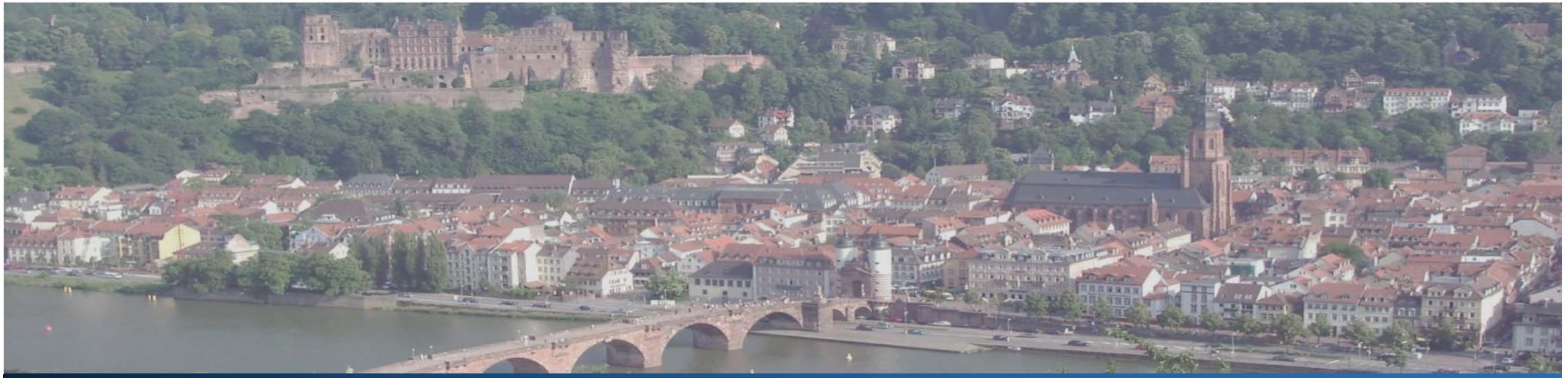




Einführung in die Praktische Informatik

Prof. Björn Ommer HCI, IWR
Computer Vision Group

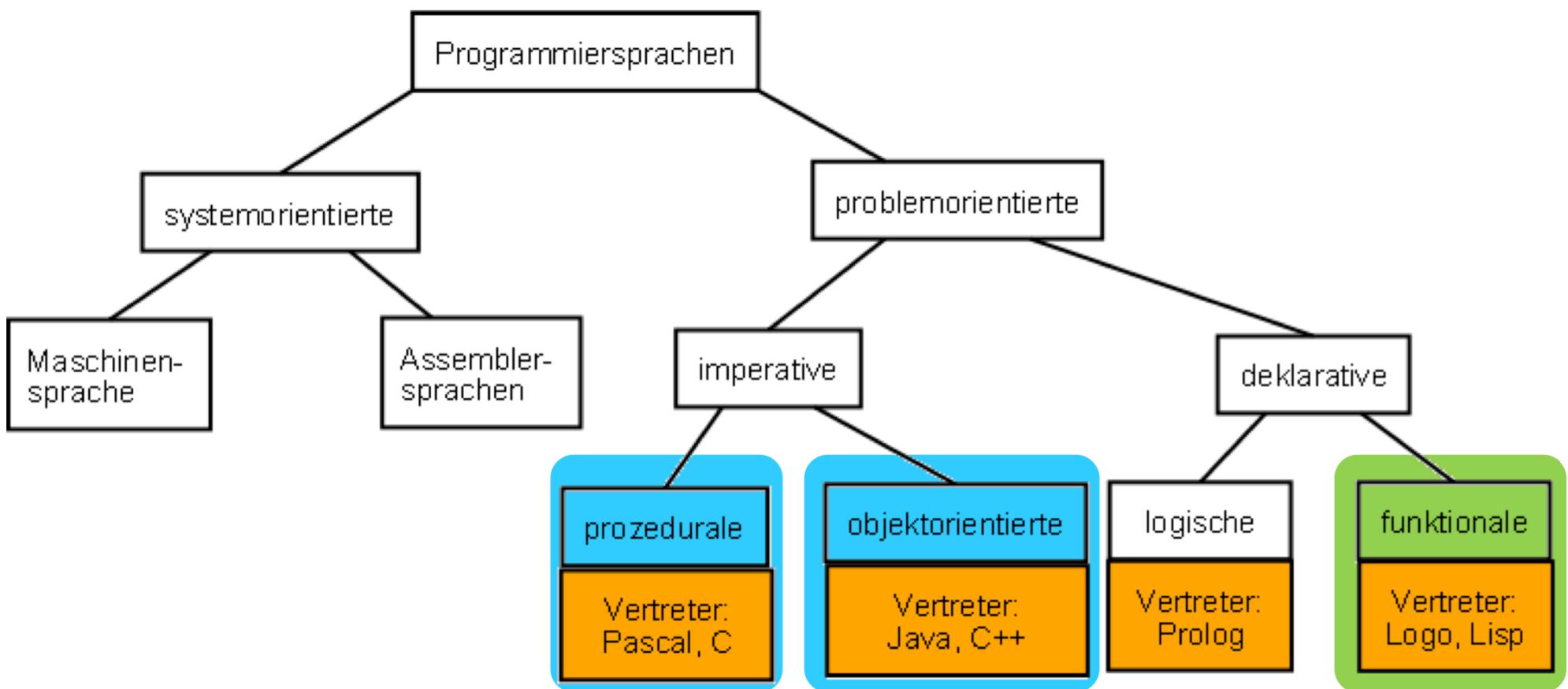


Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Programmiersprachen und -paradigmen



Zuerst etwas Notation

- Das erste C++ Programm

```
1 #include <iostream>
2
3
4 int main ()
5 {
6     // Textausgabe:
7     std::cout << "Hallo Welt!" << std::endl;
8
9     // Programm beenden
10    return 0;
11 }
```

- ▶ **#include <iostream>** lädt Ein-/Ausgabe Bibliothek
- ▶ Eigentliche Funktionalität in main()
- ▶ Kommentare beginnen mit //
- ▶ Blöcke von Kommentaren über mehrere Zeilen mit
 / * ...Kommentar – Block... * /
- ▶ Ausgaben über std::cout
 std::cout kann mit << alles "zugeworfen" werden,
 was ausgegeben werden soll
- ▶ Programm endet mit Rückgabe an Betriebssystem
return 0;

Übersetzen und Binden

- ▶ Quelltext ist von Menschen lesbare Form
- ▶ Computer will ausführbare Form
 - Übersetzen erzeugt Objektdatei
g++ -c hello.cpp
 - Binden erzeugt ausführbare Datei
g++ -o hello hello.o
- ▶ Aufrufen des Programms
./hello
Hello Welt!
- ▶ Änderungen immer im Quelltext, Erneutes übersetzen und Binden nötig.

Funktionale Programmierung

- In diesem Abschnitt beschränken wir uns bewusst auf eine sehr kleine Teilmenge von C++.

Arithmetische Ausdrücke

Beispiel: Auswertung von:

$$5 + 3 \text{ oder } ((3 + (5 * 8)) - (16 * (7 + 9))).$$

Programm: (Erste Schritte [erstes.cc])

```
#include "fcpp.hh"

int main()
{
    return print( (3+(5*8)) - (16*(7+9)) );
}
```

Übersetzen (in Unix-Shell):

```
> g++ -o erstes erstes.cc
```

Ausführung:

```
> ./erstes  
-213
```

Bemerkung:

- Ohne „-o erstes“ wäre der Name „a.out“ verwendet worden.
- Das Programm berechnet den Wert des Ausdrucks und druckt ihn auf der Konsole aus.

Wie wertet der Rechner so einen Ausdruck aus?

Die Auswertung eines zusammengesetzten Ausdruckes lässt sich auf die Auswertung der vier elementaren Rechenoperationen $+$, $-$, $*$ und $/$ zurückführen.

Dazu fassen wir die Grundoperationen als **zweistellige Funktionen** auf:

$$+, -, *, / : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}.$$

Jeden Ausdruck können wir dann äquivalent umformen:

$$((3 + (5 * 8)) - (16 * (7 + 9))) \equiv -(+(3, *(5, 8)), *(16, +(7, 9))).$$

Definition: Die linke Schreibweise nennt man **Infix-Schreibweise** (*infix notation*), die rechte **Präfix-Schreibweise** (*prefix notation*).

Bemerkung: Die Infix-Schreibweise ist für arithmetische Ausdrücke bei Hinzunahme von Präzedenzregeln wie „Punkt vor Strich“ und dem Ausnutzen des Assoziativgesetzes kürzer (da Klammern weglassen werden können) und leichter lesbar als die Präfix-Schreibweise.

Bemerkung: Es gibt auch eine **Postfix-Schreibweise**, welche zum Beispiel in HP-Taschenrechnern, dem Emacs-Programm „Calc“ oder der Computersprache Forth verwendet wird.

Die vier Grundoperationen $+, -, *, /$ betrachten wir als **atomar**. Im Rechner gibt es entsprechende Baugruppen, die diese atomaren Operationen realisieren.

Der Compiler übersetzt den Ausdruck aus der Infix-Schreibweise in die äquivalente Präfixschreibweise. Die Auswertung des Ausdrucks, d. h. die Berechnung der Funktionen, erfolgt dann **von innen nach aussen**:

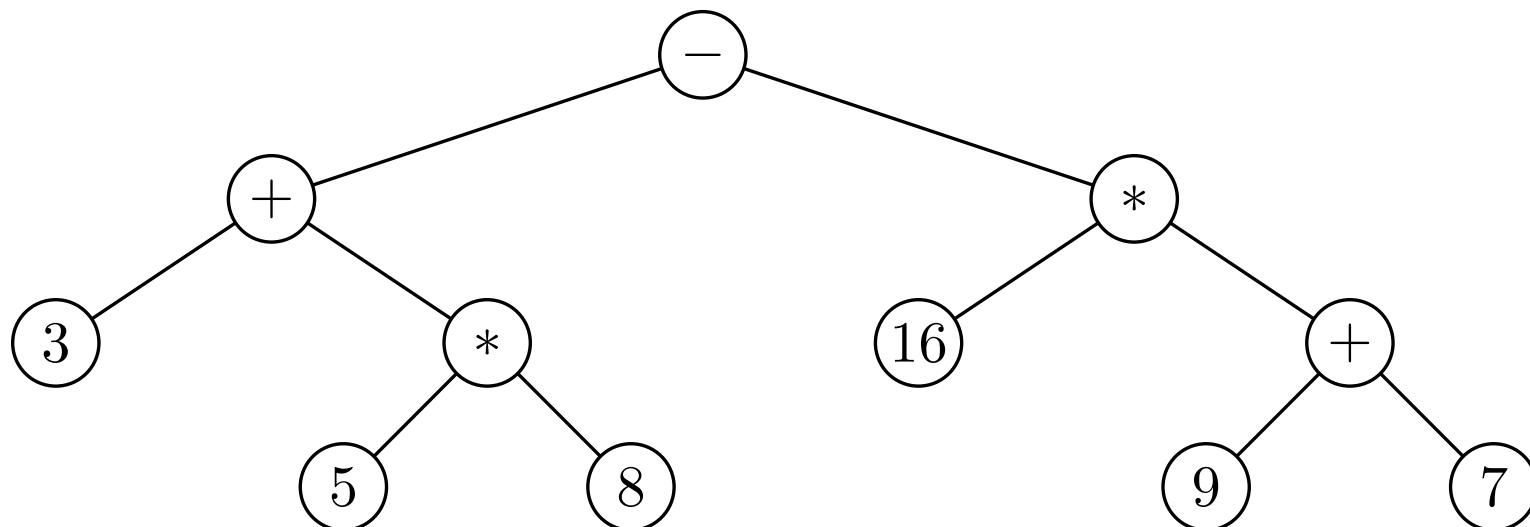
$$\begin{aligned}& -(+ (3, * (5, 8)), * (16, +(7, 9))) \\&= -(+ (3, \quad 40 \quad), * (16, +(7, 9))) \\&= -(\quad 43 \quad , * (16, +(7, 9))) \\&= -(\quad 43 \quad , * (16, \quad 16 \quad)) \\&= -(\quad 43 \quad , \quad 256 \quad) \\&= \quad \quad \quad -213\end{aligned}$$

Bemerkung: Dies ist nicht die einzige mögliche Reihenfolge der Auswertung der Teiloperationen, alle Reihenfolgen führen jedoch zum gleichen Ergebnis! (Zumindest bei nicht zu großen, ganzen Zahlen).

Bemerkung: C++ kennt die Punkt-vor-Strich-Regel und das Assoziativgesetz. Überflüssige Klammern können also weggelassen werden.

Ausdrücke als Bäume

Jeder arithmetische Ausdruck kann als binärer Baum dargestellt werden. Die Auswertung des Ausdruckes erfolgt dann **von den Blättern zur Wurzel**. In dieser Darstellung erkennt man welche Ausführungsreihenfolgen möglich sind bzw. welche Teilausdrücke gleichzeitig ausgewertet werden können (**Datenflussgraph**).



Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Funktionen

Zu den schon eingebauten Funktionen wie `+, -, *, /` kann man noch weitere **benutzerdefinierte** Funktionen hinzuzufügen.

Beispiel: Eine einstellige Funktion:

```
int quadrat( int x )
{
    return x*x;
}
```

Die erste Zeile (**Funktionskopf**) vereinbart, dass die neue Funktion namens `quadrat` als Argument eine Zahl mit Namen `x` vom Typ `int` als Eingabe bekommt und einen Wert vom Typ `int` als Ergebnis liefert.

Der **Funktionsrumpf** (*body*) zwischen geschweiften Klammern sagt, was die Funktion tut. Der Ausdruck nach `return` ist der Rückgabewert.

Eigene Funktionen

```
1 type_ergebnis funktionsname(typ_arg1 name_arg1, typ_arg2, name_arg2)
2 {
3     <code>
4
5     return ergebnis;
6 } // Funktionskörper, Definition, Implementation
```

Wir werden uns zunächst auf einen sehr kleinen Teil des Sprachumfangs von C/C++ beschränken. Dort besteht der Funktionsrumpf nur aus dem Wort `return` gefolgt von einem **Ausdruck** gefolgt von einem **Semikolon**.

