Prinzipien von Programmiersprachen
Vorlesung
Wintersemester 2007 – 2017

Johannes Waldmann, HTWK Leipzig

23. Januar 2018

Programme und Algorithmen

- Algorithmus (vgl. VL Alg. und Datenstr.)
  Vorschrift zur Lösung einer Aufgabe
- Programm (vgl. VL zu (Anwendungsorientierter) Progr.)
  Realisierung eines Algorithmus in konkreter Programmiersprache, zur Ausführung durch Maschine
- Programmiersprache
  bietet Ausdrucksmittel zur Realisierung von Algorithmen als Programme

Deutsch als Programmiersprache

§6 (2) . . . Der Zuteilungsdivisor ist so zu bestimmen, dass insgesamt so viele Sitze auf die Landeslisten entfallen, wie Sitze zu vergeben sind. Dazu wird zunächst die Gesamtzahl der Zweitstimmen aller zu berücksichtigenden Landeslisten durch die Zahl der jeweils nach Absatz 1 Satz 3 verbleibenden Sitze geteilt. Entfallen danach mehr Sitze auf die Landeslisten, als Sitze zu vergeben sind,. . .
§6 (5) Die Zahl der nach Absatz 1 Satz 3 verbleibenden Sitze wird so lange erhöht, bis jede Partei bei der zweiten Verteilung der Sitze nach Absatz 6 Satz 1 mindestens die bei der ersten Verteilung nach den Absätzen 2 und 3 für sie ermittelten zuzüglich der in den Wahlkreisen errungenen Sitze erhält, die nicht nach Absatz 4 Satz 1 von der Zahl der für die Landesliste ermittelten Sitze abgerechnet werden können.
`http://www.gesetze-im-internet.de/bwahlg/__6.html`

Beispiel: mehrsprachige Projekte

ein typisches Projekt besteht aus:

- Datenbank: SQL
- Verarbeitung: Java
- Oberfläche: HTML
- Client-Code: Java-Script

und das ist noch nicht die ganze Wahrheit:
nenne weitere Sprachen, die üblicherweise in einem solchen Projekt vorkommen

In / Into

- David Gries (1981) zugeschrieben, zitiert u.a. in McConnell: Code Complete, 2004. Unterscheide:
  - programming *in* a language
    Einschränkung des Denkens auf die (mehr oder weniger zufällig) vorhandenen Ausdrucksmittel
  - programming *into* a language
    Algorithmus → Programm
- Ludwig Wittgenstein: Die Grenzen meiner Sprache sind die Grenzen meiner Welt (sinngemäß — Ü: Original?)
- Folklore:
  A good programmer can write LISP in any language.

Sprache

- wird benutzt, um Ideen festzuhalten/zu transportieren (Wort, Satz, Text, Kontext)
- wird beschrieben durch
  - Lexik
  - Syntax
  - Semantik
  - Pragmatik
- natürliche Sprachen / formale Sprachen

Wie unterschiedlich sind Sprachen?

- weitgehend übereinstimmende Konzepte.
  - LISP (1958) = Perl = PHP = Python = Ruby = Javascript = Clojure: imperativ, (funktional),
    nicht statisch typisiert (d.h., unsicher und ineffizient)
  - Algol (1958) = Pascal = C = Java = C#
    imperativ, statisch typisiert
  - ML (1973) = Haskell:
    statisch typisiert, generische Polymorphie
- echte Unterschiede („Neuerungen“) gibt es auch
  - CSP (1977) = Occam (1983) = Go: Prozesse, Kanäle
  - Clean (1987) $\approx$ Rust (2012): Lineare Typen
  - Coq (1984) = Agda (1999) = Idris: dependent types

Konzepte

- Hierarchien (baumartige Strukturen)
  - einfache und zusammengesetzte (arithmetische, logische) Ausdrücke
  - einfache und zusammengesetzte Anweisungen (strukturierte Programme)
  - Komponenten (Klassen, Module, Pakete)
- Typen beschreiben Daten, gestatten statische Prüfung
- Namen stehen für Werte, gestatten Wiederverwendung
- flexible Wiederverwendung durch Parameter (Argumente)
  Unterprogramme: Daten, Polymorphie: Typen

Paradigmen

- imperativ
  Programm ist Folge von Befehlen (= Zuständsänderungen)
- deklarativ (Programm ist Spezifikation)
  - funktional (Gleichungssystem)
  - logisch (logische Formel über Termen)
  - Constraint (log. F. über anderen Bereichen)
- objektorientiert (klassen- oder prototyp-basiert)
- nebenläufig (nichtdeterministisch, explizite Prozesse)
- (hoch) parallel (deterministisch, implizit)

Ziele der LV

Arbeitsweise: Methoden, Konzepte, Paradigmen

- isoliert beschreiben
- an Beispielen in (bekannten und unbekannten) Sprachen wiedererkennen

Ziel:

- verbessert die Organisation des vorhandenen Wissens
- gestattet die Beurteilung und das Erlernen neuer Sprachen
- hilft bei Entwurf eigener (anwendungsspezifischer) Sprachen

Beziehungen zu anderen LV

- Grundlagen der Informatik, der Programmierung: strukturierte (imperative) Programmierung
- Softwaretechnik 1/2: objektorientierte Modellierung und Programmierung, funktionale Programmierung und OO-Entwurfsmuster
- Compilerbau: Implementierung von Syntax und Semantik

Sprachen für bestimmte Anwendungen, mit bestimmten Paradigmen:

- Datenbanken, Computergrafik, künstliche Intelligenz, Web-Programmierung, parallele/nebenläufige Programmierung

Organisation

- Vorlesung
- Übungen (alle in Z430)
  Online-Übungsaufgaben (Übungsgruppe wählen)
  `https://autotool.imn.htwk-leipzig.de/new/vorlesung/245/aufgaben`
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Klausur: 120 min, ohne Hilfsmittel

Literatur

- http://www.imn.htwk-leipzig.de/~waldmann/edu/ws17/pps/folien/
- Robert W. Sebesta: Concepts of Programming Languages, Addison-Wesley 2004, ...

