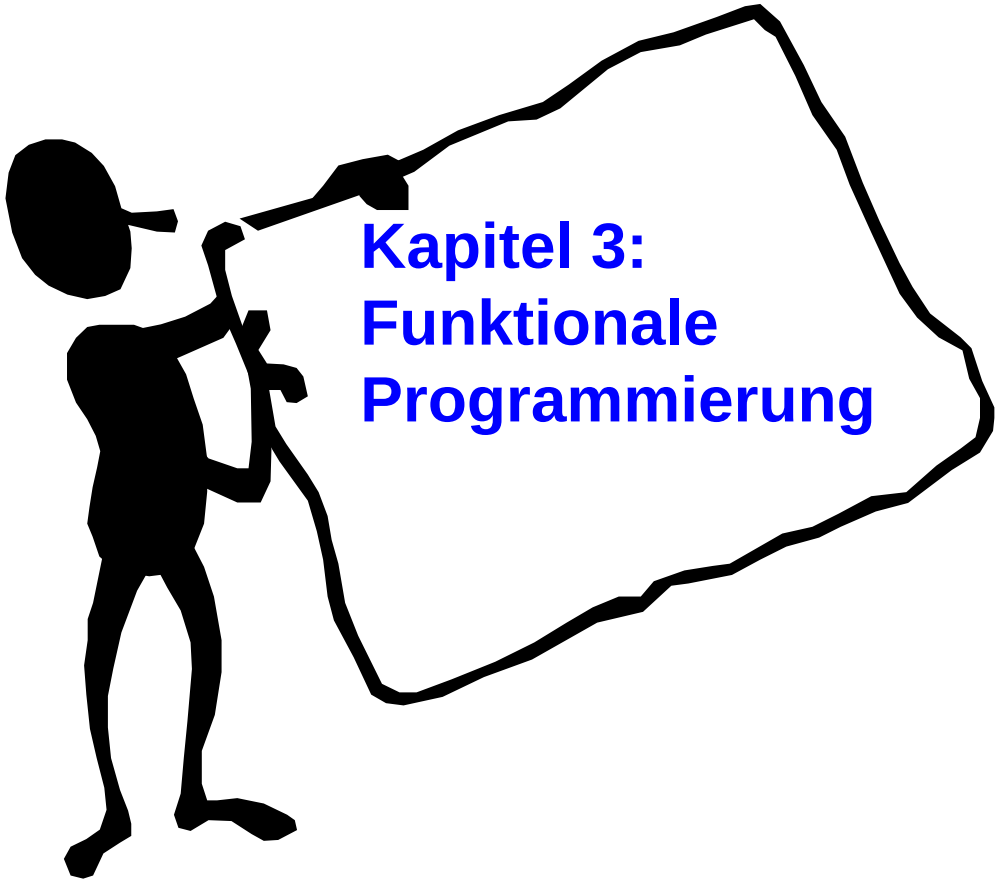
Einführung in die praktische Informatik
WS13/14

Barbara Paech, Tom-Michael Hesse

Institute of Computer Science
Im Neuenheimer Feld 326
69120 Heidelberg, Germany
http://se.ifi.uni-heidelberg.de
paech@informatik.uni-heidelberg.de

Einstieg in die Programmierung

- http://einstieg-informatik.de/



Kapitel 3: Funktionale Programmierung

3. Überblick Funktionale Programmierung

- **3.1. FC++**
- **3.2. elementare Datentypen**
- 3.3. Rekursion, Terminierung
- 3.4. Semantik

Lernziele des Kapitel 3

- Programmieren mit Funktionen
  - Eingeschränkte Notation
  - Einfache Semantik
  - Nahe an Aufgabenbeschreibung

- Wesentliche Techniken der Informatik
  - Erste Typen zur Beschreibung von Daten
  - Komposition und Fallunterscheidung als Programmiertechnik
  - Rekursion als Programmiertechnik
  - Terminierungsbeweis als Verifikationstechnik
  - Semantik eines Programmes

3. Funktionale Programmierung

- Es gibt sogenannte **Funktionale Programmiersprachen,** die i.W. nur Funktionen (höherer Ordnung) zur Programmierung zulassen
- Beispiele sind ML, Haskell, Scala

- Hier betrachten wir die **Teilmenge FC++** von C++, die nur Funktionen zulässt (um Einarbeitung in eine weitere Programmiersprache zu vermeiden).
  Nicht verwechseln mit echtem Paket
  http://cgi.di.uoa.gr/~smaragd/fc++/
- Syntax nicht so elegant wie bei echten funktionalen Programmiersprachen

3.1. FC++

- Teilmenge von C++, die nur Funktionen und Ausdrücke (keine Zuweisung !) umfasst

