

Einführung in die Programmiertechnik

Spezifikationen, Algorithmen, Programme

Spezifikationen

- „Eine **Spezifikation** ist eine **vollständige, detaillierte** und **unzweideutige** Problembeschreibung.“
 - vollständig: alle Anforderungen und Rahmenbedingungen sind beschrieben
 - detailliert: alle zugelassenen Hilfsmittel (Basisfunktionalitäten) sind beschrieben
 - unzweideutig: Konformität einer Lösung läßt sich zweifelsfrei feststellen
- **Überspezifikation**: Spezifikation stellt Forderungen auf, die für die Lösung des eigentlichen Problems nicht erfüllt sein müssen
- **Unterspezifikation**: Spezifikation erlaubt Lösungen, die das eigentliche Problem nicht lösen

Spezifikation: Ein Beispiel

- Für beliebige Zahlen M und N berechne den größten gemeinsame Teiler $\text{ggT}(M, N)$, also die größte Zahl, die sowohl M als auch N teilt
 - Vollständigkeit: Was ist der Wertebereich M und N (etwa: nur natürliche Zahlen)? Ist 0 erlaubt?
 - Detailliertheit: Welche Operationen dürfen verwendet werden (etwa: nur $+$, $-$, oder auch div und mod)?
 - Unzweideutigkeit: Was heißt berechnen? Wie soll das Rechenergebnis zugänglich gemacht werden (etwa: ausdrucken, Rückgabewert einer Funktion, ...)?

Vorbedingungen und Nachbedingungen

- Vorbedingung: Welche Ausgangssituation darf für die Problemlösung angenommen werden?
 - Beispiel: M und N sind gegeben, natürlich, und größer 0 und kleiner 32767
- Nachbedingung: Welche Situation soll am Ende des Programms vorliegen?
 - Beispiel: Variable z enthält $\text{ggT}(M,N)$

Funktionale und Nichtfunktionale Eigenschaft

- Funktionale Eigenschaft: Welche Forderungen gibt es an den Ergebniswert?
 - Beispiel: das Programm soll den ggT ermitteln
- Nichtfunktionale Eigenschaft: Wie „gut“ soll das Ergebnis ermittelt werden?
 - Beispiel: das Programm soll nicht länger als 6s benötigen