Zum Vergleich/als Hintergrund:

- Abelson, Sussman, Sussman: Structure and Interpretation of Computer Programs, MIT Press 1984
  http://mitpress.mit.edu/sicp/
- Turbak, Gifford: Design Concepts of Programming Languages, MIT Press 2008
  http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11656

Inhalt

(nach Sebesta: Concepts of Programming Languages)

- Methoden: (3) Beschreibung von Syntax und Semantik
- Konzepte:
  - (5) Namen, Bindungen, Sichtbarkeiten
  - (6) Typen von Daten, Typen von Bezeichnern
  - (7) Ausdrücke und Zuweisungen, (8) Anweisungen und Ablaufsteuerung, (9) Unterprogramme
- Paradigmen:
  - (12) Objektorientierung ((11) Abstrakte Datentypen)
  - (15) Funktionale Programmierung

Übungen

1. Anwendungsgebiete von Programmiersprachen, wesentliche Vertreter
zu Skriptsprachen: finde die Anzahl der `"*.java"`-Dateien unter `$HOME/workspace`, die den Bezeichner `String` enthalten. (Benutze eine Pipe aus drei Unix-Kommandos.)
Lösungen:

```
find workspace/ -name "*.java" | xargs grep -l Stri
find workspace/ -name "*.java"   -exec grep -l Stri
```

2. Maschinenmodelle (Bsp: Register, Turing, Stack, Funktion)
funktionales Programmieren in Haskell
(`http://www.haskell.org/`)

```
ghci
:set +t
length $ takeWhile (== '0') $ reverse $ show $ prod
```

Kellermaschine in PostScript.

```
42 42 scale 7 9 translate .07 setlinewidth .5 setgr
setgray}def 1 0 0 42 1 0 c 0 1 1{0 3 3 90 270 arc 0
```

Übung: Beispiele für Übersetzer

Java:

```
javac Foo.java # erzeugt Bytecode (Foo.class)
java Foo       # führt Bytecode aus (JVM)
```

Einzelheiten der Übersetzung:

```
javap -c Foo   # druckt Bytecode
```

C:

```
gcc -c bar.c   # erzeugt Objekt(Maschinen)code (bar
gcc -o bar bar.o # linkt (lädt) Objektcode (Resulta
./bar          # führt gelinktes Programm aus
```

Einzelheiten:

```
gcc -S bar.c # erzeugt Assemblercode (bar.s)
```

Aufgaben:

- geschachtelte arithmetische Ausdrücke in Java und C: vergleiche Bytecode mit Assemblercode
- vergleiche Assemblercode für Intel und Sparc (einloggen

Programme als Bäume

- ein Programmtext repräsentiert eine Hierarchie (einen Baum) von Teilprogrammen
- Die Semantik des Programmes wird durch Induktion über diesen Baum definiert.
- In den Blättern des Baums stehen Token,
- jedes Token hat einen Typ und einen Inhalt (eine Zeichenkette).
- dieses Prinzip kommt aus der Mathematik (arithmetische Ausdrücke, logische Formeln — sind Bäume)

Token-Typen

- reservierte Wörter (if, while, class, ...)
- Bezeichner (foo, bar, ...)
- Literale für ganze Zahlen, Gleitkommazahlen, Strings, Zeichen, ...
- Trenn- und Schlußzeichen (Komma, Semikolon)
- Klammern (runde: paren(these)s, eckige: brackets, geschweifte: braces, spitze: angle brackets)
- Operatoren (`=`, `+`, `&&`, ...)
- Leerzeichen, Kommentare (whitespace)

alle Token eines Typs bilden eine *formale Sprache*.

Formale Sprachen

- ein *Alphabet* ist eine Menge von Zeichen,
- ein *Wort* ist eine Folge von Zeichen,
- eine *formale Sprache* ist eine Menge von Wörtern.

Beispiele:

- Alphabet $\Sigma = \{a, b\}$,
- Wort $w = ababaaab$,
- Sprache $L =$ Menge aller Wörter über Σ gerader Länge.
- Sprache (Menge) aller Gleitkomma-Literale in C.

Spezifikation formaler Sprachen

man kann eine formale Sprache beschreiben:

- *algebraisch* (Sprach-Operationen)
  Bsp: reguläre Ausdrücke
- *generativ* (Grammatik), Bsp: kontextfreie Grammatik,
- durch *Akzeptanz* (Automat), Bsp: Kellerautomat,
- *logisch* (Eigenschaften),

$$\left\{ w \mid \forall p, r : \left(\begin{array}{ll} & (p < r \wedge w[p] = a \wedge w[r] = c) \\ \Rightarrow & \exists q : (p < q \wedge q < r \wedge w[q] = b) \end{array} \right) \right\}$$

Sprach-Operationen

Aus Sprachen L_1, L_2 konstruiere:

- Mengenoperationen
  - Vereinigung $L_1 \cup L_2$,
  - Durchschnitt $L_1 \cap L_2$, Differenz $L_1 \backslash L_2$;
- Verkettung $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}$
- Stern (iterierte Verkettung) $L_1^* = \bigcup_{k \geqslant 0} L_1^k$

Def: Sprache *regulär* : $\Longleftrightarrow$ kann durch diese Operationen aus endlichen Sprachen konstruiert werden.
Satz: Durchschnitt und Differenz braucht man dabei nicht.

Reguläre Sprachen/Ausdrücke

Die Menge $E(\Sigma)$ der *regulären Ausdrücke*
über einem Alphabet (Buchstabenmenge) Σ
ist die kleinste Menge E, für die gilt:

- für jeden Buchstaben $x \in \Sigma : x \in E$
  (autotool: Ziffern oder Kleinbuchstaben)
- das leere Wort $\epsilon \in E$ (autotool: `Eps`)
- die leere Menge $\varnothing \in E$ (autotool: `Empty`)
- wenn $A, B \in E$, dann
  - (Verkettung) $A \cdot B \in E$ (autotool: $*$ oder weglassen)
  - (Vereinigung) $A + B \in E$ (autotool: $+$)
  - (Stern, Hülle) $A^* \in E$ (autotool: `^*`)

Jeder solche Ausdruck beschreibt eine *reguläre Sprache*.

Beispiele/Aufgaben zu regulären Ausdrücken

Wir fixieren das Alphabet $\Sigma = \{a, b\}$.

- alle Wörter, die mit a beginnen und mit b enden: $a\Sigma^* b$.
- alle Wörter, die wenigstens drei a enthalten $\Sigma^* a \Sigma^* a \Sigma^* a \Sigma^*$
- alle Wörter mit gerade vielen a und beliebig vielen b?
- Alle Wörter, die ein aa oder ein bb enthalten: $\Sigma^*(aa \cup bb)\Sigma^*$
- (Wie lautet das Komplement dieser Sprache?)

