$(1 \to A_1) \wedge (A_1 \to A_2) \wedge (A_1 \wedge A_3 \to A_4) \wedge (A_1 \wedge A_4 \to A_5) \wedge (A_2 \wedge A_4 \to \bot)$$

setze $\alpha(A_1) = 1$.

$$(1 \to A_2) \wedge (1 \wedge A_3 \to A_4) \wedge (1 \wedge A_4 \to A_5) \wedge (A_2 \wedge A_4 \to \bot)$$

setze $\alpha(A_2) = 1$.

$$(1 \wedge A_3 \to A_4) \wedge (1 \wedge A_4 \to A_5) \wedge (1 \wedge A_4 \to \bot)$$

setze alle übrigen $\alpha(A_i) = 0$ und die Formel ist erfüllbar.

- Das Lösen von Formeln in 2-KNF (oder auch das 2-SAT Problem) ist effizient (polynomiell). **Idee:** In jeder Klausel muss mindestens eines der beiden Literale mit wahr belegt werden. Ist eines der Literale falsch, dann ist implizit, dass für eine erfüllbare Belegung das andere wahr sein muss. Diese Implikationen können in einem gerichteten Graphen modelliert werden. Wenn dieser Graph Kreise enthält, welche sowohl A und $\neg A$ enthalten, dann ist die Formel nicht erfüllbar, ansonsten ist die formel erfüllbar und die Pfade durch den Graphen implizieren ein Modell. Dies kann effizient durch Tiefensuche realisiert werden.
- Das Lösen von Formeln in 3-KNF (oder auch das 3-SAT Problem) ist NP-vollständig (Verweis auf Informatik 3 - Algorithmen und Datenstrukturen).

Mit dieser Art von Algorithmus werden SAT Probleme, also das Erfüllbarkeitsproblem für Formeln F in KNF gelöst.

1. Gegeben sei eine Formel F in KNF als Input.
2. Solange es Klauseln mit einem einzigen Literal $C = \{L\}$ als Klauselmenge gibt:
   - Setze L auf wahr.
   - Klauseln mit L werden entfernt, Literale $\neg L$ werden entfernt.

3. Solange es Literale L gibt, welche nur in einer Form (negativ/positiv) in F vorkommen:
   - Setze L auf wahr.
   - Klauseln mit L werden entfernt, Literale $\neg L$ werden entfernt.
4. Falls F keine Klauseln mehr enthält, dann gebe für F wahr zurück.
5. Falls F eine leere Klausel enthält, dann gebe für F falsch zurück.
6. Wähle ein Literal L.
7. Führe die Prozedur rekursiv mit $F \wedge \{L\}$ und $F \wedge \{\neg L\}$ und teste ob eines davon wahr ist.

7.2 Prädikatenlogik

7.2.1 Einleitung

Logik handelt vom Umgang mit **Aussagen**.

Aussagen sind sprachliche Gebilde, die entweder *wahr* oder *falsch* sind.

Beispiele

Aussagen

1. Es regnet.
2. Der Wal ist ein Fisch.
3. Die Variable x hat den Wert -1.

Keine Aussagen

1. Herzlichen Glückwunsch.
2. Wo ist der Bahnhof.

Die **Prädikatenlogik** ist eine Erweiterung der Aussagenlogik um folgende wesentliche Konzepte.

- Aussagen gelten für **Objekte** (Individuen) eines zu modellierenden **Individuenbereichs**.
- **Variablen** sind Platzhalter für Objekte des Individuenbereichs.
- **Formel** entstehen durch Variablen in Aussagen.
- Formeln werden kombiniert durch **Junktoren** und eingeschränkt durch **Quantoren**.
- Objekte können durch **Terme** bezeichnet werden.

Die heute gebräuchliche Syntax der Prädikatenlogik wurde 1889 eingeführt von Giuseppe Peano in *Arithmetices principia, nova methodo exposita* (Prinzipien der Arithmetik, nach einer neuen Methode vorgestellt).

Die Ursprünge gehen zurück auf eine Arbeit von Gottlob Frege *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens* von 1879.

Der Grundgedanke der Prädikatenlogik ist die Einführung **parametrisierter Aussagen**, in denen **Variablen** durch beliebige Objekte des zu modellierenden Individuenbereichs ersetzt werden können.

N ist eine Primzahl

Variable N, Individuenbereich die natürlichen Zahlen $\mathbb{N}$

X ist kleiner als Y

Variablen X und Y, Individuenbereich die reelen Zahlen $\mathbb{R}$

Erst nach dem Ersetzen der Variablen durch **konkrete** Objekte aus dem Individuenbereich kann ein Wahrheitswert festgestellt werden.

N ist eine Primzahl

$N = 4$, daraus folgt die Aussage ist *falsch*

$N = 5$, daraus folgt die Aussage ist *wahr*

7.2.2 Formeln

In der Prädikatenlogik heissen (parametrisierte) Aussagen **Formeln**.

- **Atomare Formeln** lassen sich nicht weiter in Teilformeln zerlegen.
- **Zusammengesetzte Formeln** sind aus Teilformeln aufgebaut.

Formale Notation für atomare Formeln, ist ein **Prädikat** gefolgt von einer Parameterliste.

Beispiel

$$\text{istPrimzahl}(N)$$

istPrimzahl ist das Prädikat, N ist die Variable.

Zweistellige Prädikate dürfen auch in **Infixnotation** verwendet werden.

$$\text{ist\_kleiner\_als}(X, Y) \quad \text{bzw.} \quad X \ \text{ist\_kleiner\_als} \ Y \quad \text{bzw.} \quad X < Y$$

ist_kleiner_als und $<$ sind Prädikate, X und Y sind Variablen.

Formeln lassen sich mit Hilfe der **Junktoren** ($\neg$, $\wedge$, $\vee$, $\oplus$, $\Rightarrow$, $\Leftrightarrow$), der Operatoren aus der Aussagenlogik, zu neuen Formeln kombinieren.

$$\text{istPrimzahl}(N) \ \wedge \ (N < 10)$$

$N = 4$, daraus folgt die Aussage ist *falsch* $N = 5$, daraus folgt die Aussage ist *wahr* $N = 11$, daraus folgt die Aussage ist *falsch*

Erinnerung

Die **Implikation** $x \Rightarrow y$ ist die Aussage *wenn* x*, dann* y oder x *impliziert* y.

Es gilt folgende Wahrheitstafel.

x	y	$x \Rightarrow y$
falsch	*falsch*	*wahr*
falsch	*wahr*	*wahr*
wahr	*falsch*	*falsch*
wahr	*wahr*	*wahr*

Die **Äquivalenz** $x \Leftrightarrow y$ ist die Aussage *wenn* x*, genau dann* y oder x *äquivalent* y.

Es gilt folgende Wahrheitstafel.

x	y	$x \Leftrightarrow y$
falsch	*falsch*	*wahr*
falsch	*wahr*	*falsch*
wahr	*falsch*	*falsch*
wahr	*wahr*	*wahr*

Quantoren können benutzt werden um Formeln zu bilden, die für manche bzw. für alle Individuen gelten sollen.

Existenzquantor

$$\exists N \in \mathbb{N} : (\text{istPrimzahl}(N) \ \wedge \ (N < 10)) \qquad \textit{wahr}$$

Es gibt eine natürliche Zahl N für die gilt, N ist eine Primzahlen und N ist kleiner als 10.

(Es gibt Primzahlen, die kleiner als 10 sind.)

Allquantor

$$\forall N \in \mathbb{N} : (\text{istPrimzahl}(N) \;\Rightarrow\; (N < 10)) \qquad \textit{falsch}$$

Für alle natürlichen Zahlen N gilt, N ist eine Primzahlen impliziert N ist kleiner als 10.

(Alle Primzahlen sind kleiner als 10.)

Schwierigkeiten

Schwierigkeiten beim Formulieren von Sachverhalten mit Quantoren und Variablen ist, dass Variablen erst **eingeführt** und **quantifiziert** werden müssen, bevor die eigentliche Aussage folgen kann.

Beispiel

Jede Primzahl ist kleiner als 10

$\downarrow$

Für alle N gilt, N ist eine Primzahlen impliziert N ist kleiner als 10

$\downarrow$

Für alle natürlichen Zahlen N gilt, N ist Primzahlen impliziert N ist kleiner als 10

$\downarrow$

$$\forall N \in \mathbb{N} : (\text{istPrimzahl}(N) \;\Rightarrow\; (N < 10))$$

Quantoren und Junktoren

Der Allquantor und die Konjunktion sind verträglich, folgende Formeln sind *wahr*.

$$\Big(\forall X : \big(P(X) \wedge Q(X)\big)\Big) \Leftrightarrow \Big(\big(\forall X : P(X)\big) \wedge \big(\forall X : Q(X)\big)\Big)$$
$$\Big(\forall X : \big(P(X) \vee Q(X)\big)\Big) \Leftarrow \Big(\big(\forall X : P(X)\big) \vee \big(\forall X : Q(X)\big)\Big) \quad (\not\Rightarrow)$$

Der Existenzquantor und die Disjunktion sind verträglich, folgende Formeln sind *wahr*.

$$\Big(\exists X : \big(P(X) \vee Q(X)\big)\Big) \Leftrightarrow \Big(\big(\exists X : P(X)\big) \vee \big(\exists X : Q(X)\big)\Big)$$
$$\Big(\exists X : \big(P(X) \wedge Q(X)\big)\Big) \Rightarrow \Big(\big(\exists X : P(X)\big) \wedge \big(\exists X : Q(X)\big)\Big) \quad (\not\Leftarrow)$$

Für die Negation gelten folgende Formeln, d.h. folgende Formeln sind *wahr*.

$$\Big(\neg\big(\forall X : P(X)\big)\Big) \Leftrightarrow \Big(\exists X : \big(\neg P(X)\big)\Big)$$
$$\Big(\neg\big(\exists X : P(X)\big)\Big) \Leftrightarrow \Big(\forall X : \big(\neg P(X)\big)\Big)$$

Folgerung

Ein Quantor wäre ausreichend, es gilt Folgendes (folgende Formeln sind *wahr*).

$$\big(\forall X : P(X)\big) \Leftrightarrow \Big(\neg\big(\exists X : \big(\neg P(X)\big)\big)\Big)$$
$$\big(\exists X : P(X)\big) \Leftrightarrow \Big(\neg\big(\forall X : \big(\neg P(X)\big)\big)\Big)$$

Die Verwendung beider Quantoren ist wegen der besseren Lesbar- und Benutzbarkeit sinnvoll.

Schachtelung

Bei Schachtelungen gleicher Quantoren können diese vertauscht werden, folgende Formeln sind *wahr*.

$$\Big(\forall X : \big(\forall Y : P(X,Y)\big)\Big) \Leftrightarrow \Big(\forall Y : \big(\forall X : P(X,Y)\big)\Big)$$
$$\Big(\exists X : \big(\exists Y : P(X,Y)\big)\Big) \Leftrightarrow \Big(\exists Y : \big(\exists X : P(X,Y)\big)\Big)$$

Bei Schachtelung verschiedener Quantoren darf in der Regel keine Vertauschung vorgenommen werden.

7.2.3 Gültigkeitsbereich

In folgenden Ausdrücken ist die Formel P der **Gültigkeitsbereich** (*scope*) des Quantors für die Variablen X.

$$\forall X : P(X) \quad \text{oder} \quad \exists X : P(X)$$

Eine Variable im Gültigkeitsbereich eines Quantors ist (durch den Quantor) **gebunden**, einen nicht gebundene Variable ist **frei**.

Beispiel

$\forall X : \Big(Q(X) \wedge (X > 2)\Big)$	X ist (durch $\forall$) gebunden
$Q(Y) \Rightarrow \Big(\exists X : R(X, Y, Z)\Big)$	X ist gebunden, Y und Z sind frei
$(X > Y) \wedge \Big(\forall X : \Big(\exists Y : (X < Y)\Big)\Big)$	X, Y sind gleichzeitig frei und gebunden

Geschlossene Formeln enthalten keine freien Variablen, Formeln mit mindestens einer freien Variable sind **offen**.

Beispiel

$\forall X : \Big(Q(X) \wedge (X > 2)\Big)$	geschlossen
$Q(Y) \Rightarrow \Big(\exists X : R(X, Y, Z)\Big)$	offen
$(X > Y) \wedge \Big(\forall X : \Big(\exists Y : (X < Y)\Big)\Big)$	offen

Das Variablen in einer Formel gleichzeitig gebunden und frei vorkommen kann durch Umbenennung der Variablen, ohne Änderung der Bedeutung, vermieden werden.

$$\Big((X > Y) \wedge \Big(\exists Y : P(Y)\Big)\Big) \Leftrightarrow \Big((X > Y) \wedge \Big(\exists Z : P(Z)\Big)\Big)$$

Ebenso ist die Bindung einer Variablen durch verschiedene Quantoren mit Umbenennung vermeidbar.

$$\Big(\Big(\forall X : P(X)\Big) \vee \Big(\exists X : Q(X)\Big)\Big) \Leftrightarrow \Big(\Big(\forall X : P(X)\Big) \vee \Big(\exists Y : Q(Y)\Big)\Big)$$

Üblich ist das Einhalten folgender **Konventionen**.

- Keine Variable in einer Formel ist gleichzeitig frei und gebunden.
- Jeder Quantor in einer Formel führt eine **neue** Variable ein, die ausserhalb seines Gültigkeitsbereichs nicht vorkommt.

Gültigkeitsbereich und Junktoren

Q macht keine Aussage über X.

$$\Big(\big(\forall X : P(X)\big) \wedge Q\Big) \Leftrightarrow \Big(\forall X : \big(P(X) \wedge Q\big)\Big)$$
$$\Big(\big(\exists X : P(X)\big) \wedge Q\Big) \Leftrightarrow \Big(\exists X : \big(P(X) \wedge Q)\big)\Big)$$
$$\Big(\big(\forall X : P(X)\big) \vee Q\Big) \Leftrightarrow \Big(\forall X : \big(P(X) \vee Q\big)\Big)$$
$$\Big(\big(\exists X : P(X)\big) \vee Q\Big) \Leftrightarrow \Big(\exists X : \big(P(X) \vee Q\big)\Big)$$
$$\Big(\big(\forall X : P(X)\big) \Rightarrow Q\Big) \Leftrightarrow \Big(\exists X : \big(P(X) \Rightarrow Q\big)\Big)$$
$$\Big(\big(\exists X : P(X)\big) \Rightarrow Q\Big) \Leftrightarrow \Big(\forall X : \big(P(X) \Rightarrow Q\big)\Big)$$
$$\Big(Q \Rightarrow \big(\forall X : P(X)\big)\Big) \Leftrightarrow \Big(\forall X : \big(Q \Rightarrow P(X)\big)\Big)$$
$$\Big(Q \Rightarrow \big(\exists X : P(X)\big)\Big) \Leftrightarrow \Big(\exists X : \big(Q \Rightarrow P(X)\big)\Big)$$

Die Namen der quantifizierten Variablen sind belanglos.

$$\big(\forall Y : P(Y)\big) \Leftrightarrow \big(\forall Z : P(Z)\big) \quad \text{und} \quad \big(\exists X : P(X)\big) \Leftrightarrow \big(\exists Y : P(Y)\big)$$

Gültigkeitsbereich und Semantik

Geschlossene Formeln sind Aussagen, d.h. der Wahrheitswert einer geschlossenen Formeln kann angegeben werden.

Offenen Formeln sind parametrisierte Aussagen, d.h. der Wahrheitswert einer offenen Formeln kann erst nach dem Ersetzen der freien Variablen durch konkrete Objekte aus dem Individuenbereich festgestellt werden.

Bemerkung

Eine offene Formel wird zu einer geschlossenen, wenn man alle freien Variablen durch konkrete Objekte ersetzt.

Beispiel

$\exists N$: istPrimzahl(N)	geschlossen, über dem Individuenbereich $\mathbb{N}$ ist der Wahrheitswert *wahr*
$\forall N$: istPrimzahl(N)	geschlossen, über dem Individuenbereich $\mathbb{N}$ ist der Wahrheitswert *falsch*
istPrimzahl(N)	offen
istPrimzahl(5)	geschlossen, Wahrheitswert *wahr*
istPrimzahl(99)	geschlossen, Wahrheitswert *falsch*

7.2.4 Terme

Terme

Terme sind Ausdrücke, die bestimmte Objekte des gerade betrachteten Individuenbereichs bezeichnen.

- **Atomare Terme** bezeichnen ein bestimmtes Objekt (Individuum) direkt.
- **Zusammengesetzte Terme** verwenden eine Funktion, die zunächst angewendet werden muss, um das bezeichnete Objekt (Individuum) zu identifizieren.

Beispiel

Die beiden nachfolgenden Termen beschreiben beide dasselbe Objekt aus dem Individuenbereichs aller bei Wikipedia aufgelisteter Personen.

„Horst Köhler, *22.02.1943“

praesident(„Bundesrepublik Deutschland“, 2010)

Die Funktion praesident hat als Argumente Terme eines möglicherweise anderen Individuenbereichs.

Atomare Terme

Die einzigen atomaren (unzerlegbaren) Terme sind Konstanten und Variablen.

- **Konstanten** sind Zeichen oder Zeichenreihen, die immer genau ein Objekt (Individuum) aus dem gerade betrachteten Individuenbereich bezeichnen.

  Beispiel

„Bundesrepublik Deutschland“	$\longrightarrow$	ein Staat, aus dem Individuenbereich der aktuell existierenden Staaten
2010	$\longrightarrow$	die Jahreszahl 2010 aus dem Individuenbereich der Jahreszahlen

- **Variablen** bezeichnen ebenfalls zu jedem Zeitpunkt genau ein Objekt, aber diese Bedeutung kann sich ändern je nachdem, mit welcher Bedeutung die Variable gerade **belegt** worden ist.

  Beispiel

  Die Variable X kann sowohl mit 2010 als auch mit „Bundesrepublik Deutschland“ belegt werden.

Zusammengesetzte Terme

Formale Notation für zusammengesetzte Terme, ist ein **Funktor** (Funktionssymbol oder Operator) gefolgt von einer Parameterliste.

Beispiel

$$\text{wurzel}(N)$$

wurzel ist ein Funktor, N ist ein Parameter.

Zweistellige Funktoren dürfen auch in **Infixnotation** verwendet werden.

$$\text{summe}(X, Y) \qquad \text{bzw.} \qquad X + Y$$

summe und + sind Funktoren, X und Y sind Parameter.

7.2.5 Sprache

Relationen

Prädikate können als **Relationsnamen** aufgefasst werden.

Aller Kombinationen von Individuen, die eine Formel wahr machen, wenn man sie für die Parameter der Formel einsetzt, bilden eine **Relation** (Tupelmenge).

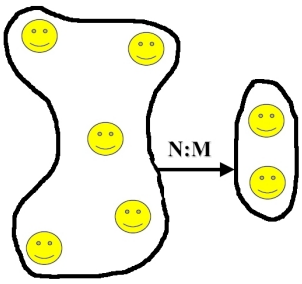
Beispiel

ist_echter_teiler_von(X, Y)

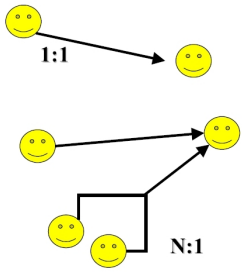
X	Y
2	4
2	6
3	6
2	8
4	8
3	9
2	10
5	10
⋮	⋮

Zuordnungen

Relation

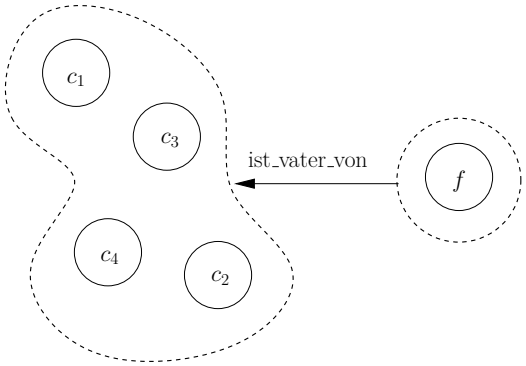


Funktion



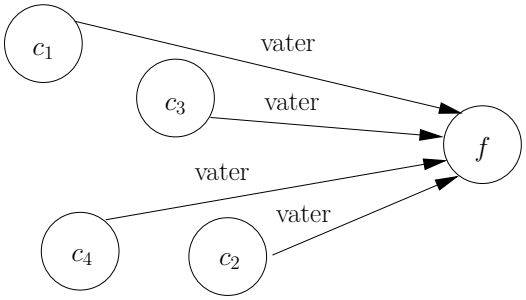
Festlegung konkreter Zuordnungen

Formel



Die **Formel** ist_vater_von(f, c_2) ist *wahr* und machte eine **Aussage über Objekte**.