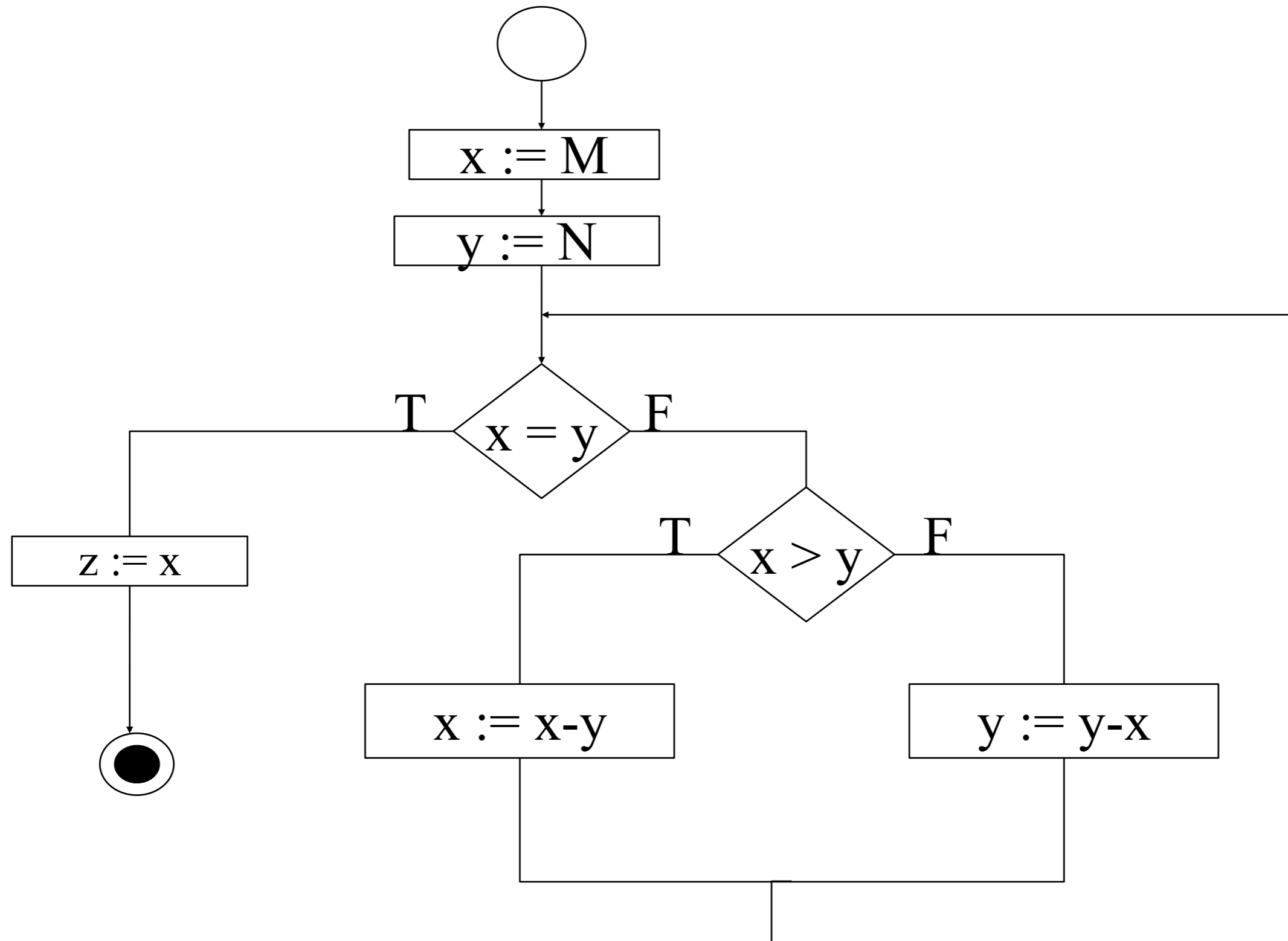
Algorithmus

- Ein Algorithmus ist eine detaillierte und explizite Vorschrift zur schrittweisen Lösung eines Problems.
 - Ausführung durch Mensch oder Maschine
- deterministischer Algorithmus: nächster Schritt eindeutig bestimmt
 - nicht-deterministischer Algorithmus: mehrere nächste Schritte möglich

Darstellung von Algorithmen

- natürliche Sprache
- Flussdiagramme
 - andere formale und informale graphische Notationen
 - Vorteil: unmittelbar verständlich
 - Nachteil: für größere Algorithmen schnell unübersichtlich
- Programmiersprachen

Flussdiagramme: ein Beispiel



Programme: Ein Beispiel

$x = M$

$y = N$

while $x \neq y$:

 if $x > y$:

$x = x - y$

 else:

$y = y - x$

$z = x$

Aufbau von Programmen

- Elementare Anweisungen
 - Zuweisung (`variable = wert`)
 - Ausgaben (`print variable`)
 - Prozedur(auf)rufe (`connect(host, port)`)
 - ...
- Kontrollstrukturen
 - Hintereinanderausführung
 - Python: untereinander folgende Zeilen
 - C, C++, Java: Hintereinanderschreiben der Anweisungen
 - Elementaranweisungen enden mit Semikolon
 - Schleifen
 - z.B. `while <Bedingung>: <aktion>`
 - Bedingte Anweisungen
 - z.B. `if <Bedingung>: <aktion> else: <aktion>`
 - Gruppierung von Anweisungen
 - Python: gleiche Einrückungstiefe

Formale Grammatiken

- Programmiersprachen unterliegen formaler Grammatik
- üblich: Lexik + Syntax
 - Lexikalische Regeln: Was ist Schlüsselwort, was sind Bezeichner, wie schreibt man Kommentare
 - Syntaxregeln: Wie gruppiert man einzelne Worte
- Definition der Grammatik oft durch (E)BNF
 - Extended Backus-Naur-Form

EBNF: Ein Beispiel

```
while_stmt ::= "while" expression ":" suite
            ["else" ":" suite]

expression ::= or_test | lambda_form

or_test    ::= and_test | or_test "or" and_test

and_test   ::= not_test | and_test "and" not_test

not_test   ::= comparison | "not" not_test

suite ::= stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT

statement ::= stmt_list NEWLINE | compound_stmt

compound_stmt ::= if_stmt
               | while_stmt
               | for_stmt
               | try_stmt
               | funcdef
               | classdef
```

Implementierung von Spezifikationen

- Spezifikation: beschreibt Problem
- Algorithmus: löst ein Problem
- einfachster Fall einer Spezifikation S : Vorbedingung P , Nachbedingung Q
- Algorithmus A löst das Problem S , wenn A , angewendet auf Eingabe mit Vorbedingung P , eine Ausgabe produziert, die Q erfüllt
 - A „implementiert“ S
 - A „löst“ S
- nicht jede Spezifikation hat eine Lösung
 - wenn es eine Lösung gibt, gibt es i.d.R. beliebig viele