Erweiterte reguläre Ausdrücke

1. zusätzliche Operatoren (Durchschnitt, Differenz, Potenz), die trotzdem nur reguläre Sprachen erzeugen
   Beispiel: $\Sigma^* \setminus (\Sigma^* ab \Sigma^*)^2$
2. zusätzliche nicht-reguläre Operatoren

   Beispiel: exakte Wiederholungen $L^{\boxed{k}} := \{w^k \mid w \in L\}$
   beachte Unterschied zu L^k
3. Markierung von Teilwörtern, definiert (evtl. nicht-reguläre) Menge von Wörtern mit Positionen darin

wenn nicht-reguläre Sprachen entstehen können, ist keine effiziente Verarbeitung (mit endlichen Automaten) möglich.
auch reguläre Operatoren werden gern schlecht implementiert (`http://swtch.com/~rsc/regexp/regexp1.html`)

Bemerkung zu Reg. Ausdr.

Wie beweist man $w \in \mathrm{L}(X)$?
(Wort w gehört zur Sprache eines regulären Ausdrucks X)

- wenn $X = X_1 + X_2$:
  beweise $w \in \mathrm{L}(X_1)$ *oder* beweise $w \in \mathrm{L}(X_2)$
- wenn $X = X_1 \cdot X_2$:
  zerlege $w = w_1 \cdot w_2$ *und* beweise $w_1 \in \mathrm{L}(X_1)$ *und* beweise $w_2 \in \mathrm{L}(X_2)$.
- wenn $X = X_1^*$:
  wähle einen Exponenten $k \in \mathbb{N}$ *und* beweise $w \in \mathrm{L}(X_1^k)$
  (nach vorigem Schema)

Beispiel: $w = abba, X = (ab^*)^*$.
$w = abb \cdot a = ab^2 \cdot ab^0 \in ab^* \cdot ab^* \subseteq (ab^*)^2 \subseteq (ab^*)^*$.

Übungen Reg. Ausdr.

- $(\Sigma^*, \cdot, \epsilon)$ ist Monoid
- . . . aber keine Gruppe, weil man im Allgemeinen nicht dividieren kann. Welche Relation ergibt sich als „Teilbarkeit“: $u \mid w := \exists v : u \cdot v = w$
- Zeichne Hasse-Diagramme der Teilbarkeitsrelation
  - auf natürlichen Zahlen $\{0, 1, \ldots, 10\}$,
  - auf Wörtern $\{a, b\}^{\leqslant 2}$
- $(\mathrm{Pow}(\Sigma^*), \cup, \cdot, \ldots, \ldots)$ ist Halbring.
  Beispiel für Distributivgesetz?
  Welches sind jeweils die neutralen Elemente der Operationen?
  (vgl. oben) Welche Relation auf Sprachen (Mengen) ergibt sich als „Teilbarkeit“ bzgl. $\cup$?
- Damit $a^{b+c} = a^b \cdot a^c$ immer gilt, muß man a^0 wie definieren?
- Block-Kommentare und weitere autotool-Aufgaben
- reguläre Ausdrücke für Tokenklassen in der Standard-Pascal-Definition
  http://www.standardpascal.org/iso7185.html#

Wort-Ersetzungs-Systeme

Berechnungs-Modell (Markov-Algorithmen)

- Zustand (Speicherinhalt): Zeichenfolge (Wort)
- Schritt: Ersetzung eines Teilwortes

Regelmenge $R \subseteq \Sigma^* \times \Sigma^*$
Regel-Anwendung:
$u \rightarrow_R v \iff \exists x, z \in \Sigma^*, (l, r) \in R : u = x \cdot l \cdot z \wedge x \cdot r \cdot z = v.$
Beispiel: Bubble-Sort: $\{ba \rightarrow ab, ca \rightarrow ac, cb \rightarrow bc\}$
Beispiel: Potenzieren: $ab \rightarrow bba$
Aufgaben: gibt es unendlich lange Rechnungen für:
$R_1 = \{1000 \rightarrow 0001110\}, R_2 = \{aabb \rightarrow bbbaaa\}$?

Grammatiken

Grammatik G besteht aus:

- Terminal-Alphabet Σ (üblich: Kleinbuchst., Ziffern)
- Variablen-Alphabet V (üblich: Großbuchstaben)
- Startsymbol $S \in V$
- Regelmenge (Wort-Ersetzungs-System) $R \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$

```
Grammatik
  { terminale
      = mkSet "abc"
  , variablen
      = mkSet "SA"
  , start = 'S'
  , regeln = mkSet
      [ ("S", "abc")
      , ("ab", "aabbA")
      , ("Ab", "bA")
      , ("Ac", "cc")
      ]
  }
```

von G erzeugte Sprache: $L(G) = \{w \mid S \rightarrow_R^* w \wedge w \in \Sigma^*\}$.

Formale Sprachen: Chomsky-Hierarchie

- (Typ 0) aufzählbare Sprachen (beliebige Grammatiken, Turingmaschinen)
- (Typ 1) kontextsensitive Sprachen (monotone Grammatiken, linear beschränkte Automaten)
- (Typ 2) kontextfreie Sprachen (kontextfreie Grammatiken, Kellerautomaten)
- (Typ 3) reguläre Sprachen (rechtslineare Grammatiken, reguläre Ausdrücke, endliche Automaten)

Tokenklassen sind meist reguläre Sprachen.
Programmiersprachen werden kontextfrei beschrieben (mit Zusatzbedingungen).

Typ-3-Grammatiken

(= rechtslineare Grammatiken)
jede Regel hat die Form

- Variable $\rightarrow$ Terminal Variable
- Variable $\rightarrow$ Terminal
- Variable $\rightarrow \epsilon$

(vgl. lineares Gleichungssystem)
Beispiele

- $G_1 = (\{a, b\}, \{S, T\}, S, \{S \rightarrow \epsilon, S \rightarrow aT, T \rightarrow bS\})$
- $G_2 = (\{a, b\}, \{S, T\}, S, \{S \rightarrow \epsilon, S \rightarrow aS, S \rightarrow bT, T \rightarrow aT, T \rightarrow bS\})$

Sätze über reguläre Sprachen

Für jede Sprache L sind die folgenden Aussagen äquivalent:

- es gibt einen regulären Ausdruck X mit $L = \mathrm{L}(X)$,
- es gibt eine Typ-3-Grammatik G mit $L = \mathrm{L}(G)$,
- es gibt einen endlichen Automaten A mit $L = \mathrm{L}(A)$.

Beweispläne:

- Grammatik $\leftrightarrow$ Automat (Variable $=$ Zustand)
- Ausdruck $\rightarrow$ Automat (Teilbaum $=$ Zustand)
- Automat $\rightarrow$ Ausdruck (dynamische Programmierung)
  $L_A(p, q, r) =$ alle Pfade von p nach r über Zustände $\leqslant q$.