Term



Der **Term** vater(c_2) bezeichnet das Objekt f, ist damit ein **Objektbezeichner**.

Formeln und Terme

Es ist nicht leicht, atomare Formeln und zusammengesetzte Terme rein syntaktisch zu unterscheiden.

- **Formel** (machen eine Aussage) `<Prädikat>(<Parameterliste>)`
- **Term** (bezeichnen Objekte) `<Funktor>(<Parameterliste>)`

Ob das syntaktische Gebilde

$$\text{magic}(1, 2^2, 3 \cdot 3)$$

ein Term oder eine Formel ist, hängt davon ab, ob das Symbol magic ein Funktor ist (eine Funktion) oder ein Prädikat (eine Relation) ist.

- ist_praesident(„Horst Köhler, *22.02.1943“) ist eine atomare Formel, die eine Aussage macht über das von dem atomaren Term „Horst Köhler, *22.02.1943“ bezeichnete Objekt.
- produkt(3, 42) ist ein zusammengesetzter Term, der ein Objekt bezeichnet und die atomaren Terme 3 und 42 als Parameter enthält.

Sprache

Prädikatenlogik bietet nur einen Sprachrahmen für das Formalisieren beliebiger Anwendungsprobleme. Die Sprache der Logik gibt es nicht.

Stattdessen benötigt jede Anwendung ihre eigene, ganz spezifische logische Sprache.

Syntaktische und semantische Grundlagen aller logischen Sprachen werden festgelegt durch Konventionen/Definitionen der traditionellen Logik.

Von Sprache zu Sprache verschieden ist das verwendetes **Vokabular**, d.h. die Menge derjenigen Zeichen, aus denen Formeln/Terme aufgebaut sind.

Das Vokabular einer konkreten logischen Sprache besteht aus zwei Teilmengen.

- Von Anwendung zu Anwendung verschieden sind Funktoren und Prädikate.

  Das anwendungsspezifisches Vokabular, die **Signatur einer logischen Sprache** muss vom jeweiligen Anwender selbst festgelegt werden.

- Für alle Anwendungen gleich sind Junktoren, Quantoren, Variablen etc.

  Der anwendungsunabhängiger Anteil ist bis auf Dialekte immer identisch (normiert).

Signatur

Die **Signatur** einer Anwendungssprache ist die Sammlung aller Funktoren und Prädikate. Frei wählbar, d.h vom Anwendungsdesigner festzulegen sind

- Prädikate (Relationsnamen),
- Funktoren (Funktionsnamen)

und deren Stelligkeiten (= Anzahl an Parametern).

Formal

Die Signatur einer Anwendungssprache über einem Individuenbereich Ω ist ein Paar $(\mathcal{P}, \mathcal{F})$ für das Folgendes gilt.

- $\mathcal{P}$ ist endliche Menge von Prädikaten P_j mit Stelligkeiten n_j $(1 \leq j \leq n)$, die Aussagen über Objekte aus Ω machen.
- $\mathcal{F}$ ist endliche Menge von Funktoren F_i mit Stelligkeiten m_i $(1 \leq i \leq m)$, die Objekte aus Ω bezeichnen.

7.2.6 Zusammenfassung

Syntax von Termen

1. Jede Variable ist ein Term.
2. Jede Konstante ist ein Term.
3. Wenn $T_1, \ldots, , T_n$ mit $n \geq 1$ Terme sind und wenn F ein n-stelliger Funktor ist, dann ist auch $F(T_1, \ldots, T_n)$ ein Term.

Wichtig

Stets bezogen auf die Signatur einer bestimmte Sprache, die einen entsprechenden Funktor definiert.

4. Sonst gibt es keine weiteren Terme.

Syntax von Formeln

1. *wahr* und *falsch* sind atomare Formeln.

2. Wenn P ein 0-stelliges Prädikat ist, dann ist P eine atomare Formel, die abkürzend für $P()$ steht.

3. Wenn $T_1, \ldots, T_n$ mit $n \geq 1$ Terme sind und wenn P ein n-stelliges Prädikat ist, dann ist $P(T_1, \ldots, T_n)$ eine atomare Formel.

   Wichtig

   Stets bezogen auf die Signatur einer bestimmte Sprache, die ein entsprechendes Prädikat definiert.

4. Wenn g eine Formel ist, dann ist auch $\neg g$ eine Formel.

5. Wenn g und h Formeln sind, dann sind auch folgende Ausdrücke Formeln.

$$g \wedge h, \quad g \vee h, \quad g \oplus h, \quad g \Rightarrow h \quad \text{und} \quad g \Leftrightarrow h$$

6. Wenn g eine Formel und X eine Variable ist, dann sind auch folgende Ausdrücke Formeln.

$$\forall X : g(X) \quad \text{und} \quad \exists X : g(X)$$

7. Sonst gibt es keine weiteren Formeln.

7.2.7 Anwendung

Anwendung der Prädikatenlogik

Prädikatenlogik ist ein Hilfsmittel vor allem für Informatik, Mathematik und Linguistik.

Bei der Konzeption und Programmierung von Expertensystemen und in der künstlichen Intelligenz kommt angewandte Prädikatenlogik zum Einsatz, z.B. die Programmiersprache Prolog (*programming in logic*).

Der Relationenkalkül, eine der theoretischen Grundlagen von Datenbankabfragesprachen wie etwa SQL, bedient sich der Prädikatenlogik als Ausdrucksmittel.

7.2.8 Beispiele

Betrachten Sie folgenden Signatur einer prädikatenlogischen Sprache mit einer Teilmenge der Lebewesen B als Individuenbereich.

Signatur

- Prädikate $\mathcal{P} = \{pilz, rosa, giftig, =\}$
  - Das Prädikat = ist zweistellig, die anderen einstellig.
  - Die Bedeutung der einstelligen Prädikate ergibt sich aus der Namensgebung.
  - Für das Prädikat = gilt, $X = Y$ ist genau dann *wahr*, wenn X und Y dasselbe Individuum bezeichnen.
- Funktoren $\mathcal{F} = \emptyset$

Aufgabe 1

Formulieren Sie für die folgenden Aussagen prädikatenlogischen Formeln, die genau dann *wahr* sind wenn die jeweilige Aussage gilt.

1. Alle rosa Pilze sind giftig.
2. Wenn ein Pilz giftig ist, dann ist er rosa.
3. Kein rosa Pilz ist giftig.

Lösung 1

Formeln

1. $\forall X : \Big((rosa(X) \wedge pilz(X)) \Rightarrow giftig(X) \Big)$

   oder

   $\forall X : \Big(rosa(X) \Rightarrow (pilz(X) \Rightarrow giftig(X)) \Big)$

2. $\forall X : \Big((pilz(X) \wedge giftig(X)) \Rightarrow rosa(X) \Big)$

   oder

   $\forall X : \Big(pilz(X) \Rightarrow (giftig(X) \Rightarrow rosa(X)) \Big)$

3. $\forall X : \neg \Big(rosa(X) \wedge pilz(X) \wedge giftig(X) \Big)$

   oder

   $\forall X : \Big((rosa(X) \wedge pilz(X)) \Rightarrow \neg giftig(X) \Big)$

Aufgabe 2

Ist die prädikatenlogischen Formel

$$\forall X : \Big(\forall Y : \Big((pilz(X) \wedge pilz(Y)) \Rightarrow (X = Y) \Big) \Big)$$

genau dann *wahr* wenn die Aussage

Es gibt genau einen Pilz.

gilt?

Begründen Sie Ihre Antwort oder geben Sie ein Gegenbeispiel an.

Hinweis

Achten Sie auf den Individuenbereich.

Lösung 2

Formel und Aussage sind nicht äquivalent.

Gegenbeispiel. Angenommen B enthält keine Pilze.

- Die Aussage

  Es gibt genau einen Pilz.

  ist *falsch*.

- Die Formel

  $$\forall X : \Big(\forall Y : \Big((pilz(X) \wedge pilz(Y)) \Rightarrow (X = Y)\Big)\Big)$$

  ist *wahr*

  Denn es gilt $(pilz(X) \wedge pilz(Y))$ ist immer *falsch*, d.h. der Gültigkeitsbereich von Y ist immer *wahr* und somit ist die Formel *wahr*.

Aufgabe 3

Korrekturvorschlag

Ist die prädikatenlogischen Formel

$$\Big[\forall X : \Big(\forall Y : \Big((pilz(X) \wedge pilz(Y)) \Leftrightarrow (X = Y)\Big)\Big)\Big]$$

genau dann *wahr* wenn die Aussage

Es gibt genau einen Pilz.

gilt?

Begründen Sie Ihre Antwort oder geben Sie ein Gegenbeispiel an.

Lösung 3

Formel und Aussage sind nicht äcquivalent.

Gegenbeispiel. Angenommen B enthält genau einen Pilz und ein Lebewesen Z, dass kein Pilz ist.

- Die Aussage

  Es gibt genau einen Pilz.

  ist *wahr*.

- Die Formel

  $$\forall X : \Big(\forall Y : \Big((pilz(X) \wedge pilz(Y)) \Leftrightarrow (X = Y)\Big)\Big)$$

  ist *falsch*.

  Denn $(pilz(Z) \wedge pilz(Z))$ ist *falsch* und $(Z = Z)$ ist *wahr*, daraus folgt die korrigierte Formel ist *falsch*.

Korrektur

Die prädikatenlogischen Formel

$$\Big[\exists Z : pilz(Z)\Big] \wedge \Big[\forall X : \Big(\forall Y : \Big((pilz(X) \wedge pilz(Y)) \Rightarrow (X = Y)\Big)\Big)\Big]$$

ist genau dann *wahr* wenn die Aussage

Es gibt genau einen Pilz.

gilt.

Beispiel

Satz 7.17 (Goldbach-Vermutung)**.** *Das Doppelte jeder natürlichen Zahl größer Zwei ist die Summe zweier Primzahlen.*

Aufgabe 1

Geben Sie über dem Individuenbereich der natürlichen Zahlen $\mathbb{N}$ die Signatur einer logischen Sprache an, die verwendet werden kann, um die **Goldbach-Vermutung** als prädikatenlogische Formel zu formulieren.

Hinweis

Für eigene Definitionen können Sie annehmen, dass Begriffe wie Summe, Differenz, Gleichheit, Ungleichheit, etc. bekannt sind und die in der Arithmetik übliche Bedeutung haben.

Lösung 1

Sei $\mathbb{N}$ der Individuenbereich der natürlichen Zahlen.

Signatur mit Prädikaten $\mathcal{P}$ und Funktoren $\mathcal{F}$.

$\mathcal{P} = \{prim, >, =\}$

$prim$	ist einstellig, $prim(n)$ ist genau dann wahr, wenn n eine Primzahl ist.
$=$	ist zweistellig, $n = m$ ist genau dann wahr, wenn n gleich m ist.
$>$	ist zweistellig, $n > m$ ist genau dann wahr, wenn n größer m ist (wird nicht gebraucht bei $\mathbb{N}_{>2}$).

$\mathcal{F} = \{+\}$

$+$	ist zweistellig, $n + m$ ist die Summe der beiden Zahlen.

Aufgabe 2

Benutzen Sie die logischen Sprache aus dem vorstehenden Aufgabenteil um eine prädikatenlogische Formel zu formulieren, die genau dann wahr ist wenn die Goldbach-Vermutung zutrifft.

Lösung 2

Formel über dem Individuenbereich $\mathbb{N}$.

$$\forall n : \Big((n > 2) \Rightarrow \Big(\exists p \in \mathbb{N} : \Big(\exists q \in \mathbb{N} : \Big(\text{prime}(p) \wedge \text{prime}(q) \wedge ((n + n) = (p + q))\Big)\Big)\Big)\Big)$$

8 Python

8.1 Einführung

Ein hervorragende Quelle für eine Einführung in Python ist der ehemalige Kurs von *Dr. Jochen Schulz*.

Mathematisch orientiertes Programmieren, Ein Kurs Python im wissenschaftlich/mathematischen Kontext zu erlernen.

Im Absprache mit Dr. Jochen Schulz werden in diesem Kapitel Texte und Programmcode aus diesem Kurs (teilweise unverändert) übernommen, ohne expliziet als Zitat gekennzeichnet zu sein.

An dieser Stelle vielen Dank an Dr. Jochen Schulz für seine freundliche Genehmigung.

Die vollständige *Python documentation* inklusive

- Tutorial
- Bibliotheks- und Sprachreferenz
- HOWTOs und FAQs
- etc.

finden Sie unter `https://docs.python.org/3/` .

In dieser Vorlesung wird **Python 3** verwendet.

Der Vorgänger Python 2 ist auch noch produktiv im Einsatz, aber leider sind Python 3 und Python 2 **nicht kompatibel**.

für neue Projekte sollte man darauf achten Python 3 zu verwenden.

Unter Linux ist das Programm `python3` sowohl ein **Python-Interpeter**, der **Python-Skripte** Zeile für Zeile ausführt, als auch eine interaktiven **Python-Shell**, die zeilenweise Code entgegen nimmt.

Python-Interpreter

Ein Python-Skript ist eine Textdatei, die Python-Programmcode enthält und, nach Konvention, mit `.py` endet.

Ein Python-Skript kann zur Abarbeitung an den Python-Interpeter übergeben werden.

Fügt man einem Python-Skript als erste Zeile den passenden *shebang*, z.B. **`#!/usr/bin/python3`**, hinzu und ist die Datei ausführbar kann man das Python-Skript direkt auf der Kommandozeile starten.

Beispiel

Datei **`hello.py`**, Version 1

```
1 print('hello world')
```

```
> python3 hello.py
hello world
```

Datei **`hello.py`**, Version 2

```
1 #!/usr/bin/python3
2 print('hello world')
```

```
> chmod u+x hello.py
> ./hello.py
hello world
```

Python-Shell

Startet man **`python3`** erhält man folgende Ausgabe.

```
> python3
Python 3.x.x (default, <date>, <time>)
[GCC 9.x.x] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Das Prompt **`>>>`** gibt an, dass man sich in der interaktiven Python-Shell befindet und Code zeilenweise eingeben kann.

Beendet wird die Python-Shell mit *Ctrl-D* (*Strg-D*).

```
>>> 42
42
>>> 2+3
5
```

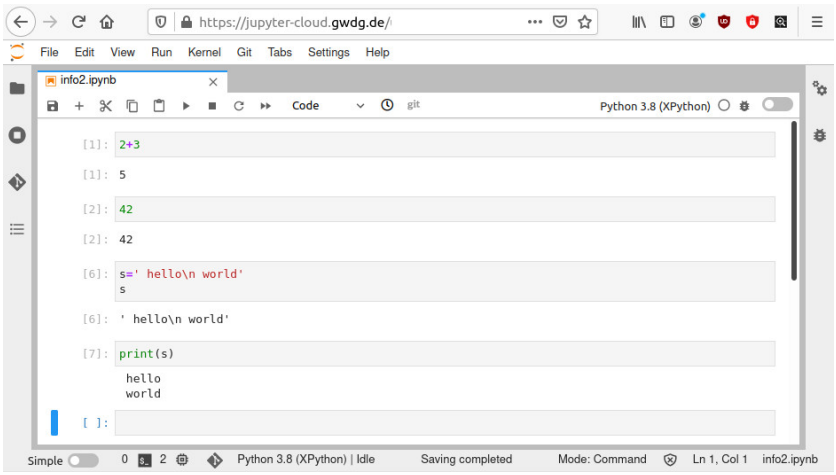
Blöcke (Prompt `...`) werden mit einer Leerzeile abgeschlossen, eingerückt werden kann mit *Tab* (*Tabulator*) oder *Spaces* (*Leerzeichen*).

```
>>> if True:
...     'hello world'
...
'hello world'
```

Jupyter

Jupyter ist eine Web-Anwendung, mit der (verteilte) Dokumente erstellt und bearbeitet werden können, die Code (inklusive Auswertung), Visualisierung und erläuternden Text enthalten.

GWDG Jupyter `https://jupyter-cloud.gwdg.de/` stellt u.a. einen Python3-Kernel bereit, der für die Beispiele dieser Veranstaltung und Experimente genutzt werden.



In Jupyter können Zellen mehrere Zeilen enthalten.

Die Tastenkombination *Shift-Return* (*Umschalt-Eingabe*) veranlasst das Auswerten einer Zelle.

Python-Skripte

Python-Skripte sind eine Liste von **Befehlen** (*statements*), die sequentiell ausgeführt werden.

Mögliche Befehle lassen sich (unvollständig) unterteilen in **einfache** und **zusammengesetzte Befehle**.

Einfache Befehle werden üblicherweise in einer einzelnen Zeile geschrieben.

- Ausdrücke
- Zuweisungen
- Kontroll-Fluss-Befehle (`return`, `continue`, ...)
- ...

Zusammengesetzte Befehle

- Funktionsdefinitionen (`def`)
- Konditionale (`if`)
- Schleifen (`for`, `while`)
- Objekt-Klassen-Definitionen (`class`)
- ...

Ausdrücke

Ausdrücke (*expressions*) bestehen aus Werten, Variablen und aus Operatoren, die diese kombinieren.

Beispiele

```
>>> 2+3
5
```

```
>>> 42
42
```

Ausdrücke haben selbst einen Wert; Ausdrücke, die als Befehl oder Teil von Befehlen auftreten, werden ausgewertet, d.h. so lange gemäß einer Reihe von Regeln reduziert, bis ein Wert übrig bleibt.

Hinweis

Die interaktive Shell zeigt den Wert jedes Ausdrucks an. Benutzt man ein Jupyter-Notebook und ist der letzte Befehl ein Ausdruck, so wird der Wert

diese Ausdrucks angezeigt. Ansonsten muss man für Ausgaben die `print` Funktion verwenden (siehe *Funktionen*).

Kommentare

Jede Eingabe wird generell als Code interpretiert und ausgeführt.

Will man **Kommentare** hinzufügen, die nicht interpretiert werden sollen, geht das mit dem Sonderzeichen `#`.

Der Text, der in einer Zeile auf das `#`-Zeichen folgt, wird nicht interpretiert und ausgeführt.

```
>>> 2+3   # an expression
5
```

Funktionen

Neben einfachen arithmetischen Operatoren können Ausdrücke Aufrufe von **Funktionen** enthalten.

Python-Funktionen sind ähnlich wie Funktionen (Abbildungen) im mathematischen Sinn.

Die Syntax für Funktionsaufrufe ist wie folgt.

```
function_name(argument1, argument2, ...)
```

Ein Funktionsaufruf wird zu einem Wert, dem **Rückgabewert**, ausgewertet und eine Funktion kann zusätzlich einen **Effekt** haben.

Die **Argument** von Funktionen sind Ausdrücke.

Bemerkungen

- In funktionalen Programmiersprachen (z.B. Haskell) haben Funktionen, wie in der Mathematik, keinen Effekt.
- Anders als z.B in Haskell muss die Argumentliste beim Funktionsaufruf in runde Klammern eingeschlossen und durch Kommata getrennt werden.

Beispiel

Zur Betragsfunktion

$$|x| = \begin{cases} x & \text{für } x \geq 0 \\ -x & \text{sonst} \end{cases}$$

gibt es eine Entsprechung in Python, die Funktion **abs**.

```
>>> abs(-1)
1
```

Der Funktionsaufruf wird zum Rückgabewert ausgewertet, mit dem dann wieder Ausdrücke und Funktionsaufrufe gebildet werden können.

