Grundlagen der Praktischen Informatik

Basierend auf dem Skript von Henrik Brosenne

Stefan Siemer
Georg-August-Universität Göttingen
Institut für Informatik

Sommersemester 2024

Inhaltsverzeichnis

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~~

Saalübung
ab zweiter Vorlesungswoche.

Dienstag	etwa 16:00 – 17:30 Uhr	MN 08

Übungsgruppen IfI, wöchentlich
ab dritter Vorlesungswoche.

Termine und Organisation der Übungen werden noch bekannt gegeben.

Lehrmanagement/Kommunikation

Die Vorlesung und die Übungen werden über das

- Das Lehrmanagementsystem *Stud.IP*.
- Die Kommunikationsplattform *GWDG RocketChat*.

abgewickelt.

Stud.IP/RocketChat

Tragen sie sich in Stud.IP in die Veranstaltung *Grundlagen der Praktischen Informatik (Informatik II)* im SoSe 2024 ein.

- Die Folien und Videos von Vorlesung und Saalübung werden dort hinterlegt.
- Die Übungen werden dort ausgegeben.
- Einige Übungen erfordern die Benutzung von dem E-Lernmodul ILIAS.

Melden Sie sich in GWDG RocketChat in folgenden Kannälen an.

- `https://chat.gwdg.de/channel/GdPI-2024SoSe-news`
  zum Lesen der Ankündigungen zur Veranstaltung
  Invite Link: `https://chat.gwdg.de/invite/sRxFTD`
- `https://chat.gwdg.de/channel/GdPI-2024SoSe-stud`
  zum Lesen und Schreiben von Fragen zu Vorlesung und Übung
  Invite Link: `https://chat.gwdg.de/invite/yuHPWv`

Übung - FlexNow

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

Die Anmeldung zur Übung in FlexNow ist **Voraussetzung** für die

- **Teilnahme am Übungsbetrieb**.
- Die **Anmeldung zur Klausur**.

Die vorläufige **An- und Abmeldefrist** für die Übung in FlexNow ist

Dienstag, 30.04.2024, 23:55 Uhr.

Melden Sie sich **rechtzeitig** in **FlexNow** an. Wir benutzen die Daten im FlexNow vom 15.04. um mit der Gruppenaufteilung anzufangen. Wer bis dahin nicht angemeldet ist hat weniger Freiheiten bei der Auswahl der Gruppe.

Modulprüfung

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

E-Klausur

90 Minuten

E-Prüfungsraum, MZG 1.116

07.08.2024, 14-18 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.

Zulassungsbedingung zur Klausur 50% der Punkte aus allen Übungen und damit erfolgreiche Absolvierung von B.Inf.1102.Ue.

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 (eventuell).
- 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, Pandas zur Datenauswertung)
7. Kryptographie

Themenüberblick (Änderungen)

Änderung zum Vorjahr:

1. Telematik

Geplante Änderungen zum nächsten Jahr:

1. Rechnermodelle
2. Betriebssysteme
3. Parser mit Haskell
4. Pandas und Logik ausbauen

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 Wertebereich 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 einen 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 Funktion

Konstante Funktion werden durch explizite Angabe des Werts definiert.

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

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

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

Bemerkung

`--` leitet ein 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 ein konstanten 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, das es sich um einen Gleitkomma-Typ handelt.

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

Ein 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.

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

Alternativ kann man in 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, Abbildung der Argumente aus dem Definitionsbereich durch Funktion/Operator auf einen Wert des Wertebereich, 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 möglichest effiziente Auswertung zu erreichen, z.B. soll eine Funktion mit identischen Argumenten nur einmal ausgewertet werden.

Pattern können auch weniger allgemein gültig 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 unter 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]*

Anhängig 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

Definition 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 Definition, 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 Definition 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 Version 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 Ausnahmen (*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)*.

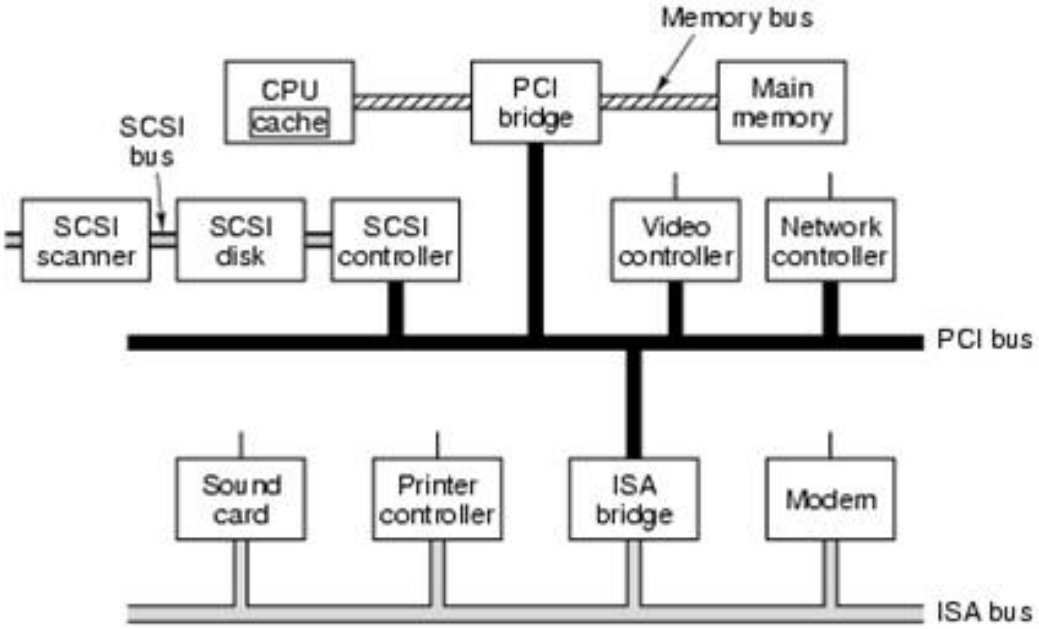
Das **Ein-/Ausgabewerk** steuert die Ein- und Ausgabe von Daten.

- 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.