Bemerkung: C++ ist eine **streng typgebundene** Programmiersprache (*strongly typed*), d. h. jedem **Bezeichner** (z. B. `x` oder `quadrat`) ist ein **Typ** zugeordnet. Diese Typzuordnung kann nicht geändert werden (**statische Typbindung**, *static typing*).

Dies ist in völliger Analogie zur Mathematik:

$$x \in \mathbb{Z}, \quad f : \mathbb{N} \rightarrow \mathbb{N}.$$

Bemerkung: Der Typ `int` entspricht dabei (kleinen) ganzen Zahlen. Andere Typen sind `float`, `double`, `char`, `bool`. Später werden wir sehen, dass man auch neue Typen hinzufügen kann.

Programm: (Verwendung [quadrat.cc])

```
#include "fcpp.hh"

int quadrat( int x )
{
    return x*x;
}

int main()
{
    return print( quadrat( 3 ) + quadrat( 4+4 ) );
}
```

Bemerkung: Damit können wir die Bedeutung aller Elemente des Programmes verstehen.

- Neue Funktionen kann man (in C) nur in Präfix-Schreibweise verwenden.
- `main` ist eine Funktion ohne Argumente und mit Rückgabetyp `int`.
- `#include "fcpp.hh"` ist ein sogenannter **Include-Befehl**. Er sorgt dafür, dass die in der Datei `fcpp.hh` enthaltenen Erweiterungen von C++, etwa zusätzliche Funktionen, verwendet werden können. `fcpp.hh` ist nicht Teil des C++ Systems, sondern wird von uns für die Vorlesung zur Verfügung gestellt (erhältlich auf der Webseite). Achtung: Die Datei muss sich im selben Verzeichnis befinden wie das zu übersetzende Programm damit der Compiler diese finden kann.
- `print` ist eine Funktion mit Rückgabewert 0 (unabhängig vom Argument), welche den Wert des Arguments auf der Konsole ausdrückt (**Seiteneffekt**). Die Definition dieser Funktion ist in der Datei `fcpp.hh` enthalten.
- Die Programmausführung beginnt immer mit der Funktion `main` (sozusagen das **Startsymbol**).

Datentypen und Variablen

- in C++ wird jeder Speicherzelle ein Typ zugewiesen (legt Größe und Bedeutung der Speicherzelle fest)
- wichtige Typen: “int” für ganze Zahlen, “double” für reelle Zahlen, “std::string” für Text
- zugehörige Literale: bei int 12, -3; double -1.02, $1.2e^{-4} = 1.2 \cdot 10^{-4}$; std::string "text"

```
double a=...;
double b=  ;
double c=  ;
double p=-0,5*b/a;
double q=c/a;
double discr=std::sqrt(p*p-q);
double x1 = p+discr;
double x2 = p-discr;
std::cout << "x1:" << x1 << "x2:" << x2 << endl;
//oder
std::cout << "x1:" << x1 << "x2:" << x2 << "\n";
```

Overloading

```
int sq(int x)          double sq(double x)
{
    return x*x;        {
    return x*x;
}
```

beide Varianten dürfen in C++ gleichzeitig definiert sein
“overloading” ⇒ C++ wählt automatisch die richtige
Variable anhand des Argumenttyps (overload resolution)

```
int x=2;
double y=1.1;
int x2=sq(x); //int-Variante
double y2=sq(y); //double-Variante
```

Selektion

Fehlt noch: Steuerung des Programmverlaufs in Abhängigkeit von Daten.

Beispiel: Betragsfunktion

$$|x| = \begin{cases} -x & x < 0 \\ x & x \geq 0 \end{cases}$$

Um dies ausdrücken zu können, führen wir eine spezielle **dreistellige** Funktion cond ein:

Programm: (Absolutwert [absolut.cc])

```
#include "fcpp.hh"

int absolut( int x )
{
    return cond( x<=0, -x, x );
}

int main()
{
    return print( absolut( -3 ) );
}
```

Der Operator cond erhält drei Argumente: Einen **Boolschen Ausdruck** und zwei normale Ausdrücke. Ein Boolescher Ausdruck hat einen der beiden Werte „wahr“ oder „falsch“ als Ergebnis. Ist der Wert „wahr“, so ist das Resultat des cond-Operators der Wert des zweiten Arguments, ansonsten der des dritten.

Bemerkung: cond kann keine einfache **Funktion** sein:

- cond kann auf verschiedene Typen angewendet werden, und auch der Typ des Rückgabewerts steht nicht fest.
- Oft wird cond nicht alle Argumente auswerten dürfen, um nicht in Fehler oder Endlosschleifen zu geraten.

Bemerkung: Damit haben wir bereits eine Menge von Konstrukten kennengelernt, die turing-äquivalent ist!

Programm: (Elementare funktionale Programmierung [alles_funktional.cc])

```
#include "fcpp.hh"

int quadrat( int x ) { return x*x; }

int absolut( int x ) { return cond( x<=0, -x, x ); }

int main()
{
    return print( absolut(-4) * (7*quadrat(3)+8) );
}
```

Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Syntaxbeschreibung in (E)BNF

Die Regeln nach denen wohlgeformte Sätze einer Sprache erzeugt werden, nennt man **Syntax**.

Die Syntax von Programmiersprachen ist recht einfach. Zur Definition verwendet man eine spezielle Schreibweise, die erweiterte Backus⁹-Naur¹⁰ Form (EBNF):

Man unterscheidet in der EBNF folgende Zeichen bzw. Zeichenketten:

⁹ John Backus, 1924–2007, US-amerik. Informatiker.

¹⁰ Peter Naur, geb. 1928, dänischer Informatiker.

EBNF

- Unterstrichene Zeichen oder Zeichenketten sind Teil der zu bildenden, wohlgeformten Zeichenkette. Sie werden nicht mehr durch andere Zeichen ersetzt, deshalb nennt man sie **terminale Zeichen**.
- Zeichenketten in spitzen Klammern, wie etwa $\langle Z \rangle$ oder $\langle \text{Ausdruck} \rangle$ oder $\langle \text{Zahl} \rangle$, sind Symbole für noch zu bildende Zeichenketten. Regeln beschreiben, wie diese Symbole durch weitere Symbole und/oder terminale Zeichen ersetzt werden können. Da diese Symbole immer ersetzt werden, nennt man sie **nichtterminale Symbole**.
- $\langle \epsilon \rangle$ bezeichnet das „leere Zeichen“.
- Die normal gesetzten Zeichen(ketten)

$$\text{ ::= } \quad | \quad \{ \quad \} \quad \}^+ \quad [\quad]$$

sind Teil der Regelbeschreibung und tauchen nie in abgeleiteten Zeichenketten auf. (Es sei denn sie sind unterstrichen und somit terminale Zeichen).

EBNF

- (Alternativ findet man auch die Konvention terminale Symbole in Anführungszeichen zu setzen und die spitzen Klammern bei nichtterminalen wegzulassen).

Jede Regel hat ein Symbol auf der linken Seite gefolgt von „::=“. Die rechte Seite beschreibt, durch was das Symbol der linken Seite ersetzt werden kann.

Beispiel:

$$\begin{aligned}\langle A \rangle & ::= \underline{a} \langle A \rangle \underline{b} \\ \langle A \rangle & ::= \langle \epsilon \rangle\end{aligned}$$

Ausgehend vom Symbol $\langle A \rangle$ kann man somit folgende Zeichenketten erzeugen:

$$\langle A \rangle \rightarrow \underline{a} \langle A \rangle \underline{b} \rightarrow \underline{\underline{a}} \langle A \rangle \underline{\underline{b}} \rightarrow \dots \rightarrow \underbrace{\underline{a} \dots \underline{a}}_{n \text{ mal}} \langle A \rangle \underbrace{\underline{b} \dots \underline{b}}_{n \text{ mal}} \rightarrow \underbrace{\underline{a} \dots \underline{a}}_{n \text{ mal}} \underbrace{\underline{ab} \dots \underline{ab}}_{n \text{ mal}}$$

Bemerkung: Offensichtlich kann es für ein Symbol mehrere Ersetzungsregeln geben. Wie im MIU-System ergeben sich die wohlgeformten Zeichenketten durch alle möglichen Regelanwendungen.

Kurzschreibweisen

Oder:

Das Zeichen „ | “ („oder“) erlaubt die Zusammenfassung mehrerer Regeln in einer Zeile. Beispiel: $\langle A \rangle ::= \underline{a} \langle A \rangle \underline{b} \mid \langle \epsilon \rangle$

Option:

$\langle A \rangle ::= [\langle B \rangle]$ ist identisch zu $\langle A \rangle ::= \langle B \rangle \mid \langle \epsilon \rangle$

Wiederholung mit $n \geq 0$:

$\langle A \rangle ::= \{ \langle B \rangle \}$ ist identisch mit $\langle A \rangle ::= \langle A \rangle \langle B \rangle \mid \langle \epsilon \rangle$

Wiederholung mit $n \geq 1$:

$\langle A \rangle ::= \{ \langle B \rangle \}^+$ ist identisch zu
 $\langle A \rangle ::= \langle A \rangle \langle B \rangle \mid \langle B \rangle$

Beispiel: Die EBNF wird auch in der UNIX Dokumentation verwendet:

EBNF zur Beschreibung der Syntax von UNIX Befehlen

BSDTAR(1) BSD General Commands Manual

BSDTAR(1)

NAME

tar -- manipulate tape archives

SYNOPSIS

```
tar [bundled-flags <args>] [<file> | <pattern> ...]
tar {-c} [options] [files | directories]
tar {-r | -u} -f archive-file [options] [files | directories]
tar {-t | -x} [options] [patterns]
```

Syntaxbeschreibung für FC++

Die bisher behandelte Teilmenge von C++ nennen wir FC++ („funktionales C++“) und wollen die Syntax in EBNF beschreiben.

Syntax: (Zahl)

$$\langle \text{Zahl} \rangle ::= [\pm | \div] \{ \langle \text{Ziffer} \rangle \}^+$$

Syntax: (Ausdruck)

$$\begin{aligned}\langle \text{Ausdruck} \rangle &::= \langle \text{Zahl} \rangle \mid [\div] \langle \text{Bezeichner} \rangle \mid \\ &\quad (\langle \text{Ausdruck} \rangle \langle \text{Operator} \rangle \langle \text{Ausdruck} \rangle) \mid \\ &\quad \langle \text{Bezeichner} \rangle [\langle \text{Ausdruck} \rangle \{ , \langle \text{Ausdruck} \rangle \}] \mid \\ &\quad \langle \text{Cond} \rangle \\ \langle \text{Bezeichner} \rangle &::= \langle \text{Buchstabe} \rangle \{ \langle \text{Buchstabe oder Zahl} \rangle \} \\ \langle \text{Operator} \rangle &::= \pm | \div | * | /\end{aligned}$$

Wegelassen: Regeln für $\langle \text{Buchstabe} \rangle$ und $\langle \text{Buchstabe oder Zahl} \rangle$.

Diese einfache Definition für Ausdrücke enthält weder Punkt-vor-Strich noch das Weglassen von Klammern aufgrund des Assoziativgesetzes!

Hier die Syntax einer Funktionsdefinition in EBNF:

Syntax: (Funktionsdefinition)

```
<Funktion>      ::= <Typ> <Name> ( <formale Parameter> )
                      { <Funktionsrumpf> }
<Typ>            ::= <Bezeichner>
<Name>           ::= <Bezeichner>
<formale Parameter> ::= [ <Typ> <Name> { , <Typ> <Name> } ]
```

Die Argumente einer Funktion in der Funktionsdefinition heissen **formale Parameter**. Sie bestehen aus einer kommasseparierten Liste von Paaren aus Typ und Name. Damit kann man also n -stellige Funktionen mit $n \geq 0$ erzeugen.

Regel für den Funktionsrumpf:

```
<Funktionsrumpf> ::= return <Ausdruck> ;
```

Hier ist noch die Syntax für die Selektion:

Syntax: (Cond)

```
<Cond>      ::= cond ( <BoolAusdr> , <Ausdruck> , <Ausdruck> )
<BoolAusdr> ::= true | false | ( <Ausdruck> <VgIOP> <Ausdruck> ) |
                  ( <BoolAusdr> <LogOp> <BoolAusdr> ) |
                  ! ( <BoolAusdr> )
<VgIOP>     ::= == | != | <= | >= | <=> | >=>
<LogOp>     ::= && | ||
```

Bemerkung: Beachte dass der Test auf Gleichheit als == geschrieben wird!

Syntax: (FC++ Programm)

```
<FC++-Programm> ::= { <Include> } { <Funktion> }+
<Include>       ::= #include < <DateiName> >
```

Bemerkung: (Leerzeichen) C++ Programme erlauben das Einfügen von Leerzeichen, Zeilenvorschüben und Tabulatoren („whitespace“) um Programme für den Menschen lesbarer zu gestalten. Hierbei gilt folgendes zu beachten:

- Bezeichner, Zahlen, Schlüsselwörter und Operatorzeichen dürfen keinen Whitespace enthalten:
 - zaehler statt zae hler,
 - 893371 statt 89 3371,
 - return statt re tur n,
 - && statt & &.
- Folgen zwei Bezeichner, Zahlen oder Schlüsselwörter nacheinander so muss ein Whitespace (also mindestens ein Leerzeichen) dazwischen stehen:
 - int f(int x) statt intf(intx),
 - return x; statt returnx;.

Kommentare

Mit Hilfe von Kommentaren kann man in einem Programmtext Hinweise an einen menschlichen Leser einbauen. Hier bietet C++ zwei Möglichkeiten an:

```
// nach // wird der Rest der Zeile ignoriert
/* Alles dazwischen ist Kommentar (auch über
   mehrere Zeilen)
*/
```

Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



EBNF und kontextfreie Sprachen

Die obige Syntaxbeschreibung mit EBNF ist nicht mächtig genug, um fehlerfrei übersetzbare C++ Programme zu charakterisieren. So enthält die Syntaxbeschreibung üblicherweise nicht solche Regeln wie:

- Kein Funktionsname darf doppelt vorkommen.
- Genau eine Funktion muss `main` heißen.
- Namen müssen an der Stelle bekannt sein wo sie vorkommen.

Bemerkung: Mit Hilfe der EBNF lassen sich sogenannte **kontextfreie Sprachen** definieren. Entscheidend ist, dass in EBNF-Regeln links immer nur genau ein nicht-terminales Symbol steht. Zu jeder kontextfreien Sprache kann man ein Programm (genauer: einen **Kellerautomaten**) angeben, das für jedes vorgelegte Wort in endlicher Zeit entscheidet, ob es in der Sprache ist oder nicht. Man sagt: kontextfreie Sprachen sind **entscheidbar**. Die Regel „Kein Funktionsname darf doppelt vorkommen“ lässt sich mit einer kontextfreien Sprache nicht formulieren und wird deshalb extra gestellt.