Kontextfreie Sprachen

Def (Wdhlg): G ist kontextfrei (Typ-2), falls $\forall(l, r) \in R(G) : l \in V$.
geeignet zur Beschreibung von Sprachen mit hierarchischer Struktur.

```
Anweisung -> Bezeichner = Ausdruck
    | if Ausdruck then Anweisung else Anweisung
Ausdruck -> Bezeichner | Literal
    | Ausdruck Operator Ausdruck
```

Bsp: korrekt geklammerte Ausdrücke:
$G = (\{a, b\}, \{S\}, S, \{S \to aSbS, S \to \epsilon\})$.
Bsp: Palindrome:
$G = (\{a, b\}, \{S\}, S, \{S \to aSa, S \to bSb, S \to \epsilon)$.
Bsp: alle Wörter w über $\Sigma = \{a, b\}$ mit $|w|_a = |w|_b$

Klammer-Sprachen

Abstraktion von vollständig geklammerten Ausdrücke mit zweistelligen Operatoren

`(4*(5+6)-(7+8))` $\Rightarrow$ `(()())` $\Rightarrow$ $aababb$

Höhendifferenz: $h : \{a, b\}^* \to \mathbb{Z} : w \mapsto |w|_a - |w|_b$

Präfix-Relation: $u \leqslant w :\iff \exists v : u \cdot v = w$

Dyck-Sprache: $D = \{w \mid h(w) = 0 \wedge \forall u \leqslant w : h(u) \geqslant 0\}$

CF-Grammatik: $G = (\{a, b\}, \{S\}, S, \{S \to \epsilon, S \to aSbS\})$

Satz: $L(G) = D$. Beweis (Plan):

$L(G) \subseteq D$ Induktion über Länge der Ableitung

Übungen

- Beispiele Wort-Ersetzung ($ab \rightarrow baa$, usw.)
- Dyck-Sprache: Beweis $L(G) \subseteq D$
  (Induktionsbehauptung? Induktionsschritt?)
- Dyck-Sprache: Beweis $D \subseteq L(G)$
- CF-Grammatik für $\{w \mid w \in \{a, b\}^*, |w|_a = |w|_b\}$
- CF-Grammatik für $\{w \mid w \in \{a, b\}^*, 2 \cdot |w|_a = |w|_b\}$

(erweiterte) Backus-Naur-Form

- Noam Chomsky: Struktur natürlicher Sprachen (1956)
- John Backus, Peter Naur: Definition der Syntax von Algol (1958)

Backus-Naur-Form (BNF) $\approx$ kontextfreie Grammatik

```
<assignment> -> <variable> = <expression>
<number> -> <digit> <number> | <digit>
```

Erweiterte BNF

- Wiederholungen (Stern, Plus) `<digit>^+`
- Auslassungen

```
if <expr> then <stmt> [ else <stmt> ]
```

kann in BNF übersetzt werden

Ableitungsbäume für CF-Sprachen

Def: ein geordneter Baum T mit Markierung $m : T \to \Sigma \cup \{\epsilon\} \cup V$ ist Ableitungsbaum für eine CF-Grammatik G, wenn:

- für jeden inneren Knoten k von T gilt $m(k) \in V$
- für jedes Blatt b von T gilt $m(b) \in \Sigma \cup \{\epsilon\}$
- für die Wurzel w von T gilt $m(w) = S(G)$ (Startsymbol)
- für jeden inneren Knoten k von T mit Kindern $k_1, k_2, \ldots, k_n$ gilt $(m(k), m(k_1)m(k_2)\ldots m(k_n)) \in R(G)$ (d. h. jedes $m(k_i) \in V \cup \Sigma$)
- für jeden inneren Knoten k von T mit einzigem Kind $k_1 = \epsilon$ gilt $(m(k), \epsilon) \in R(G)$.

Ableitungsbäume (II)

Def: der *Rand* eines geordneten, markierten Baumes (T, m) ist die Folge aller Blatt-Markierungen (von links nach rechts).
Beachte: die Blatt-Markierungen sind $\in \{\epsilon\} \cup \Sigma$, d. h. Terminalwörter der Länge 0 oder 1.
Für Blätter: $\text{rand}(b) = m(b)$, für innere Knoten:
$\text{rand}(k) = \text{rand}(k_1)\,\text{rand}(k_2)\ldots\text{rand}(k_n)$
Satz: $w \in L(G) \iff$ existiert Ableitungsbaum (T, m) für G mit $\text{rand}(T, m) = w$.

Eindeutigkeit

Def: G heißt *eindeutig*, falls $\forall w \in L(G)$ *genau ein* Ableitungsbaum (T, m) existiert.
Bsp: ist $\{S \to aSb|SS|\epsilon\}$ eindeutig?
(beachte: mehrere Ableitungen $S \to_R^* w$ sind erlaubt, und wg. Kontextfreiheit auch gar nicht zu vermeiden.)

Die naheliegende Grammatik für arith. Ausdr.

```
expr -> number | expr + expr | expr * expr
```

ist mehrdeutig (aus *zwei* Gründen!)
Auswege:

- Transformation zu eindeutiger Grammatik (benutzt zusätzliche Variablen)
- Operator-Assoziativitäten und -Präzedenzen

Assoziativität

- Definition: Operation ist *assoziativ*
- Bsp: Plus ist nicht assoziativ (für Gleitkommazahlen) (Ü)
- für nicht assoziativen Operator $\odot$ muß man festlegen, was $x \odot y \odot z$ bedeuten soll:

$$(3 + 2) + 4 \stackrel{?}{=} 3 + 2 + 4 \stackrel{?}{=} 3 + (2 + 4)$$

$$(3 - 2) - 4 \stackrel{?}{=} 3 - 2 - 4 \stackrel{?}{=} 3 - (2 - 4)$$

$$(3 * *2) * *4 \stackrel{?}{=} 3 * *2 * *4 \stackrel{?}{=} 3 * *(2 * *4)$$

- . . . und dann die Grammatik entsprechend einrichten

Assoziativität (II)

`X1 + X2 + X3` auffassen als `(X1 + X2) + X3`

Grammatik-Regeln

```
Ausdruck -> Zahl | Ausdruck + Ausdruck
```

ersetzen durch

```
Ausdruck -> Summe
Summe    -> Summand | Summe + Summand
Summand  -> Zahl
```

Präzedenzen

$$(3 + 2) * 4 \stackrel{?}{=} 3 + 2 * 4 \stackrel{?}{=} 3 + (2 * 4)$$

Grammatik-Regel

```
summand -> zahl
```

erweitern zu

```
summand -> zahl | produkt
produkt -> ...
```

(Assoziativität beachten)

Zusammenfassung Operator/Grammatik

Ziele:

- Klammern einsparen
- trotzdem eindeutig bestimmter Syntaxbaum

Festlegung:

- Assoziativität:
  bei Kombination eines Operators mit sich
- Präzedenz:
  bei Kombination verschiedener Operatoren

Realisierung in CFG:

- Links/Rechts-Assoziativität $\Rightarrow$ Links/Rechts-Rekursion
- verschiedene Präzedenzen $\Rightarrow$ verschiedene Variablen

Übung Operator/Grammatik

Übung:

- Verhältnis von plus zu minus, mal zu durch?
- Klammern?
- unäre Operatoren (Präfix/Postfix)?