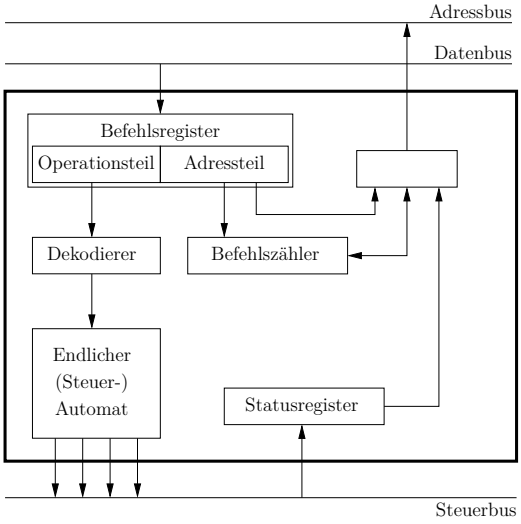


PCI (Peripheral Component Interconnect) Bus.

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.

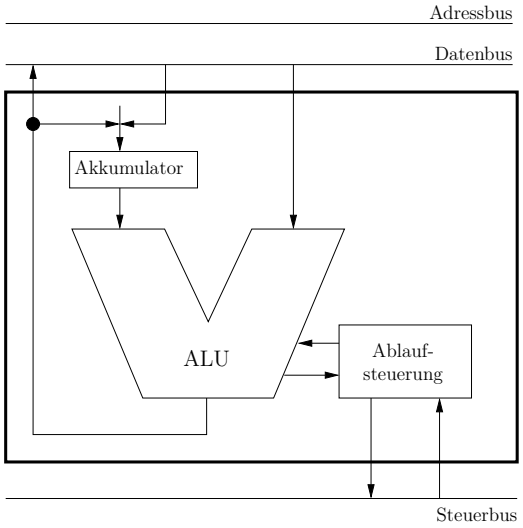
Das Statusregister (*status register, SR*) nimmt Rückmeldungen des Systems auf.

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

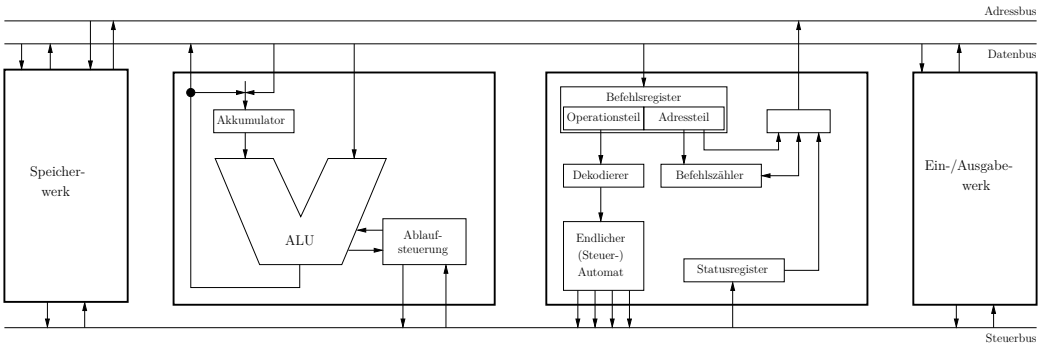


Das Rechenwerk bildet zusammen mit dem Steuerwerk den Hauptprozessor (CPU).

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 aufeinander folgende Befehle in aufeinander folgenden 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.