Von regulären Sprachen zu kontextfreien

- Endliche Automaten (endl. Anzahl Zustände) erzeugen reguläre Sprachen:

$A \rightarrow aB$ (rechtsregulär) oder

$A \rightarrow Ba$ (linksregulär)

$A \rightarrow a$

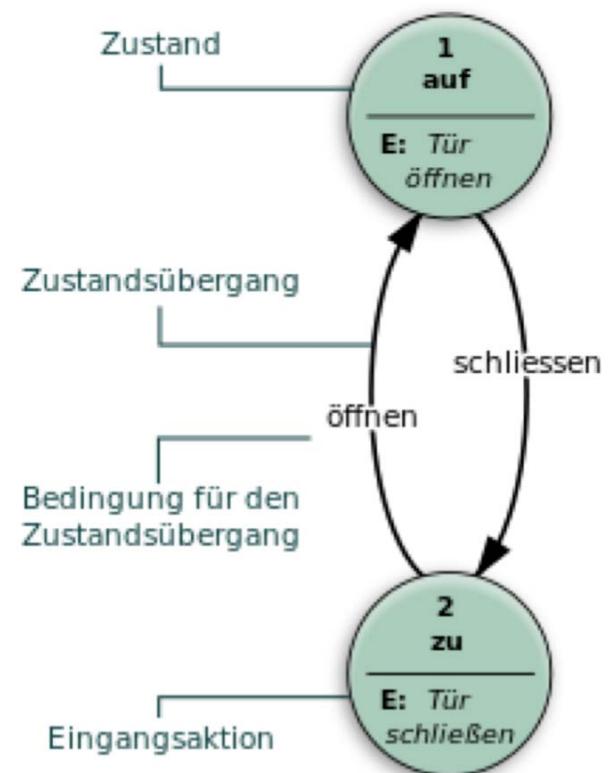
$A \rightarrow \epsilon$

$A, B \in N, a \in T$

Nur links- oder nur rechtsreguläre
Produktionen

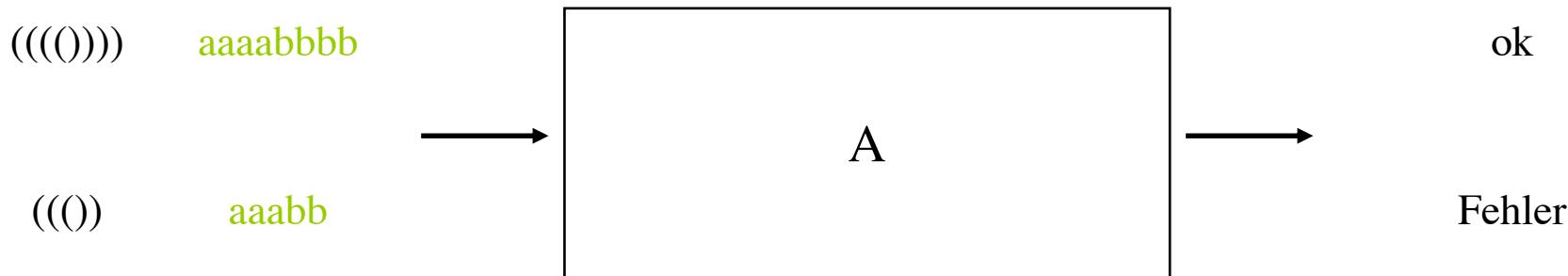
T: Terminalsymbol

N: Nichtterminalsymbole



Grenzen von endlichen Automaten

Problem: Kann man Klammersprachen mit endlichen Automaten erkennen?



Wir betrachten hier korrekte Klammerausdrücke der Gestalt $((\dots))$, die nach einer Anzahl öffnender Klammern genauso viele schließende Klammern haben. So ist $((()$ ein korrekter Klammerausdruck, während die Ausdrücke $(($ und $(()))$ keine korrekten Klammerausdrücke in unserem Sinne sind.

Versuche, einen endlichen Automaten A zur Erkennung solcher Klammerausdrücke zu konstruieren, scheitern an der Schwierigkeit, die Anzahl der öffnenden Klammern im Automaten mitzuzählen. Es scheint, dass diese Schwierigkeit bei endlichen Automaten - die ja eine feste Anzahl von Zuständen haben - unüberwindbar ist. Die folgenden Argumentationen zeigen, dass das tatsächlich der Fall ist.

Grenzen von endlichen Automaten

Gibt es einen endlichen Automaten, der $L = \{a^n b^n \mid n = 1, 2, 3, \dots\}$ erkennt?



Angenommen, es gibt einen endlichen Automaten A mit $L(A) = L$. Dieser Automat A hat eine feste Anzahl Zustände, etwa $m = 15$ (die Zahl 15 ist hier willkürlich gewählt, sie spielt für die Argumentation keine Rolle).

Wie wählen nun ein Wort $w = a^k b^k$ aus $L = \{a^n b^n \mid n = 1, 2, 3, \dots\}$ aus mit $k > m$, etwa $k = 16$. Bei der Abarbeitung des Wortes $w = a^k b^k$ muss bereits bei der Verarbeitung der 16 a's mindestens ein Zustand z mindestens zweimal durchlaufen werden, denn es gibt mehr a's als Zustände.

Grenzen von endlichen Automaten

Der Automat akzeptiert folglich auch Wörter, die nicht zu $L = \{a^n b^n \mid n = 1, 2, 3, \dots\}$ gehören. Das steht aber im Widerspruch zur Annahme, dass der Automat A die Sprache $L = \{a^n b^n \mid n = 1, 2, 3, \dots\}$ erkennt.

Da die Annahme, dass es einen endlichen Automaten gibt, der die Sprache $L = \{a^n b^n \mid n = 1, 2, 3, \dots\}$ erkennt, zu einem Widerspruch führt, muss die Annahme falsch sein.

Satz (über die Grenzen von endlichen Automaten):

Die Sprache $L = \{a^n b^n \mid n = 1, 2, 3, \dots\}$ kann nicht von einem endlichen Automaten erkannt werden. Sie ist also **nicht regulär**.

Chomsky-Hierarchie

Typ	Grammatik	äquivalenter Automat
0	allgemein	(nicht)deterministische Turingmaschine
1	kontextsensitiv	nichtdet. linear beschr. Turingmaschine
2	kontextfrei	nichtdeterministischer Kellerautomat
3	regulär	(nicht)deterministischer endl. Automat

Kontextfreie Grammatiken

Kontextfreie Grammatiken, kurz:

KFGs

werden zur Modellierung von

beliebig tief geschachtelten baumartigen Strukturen

eingesetzt.

KFGs werden unter Anderen angewandt,

- um **Programmiersprachen** wie etwa Java, C oder Pascal zu beschreiben und spielen auch beim „Compilerbau“ eine zentrale Rolle,
- um **Datenaustauschformate**, d.h. Sprachen (wie etwa HTML oder XML) als Schnittstelle zwischen Software-Werkzeugen, zu beschreiben,
- um **Bäume** zur Repräsentation strukturierter Daten (z.B. XML) zu beschreiben.

KFG – formale Definition

Eine kontextfreie Grammatik

$$G = (\Sigma, V, S, P)$$

besteht aus

- einer endlichen Menge Σ von **Terminalen** und einer endlichen Menge V von **Nichtterminalen** (oder **Variablen**).
 - ▶ Die Mengen Σ und V sind disjunkt, d.h. $\Sigma \cap V = \emptyset$ gilt.
 - ▶ Die Menge $W := \Sigma \cup V$ heißt **Vokabular**,
die Elemente in W nennt man auch Symbole,
- einem Symbol $S \in V$, dem **Startsymbol** und
- einer endlichen Menge

$$P \subseteq V \times W^*$$

von **Produktionen**.

Für eine Produktion $(A, x) \in P$ schreiben wir meistens $A \rightarrow x$.

Bsp.: Arithmetische Ausdrücke

Wir möchten alle „wohl-gebideten“ arithmetische Ausdrücke beschreiben,

- die über den Zahlen 1, 2, 3 gebildet sind und
- die Operatoren $+, -, \cdot$ sowie Klammern $(,)$ benutzen.

Beispiele für wohl-gebildete arithmetische Ausdrücke sind

$$(1 + 3) \cdot (2 + 2 + 3) - 1$$

und

$$(1 + 3) \cdot ((2 + 2 + 3) - 1).$$

Bsp.: Arithmetische Ausdrücke

Wir betrachten die KFG $G_{AA} := (\Sigma, V, S, P)$ mit

- Terminalalphabet $\Sigma := \{1, 2, 3, +, -, \cdot, (,)\}$
- Nichtterminalalphabet $V := \{\text{Ausdruck}, \text{Operator}\}$
- Startsymbol $S := \text{Ausdruck}$
- und der Produktionsmenge

$$\begin{aligned} P := \{ & \quad \text{Ausdruck} \rightarrow 1, \\ & \quad \text{Ausdruck} \rightarrow 2, \\ & \quad \text{Ausdruck} \rightarrow 3, \\ & \quad \text{Ausdruck} \rightarrow \text{Ausdruck Operator Ausdruck}, \\ & \quad \text{Ausdruck} \rightarrow (\text{Ausdruck}), \\ \\ & \quad \text{Operator} \rightarrow +, \\ & \quad \text{Operator} \rightarrow -, \\ & \quad \text{Operatur} \rightarrow \cdot \} \end{aligned}$$

Kompaktere Notation

Wir fassen Zeilen, die das gleiche Nichtterminal auf der linken Seite des Pfeils aufweisen, zu einer einzigen Zeile zusammen.

Damit können wir die Produktionsmenge P auch kurz wie folgt beschreiben:

$$\begin{aligned} P = \{ & \text{ Ausdruck } \rightarrow 1 \mid 2 \mid 3 , \\ & \text{ Ausdruck } \rightarrow \text{ Ausdruck Operator Ausdruck } \mid (\text{ Ausdruck }) , \\ & \text{ Operator } \rightarrow + \mid - \mid \cdot \} . \end{aligned}$$

Die Produktion $\text{Ausdruck} \rightarrow \text{Ausdruck Operator Ausdruck}$ können wir auffassen

- als **Strukturregel**, die besagt „Ein Ausdruck besteht aus einem Ausdruck, gefolgt von einem Operator, gefolgt von einem Ausdruck — oder als
- **Ersetzungsregel**, die besagt, dass das „Symbol Ausdruck durch das Wort Ausdruck Operator Ausdruck ersetzt werden kann.“

Ableitungsschritte

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

Falls

$$A \rightarrow x$$

eine Produktion in P ist und $u \in W^*$ und $v \in W^*$ beliebige Worte über dem Vokabular $W = \Sigma \cup V$ sind, so schreiben wir

$$uAv \implies_G uxv \quad (\text{bzw. kurz: } uAv \implies uxv)$$

und sagen, dass uAv in einem **Ableitungsschritt** zu uxv umgeformt werden kann.

Ableitungen

Eine **Ableitung** ist eine endliche Folge von hintereinander angewendeten Ableitungsschritten.

Für Worte $w \in W^*$ und $w' \in W^*$ schreiben wir

$$w \xrightarrow{*} G w' \quad (\text{bzw. kurz: } w \xrightarrow{*} w'),$$

um auszusagen, dass es eine endliche Folge von Ableitungsschritten gibt, die w zu w' umformt.

Spezialfall: Diese Folge darf auch aus 0 Ableitungsschritten bestehen, d.h. f.a. $w \in W^*$ gilt: $w \xrightarrow{} G w$.

KFG und ihre Sprache

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

Die **von G erzeugte Sprache $L(G)$** ist die Menge aller Worte über dem Terminalalphabet Σ , die aus dem Startsymbol S abgeleitet werden können. D.h.:

$$L(G) := \{w \in \Sigma^* : S \xrightarrow{*} G w\}.$$

Produktionsregeln der Form:

$$A \rightarrow \gamma$$

$$A \in N, \gamma \in V^*$$

$V := N \cup T$... gesamtes *Vokabular*)

T: Terminalsymbol

N: Nichterminalsymbole

Bsp.: Wohlgeformte Klammerausdrücke

Sei D die Sprache aller wohl-geformten Klammerausdrücke über $\Sigma = \{ (,) \}$.

① **Wohl-geformte Klammerausdrücke** sind

- ▶ $(), ((()), ((())()()((()), ((())$

② **Nicht wohl-geformt** sind

- ▶ $((()), ((()$

Es ist $D = L(G)$ für die kontextfreie Grammatik $G = (\Sigma, \{S\}, S, P)$ mit den Produktionen

$$S \rightarrow SS \mid (S) \mid \epsilon.$$

Eine Sprache $L \subseteq \Sigma^*$ heißt genau dann

kontextfrei,

wenn es eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$ gibt mit

$$L = L(G).$$

KFG und Programmiersprachen: Bsp. Pascal

Wir beschreiben einen (allerdings sehr kleinen) Ausschnitt von Pascal durch eine kontextfreie Grammatik.

- Wir benutzen das Alphabet $\Sigma = \{a, \dots, z, ;, :=, \text{begin}, \text{end}, \text{while}, \text{do}\}$ und
- die Variablen S , statements, statement, assign-statement, while-statement, variable, boolean, expression.
- variable, boolean und expression sind im Folgenden nicht weiter ausgeführt.

S	\rightarrow	begin statements end
statements	\rightarrow	statement statement ; statements
statement	\rightarrow	assign-statement while-statement
assign-statement	\rightarrow	variable := expression
while-statement	\rightarrow	while boolean do statements

KFG und Programmiersprachen

Lassen sich die **syntaktisch korrekten** Programme einer modernen Programmiersprache durch eine kontextfreie Sprache definieren?

- 1. Antwort: **Nein.** In Pascal muss zum Beispiel sichergestellt werden, dass Anzahl und Typen der formalen und aktuellen Parameter übereinstimmen.
 - ▶ Die Sprache $\{ww : w \in \Sigma^*\}$ wird sich als **nicht** kontextfrei herausstellen.
- 2. Antwort: **Im Wesentlichen ja**, wenn man „Details“ wie Typ-Deklarationen und Typ-Überprüfungen ausklammert:
 - ▶ Man beschreibt die Syntax durch eine kontextfreie Grammatik, die alle syntaktisch korrekten Programme erzeugt.
 - ▶ Allerdings werden auch syntaktisch inkorrekte Progamme (z.B. aufgrund von Typ-Inkonsistenzen) erzeugt.