```
>>> abs(abs(-1)-2)
1
```

Beispiel

Die Funktion **print** wird zum speziellen Wert **None** ausgewertet, d.h. praktisch hat **print** keinen Rückgabewert, und hat den Effekt, dass sein Argument auf die Standardausgabe geschrieben wird.

```
>>> print(10)
10
```

Bemerkung

Der Effekt von **print** unterscheidet sich von der implizieten Ausgaben des Wertes eines Ausdrucks, bei der das Objekt, das den Wert enthält, implizit ausgegeben wird.

```
>>> print(10)
10
>>> 10
10
>>> print('hello')
hello
>>> 'hello'
'hello'
>>> print(' hello\n world')
 hello
 world
>>> ' hello\n world'
' hello\n world'
```

Variablen und Zuweisungen

Alle Werte in Python sind Objekte (siehe *Objekte*), die in irgendeiner geeigneten Form im Arbeitsspeicher liegen.

Variablen haben **Bezeichner** (*identifier*) und speichern Referenzen auf Objekte, d.h. mit Variablen können Objekte referenziert werden.

Eine **Zuweisung** hat folgende Form.

```
identifier = expression
```

Der Ausdruck **`expression`** wird ausgewertet, und das Resultat wird der Variable mit dem Bezeichner **`identifier`** zugewiesen.

Tritt in folgenden Ausdrücken derselbe Bezeichner auf, wird er durch den der Variable zugewiesenen Wert ersetzt. Dabei wird der Ausdruck nicht noch einmal ausgewertet.

Hat der Ausdruck einen Effekt, tritt dieser nur einmal auf, nämlich zum Zeitpunkt der Zuweisung.

Beispiel

```
>>> a=3+5
>>> print(a)
8
>>> b=a
>>> print(b)
8
```

Die Variablen **`a`** und **`b`** referenzieren dasselbe Objekt mit dem Wert 8, welches Ergebnis des Ausdrucks **`3+5`** ist.

Die linke Seite einer Zuweisung muss eine Variable sein, die das Objekt, welches das Ergebnis der rechten Seite repräsentiert, referenzieren kann.

```
>>> 2*a=4
  File "<stdin>", line 1
SyntaxError: cannot assign to operator
```

Auf diese Weise z.B. Gleichungen zu lösen ist nicht möglich.

Bezeichner (*identifier*)

Bezeichner bestehen aus

- Groß- und Kleinbuchstaben `A-Z` und `a-z`,
- Ziffern `0-9`,
- dem Unterstrichen _.

Bezeichner

- dürfen nicht mit einer Ziffer beginnen,
- unterscheiden zwischen Groß- und Kleinschreibung,
- sollten keine Umlaute oder sonstige Sonderzeichen enthalten.

Bemerkung

Die Beschränkung auf die Buchstaben und Ziffern des ASCII (*American Standard Code for Information Interchange*) ist eine Konvention. Der Interpreter akzeptiert auch Unicode Buchstaben und Ziffern, z.B. `ä` oder α (*code point* `U+03B1`).

In dieser Veranstaltung halten wir uns an die Konvention.

Schlüsselwörter (*keywords*)

Es gibt eine Reihe von Schlüsselwörtern (*keywords*), die reserviert sind und nicht als Bezeichner verwendet werden dürfen.

Die Schlüsselwörter können in verschiedenen Python-Versionen unterschiedlich sein, d.h. einige könnten hinzugefügt oder einige entfernt werden.

Eine Liste der Schlüsselwörter der eingesetzten Version bekommt man mit Hilfe des Moduls `keyword` (siehe *Module*).

```
>>> import keyword
>>> keyword.kwlist
'False', 'None', 'True', 'and', 'as', 'assert', 'async',
'await', 'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Klassen und Objekte

Eine **Klasse** beschreibt die gemeinsamen Attribute (Variablen) und Methoden (Funktionen) einer Menge von gleichartigen Instanzen dieser Klasse.

Instanzen einer Klasse werden **Objekte** (dieser Klasse) genannt.

Die Klasse eines Objekts bestimmt, wie das Objekt im Speicher abgelegt wird und welche Operationen mit dem Objekt durchgeführt werden können.

Alle Werte in Python sind Objekte.

Die Klasse eines Objekts wird auch als **Typ** des Objekts bezeichnet.

Beispiel

Mit der Funktion `type` kann der Typ/die Klasse eines Objekts ermittelt werden.

```
>>> type(10)
<class 'int'>
>>> type(None)
<class 'NoneType'>
```

In Python haben Variablen keinen Typ, sondern nur Objekte (Werte).

Der Typ einen Ausdrucks, dazu gehört auch ein Variablenname, ist der Typ der Wertes, zu dem der Ausdruck ausgewertet wird.

```
>>> a=10
>>> type(a)
<class 'int'>
>>> a=True
>>> type(a)
<class 'bool'>
>>> a=None
>>> type(a)
<class 'NoneType'>
```

Ein Objekt kann selbst wieder Variablen (**Objekt-Attribute**) und Funktionen (**Objekt-Methoden**) besitzen.

Es gilt i.A. die Regel das eine Klasse alle Funktionen definiert, die auf Objekte dieser Klasse angewendet werden können.

Python hat eingebaute Methoden (*built-in functions*), z.B. **`print`** oder **`type`**, die global verfügbar sind und auf (beliebige) Objekte angewendet werden können.

Objekt-Methoden (Funktionen) werden durch das Objekt, gefolgt von einem Punkt, gefolgt von dem Namen der Funktion aufgerufen.

```
object.function()
```

Was der Methodenaufruf tut, hängt nicht nur von den Argumenten, sondern auch vom Objekt selbst ab. Insbesondere können Methoden den Zustand, d.h. die Attribute des Objekt verändern.

Objekt-Attribute (Variablen) werden durch das Objekt, gefolgt von einem Punkt, gefolgt von dem Variablennamen aufgerufen.

```
objekt.variable
```

Beispiel

Die *built-in function* **`bin`** konvertiert eine Integer-Zahl (Objekt der Klasse **`int`**) in eine binäre Zeichenkette mit dem Präfix **`0b`**.

Die Klasse **`int`** definiert eine Objekt-Methode **`int.bit_length()`** die jedes Objekt der Klasse **`int`** (Integer-Zahl) besitzt.

`int.bit_length()` gibt die Anzahl der Bits zurück, die notwendig sind, um eine Ganzzahl als binäre Zeichenkette darzustellen, ohne Vorzeichen und führende Nullen.

```
>>> a=550
>>> bin(a)
'0b1000100110'
>>> a.bit_length()
10
```

Elementare Datentypen (*built-in types*)

Elementare Datentypen (*built-in types*) werden durch den Interpreter global bereitgestellt.

Numerische Typen (*numeric types*)

- Wahrheitswert (`bool`)
- Ganze Zahl (`int`)
- Gleitkommazahl (`float`)
- Komplexe Zahl (`complex`)

Sequenz-Typen *sequence types*

- String/Zeichenkette (`str`)
- Liste (`list`)
- Tupel (`tuple`)
- Bereich (`range`)

Zuordnungs-Typen *mapping types*

- Dictionary/Wörterbücher (`dict`)

Bemerkung. Komplexe Zahlen werden wir nicht behandeln.

Die Objekte der meisten elementaren Datentypen (z.B. `bool`, `int`, `float`, `string`, `tuple`, `range`) sind *immutable* (unveränderlich).

Nachdem ein *immutable* Objekt erstellt und ihm ein Wert zugewiesen wurde, kann man diesen Wert und damit auch den Zustand des Objekts nicht mehr ändern.

Einige elementaren Datentypen (z.B. `list`, `dict`) sind *mutable*, der Zustand des Objektes, kann mit geeigneten Objekt-Methoden verändert werden.

8.2 Numerische Typen

8.2.1 Wahrheitswerte (`bool`)

Wahrheitswerte (*booleans*) repräsentieren Wahr/Falsch-Werte und haben den Typ `bool`.

```
True
False
```

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Es gibt drei **Operatoren auf Wahrheitswerten**.

Operator	Bedeutung
`x or y`	wenn `x` *wahr* ist, dann liefere `x`, sonst `y`
`x and y`	wenn `x` *falsch* ist, dann liefere `x`, sonst `y`
`not x`	wenn `x` *falsch* ist, dann liefere `True`, sonst `False`

Die Operatoren `and` und `or` garantieren die **bedingte Auswertung** (*short-circuit evaluation*), d.h.

- `or` wertet den zweiten Ausdruck nur aus, wenn der erste *falsch* ist,
- `and` wertet den zweiten Ausdruck nur aus, wenn der erste *wahr* ist,

weil im jeweils anderen Fall das Ergebnis der Auswertung bereits fest steht.

Es gibt acht **Vergleichsoperatoren** die einen Wahrheitswert zurückliefern.

Operator	Bedeutung
`<`	echt kleiner
`<=`	kleiner oder gleich
`>`	echt größer
`>=`	größer oder gleich
`==`	gleich
`!=`	ungleich
`is`	gleiches Objekt
`is not`	unterschiedliches Objekt

8.2.2 Ganze Zahlen (`int`)

Ganze Zahlen haben den Typ `int`.

```
>>> 1 + 3
4
>>> -1 * 9
-9
>>> 2 ** 3    # 2 to the power 3
8
```

Ganze Zahlen können beliebig groß sein, solange genug Arbeitsspeicher vorhanden ist.

```
>>> 2 ** 1000
10715086071862673209484250490600018105614048117055336074437503883703510511
```

Division ergibt (seit Python 3) immer eine Gleitkommazahl (siehe *Gleitkommazahlen*).

```
>>> 9/3
3.0
```

Ganzzahlige Divison ist die Division mit Rest (*remainder*), das Ergebnis wird ganzzahligen durch Abrunden.

```
>>> 9//3      # remainder 0
3
>>> 10//3     # remainder 1
3
>>> -10//3    # remainder 2
-4
>>> 10//-3    # remainder -2
-4
```

Modulo berechnet den Rest der ganzzahligen Division, es gilt **`n==(n//m)*m+n%m`** .

```
>>> 9%3
0
>>> 10%3
1
>>> -10%3
2
>>> 10%-3
-2
```

8.2.3 Gleitkommazahlen (`float`)

Gleitkommazahlen sind Zahlen mit begrenzter Genauigkeit; nur eine beschränkte Anzahl an signifikanten Stellen kann gespeichert werden (meist 53 Bits). Diese haben den Typ **`float`**.

Die Eingabe von **`float`** geschieht implizit durch die Angabe eines Punktes für die Nachkommastellen.

```
>>> type(3.0)
<class 'float'>
>>> type(3.)
<class 'float'>
```

Durch die begrenzte Genauigkeit können Rundungsfehler auftreten und Stellen wegfallen.

```
>>> 3 * 0.1
0.30000000000000004
>>> 0.1 == 0.10000000000000001
True
```

8.2.4 Operationen

Die numerischen Typen unterstützen u.a. die folgenden **arithmetischen Operationen**.

Operation	Bedeutung
`x+y`	Summe von `x` und `y`
`x-y`	Differenz von x und `y`
`x*y`	Produkt on `x` und `y`
`x/y`	Quotient von `x` und `y`
`x//y`	abgerundeter Quotient von `x` und `y`
`x%y`	Rest von `x//y`
`-x`	`x` negiert
`+x`	`x`
`abs(x)`	Betrag von `x`
`int(x)`	`x` umgewandet in eine ganze Zahl
`float(x)`	`x` umgewandet in eine Gleitkommazahl
`pow(x,y)`	`x` hoch `y`
`x**y`	`x` hoch `y`

Die Typen `bool` und `int` unterstützt die folgenden **Bit-Operatoren**.

Operator	Bedeutung
`x|y`	bitweise *or* von `x` und `y`
`x^y`	bitweise *exclusive or* von `x` und `y`
`x&y`	bitweise *and* von `x` und `y`
`x<<n`	`x` verschoben nach links um `n` Bits
`x>>n`	`x` verschoben nach rechts um `n` Bits
`~x`	bitweise Invertierung von `x`

Bemerkung

für alle Operatoren ist einen Priorität und Assoziativität festgelegt, die Reihenfolge und Auswertungsrichtung eines Ausdrucks mit mehreren Operatoren bestimmt. Darauf wird hier nicht weiter eingegangen.

Ist die Auswertungsreihenfolge wichtig, muss passende geklammert (mit runden Klammern) werden.

8.3 Sequenz-Typen

8.3.1 Strings (`str`)

Strings (Zeichenketten) haben den Typ **`str`** und werden mit einfachen `'` oder doppelten `''` Anführungszeichen gekennzeichnet.

```
>>> s='hello world'
>>> s
'hello world'
>>> type(s)
<class 'str'>
```

Strings können beliebige Sonderzeichen enthalten.

```
>>> print('öäü')
oau
```

Mehrere Strings direkt hintereinander werden zusammengefügt, was manchmal bei sehr langen Zeilen nützlich sein kann.

```
>>> print('Lorem ipsum dolor sit amet, consetetur sadipscing '
... 'elitr, sed diam nonumy eirmod tempor invidunt ut labore '
... 'et dolore magna aliquyam erat, sed diam voluptua.')
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
```

Mehrzeilige Strings können auch mit drei `'` oder `''` geschrieben werden.

```
>>> print('''Ein String
... der ueber mehrere Zeilen
... geht und Zeilenumbrueche
... direkt enthaelt.''')
Ein String
der ueber mehrere Zeilen
geht und Zeilenumbrueche
direkt enthaelt.
```

Bemerkung

Die Python-Shell bemerkt die fehlende schließende Klammer/die fehlenden drei ’ und beginnt einen neue Zeile mit ... für den Rest der Parameterliste/des mehrzeiligen Strings.

In eine Python-Skript können Sie die Zeilen einfach untereinander schreiben.

Der Backslash \ wird verwendet, um Zeichen zu codieren, die selbst eine besondere Bedeutung haben, wie z.B. die Anführungszeichen \’ und \’’ oder den Backslash selbst \\.

Weiterhin kann man so auch Zeichen codieren, die man nicht ohne weiteres eingegeben kann z.B. den Zeilenumbruch **n**.

```
>>> print('String with quote \', double quote ", '
... 'newline\nand backslash \\.')
String with quote ', double quote ", newline
and backslash \.
```

Bemerkungen

- Die mit einem Backslash beginnenden Zeichenfolgen nennt man *escape sequences*.
- Mit **u** kann man einen Unicode *code point* angegeben, z.B. **"\u03B1"** für α.
- Innerhalb von einfachen Anführungszeichen müssen doppelte Anführungszeichen nicht als *escape sequences* dargestellt werden und umgekehrt.

Die Klasse **str** definiert die Objekt-Methode **format**, die auf jedem Objekt von **str** aufgerufen werden kann und es erlaubt in Strings Platzhalter durch die Stringrepräsentationen beliebiger Werte zu ersetzen.

Eine **kurze** Einführung.

```
>>> s = 'Pi is approximately {}. Another approximation is {}/{}.'
>>> a = 3.1415
>>> b = 22
>>> c = 7
>>> print(s.format(a, b, c))
Pi is approximately 3.1415. Another approximation is 22/7.
```

Dabei wird der n-te Platzhalter {} durch das n-te Argument der `format`-Funktion ersetzt.

Dasselbe kann man durch sogenannte **f-strings** (*formatted string literals*) erreichen. Ein `f` direkt vor dem String sorgt dafür, dass man direkt Variablen und Ausdrücke in die Platzhalter schreiben kann.

```
>>> print(f'Pi is approximately {a}.')
Pi is approximately 3.1415.
>>> print(f'Another approximation is {b}/{c}.')
Another approximation is 22/7.
```

8.3.2 Listen (`list`)

Listen (`list`) sind **veränderbare** geordnete Sammlungen von beliebigen Objekten.

```
[a, b, c, ...]
```

- Eine Liste ist eine Aneinanderreihung beliebiger Objekte, die durch Kommata getrennt sind.
- Eine Liste ist in eckige Klammern `[.., ..]` eingeschlossen.
- Elemente sind geordnet (daher indizierbar)
- Listen sind *mutable*.

Bemerkung

Obwohl Listen beliebige Objekte aufnehmen können, ist die Hauptanwendung von Listen die Verwaltung einer **variablen Anzahl** von Objekten **eines einzelnen Typs** oder eng verwandter Typen (z.B. `int` und `float`).

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> a
[3, False, 1.4, 'text']
>>> type(a)
<class 'list'>
>>> len(a)       # length of list
4
```

Zugriff auf Listenelemente geht über die Angabe des Index in eckigen Klammern.

```
list[index]
```

list und **index** sind Ausdrücke, die zu einer Liste bzw. einem Integer ausgewertet werden.

- Die Indizierung beginnt bei 0.
- Negative Indizes zählen von rechts.

Bemerkung

Eckige Klammern, die sich direkt an einen Ausdruck anschließen, sind im Allgemeinen ein Zugriff per Index auf einen *sequence type*.

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> a[0]
3
>>> a[2]
1.4
>>> a[-1]
'text'
```

Teillisten (*slices*) können extrahiert werden, indem Indizes für Start und Ende der Teilliste angegeben werden.

```
liste[start:stop]
```

Die Teilliste enthält das Element mit Index **`start`**, aber **nicht** das Element mit Index **`stop`**.

Wird einer der Indizes weggelassen, starten Teillisten am Anfang oder gehen bis zum Ende.

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> a[2:4]      # a[4] is not valid
[1.4, 'text']
>>> a[0:-2]
[3, False]
>>> a[:2]
[3, False]
>>> a[2:]
[1.4, 'text']
>>> a[:]
[3, False, 1.4, 'text']
>>> a[1:1]
[]
```

Mit dem Operator **`+`** können Listen verkettet werden.

```
list1 + list2
```

Der Operator **`in`** testet, ob ein Element in einer Liste gespeichert ist.

```
value in list
```

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> b=a+[True, 10]
>>> b
[3, False, 1.4, 'text', True, 10]
>>> True in a
False
>>> True in b
True
```

Listen sind *mutable*, d.h. sie können nach ihrer Konstruktion im Speicher bearbeitet werden, z.B. über einen Zuweisungen an einen Index einer Liste.

```
list[index] = value
```

Die Zuweisungen an den Index **index** der Liste **list** weist nicht dem Namen **list[index]** den Wert **value** zu (vergleiche *Zuweisungen*), sondern ändert in der Variablen **list** den Eintrag am Index **index** auf den Wert **value**.

Erinnerung. Strings sind *immutable*.

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> a[1] = -1
>>> a
[3, -1, 1.4, 'text']
```

```
>>> s = 'abcdefghi'
>>> s[3] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Die Objekt-Methode **pop** entfernt ein Element vom Ende der Liste und liefert dieses zurück. Das geht auch mit einem Index als optionales Argument.

```
value = list.pop()
value = list.pop(index)
```

Man kann auch das erste Auftreten eines Element aus einer Liste entfernen.

```
liste.remove(element)
```

Beispiel

```
>>> a = [3, False, 1.4, 'text']
>>> v = a.pop()
>>> v
'text'
>>> a
[3, -1, 1.4]
>>> a.pop(1)
-1
>>> a
[3, 1.4]
>>> a.remove(3)
>>> a
[1.4]
```

Manche Operationen gibt es sowohl als *in-place* (verändern das Objekt) als auch als *out-of-place* (liefern ein neues Objekt zurück) Version. Ein Beispiel dafür ist das Sortieren von Listen.

Die *built-in functions* **`sorted`** erzeugt eine neue Liste, die die Elemente der als Argument übergebenen Liste in sortierter Reihenfolge enthält. Die ursprüngliche Liste bleibt unverändert.

```
sorted(list)
```

Die Objekt-Methode **`sort`** sortiert die Elemente der Liste, auf der sie aufgerufen wird.

```
list.sort
```

Beispiel

```
>>> a = [1, 3, 2, -1]
>>> sorted(a)
[-1, 1, 2, 3]
>>> a
[1, 3, 2, -1]
>>> a.sort()
>>> a
[-1, 1, 2, 3]
```

Der Zuweisungsoperator weist Variablen Referenzen auf Objekte zu.

Wenn ein *mutable* Objekt verändert wird hat das einen Effekt für alle Variablen, die dieses Objekt referenzieren.