Genau erläutert wird diese sogenannten *list comprehensions* später.

```
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

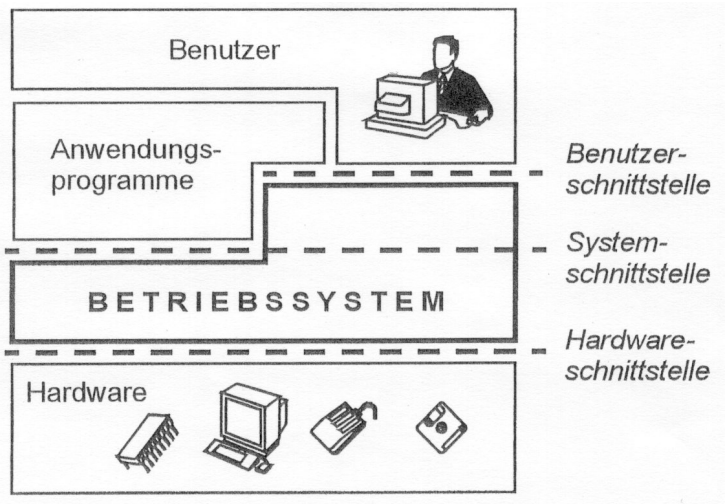


Abbildung 1: Position des Betriebssystems (Quelle: Carsten Vogt, Betriebssysteme, Spektrum Akademischer Verlag, 2001)

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).

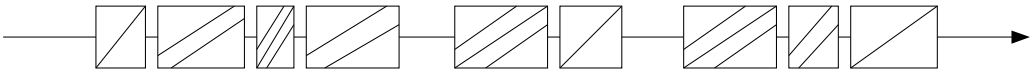
Einzelbenutzerbetrieb

> **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.

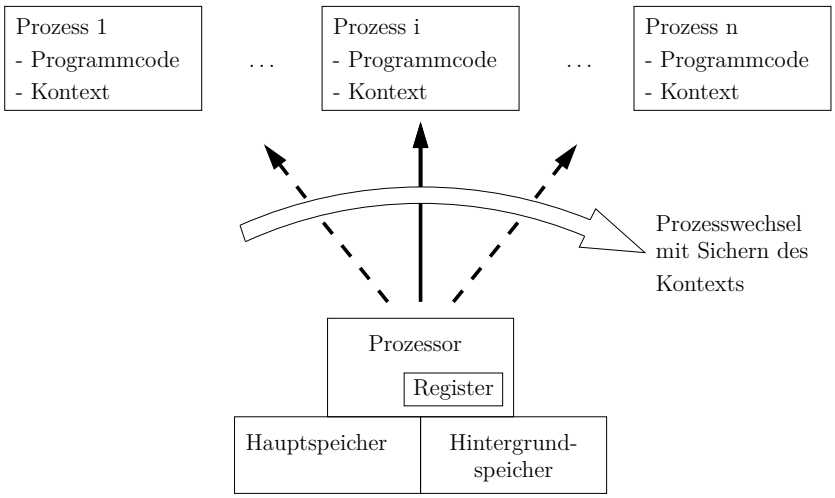
Eine Aufgabe des Betriebssystems besteht darin, den verschiedenen Prozessen Prozessorzeit zuzuteilen (engl. ***scheduling)***.

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



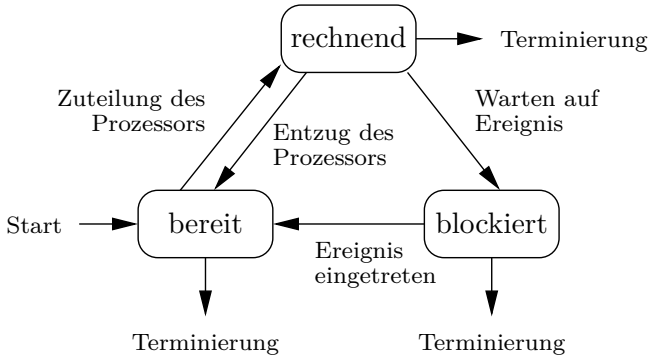
Ein Prozess befindet sich zu jedem Zeitpunkt in einem bestimmten **Zustand**. Es finden **dynamisch Zustandsübergänge**, also Veränderungen des Prozesszustands statt.

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.

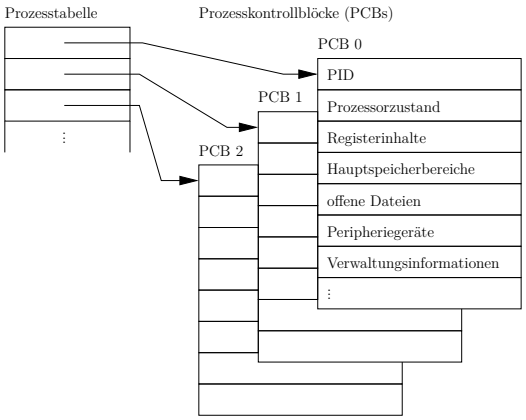
Zustandsübergänge



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.



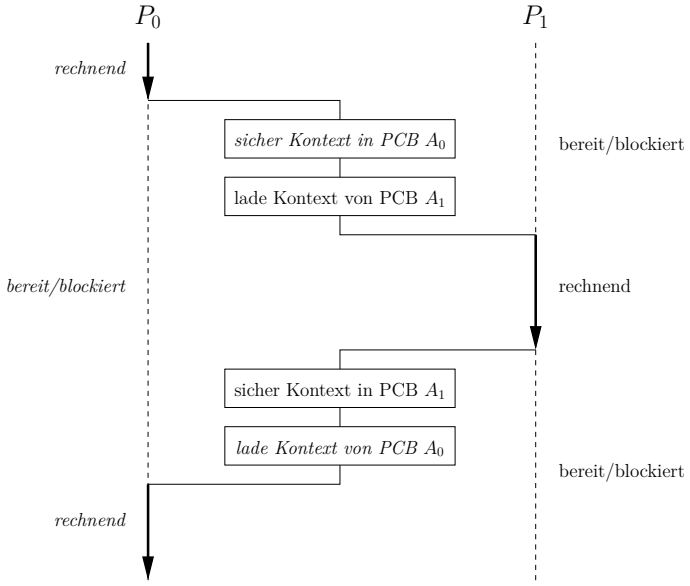
Prozesswechsel (Dispatching)

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.