- Funktionales Programm
  - Definition von Daten
  - Definition von Funktionen
  - Ausdruck, der die Funktion verwendet

- Funktion
  - Kopf: Schnittstellenbeschreibung, d.h. Deklaration von Name, (Eingabe-) Parametern und des Ergebnisses
  - Rumpf: Definition des Ergebnisses als Ausdruck

3.1. Beispiel Funktion in FC++


```
int add10 (int n)

  {
    return n+10;
    }
```

Name add10,
Parameter int n,
Ergebnistyp int

Rumpf

3.1. Funktionsdefinition in FC++

- <Funktion> ::=
  <Typ> <Name< (<formale Parameter>)
  { <Funktionsrumpf> }

Achtung! Zeichen wie „{„ kommen in EBNF und FC++ vor. Als Teil von FC++ sind sie unterstrichen (und rot), da Terminale.

- <formale Parameter> ::=
  < ε> | <Typ> <Name> {, <Typ> <Name>}

- <Funktionsrumpf> ::=
  `return` <Ausdruck>; |
  `cout << ` <Ausdruck> `<< endl;`

Ergebnisrückgabe

Bildschirmausgabe

- *Vereinfacht, in C++ mehr möglich*

3.1. Ausdruck in FC++

- <Ausdruck> ::=
  <Zahl> | [-] <Name> |

  Zahlen, Parameter

  (<Ausdruck> <Operator> <Ausdruck>) |

  Operator, z.B. +,-,*,/

  <Name> ([<Ausdruck> {, <Ausdruck>}]) |

  Funktionsaufruf, z.B. f(n), auch f(g(n))

  <Cond>

  Fallunterscheidung

- *Etwas vereinfacht, in C++ mehr möglich*

3.1. Fallunterscheidung in FC++

- <Cond> ::=
  (<BoolAusdr>) ? <Ausdruck> : <Ausdruck>

Fallunterscheidung, z.B. (x==3)? x : (x+1)

- <BoolAusdr> ::=
  `true` | `false` |
  (<Ausdruck> <VglOp> <Ausdruck>) |
  (<BoolAusdr> <LogOp> <BoolAusdr>) |
  ! <BoolAusdr>

Negation

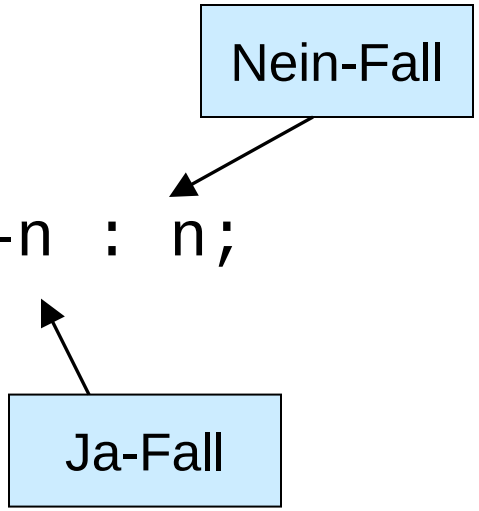
- <VglOp> ::= == | != | < | > | <= | >=
- <LogOp> ::= && | ||

And, Or

3.1. Beispiel Fallunterscheidung


```
int Betrag (int n)

  {
  return  (n <= 0) ? –n : n;
  }
```



3.1. FC++ Programm

- < einfaches Programm> ::=
  {<include> } [using namespace std;]
  {<Funktion>}+

- <include> ::= #include < <Name> >

Verwendung vordefinierter Elemente

- Typisches Beispiel

```
#include <iostream>
using namespace std;
int f (..) {..}

int main () { cout << f(..) << endl; }
```

Main-Funktion IMMER nötig

3.1. Kommentare im Programm

- *//text* einzeiliger Kommentar
- */*text*/* Kommentar über mehrere Zeilen möglich

- Programmier- und Kommentierrichtlinien kommen noch!

3.1. Komposition

- Funktionsrumpf enthält beliebige Ausdrücke insbesondere Aufrufe von anderen Funktionen

- Wie in der Mathematik:

KreisFläche: N -> N

KreisFläche(d) = $\pi * (d/2)^2$

ZylinderVolumen: N x N -> N

ZylinderVolumen(h, d) = KreisFläche(d) * h

3.1. Beispiel Komposition in FC++


```
#include <iostream>
using namespace std;

int add10 (int n)
{return n+10;
}

int betrag (int n)
{ return   (n <= 0)? -n : n;
    }

int main()
{
    cout << betrag(-45) << endl;
    cout << add10(betrag(-45)) << endl;

    return 0;
}
```