Beispiel

```
>>> a = [1, 3, 2, -1]
>>> b = a
>>> a
[1, 3, 2, -1]
>>> b
[1, 3, 2, -1]
>>> a.sort()
>>> a
[-1, 1, 2, 3]
>>> b
[-1, 1, 2, 3]
```

8.3.3 Tupel (`tupel`)

Tupel (`tupel`) sind **unveränderbare** geordnete Sammlungen von beliebigen Objekten.

```
(a, b, c, ...)
```

- Ein Tupel ist eine Aneinanderreihung beliebiger Objekte, die durch Kommata getrennt sind.
- Ein Tupel ist in runde Klammern `(.., ..)` eingeschlossen.
- Ein Tupel mit einem einzelnen Element benötigt ein zusätzliches Komma, um es von einem eingeklammerten Wert zu unterscheiden.
- Tupel sind ähnlich wie Listen, allerdings *immutable*.

Bemerkung

Die Hauptanwendung von Tupeln ist die Verwaltung einer **festen Anzahl** von Objekten **beliebigen Typs**.

Beispiel

```
>>> a = (1, 2, -1)
>>> a[1]
2
>>> sorted(a)
[-1, 1, 2]
>>> (10)
10
>>> (10,)
(10,)
>>> a[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

8.3.4 Dictionary (`dict`)

Ein **Dictionary** ordnet hashbare Schlüssel beliebigen Objekten zu.

```
{key1: value1, key2: value2, ...}
```

- Ein Dictionary enthält Zuordnungen der Form **`key: value`**.
- **`key`** können alle *hashable objects* sein, insbesondere die Objekte der *immutable built-in types*.
- **`value`** kann beliebigen Typ haben, also auch *mutable* sein.
- Dictionaries sind *mutable*.

Auf ein Dictionary zugreifen (lesen und schreiben) kann man u.a. mit **`dict[key]`**.

Entnehmen (Löschen und Zurückgeben) von Einträgen erfolgt mit der Objekt-Methode **`dict.pop()`**.

Dictionaries können mittels der Objekt-Methode **`dict.update()`** aneinander gehängt werden. **Vorsicht** bei bereits vorhandenen Schlüsseln werden die Objekte überschrieben.

Beispiel

```
>>> d = {1: 10, 'key-text': [True, False],
... (1, 2, 3): 'value-text'}
>>> d[1]
10
>>> d['key-text']
[True, False]
>>> d[(1, 2, 3)]
'value-text'
>>> d[2] = 2
>>> d
{1: 10, 'key-text': [True, False], (1, 2, 3): 'value-text', 2: 2}
>>> d.pop('key-text')
[True, False]
>>> d
{1: 10, (1, 2, 3): 'value-text', 2: 2}
>>> e = {2:5, 'x':44}
>>> d.update(e)
>>> d
{1: 10, (1, 2, 3): 'value-text', 2: 5, 'x': 44}
```

Operationen

Die folgenden Operationen werden von den elementare Sequenz-Typen (*built-in sequence types*), wie z.B. **str** und **list**, unterstützt.

Operation	Bedeutung
x in s	**True** wenn ein Element von **s** gleich **x** ist, sonst **False**
x not in s	**False**, wenn ein Element von **s** gleich **x** ist, sonst **True**
s+t	die Verkettung von **s** und **t**
s*n	**n**-malige Verkettung von **s** (mit sich selbst)
n*s	siehe **s*n**
s[i]	Element mit Index **i** in **s**, Zählung beginnt bei 0
s[i:j]	Abschnitt aus **s** von Index **i** bis **j-1**
s[i:j:k]	Abschnitt aus **s** von **i** bis **j-1** mit Schrittlänge **k**
len(s)	Länge von **s**
min(s)	kleinstes Element von **s**
max(s)	größtes Element von **s**
s.count(x)	Anzahl der Vorkommen von **x** in **s**

8.4 Funktionen

Die meisten nicht-trivialen Programme haben Code-Teile, die mehrfach ge-

nutzt werden sollen. Es bietet sich an, diese in selbst-definierte Funktionen auszulagern.

Funktionen mit klar definiertem Verhalten, Argumenten und Rückgabewerten erhöhen die Verständlichkeit von Programmen.

Funktionen werden in Python mit dem Schlüsselwort **`def`** definiert.

```
1 def function_name(paramter1, parameter2, ...):
2     statement1
3     statement2
4     ...
```

Über den Bezeichner **`function_name`** kann die Funktion aufgerufen werden.

Die Elemente der **Parameterliste** sind ebenfalls Bezeichner, über die innerhalb der Funktion auf die übergebenen Argumente zugegriffen werden kann.

In der Funktionsdefinition folgt nach **`def`** der Bezeichner der Funktion, die Parameterliste in runden Klammern und ein Doppelpunkt.

In der nächsten Zeile beginnt der Code-Block. Python entscheidet durch Einrückung, welche Befehle zum Block gehören, deshalb muss jede Zeile des Blocks in gleicher Weise, d.h. auch mit denselben Zeichen, eingerückt sein.

Bemerkung. In der Python-Shell kann mit *Tab* (*Tabulator*) eingerückt werden, in Skripten ist eine Einrückung mit vier *Spaces* (*Leerzeichen*) üblich.

Bei jedem Aufruf der Funktion werden die Befehle des Code-Blocks der Reihe nach abgearbeitet. Innerhalb der Funktion kann der Befehl

```
return expression
```

verwendet werden, um die Funktion zu beenden und den Wert des Ausdrucks **`expression`** an den Aufrufer zurückzuliefern.

Bemerkung

Ein **`return`** ohne Argument liefert **`None`**. Eine Funktion ohne explizites **`return`** wird implizit um ein **`return`** (ohne Argument) ergänzt.

Beispiel

Datei **function-example.py**

```
#!/usr/bin/python3

def print_and_add(x, y):      # function definition
    print('x =', x)
    print('y =', y)
    return x+y
    print('line will not reached')

print(print_and_add(2, 3))    # function call
```

```
> ./function_example.py
x = 2
y = 3
5
```

Erinnerung

Eine Datei muss ausführbar sein, um sie auf der Kommandozeile zu starten.

```
> chmod u+x function-example.py
```

Alternativ kann man die Datei auch direkt an den Python-Interpreter übergeben.

```
> python3 function-example.py
```

8.5 Kontrollstrukturen

8.5.1 Konditionale (`if`)

Konditionale erlauben es, Code-Teile nur dann auszuführen, wenn eine bestimmte Bedingung erfüllt/nicht erfüllt ist.

```
if expression:
    statements
    ...
elif expression:
    statements
    ...
elif expression:
    statements
```

```
    ...
else:
    statements
    ...
```

- Die **if**-/**elif**-/**else**-Blöcke werden von oben nach unten abgearbeitet.
- für **if**/**elif** wird der jeweilige Ausdruck **expression** ausgewertet.
  - Hat der Ausdruck **expression** den Wahrheitswert **False**, wird zum nächsten **elif**- bzw. **else**-Block gesprungen.
  - Hat der Ausdruck **expression** den Wahrheitswert **True**, werden die Befehle **statements** des Blocks ausgeführt. Danach werden keine weiteren **elif**- oder **else**-Blöcke mehr abgearbeitet.
  - Ist keine Ausdruck von **if**/**elif** **True**, werden die Befehle **statements** des **else**-Blocks ausgeführt.
- Der **if**-Teil ist obligatorisch, **elif** und **else** sind optional.

Beispiel

```
>>> def absolute_value(x):
...     if x > 0:
...         return x
...     else:
...         return -x
...
>>> print(absolute_value(1))
1
>>> print(absolute_value(-2))
2
```

Bemerkung

In diesem Kontext haben nicht nur Ausdrücke vom Typ **bool** einen Wahrheitswert.

Z.B. haben `0`, die leere Liste `[]` und der leere String `''` den Wahrheitswert **False**.

Als Faustregel gilt, dass *leere* Werte (oft) den Wahrheitswert **False** haben.

8.5.2 Schleifen (`for`, `while`))

Schleifen dienen dazu, eine Reihe von Befehlen mehrfach auszuführen, wobei die Anzahl der Wiederholungen selbst programmatisch geregelt werden kann.

Eine **`for`**-Schleife durchläuft ein iterierbares Objekte (*iterable*) mit einer Schleifevariable, die das aktuelle Element in der Schleife verfügbar macht.

```
for identifier in iterable:
    statements
    ...
```

- **`identifier`** ist der Bezeichner der Schleifevariable.
- **`iterable`** ist ein iterierbares Objekte.

Bemerkung

String, Listen, Tupel und Dictionaries sind *iterable*.

Beispiel

Bei Strings, Listen und Tupel erfolgt die Iteration über alle Elemente der Liste, von links nach rechts.

```
>>> s=''
>>> for c in 'short text':
...     s=s+c
...
>>> s
'short text'
>>> for val in [1, 'zwei', (3,4)]:
...     print(val)
...
1
zwei
(3, 4)
>>> sum=0
>>> for x in (1, 5, 7, 22, 4):
...     sum+=x
...
>>> sum
39
```

Beispiel

Bei Dictionaries erfolgt die Iteration über die Schlüssel, in unbestimmter Reihenfolge.

```
>>> d = {1: 'one', 2: 'two'}
>>> for key in d:
...     print('{0} => {1}'.format(key, d[key]))
...
1 => one
2 => two
```

Ein Dictionary liefert mit der Objekt-Methoden

- **keys** eine Liste aller Schlüssel,
- **values** eine Liste aller Werte,
- **items** eine Liste aller **(key, value)**-Tupel,

über die dann interiert werden kann.

Bemerkung

Listen und Tupel unterstützen Mehrfachbelegung (*multiple assignment*).

```
>>> x, y, z  = (1,2,3)
```

Beispiel

```
>>> d={1: 'one', 2: 'two'}
>>> for key in d.keys():
...     print(key)
...
1
2
>>> for value in d.values():
...     print(value)
...
one
two
>>> for key, val in d.items():
...     print('{0} => {1}'.format(key, val))
...
1 => one
2 => two
```

Ranges (Zahlenreihen) werden, ähnlich wie die Indizes bei *slices*, mit Zahlenwerte von **start** bis **stop-1** und Schrittweite **step** erzeugt.

```
range(start, stop, step)
```

- **start** und **step** sind optional.
- Ranges sind *immutable* und *iterable*.

Bemerkung

Der Vorteil von Ranges gegenüber Listen oder Tupel ist, dass ein Range-Objekt immer die gleiche (kleine) Menge an Speicher benötigt, unabhängig von der Größe des Bereichs, den es repräsentiert, da es nur die Start-, Stop- und Step-Werte speichert und die einzelnen Elemente und Teilbereiche nach Bedarf berechnet.

Beispiel

```
>>> for val in range(5):
...     print(val)
...
0
1
2
3
4
>>> for val in range(-3, 0):
...     print(val)
...
-3
-2
-1
>>> for val in range(3, 10, 2):
...     print(val)
...
3
5
7
9
```

Die **while**-Schleife wiederholt einen Block solange wie eine bestimmte Bedingung erfüllt ist.

```
while expression:
    statements
    ...
```

- Der Ausdruck **expression** wird ausgewertet.
  - Ist der Ausdruck **True** wird der Code-Block ausgeführt. Nach dem Ende des Blocks geht die Ausführung wieder in die Zeile **while** ... .
  - Ist der Ausdruck **False** geht die Ausführung hinter dem Code-Block weiter.

Bemerkung

Es kann sein, dass der Block überhaupt nicht oder fortlaufenden ausgeführt wird.

Beispiel

Euklidischer Algorithmus zur Bestimmung des *greatest common divisor* (*gcd*) zweier Zahlen n und m

Solange $n \neq m$

- Ersetze die größere der beiden Zahlen n und m durch die Differenz zwischen größerer und kleinerer Zahl.

Gilt $n = m$ ist dies der *gcd*.

Datei **gcd.py**

```
#!/usr/bin/python3
# greatest common divisor
def gcd(n, m):
    print('start')
    while n != m:
        print(f'loop: n = {n} m = {m}')
        if n > m:
            print('  n is greater')
            n = n - m
        else:
            print('  m is greater')
            m = m - n
    print(f'stop: n = m = {n}')
    return n
```

```
gcd(24, 42)
```

```
> ./gcd.py
start
loop: n = 24 m = 42
  m is greater
loop: n = 24 m = 18
  n is greater
loop: n = 6 m = 18
  m is greater
loop: n = 6 m = 12
  m is greater
stop: n = m = 6
```

8.6 Klassen und Objekte

- Klassen geben die Möglichkeit Daten mit Funktionalitäten zentral zu vereinigen.
- Eine neue Klasse ist ein neuer Typ Objekt von dem Instanzen erschaffen werden können.
- Jede Instanz (Objekt) einer Klasse ist definiert durch Attribute (Variablen), welche den internen Status verwalten.
- Jede Klasse definiert Methoden (Funktionen) mit denen Instanzen dieser Klasse ihren internen Status verändern oder in geeigneter Form nach außen geben können.

```
class ClassName:
    <Statement-1>
    .
    .
    .
    <Statement-N>
```

- Die meisten Statements sind Funktionsdefinitionen.
- Klassendefinitionen müssen ausgeführt werden bevor sie benutzt werden können.
- Das Ergebnis der Ausführung ist ein Klassenobjekt.

Klassenobjekte unterstützen zwei Arten von Operationen: Attributreferenzierung und Instanziierung.

```
class MyClass:
    i = 42

    def f(self):
        return 'hello world'
```

- Attributreferenzierung benutzt die Syntax `obj.attrName`.
- `MyClass.i` und `MyClass.f` sind valide Attributreferenzierungen.
- `MyClass.i` gibt einen Interger zurück.
- `MyClass.f` gibt eine Funktion zurück.
- Die Referenzen können benutzt werden um neue Werte zuzuweisen.
- Klasseninstanziierung benutzt die Funktions-Syntax.
- Klassenname als parameterlose Funktion gibt eine leere Instanz dieser Klasse zurück.

Im folgenden Beispiel wir eine neue Instanz der Klasse erstellt und von der variable `x` referenziert.

```
x = MyClass()
```

Diese Instanziierung erstellt ein neues leeres Objekt dieser Klasse.

- Die spezielle Funktion `__init__()` kann benutzt werden um die Instanziierung zu steuern.
- Argumente werden dabei bei der Instanziierung an `__init__()` weitergegeben.

```
>>>class Complex:
...
...   def __init__(self, realpart, imagpart):
...     self.r = realpart
...     self.i = imagpart
...
>>>x = Complex(3.0, -4.5)
x.r, x.i
(3.0, -4.5)
```

- Ein Objekt besitzt nach der Erstellung eigene Variablen (**Objekt-Attribute**) und Funktionen (**Objekt-Methoden**).

- Die Klasse definiert alle Funktionen, welche auf Objekten dieser Klasse angewendet werden können.
- Objekt-Methoden (Funktionen) werden durch das Objekt, gefolgt von einem Punkt, gefolgt von dem Namen der Funktion aufgerufen (`object.function()`).
- Objekt-Attribute (Variablen) werden durch das Objekt, gefolgt von einem Punkt, gefolgt von dem Variablennamen aufgerufen (`objekt.variable`).
- Es gibt eingebaute Methoden (*built-in functions*), z.B. **`print`** oder **`type`**, die global verfügbar sind und auf (beliebige) Objekte angewendet werden können.

Die Werte von Instanzvariabeln sind einzigartig mit jeder einzelnen Instanz (je nach Initialisierung). Klassenvariabeln sind Attribute und Methoden, die von allen Instanzen dieser Klasse geteilt werden.

```
>>>class Komplex:
...   kind = "Dies hier ist eine Komplexe Zahl."
...
...   def __init__(self, realpart, imagpart):
...     self.r = realpart         # Instanzvariable
...     self.i = imagpart         # Instanzvariable
...
>>>x = Komplex(3.0, -4.5)
>>>y = Komplex(5.0, 2.5)
>>>print(x.r)
>>>print(y.r)
>>>print(x.kind)
>>>print(y.kind)
3.0
5.0
Dies hier ist eine Komplexe Zahl.
Dies hier ist eine Komplexe Zahl.
```

Polymorphimus beschreibt in den meisten Programmiersprachen Methoden und Operatoren, welche mit dem gleichen Bezeichner auf unterschiedlichen Objekten ausgeführt werden können. In Python sind zum Beispiel die Funktionen `print()` und `len()` auf unterschiedlichen Objekten ausführbar.

```
>>>len(["Hallo", "Welt"])
2
>>>len("Hallo Welt")
10
```

Für ein Objekt einer eigenen Klasse `myobj` kann man z.B. in der Klasse mit `__str__()` eine Stringrepräsentation definieren, welche beim Aufruf von `print(myobj)` aufgerufen wird.

```
>>>class Komplex:
...   kind = "Dies hier ist eine Komplexe Zahl."
...
...   def __init__(self, realpart, imagpart):
...     self.r = realpart         # Instanzvariable
...     self.i = imagpart         # Instanzvariable
...   def __str__(self):      # Methodenname für Stringrepräsentation str(
        return "Real: " + str(self.r) + " Imag: " + str(self.i)
>>>x = Komplex(3.0, -4.5)
>>>print(x)
Real: 3.0 Imag: -4.5
```

In einer Funktionsdefinition können Default-Werte mittels `=defWert` angegeben werden.

```
>>>class Komplex:
...   kind = "Dies hier ist eine Komplexe Zahl."
...
...   def __init__(self, realpart, imagpart=0.0):
...     self.r = realpart         # Instanzvariable
...     self.i = imagpart         # Instanzvariable
...   def __str__(self):      # Methodenname für Stringrepräsentation str(
        return "Real: " + str(self.r) + " Imag: " + str(self.i)
>>>x = Komplex(3.0)
>>>print(x)
Real: 3.0 Imag: 0.0
```

Default-Werte werden meistens in Dokumentationen der Funktionen erklärt z.B. in `sort(*, key=None, reverse=False)` beschreibt `key` welcher Key zum Sortieren benutzt werden soll und `reverse` ob der Vergleichsoperator umgedreht werden soll.

- Vererbung erlaubt es einer Klasse Methoden und Attribute von einer anderen Klasse vordefiniert zu bekommen.
- Die Eltern-Klasse ist die Klasse von der geerbt wird (Base-Klasse).
- Die Kind-Klasse ist die Klasse, die von einer anderen Klasse erbt (Derived-Klasse).

Die Syntax ist dabei wie bei der Erstellung einer normalen Klasse mit dem Zusatz der Base-Klasse in runden Klammern.

```
class DerivedClassName(BaseClassName):
```

```
    <Statement-1>
    .
    .
    .
    <Statement-N>
```

Methoden und Attribute, welche in der Derived-Klasse nicht gefunden werden, werden in der Base-Klasse gesucht. Mit der Funktion `super()` kann man expliziet die Base-Klasse ansteuern, z.B. bei der Erstellung eines Objektes mit `__init__()`.

```
class Person:
  def __init__(self, firstname, lastname, age): #Konstruktor
    self.firstname = firstname #self ist Selbstreferenz
    self.lastname = lastname
    self.age = age

  def __str__(self): #Stringrepräsentation bei str(obj)
    return f"({self.firstname}, {self.lastname}, {self.age})"

  def __repr__(self): #Stringrepräsentation z.B. in Listen
    return str(self)

  def __eq__(self, other): #Gleichheits-Operator
    return self.lastname == other.lastname and \
        self.firstname == other.firstname

  def __hash__(self): #Hashfunktion z.B. für Dict-Keys
    return hash((self.firstname, self.lastname))
```

```
class Employe(Person): #Vererbung
  def __init__(self, firstname, lastname, age, id, salary):
    super().__init__(firstname, lastname, age) #Base-Klasse
    self.id = id
    self.salary = salary

  def __gt__(self, other): #Vergleichsoperator z.B. für sort()
    return self.salary < other.salary
```

8.7 Pandas

pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

Sie finden die Informationen des nächsten Kapitels und sehr viel extra Wissenswertes zum nachlesen unter `https://pandas.pydata.org/`.

Das Pandas Modul wird wie folgt in das eigene Skript geladen.