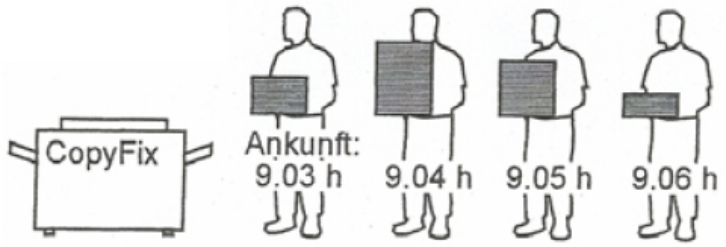


Abbildung 2: First Come First Served (Quelle: Carsten Vogt, Betriebssysteme, Spektrum Akademischer Verlag, 2001)

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?

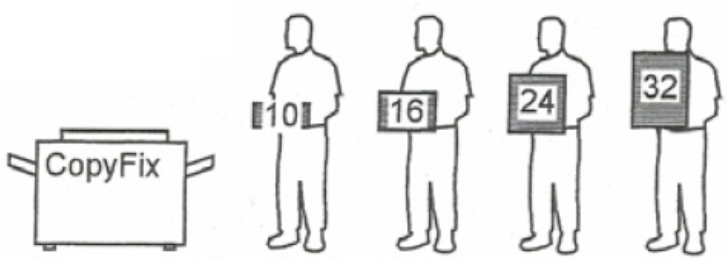


Abbildung 3: Shortest Job First (Quelle: Carsten Vogt, Betriebssysteme, Spektrum Akademischer Verlag, 2001)

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.

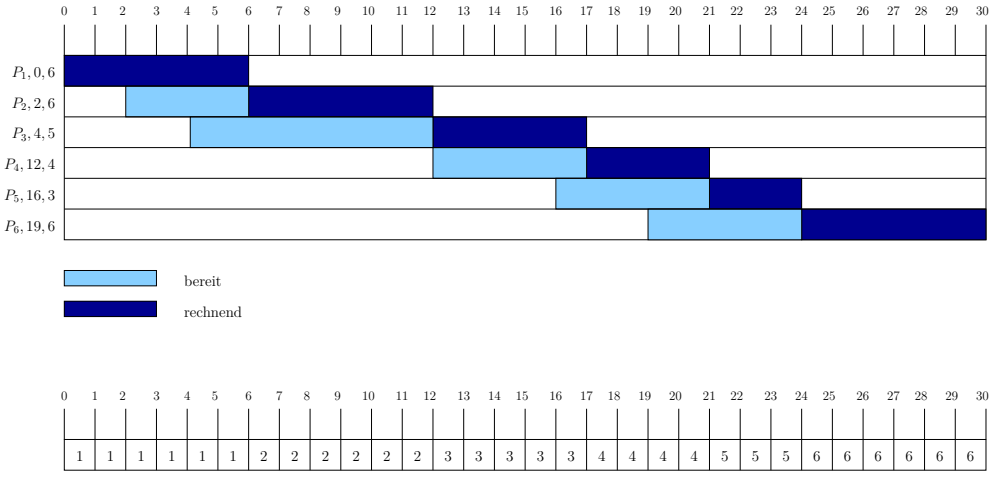
Beispiel

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

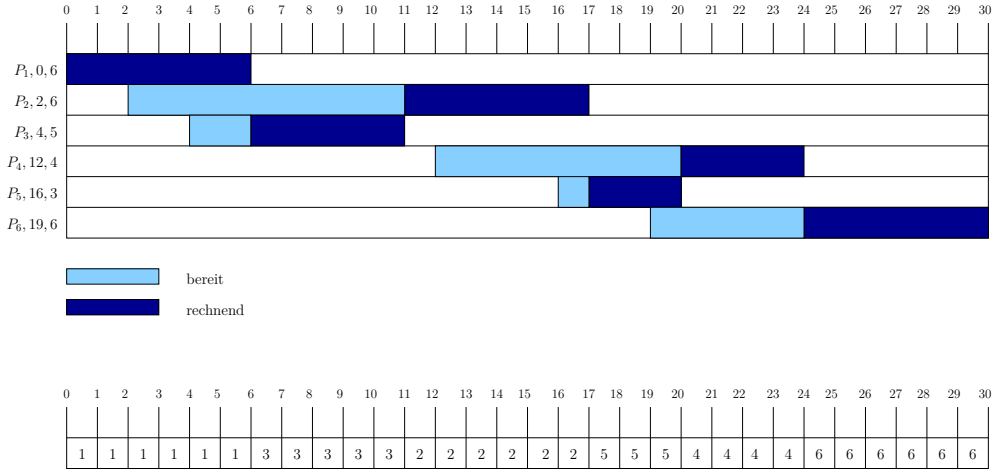
Prozesse	P_1	P_2	P_3	P_4	P_5	P_6
Ankunftszeit a_i	0	2	4	12	16	19
Rechenzeit t_i	6	6	5	4	3	6

Beispiel, FCFS

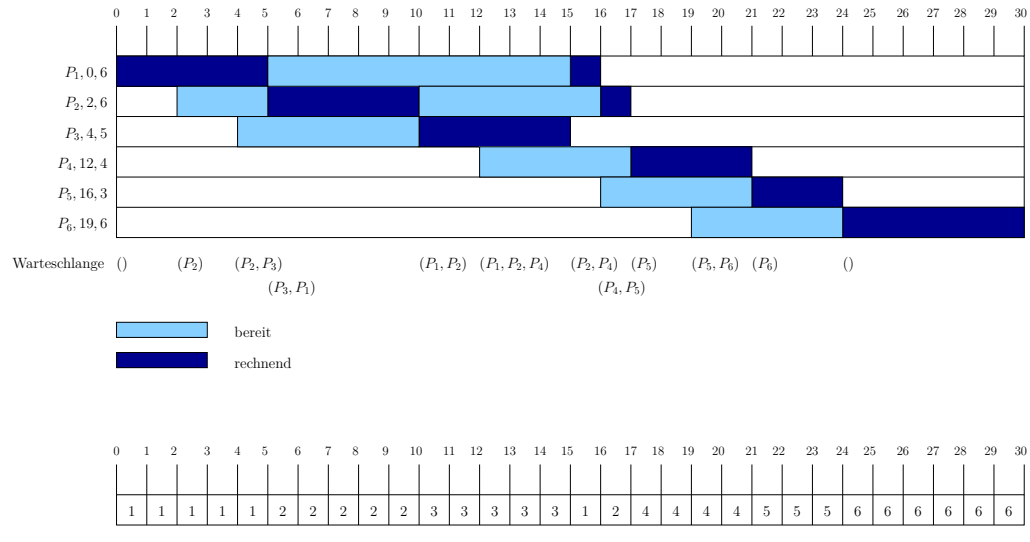
Darstellung des Schedules als *Gantt Chart* (nach Henry L. Gantt 1861-1919) oder *Balkenplan*.



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**.

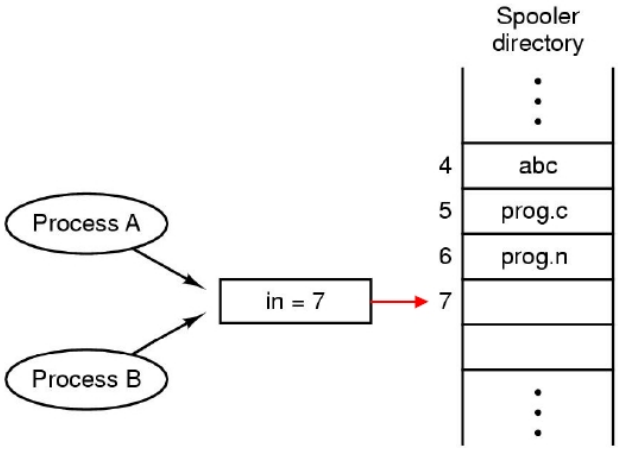
- **Geräte.** Drucker, Platten, usw.
- **Daten.** Dateien, Shared Memory, usw.

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

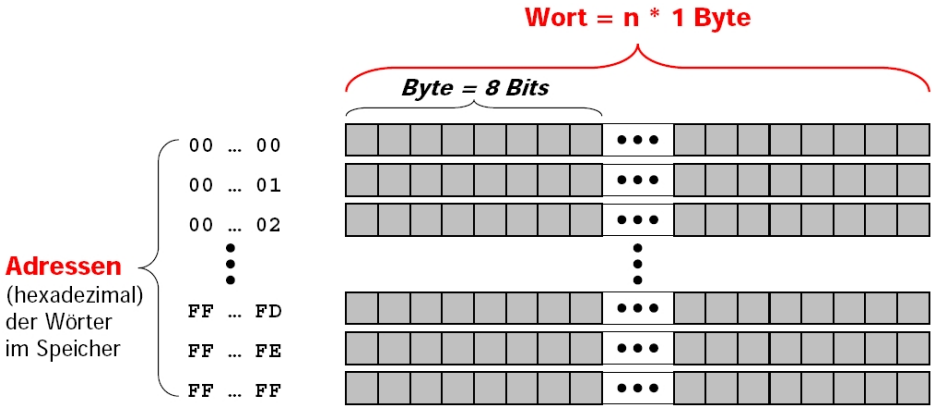
```
1 // main process
2 global mutex m;
3 start_concurrent(A, B);
```