3.2. Was ist ein Datentyp?

- Ein Datentyp ist ein Wertebereich
  Beispiele für Werte: `1,“Die wilde Wutz“, true`

- Oft verbunden mit speziellen Operatoren
  - z.B. `+, *, ==, >=, …` für `int`
  - Operatoren sind Funktionen und gehören streng genommen nicht zum Datentyp

- *Alle Werte aller Typen kann man durch Zahlen (Bits & Bytes) darstellen. Dies nennt man Codierung.*

3.2. Datentyp Bool

Werte: true, false
Operatoren: ¬, ∧, ∨

! Negation in C++

	¬
true	false
false	true

&& Und in C++

∧	true	false
true	true	false
false	false	false

|| Oder in C++

∨	true	false
true	true	true
false	true	false

3.2. Boolesche Algebra

- Involutionsgesetz $\neg\neg x = x$
- true, false-Gesetze: $x \vee \neg x = true$, $\neg true = false$
- Neutralitätsgesetz $x \vee false = x$, $x \wedge true = x$
- Gesetz von de Morgan $\neg (x \wedge y) = (\neg x) \vee (\neg y)$
  $\neg (x \vee y) = (\neg x) \wedge (\neg y)$
- Idempotenzgesetz $x \wedge x = x$, $x \vee x = x$
- Kommutativgesetz $x \wedge y = y \wedge x$, $x \vee y = y \vee x$
- Assoziativgesetz $(x \wedge y) \wedge z = x \wedge (y \wedge z)$
  $(x \vee y) \vee z = x \vee (y \vee z)$
- Absorptionsgesetz $x \wedge (x \vee y) = x$
  $x \vee (x \wedge y) = x$
- Distributivgesetz $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
  $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

3.2. elementare Datentypen in FC++

- Bisher haben wir `int` und `bool` benutzt

- Beispiele für elementare Datentypen

  `int` ~ eine ganze Zahl (32 bit)

  `bool` ~ true oder false (1 bit)

  `char` ~ ein ASCII Zeichen (8 bit)

  `double` ~ Fließkommazahl (64 bit)

3.2. Datentyp String (Zeichenketten)

- C++ Darstellung von Zeichenketten (Zeichen siehe 2.2.)
- Werte char : `'a', …'?',…`
- Werte string : `"text","oder","so",""`
  - Achtung! Unterschied `"a" und 'a'`
- Operatoren
  - Konkatenation `s ° t` (in C++ komplizierter)
  - Länge `||`: in C++ `length`:
    `length("") = 0, length(s1…sn) = n`

- Zugriff auf einzelnes Zeichen
  - `"text"[0] = ,t'`
  - `"text"[1] = ,e'`
  - *Frage:* `"text"[7] =`

Zählung beginnt mit 0!

Ist nicht definiert => zufällig

3.2. String in FC++


```
#include <iostream>
#include <string>
using namespace std;

int f (string s)
{ return s.length();
}

int main ()
{cout << “text“ << endl;
 cout << f(“text“) <<endl;
return 0;
}
```

String-Paket benutzen

Operator length

Konkatenation mit unseren bisherigen Ausdrucksmitteln nicht möglich Kommt noch!

3.2. Eingabe in FC++

- <Funktionsrumpf> ::=
  `return` <Ausdruck>`;` |
  `cout << ` <Ausdruck> `<< endl;` |
  <Typ> <Name > `; cin >>` <Name>

Erweiterte Definition von Funktionsrumpf für Eingabe

- <Name> dient zur Zwischenspeicherung der Eingabe

```
int main()
{ char c;
cin >> c;
cout << c << endl;};
```

3.2. Erste Zusammenfassung FC++

- Funktionsdeklaration
- Komposition, Fallunterscheidung
- Ein/Ausgabe
- Wiederverwendung anderer Programme

- Elementare Datentypen char, bool, int, double
- Zeichenketten string

- Gute Einführung in die Algorithmenentwicklung
  - G. Pomberger, H. Dobler: Algorithmen und Datenstrukturen – Eine systematische Einführung in die Programmierung, Pearson Studium, 2008
- Gute Einführung in die Programmierung im Kleinen (in C++ und JAVA)
  - H. Helmke, F. Höppner, R. Isernhagen: Einführung in die Softwareentwicklung, Hanser, 2007
- Kompaktes C++-Buch
  - M: Meyer: C++ programmieren, Pearson Studium, 2004
- Ausführliche C++-Referenz
  - B: Stroustrup: Die C++-Programmiersprache, Addison-Wesley, 2000 (auch als Studierendenausgabe)

Kapitel 7: Beschreibungs-techniken

7. Überblick Beschreibungstechniken

- **7.1. Programmspezifikation**
- 7.2. Klassendiagramm
- 7.3. Sequenzdiagramm
- 7.4. Zusammenhängendes Beispiel

Lernziele des Kapitels 7

- Einfache Programme verständlich beschreiben
  - Programmspezifikation: Beschreibung des Ein/Ausgabeverhaltens
  - Klassendiagramme und Sequenzdiagramme: Beschreibung des Zusammenwirkens verschiedener Klassen

7. Beschreibungstechniken

- Ziel: Beschreibung der Anforderungsspezifikation oder der Entwurfsspezifikation, so dass alle Beteiligten verstehen, worum es geht.
- In der Praxis Dokumente bis zu mehreren 1000 Seiten in einer Vielzahl von Notationen.