(E)BNF und die Syntaxanalyse

Die Backus-Naur Normalform ([BNF](#)) wird zur Formalisierung der Syntax von Programmiersprachen genutzt.

- Sie ist ein „Dialekt“ der kontextfreien Grammatiken.

Produktionen der Form

$$X \rightarrow aYb$$

(mit $X, Y \in V$ und $a, b \in \Sigma$) werden in BNF notiert als

$$\langle X \rangle ::= a \langle Y \rangle b$$

- **Beispiel:** Eine Beschreibung der [Syntax von Java](#) in einer Variante der BNF auf <http://docs.oracle.com/javase/specs/jls/se8/html/index.html>

Eine effiziente [Syntaxanalyse](#) ist möglich.

Frage: Was ist eine Syntaxanalyse?

Antwort: Die Bestimmung einer Ableitung bzw. eines Ableitungsbaums.

Und was ist ein Ableitungsbaum?

Ableitungen und Ableitungsbäume

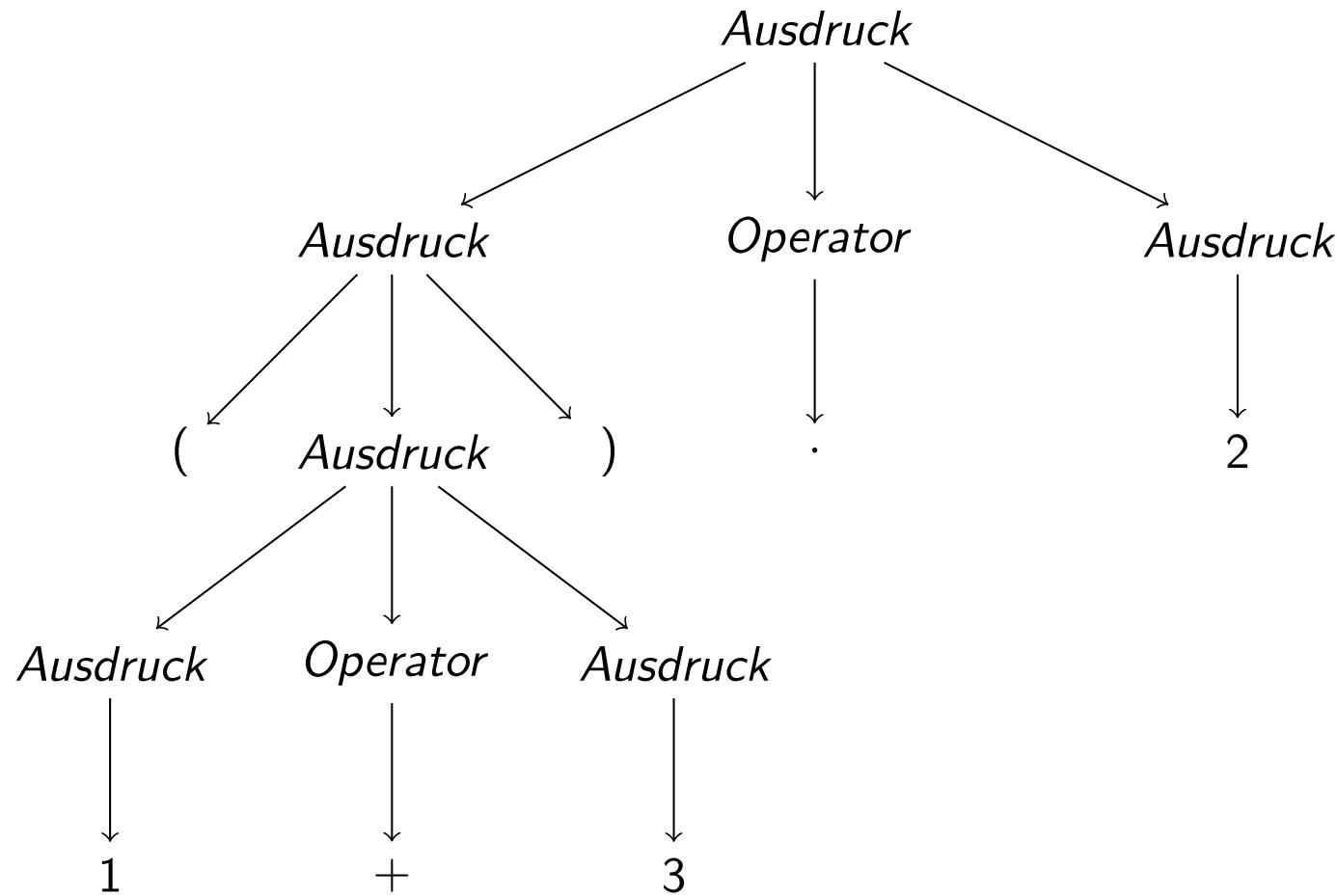
Ableitungen lassen sich am besten mit **Ableitungsbäumen** veranschaulichen.

Betrachte dazu in G_{AA} die Ableitung

Ausdruck \implies *Ausdruck Operator Ausdruck*
 \implies (*Ausdruck*) *Operator Ausdruck*
 \implies (*Ausdruck Operator Ausdruck*) *Operator Ausdruck*
 \implies (*Ausdruck + Ausdruck*) *Operator Ausdruck*
 \implies (*Ausdruck + Ausdruck*) · *Ausdruck*
 \implies (1 + *Ausdruck*) · *Ausdruck*
 \implies (1 + 3) · *Ausdruck*
 \implies (1 + 3) · 2 ,

Ableitungen und Ableitungsbäume

Diese Ableitung hat den folgenden **Ableitungsbaum**:



Beachte: Ein Ableitungsbaum kann mehrere Ableitungen repräsentieren.

Ableitungsbäume, Def. (1/2)

Sei $G = (\Sigma, V, S, P)$ eine KFG und sei $w \in L(G)$.

Jede Ableitung

$$S \xrightarrow{*} G w$$

lässt sich als gerichteter Baum darstellen, bei dem

1. jeder Knoten mit einem Symbol aus $\Sigma \cup V \cup \{\varepsilon\}$ markiert ist und
2. die Kinder jedes Knotens eine festgelegte Reihenfolge haben.
 - ▶ In der Zeichnung eines Ableitungsbäums werden von links nach rechts zunächst das „erste Kind“ dargestellt, dann das zweite, dritte etc.
 - ▶ Der Ableitungsbau ist also ein geordneter Baum.

Ableitungsbäume, Def. (2/2)

3. Die Wurzel des Baums ist mit dem Startsymbol S markiert.
4. Jeder Knoten mit seinen Kindern repräsentiert die Anwendung einer Produktion aus P , also einer Produktion

$$A \rightarrow x \text{ mit } A \in V, x \in (V \cup \Sigma)^+.$$

Die Anwendung der Produktion wird im Ableitungsbaum repräsentiert durch einen Knoten v , der mit dem Symbol A markiert ist.

- ▶ Wenn $x \in (V \cup \Sigma)^+$, dann hat v genau $|x|$ viele Kinder, so dass das i -te Kind mit dem i -ten Symbol von x markiert ist (f.a. $i \in \{1, \dots, |x|\}$).
- ▶ Wenn $x = \varepsilon$, dann hat v genau ein Kind, das mit ε markiert ist.

Wie versteht ein Compiler ein syntaktisch korrektes Programm p ?

Indem der Compiler den Ableitungsbaum von p bestimmt.

Und wenn es mehrere Ableitungsbäume für p gibt?

Die Spezifikation der Programmiersprache, –also die KFG im Fall von Java– **muss garantieren**, dass es stets nur einen Ableitungsbaum gibt.

*Solche KFGs heißen **eindeutig**.*

KFG - Zusammenfassung

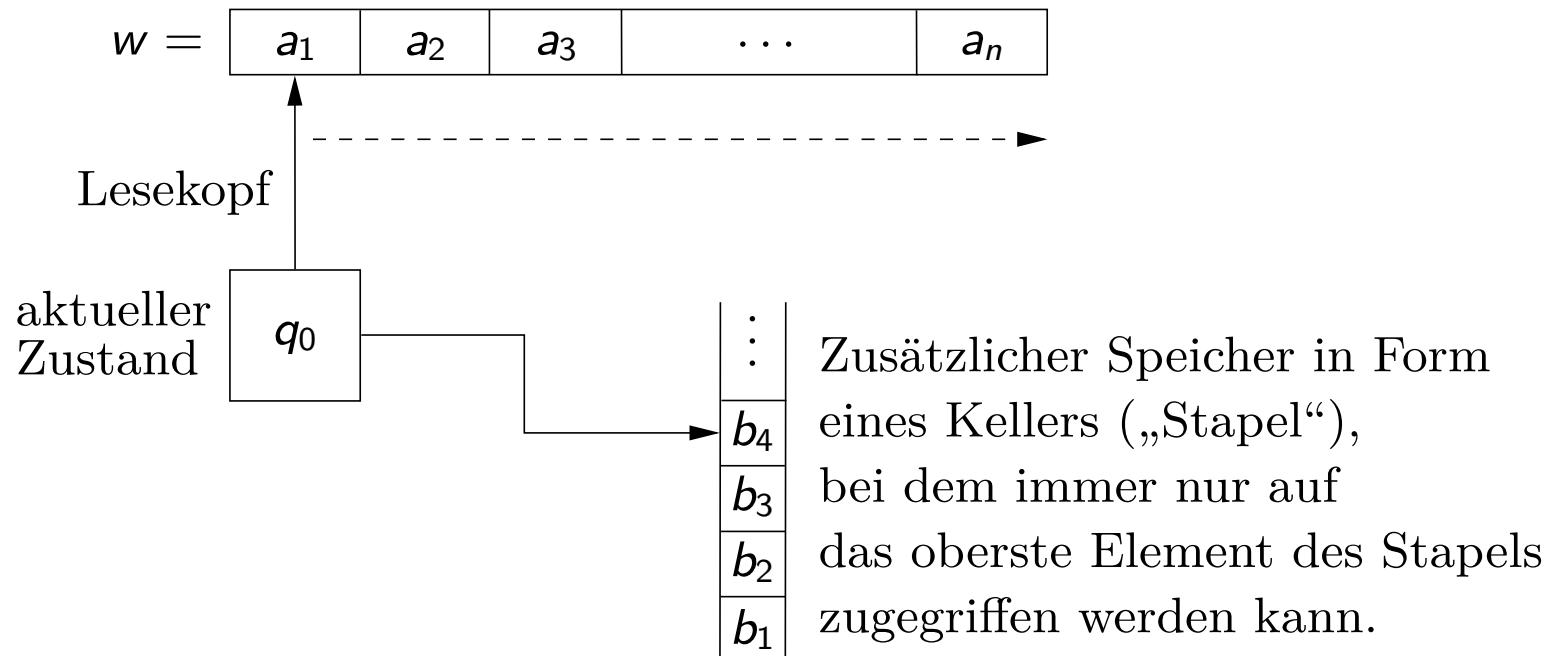
KFGs werden eingesetzt, um

rekursiv definierte Strukturen

zu modellieren.

KFG – Ausblick: Kellerautomaten

Schematische Darstellung der Verarbeitung eines Eingabeworts durch einen **Kellerautomaten**:



In der Theoretischen Informatik wird gezeigt, dass kontextfreie Grammatiken und nichtdeterministische Kellerautomaten genau die Klasse der kontextfreien Sprachen erzeugen, bzw. akzeptieren.

Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Lambda Kalkül



- 1930er von Alonzo Church und Stephen Cole Kleene eingeführt
- Formale Sprache zur Untersuchung von Funktionen
- Erlaubt klare Definition, einer berechenbaren Funktion
- Wichtiges Konstrukt für die Theoretische Informatik
- Der Lambda-Kalkül hat die Entwicklung funktionaler Programmiersprachen wesentlich beeinflusst.

Motivation

- Funktion die x inkrementiert: $x \mapsto x + 1$
- Formalisierung im λ -Kalkül: $\lambda x. x + 1$
- Freie Variable x durch λ -Abstraktion gebunden
- Bindung im Allgemeinen:
 - Mengenlehre $\{x \mid \Phi(x)\}$
 - Analysis $\int_0^1 f(x) dx$
 - Logik $\forall x \Phi(x)$ und $\exists x \Phi(x)$

λ -Abstraktion

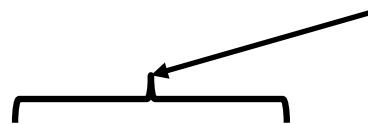
- Abstrahierte Variable muss nicht im Term vorkommen: $\lambda x. 1 \Leftrightarrow x \mapsto 1$
- $\lambda x. y \Leftrightarrow x \mapsto y$
- Nachträgliche Abstraktion von y :
 $\lambda y. \lambda x. y \Leftrightarrow$ Funktion die konstant y ist
- Argumente die Funktionen sind:
 - $f^2 = f \circ f \Leftrightarrow \lambda x. f(f(x))$
 - $f \mapsto f^2 = f \circ f \Leftrightarrow \lambda f. \lambda x. f(f(x))$

Applikation

- Anwendung von λ -Termen auf ein Argument (=Applikation): schreibe $f x$ statt $f(x)$
- Linksassoziativ:
 $f x y = (f x) y$ steht für $(f(x)) (y)$

β -Konversion

- Argument a auf $\lambda x. \theta$ anwenden: $(\lambda x. \theta) a$
⇒ in θ jedes Vorkommen von x durch a ersetzen:

$$(\lambda x. \theta)a \equiv \theta[x \leftarrow a]$$


α -Konversion

- **Namen von gebundenen Variablen sind beliebig:** $\lambda x. x \equiv \lambda y. y$
- **Allgemein:** $\lambda x. \theta \equiv \lambda y. \theta[x \leftarrow y]$

λ -Kalkül: Formale Definition

$\langle \text{Term} \rangle ::= \langle \text{Var} \rangle \ (\textit{Variable})$
| $(\langle \text{Term} \rangle \ \langle \text{Term} \rangle) \ (\textit{Applikation})$
| $\lambda \langle \text{Var} \rangle. \ \langle \text{Term} \rangle \ (\textit{Abstraktion})$

Bemerkung: im Lambda-Kalkül wird jeder Term als einstellige Funktion verstanden (eine freie Var.). Zweistellige Fkt.nen (z.B. Addition) müssen auf Umwegen substituiert werden.

Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Syntax & Semantik: Substitutionsmodell

Selbst wenn ein Programm vom Übersetzer fehlerfrei übersetzt wird, muss es noch lange nicht korrekt funktionieren. Was das Programm tut bezeichnet man als *Semantik* (Bedeutungslehre). Das in diesem Abschnitt vorgestellte **Substitutionsmodell** kann die Wirkungsweise **funktionaler** Programme beschreiben.

Definition: (Substitutionsmodell) Die Auswertung von Ausdrücken geschieht wie folgt:

1. $<\text{Zahl}>$ wird als die Zahl selbst ausgewertet.
2. $<\text{Name}> (<a_1>, <a_2>, \dots, <a_n>)$ wird für Elementarfunktionen folgendermaßen ausgewertet:
 - (a) Werte die Argumente aus. Diese sind wieder Ausdrücke. Unsere Definition ist also **rekursiv!**
 - (b) Werte die Elementarfunktion $<\text{Name}>$ auf den so berechneten Werten aus.

Syntax & Semantik: Substitutionsmodell

3. $\langle \text{Name} \rangle (\langle a_1 \rangle, \langle a_2 \rangle, \dots, \langle a_n \rangle)$ wird für benutzerdefinierte Funktionen folgendermaßen ausgewertet:

- (a) Werte die Argumente aus.
- (b) Werte den Rumpf der Funktion $\langle \text{Name} \rangle$ aus, wobei jedes Vorkommen eines formalen Parameters durch den entsprechenden Wert des Arguments ersetzt wird. Der Rumpf besteht im wesentlichen ebenfalls wieder aus der Auswertung eines Ausdrucks.

4. cond ($\langle a_1 \rangle, \langle a_2 \rangle, \langle a_3 \rangle$) wird ausgewertet gemäß:

- (a) Werte $\langle a_1 \rangle$ aus.
- (b) Ist der erhaltene Wert `true`, so erhält man den Wert des cond-Ausdrucks durch Auswertung von $\langle a_2 \rangle$, ansonsten von $\langle a_3 \rangle$. **Wichtig:** nur **eines** der beiden Argumente $\langle a_2 \rangle$ oder $\langle a_3 \rangle$ wird ausgewertet.

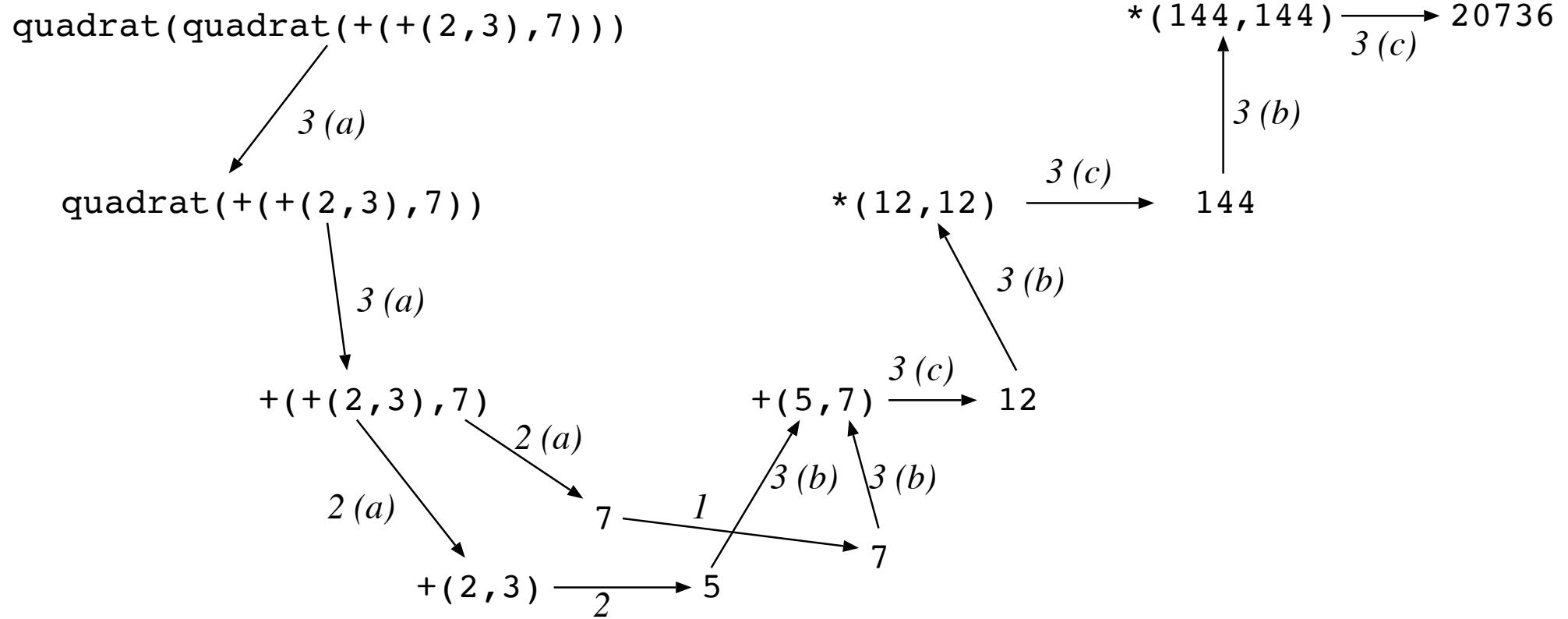
Bemerkung: Die Namen der formalen Parameter sind egal, sie entsprechen sogenannten **gebundenen Variablen** in logischen Ausdrücken.

Beispiel:

```
quadrat(3) = *( 3, 3 ) = 9
```

Beispiel:

```
quadrat( quadrat( (2+3)+7 ) )
= quadrat( quadrat( +( +( 2, 3 ), 7 ) ) )
= quadrat( quadrat( +( 5 , 7 ) ) )
= quadrat( quadrat( 12 ) )
= quadrat( *( 12, 12 ) )
= quadrat( 144 )
= *( 144, 144 )
= 20736
```



Aufbau einer Kette von verzögerten Funktionsaufrufen.

Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Linear-rekursive Prozesse

Beispiel: (Fakultätsfunktion) Sei $n \in \mathbb{N}$. Dann gilt

$$\begin{aligned} n! &= \prod_{i=1}^n i, \\ &= 1 \cdot 2 \cdot 3 \cdot \dots \cdot n. \end{aligned}$$

Oder **rekursiv**:

$$n! = \begin{cases} 1 & n = 1, \\ n(n-1)! & n > 1. \end{cases}$$

Programm: (Rekursive Berechnung der Fakultät [fakultaet.cc])

```
#include "fcpp.hh"

int fakultaet( int n )
{
    return cond( n<=1, 1, n*fakultaet(n-1) );
}

int main()
{
    return print( fakultaet(5) );
}
```

Die Auswertung kann mithilfe des Substitutionsmodells wie folgt geschehen:

```
fakultaet(5) = *( 5, fakultaet(4) )
                = *( 5, *( 4, fakultaet(3) ) )
                = *( 5, *( 4, *( 3, fakultaet(2) ) ) )
                = *( 5, *( 4, *( 3, *( 2, fakultaet(1) ) ) ) )
                = *( 5, *( 4, *( 3, *( 2, 1
                                ) ) ) )
                = *( 5, *( 4, *( 3, 2
                                ) ) ) )
                = *( 5, *( 4, 6
                                ) ) )
                = *( 5, 24
                                ) )
                = 120
```

Definition: Dies bezeichnen wir als **linear rekursiven Prozess** (die Zahl der **verzögerten** Operationen wächst linear in n). Die Aufrufe formen eine lineare Kette von Funktionsaufrufen.

Linear-iterative Prozesse

Interessanterweise lässt sich die Kette verzögerter Operationen bei der Fakultätsberechnung vermeiden. Betrachte dazu folgendes Tableau von Werten von n und $n!$:

n	1	2	3	4	5	6	...
		↓	↓	↓	↓	↓	
$n!$	1	→ 2	→ 6	→ 24	→ 120	→ 720	...

Idee: Führe das Produkt als zusätzliches Argument mit.

Programm: (Iterative Fakultätsberechnung [fakultaetiter.cc])

```
#include "fcpp.hh"

int faklter( int produkt, int zaehler, int ende )
{
    return cond( zaehler>ende,
                  produkt,
                  faklter( produkt*zaehler, zaehler+1, ende ) );
}

int fakultaet( int n )
{
    return faklter( 1, 1, n );
}

int main()
{
    return print( fakultaet(5) );
}
```

Die Analyse mit Hilfe des Substitutionsprinzips liefert:

```
fakultaet(5) = fakIter( 1, 1, 5 )
              = fakIter( 1, 2, 5 )
              = fakIter( 2, 3, 5 )
              = fakIter( 6, 4, 5 )
              = fakIter( 24, 5, 5 )
              = fakIter( 120, 6, 5 )
              = 120
```

Hier wird allerdings von folgender Optimierung ausgegangen: In fakIter wird das Ergebnis des rekursiven Aufrufes von fakIter ohne weitere Verarbeitung zurückgegeben. In diesem Fall muss keine Kette verzögerter Aufrufe aufgebaut werden, das Endergebnis entspricht dem Wert der innersten Funktionsauswertung. Diese Optimierung kann vom Compiler durchgeführt werden (tail recursion).

Programm: (Ausgabe des Programmverlaufs [fakultaetiter_mit_ausgabe.cc])

```
#include "fcpp.hh"

int fakIter( int produkt, int zaehler, int ende )
{
    return cond( zaehler>ende,
                  produkt,
                  print( "Fak:", zaehler,
                         produkt*zaehler,
                         fakIter( produkt*zaehler,
                                   zaehler+1,
                                   ende ) ) );
}

int fakultaet( int n ) { return fakIter( 1, 1, n ); }
int main() { return dump( fakultaet(10) ); }
```

Fak: 10 3628800

Fak: 9 362880

Fak: 8 40320

Fak: 7 5040

Fak: 6 720

Fak: 5 120

Fak: 4 24

Fak: 3 6

Fak: 2 2

Fak: 1 1

- $\text{print}(f(x)) = f(x)$, Wert von $f(x)$ wird gedruckt.
- $\text{print}(f(x), g(x)) = g(x)$, Wert von $f(x)$ wird gedruckt.
- $\text{print}(f(x), g(x), h(x)) = h(x)$, Wert von $f(x)$ und $g(x)$ wird gedruckt.

Linear iterativer Prozess

Sprechweise: Dies nennt man einen **linear iterativen Prozess**. Der Zustand des Programmes lässt sich durch eine feste Zahl von Zustandsgrößen beschreiben (hier die Werte von `zaehler` und `produkt`). Es gibt eine Regel wie man von einem Zustand zum nächsten kommt, und es gibt den Endzustand.

Bemerkung:

- Von einem Zustand kann man ohne Kenntnis der Vorgeschichte aus weiterrechnen. Der Zustand fasst alle bis zu diesem Punkt im Programm durchgeföhrten Berechnungen zusammen.
- Die Zahl der durchlaufenen Zustände ist proportional zu n .
- Die Informationsmenge zur Darstellung des Zustandes ist konstant.
- Bei geeigneter Implementierung ist der Speicherplatzbedarf konstant.
- Beim Lisp-Dialekt Scheme wird diese Optimierung von am Ende aufgerufenen Funktionen (*tail-call position*) im **Sprachstandard** verlangt.
- Bei anderen Sprachen (auch C++) ist diese Optimierung oft durch Compiler-einstellungen erreichbar (nicht automatisch, weil das Debuggen erschwert wird), ist aber nicht Teil des Standards.
- Beide Arten von Prozessen werden durch rekursive Funktionen beschrieben!

Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Baumrekursion

Beispiel: (Fibonacci-Zahlen)

$$\text{fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \text{fib}(n - 1) + \text{fib}(n - 2) & n > 1 \end{cases} .$$

Die Folge der Fibonacci Zahlen modelliert (unter anderem) das Wachstum einer Kaninchenpopulation unter vereinfachten Annahmen. Sie ist benannt nach Leonardo di Pisa.¹¹

¹¹Leonardo di Pisa (auch Fibonacci), etwa 1180–1241, ital. Rechenmeister in Pisa.