Terminierung

- häufige Forderung an Algorithmus: er soll letztlich **terminieren** (also: beendet sein)
 - Ausnahmen: interaktive Programme, die prinzipiell beliebig lange laufen (Textverarbeitung, Computerspiel)
- Wird Algorithmus stets terminieren (unabhängig von der Eingabe)?
 - oft schwierig zu beantworten, weil Algorithmus Schleifen enthält
 - im Allgemeinen **unentscheidbar** (**Halteproblem**)
 - Alan Turing, 1936 (bewiesen für Turingmaschinen)

Terminierung: Ein Beispiel

- Ulam-Algorithmus

- nach Stanisław Marcin Ulam

- Beginne mit einer beliebigen Zahl n . Ist sie ungerade, multipliziere sie mit 3 und addiere 1, ansonsten halbiere sie. Fahre so fort, bis 1 erreicht ist.

```
while n > 1:
```

```
    if n % 2 == 1:
```

```
        n = 3*n+1
```

```
    else:
```

```
        n = n/2
```

Vom Algorithmus zum Programm

- Algorithmus zunächst informal gegeben
 - oft „im Kopf“
- Formalisierung als Programm
 - Festlegen von Parametern des Algorithmus
 - Python: Parameter für Prozeduren, evtl. Kommandozeilenparameter
 - Werte für Parameter evtl. auch in Programm kodiert
 - Einhalten der Programmsyntax (z.B. „Programmkopf“)
 - Python: kein Programmkopf
 - Deklaration von Variablen
 - Python: keine Deklarationen erforderlich
- Eingabe des Programms als reinen Text in Datei
 - üblich: Dateiendung deklariert die Programmiersprache
 - Python: interaktive Eingabe des Programms ist auch möglich

Ein/Ausgabe

- „interessante“ Programme konsumieren Eingaben des Nutzers und produzieren Ausgaben
- Textuelle Schnittstellen (CUI – *command-line user interface*)
 - Eingaben in Terminal über Tastatur, Ausgaben auf Terminal
- Grafische Schnittstelle (GUI – *graphical user interface*)
 - Eingabe über Maus und Tastatur
 - Flexible Eingabeelemente (Menüs, Schaltflächen, Textfelder)
 - Ausgabe als Rastergrafik
 - Flexible Verwendung von Schrift und grafischen Darstellungen

Ein/Ausgabe in Python

- **Textmodus:**
 - Ausgabe über print-Anweisung
 - print <Ausdruck>, <Ausdruck>, ...
 - Eingabe über input() und raw_input()
 - input() versteht Eingabe als Pythontext, wandelt Text z.B. in Zahlen um
 - raw_input() liefert eingegebenen Text direkt zurück
- **Grafik:**
 - GUI-Programmierung mithilfe von Bibliotheken
 - z.B. Tkinter

Datentypen

- Werte besitzen im Programm einen **Datentyp**
 - oft synonym mit „Datenstruktur“
- Ein Datentyp ist eine Menge gleichartiger Daten, auf denen eine Sammlung von Operationen definiert ist. Eine Operation ist dabei eine Verknüpfung, die einer festen Zahl von Eingabedaten ein Ergebnis zuordnet
- Stelligkeit (Stellenzahl) einer Operation: Zahl der Operanden
 - Beispiel: „+“ ist zweistellig (binär)
 - „+“: $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$
- Beispiel: Taschenrechner ist oft eingeschränkt auf einen einzigen Datentyp (Gleitkommazahl)
 - Operationen variieren zwischen verschiedenen Modellen

Datentypen in Python

- bool (ab Python 2.3)
 - Werte True, False
 - Operationen and, or, not
- int
 - Ganze Zahlen
 - Wertebereich durch Prozessorarchitektur bestimmt (üblich: 32-bit)
 - Literale (Werte im Quelltext): Dezimal (117), Oktal (führende 0, 067), Hexadezimal (führendes 0x, 0x20AC)
 - Operationen nach int: +, - (unär und binär), *, /, %, **, <<, >>
 - Operationen nach bool: ==, !=, <, <=, >, >=
- long
 - Ganze Zahlen
 - Wertebereich nicht beschränkt
 - Literale enden mit l oder L (z.B. 100000000000000L)
 - int-Operationen, die den Wertebereich von int verlassen, liefern long
- Kein Datentyp für natürliche Zahlen

Beispiel: ggT mit Integer-Division

$x = M$

$y = N$

while $(x > 0)$ and $(y > 0)$:

 if $x > y$:

$x = x \% y$

 else:

$y = y \% x$

$z = x + y$

Datentypen in Python (2)

- float

- Gleitkommazahlen
- üblicherweise repräsentiert durch 64-bit IEEE-754
- Literale: z.B. 3.1415, 113e-4

- allgemeiner:

```
floatnumber      ::=      pointfloat | exponentfloat
pointfloat      ::=      [intpart] fraction | intpart "."
exponentfloat   ::=      (intpart | pointfloat) exponent
intpart         ::=      digit+
fraction        ::=      "." digit+
exponent        ::=      ("e" | "E") ["+" | "-"] digit+
```

- Operationen wie int (außer <<, >>)
- zusätzlich: vordefinierte Funktionen
 - abs(-5.0), round(10.7)
 - Standardmodul `math` enthält weitere Operationen

Operatorüberladung

- arithmetische Operatoren arbeiten auf verschiedenen Typen
- Ergebnistyp hängt von Operandentypen ab
 - $\text{int} + \text{int} \Rightarrow (\text{int} \cup \text{long})$
 - $\text{long} + \text{long} \Rightarrow \text{long}$
 - $\text{float} + \text{float} \Rightarrow \text{float}$
- Typanpassung bei gemischten Operationen
 - „speziellerer“ Typ (Teilmenge) wird in allgemeineren Typ (Obermenge) umgewandelt
 - Beispiel: $\text{int} + \text{float} \Rightarrow \text{float}$

Datentypen in Python (3)

- **str**
 - Zeichenketten (byte strings)
 - Literale: Zeichenfolgen in Anführungszeichen
 - einfach: 'Hallo'
 - doppelt: "Hallo"
 - dreifach-einfach: `"""Ein Text, der über mehrere Zeilen geht"""`
 - dreifach-doppelt: `"""Desgleichen"""`
 - Operationen: +, *, []
 - "Hallo, " + "Welt"
 - "-=" * 10 + "-"
 - "Hallo"[2]
 - relationale Operatoren: ==, !=, <, <=, >, >=
 - Eingebaute Funktionen
 - len("String"), ord("Z"), chr(42)

Sonderzeichen in Strings

- **Anführungszeichen selbst als Teil des Strings?**
 - Lösung 1: man nehme das jeweils andere Anführungszeichen
 - Lösung 2: Spezialsyntax: `\` (Zeichen nach *Backslash* gilt nicht als Stringende)
 - *Escaping*
- **Weitere Sonderzeichen:**
 - `\n`: Zeilenende
 - `\t`: Tabulator
 - `\0`: Nullbyte
 - `\xHH`: Hexadezimal-notiertes Byte (z.B. `\xF6`)
 - `\\`: der Backslash selbst
 - ...
- **Ausnahme: *raw strings***
 - `r"c:\temp\foo.txt"`

Datentypen in Python (4)

- unicode
 - Zeichenkette (character string)
 - Literale: wie str, mit führendem u
 - z.B. u"Hallo"
 - weitere Escape-Zeichen: \u20A3, \N{MICRO SIGN}
 - Operationen wie str
 - Vordefinierte Funktionen: ord, unichr (z.B. unichr(1013))

Weitere Stringoperationen

- Stringindizierung beginnend bei 0, bis len(s)-1
- negative Indizes: Zählung von hinten beginnend
- Slicing: Auswahl von Teilstrings, durch Angabe von Start- und Endindex
 - halboffenes Intervall [Start, Ende)
 - "Hallo, Welt!"[2:5]
- Methoden: Operationen der Form <string>.<operation>
 - "Hallo".upper()
 - "Hallo, Welt!".find("Wel")
 - ...

Der Kern imperativer Sprachen

- **Variablen** in Programmiersprachen: Bindungen zwischen Namen und Wert
 - historisch: benannte Speicherplätze (Speicheradressen)
 - **Variablenbelegung**
- Typ einer Variablen: Beschränkung des Wertebereichs einer Variablen (Menge der möglichen Belegungen)
 - aus Variablentyp ergibt sich oft Zahl der benötigten Bytes sowie Interpretation der Speicherinhalts
 - Python: Variablen sind nicht typisiert (keine Wertebereichseinschränkung)
 - Jeder Wert "merkt" sich seinen eigenen Typ
- Zuweisung: Änderung der Bindung zwischen Variable und Wert
- imperative Sprachen: Bedeutung des Programms entsteht durch Folge von Zuweisungen und Ein-/Ausgabeoperationen