- Wir verwenden hier ein Template zur Programmspezifikation, das Anforderungen an kleine Programmen kompakt beschreibt.

7.1. Template Programmspezifikation

- Eingaben: Typ und Name der Eingaben des Menschen
- Ausgaben: Typ und Namen der Ausgaben der Maschine auf dem Bildschirm
- Pre: Einschränkungen der Eingaben, ohne die das Programm eine falsche Ausgabe berechnet
- Post: Inhaltliche Beschreibung des Ergebnis, ohne den Algorithmus zur Berechnung vorweg zu nehmen

- Auch zur Beschreibung einzelner Funktionen verwendbar (Ein/Ausgabe dann nicht über Bildschirm)

7.1. Beispiel Programmspezifikation

- Eingabe : Zeichenkette s, Zahl l
- Ausgabe : Wahrheitswert w
- Pre: keine
- Post: w gilt genau dann, wenn |s| = l

7.1. Lösung für die Beispielspezifikation


```
#include <iostream>
#include <string>
using namespace std;
bool Laenge (string s, int l)
{
}

int main (){
string s; int l;
cout << "Eingabestring" << endl;
cin >> s;
cout << "Eingabelänge" << endl;
cin >> l;
cout << "Ausgabe"<< "   " << Laenge(s,l) << endl;
}
```

7.1. Weiteres Beispiel Programmspez.

- Eingabe : Zeichenkette s, Zeichenkette t
- Ausgabe : Zahl z
- Pre: keine
- Post: z = 0 falls |s| =|t|
  z = 1 falls |s| > |t|
  z = -1 falls |s| < |t|

7.1. Lösung weiteres Beispiel


```
#include <iostream>
#include <string>
using namespace std;
int vergleichLaenge (string s1, string s2){



}

int main ()
{
....
return 0;
}
```

7.1. weiteres Beispiel Programmspezifikation

- Eingabe : Ganze Zahlen a,b
- Ausgabe : Ganze Zahl c
- Pre: a > 0, b > 0
- Post: c ist größter gemeinsamer Teiler von a,b

- *Wie kann man das in FC++ lösen?*

Nächste Woche durch Rekursion

Kapitel 8: Qualitäts-sicherung

8. Überblick Qualitätssicherung

- **8.1. Zusicherungen**
- **8.2. Test**
- 8.3. Programmierrichtlinien
- 8.4. Debugging
- 8.5. Inspektion
- 8.6. Schnittstellenbeschreibung
- 8.7. Ausnahmebehandlung

Lernziele des Kapitels 8

- Die Qualität kleinerer Programme (unter 1000 Zeilen Code) mit typischen Techniken sicherstellen
  - Code verständlich gestalten
    - Programmierrichtlinien, Schnittstellenbeschreibung, Ausnahmebehandlung
  - Code systematisch überprüfen
    - Statisch (ohne Ablauf)
      - Inspektionen
    - Dynamisch
      - Zusicherungen, Test, Debugging

8.1. Zusicherungen

- Zusicherungen im Code ermöglichen es, zur Laufzeit Bedingungen zu überprüfen
- Falls nicht erfüllt, wird das Programm mit einer Fehlermeldung angehalten.
- Ist sinnvoll, für wesentliche Bedingungen, ohne die ein weiterer Programmablauf keinen Sinn macht.
- In C++ wird dafür das Konstrukt assert verwendet

8.1. Beispiel: Assert


```
#include <iostream>
#include <cassert>
using namespace std;

int add10 (int n)
{ assert (n >= 0);
return n+10;
}

int betrag (int n)
{ return   (n <= 0)? -n : n;
    }

int main()
{   cout << betrag(-45) << endl;
    cout << add10(betrag(-45)) <<endl;
    cout << add10(-45) << endl;
    return 0;
}
```

Hinzunahme eines Pakets

Überprüfung der Bedingung

Was passiert?