Das hängende *else*

naheliegende EBNF-Regel für Verzweigungen:

```
<statement> -> if <expression>
    then <statement> [ else <statement> ]
```

führt zu einer mehrdeutigen Grammatik.
Dieser Satz hat zwei Ableitungsbäume:

```
if X1 then if X2 then S1 else S2
```

- Festlegung: das „in der Luft hängende“ (dangling) `else` gehört immer zum letzten verfügbaren `then`.
- Realisierung durch Grammatik mit (Hilfs-)Variablen

  `<statement>, <statement-no-short-if>`

Statische und dynamische Semantik

Semantik = Bedeutung

- statisch (kann zur Übersetzungszeit geprüft werden)
  Beispiele:
  - Typ-Korrektheit von Ausdrücken,
  - Bedeutung (Bindung) von Bezeichnern

  Hilfsmittel: Attributgrammatiken
- dynamisch (beschreibt Ausführung des Programms)
  operational, axiomatisch, denotational

Attributgrammatiken (I)

- Attribut: Annotation an Knoten des Syntaxbaums.
  A : Knotenmenge $\rightarrow$ Attributwerte (Bsp: $\mathbb{N}$)
- Attributgrammatik besteht aus:
  - kontextfreier Grammatik G (Bsp: $\{S \rightarrow e \mid mSS\}$)
  - für jeden Knotentyp (Terminal + Regel)
    eine Menge (Relation) von erlaubten Attribut-Tupeln
    $(A(X_0), A(X_1), \ldots, A(X_n))$
    für Knoten X_0 mit Kindern $[X_1, \ldots, X_n]$

    $S \rightarrow mSS$, $A(X_0) + A(X_3) = A(X_2)$;
    $S \rightarrow e$, $A(X_0) = A(X_1)$;
    Terminale: $A(e) = 1, A(m) = 0$

Attributgrammatiken (II)

ein Ableitungsbaum mit Annotationen ist
korrekt bezüglich einer Attributgrammatik, wenn

- zur zugrundeliegenden CF-Grammatik paßt
- in jedem Knoten das Attribut-Tupel (von Knoten und Kindern) zur erlaubten Tupelmenge gehört

Plan:

- Baum beschreibt Syntax, Attribute beschreiben Semantik

Ursprung: Donald Knuth: Semantics of Context-Free Languages, (Math. Systems Theory 2, 1968)
technische Schwierigkeit: Attributwerte effizient bestimmen.
(beachte: (zirkuläre) Abhängigkeiten)

Donald E. Knuth

- The Art Of Computer Programming (1968, ...)
  (Band 3: Sortieren und Suchen)
- TEX, Metafont, Literate Programming (1983, ...)
  (Leslie Lamport: LATEX)
- Attribut-Grammatiken (1968)
- Anwendung der Landau-Notation ($O(f)$, Analysis) und Erweiterung (Ω, Θ) für asymptotische Komplexität
- ...

`http://www-cs-faculty.stanford.edu/~uno/`

Arten von Attributen

- synthetisiert:
  hängt nur von Attributwerten in Kindknoten ab
- ererbt (inherited)
  hängt nur von Attributwerten in Elternknoten und (linken) Geschwisterknoten ab

Wenn Abhängigkeiten bekannt sind, kann man Attributwerte durch Werkzeuge bestimmen lassen.

Attributgrammatiken–Beispiele

- Auswertung arithmetischer Ausdrücke (dynamisch)
- Bestimmung des abstrakten Syntaxbaumes
- Typprüfung (statisch)
- Kompilation (für Kellermaschine) (statisch)

Konkrete und abstrakte Syntax

- konkreter Syntaxbaum = der Ableitungsbaum
- abstrakter Syntaxbaum = wesentliche Teile des konkreten Baumes

unwesentlich sind z. B. die Knoten, die zu Hilfsvariablen der Grammatik gehören.
abstrakter Syntaxbaum kann als synthetisiertes Attribut konstruiert werden.

```
E -> E + P   ;   E.abs = new Plus(E.abs, P.abs)
E -> P       ;   E.abs = P.abs
```

Regeln zur Typprüfung

. . . bei geschachtelten Funktionsaufrufen

- Funktion f hat Typ $A \rightarrow B$
- Ausdruck X hat Typ A
- dann hat Ausdruck $f(X)$ den Typ B

Beispiel

```
class C {
  static class A {}  static class B {}
  static B f (A y) { .. }
  static A g (B x) { .. }
  ..
  .. C.g (C.f (new C.A()))  .. }
```

Übung Attributgrammatiken/SableCC

- SableCC: `http://sablecc.org/`
  SableCC is a parser generator for building compilers, interpreters . . . , strictly-typed abstract syntax trees and tree walkers
- Syntax einer Regel

```
linke-seite { -> attribut-typ }
    = { zweig-name } rechte-seite { ->  attribut-
```

- Quelltexte:

```
git clone https://gitlab.imn.htwk-leipzig.de/wal
```

  Benutzung:

```
cd pps-ws15/rechner ; make ; make test ; make cl
```

  (dafür muß `sablecc` gefunden werden, siehe `http://www.imn.htwk-leipzig.de/~waldmann/etc/pool/`)
- Struktur:
  - `rechner.grammar` enthält Attributgrammatik, diese beschreibt die Konstruktion des *abstrakten Syntaxbaumes (AST)* aus dem Ableitungsbaum (konkreten Syntaxbaum)
  - `Eval.java` enthält Besucherobjekt, dieses beschreibt die

Dynamische Semantik

- operational:
  beschreibt Wirkung von Anweisungen durch Änderung des Speicherbelegung
- denotational:
  ordnet jedem (Teil-)Programm einen Wert zu, Bsp: eine Funktion (höherer Ordnung).
  Beweis von Programmeigenschaften durch Term-Umformungen
- axiomatisch (Bsp: Hoare-Kalkül):
  enthält Schlußregeln, um Aussagen über Programme zu beweisen