Zuweisungen in Python

- Elementare Zuweisung: `<variable> = <wert>`
`x = 5`
- Mehrfachzuweisung: `<variable> = <variable> = ... = <wert>`
`a = b = c = 0.0`
- Kombination von arithmetischem Operator und Zuweisung
 - `<variable> <op>= <wert>`
 - bedeutet (i.d.R.) `<variable> = <variable> <op> <wert>``x += 3`
`y *= 2`
`a /= b`
- Weitere Formen von `<variable>` später
`x[4] = y`
`person.age += 1`
`a, b = b, a`

Kontrollstrukturen

- 3 Kontrollstrukturen genügen für beliebige Algorithmen:
 - sequentielle Komposition
 - Alternativanweisung
 - while-Schleife
- "genügen für beliebige Algorithmen": Turing-vollständig
 - Church-Turing-These: Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der intuitiv berechenbaren Funktionen.

Sequentielle Komposition

- Python: Untereinanderschreiben
- Pascal: $A_1;A_2;A_3;\dots A_n$
- Semantik der sequentiellen Komposition: $A_1\dots A_n$ werden hintereinander ausgeführt
 - es sei denn, eine Ausnahme tritt mittendrin auf
- Beispielspezifikation:

Gib Wechselgeld für einen Betrag zwischen 0 und 100 Cent. Es stehen jeweils genügend Münzen im Wert von 1, 2, 5, 10, 50 Cent und 1 € zur Verfügung. Ziel ist es, mit möglichst wenig Münzen auszukommen.

Sequentielle Komposition (2)

```
rest = Betrag
k6 = rest / 100      # Zahl der Euro-Münzen
rest = rest % 100
k5 = rest / 50      # Zahl der 50-Cent-Stücke
rest = rest % 50
k4 = rest / 10      # Zahl der 10-Cent-Stücke
rest = rest % 10
k3 = rest / 5       # Zahl der 5-Cent-Stücke
rest = rest % 5
k2 = rest / 2       # Zahl der 2-Cent-Stücke
k1 = rest % 2       # Zahl der 1-Cent-Stücke
```


Sequentielle Komposition (3)

rest = Betrag

k6 = rest / 100 # Zahl der Euro-Münzen

rest %= 100

k5 = rest / 50 # Zahl der 50-Cent-Stücke

rest %= 50

k4 = rest / 10 # Zahl der 10-Cent-Stücke

rest %= 10

k3 = rest / 5 # Zahl der 5-Cent-Stücke

rest %= 5

k2 = rest / 2 # Zahl der 2-Cent-Stücke

k1 = rest % 2 # Zahl der 1-Cent-Stücke

Sequentielle Komposition (4)

```
rest = Betrag
```

```
k6, rest = divmod(rest, 100) # 1 Euro
```

```
k5, rest = divmod(rest, 50) # 50 Cent
```

```
k4, rest = divmod(rest, 10) # 10 Cent
```

```
k3, rest = divmod(rest, 5) # 5 Cent
```

```
k2, k1 = divmod(rest, 2) # 2 Cent, 1 Cent
```

Die Alternativanweisung

- Auswahl zwischen zwei Anweisungen A_1 und A_2 , in Abhängigkeit von Bedingung B
 - Python:
if B :
 A_1
else:
 A_2
- Semantik der Alternativanweisung: Zuerst wird B ausgewertet. Ist das Ergebnis wahr, wird A_1 ausgeführt, sonst A_2
 - Python: falsch sind False, 0, 0L, 0.0, 0j, None, [] (leere Liste), () (leeres Tupel), {} (leeres Dictionary); wahr ist (i.d.R.) alles andere

Die Alternativanweisung (2)

- Verschachtelte Alternativen

```
if B1:
```

```
    if B2:
```

```
        if B3:
```

```
            A1
```

```
        else:
```

```
            A2
```

```
    else:
```

```
        if B4:
```

```
            A3
```

```
        else:
```

```
            A4
```

```
    else:
```

```
        A5
```

Die *while*-Schleife

`while B:`

`A`

- Semantik der *while*-Schleife:
 1. Werte B aus
 2. Falls B erfüllt ist, führe A aus und setze dann mit 1. fort
 3. (Anderenfalls ist die *while*-Schleife beendet)
- Erweiterungen der *while*-Schleife in Python:
 - *break*-Anweisung (Abbruch der Schleife)
 - *continue*-Anweisung (Sofortiges Fortsetzen mit Schritt 1)
 - *else*-Zweig (Ausführen einer zusätzlichen Anweisung in Schritt 3)

Die *while*-Schleife (2)

- Beispiel: Berechnung der Fakultät

$n = 1$

$\text{fact} = 1$

$\text{while } n < M:$

$\text{fact} *= n$

$n += 1$

$$n! = \prod_{i=1}^n i$$