8.1. Verwendung von Assert

- Typischerweise wird Assert nur im Testbetrieb verwendet
- Kann in C++ abgeschaltet werden durch
  #define NDEBUG
  vor #include

8.1. Beispiel NDEBUG


```
#define NDEBUG

#include <iostream>
#include <cassert>
using namespace std;

int add10 (int n)
{  assert (n >= 0);
   return n+10;
}

int betrag (int n)
{  return   (n <= 0)? -n : n;
    }

int main()
{
    cout << betrag(-45) << endl;
    cout << add10(betrag(-45)) << endl;
    cout << add10(-45) << endl;
    return 0;
}
```

Macht assert unwirksam

8.2. Grundidee Testen

- Testen ermöglicht Fehler zur Laufzeit zu finden
- Grundidee: Festlegung von Eingaben und erwarteten Ergebnissen
- Testfall: Eingabewerte und erwartetes Ergebnis
- Testen kann nicht die Korrektheit eines Programmes zeigen, nur Fehler aufdecken

8.2. Testfalldefinition

- Vollständiger Test: überprüfe für alle Eingabewerte, ob Ausgabewerte korrekt
- Vollständiger Test nicht möglich, aufgrund von unendlich vielen Eingabemöglichkeiten => systematische Testfall-Auswahl nötig
- Auswahl heißt NICHT: Ausprobieren
- Geschickter Test: wähle Testfälle aus, so dass möglichst alle kritischen Fälle abgedeckt werden.

8.2. Kritische Fälle

- Ergeben sich aus der Komplexität
  - Der Eingabewerte
  - Der Abläufe
  - Der Ausgabewerte

- Blackbox-Test: Testfälle nur aufgrund der Aufgabenspezifikation ausgewählt (Wertebereiche für Ein- und Ausgabe)

- Whitebox-Test: Testfälle so ausgewählt, dass alle wichtigen Abläufe abgedeckt sind

8.2.1.Black-Box: Äquivalenzklassentest

- Äquivalenzklasse: Teilmenge der möglichen Eingabewerte
- Annahme: Programm reagiert für alle Werte aus der Äquivalenzklasse prinzipiell gleich
- Systematik:
  - Zerlege Menge der Eingabewerte in disjunkte Äquivalenzklassen (Partition)
  - Wähle Testfälle so, dass sie alle Äquivalenzklassen abdecken : mindestens ein Vertreter (Repräsentant) pro Äquivalenzklasse

8.2.1. Typische Äquivalenzklassen

- Gültige/Ungültige Wertebereiche (insbesondere bei komplexen Formaten)
  - Überprüfe auch Reaktion auf unerlaubte Eingaben!
- Grenzwerte: falls Werte in der Äquivalenzklasse geordnet, teste an jedem Rand den exakten Grenzwert, sowie die beiden benachbarten Werte (innerhalb bzw. außerhalb)
  - Man macht oft Fehler bei Grenzfällen
- Unterteilung nach verschiedenen Ausgabewerten (ggf. auch Äquivalenzklassen der Ausgabewerte)

8.2.1. Beispiel Äquivalenzklassen

- Eingabe: Ganze Zahl n
- Ausgabe: Ganze Zahl m >= 0
- pre: true;
- post: m = |n|

- Wesentlicher Unterschied bei Eingabewerten (bzgl. erwarteter Ausgabe): n<=0, n>0
  - Keine ungültigen Eingaben (bzw. Nichtzahlwerte ungültig)
  - Gültige Klasse für n: Gk1 = (..0] : Repräsentant z.B. -23
  - Gültige Klasse für n: Gk2 = (0..) : Repräsentant z.B. 95
  - Grenzwerte: Repräsentant -1,0,1
- Testfälle (-23,23), (95,95), (0,0) (-1,1), (1,1)

8.2.1. Hilfe für Algorithmus

- Äquivalenzklassen geben auch oft Anhalt für die Algorithmenentwicklung:

Was passiert bei Buchstabe als Eingabe?

```
int betrag (int n)
{return (n<=0)? -n : n
}
```

Unterscheidet die beiden Äquivalenzklassen

- Agile Softwareentwicklung (extreme programming) empfiehlt Test-First, d.h. Definition der Testfälle vor der Algorithmenentwicklung

8.2.1. Weiteres Beispiel

- Eingabe: Ganze Zahl n
- Ausgabe: Ganze Zahl m
- pre: n > 0;
- post: m = 100 / n

- Wesentlicher Unterschied (bzgl. Eingabe, aufgrund von pre): n<=0, n>0
  - Ungültige Klasse für n: Uk = (..0] : Repräsentant z.B. -17
  - Gültige Klasse für n: Gk = (0..) : Repräsentant z.B. 86
  - Grenzwert: Repräsentant -1,0,1
- Testfälle (-17,“Eingabe nicht erlaubt“), (86,1), (0,“Eingabe nicht erlaubt“) (-1,“Eingabe nicht erlaubt“), (1,100)