Bsp. Operationale Semantik (I)

arithmetischer Ausdruck $\Rightarrow$ Programm für Kellermaschine
$3 * x + 1 \Rightarrow$ push 3, push x, mal, push 1, plus

- Code für Konstante/Variable c : `push c;`
- Code für Ausdruck $x \circ y$: `code(x); code(y); o;`
- Ausführung eines binären Operators $\circ$:
  `x <- pop; y <- pop; push (x o y);`

Der erzeugte Code ist synthetisiertes Attribut!
Beispiele: Java-Bytecode (javac, javap),
CIL (gmcs, monodis)
Bemerkung: soweit scheint alles trivial—interessant wird es bei Teilausdrücken mit Nebenwirkungen, Bsp. `x++ - --x;`

Bsp: Operationale Semantik (II)

Schleife

```
while (B) A
```

wird übersetzt in Sprungbefehle

```
    if (B) ...
```

(vervollständige!)
Aufgabe: übersetze `for(A; B; C) D` in while!

Denotationale Semantik

Beispiele

- jedes (nebenwirkungsfreie) *Unterprogramm* ist eine Funktion von Argument nach Resultat
- jede *Anweisung* ist eine Funktion von Speicherbelegung nach Speicherbelegung

Vorteile denotationaler Semantik:

- Bedeutung eines Programmes $=$ mathematisches Objekt
- durch Term beschreiben, durch äquivalente Umformungen verarbeiten (equational reasoning)

Vorteil deklarativer Programierung:
Programmiersprache *ist* Beschreibungssprache

Beispiele Denotationale Semantik

- jeder arithmetische Ausdruck (aus Konstanten und Operatoren)
  beschreibt eine Zahl
- jeder aussagenlogische Ausdruck (aus Variablen und Operatoren)
  beschreibt eine Funktion (von Variablenbelegung nach Wahrheitswert)
- jeder reguläre Ausdruck
  beschreibt eine formale Sprache
- jedes rekursive definierte Unterprogramm
  beschreibt eine Funktion (?)

Beispiel: Semantik von Unterprogr.

Unterprogramme definiert durch Gleichungssysteme.
Sind diese immer lösbar? (überhaupt? eindeutig?)
Geben Sie geschlossenen arithmetischen Ausdruck für:

```
f (x) = if x > 52
        then x - 11
        else f (f (x + 12))
```

```
g(x,y) =
  if x <= 0 then 0
  else if y <= 0 then 0
  else 1 + g (g (x-1, y), g (x, y-1))
```

Axiomatische Semantik

Notation für Aussagen über Speicherbelegungen:

```
{ V } A { N }
```

- für jede Belegung *s*, in der Vorbedingung *V* gilt:
- wenn Anweisung A ausgeführt wird,
- und Belegung *t* erreicht wird, dann gilt dort Nachbedingung *N*

Beispiel: `{ x >= 5 } y := x + 3 { y >= 7 }`
Gültigkeit solcher Aussagen kann man

- beweisen (mit Hoare-Kalkül)
- prüfen (testen)

Beachte: `{x >= 5} while (true) ; {x == 42}`

Eiffel

Bertrand Meyer, `http://www.eiffel.com/`

```
class Stack [G]      feature
    count : INTEGER
    item : G is require not empty do ... end
    empty : BOOLEAN is do .. end
    full  : BOOLEAN is do .. end
    put (x: G) is
       require not full do ...
       ensure not empty
              item = x
              count = old count + 1
```

Beispiel sinngemäß aus: B. Meyer: Object Oriented Software Construction, Prentice Hall 1997

Hoare-Kalkül

Kalkül: für jede Form der Anweisung ein Axiom, Beispiele:

- Zuweisung: `{ N[x/E] } x := E { N }`
- `wenn { V } C { Z } und { Z } D { N }`
  `dann { V } C; D { N }`
- `wenn { V } A { N } und V' => V und N => N'`
  `dann { V' } A { N' }`
- `wenn { V und B } C { N }`
  `und { V und not B } D { N }`
  `dann { V } if (B) then C else D { N }`
- Schleife . . . benötigt Invariante

Axiom für Schleifen

```
wenn  { I and B } A { I },
dann  { I } while (B) do A { I and not B }
```

Beispiel:

```
Eingabe int p, q;
// p = P und q = Q
int c = 0;
// inv: p * q + c = P * Q
while (q > 0) {
   ???
}
// c = P * Q
```

Moral: erst Schleifeninvariante (Spezifikation), dann Implementierung.

Übungen (Stackmaschine)

Schreiben Sie eine Java-Methode, deren Kompilation genau diesen Bytecode erzeugt: a)

```
public static int h(int, int);
  Code:
     0: iconst_3
     1: iload_0
     2: iadd
     3: iload_1
     4: iconst_4
     5: isub
     6: imul
     7: ireturn
```

b)

```
public static int g(int, int);
  Code:
     0: iload_0
     1: istore_2
```

Übungen (Invarianten)

Ergänze das Programm:

```
Eingabe: natürliche Zahlen a, b;
// a = A und b = B
int p = 1; int c = ???;
// Invariante:  c^b * p = A^B
while (b > 0) {
    ???
    b = abrunden (b/2);
}
Ausgabe: p; // p = A^B
```

Warum Typen?

- Typ ist Menge von Werten mit Operationen
- für jede eigene Menge von Werten (Variablen) aus dem *Anwendungsbereich* benutze eine eigenen Typ
- halte verschiedene Typen sauber getrennt, mit Hilfe der Programmiersprache
- der Typ einer Variablen/Funktion ist ihre beste Dokumentation

Historische Entwicklung

- keine Typen (alles ist Maschinenwort)
- vorgegebene Typen (Fortran: Integer, Real, Arrays)
- benutzerdefinierte Typen

  (algebraische Datentypen;

  Spezialfälle: enum, struct, class)
- abstrakte Datentypen (interface)
- polymorphe Typen (z.B. `List<E>`, aber auch Arrays)
- (data) dependent types (z.B. in Agda, Idris)

Überblick

- einfache (primitive) Typen
  - Zahlen, Wahrheitswerte, Zeichen
  - benutzerdefinierte Aufzählungstypen
  - Teilbereiche
- zusammengesetzte (strukturierte) Typen
  - Produkt (records)
  - Summe (unions) (Spezialfall: Aufzählungen)
  - rekursive Typen
  - Potenz (Funktionen: Arrays, (Tree/Hash-)Maps, Unterprogramme)
  - Verweistypen (Zeiger)