```
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.

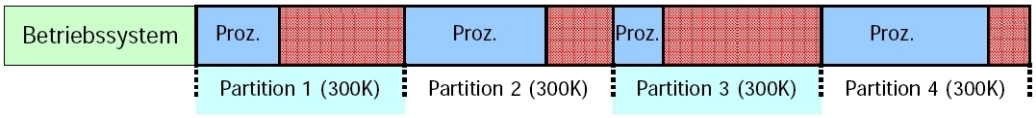
Ein Adressraum ist eine Menge von Adressen, die ein Prozess zur Adressierung des Speichers nutzen kann.

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

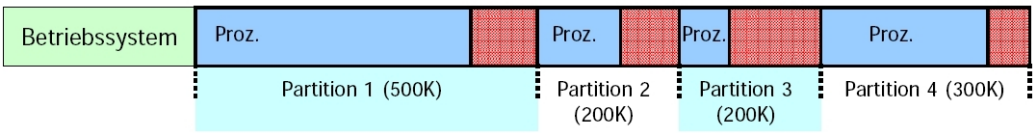
Aus Sicht des Prozesses ist der Adressraum der ganze zur Verfügung stehende Speicher, der in der Regel mit der Adresse 0 beginnt. Die Endadresse ist abhängig von der Größe des zugeteilten Speicherbereichs.

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.

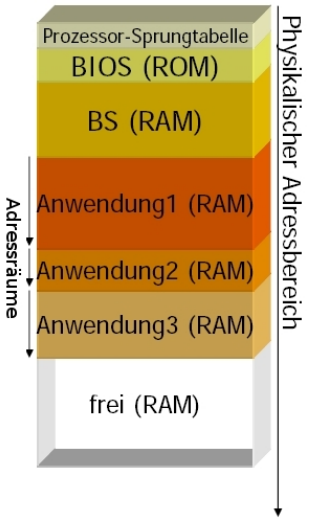
Beispiel UNIX mit dynamischer Relokation

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.

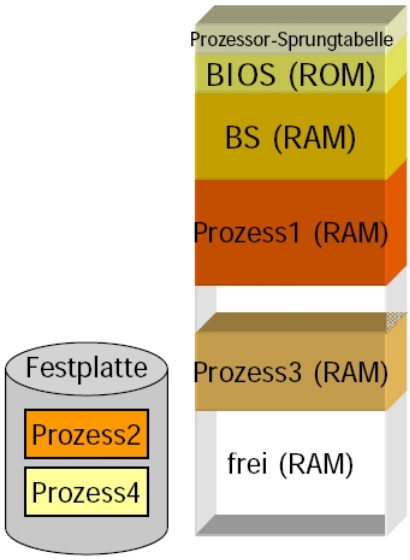
Es laufen soviele Prozesse, dass der physikalische Speicher nicht groß genug ist um alle gleichzeitig aufzunehmen.

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.

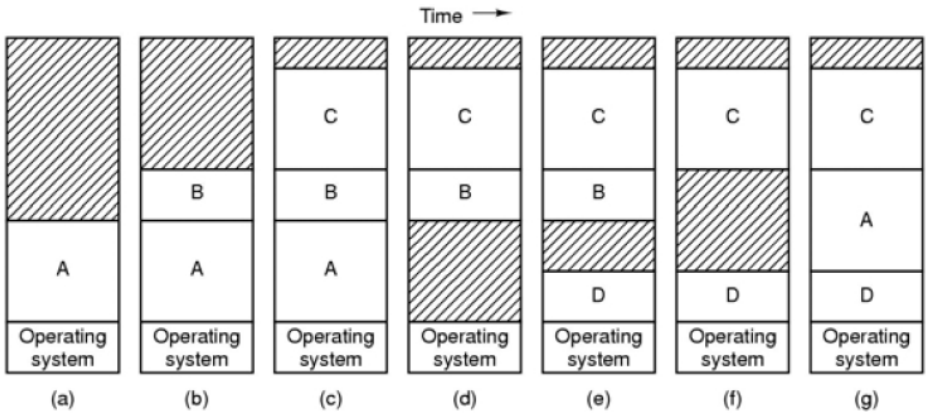
- Auslagern von (z.B. blockierten) Prozessen auf die Festplatte.
- Einlagern von (z.B. bereiten) Prozesses in den Speicher.

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).

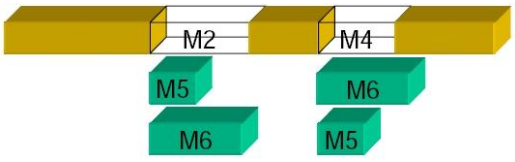


Beim Einlagern kann sich der physikalische Adressraum (die Partition) des Prozesses ändern. Das heißt bei dynamischer Relokation müssen beim Einlagern eines Prozesses u.U. die Werte im Basis- und Limitregister angepasst werden.

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.

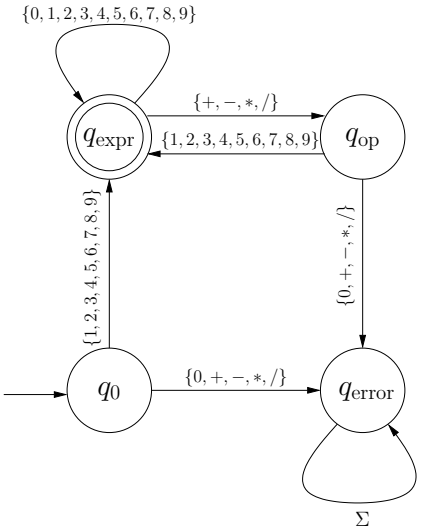
Ein Zustandsgraphen ist ein gerichteter Graph.

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.

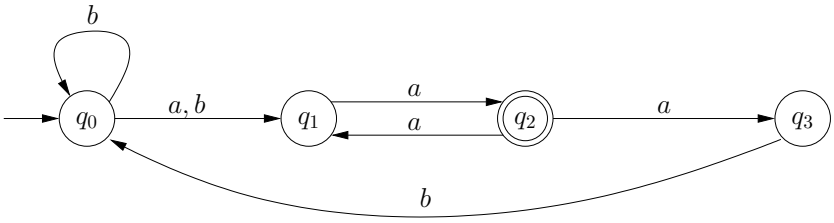
Der Automat befinde sich, nach dem Abarbeiten der Eingabe bis zum Zeichen w_i (inclusive), gleichzeitig in den Zuständen $U \subseteq Q$.