8.2.1. Mehrere Eingaben (1)

- Testfallableitung bei **mehreren** Eingaben:
  - Kombiniere alle gültigen Äquivalenzklassen der verschiedenen Eingaben
  - Kombiniere jede ungültige Äquivalenzklasse mit einer Kombination von gültigen Äquivalenzklassen
  - Betrachte kritische Kombinationen

8.2.1. Mehrere Eingaben (2)

- Vollständige Kombination meist zu aufwändig

- Vereinfachung:
  - Nur häufige Kombinationen
  - Nur Testfälle mit Grenzwerten
  - Nur Abdeckung paarweiser Kombinationen

- Minimal:
  - Jede gültige Klasse kommt in einem Testfall mit lauter gültigen Klassen vor
  - Und zusätzlich: Jede ungültige Klasse kommt in einem Testfall vor

8.2.1. Beispiel Mehrere Eingaben

- Eingabe: ganze Zahlen a,b;
- Ausgabe: ganze Zahl c
- pre: a >= 0, b >= 0 ;
- post: Falls a = 0 oder b = 0, c = 0
  Ansonsten c = ggT(a,b)

- Wesentlicher Unterschied (bzgl. Eingabe) : a<0 oder b<0
- Wesentlicher Unterschied (bzgl. Ausgabe) : a=0 oder b=0
  - Ungültige Klasse für a: UKa = (..0),
  - Ungültige Klasse für b: UKb = (..0)
  - Gültige Klasse für a: Ka1 = {0}, Ka2 = (0..)
  - Gültige Klasse für b: Kb1 = {0}, Kb2 = (0..)
  - Kombinationen Ka1 * Kb1, Ka1 * Kb2, Ka2 * Kb1, Ka2 * Kb2, UKa * Kb2, Ka1 * Ukb

8.2.1. Beispiel Testfälle

- Wesentlicher Unterschied (bzgl. Eingabe) : a<0,b<0
- Wesentlicher Unterschied (bzgl. Ausgabe) : a=0,b=0
  - Ungültige Klasse für a: UKa = (..0),
  - Ungültige Klasse für b: UKb = (..0)
  - Gültige Klasse für a: Ka1 = {0}, Ka2 = (0..)
  - Gültige Klasse für b: Kb1 = {0}, Kb2 = (0..)
  - Kombinationen Ka1 * Kb1, Ka1 * Kb2, Ka2 * Kb1, Ka2 * Kb2, UKa * Kb2, Ka1 * Ukb
- *Testfälle:*

8.2.1. Beispiel Vereinfachung

- Ungültige Klasse für a: UKa = (..0),
- Ungültige Klasse für b: UKb = (..0)
- Gültige Klasse für a: Ka1 = {0}, Ka2 = (0..)
- Gültige Klasse für b: Kb1 = {0}, Kb2 = (0..)
- Kombinationen Ka1 * Kb1, Ka1 * Kb2, Ka2 * Kb1, Ka2 * Kb2, UKa * Kb2, Ka1 * Ukb

- Vereinfachte Kombinationen:
  - Nur für Grenzwerte
    UKa = UKb = -1, Ka2 = Kb2 = 1
    spart nichts
- Minimale Kombinationen
  - Ka1*Kb1, Ka2*Kb2, UKa*Kb2, Ka2*UKb

8.2.1. Beispiel Vollständigkeit

- *Sind dadurch wirklich alle kritischen Fälle von ggT(a,b) abgedeckt?*

- Kritische Fälle entstehen auch durch Kombination von Eingaben
  - Ergebnis korrekt, falls a bzw. b keine echten Teiler haben?
  - Ergebnis korrekt, falls a = b?
  - Ergebnis korrekt, falls a Teiler von b oder b Teiler von a?
  - Ergebnis korrekt, falls a und b keine echten gemeinsamen Teiler haben?

8.2.1. Weiteres Beispiel Kombination

- Eingabe: ganze Zahlen x,y
- Ausgabe: ganze Zahl c >= 0
- pre: true
- post: c = |x+y|

- Kritische Kombinationen (je nachdem, ob Ergebnis positiv oder negativ):
  - Komb1: x<0, y<0 c = -(x+y)
  - Komb2: x>=0, y<0, x<|y| c = -(x+y)
  - Komb3: x >=0, y<0, x >= |y| c = x+y
  - Komb4: x<0, y>=0, y < |x| c = -(x+y)
  - Komb5: x<0, y>=0, y >= |x| c = x+y
  - Komb6: x>=0, y>=0 c = x+y