```
import pandas
```

Wir können mit dem Schlüsselwort `as` einen Alias definieren.

```
import pandas as pd
```

8.7.1 Datenstrukturen

- Eine `Series` ist ein eindimensionales gelabeltes Array, welches jeden möglichen Datentypen enthalten kann (z.B. Int, String, Float, Python-Objekte).
- Die Achsen werden als `index` bezeichnet.

Wir initialisieren eine `Series` Datenstruktur wie folgt:

```
s = pd.Series(data, index=index)
```

- `index` ist eine Liste von Labeln und muss die gleiche Länge wie `data` haben.
- `data` kann z.B. eine Liste sein, ein Skalarer Wert, ein Python-Dict oder vieles mehr.

`data` als Liste.

```
>>>pd.Series([1,2,3], index=["a", "b", "c"])
a    1
b    2
c    3
dtype: int64
```

`data` als Skalar.

```
>>>pd.Series(5, index=["a", "b", "c"])
a    5
b    5
c    5
dtype: int64
```

`data` als Python-Dict. Die Indices wählen die Werte aus. Existiert ein Wert nicht bekommen wir `nan` – not a number.

```
>>>d = {"a": 0.0, "b": 1.0, "c": 2.0}
>>>pd.Series(d, index=["b", "c", "d", "a"])
b    1.0
```

```
c    2.0
d    NaN
a    0.0
dtype: float64
```

Wir demonstrieren Zugriffe auf die Daten an folgenden Beispielen:

```
>>>s = pd.Series([1,2,3,4,5,6],
                 index=["a", "b", "c", "d", "e", "f"])
>>>s.iloc[2]    #Zugriff auf Integfer Location.
3
>>>s.iloc[2:]   #Slice
c    3
d    4
e    5
f    6
dtype: int64
>>>s[s>3]       #Boolscher Filter
d    4
e    5
f    6
dtype: int64
```

Wir demonstrieren Zugriffe auf die Daten an folgenden Beispielen:

```
>>>s = pd.Series([1,2,3,4,5,6],
                 index=["a", "b", "c", "d", "e", "f"])
>>>s.iloc[[5,3,2]]      #Liste von Zeilen-Indices
f    6
d    4
c    3
dtype: int64
>>>s["c"]           #Zugriff via Index
3
>>>s["e"] = 12.0    #Neuen Wert zuweisen.
>>>s
a     1
b     2
c     3
d     4
e    12
f     6
dtype: int64
```

Überlicherweise braucht man keine Schleifen um die Gesamtheit der Werte in einer `Series` zu ändern.

```
>>>s = pd.Series([1,2,3,4,5,6],
```

```
                  index=["a", "b", "c", "d", "e", "f"])
>>>s*2          #Jeden Wert mit 2 multiplizieren.
a     2
b     4
c     6
d     8
e    10
f    12
dtype: int64
>>>s.mean()     #Built-In Funktionen für den Mittelwert.
3.5
```

- Ein `DataFrame` bringt das Konzept einer `Series` auf ein 2-dimensionales Level.
- Wir bekommen benannte Spalten und Zeilen mit potenziell unterschiedlichen Datentypen in jeder Spalte.
- Vergleichbar zu z.B. Excel-Tabellen, SQL-Datenbanken, Dict von Series (alles was auf relationaler Algebra basiert).

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
```

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>pd.DataFrame(d, index=["d", "b", "a"])
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0
>>>pd.DataFrame(d, index=["d", "b", "a"],
                  columns=["two", "three"])
```

```
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

Wir betrachten den `DataFrame` semantisch und syntaktisch als Dict von `Series`-Objekten. So ergeben sich folgende Operationen auf `DataFrames`. Das Zugreifen und Hinzufügen von Spalten.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
        "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df["one"]    #Auswahl einer Spalte
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
>>>df["three"] = df["one"] * df["two"] #Neue Spalte
>>>df["flag"] = df["one"] > 2          #Neue Spalte
>>>df
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Das Löschen von Spalten.

```
>>>del df["two"]           #Spalte Löschen
>>>three = df.pop("three") #Spalte Löschen und Zurückgeben
>>>df
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

Das Einfügen eines Skalars.

```
>>>df["foo"] = "bar"
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
```

```
c  3.0   True  bar
d  NaN  False  bar
```

Das Einfügen einer `Series` die nicht über alle Indices wie der `DataFrame` verfügt.

```
>>>df["one_trunc"] = df["one"][:2]
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

- Spalte auswählen `df[col]` gibt eine `Series` zurück.
- Zeile durch Label auswählen `df.loc[label]` gibt eine `Series` zurück.
- Zeile durch Integer-Location auswählen `df.iloc[loc]` gibt eine `Series` zurück.
- Zeilen Slicen `df[5:10]` gibt einen DataFrame zurück.
- Zeilen durch boolschen Vektor selektieren `df[bool_vec]` gibt einen DataFrame zurück.

Lambda Funktionen sind kleine anonyme Funktionen, welche beliebig viele Argumente übergeben bekommen, aber nur einen einzigen Ausdruck auswerten. Die Syntax ist dabei: `lambda arg1, arg2, arg3 : expression`. Lambdas benutzt man meistens, wenn Funktionen als Argumente übergeben werden.

```
>>>l = lambda x1, x2: x1 + x2 + 5
>>>print(l(3,5))
13
```

Mit der Funktion `apply` können wir entlang der Spalten (Default) oder Zeilen (Parameter `axis=1` übergeben) eines DataFrames Funktionen anwenden. Die Methode wird auf einem DataFrame ausgeführt und bekommt eine Funktion (z.B. ein lambda) übergeben.

```
>>>df = pd.DataFrame([[1,2], [3,4], [5,6]], columns=['A', 'B'])
>>>df
   A  B
0  1  2
1  3  4
2  5  6
>>>df.apply(lambda x : x*x)
```

```
   A  B
0  1  4
1  9  16
2  25  36
```

Mit der Funktion `assign` können wir neue Spalten erstellen, welche potenziell aus bestehenden Spalten abgeleitet werden können. Die Funktion bekommt als Argumente Spalten-keys und deren Werte übergeben. Dabei können die Werte z.B. `Series` sein oder auch Funktionen, die neue Werte aus übergebenen Tupeln erstellen. Weiter rechts stehende Argumente können bereits auf Ergebnisse aus vorherigen Argumenten zugreifen.

```
>>>dfa = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]}
>>>dfa.assign(C=lambda x: x["A"] + x["B"],
              D=lambda x: x["A"] + x["C"])
   A  B  C   D
0  1  4  5   6
1  2  5  7   9
2  3  6  9  12
```

8.7.2 Analyse Funktionen

Mit `shape` bekommen wir ein 2-Tuple mit den Dimensionen unseres DataFrames zurück. Dabei ist `shape[0]` die Anzahl der Zeilen und `shape[1]` die Anzahl der Spalten.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
        "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.shape
(4, 2)
```

Mit den Funktionen `head()` (und `tail()`) bekommen wir einen kleinen Teil des DataFrames. Der übergebene Wert gibt dabei an wie viele Zeilen wir berücksichtigen wollen (Default-Wert 5). Mit `head(n)` (`tail(n)`) bekommen wir die ersten (letzten) n Zeilen.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
        "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
```

```
                        index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.head(2)
        one     two
a       1.0     1.0
b       2.0     2.0
```

Mit der Funktion `count()` kann die Anzahl der nicht-NA Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.count()
one    3
two    4
dtype: int64
```

Mit der Funktion `sum()` kann die Summe aller Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.sum()
one     6.0
two    10.0
dtype: float64
```

Mit der Funktion `mean()` kann der Durchschnitt aller Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.mean()
```

```
one    2.0
two    2.5
dtype: float64
```

Mit der Funktion `median()` kann der Median aller Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.median()
one    2.0
two    2.5
dtype: float64
```

Mit den Funktion `min()` (und `max()`) kann das Minimum (Maximum) aller Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.min()
one    1.0
two    1.0
dtype: float64
```

Mit der Funktion `std()` kann die Standardabweichung der Werte (z.B. der Spalten) berechnet werden.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.std()
one    1.000000
two    1.290994
dtype: float64
```

Mit der Funktion `unique()` bekommen wir aus einer Series ein Array aller einzigartigen Werte zurück. Wir werden nicht weiter auf Arrays eingehen. Für uns reicht aus, dass wir über diese wie Listen iterieren können.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 3.0, 4.0],
                      index=["a", "b", "c", "d", "e"])}
>>>df = pd.DataFrame(d)
>>>df["two"].unique()
array([1., 2., 3., 4.])
```

Mit `values` bekommen wir aus einer Series ein Array aller Werte zurück. Wir werden nicht weiter auf Arrays eingehen. Für uns reicht aus, dass wir über diese wie Listen iterieren können.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df["one"].values
array([ 1.,  2.,  3., nan])
```

Mit der Funktion `describe()` bekommt man eine statistische Übersicht (z.B. der Werte in den Spalten). Die Prozentzahlen stehen dabei für entsprechende Perzentile.

```
>>>d = { "one":
            pd.Series([1.0, 2.0, 3.0],
                      index=["a", "b", "c"]),
         "two":
            pd.Series([1.0, 2.0, 3.0, 4.0],
                      index=["a", "b", "c", "d"])}
>>>df = pd.DataFrame(d)
>>>df.describe()
count   3.0     4.000000
mean    2.0     2.500000
std     1.0     1.290994
min     1.0     1.000000
25%     1.5     1.750000
50%     2.0     2.500000
75%     2.5     3.250000
max     3.0     4.000000
```

8.7.3 Einlesen und Auffüllen

Eine CSV-Datei (comma-separated values) ist ein Format um Daten tabellarisch abzuspeichern.

- Eine Zeile wird mit `\n` abgeschlossen.
- Werte innerhalb der Zeile werden mit einem Delimiter/Separator (Default Wert `','`) getrennt.
- Die erste Zeile gibt meistens die Namen der Spalten an.

```
Index,one,two
a,1,1
b,2,2
c,3,3
d,,4
```

Mit der Pandas Funktion `read_csv()` können wir eine csv-Datei direkt in einen DataFrame laden.

- Das erste Argument ist der Pfad zur Datei.
- Mit `sep=...` kann man einen Separator/Delimiter spezifizieren.
- Mit `index_col="Index"` kann man einen Spalte zum Index promoten.

```
>>>df = pd.read_csv("Test.csv", sep=',', index_col="Index")
>>>df
        one     two
Index
a       1.0     1
b       2.0     2
c       3.0     3
d       NaN     4
```

Mit den Funktion `fillna()`,`ffill()` und `bfill()` können wir fehlenden Werten neue Werte zuweisen.

- Wir können z.B. mit `fillna(n)` alle fehlenden Werte durch einen Default-Wert n ersetzen.
- Den letzten/nächsten validen Werte benutzen (`ffill()` und `bfill()`).

```
>>>df = pd.read_csv("Test.csv", sep=',', index_col="Index")
>>>df.fillna(0)
        one     two
Index
a       1.0     1
b       2.0     2
```

```
c       3.0     3
d       0.0     4
```

8.7.4 Kombinieren von Daten

Die Funktion `concat()` konkateniert Series und DataFrame Objekte entlang der angegebenen Achse und kann dabei die Menge der jeweiligen Indices schneiden oder vereinigen. Die Funktion bekommt eine Liste oder ein Dictionary mit kompatiblen Typen (Series, DataFrame) übergeben.

```
>>>df1 = pd.DataFrame({"A": ["A0", "A1"],
                        "B": ["B0", "B1"],
                        "C": ["C0", "C1"],
                        "D": ["D0", "D1"],}, index=[0, 1])

>>>df2 = pd.DataFrame({"A": ["A2", "A3"],
                        "B": ["B2", "B3"],
                        "C": ["C2", "C3"],
                        "D": ["D2", "D3"],}, index=[2, 3])

>>>pd.concat([df1, df2])
   A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
2  A2  B2  C2  D2
3  A3  B3  C3  D3
```

Mit dem Keyword `join` wird spezifiziert was mit jenen Achsen-Werten gemacht werden soll, welche nicht im ersten DataFrame vorkommen. Mit `join="outer"` wird die Vereinigung aller Werte gebildet.

```
>>>df1 = pd.DataFrame({"A": ["A0", "A1"],
                        "B": ["B0", "B1"],
                        "C": ["C0", "C1"],
                        "D": ["D0", "D1"],}, index=[0, 1])

>>>df3 = pd.DataFrame({"A": ["A1", "A3"],
                        "B": ["B1", "B3"],
                        "C": ["C1", "C3"],
                        "D": ["D1", "D3"],}, index=[1, 3])

>>>pd.concat([df1,df3], axis=1, join="outer")
   A   B   C   D   A   B   C   D
0  A0  B0  C0  D0  NaN NaN NaN NaN
1  A1  B1  C1  D1  A1  B1  C1  D1
3  NaN NaN NaN NaN A3  B3  C3  D3
```

Mit dem Keyword `join` wird spezifiziert was mit jenen Achsen-Werten gemacht werden soll, welche nicht im ersten DataFrame vorkommen. Mit `join="inner"` wird der Schnitt aller Werte gebildet.

```
>>>df1 = pd.DataFrame({"A": ["A0", "A1"],
                        "B": ["B0", "B1"],
                        "C": ["C0", "C1"],
                        "D": ["D0", "D1"],}, index=[0, 1])

>>>df3 = pd.DataFrame({"A": ["A1", "A3"],
                        "B": ["B1", "B3"],
                        "C": ["C1", "C3"],
                        "D": ["D1", "D3"],}, index=[1, 3])

>>>pd.concat([df1,df3], axis=1, join="inner")
  A  B  C  D  A  B  C  D
1 A1 B1 C1 D1 A1 B1 C1 D1
```

Es kann eine Series-Objekt als Spalte an einen DataFrame angefügt werden.

```
>>>df1 = pd.DataFrame({"A": ["A0", "A1"],
                        "B": ["B0", "B1"],
                        "C": ["C0", "C1"],
                        "D": ["D0", "D1"],}, index=[0, 1])

>>>column = pd.Series(["E0", "E1"], index=[0,1], name="E")
>>>pd.concat([df1,column], axis=1)
   A   B   C   D   E
0  A0  B0  C0  D0  E0
1  A1  B1  C1  D1  E1
```

Es kann eine Series-Objekt als Zeile an einen DataFrame angefügt werden. Dazu wird aus dem Series-Objekt mit `to_frame()` ein DataFrame-Objekt und dieses dann mit `.T` transponiert.

```
>>>df1 = pd.DataFrame({"A": ["A0", "A1"],
                        "B": ["B0", "B1"],
                        "C": ["C0", "C1"],
                        "D": ["D0", "D1"],}, index=[0, 1])

>>>row = pd.Series(["A2", "B2", "C3", "D3"],
                    index=["A","B","C","D"], name="3")
>>>pd.concat([df1,row.to_frame().T], axis=0)
         A       B       C       D
0       A0      B0      C0      D0
1       A1      B1      C1      D1
3       A2      B2      C3      D3
```

Mit `merge(left, right)` kombiniert man zwei DataFrame-Objekte wie in relationalen Datenbanken z.B. SQL.

- Spalten auswählen, welche als Keys zum Zusammenfügen benutzt werden.
- Key-Kombinationen, welche mehrfach in den Tabellen vorkommen erzeugen ein kartesisches Produkt.
- Mit `how=` wird angegeben, welche Keys in der erzeugten Tabelle vorkommen.
- Wenn ein Key in einer der Tabellen nicht vorkommt, werden entsprechende `NA` hinzugefügt.

Methoden für `how=<methode>` umfassen:

- Mit `left` werden Keys aus dem linken DataFrame benutzt.
- Mit `right` werden Keys aus dem rechten DataFrame benutzt.
- Mit `outer` werden Keys aus der Vereinigung beider benutzt.
- Mit `inner` werden Keys aus dem Schnitt beider benutzt.
- Mit `cross` wir das kartesische Produkt aller Einträge gebildet.

```
>>>left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                        "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})

>>>right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
                         "key2": ["K0", "K0", "K0", "K0"],
                         "C": ["C0", "C1", "C2", "C3"],
                         "D": ["D0", "D1", "D2", "D3"]})
>>>pd.merge(left, right, how="left", on=["key1", "key2"])
        key1    key2    A       B       C       D
0       K0      K0      A0      B0      C0      D0
1       K0      K1      A1      B1      NaN     NaN
2       K1      K0      A2      B2      C1      D1
3       K1      K0      A2      B2      C2      D2
4       K2      K1      A3      B3      NaN     NaN
```

```
>>>left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                        "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})

>>>right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
```

```
                          "key2": ["K0", "K0", "K0", "K0"],
                          "C": ["C0", "C1", "C2", "C3"],
                          "D": ["D0", "D1", "D2", "D3"]})
>>>pd.merge(left, right, how="right", on=["key1", "key2"])
        key1    key2    A       B       C       D
0       K0      K0      A0      B0      C0      D0
1       K1      K0      A2      B2      C1      D1
2       K1      K0      A2      B2      C2      D2
3       K2      K0      NaN     NaN     C3      D3
```

```
>>>left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                        "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})

>>>right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
                         "key2": ["K0", "K0", "K0", "K0"],
                         "C": ["C0", "C1", "C2", "C3"],
                         "D": ["D0", "D1", "D2", "D3"]})
>>>pd.merge(left, right, how="outer", on=["key1", "key2"])
        key1    key2    A       B       C       D
0       K0      K0      A0      B0      C0      D0
1       K0      K1      A1      B1      NaN     NaN
2       K1      K0      A2      B2      C1      D1
3       K1      K0      A2      B2      C2      D2
4       K2      K0      NaN     NaN     C3      D3
5       K2      K1      A3      B3      NaN     NaN
```

```
>>>left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                        "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})

>>>right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
                         "key2": ["K0", "K0", "K0", "K0"],
                         "C": ["C0", "C1", "C2", "C3"],
                         "D": ["D0", "D1", "D2", "D3"]})
>>>pd.merge(left, right, how="inner", on=["key1", "key2"])
        key1    key2    A       B       C       D
0       K0      K0      A0      B0      C0      D0
1       K1      K0      A2      B2      C1      D1
2       K1      K0      A2      B2      C2      D2
```

```
>>>left = pd.DataFrame({"key1": ["K0", "K0", "K1", "K2"],
                        "key2": ["K0", "K1", "K0", "K1"],
                        "A": ["A0", "A1", "A2", "A3"],
                        "B": ["B0", "B1", "B2", "B3"]})
```

```
>>>right = pd.DataFrame({"key1": ["K0", "K1", "K1", "K2"],
                         "key2": ["K0", "K0", "K0", "K0"],
                         "C": ["C0", "C1", "C2", "C3"],
                         "D": ["D0", "D1", "D2", "D3"]})
>>>pd.merge(left, right, how="cross")
  key1_x key2_x A   B   key1_y  key2_y  C   D
0 K0     K0     A0  B0  K0      K0      C0  D0
1 K0     K0     A0  B0  K1      K0      C1  D1
2 K0     K0     A0  B0  K1      K0      C2  D2
3 K0     K0     A0  B0  K2      K0      C3  D3
...
```

8.7.5 Group by: split-apply-combine

Mit Group by wird generell ein Prozess bezeichnet, welcher ein oder mehrere der folgenden Schritte beinhaltet:

- **Splitting**: Teilen der Daten nach gegebenen Kriterien in Gruppen.
- **Applying**: Funktionen auf einzelne Gruppen anwenden.
- **Combining**: Ergebnisse in geeigneter Weise abspeichern.

Im Schritt Applying können z.B. folgende Dinge gemacht werden:

- **Aggregation**: Zusammenfassungen von den Daten der Gesamten Gruppe (z.B. Anzahl Einträge, Summe, Median).
- **Transformation**: Gruppenspezifische Berechnungen/Anpassungen (z.B. NA's gruppenweise ersetzen).
- **Filtration**: Gruppen oder Daten ausfiltern, auf Basis von Gruppeneigenschaften.