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.

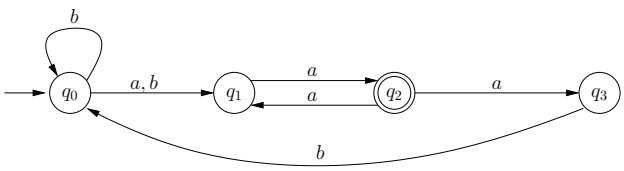
  Bemerkung. Es gibt nur noch markierte Knoten. Deshalb hat jeder Knoten für jedes $a \in \Sigma$ genau eine ausgehende Kante.
- Der Startzustand des DEA ist der Knoten, der den Startzustand des NEA repräsentiert.

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

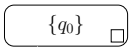
Beispiel

Schritt 1

NEA

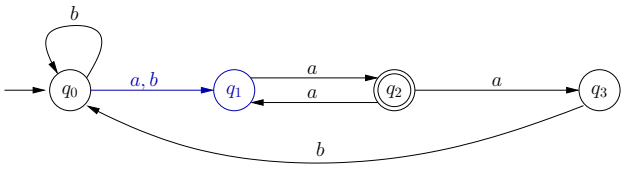


Graph

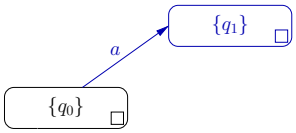


Schritt 2 (Auswahl)

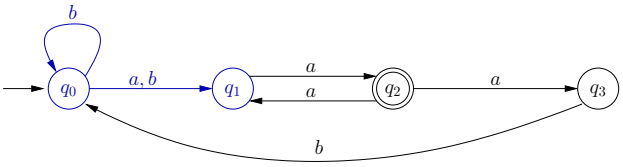
NEA



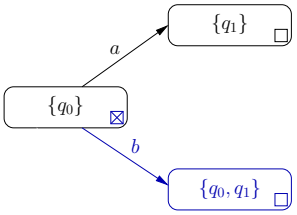
Graph



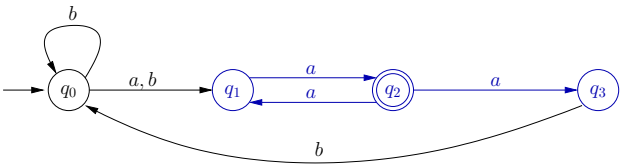
NEA



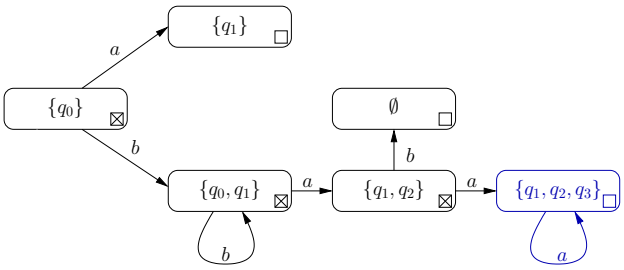
Graph



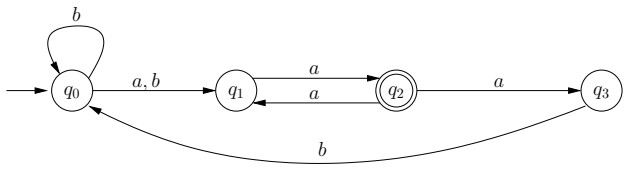
NEA



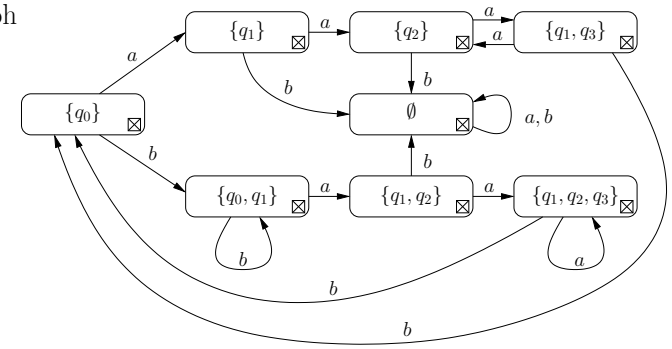
Graph



NEA

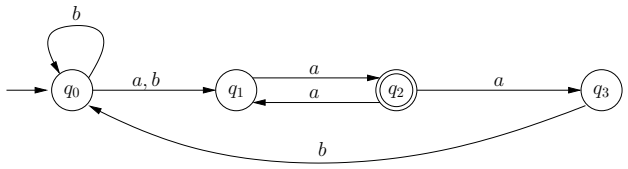


Graph

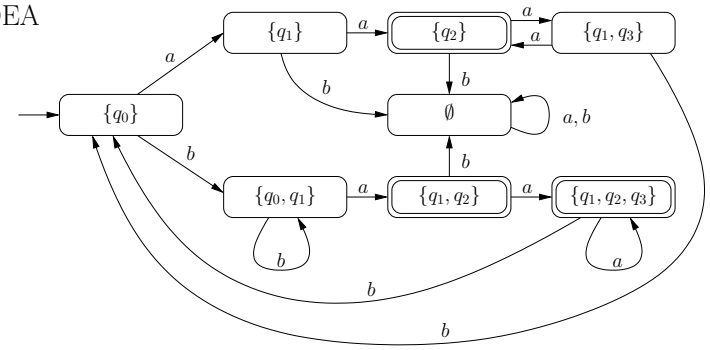


Schritt 3

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

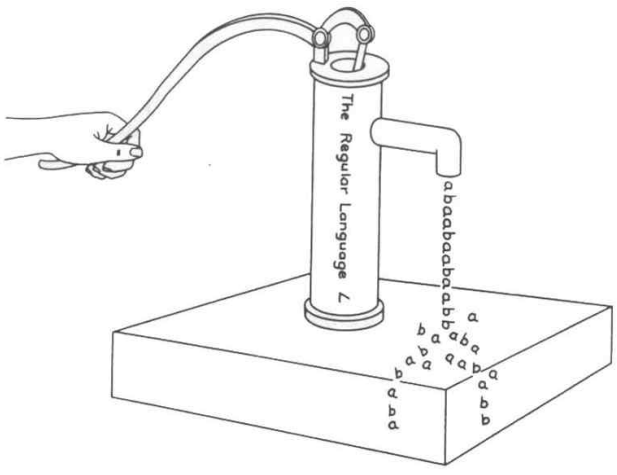
- $\emptyset$,
- $\{\varepsilon\}$,
- $\{a\}$ für jedes $a \in \Sigma$.