8.2.1. Beispiel Äquivalenzklassen für Funktion `finde`

- Eingabe : String a, Zeichen k,
- Ausgabe : Ganze Zahl result
- Pre: |a| > 0, Zeichen von a sind aufsteigend geordnet (z.B. „xyz“))
- Post: Index, an dem Zeichen k in a vorkommt, sonst -1
  (result >= 0) $\wedge$ a[result] = k
  $\vee$ (result = -1) $\wedge$ (forall i: 0 <= i < |a|: a[i] <> k)

- *Was ist das Ergebnis von*
  - finde (“abc“,‘c‘)
  - finde(“aba“,‘a‘)

- *Äquivalenzklassen?*

8.2.1. Testfälle für Funktion `finde`

- Testfälle:

String a	Zeichen k	Ergebnis (result)
ungeordnet	char	ungültig
“”	char	ungültig
“x“, x in char	k = ‚x‘	0
“y“, y in char	k <> ‚y‘	-1
“..x.. “,	k = ‚x‘	Position von x
“ ... “	k = ‚x‘ (‚x‘ nicht in a)	-1
“x.... “	k = ‚x‘	0
“...x“	k = ‚x‘	(Länge von a) -1

3. und 8. Erste Zusammenfassung

- Mit Funktionen kann man schon einiges machen.
- Agile Programmierung: Test-First, d.h. schreibe Testfälle vor dem Code.
- FC++ ist gar nicht so schlimm….

8. Literatur

- K. Beck: Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999

- A. Spillner, T.Linz: Basiswissen Softwaretest, dpunkt Verlag, 2002

Kapitel 2: Grundlagen der Formalisierung

2. Überblick Formale Grundlagen

- 2.1. Wdh. Mengen
- 2.2. Formale Sprache
- **2.3. Induktion**
- 2.4. Aussagenlogik
- 2.5. Relationen und Graphen

2.3. Lernziele

- Wesentliche Techniken der Informatik
  - Induktion als Verifikationstechnik

- *Wie kann man Allaussagen beweisen?*
- Z.B. „Jeder Geldbetrag von mindestens 4 cent lässt sich allein mit 2- oder 5-cent-Münzen bezahlen.

2.3. Grundidee Induktion (1)

- **Induktion**: Vom Einfachen zum Komplizierten
  - Wenn wir wissen, dass die Aussage für einen Geldbetrag x gilt, dann können wir daraus auf eine Lösung für x+1 schließen

- x liege in z 2-cent und f 5-cent Münzen vor
  x = z*2+f*5
- *Wie können wir daraus Münzen für x+1 ableiten?*
  - Falls f >= 1: dann nehme eine 5-cent Münze weg und ersetze sie durch drei 2-cent Münzen
    x+1 = (z+3)*2 + (f-1)*5
  - Falls f = 0: da x >= 4 (war Voraussetzung), ist z >= 2: dann nehme zwei 2-cent Münzen weg und ersetze sie durch eine 5-cent Münze
    x+1 = (z-2)*2 + (f+1)*5

2.3. Grundidee Induktion (2)

- Wir haben noch nicht gezeigt, dass die Aussage für alle x gilt
- Aber wir können das Argument beginnend vom Startwert x = 4 endlich oft wiederholen um einen beliebigen Geldwert darzustellen
  - x = 4:
    - nehme 2 2-cent Münzen
  - x = 5: wende das Verfahren von vorher an,
    - also 1 5-cent Münze
  - x = 6: wende das Verfahren von vorher an
    - Also 3 2-cent Münzen
  - x = 7: wende das Verfahren von vorher an
    - Also 1 2-cent Münze und 1 5-cent Münze
  - …..

2.3. Formalisierung Induktion

- Induktiver Beweis, d.h. Beweis, dass Aussage A für alle natürlichen Zahlen gilt:
  - Induktionsbasis: Zeige A(0)
  - Induktionsvoraussetzung: A(n)
  - Induktionsschritt: Zeige A(n+1) unter der Annahme, dass die Induktionsvoraussetzung gilt

- Die Gültigkeit des Induktionsprinzip lässt sich aufbauend auf dem axiomatischen Aufbau der natürlichen Zahlen nach Peano nachweisen (siehe Mathematikvorlesung)

2.3. weiteres Beispiel Induktion

- Behauptung A: für alle natürlichen Zahlen n gilt:
  0+…+n = n*(n+1)/2 (Summenformel)
- Induktionsbasis A(0): 0 = 0*1/2
- Induktionsvoraussetzung A(n), d.h.
  Es gilt 0+…+n = n*(n+1)/2
- Induktionsschritt A(n+1), d.h.
  zu zeigen ist 0+…+(n+1) = (n+1)*(n+1+1)/2
  - 0+..+(n+1) = (0+…n)+(n+1)
  - = wg. A(n): (n*(n+1)/2) +(n+1)
  - = wg. gemeinsamen Nenner: (n*(n+1)+2*(n+1)) /2
  - = wg. Ausklammerung: (n+2)*(n+1)/2
  - Also gilt A(n+1)