Es soll für jedes Label festgelegt werden zu welchem Gruppennamen es zugeordnet wird:

- Durch eine Funktion.
- Ein Dictionary, welches zu jedem Label einen Gruppennamen ausgibt.
- Einträge von Spalten/Zeilen, welche gleichzeitig den Gruppennamen definieren.
- Listen von oben genannten Dingen.

```
>>>speeds = pd.DataFrame([("bird", "Falconiformes", 389.0),
                          ("bird", "Psittaciformes", 24.0),
                          ("mammal", "Carnivora", 80.2),
```

```
                        ("mammal", "Primates"),
                        ("mammal", "Carnivora", 58)],
      index=["falcon", "parrot", "lion", "monkey", "leopard"],
      columns=("class", "order", "max_speed"))

>>>speeds.groupby("class")
<pandas.core.groupby.generic.DataFrameGroupBy object at ...>
>>>speeds.groupby("class").max()
        order   max_speed
class
bird    Psittaciformes  389.0
mammal  Primates        80.2
```

Iteration durch Gruppennamen und entsprechende DataFrames. Dazu das Beispiel von den vorherigen Folien:

```
>>>groups = speeds.groupby("class")
>>>for c,g in groups:
>>>... print(c)
>>>... print(g)
bird
       class           order  max_speed
falcon  bird   Falconiformes      389.0
parrot  bird  Psittaciformes       24.0
mammal
          class      order  max_speed
lion     mammal  Carnivora       80.2
monkey   mammal   Primates        NaN
leopard  mammal  Carnivora       58.0
```

Zugriff auf eine einzelne Gruppe. Dazu das Beispiel von den vorherigen Folien:

```
>>>groups = speeds.groupby("class")
>>>groups.get_group("mammal")
        class   order   max_speed
lion    mammal  Carnivora       80.2
monkey  mammal  Primates        NaN
leopard         mammal  Carnivora       58.0
```

Eine Aggregation ist eine Operation, welche die Dimension des Gruppen-Objekts reduziert. Das Ergebnis einer Aggregation ist meistens ein Wert für die gesamte Gruppe (z.B. Summe, Maximum, Minimum).

```
>>>groups = speeds.groupby("class")
>>>groups.max()
Selection deleted
groups.max()
        order   max_speed
```

```
class
bird    Psittaciformes  389.0
mammal  Primates        80.2
```

Bekannte Built-In Aggregation-Funktionen:

- `count()` berechnet die Anzahl nicht-NA Einträge jeder Gruppe.
- `max()` berechnet das Maximum in jeder Gruppe (analog Minimum).
- `mean()` berechnet den Durchschnitt der Werte in jeder Gruppe.
- `median()` berechnet den Median der Werte in jeder Gruppe.
- `nunique()` berechnet die Anzahl einzigartiger Werte in jeder Gruppe.
- für viele weitere Built-In Funktionen kann man in die Dokumentation schauen.

Transformationen sind Group by Operationen, bei denen die ursprüngliche Indexierung beibehalten wird und nicht die neue Indexierung der Gruppennamen benutzt wird.

```
>>>groups = speeds.groupby("class")["max_speed"]
>>>groups.bfill()
falcon     389.0
parrot      24.0
lion        80.2
monkey      58.0
leopard     58.0
Name: max_speed, dtype: float64
```

Bekannte Built-In Transformationen-Funktionen:

- `bfill()` berechnet Werte für NA Werte innerhalb der Gruppe (analog `ffill()`).
- `cummax()` berechnet das kummulative Maximum in jeder Gruppe (analog Minimum).
- `cumsum()` berechnet die kummulative Summe in jeder Gruppe.
- `diff()` berechnet die Differenz benachbarter Werte.
- für viele weitere Built-In Funktionen kann man in die Dokumentation schauen.

Filtrationen sind Group by Operationen, bei denen ganze Gruppen, Teile von Gruppen oder beides ausgefiltert werden. Eine Filtration gibt eine gefilterte Version des aufrufenden Objektes zurück.

Bekannnte Built-In Funktionen sind:

- `head()` wählt die oberste(n) Zeile(n) jeder Gruppe aus.
- `tail()` wählt die unterste(n) Zeile(n) jeder Gruppe aus.
- `nth()` wählt die n-te Zeile jeder Gruppe aus.

```
>>>speeds.groupby("class").nth(1)
         class           order  max_speed
parrot    bird  Psittaciformes       24.0
monkey  mammal        Primates        NaN
```

Mit der `filter()` Funktion kann eine Nutzerdefinierte Funktion bekommen und ganze Gruppen auf Grundlage dieser Funktion zu `True` oder `False` evaluieren. Alle zu `True` evaluierten Gruppen werden zurückgegeben.

```
>>>sf = pd.Series([1, 1, 2, 3, 3, 3])
>>>sf.groupby(sf).filter(lambda x: x.sum() > 2)
3    3
4    3
5    3
dtype: int64
```

9 Kryptographie

9.1 Einführung

Literatur

A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter
Moderne Verfahren der Kryptographie,
Vieweg+Teubner, Mai 2010.

C. Damm
Kryptographie,
Skript zur Vorlesung, 2004.

B. Schneier, N. Ferguson
Practical Cryptography,
John Wiley & Sons, 2003.

A. J. Menezes, P. C. Van Oorschot, S. A. Vanstone
Handbook of Applied Cryptography,
Crc Press, Oktober 1996.

Geheimhaltung

Wie kann man mit jemanden vertraulich kommunizieren, d.h. kein Unbeteiligter soll Kenntnis von der übermittelten Nachricht erhalten?

Das Problem der Übermittlung (und Speicherung) geheimer Nachrichten kann man durch verschiedene Maßnahmen lösen.

- Organisatorische Maßnahmen
- Physikalische Maßnahmen
- Kryptographische Maßnahmen

Organisatorische Maßnahmen

Beispiel

- Ein Gespräch während eines einsamen Waldspaziergangs.

- Übermittlung einer Nachricht durch einen vertrauenswürdigen Boten.
- Einstufung vertraulicher Dokumente als Verschlusssache.

Physikalische Maßnahmen

Beispiel

- Verstecken der Informationen in einem Tresor.
- Übermitteln der Nachricht in einem versiegelten Brief.
- Verheimlichen der Existenz der Nachricht, z.B. durch Geheimtinte.

Kryptographische Maßnahmen

Kryptographische Maßnahmen verändern bzw. entstellen (**verschlüsseln**, **chiffrieren**, *encrypt*) die Nachricht bzw. den **Klartext** (*plaintext*).

Dadurch ist die Nachricht für einen Außenstehenden nicht mehr erkennbar und die übertragene Information, der **Geheimtext** (*ciphertext*), erscheint diesem (meist) völlig unsinnig.

Ein berechtigter Empfänger kann den Klartext aber (leicht) wieder herstellen (**entschlüsseln**, **dechiffrieren**, *decrypt*).

Der älteste Zweig der klassischen Kryptographie beschäftigt sich mit der **Geheimhaltung** von Nachrichten **durch Verschlüsselung**.

Codierung

Klassische Verschlüsselungsalgorithmus arbeiten meistens mit Nachrichten einer natürlichen, d.h. von Menschen gesprochenen, Sprache.

Diese Nachrichten enthalten Formatierungen (z.B. Groß- und Kleinschreibung) und zusätzliche Zeichen (z.B. Interpunktionszeichen), die für das Verständnis der Nachricht nicht notwendig sind.

Unter der **Codierung** (*encoding*) einer Nachricht versteht man ihre Umwandlung von einem Alphabet in ein anderes Alphabet.

Der Aufwand für die Ver- und Entschlüsselung steigt mit der Größe des Alphabets, deshalb wird bei den meisten klassischen Verschlüsselungsalgorithmus die Nachricht zu einem Klartext über einem kleineren Alphabet codiert.

Bemerkung

Moderne Verschlüsselungsalgorithmen arbeiten mit Bit-Blöcken/Streams. Digitale Daten sind i.d.R. schon in dieser Form codiert. Z.B. ergibt sich die zu einem Text gehörige Bitfolge, aus der verwendeten Zeichencodierung (ASCII, UTF-8, etc.).

Beispiel

Die Funktion **encode** wandelt alle Buchstaben in Großbuchstaben um und entfernt alle nicht in **alphabet** enthaltenden Zeichen.

```
1 alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2
3 def encode(text):
4     e = ""
5     for t in text:
6         if (t.upper() in alphabet):
7             e = e + t.upper()
8     return e
```

Die Nachricht

```
XV. Maerz Caesar treffen, Dolche nicht vergessen
```

wird so zum Klartext

```
XVMAERZCAESARTREFFENDOLCHENICHTVERGESSEN
```

codiert.

9.2 Symmetrische Verschlüsselung

Bei der **symmetrischen Verschlüsselung** besitzen der Sender und die berechtigten Empfänger eine gemeinsame geheime Zusatzinformation (den **Schlüssel**), darin unterscheiden sie sich von den Außenstehenden.

Derselbe Schlüssel wird sowohl vom Sender zum Verschlüsseln des Klartext verwendet, als auch vom Empfänger für das Entschlüsseln des Geheimtext benötigt.

Beim Sender ist das kein Problem, da er den Schlüssel gewählt/erzeugt hat.

Dem Empfänger fehlt dieser Schlüssel erstmal, deswegen ist es bei der symmetrischen Verschlüsselung sehr wichtig, dass der Schlüssel auf einem sicheren Übertragungsweg an den Empfänger weitervermittelt wird.

Anschauung

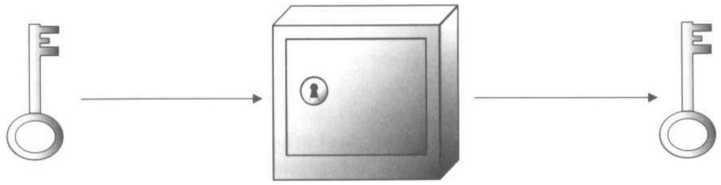


Abbildung 5: Symmetrische Verschlüsselung, Anschauung
(Quelle: A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter; Moderne Verfahren der Kryptographie)

Verschlüsseln schützt die Nachricht davor gelesen zu werden.

Man kann sich vorstellen, dass der Sender die Nachricht in einen Tresor legt und mit Hilfe seines Schlüssels abschließt.

Der Tresor wird samt Inhalt an den Empfänger geschickt. Dieser hat einen identischen Schlüssel, um den Tresor zu öffnen und die Nachricht zu lesen.

In der Kryptographie werden die Nachrichten nicht durch physikalische Maßnahmen geschützt, sondern durch mathematische Methoden.

Verschlüsselungsalgorithmus

Definition 9.1. Ein **symmetrischer Verschlüsselungsalgorithmus** besteht aus einer Funktion f mit zwei Eingabewerten, dem **Schlüssel** k und dem **Klartext** m, die Ausgabe ist der **Geheimtext** c, der sich aus k und m ergibt.

Hinweis

Da Kryptographie/Kryptoanalyse mit Wahrscheinlichkeitrechnung verzahnt ist, wo das Symbol p (*probability*) häufig benutzt wird, wird für Klartext das Symbol m (*plaintext message*) verwendet.

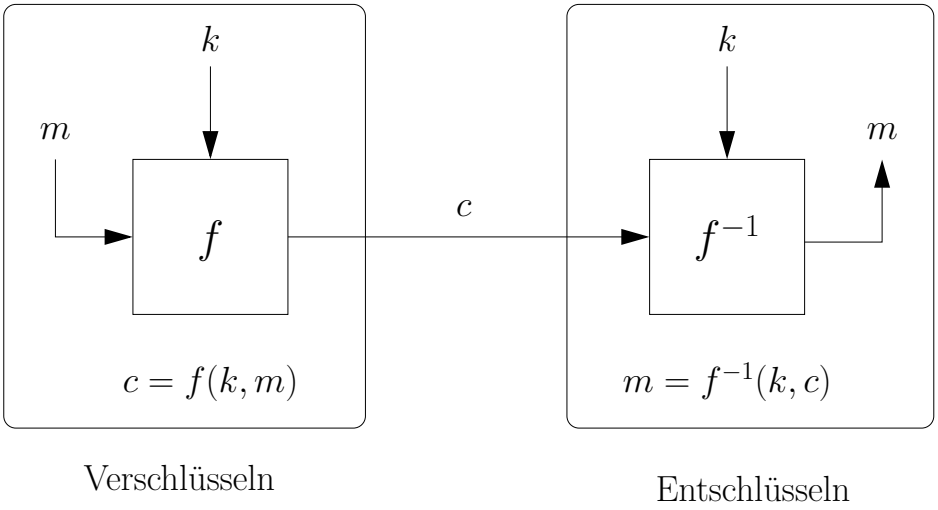
Die Verschlüsselungsfunktion f muss **umkehrbar** sein, d.h. es muss eine Funktion f^{-1} geben, die die Wirkung von f rückgängig macht.

Mit dem Schlüssel k und dem Gemeintext c kann man mit f^{-1} den Klartext m rekonstruieren.

Angenommen Sender und Empfänger benutzen den gemeinsamen (geheimen) Schlüssel k.

- Der Sender verschlüsselt einen Klartext m, indem er den Gemeintext c berechnet, $c = f(k, m)$ (oft auch $f_k(m)$).
- Der Empfänger rekonstruiert den Klartext m, indem er den Geheimtext c entschlüsselt, $m = f^{-1}(k, c) = f_k^{-1}(c)$

Funktionsschema



Verschlüsseln und Entschlüsseln müssen in einer sicheren Umgebung stattfinden, das wird im Bild durch die Kästen symbolisiert.

9.3 Blockchiffren und Stromchiffren

Verschlüsselungsverfahren

In der Regel sollen große Nachrichten oder Nachrichtenströme, d.h. eine kontinuierliche Abfolge von Zeichen, verschlüsselt werden, was mit einer einmalige Anwendung der Verschlüsselungsfunktion nicht umsetztbar ist.

Blockchiffren und **Stromchiffren** sind Verfahren für die Anwendung von Verschlüsselungsalgorithmen auf große Nachrichten oder Nachrichtenströme.

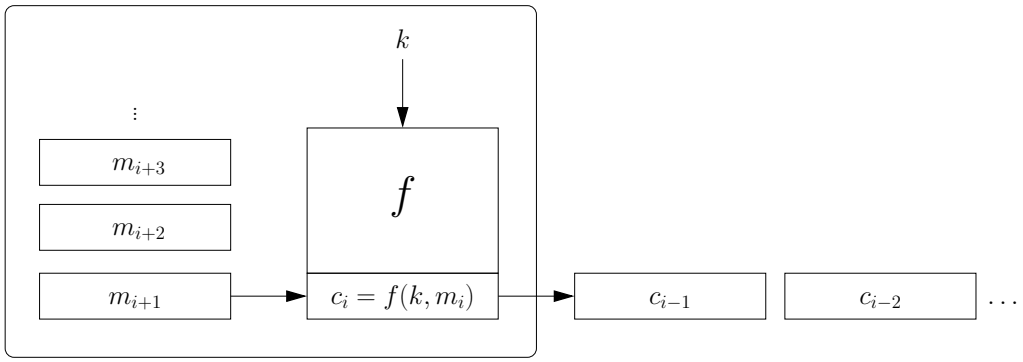
Beide Verschlüsselungsverfahren teilen den Klartext in Blöcke gleicher Größe $m_1, m_2, \ldots$ auf.

Durch die Verschlüsselung entstehen Geheimtextblöcke $c_1, c_2, \ldots$ gleicher Größe.

Blockchiffre

- Die Klartextblöcke werden unabhängig voreinander mit der Verschlüsselungsfunktion und **demselben Schlüssel** einzeln zu Geheimtextblöcken chiffriert.
- Die Geheimtextblöcke werden entsprechend einzeln dechiffriert.

Blockchiffre mit Verschlüsselungsfunktion f und Schlüssel k.



Verschlüsseln

Caesar-Verschlüsselung

Die **Caesar-Verschlüsselung** ist eine Blockchiffren mit symmetrischem Verschlüsselungsalgorithmus.

Ein Klartext m kann eine beliebige Zeichenfolge über dem Alphabet $\Sigma = \{A, B, \ldots, Z\}$ der 26 Großbuchstaben sein, $m \in \Sigma^*$.

Ein Klartext wird in Blöcke der Größe ein Zeichen aufgeteilt.

Der Schlüssel k ist ein Zeichen aus dem Alphabet Σ.

Die symmetrische Verschlüsselungsfunktion $f_k : \Sigma \to \Sigma$ verschiebt das übergebene Zeichen im Alphabet zyklisch nach **rechts**, dabei wir die Anzahl der zu verschiebenden Stellen von der Position des Schlüssels k im Alphabet bestimmt.

Die symmetrische Entschlüsselungsfunktion $f_k^{-1} : \Sigma \to \Sigma$ verschiebt entsprechend, abhängig von k, das übergebene Zeichen im Alphabet zyklisch nach **links**.

Beispiel

Caesar-Verschlüsselung mit Schlüssel C.

Die Position von Schlüssel C im Alphabt ist 3, d.h f_C (f_C^{-1}) verschiebt ein übergebenes Zeichen um 3 Stellen im Alphabet zyklisch nach rechts (links).

$$
\begin{array}{ll}
f_C(A) = D & f_C^{-1}(D) = A \\
f_C(B) = E & f_C^{-1}(E) = B \\
\ldots & \ldots \\
f_C(W) = Z & f_C^{-1}(Z) = W \\
f_C(X) = A & f_C^{-1}(A) = X \\
f_C(Y) = B & f_C^{-1}(B) = Y \\
f_C(Z) = C & f_C^{-1}(C) = Z
\end{array}
$$

CAESAR wird zeichenweise mit C verschlüsselt zu *FDHVDU*, das wird zeichenweise mit C entschlüsselt wieder zu *CAESAR*.

Beispiel - Python

Caesar-Verschlüsselung in Python

```
1  alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2
3  def char_to_index(c):
4      return (alphabet.index(c.upper()))
5
6  def index_to_char(n):
7      return alphabet[n % len(alphabet)]
8
9  def caesar(key, plain, deciphering=False):
10     k = char_to_index(key) + 1
11     if (deciphering): k = -k
12     c = ""
13     for p in plain:
14         if (p not in alphabet):
```

```
15              raise ValueError('"' + p + '" is not an alphabet charater')
16          c = c + index_to_char(char_to_index(p) + k)
17      return c
```

Funktionen können mit Parametern definiert werde, für die ein Standardwert (*default value*) festgelegt wird.

Wird für diese Parameter beim Aufruf der Funktion ein Argument übergeben, referenziert der Parameter in der Funktion dieses Argument. Wird kein Argument übergeben, referenziert der Parameter den Standardwert.

```
def caesar(key, plain, deciphering=False):
```

Wenn ein Block, der durch einen Doppelpunkt eingeleitet wird, nur aus einer Ausweisung besteht, kann diese Anweisung direkt in die Zeile mit dem Doppelpunkt geschrieben werden.

```
if (deciphering): k = -k
```

Python unterstützt **Exceptions**.

Auch wenn ein Ausdruck syntaktisch korrekt ist, kann bei dessen Ausführung einen Fehler auftreten. Ein Exception ist so ein Fehler, der vom Laufzeitsystem erkannt wurde.

Exception können im Code selbst behandelt werden, passiert das nicht behandelt sie das Laufzeitsystem, üblicherweise mit einer Fehlermeldungen, inklusive nützlicher Zusatzinformationen, z.B. der Codezeile, die die Exception verursacht hat.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Es gibt *built-in exceptions*, z.B. **`ValueError`**, das sind Klassen von den man, mit einem Konstruktor gleichen Namens, Objekte erzeugen kann.