Programm: (Fibonacci rekursiv [fibonacci.cc])

```
#include "fcpp.hh"

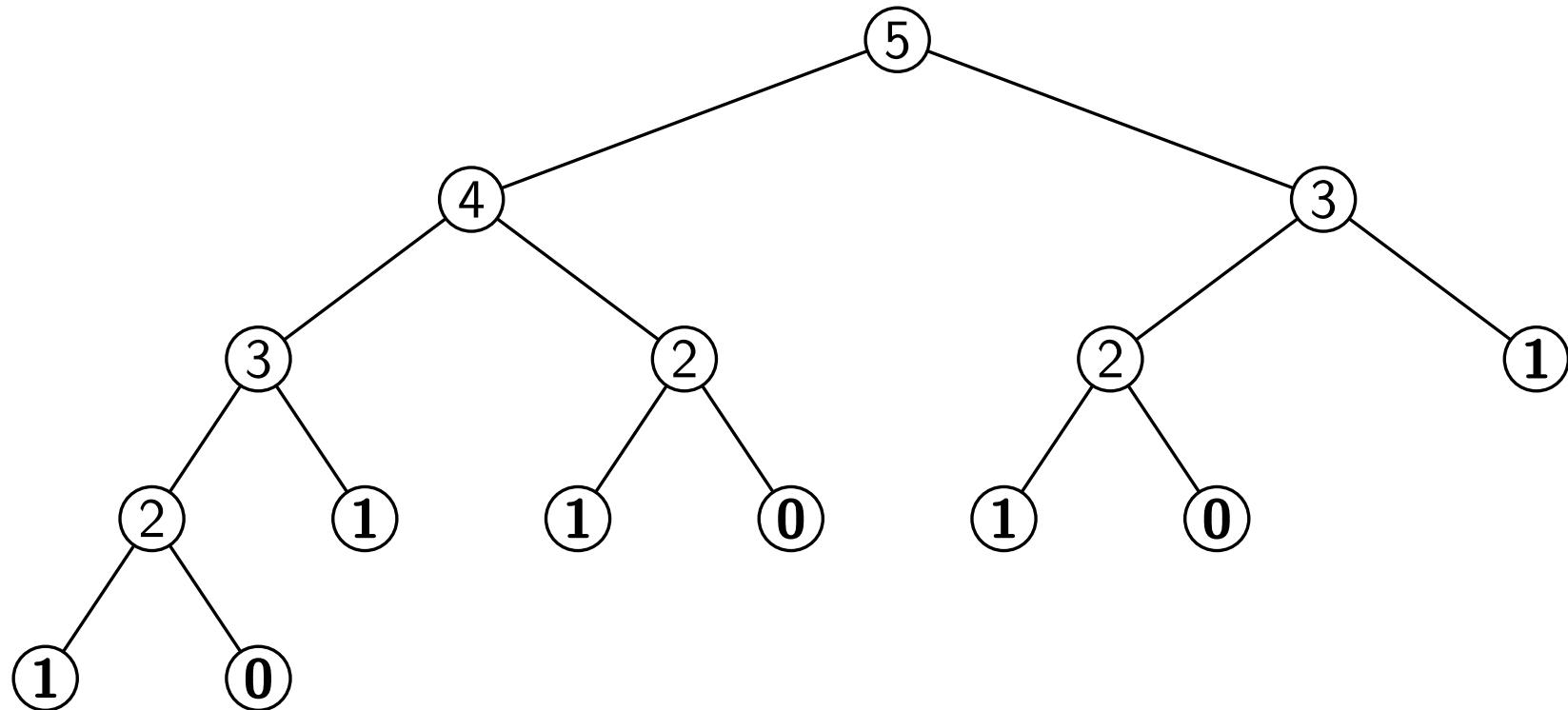
int fib( int n )
{
    return cond( n==0, 0 ,
                 cond( n==1, 1 ,
                       fib( n-1 ) + fib( n-2 ) ) );
}

int main( int argc , char *argv[] )
{
    return print( fib( readarg_int( argc , argv , 1 ) ) );
}
```

Auswertung von `fib(5)` nach dem Substitutionsmodell:

```
fib(5)
= +(fib(4),fib(3))
= +(+(fib(3),fib(2)),+(fib(2),fib(1)))
= +(+(+(fib(2),fib(1)),+(fib(1),fib(0))),+(+(fib(1),fib(0)),fib(1)))
= +(+(+(+(fib(1),fib(0)),fib(1)),+(fib(1),fib(0))),+(+(fib(1),fib(0)),fib(1)))
= +(+(+(+(1      ,0      ),1      ),+(1      ,0      )),+(+(1      ,0      ),1      ))
= +(+(+(1              ,1      ),1              ),+(1              ,1      ))
= +(+(2              ,1              ),2              )
= +(3              ,2              )
= 5
```

Graphische Darstellung des Aufrufbaumes



$\text{fib}(5)$ baut auf $\text{fib}(4)$ und $\text{fib}(3)$, $\text{fib}(4)$ baut auf $\text{fib}(3)$ und $\text{fib}(2)$, usw.

Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Bezeichnung: Der Rekursionsprozess bei der Fibonaccifunktion heißt daher **baumrekursiv**.

Frage:

- Wie schnell wächst die Anzahl der Operationen bei der rekursiven Auswertung der Fibonaccifunktion?
- Wie schnell wächst die Fibonaccifunktion selbst?

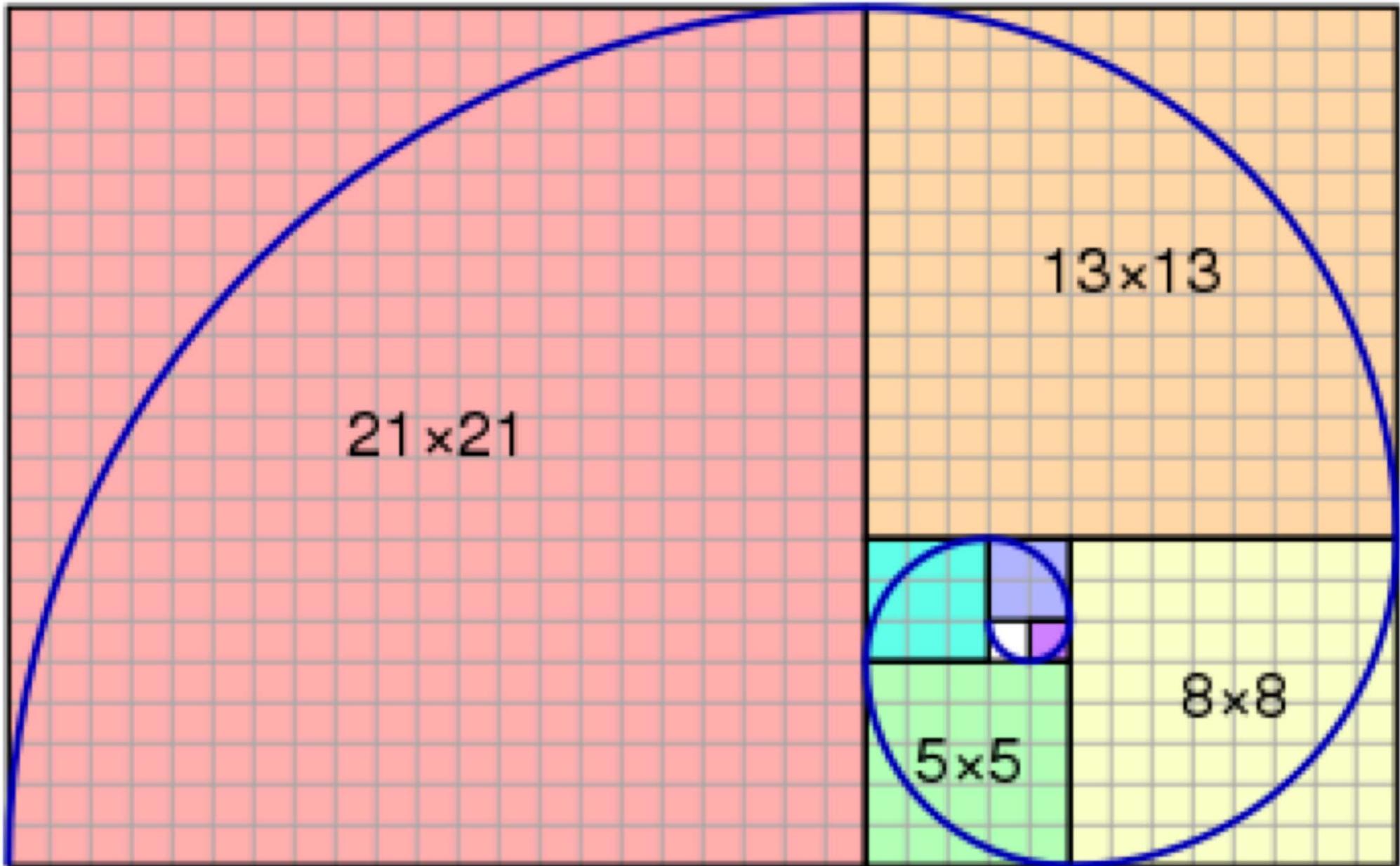
21

13

3	2
1	1

5

8



Antwort: (Wachstum von fib). $F_n := \text{fib}(n)$ erfüllt die **lineare 3-Term-Rekursion**

$$F_n = F_{n-1} + F_{n-2}$$

Die Lösungen dieser Gleichung sind von der Form $a\lambda_1^n + b\lambda_2^n$, wobei $\lambda_{1/2}$ die Lösungen der quadratischen Gleichung $\lambda^2 = \lambda + 1$ sind, also $\lambda_{1/2} = \frac{1 \pm \sqrt{5}}{2}$. Die Konstanten a und b werden durch die Anfangsbedingungen $F_0 = 0, F_1 = 1$ festgelegt und damit ergibt sich

$$F_n = \underbrace{\frac{1}{\sqrt{5}}}_{a} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \underbrace{\frac{1}{\sqrt{5}}}_{b} \left(\frac{1 - \sqrt{5}}{2} \right)^n \approx \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n$$

für große n , da $|\lambda_2| < 1$.

Bemerkung: $\lambda_1 \approx 1.61803$ ist der *goldene Schnitt*.

Antwort: (Aufwand zur rekursiven Berechnung von $\text{fib}(n)$)

- Der **Gesamtaufwand** A_n zur Auswertung von $\text{fib}(n)$ ist **größer gleich** einer Konstante c_1 multipliziert mit der Zahl B_n der **Blätter** im Berechnungsbaum:

$$A_n \geq c_1 B_n.$$

Die Zahl der Blätter B_n erfüllt die Rekursion:

$$B_0 = 1, \quad B_1 = 1, \quad B_n = B_{n-1} + B_{n-2}, \quad n > 1$$

woraus man

$$B_n = \text{fib}(n+1) \geq \frac{\lambda_1}{\sqrt{5}} \lambda_1^n - \epsilon_1$$

ersieht (Beachte $B_0 = 1!$). Die Ungleichung gilt für $n \geq N_1(\epsilon_1)$.

- Der Gesamtaufwand A_n zur Auswertung von `fib(n)` ist **kleiner gleich** einer Konstante c_2 multipliziert mit der Anzahl G_n der Knoten im Baum:

$$A_n \leq c_2 G_n.$$

Diese erfüllt:

$$G_0 = 1, \quad G_1 = 1, \quad G_n = G_{n-1} + G_{n-2} + 1, \quad n > 1.$$

Durch die Transformation $G_n = G'_n - 1$ ist dies äquivalent zu

$$G'_0 = 2, \quad G'_1 = 2, \quad G'_n = G'_{n-1} + G'_{n-2}, \quad n > 1.$$

Mit den Methoden von oben erhält man

$$G'_n = \left(1 + \frac{1}{\sqrt{5}}\right) \lambda_1^n + \left(1 - \frac{1}{\sqrt{5}}\right) \lambda_2^n \leq \left(1 + \frac{1}{\sqrt{5}}\right) \lambda_1^n + \epsilon_2$$

für $n \geq N_2(\epsilon_2)$.

Damit erhalten wir also zusammengefasst:

$$c_1 \frac{\lambda_1}{\sqrt{5}} \lambda_1^n - c_1 \epsilon_1 \leq A_n \leq c_2 \left(1 + \frac{1}{\sqrt{5}}\right) \lambda_1^n + c_2 \epsilon_2$$

für $n \geq \max(N_1(\epsilon_1), N_2(\epsilon_2))$.

Bemerkung:

- Der Rechenaufwand wächst somit **exponentiell**.
- Der Speicherbedarf wächst hingegen nur **linear** in n .

Auch die Fibonaccizahlen kann man iterativ berechnen indem man die aktuelle Summe mitführt:

Programm: (Fibonacci iterativ [fibiter.cc])

```
#include "fcpp.hh"

int fibIter( int letzte, int vorletzte, int zaehler )
{
    return cond( zaehler==0,
                 vorletzte,
                 fibIter( vorletzte+letzte, letzte, zaehler-1 ) );
}

int fib( int n )
{
    return fibIter( 1, 0, n );
}

int main( int argc, char *argv[] )
{
    return print( fib( readarg_int( argc, argv, 1 ) ) );
}
```

Hier liefert das Substitutionsmodell:

```
fib(2)
= fibIter(1,0,2)
= cond( 2==0, 0, fibiter(1,1,1))
= fibiter(1,1,1)
= cond( 1==0, 1, fibiter(2,1,0))
= fibIter(2,1,0)
= cond( 0==0, 1, fibiter(3,2,-1))
= 1
```

Vergleich linear iterative & baumrekursiv

Bemerkung:

- Man braucht hier offenbar drei Zustandsvariablen.
- Der Rechenaufwand des linear iterativen Prozesses ist proportional zu n , also viel kleiner als der baumrekursive.

Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Größenordnungen für Komplexitätsaussagen

Es gibt eine formale Ausdrucksweise für Komplexitätsaussagen wie „der Aufwand zur Berechnung von $\text{fib}(n)$ wächst exponentiell“.

Sei n ein Parameter der Berechnung, z. B.

- Anzahl gültiger Stellen bei der Berechnung der Quadratwurzel
- Dimension der Matrix in einem Programm für lineare Algebra
- Größe der Eingabe in Bits

Mit $R(n)$ bezeichnen wir den Bedarf an Resourcen für die Berechnung, z. B.

- Rechenzeit
- Anzahl auszuführender Operationen
- Speicherbedarf

Definition:

- $R(n) = \Omega(f(n))$, falls es von n unabhängige Konstanten c_1, n_1 gibt mit

$$R(n) \geq c_1 f(n) \quad \forall n \geq n_1.$$

- $R(n) = O(f(n))$, falls es von n unabhängige Konstanten c_2, n_2 gibt mit

$$R(n) \leq c_2 f(n) \quad \forall n \geq n_2.$$

- $R(n) = \Theta(f(n))$, falls $R(n) = \Omega(f(n)) \wedge R(n) = O(f(n))$.

Beispiel: $R(n)$ bezeichne den Rechenaufwand der rekursiven Fibonacci-Berechnung:

$$R(n) = \Omega(n), \quad R(n) = O(2^n), \quad R(n) = \Theta(\lambda_1^n)$$

Bezeichnung:

$R(n) = \Theta(1)$

konstante Komplexität

$R(n) = \Theta(\log n)$

logarithmische Komplexität

$R(n) = \Theta(n)$

lineare Komplexität

$R(n) = \Theta(n \log n)$

fast optimale Komplexität

$R(n) = \Theta(n^2)$

quadratische Komplexität

$R(n) = \Theta(n^p)$

polynomiale Komplexität

$R(n) = \Theta(a^n)$

exponentielle Komplexität

Beispiel 1: Telefonbuch

Wir betrachten den Aufwand für das Finden eines Namens in einem Telefonbuch der Seitenzahl n .

Algorithmus: (A1) Blättere das Buch von Anfang bis Ende durch.

Satz: Sei $C_1 > 0$ die (maximale) Zeit, die das Durchsuchen einer Seite benötigt. Der maximale Zeitaufwand $A_1 = A_1(n)$ für Algorithmus A1 ist dann abschätzbar durch

$$A_1(n) = C_1 n$$

Algorithmus: (A2) Rekursives Halbieren.