2.3. noch ein weiteres Beispiel Induktion

- Behauptung A: für alle natürlichen Zahlen n gilt:
  $n^2 = (2*1-1)+\ldots+(2*n-1)$
- Induktionsbasis A(0): leere Summe 0
- Induktionsvoraussetzung A(n), d.h.
  Es gilt $n^2 = (2-1)+\ldots+(2n-1)$
- Induktionsschritt A(n+1), d.h. zu zeigen ist
  $(2-1)+\ldots+(2(n+1)-1) = (n+1)^2$

2.3. noch ein weiteres Beispiel Induktion

- Behauptung A: für alle natürlichen Zahlen n gilt:
  $n^2 = (2-1)+\ldots+(2n-1)$
- Induktionsbasis A(0): leere Summe 0
- Induktionsvoraussetzung A(n), d.h.
  Es gilt $n^2 = (2-1)+\ldots+(2n-1)$
- Induktionsschritt A(n+1), d.h. zu zeigen ist
  $(2-1)+\ldots+(2(n+1)-1) = (n+1)^2$
- Beweis:
  - $(2-1)+\ldots+ (2(n+1)-1) = ((2-1)+\ldots+(2n-1))+ (2(n+1)-1)$
  - = wg. A(n): $n^2 + (2n + 2 -1) = n^2 + 2n + 1$
  - = wg. Binomischer Regel: $(n+1)^2$
  - Also gilt A(n+1)

2.3. Verallgemeinerte Induktionsvoraussetzung

- Man kann die Induktionsvoraussetzung verallgemeinern, d.h. sie gilt für alle kleineren Zahl als n:
  A(k) gilt für alle k <= n
- Beispiel: A(n): Jede natürliche Zahl n >= 2 ist ein Produkt von Primzahlen
- Basis: 2 ist eine Primzahl
- Voraussetzung: A(k) gilt für alle k <= n
- Schritt:
  - n+1 ist eine Primzahl: A(n+1) gilt
  - n+1 ist keine Primzahl,
    - also n+1 = a*b und a <= n, b <= n, da echte Teiler,
    - Wg. A(a) und A(b) gilt auch A (a*b)

2.3. Induktion für formale Sprachen

- Das Induktionsprinzip lässt sich auch für andere Mengen anwenden, die induktiv definiert sind.
- Beispiel: durch Regeln definierte Sprachen

- Z.B. sei $A = \{a,s,w\}$, $WAS^3 \subset A^*$ definiert durch
  - Basismenge: $wa \in WAS^3$
  - Erzeugungsregel: Sei $X \in A^*$: Dann
    $wX \in WAS^3 \Rightarrow wXX \in WAS^3$

- Beweis über die Anzahl der Ableitungsschritte (also der Anwendung der Erzeugungsregeln – können auch mehrere sein) für ein Wort aus dieser Menge

2.3. Beispiel: Induktion für formale Sprache

- sei $A = \{a,s,w\}$, $WAS^3 \subset A^*$ definiert durch
  - Basismenge: $wa \in WAS^3$
  - Erzeugungsregel: Sei $X \in A^*$: Dann $wX \in WAS^3 \Rightarrow wXX \in WAS^3$
- Behauptung: Für alle $X \in WAS^3$ gilt: $X \neq waaa$
- Verallgemeinert: Für alle $X \in WAS^3$ gilt: $X = wa$ oder $X = waa$ oder $|X| > 4$
- Induktionsbasis: Behauptung gilt für Basismenge, also wa
- Induktionsvoraussetzung: gilt für alle durch höchstens n-malige Anwendung der Erzeugungsregel erzeugte Worte X
- Induktionsschritt: X durch (n+1)-malige Anwendung erzeugt
  - $X = wYY$ und $wY \in WAS^3$ wurde durch weniger als n Anwendungen erzeugt
  - wg. Induktionsvoraussetzung gilt $wY = wa$ oder $wY = waa$ oder $|wY| > 4$
  - also $X = waa$ oder $X = waaaa$ oder $|X| = |wYY| > 4 + |Y|$
  - also gilt die Behauptung

2.3. Zusammenfassung

- Induktion wichtige Grundlage
- Hilft auch bei Korrektheitsüberlegungen für Programme
- Ähnlich zu Rekursion

2.3. Literatur

- Ch. Meinel, M. Mundhenk, Mathematische Grundlagen der Informatik, Teubner, 2006 (3. Auflage)