Dem Konstruktor von **`ValueError`** kann man als Argument noch eine String, z.B. einer Fehlermeldung, übergeben

```
>>> e=ValueError("unknown reason")
>>> e
ValueError('unknown reason')
>>> type(e)
<class 'ValueError'>
>>> print(e)
unknown reason
```

Mit der *built-in function* **raise** kann man eine Exception auslösen.

```
>>> e=ValueError("unknown reason")
>>> raise(e)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: unknown reason
```

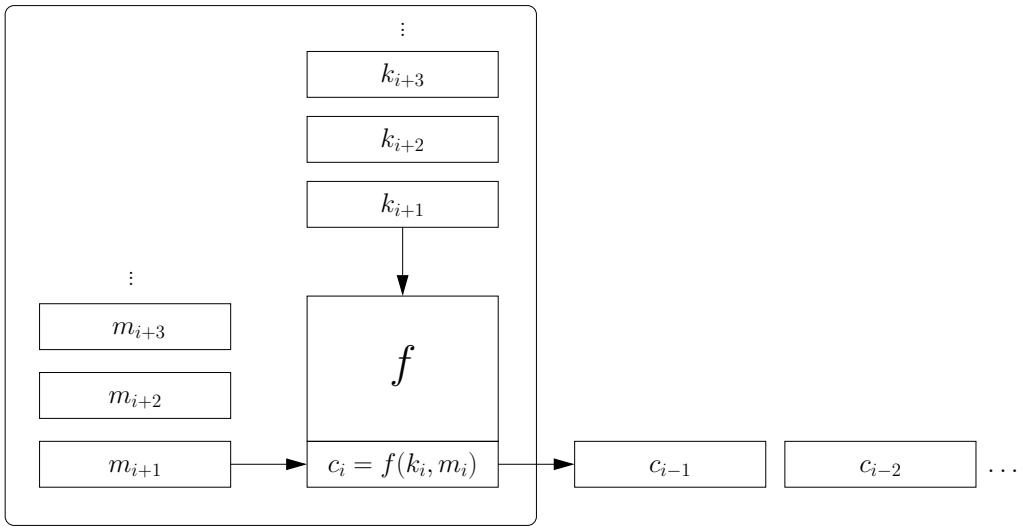
Bemerkung

Wie man Exceptions behandelt wird hier nicht besprochen, siehe dazu *Python Documentation, 8. Errors and Exceptions* `https://docs.python.org/3/tutorial/errors.html`.

Stromchiffre

- Eine **Schlüsselstrom** $k_1, k_2, \ldots$ wird erzeugt, sodass für jeden Klartextblock ein eigener Schlüssel vorliegt.
- Ein Klartextblock wird jeweils mit der Verschlüsselungsfunktion und dem zugehörigen Schlüssel des Schlüsselstroms zu einem Geheimtextblock chiffriert.
- Ein Geheimtextblock kann nur mit den zugehörigen Schlüssel des Schlüsselstroms dechiffriert werden.

Stromchiffre mit Verschlüsselungsfunktion f und Schlüsselstrom $k_1, k_2, \ldots$.



Verschlüsseln

One-Time-Pad

One-Time-Pad (wörtlich *Einmal-Block*) ist eine Stromchiffre für (große) Nachrichten bekannter Länge mit symmetrischem Verschlüsselungsalgorithmus.

Es werden **zufällig** genauso viele Schlüssel erzeugt, wie Klartextblöcke zu verschlüsseln sind, dabei haben die Schlüssel mindestens die gleiche Länge wie die Klartextblöcke.

Jeder Schlüssel wird nur **einmal** verwendet, um einen Klartextblock mit der Verschlüsselungsfunktion in einen Geheimtextblock zu chiffrieren.

Bemerkung

Da es für jeden Klartextblock einen Schlüssel gibt, die Schlüssel zufällig gewählt und nur einmal verwendet werden, kann man zeigen, dass One-Time-Pad ein absolut sicheres Verfahren ist.

Bemerkung

One-Time-Password (*Einmalkennwort*) ist ein anderes Verfahren, das z.B. durch Transaktionsnummern-Listen (TAN-Listen) beim Online-Banking Anwendung findet.

Beispiel

Für ein Umsetzung von One-Time-Pad wird die Verschlüsselungsfunktion $f_k : \Sigma \to \Sigma$ des Caesar-Verschlüsselung mit Alphabet $\Sigma = \{A, B, \ldots, Z\}$ benutzt.

Um den, aus der Nachricht

`XV. Maerz Caesar treffen, Dolche nicht vergessen`

codierten, Klartext

`XVMAERZCAESARTREFFENDOLCHENICHTVERGESSEN`

zu verschlüsseln wird eine ebenso lange zufällige Schlüsselfolge

`CTOSHBFEXKKSSMPJMYEXOPFRBYNZASDTZUXTSAWP`

erzeugt, z.B. durch Beobachtung der Bewegung von Blättern auf dem *Forum Romanum*.

Die Zeichen des Klartext werden einzeln mit der Verschlüsselungsfunktion f_k und jeweils einem Zeichen k der Schlüsselfolge zum Geheimtext

`APBTMTFHYPDTKGHOSEJLSERUJDBIDAXPEMEYLTBD`

chiffriert.

Beispiel - Python

One-Time-Pad mit Caesar-Verschlüsselung in Python

```
1 def onetimepad(key, plain, deciphering=False):
2     if (len(key) != len(plain)):
3         raise IndexError("length of key and plaintext did not match")
4     c = ""
5     for k, p in zip(key, plain):
6         c = c + caesar(k, p, deciphering)
7     return c
```

Die *built-in function* **zip** nimmt als Argumente n iterierbare Objekte entgegen und fasst diese in einem neuen iterierbaren Objekte von n-Tupeln zusammen.

Das i-te Tupel enthält das i-te Objekt aus jedem der n übergebenen Argumente. Die Anzahl der Tupel entspricht der Anzahl Elemente des kürzesten Arguments.

```
>>> z=zip("ABCD", "123", "ZYXW")
>>> list(z)
[('A', '1', 'Z'), ('B', '2', 'Y'), ('C', '3', 'X')]
```

Mehrere (gleichlange) iterierbare Objekte kann man mit **for**, Mehrfachbelegung (*multiple assignment*) und **zip** komfortable gleichzeitig durchlaufen.

```
>>> for c, x in zip("ABC", "123"):
...     print(f"{c} {x}")
...
A 1
B 2
C 3
```

Schlüsselerzeugung

Zum Austausch von verschlüsselten Nachrichten ist One-Time-Pad mit diesem Verfahren der Schlüsselerzeugung aber nur bedingt geeignet, denn zum Entschlüsseln braucht man die Schlüsselfolge, die genauso lang wie der Klartext ist.

Die Schlüsselfolge vorher, für einer später zu versendende Nachricht, auszutauschen oder parallel zum Geheimtext (z.B. durch zwei unabhängige Boten) auszutauschen ist denkbar, dadurch degeneriert das Verfahren aber zu einer Blockchiffre mit einem großen Block und einem großen Schlüssel.

Benötigt wird ein Algorithmus, der aus einem (kleinen) Initial-Schlüssel einen (großen) Schlüssel für One-Time-Pad berechnet, der möglichst nicht von einem wirklich zufällig erzeugten Schlüssel unterschieden werden kann.

9.4 Pseudozufall

In der Kryptographie spielen Zufallszahlen und Zufallsfolgen eine wichtige Rolle.

Dabei gibt es verschiedenen Aspekte.

- In vielen Kommunikationsprotokollen muss an einer bestimmten Stelle einen zufälligen Wert wählen. Dabei hängt die Sicherheit des Protokolls direkt davon ab **wie zufällig** der Wert gewählt wurde.

Im eigenen Interesse müssen die Kommunikationspartner darauf achten den Wert mit einem möglichst guten Zufallsgenerator zu wählen.

- Oft ist es nicht praktikabel oder sogar unmöglich echte Zufallszahlen und Zufallsfolgen zu verwenden. Das ist z.B. dann der Fall, wenn die Zahlen oder Folgen von mehreren erzeugt werden müssen.

  In diesen Fällen werden von Folgen von **Pseudozufallszahlen** verwendet, die mit Hilfe eines deterministischen Algorithmus berechnet werden, aber nur sehr schwer von wirklicher Zufälligkeit unterschieden werden können.

Echte Zufallszahlen und Zufallsfolgen werden mit Hilfe physikalischer Phänomene erzeugt, z.B. mit Hilfe des Rauschens elektronischer Bauelemente, dem radioaktiven Zerfall oder dem Konvektionsströmungen in einer Lavalampe.

Ein klassisches Beispiel für das Erzeugen echte Zufallsfolgen ist das Werfen einer *fairen Münze*.

Bei der Beurteilung der Güte eines Zufallsfolgengenerators ist ein Problem, dass man die Zufälligkeit einer Folge **nicht** beweisen kann.

Die nicht Zufälligkeit einer Folge lässt sich beweisen, durch Angabe eines Algorithmus zur Erzeugung der Folge.

Bei Pseudozufallsfolgen steht man prinzipiell vor dem gleichen Problem. Es gibt kein Kriterien um echte Zufallsfolgen zu bewerten, deshalb kann man nicht entscheinden wie zufällig eine Pseudozufallsfolge aussieht.

Wiederum kann man nur das Gegenteil nachweisen, nämlich dass eine Pseudozufallsfolge nicht zufällig aussieht.

Linearer Kongruenzgenerator

In den Bibliotheken vieler Programmiersprachen ist ein **linearer Kongruenzgenerator** (*linear congruential generator*) zur Erzeugung von Pseudozufallen implementiert.

Ein linearer Kongruenzgenerator berechnet eine Folge $x_1, x_2, x_3, \ldots$ von Pseudozufallen aus den Parametern $m, a, c, x \in \mathbb{N}$.

- Modul (*modulus*) $m > 0$

- Faktor (*multiplier*) $0 < a < m$
- Inkrement (*increment*) $0 \leq c < m$
- Startwert (*seed*) $0 \leq x < m$

Es gilt

$$x_i = ax_{i-1} + c \mod m$$

mit $x_0 = x$ für $i > 0$.

Bemerkung

Mathematisch korrekt handelt es sich mit $c > 0$ um einen affinen Abbildung, aber die Bezeichnung linearer Kongruenzgenerator hat sich etabliert.

Standardkonfiguration

Für die unterschiedlichen Implementierungen gibt es ebenso viele Standardkonfiguration, die gute Pseudozufallsfolgen produzieren sollen.

Die Funktionen `lrand48`/`nrand48` der C-Standardbibliothek (POSIX+glibc) und `java.util.Random` benutzen folgende Konfiguration.

- $m = 2^{48}$
- $a = 25214903917$
- $c = 11$
- berechnet werde jeweils 48-Bit, aber zurückgegeben werden nur die 32 höchstwertigen Bits

Beispiel - Python

Linearer Kongruenzgenerator in Python

```
1 def lcg(m, a, c, seed=0, cut=0, count=1):
2     k=[]
3     x=seed
4     for _ in range(count):
5         y=(a*x+c)%m
6         k.append(y//2**cut)
7         x=y
8     return k
```

Wenn man **lcg** mit der vorstehenden Konfiguration, sowie **seed=5**, **cut=16** ($2^{48}/2^{16} = 2^{32}$) und **count=10** aufruft

```
print(lcg(2**48, 25214903917, 11, 5, 16, 10))
```

bekommt man folgende 10 Pseudozufallszahlen.

```
[1923744, 2816743950, 832832900, 735168418, 2203812749,
1955956184, 776948653, 3341115479, 2902097218, 3472087760]
```

In der **for**-Schleife wird der Unterstrich _ als Platzhalter für die Schleifenvariable benutzt, weil diese im Rumpf der Schleife nicht benötigt wird.

Man kann mit Standwerten belegten Parametern

```
def lcg(m, a, c, seed=0, cut=0, count=1):
```

Argumente nicht nur über die Position in der Argumentliste (*positional argument*),

```
print(lcg(2**8, 37, 11, 7))
```

sondern auch über den Namen des Parameters (*keyword argument*)

```
print(lcg(2**8, 37, 11, seed=7, count=257))
```

zuordnen.

Nach dem ersten *keyword argument*, können nur noch solche Argumente folgen. Die Reihenfolge der *keyword arguments* ist beliebig.

```
print(lcg(2**8, 37, 11, count=257, seed=7, cut=2))
```

Bewertung

Linearen Kongruenzgeneratoren können Pseudozufallszahlen mit guten statistischen Eigenschaften erzeugten, eignen sich aber für die meisten kryptographischen Anwendungen nicht.

Man kann, mit vertretbarem Aufwand, aus wenigen Werten alle künftigen berechnen ohne den *seed* zu kennen.

Kennt man z.B. die aufeinanderfolgende Werte x, y, z und den Modul m, dann erhält man a und c durch Auflösen der beiden folgenden Gleichungen.

$$y = ax + c \mod m$$
$$z = ay + c \mod m$$

Mit a und c kann man alle auf z folgenden Werte berechnen.

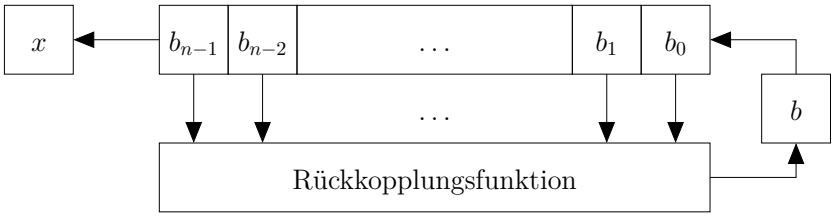
Bemerkungen

- Nicht alle Konfigurationen sind geeignet gute Pseudozufallszahlen zu erzeugen.
- Manche Konfigurationen erschweren das Auflösen der Gleichungen, aber prinzipiell bleibt das ein lösbares Problem.

Lineare Schieberegister mit Rückkopplung

Lineare Schieberegister werden häufig in der Kryptographie verwendet, da sie leicht in digitalen Schaltwerken zu realisieren sind.

Die Schieberegister mit Rückkopplung (*linear feedback shift register, LSR*) bestehen aus zwei Teilen, dem Schieberegister und der Rückkopplungsfunktion.



Die Funktion eines n-stelligen Schieberegisters.

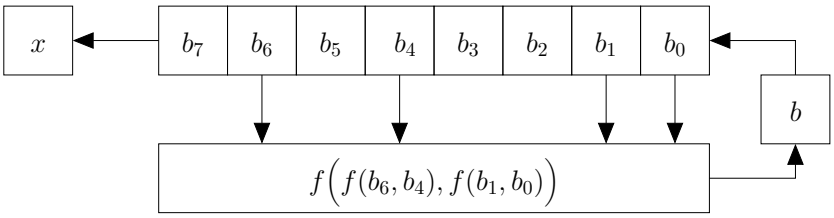
- Die Ausgabe des Schieberegister ist $x = b_{n-1}$.
- Der neue Wert b wird in Abhängigkeit von $b_{n-1}, \ldots, b_0$ und der Rückkopplungsfunktion berechnet.
- Die Einträge des Schieberegister werden mit folgender Regel nach links verschoben $b_{i+1} = b_i$ für $i = 0, \ldots, n-2$.
- Von links wird b in das Schieberegister hinein geschoben $b_0 = b$.

Eine Rückkopplungsfunktion kann mehr oder weniger kompliziert sein.

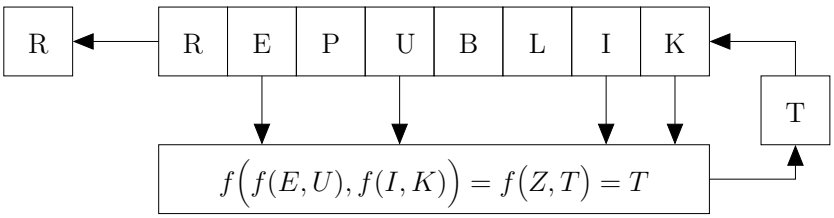
Beispiel

Sei, wie im One-Time-Pad-Beispiel,

- $\Sigma = \{A, B, \ldots, Z\}$ das Alphabet,
- $f(k, m) : \Sigma \times \Sigma \to \Sigma$ die Verschlüsselungsfunktion der Caesar-Verschlüsselung.



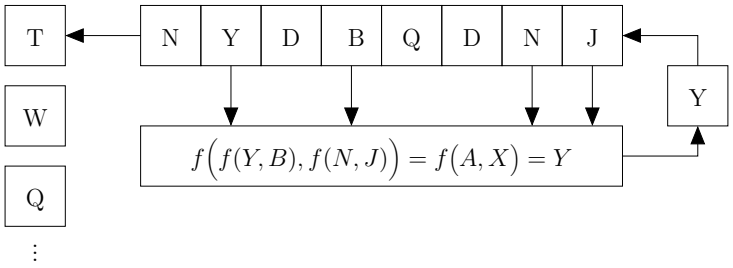
Initialisierung des Schieberegisters mit dem Schlüssel `REPUBLIK`.



Dann ergibt sich der Schlüsselstrom

```
REPUBLIKTWXFAJSUQKRWTAAKTQIUPFXGQWOZBQWT
```

und folgender Zustand des Schieberegisters.



Mit dem Schlüssel `REPUBLIK` können mit Hilfe eines lineare Schieberegisters mit (öffentlich) bekannter Rückkopplungsfunktion der Sender und jeder berechtigte Empfänger einen identischen Schlüsselstrom

```
REPUBLIKTWXFAJSUQKRWTAAKTQIUPFXGQWOZBQWT
```

erzeugen mit dem der Sender den Klartext

```
XVMAERZCAESARTREFFENDOLCHENICHTVERGESSEN
```

zum Geheimtext

```
PACVGDINUBQGSDKZWQWKXPMNBVWDSNRCVOVEUJBH
```

verschlüsseln kann und die berechtigten Empfänger den Klartext wiederherstellen können.

Um so besser die Pseudozufallszahlen, um schwerer fällt es einem Angreifer den Geheimtext zu entschlüsseln (siehe One-Time-Pad).

Linear/nicht linear

Folgen, die mit linearen Schieberegistern erzeugt werden haben sehr gute statistische Eigenschaften, d.h. die einzelnen Buchstaben, Bits, etc. sind in einer ausreichend großen Stichprobe gleichmäßig verteilt.

Trotzdem kann bei dieser einfachen Erzeugung des Schlüsselstroms schon aus relativ kurzen bekannten Schlüsselfolgen (nicht Geheimtexten) der Schlüssel mit vertretbarem Aufwand berechnet werden, weil ein **lineares** Verfahren benutzt wurde.

In der Praxis werden deshalb **nicht lineare** Verfahren verwendet, d.h. entweder werden nicht lineare Schieberegister (mit komplexerer Rückkopplungsfunktion) benutzt oder es werden mehrere lineare Schieberegister nicht linear gekoppelt.

9.5 Kryptosystem

Ein **Kryptosystem** besteht aus

- Endlichen Alphabeten
  - Γ_M der Klartextzeichen
  - $\Sigma_M \subseteq \Gamma_M^*$ der Klartextsymbole
  - Γ_C der Geheimtextzeichen
  - $\Sigma_C \subseteq \Gamma_C^*$ der Geheimtextsymbole
- Klartextmenge $M \subseteq \Sigma_M^*$
- Geheimtextmenge $C \subseteq \Sigma_C^*$
- Schlüsselmenge $K \neq \emptyset$
- Verschlüsselungsabbildungen $e : M \times K \to C$
- Entschlüsselungsabbildungen $d : C \times K \to M$
- Es gilt, dass für jedes $k' \in K$ (Schlüssel) ein $k'' \in K$ (Gegenschlüssel) existiert, sodass gilt $d(e(m, k'), k'') = m$ für alle $m \in M$.

Ein Kryptosystem heißt **symmetrisch**, wenn Schlüssel und Gegenschlüssel gleich sind, d.h. für alle $k \in K$ gilt $d(e(m, k), k) = m$ für alle $m \in M$.

Asymmetrische Kryptosystem, bei denen sich Schlüssel und Gegenschlüssel unterscheiden, sind erst seit Ende der siebziger Jahre bekannt.

Kryptosysteme werden auch nach der Wirkung von Verschlüsselungs- und Entschlüsselungsabbildungen unterschieden.

Nachfolgend werden Substitutions-Chiffren und Transpositions-Chiffren betrachtet.

Transposition

Bei einer **Transpositions-Chiffre** bleiben die Symbole *was* sie sind, aber nicht *wo* sie sind.