1. Setze $[a_1 = 1, b_1 = n]$, $i = 1$;
2. Ist $a_i = b_i$ durchsuche Seite a_i ; Fertig;
3. Setze $m = (a_i + b_i)/2$ (ganzzahlige Division);
4. Falls Name **vor** Seite m
setze $[a_{i+1} = a_i, b_{i+1} = m]$, $i = i + 1$, gehe zu 2.;
5. Falls Name **nach** Seite m
setze $[a_{i+1} = m, b_{i+1} = b_i]$, $i = i + 1$, gehe zu 2.;
6. Durchsuche Seite m ; Fertig;

Satz: Sei $C_1 > 0$ die (maximale) Zeit, die das Durchsuchen einer Seite benötigt, und $C_2 > 0$ die (maximale) Zeit für die Schritte 3–5. Der maximale Zeitaufwand $A_2 = A_2(n)$ für Algorithmus A2 ist dann abschätzbar durch

$$A_2(n) = C_1 + C_2 \log_2 n$$

Man ist vor allem an der Lösung großer Probleme interessiert. Daher interessiert der Aufwand $A(n)$ für große n .

Satz: Für große Telefonbücher ist Algorithmus 2 „besser“, d. h. der maximale Zeitaufwand ist kleiner.

Beweis:

$$\frac{A_1(n)}{A_2(n)} = \frac{C_1 n}{C_1 + C_2 \log_2 n} = \frac{n}{1 + \frac{C_2}{C_1} \log_2 n} \rightarrow +\infty$$

Beobachtung:

- Die genauen Werte von C_1, C_2 sind für diese Aussage unwichtig.
- Für spezielle Eingaben (z. B. Andreas Aalbert) kann auch Algorithmus 1 besser sein.

Bemerkung: Um „Algorithmus 2 ist für große Telefonbücher besser“ zu schließen, reichen die Informationen $A_1(n) = O(n)$ und $A_2(n) = O(\log n)$ aus. Man beachte auch, dass wegen $\log_2 n = \frac{\log n}{\log 2}$ gilt $O(\log_2 n) = O(\log n)$.

Doppelt rekursiver Prozess: Bsp. Wechselgeld

Aufgabe: Ein gegebener Geldbetrag ist unter Verwendung von Münzen zu 1, 2, 5, 10, 20 und 50 Cent zu wechseln. Wieviele verschiedene Möglichkeiten gibt es dazu?

Beachte: Die Reihenfolge in der wir die Münzen verwenden ist egal.

Idee: Es sei der Betrag a mit n verschiedenen Münzarten zu wechseln. Eine der n Münzarten habe den Nennwert d . Dann gilt:

- Entweder wir verwenden eine Münze mit Wert d , dann bleibt der Rest $a - d$ mit n Münzarten zu wechseln.
- Wir verwenden die Münze mit Wert d *überhaupt nicht*, dann müssen wir den Betrag a mit den verbleibenden $n - 1$ Münzarten wechseln.

Folgerung: Ist $A(a, n)$ die Anzahl der Möglichkeiten den Betrag a mit n Münzarten zu wechseln, und hat Münzart n den Wert d , so gilt

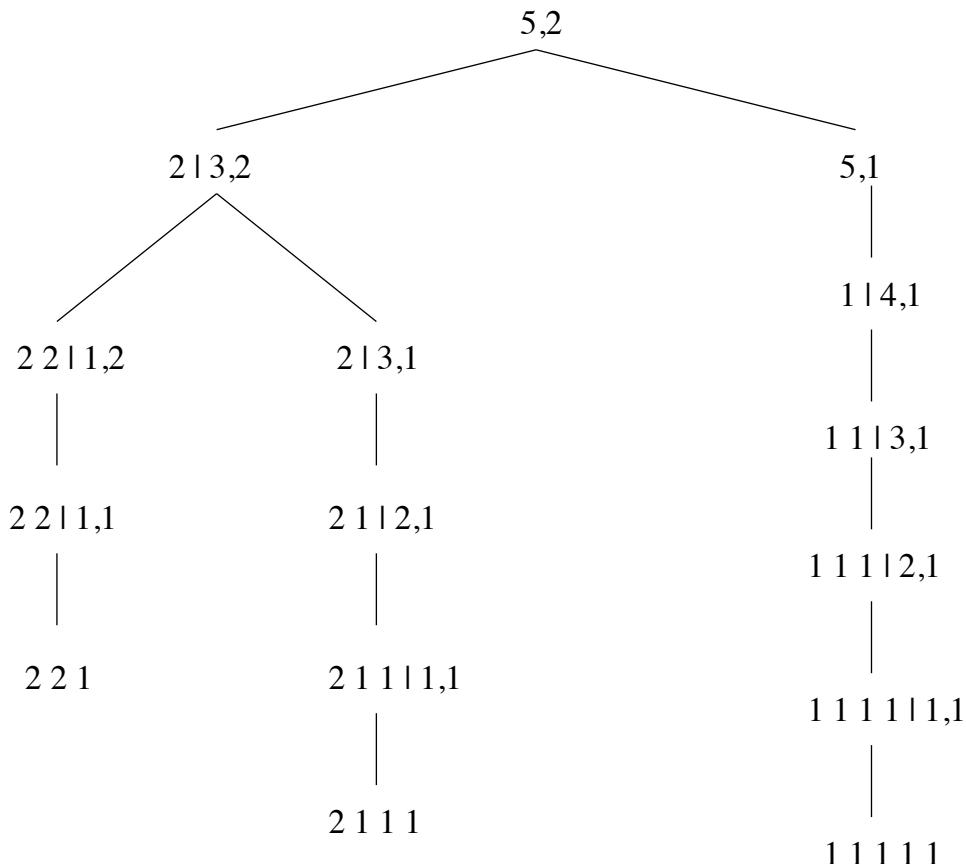
$$A(a, n) = A(a - d, n) + A(a, n - 1)$$

Dies ist ein Beispiel für eine Rekursion in zwei Argumenten.

Bemerkung: Es gilt auch:

- $A(0, n) = 1$ für alle $n \geq 0$. Wenn der Betrag a den Wert 0 erreicht hat haben wir den ursprünglichen Betrag gewechselt. ($A(0, 0)$ kann nicht vorkommen).
- $A(a, n) = 0$ falls $a > 0$ und $n = 0$. Der Betrag kann nicht gewechselt werden.
- $A(a, n) = 0$ falls $a < 0$. Der Betrag kann nicht gewechselt werden.

Beispiel: Wechseln von 5 Cent in 1 und 2 Centstücke:



Bemerkung: Dies ist wieder ein **baumrekursiver** Prozess.

Programm: (Wechselgeld zählen [wechselgeld.cc])

```
#include "fcpp.hh"

// uebersetze Muenzart in Muenzwert
int nennwert( int nr )
{
    return
        cond( nr==1, 1,
              cond( nr==2, 2,
                    cond( nr==3, 5,
                          cond( nr==4, 10,
                                cond( nr==5, 20,
                                      cond( nr==6, 50, 0 )
                                         ) ) ) ) );
}
```

•
•
•

•
•
•

```
int wg( int betrag, int muenzarten )
{
    return cond( betrag==0, 1,
                 cond( betrag<0 || muenzarten==0, 0,      => Fehler
                        wg( betrag, muenzarten-1 ) +
                        wg( betrag - nennwert(muenzarten),
                            muenzarten ) ) );
}

int wechselgeld( int betrag )
{
    return wg( betrag, 6 );
}

int main( int argc, char *argv[] ) {
    return print( wechselgeld( readarg_int( argc, argv, 1 ) ) );
}
```

Hier einige Resultate:

`wechselgeld(50) = 451`

`wechselgeld(100) = 4562`

`wechselgeld(200) = 69118`

`wechselgeld(300) = 393119`

Bemerkung: Ein iterativer Lösungsweg ist hier nicht ganz so einfach.

Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Zahlendarstellung im Rechner

In der Mathematik gibt es verschiedene Zahlenmengen:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}.$$

Diese Mengen enthalten alle unendlich viele Elemente, im Computer entsprechen die diversen Datentypen jedoch nur endlichen Mengen.

Um Zahlen aus \mathbb{N} darzustellen, benutzt man ein **Stellenwertsystem** zu einer **Basis** $\beta \geq 2$ und **Ziffern** $a_i \in \{0, \dots, \beta - 1\}$

Dann bedeutet

$$(a_{n-1}a_{n-2}\dots a_1a_0)_{\beta} \equiv \sum_{i=0}^{n-1} a_i \beta^i$$

Dabei ist n die Wortlänge. Es sind somit die folgenden Zahlen aus \mathbb{N}_0 darstellbar:

$$0, 1, \dots, \beta^n - 1$$

Am häufigsten wird $\beta = 2$, das **Binärsystem**, verwendet.

Umgang mit negativen Zahlen

In der Mathematik ist das Vorzeichen eine separate Information welche 1 Bit zur Darstellung benötigt.

Im Rechner wird bei ganzen Zahlen zur Basis $\beta = 2$ eine andere Darstellung gewählt, die **Zweierkomplementdarstellung**.

Früher war auch das **Einerkomplement** gebräuchlich.

Einer- und Zweierkomplement

Definition: Sei $(a_{n-1}a_{n-2}\dots a_1a_0)_2$ die Binärdarstellung von $a \in [0, 2^n - 1]$. Dann heisst

$$e_n(a) = e_n((a_{n-1}a_{n-2}\dots a_1a_0)_2) = (\bar{a}_{n-1}\bar{a}_{n-2}\dots \bar{a}_1\bar{a}_0)_2$$

das **Einerkomplement** von a , wobei $\bar{a}_i = 1 - a_i$.

Definition: Sei $a \in [0, 2^n - 1]$. Dann heisst

$$z_n(a) = 2^n - a$$

das **Zweierkomplement** von a .

Es gilt: $e_n(e_n(a)) = a$ und $z_n(z_n(a)) = a$!

Das Zweierkomplement einer Zahl kann sehr einfach und effizient berechnet werden.

Sei $a \in [0, 2^n - 1]$. Dann folgt aus der Identität

$$a + e_n(a) = 2^n - 1$$

die Formel

$$z_n(a) = 2^n - a = e_n(a) + 1.$$

Somit wird **keine** Subtraktion benötigt sondern es genügt das Einerkomplement und eine Addition von 1. (Und das kann noch weiter vereinfacht werden).

Definition: Die Zweierkomplementdarstellung einer Zahl ist eine bijektive Abbildung

$$d_n : [-2^{n-1}, 2^{n-1} - 1] \rightarrow [0, 2^n - 1]$$

welche definiert ist als

$$d_n(a) = \begin{cases} a & 0 \leq a < 2^{n-1} \\ 2^n - |a| & -2^{n-1} \leq a < 0 \end{cases}.$$

Die negativen Zahlen $[-2^{n-1}, -1]$ werden damit auf den Bereich $[2^{n-1}, 2^n - 1]$ positiver Zahlen abgebildet.

Sei $d_n(a) = (x_{n-1}x_{n-2}\dots x_1x_0)_2$ dann ist a positiv falls $x_{n-1} = 0$ und a negativ falls $x_{n-1} = 1$.

Es gilt $d_n(-1) = 2^n - 1 = (1, \dots, 1)_2$ und $d_n(-2^{n-1}) = 2^n - 2^{n-1} = 2^{n-1}(2-1) = 2^{n-1} = (1, 0, \dots, 0)_2$.

Operationen mit der Zweierkomplementdarstellung

Die **Ein-/Ausgabe** von ganzen Zahlen erfolgt (1) im Zehnersystem und (2) mittels separatem Vorzeichen.

Bei der **Eingabe** einer ganzen Zahl wird diese in das Zweierkomplement umgewandelt:

- Lese Betrag und Vorzeichen ein und teste auf erlaubten Bereich
- Wandle Betrag in das Zweiersystem
- Für negative Zahlen berechne das Zweierkomplement

Bei der **Ausgabe** gehe umgekehrt vor.

Im folgenden benötigen wir noch eine weitere Operation.

Definition: Sei $x = (x_{m-1}x_{m-2}\dots x_1x_0)_2$ eine m -stellige Binärzahl. Dann ist

$$s_n((x_{m-1}x_{m-2}\dots x_1x_0)_2) = \begin{cases} (x_{m-1}x_{m-2}\dots x_1x_0)_2 & m \leq n \\ (x_{n-1}x_{n-2}\dots x_1x_0)_2 & m > n \end{cases}$$

die **Beschneidung** auf n -stellige Binärzahlen.

Die Addition von Zahlen in der Zweierkomplementdarstellung gelingt mit

Satz: Sei $n \in \mathbb{N}$ und $a, b, a + b \in [-2^{n-1}, 2^{n-1} - 1]$. Dann gilt

$$d_n(a + b) = s_n(d_n(a) + d_n(b)).$$

Es genügt eine einfache Addition. Beachtung der Vorzeichen entfällt!

Beweis. Wir unterscheiden drei Fälle (mittlerer Fall steht für zwei).

$a, b \geq 0$. Damit ist auch $a + b \geq 0$. Also

$$s_n(d_n(a) + d_n(b)) = s_n(a + b) = a + b = d_n(a + b).$$

$a < 0, b \geq 0$. $a + b$ kann positiv oder negativ sein. Mit $a = -|a|$:

$$\begin{aligned} s_n(d_n(a) + d_n(b)) &= s_n(2^n - |a| + b) = s_n(2^n + (a + b)) \\ &= \begin{cases} a + b & 0 \leq a + b < 2^{n-1} \\ 2^n - |a + b| & -2^{n-1} \leq a + b < 0 \end{cases}. \end{aligned}$$

$a, b < 0$. Damit ist auch $a + b < 0$ und $2^n - |a + b| < 2^n$:

$$\begin{aligned} s_n(d_n(a) + d_n(b)) &= s_n(2^n - |a| + 2^n - |b|) = s_n(2^n + 2^n + a + b) \\ &= s_n(2^n + 2^n - |a + b|) = 2^n - |a + b|. \end{aligned}$$

Beispiel: (Zweierkomplement) Für $n = 3$ setze

$$\begin{array}{rcl} 0 & = & 000 \\ 1 & = & 001 \\ 2 & = & 010 \\ 3 & = & 011 \end{array} \quad \begin{array}{rcl} -1 & = & 111 \\ -2 & = & 110 \\ -3 & = & 101 \\ -4 & = & 100 \end{array}$$

Solange der Zahlenbereich nicht verlassen wird, klappt die normale Arithmetik ohne Beachtung des Vorzeichens:

$$\begin{array}{rcl} 3 & \rightarrow & 011 \\ -1 & \rightarrow & 111 \\ \hline 2 & \rightarrow & [1]010 \end{array}$$

$$s_n((x_{m-1}x_{m-2}\dots x_1x_0)_2) = \begin{cases} (x_{m-1}x_{m-2}\dots x_1x_0)_2 & m \leq n \\ (x_{n-1}x_{n-2}\dots x_1x_0)_2 & m > n \end{cases}$$

Die **Negation** einer Zahl in Zweierkomplementdarstellung.