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.

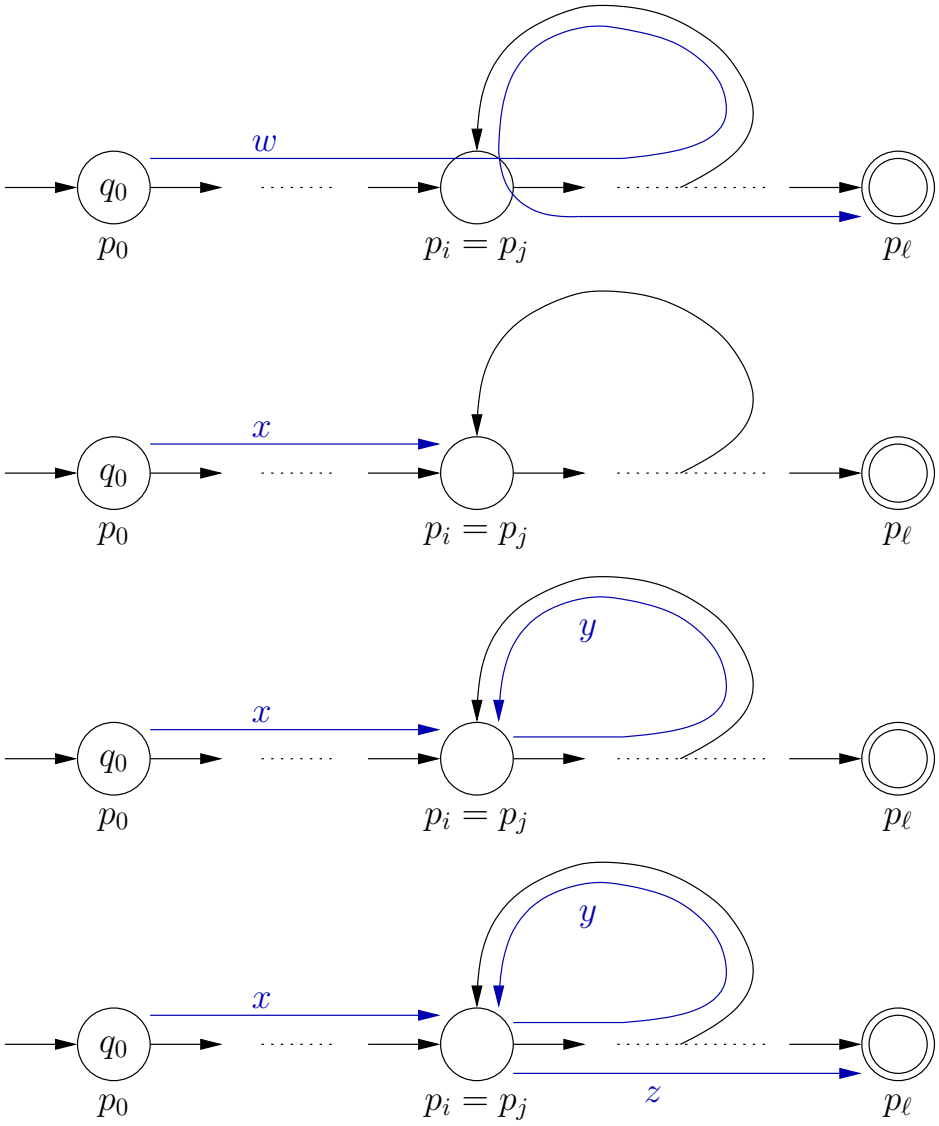
Wähle die Pumping-Konstante $k = |Q|$, wobei $|Q|$ die Anzahl der Zustände von $\mathcal{A}$ ist.

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 $xy^iz \in L$ und mit $\hat{p} := p_i = p_j$ folgt

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

  das heißt $p_0 xy^i z = p_\ell$.

□

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 $xy^iz \notin L$.

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

Aufpumpen von y. Sei $i \geq 2$, dann ist $xy^iz = a^{k-s}a^{si}b^k \notin L$, 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 $F := \{E\}$ 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.

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

Die 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.

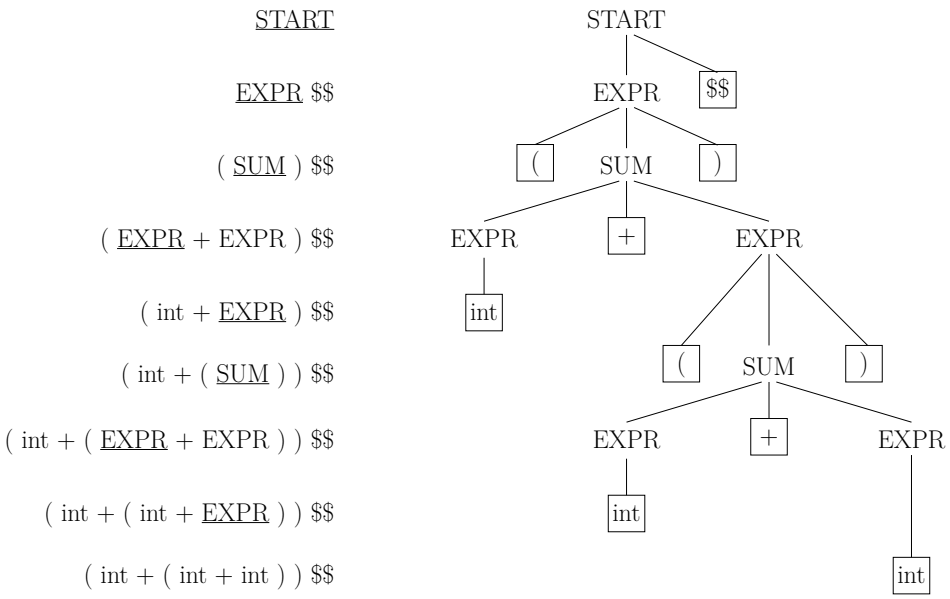
Die **Blätter** eines Ableitungsbaumes entsprechen den Terminalsymbolen der Grammatik.