Abhängig vom Schlüssel wird die Position der Symbole mit der Permutation π vertauscht.

$$m_1 m_2 \dots m_t \mapsto m_{\pi(1)} m_{\pi(2)} \dots m_{\pi(t)} = c_1 c_2 \dots c_t$$

Beispiel. Matrixtransposition

Bei der **Matrixtransposition** wird der Klartext in Zeilen gleicher Länge angeordnet (unvollständige Zeilen werden aufgefüllt).

Zur Bildung des Geheimtextes wird der so angeordnete Text spaltenweise zurückgeschrieben.

Beispiel

Klartext zeilenweise (Zeilenlänge 5) anordnen.

```
XVMAE
RZCAE
SARTR
EFFEN
DOLCH
ENICH
TVERG
ESSEN
```

Geheimtext spaltenweise zurückschreiben.

```
XRSEDETEVZAFONVSMCRFLIESAATECCREEERNHHGN
```

Substitution

Bei einer **Substitutions-Chiffre** bleiben die Symbole *wo* sie sind, aber nicht *was* sie sind.

Substitutions-Chiffren erzeugen die Verschlüsselungsabbildungen mit Substitutionen (Ersetzungen).

Eine Substitution ist eine **injektive Abbildung**,

$$s_k : \Sigma_M \to \Sigma_C \quad \text{für } k \in K$$

die Symbole des Klartextalphabets auf Symbole des Geheimtextalphabets abbildet.

Es gilt Klartextmenge $M = \Sigma_M^*$ und Geheimtextmenge $C = \Sigma_M^*$.

Monoalphabetische Substitution

- Benutzung eine Alphabets (historisch), d.h. Benutzung genau einer Substitution, somit werden an jeder Stelle des Klartext gleiche Klartextsymbole durch dieselben Geheimtextsymbole ersetzt.
- Die Substitution

  $$s_k : \Sigma_M \to \Sigma_C \quad \text{für } k \in K$$

  induziert eine Verschlüsselungsabbildung

  $$e(m_1 m_2 \dots m_t, k) = s_k(m_1) s_k(m_2) \dots s_k(m_t) = c_1 c_2 \dots c_t$$

Polyalphabetische Substitution

- Benutzung mehrerer Alphabete (historisch), d.h. wechselnde Benutzung mehrere Substitutionen. Das kann man über die Anwendung einer Substitution, für die die verschiedenen Alphabete verschmolzen werden, mit unterschiedlichen Schlüsseln erreicht werden.
- Die Substitution

  $$s_k : \Sigma_M \to \Sigma_C \quad \text{für } k \in K$$

  und die Schlüsselmenge $k = \{k_1, \dots, k_p\} \subseteq K$ induziert eine Verschlüsselungsabbildung

  $$e(m_1 m_2 \dots m_t, k) = s_{\alpha(1)}(m_1) s_{\alpha(2)}(m_2) \dots s_{\alpha(t)}(m_t) = c_1 c_2 \dots c_t$$

  mit der Abbildung $\alpha : \{1, \dots, t\} \to \{k_1, \dots, k_p\}$.

Monographische Substitution

- Ein Klartextzeichen entspricht einem Klartextsymbol, d.h. $\Sigma_M = \Gamma_M$.

Beispiele

- Strom/Blockchiffren bei denen die Blöcke jeweils einzelne Zeichen sind

- Caesar-Chiffre ist eine monoalphabetische und monographische Substitution.

Polygraphische Substitution

- Die Klartextsymbole umfassen jeweils mehrere Zeichenpaare, Zeichentripel, etc. (allgemein **n-Gramme**), d.h. $\Sigma_M = \underbrace{\Gamma_M \times \ldots \times \Gamma_M}_{\text{n-mal}} = \Gamma_M^n$.

Beispiel. Vigenère, One-Time-Pad.

Die Vigenère-Chiffre stammt aus dem 16. Jahrhundert und wurde von dem französischen Kryptographen Blaise de Vigenère (1523-1596) entwickelt.

Basiert auf der Verwendung der Caesar-Chiffre, allerdings mit periodisch wechselnden Schlüsseln.

Galt lange Zeit als nicht zu knacken, insbesondere war das Ermitteln der Schlüssellänge problematisch, und konnte erst um 1850 entziffert werden.

Beispiel

Klartext: XVMAERZC AESARTRE FFENDOLC HENICHTV ERGESSEN
Schlüsselfolge: REPUBLIK REPUBLIK REPUBLIK REPUBLIK REPUBLIK
Geheimtext: PACVGDIN SJIVTFAP XKUIFAUN ZJDDETCG WWWZUENY

Beispiel - Python

Vigenère-Chiffre mit Caesar-Verschlüsselung in Python

```
1 def vigenere(key, plain, deciphering=False):
2     c=""
3     i=0
4     for p in plain:
5         c=c+caesar(key[i], p, d)
6         i=(i+1)%len(key)
7     return c
```

9.6 Kryptoanalyse

Die Dechiffrierung ohne Kenntnis der Geheiminformation, das *Brechen der Chiffre*, wird **Kryptoanalyse** genannt.

Grundannahme (**Kerckhoffs' Maxime**, 1883)

Die Sicherheit einer Chiffre darf nicht darauf beruhen, dass der Angreifer (Kryptoanalyst) das benutzte Verfahren nicht kennt.

Angriffsarten und Analysemethoden

Bei allen Angriffen ist die Verschlüsselungsfunktion e bekannt.

Das Klartextblöcke m' mit der Verschlüsselungsfunktion e und einen Schlüssel k zu Geheimtextblöcken c' verschlüsselt werden, d.h. $c' = e(m', k)$, ist ebenfalls bekannt.

ciphertext-only

- Beobachtet: Geheimtext c
- Gesucht:
  - Klartext m

  - Schlüssel k
  - Blocklänge des Geheimtexts, d.h. $c_1 \dots c_j = c$ (wenn nicht bekannt)
  - Algorithmus, um $m_{j+1}, \dots$ aus $c_{j+1}, \dots$ herzuleiten

Beispiel. Ein Lauscher (englisch *eavesdropper*), z.B. in einem Netzwerk.

known-plaintext

- Beobachtet: Klartext $m_1 \dots m_j$ und zugehöriger Geheimtext $c_1 \dots c_j$
- Gesucht
  - Schlüssel k
  - Algorithmus, um $m_{j+1}, \dots$ aus $c_{j+1}, \dots$ herzuleiten

Beispiel. Wiederkehrende Anfangs- und Schlußformeln.

chosen-plaintext

- Wähle Klartext $m_1 \dots m_j$, beobachte zugehörigen Geheimtext $c_1 \dots c_j$
- Gesucht
  - Schlüssel k
  - Algorithmus, um $m_{j+1}, \dots$ aus $c_{j+1}, \dots$ herzuleiten

adaptive chosen-plaintext

- Wähle m_1, beobachte c_1, wähle m_2, usw.

Beispiel.

- Freund-Feind-Erkennung (*challenge response protocol*)
- Public-Key-Systeme

Sichere/unsichere Kryptosystem

System heißt **sicher** gegen bestimmten Angriffstyp, wenn ein potentieller Angreifer die erforderlichen Berechnungen nicht mit **vertretbarem Aufwand** durchführen kann.

Nachweis der Sicherheit ist schwierig ⇒ Rückführung auf anerkannt schwierige Probleme

- Faktorisierung großer Zahlen
- NP-vollständige Probleme
- ...

Vorsicht. Schwierige Probleme können auch einfache Instanzen haben!

In der Regel ist es einfacher, die Unsicherheit eines Systems gegen bestimmte Angriffe nachzuweisen.

Offenbar unsicher gegen *chosen-plaintext*-Angriffe sind

- Caesar,
- Vigenère,
- Matrixtransposition.

Diese Verfahren sind ebenfalls unsicher gegen *ciphertext-only*-Angriffe, wenn **genügend viel** Geheimtext vorhanden und der Klartextraum hinreichend gut bekannt ist.

Klartextraum

Kenntnisse des Angreifers über den Klartextraum.

- Sprache (natürliche Sprache, Programmiersprache, ...)
- häufige Wörter (Kontext!)
- Sprachstatistik (Symbol-, Bigramm-, Trigramm-, ..., -Häufigkeiten)
- Buchstabenmuster (z.B. 1221 im Englischen: cabbage, ballast, apparant, ...)
- Randinformationen

- ...

Häufigkeitsmerkmale prägen natürliche Sprachen sehr stark ⇒ wichtigster Einstiegspunkt für Kryptoanalyse

Nützlich sind **Häufigkeitsreihenfolgen**, i.d.R. aber allein nicht ausreichend.

deutsch (verschiedene Quellen):

`enrisdutaghlobmfzkcwvjpqxy` (1840)
`enirsahtudlcgmwfbozkpjvqxy` (1863)
...
`enisratduhglcmwobfzkvpjqxy` (1955)

englisch (verschiedene Quellen):

`etaoinshrdlucmfwypvbgkqjxz` (1884)
`etoanirshdlcufmpywgbvkxjqz` (1893)
...
`etaoinsrhldcumfpgwybvkxjqz` (1982)

9.7 Asymmetrische Verschlüsselung

Die **asymmetrische Verschlüsselung** verwendet zwei unterschiedliche Schlüssel.

Es wird ein **Öffentlicher** und ein **privater** Schlüssel verwendet, die zueinander komplementär sind.

Daten, die mit dem Öffentlichen Schlüssel verschlüsselt werden (verschlüsseln von Nachrichten), können mit dem private Schlüssel entschlüsselt werden.

Daten, die mit dem privaten Schlüssel verschlüsselt werden (signieren von Nachrichten), können mit dem öffentlichen Schlüssel entschlüsselt werden.

Entscheidend bei der asymmetrischen Verschlüsselung ist, dass der private Schlüssel nicht aus dem öffentlichen Schlüssel abgeleitet werden kann.

Bei der asymmetrischen Verschlüsselung erzeugt jeder Teilnehmer ein Schlüsselpaar, behält den privaten Schlüssel und machten den öffentlichen Schlüssel den anderen Teilnehmern zugänglich.

Für den Austausch von verschlüsselten Nachrichten chiffriert der Sender die Daten mit dem öffentlichen Schlüssel des Empfängers und schickt sie diesem. Der Empfänger kann die Daten anschließend mit Hilfe seines privaten Schlüssels dechiffrieren.

Anschauung

Man kann einem Empfänger eine verschlüsselte Nachricht schicken, ohne eine Geheiminformation zu besitzen.

Das kann man sich wie das Einwerfen einer Nachricht in einen Briefkasten vorstellen.

Jeder, der Zugang zum Briefkasten (dem öffentlichen Schlüssel) hat, kann eine Nachricht einwerfen (Verschlüsseln).

Nur der Empfänger kann mit seinem privaten Schlüssel den Briefkasten öffnen und die Nachricht entnehmen (Entschlüsseln).

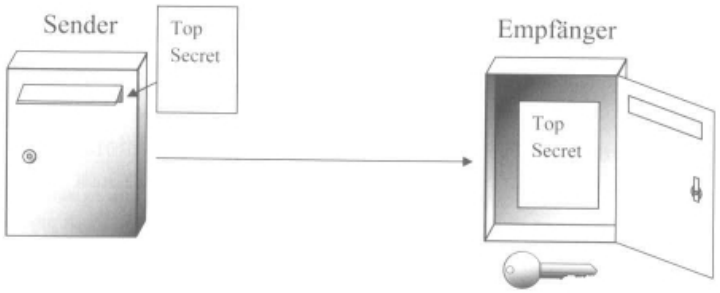


Abbildung 6: Asymmetrische Verschlüsselung, Anschauung
(Quelle: A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter; Moderne Verfahren der Kryptographie)

Verschlüsselungsalgorithmus

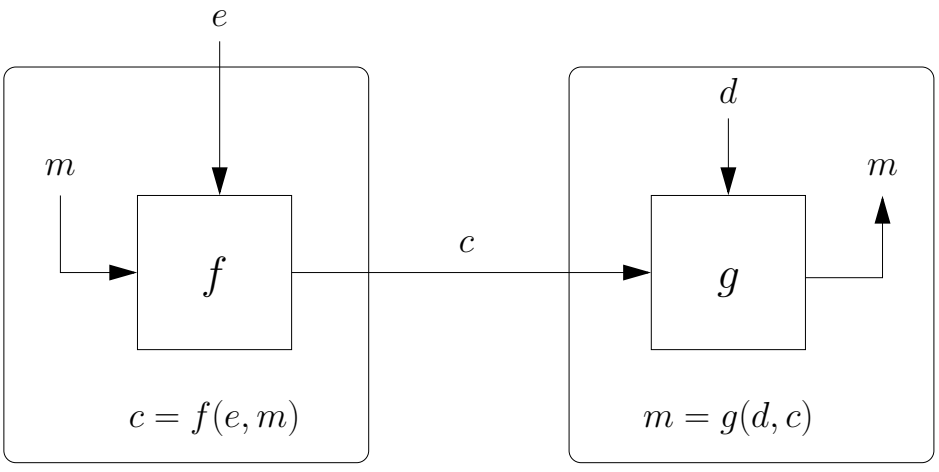
Definition 9.2. Ein **asymmetrischer Verschlüsselungsalgorithmus** besteht aus

- einer Funktion f mit zwei Eingabewerten, dem **öffentlichen Schlüssel** e und einem Text m, die Ausgabe ist der Text $f(e, m)$,
- einer Funktion g mit zwei Eingabewerten, dem **privaten Schlüssel** d und einem Text m, die Ausgabe ist der Text $g(d, m)$.

Für alle Texte m gelten die folgenden Beziehungen zwischen f und g.

$$g\Big(d, f(e, m)\Big) = m$$
$$f\Big(e, g(d, m)\Big) = m$$

Funktionsschema



Verschlüsseln
öffentlicher Schlüssel

Entschlüsseln
privater Schlüssel

Einwegfunktion

Eine **Einwegfunktion** ist eine Funktion, die einfach auszuführen, aber schwer (nur mit sehr großem Aufwand) zu invertieren ist.

Etwas formaler ist eine Einwegfunktion $f : X \to Y$ eine Abbildung für zwei Mengen X, Y, sodass Folgendes gilt.

- Für alle $x \in X$ ist $f(x)$ leicht zu berechnen.
- Für (fast) jedes $y \in Y$ ist es schwer ein Urbild $x \in X$, d.h. ein $x \in X$ mit $y = f(x)$, zu finden.

Einwegfunktionen spielen in der theoretischen und der praktischen Kryptographie eine entscheidende Rolle.

Beispiel

Die Funktion, die über Nachschlagen in einem gedruckten Telefonbuch einem Namen eine Telefonnummer zuordnet ist leicht auszuführen, da die Namen im Telefonbuch alphabetisch geordnet sind.

Allerdings ist die Umkehrung, die Zuordnung einer Telefonnummer zu einem Namen, mit Hilfe eine gedruckten Telefonbuchs ein sehr schwieriges Unterfangen.

Trapdoor-Einwegfunktion

Einwegfunktionen finden in der Kryptographie Verwendung, wenn alle Beteiligten die Funktion anwenden dürfen und kein Beteiligter die Umkehrfunktion kennt bzw. ermitteln kann, z.B. zur Integritätsprüfung von Daten.

Für die asymmetrische Verschlüsselung benötigt man ein erweitertes Konzept, da für Verschlüsseln und Entschlüsseln, sowohl Funktion als auch Umkehrfunktion benötigt werden.

Eine **Trapdoor-Einwegfunktion** ist eine Einwegfunktion, also eine eigentlich schwer zu invertierende Funktion, es sei denn man kennt die Zusatzinformation (die *trapdoor* zu Deutsch *Falltür*, im Deutschen oft *Hintertür*), mit deren Hilfe man die Funktion leicht invertieren kann.

Beispiel

Ist $n \in \mathbb{N}$ (groß und) **keine** Primzahl, dann ist das Quadrieren modulo n

$$\begin{aligned} f : \mathbb{N} &\to \{0, \ldots, n-1\} \\ x &\mapsto x^2 \mod n \end{aligned}$$

eine Einwegfunktion.

Seien p, q zwei (große) Primzahlen und $n = pq$, dann ist das Quadrieren modulo n eine Trapdoor-Einwegfunktion.

Die Trapdoor ist in diesem Fall die Faktorisierung von n, also die Faktoren p, q. Mit dieser Information ist das Invertieren einfach möglich.

Allgemein kann man zeigen, dass für zwei (große) Primzahlen p, q und $n = pq$ die Potenzfunktion

$$\begin{aligned} f : \mathbb{N} &\to \{0, \ldots, n-1\} \\ x &\mapsto x^k \mod n \end{aligned}$$

für beliebiges $k > 1$ eine Trapdoor-Einwegfunktion ist, mit Trapdoor p, q.

Ein weitere Trapdoor-Einwegfunktion ist die Exponentialfunktion.

Für zwei (große) Primzahlen p, q und $n = pq$ ist

$$\begin{aligned} f : \mathbb{N} \times \mathbb{N} &\to \{0, \ldots, n-1\} \\ f(m, x) &= m^x \mod n \end{aligned}$$

eine Trapdoor-Einwegfunktion, mit Trapdoor p, q.

Grundlagen

Satz 9.3. *Sei $n = pq$ das Produkt zweier Primzahlen $p, q \in \mathbb{N}$.*

Dann gilt für alle natürlichen Zahl m, k mit $m < n$ folgende Gleichung.

$$m^{k(p-1)(q-1)+1} \mod n = m$$

Rechenregeln

Für natürliche Zahlen $x, i, j, n \in \mathbb{N}$ gilt Folgendes.

$$\left(x^i \mod n\right)^j \mod n = \left(x^i\right)^j \mod n = x^{i \cdot j} \mod n$$

RSA-Algorithmus

Der RSA-Algorithmus wurde 1978 von Rivest, Shamir and Adleman erfunden, der Algorithmus basiert auf folgendem Prinzip.

Sei $n = pq$ das Produkt zweier Primzahlen $p, q \in \mathbb{N}$.

Wähle bzw. berechne die Schlüssel $e, d \in \mathbb{N}$, sodass für ein $k \in \mathbb{N}$ gilt

$$e \cdot d = k(p-1)(q-1) + 1 \ .$$

Dann gilt das Folgende für jedes $m \in \mathbb{N}$ mit $m < n$.

$$\begin{aligned} \left(m^e \mod n\right)^d \mod n &= m^{e \cdot d} \mod n = m \\ \left(m^d \mod n\right)^e \mod n &= m^{e \cdot d} \mod n = m \end{aligned}$$

Somit sind die Funktionen f und g identisch.

$$f(r, m) = g(r, m) = m^r \mod n$$

RSA-Schlüssel

Der **erweiterte euklidische Algorithmus** liefert für zwei natürliche Zahlen $a, b \in \mathbb{N}$ den größten gemeinsamen Teiler und zwei weitere ganze Zahlen $x, y \in \mathbb{Z}$, sodass folgendes gilt.

$$\gcd(a, b) = xa + yb$$

Dieser Algorithmus wird bei der Berechnung der Schlüssel verwendet.

Gesucht

- (e, n) öffentlicher Schlüssel (*public key*)
- (d, n) privater Schlüssel (*private key*)

Wähle zwei (große) ungleiche Primzahlen p, q und berechne den RSA-Modul $n = pq$.

Wähle eine Zahl e für die gilt

- $1 < e < (p-1)(q-1) - 1$
- e und $(p-1)(q-1)$ sind teilerfremd
- $\gcd\Big(e, (p-1)(q-1)\Big) = xe + y(p-1)(q-1)$ mit $x > 0$

Bemerkung. Eulersche φ-Funktion $\varphi(n) = (p-1)(q-1)$

Aus $xe + y(p-1)(q-1) = 1$ mit $x > 0, y < 0$ folgt

$$\begin{aligned} & xe + y(p-1)(q-1) = 1 \\ \iff \quad & xe = (-y)(p-1)(q-1) + 1 \end{aligned}$$

d.h. mit $d = x \in \mathbb{N}$ und $k = -y \in \mathbb{N}$ gilt

$$e \cdot d = k(p-1)(q-1) + 1$$

somit ist ein Schlüsselpaar

- (e, n) öffentlicher Schlüssel (*public key*)
- (d, n) privater Schlüssel (*private key*)

gefunden.