Satz: Sei $n \in \mathbb{N}$ und $a \in [-2^{n-1} + 1, 2^{n-1} - 1]$. Dann gilt

$$d_n(-a) = s_n(2^n - d_n(a)) = s_n(e_n(d_n(a)) + 1).$$

Beweis.

$a = 0$. $d_n(-0) = d_n(0) = s_n(2^n - d_n(0))$. Nur dieser Fall braucht die Anwendung von s_n .

$0 < a < 2^{n-1}$. Also ist $-a < 0$ und somit

$$d_n(a) = \begin{cases} a & 0 \leq a < 2^{n-1} \\ 2^n - |a| & -2^{n-1} \leq a < 0 \end{cases}$$

$$d_n(-a) = 2^n - |a| = 2^n - a = 2^n - d_n(a) = s_n(2^n - d_n(a)).$$

$-2^{n-1} < a < 0$. Also ist $-a > 0$ und somit

$$d_n(-a) = d_n(|a|) = |a| = 2^n - (2^n - |a|) = 2^n - d_n(a) = s_n(2^n - d_n(a)).$$

Schließlich behandeln wir noch die **Subtraktion**.

Diese wird auf die Addition zurückgeführt:

$$\begin{aligned} d_n(a - b) &= d_n(a + (-b)) && a - b = a + (-b) \\ &= s_n(d_n(a) + d_n(-b)) && \text{Satz über Addition} \\ &= s_n(d_n(a) + s_n(2^n - d_n(b))) && \text{Satz über Negation.} \end{aligned}$$

Natürlich vorausgesetzt, dass alles im erlaubten Bereich ist.

Gebräuchliche Zahlenbereiche in C++

$\beta = 2$ und $n = 8, 16, 32$:

char	-128 . . . 127
unsigned char	0 . . . 255
short	-32768 . . . 32767
unsigned short	0 . . . 65535
int	-2147483648 . . . 2147483647
unsigned int	0 . . . 4294967295

Bemerkung: Die genaue Größe legt der C++ Standard nicht fest, diese kann aber abgefragt werden.

Bemerkung: Achtung: bei Berechnungen mit ganzen Zahlen führt Überlauf, d. h. das Verlassen des darstellbaren Zahlenbereichs üblicherweise **nicht** zu Fehlermeldungen. Es liegt in der Verantwortung des Programmierers solche Fehler zu vermeiden.

Darstellung reeller Zahlen

Neben den Zahlen aus \mathbb{N} und \mathbb{Z} sind in vielen Anwendungen auch reelle Zahlen \mathbb{R} von Interesse. Wie werden diese im Computer realisiert?

Festkommazahlen

Eine erste Idee ist die **Festkommazahl**. Hier interpretiert man eine gewisse Zahl von Stellen als *nach dem Komma*, d. h.

$$(a_{n-1}a_{n-2}\dots a_q.a_{q-1}\dots a_0)_{\beta} \equiv \sum_{i=0}^{n-1} a_i \beta^{i-q}$$

Beispiel: Bei $\beta = 2, q = 3$ hat man drei Nachkommastellen, kann also in Schritten von $1/8$ auflösen.

Bemerkung:

- Jede Festkommazahl ist rational, somit können irrationale Zahlen nicht exakt dargestellt werden.
- Selbst einfache rationale Zahlen können je nach Basis nicht exakt dargestellt werden. So kann $0.1 = 1/10$ mit einer Festkommazahl zur Basis $\beta = 2$ für kein n exakt dargestellt werden.
- Das Ergebnis elementarer Rechenoperationen $+, -, *, /$ muss nicht mehr darstellbar sein.
- Festkommazahlen werden nur in Spezialfällen verwendet, etwa um mit Geldbeträgen zu rechnen. In vielen anderen Fällen ist die im nächsten Abschnitt dargestellte Fließkommaarithmetik brauchbarer.

Fließkommaarithmetik

Vor allem in den Naturwissenschaften wird die Fließkommaarithmetik (Gleitkommaarithmetik) angewendet. Eine Zahl wird dabei repräsentiert als

$$\pm \left(a_0 + a_1\beta^{-1} + \dots + a_{n-1}\beta^{-(n-1)} \right) \times \beta^e$$

Die Ziffern a_i bilden die Mantisse und e ist der Exponent (eine ganze Zahl gegebener Länge). Wieder wird $\beta = 2$ am häufigsten verwendet. Das Vorzeichen ist ein zusätzliches Bit.

Typische Wortlängen

float: 23 Bit Mantisse, 8 Bit Exponent, 1 Bit Vorzeichen entsprechen

$$23 \cdot \log_{10} 2 = 23 \cdot \frac{\log 2}{\log 10} \approx 23 \cdot 0.3 \approx 7$$

dezimalen Nachkommastellen in der Mantisse.

double: 52 Bit Mantisse, 11 Bit Exponent, 1 Bit Vorzeichen entsprechen $52 \cdot 0.3 \approx 16$ dezimalen Nachkommastellen in der Mantisse.

Referenz: Genaueres findet man im IEEE-Standard 754 (floating point numbers).

Fehler in der Fließkommaarithmetik

Darstellungsfehler $\beta = 10, n = 3$: Die reelle Zahl 3.14159 wird auf 3.14×10^0 gerundet. Der Fehler beträgt maximal 0.005, man sagt $0.5ulp$, **ulp** heißt *units last place*.

Bemerkung:

- Wenn solche fehlerbehafteten Daten als Anfangswerte für Berechnungen verwendet werden, können die Anfangsfehler erheblich vergrößert werden.
- Durch **Rundung** können weitere Fehler hinzukommen.
- Vor allem bei der Subtraktion kann es zum Problem der **Auslöschung** kommen, wenn beinahe gleichgroße Zahlen voneinander abgezogen werden.

Beispiel: Berechne $b^2 - 4ac$ in $\beta = 10, n = 3$ für $b = 3.34, a = 1.22, c = 2.28$.
Eine exakte Rechnung liefert

$$3.34 \cdot 3.34 - 4 \cdot 1.22 \cdot 2.28 = 11.1556 - 11.1264 = 0.0292$$

Mit Rundung der Zwischenergebnisse ergibt sich dagegen

$$\dots 11.2 - 11.1 = 0.1$$

Der **absolute Fehler** ist somit $0.1 - 0.0292 = 0.0708$. Damit ist der **relative Fehler** $\frac{0.0708}{0.0292} = 240\%$! Nicht einmal eine Stelle des Ergebnisses $1.00 \cdot 10^{-1}$ ist korrekt!

Typkonversion

Im Ausdruck $5/3$ ist „/“ die ganzzahlige Division ohne Rest. Bei $5.0/3.0$ oder $5/3.0$ oder $5.0/3$ wird hingegen eine Fließkommadivision durchgeführt.

Will man eine gewisse Operation erzwingen, kann man eine explizite **Typkonversion** einbauen:

$((double) x)/3$	Fließkommadivision
$((int) y)/((int) 3)$	Ganzzahldivision

Wurzelberechnung mit dem Newtonverfahren

Problem: $f : \mathbb{R} \rightarrow \mathbb{R}$ sei eine „glatte“ Funktion, $a \in \mathbb{R}$. Wir wollen die Gleichung

$$f(x) = a$$

lösen.

Beispiel: $f(x) = x^2 \rightsquigarrow$ Berechnung von Quadratwurzeln.

Mathematik: \sqrt{a} ist die positive Lösung von $x^2 = a$.

Informatik: Will **Algorithmus** zur Berechnung des Zahlenwerts von \sqrt{a} .

Ziel: Konstruiere ein **Iterationsverfahren** mit folgender Eigenschaft: zu einem Startwert $x_0 \approx x$ finde x_1, x_2, \dots , welche die Lösung x immer besser approximieren.

Definition: (Taylorreihe)

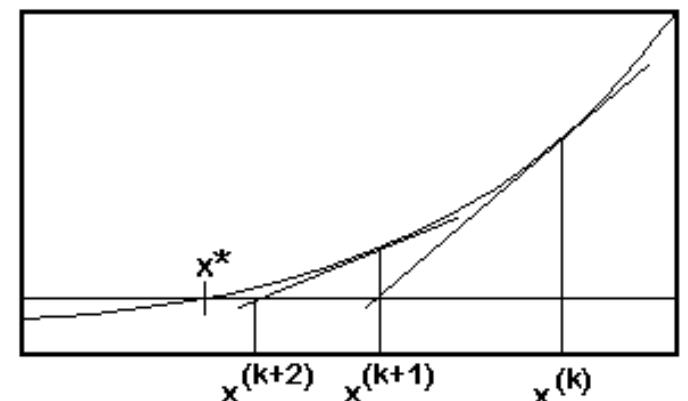
$$f(x_n + h) = f(x_n) + hf'(x_n) + \frac{h^2}{2}f''(x_n) + \dots$$

Wir vernachlässigen nun den $O(h^2)$ -Term ($|f''(x)| \leq C$, kleines h) und verlangen $f(x_n + h) \approx f(x_n) + hf'(x_n) \stackrel{!}{=} a$. Dies führt zu

$$h = \frac{a - f(x_n)}{f'(x_n)}$$

und somit zur **Iterationsvorschrift**

$$x_{n+1} = x_n + \frac{a - f(x_n)}{f'(x_n)}.$$



Beispiel: Für die Quadratwurzel erhalten wir mit $f(x) = x^2$ und $f'(x) = 2x$ die Vorschrift

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right).$$

Abbruchkriterium: $|f(x_n) - a| < \varepsilon$ für eine vorgegebene (kleine) Zahl ε .

Programm: (Quadratwurzelberechnung [newton.cc])

```
#include "fcpp.hh"

bool gut_genug( double xn, double a ) {
    return fabs( xn*xn - a ) <= 1e-15;
}

double wurzellter( double xn, double a ) {
    return cond( gut_genug( xn, a ),
                 xn,
                 wurzellter( 0.5*(xn + a/xn), a ) );
}

double wurzel( double a )
{
    return wurzellter( 1.0, a );
}
```

```
int main( int argc, char *argv[] )
{
    return
        print( wurzel( readarg_double( argc, argv, 1 ) ) );
}
```

Hier ist die Auswertung der Wurzelfunktion im Substitutionsmodell (nur die Aufrufe von `wurzelIter` sind dargestellt):

```
wurzel(2)
= wurzelIter(1,2)
= wurzelIter(1.5,2)
= wurzelIter(1.41666666666667407,2)
= wurzelIter(1.4142156862745098866,2)
= wurzelIter(1.4142135623746898698,2)
= wurzelIter(1.4142135623730951455,2)
= 1.4142135623730951455
```

Bemerkung:

- Die `print`-Funktion sorgt dafür, dass 16 Stellen bei Fließkommazahlen ausgegeben werden.
- Unter gewissen Voraussetzungen an f kann man zeigen, dass sich die Zahl der gültigen Ziffern mit jedem Schritt verdoppelt.

Funktionale Programmierung

- Auswertung von Ausdrücken
- Funktionen & Selektion
- Syntaxbeschreibung in BNF
- EBNF und kontextfreie Sprachen
- Lambda Kalkül
- Substitutionsmodell
- Linear-rekursive & -iterative Prozesse
- Baumrekursion
- Größenordnungen für Komplexitätsaussagen
- Zahlendarstellung im Rechner
- Fortgeschrittene funktionale Programmierung



Fortgeschrittene funktionale Programmierung

Funktionen in der Mathematik

Definition: Eine **Funktion** $f : X \rightarrow Y$ ordnet jedem Element einer Menge X genau ein Element der Menge Y zu.

In der Mathematik ist es nun durchaus üblich, nicht nur einfache Beispiele wie etwa $f : X \rightarrow Y$ mit $X = Y = \mathbb{R}$ zu betrachten. Im Gegenteil: in wichtigen Fällen sind X und/oder Y selbst Mengen von Funktionen.

Beispiele:

- Ableitung: Funktionen \rightarrow Funktionen; $X = C^1([a, b])$, $Y = C^0([a, b])$
- Stammfunktion: Funktionen \rightarrow Funktionen
- Integraler Mittelwert: Funktionen \rightarrow Zahlen

In C und C++ ist es ebenfalls möglich Funktionen als Argumente an Funktionen zu übergeben. Dazu stehen zwei Möglichkeiten zur Verfügung:

- Funktionszeiger (behandeln wir nicht)
- Funktoren (behandeln wir im Rahmen der Objektorientierung)

Bei kompilierten Sprachen werden alle Funktionen zur Übersetzungszeit erzeugt (aber arbeiten auf zur Laufzeit erzeugten Daten).

Interpretierte Sprachen erlauben auch die Erzeugung des Codes selbst zur Laufzeit.

Beispiel: Beispiel mit sog. Lambdaausdruck aus C++ 11:

```
#include "fcpp.hh"

typedef int (*F)( int n ); // Datentyp Funktionszeiger

int apply( F f, int arg ) // apply wendet f auf arg an
{
    return f(arg);
}

int main( int argc, char *argv[] )
{
    return print( apply( [] ( int n ) { return n+5; }, // anonyme Fkt.
                      readarg_int( argc, argv, 1 ) ) );
}
```

Warum funktionale Programmierung?

Mathematisch am besten verstanden.

Vorteilhaft, wenn Korrektheit von Programmen gezeigt werden soll.

Die funktionale Programmierung kommt mit wenigen Konzepten aus C++ aus:
Auswertung von Ausdrücken, Funktionsdefinition, cond-Funktion.

Bestimmte Probleme lassen sich mit rekursiv formulierten Algorithmen (und nur mit diesen) sehr elegant lösen.

Funktionales Programmieren ist nicht für alle Situationen die beste Wahl! Zum Beispiel legt die Interaktion mit der Außenwelt oder ihre **effiziente** Nachbildung oft andere **Paradigmen** nahe.

Ungeschickte rekursive Formulierungen können zu sehr langen Laufzeiten führen.
Geeignete Formulierung und Endrekursion können dies vermeiden.