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

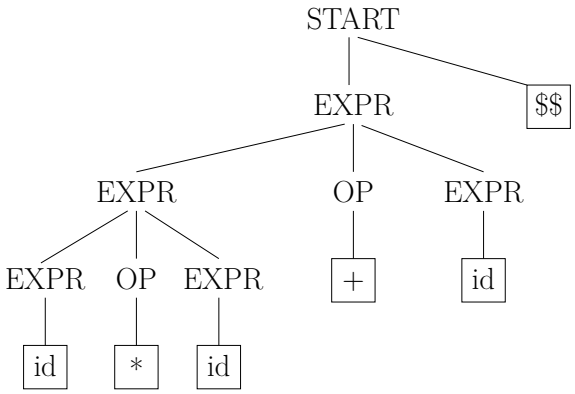
Beispiel 6.32.

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

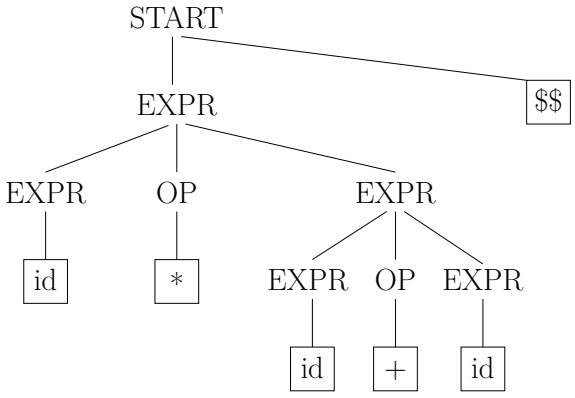
  START → EXPR $$
  EXPR → id | EXPR OP EXPR
  OP → + | - | * | /

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;
```

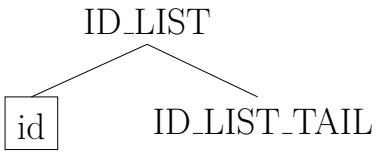
Satz (Tokenfolge)
`id,id,id;`$$

Ableitungsbaum

ID_LIST

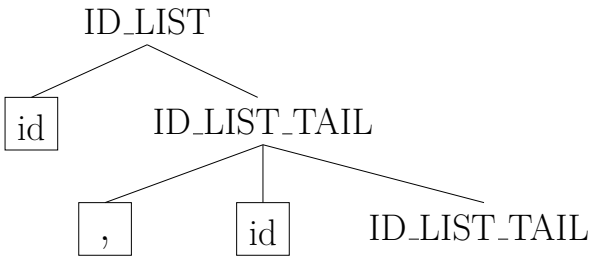
Satz (Tokenfolge)
`id`,id,id;$$

Ableitungsbaum



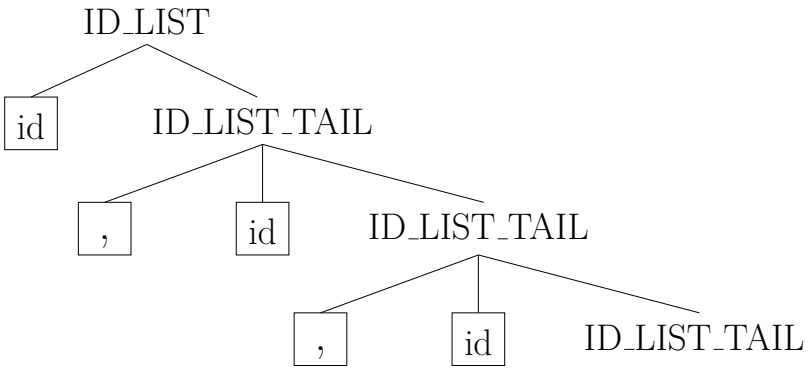
Satz (Tokenfolge)
~~id~~ `,` id,id;$$

Ableitungsbaum



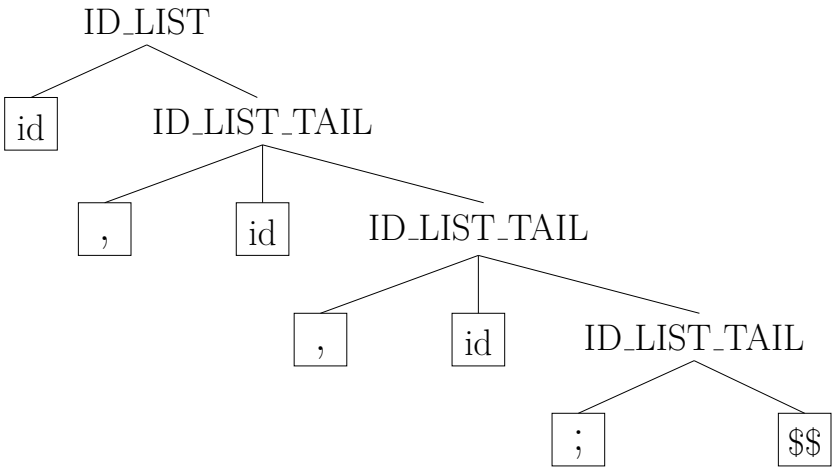
Satz (Tokenfolge)
~~id,id~~ `,` id;$$

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

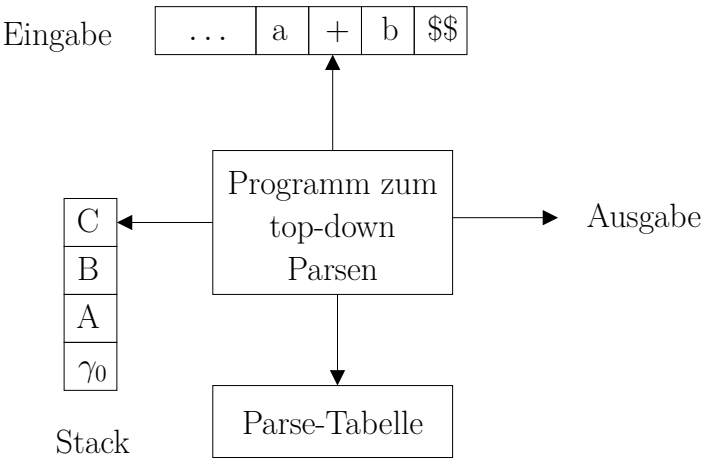
Pearson Studium.

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	$F \to \mathbf{id}$			$F \to (\,E\,)$		