Zahlenbereiche

- Maschinenzahlen (oft im Sprachstandard festgelegt)
  - ganze Zahlen (in binärem Zweierkomplement)
  - gebrochene Zahlen (in binärer Gleitkommadarstellung)
    Goldberg 1991: *What Every Computer Scientist Should Know About Floating-Point Arithmetic*
    `http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`
- Abstraktionen (oft in Bibliotheken, Bsp. `https://gmplib.org//manual/`)
  - beliebig große Zahlen
  - exakte rationale Zahlen

Aufzählungstypen

können einer Teilmenge ganzer Zahlen zugeordnet werden

- durch Sprache vorgegeben: z.B. int, char, boolean
- anwendungsspezifische (benutzerdef.) Aufzählungstypen

```
typedef enum {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
} day;
```

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | S
```

Ü: `enum` in Java

Designfragen:

- automatische oder manuelle Konversion zw. Aufzählungstypen

Maßeinheiten in F#

physikalische Größe $=$ Maßzahl $\times$ Einheit.
viele teure Softwarefehler durch Ignorieren der Einheiten.
in F# (Syme, 200?), aufbauend auf ML (Milner, 197?)

```
[<Measure>] type kg ;;
let x = 1<kg> ;;
x * x ;;
[<Measure>] type s ;;
let y = 2<s> ;;
x * y ;;
x + y ;;
```

http://msdn.microsoft.com/en-us/library/dd233243.aspx

Zeichen und Zeichenketten

- das naive Modell ist:
  - Zeichen paßt in (kurze) Maschinenzahl (z.B. `char = byte`)
  - Zeichenketten sind (Zeiger auf) Arrays

  das ist historisch begründet (US-amerikanische Hardware-Hersteller)
- das umfassende Modell ist `http://www.unicode.org/versions/Unicode9.0.0/`
  jedes Zeichen wird durch *encoding scheme* (z.B. UTF8) auf *Folge* von Bytes abgebildet.

Zusammengesetzte Typen

Typ = Menge, Zusammensetzung = Mengenoperation:

- Produkt (record, struct)
- disjunkte Summe (union, case class, enum)
- Rekursion,
  z.B. `data Tree a = ... | Branch (Tree a) ...`
- Potenz (Funktion),
  z.B. `type Sorter a = (List a -> List a)`

Produkttypen (Records)

$R = A \times B \times C$
Kreuzprodukt mit benannten Komponenten:

```
typedef struct {
    A foo; B bar; C baz;
} R;

R x; ...  B x.bar; ...
```

erstmalig in COBOL ($\leqslant$ 1960) (Interview mit der Erfinderin:
`http://archive.computerhistory.org/resources/text/Oral_History/Hopper_Grace/102702026.05.01.pdf`)
Übung: Record-Konstruktion (in C, C++)?

Summen-Typen

$R = A \cup B \cup C$
disjunkte (diskriminierte) Vereinigung (Pascal)

```
type tag = ( eins, zwei, drei );
type R = record case t : tag of
    eins : ( a_value : A );
    zwei : ( b_value : B );
    drei : ( c_value : C );
end record;
```

nicht diskriminiert (C):

```
typedef union {
    A a_value; B b_value; C c_value;
}
```

Vereinigung mittels Interfaces

I repräsentiert die Vereinigung von *A* und *B*:

```
interface I { }
class A implements I { int foo; }
class B implements I { String bar; }
```

Notation dafür in Scala (`http://scala-lang.org/`)

```
abstract class I
case class A (foo : Int) extends I
case class B (bar : String) extends I
```

Verarbeitung durch *Pattern matching*

```
def g (x : I): Int = x match {
    case A(f) => f + 1
    case B(b) => b.length()  }
```

Rekursiv definierte Typen

Haskell (`http://haskell.org/`)

```
data Tree a = Leaf a
            | Branch ( Tree a ) ( Tree a )
data List a = Nil | Cons a ( List a )
```

Java

```
interface Tree<A> { }
class Leaf<A> implements Tree<A> { A key }
class Branch<A> implements Tree<A>
  { Tree<A> left, Tree<A> right }
```

das ist ein *algebraischer Datentyp*,
die Konstruktoren (Leaf, Nil) bilden die Signatur der Algebra,
die Elemente der Algebra sind Terme (Bäume)

Potenz-Typen

$B^A := \{f : A \to B\}$ (Menge aller Funktionen von A nach B)
ist sinnvolle Notation, denn $|B|^{|A|} = \left|B^A\right|$
spezielle Realisierungen:

- Funktionen (Unterprogramme)
- Wertetabellen (Funktion mit endlichem Definitionsbereich) (Assoziative Felder, Hashmaps)
- Felder (Definitionsbereich ist Aufzählungstyp) (Arrays)
- Zeichenketten (Strings)

die unterschiedliche Notation dafür (Beispiele?) ist bedauerlich.

Felder (Arrays)

Motivation: `a[i] = * (a + w * i)`
Zugriff auf beliebiges Element mit wenig Befehlen
Design-Entscheidungen:

- welche Index-Typen erlaubt? (Zahlen? Aufzählungen?)
- Bereichsprüfungen bei Indizierungen?
- Index-Bereiche statisch oder dynamisch?
- Allokation statisch oder dynamisch?
- Initialisierung?
- mehrdimensionale Felder gemischt oder rechteckig?

Felder in C

```
int main () {
    int a [10][10];
    a[3][2] = 8;
    printf ("%d\n", a[2][12]);
}
```

statische Dimensionierung, dynamische Allokation, keine Bereichsprüfungen.

Form: rechteckig, Adress-Rechnung:

```
int [M][N];
a[x][y]  ==>  *(&a + (N*x + y))
```

Felder in Javascript

https://news.ycombinator.com/item?id=14675706

```
var arr1 = []; arr1[2147483648]=1;
  // arr1.length == 0
var arr2 = []; arr2[2147483647]=1;
  // arr2.length == 2147483648
var arr3 = []; arr3[-1]=1;
  // arr3.length == 0
...
```

Ü: stimmt das (noch)? Ausprobieren und mit Sprachspezifikation vergleichen (https://tc39.github.io/ecma262/)

Felder in Java

```
int [][] feld =
          { {1,2,3}, {3,4}, {5}, {} };
for (int [] line : feld) {
    for (int item : line) {
       System.out.print (item + " ");
    }
    System.out.println ();
}
```

dynamische Dimensionierung und Allokation, Bereichsprüfungen. Nicht notwendig rechteckig.

Felder in C#

Unterschiede zwischen

- `int [][] a`
- `int [,]  a`

in

- Benutzung (Zugriff)
- Initialisierung durch Array-Literal

Nicht rechteckige Felder in C?

Das geht:

```
int a [] = {1,2,3};
int b [] = {4,5};
int c [] = {6};
    e    = {a,b,c};
printf ("%d\n", e[1][1]);
```

aber welches ist dann der Typ von `e`?
(es ist nicht `int e [][]`.)

Kosten der Bereichsüberprüfungen

es wird oft als Argument für C (und gegen Java) angeführt, daß die erzwungene Bereichsüberprüfung bei jedem Array-Zugriff so teuer sei.
sowas sollte man erst glauben, wenn man es selbst gemessen hat.
modernen Java-Compiler sind *sehr clever* und können *theorem-prove away (most) subscript range checks*
das kann man auch in der Assembler-Ausgabe des JIT-Compilers sehen.
`http://www.imn.htwk-leipzig.de/~waldmann/etc/safe-speed/`

Verweistypen

- Typ T, Typ der Verweise auf T.
- Operationen: new, put, get, delete
- ähnlich zu Arrays (das Array ist der Hauptspeicher)

explizite Verweise in C, Pascal
implizite Verweise:

- Java: alle nicht primitiven Typen sind Verweistypen, De-Referenzierung ist implizit
- C#: class ist Verweistyp, struct ist Werttyp

Verweis- und Wertsemantik in C#

- für Objekte, deren Typ `class ...` ist:
  Verweis-Semantik (wie in Java)
- für Objekte, deren Typ `struct ...` ist:
  Wert-Semantik

Testfall:

```
class s {public int foo; public string bar;}
s x = new s(); x.foo = 3; x.bar = "bar";
s y = x; y.bar = "foo";
Console.WriteLine (x.bar);
```

und dann `class` durch `struct` ersetzen

Algebraische Datentypen in Pascal, C

Rekursion unter Verwendung von Verweistypen
Pascal:

```
type Tree = ^ Node ;
type Tag = ( Leaf, Branch );
type Node = record case t : Tag of
  Leaf : ( key : T ) ;
  Branch : ( left : Tree ; right : Tree );
end record;
```

C: ähnlich, benutze typedef

Null-Zeiger: der Milliarden-Dollar-Fehler

- Tony Hoare (2009): [The null reference] has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
  (`https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-`
- Das Problem sind nicht die Zeiger selbst, sondern daß (in vielen Sprachen) der Wert `null` zu jedem Zeigertyp gehört — obwohl er gar kein Zeiger ist
- vgl. auch Diskussion auf `https://news.ycombinator.com/item?id=11798518`

Übung Typen

- für Mengen
  $A = \emptyset, B = \{0\}, C = \{1, 2\}, D = \{3, 4, 5\}, E = \{6, 7, 8, 9\}$,
  geben Sie an:
  - alle Elemente von $A \times C, B \times D, A \cup B, B^A, A^B, C^B, B^C, C^D$
  - ein Element aus $(C \times D)^E$
  - die Kardinalitäten von $(C \times D)^E, C^{D \cup E}$
- algebraische Datentypen und Polymorphie in Haskell
  (vgl. VL Fortgeschrittene Programmierung (Bachelor)
  `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss16/fop/folien/#(20)`,
  `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss16/fop/folien/#(41)`)
- Arrays in C (Assemblercode anschauen)
- rechteckige und geschachtelte Arrays in C#
- Wert/Verweis (struct/class) in C#

Variablen

vereinfacht: Variable bezeichnet eine (logische) Speicherzelle
genauer: Variable besitzt Attribute

- Name
- Adresse
- Wert
- Typ
- Lebensdauer
- Sichtbarkeitsbereich

Bindungen dieser Attribute *statisch* oder *dynamisch*

Namen in der Mathematik

- ein Name bezeichnet einen unveränderlichen Wert
  $e = \sum_{n \geqslant 0} \frac{1}{n!}, \quad \sin = (x \mapsto \sum_{n \geqslant 0} (-1)^n \frac{x^{2n+1}}{(2n+1)!})$
- auch n und x sind dabei lokale Konstanten (werden aber gern „Variablen“ genannt)
- auch die „Variablen“ in Gleichungssystemen sind (unbekannte) Konstanten $\{x + y = 1 \wedge 2x + y = 1\}$

in der Programmierung:

- Variable ist Name für Speicherstelle (= konstanter Zeiger)
- implizite Dereferenzierung beim Lesen und Schreiben
- Konstante: Zeiger auf schreibgeschützte Speicherstelle

Namen

- welche Buchstaben/Zeichen sind erlaubt?
- reservierte Bezeichner?
- Groß/Kleinschreibung?
- Konvention: `long_name` oder `longName` (camel-case)

  (Fortran: `long name`)

  im Zweifelsfall: Konvention der Umgebung einhalten
- Konvention: Typ im Namen (Bsp.: `myStack = ...`)
  - verrät Details der Implementierung
  - ist ungeprüfte Behauptung

  besser: `Stack<Ding> rest_of_input = ...`

Typen für Variablen

- dynamisch (Wert hat Typ)
- statisch (Name hat Typ)
  - deklariert (durch Programmierer)
  - inferiert (durch Übersetzer)
    z. B. `var` in C#3

Vor/Nachteile: Lesbarkeit, Sicherheit, Kosten

Dynamisch getypte Sprachen

Daten sind typisiert, Namen sind nicht typisiert.
LISP, Clojure, PHP, Python, Perl, Javascript, ...

```
var foo = function(x) {return 3*x;};
foo(1);
foo = "bar";
foo(1);
```

Statisch getypte Sprachen

Daten sind typisiert, Namen sind typisiert
Invariante:

- zur Laufzeit ist der *dynamische Typ* des Namens (der Typ des Wertes des Namens)
  immer gleich dem *statischen Typ* des Namens (der deklariert oder inferiert wurde)

woher kommt der statische Typ?

- Programmierer deklariert Typen von Namen
  C, Java, ...
- Compiler inferiert Typen von Namen
  ML, F#, Haskell, C# (var)

Typdeklarationen

im einfachsten Fall (Java, C#):

```
Typname Variablenname [ = Initialisierung ] ;
int []  a = { 1, 2, 3 };
Func<double,double> f = (x => sin(x));
```

gern auch komplizierter (C): dort gibt es keine Syntax für Typen, sondern nur für Deklarationen von Namen.

```
double f (double x) { return sin(x); }
int * p;
double ( * a [2]) (double) ;
```

Beachte: `*` und `[]` werden „von außen nach innen“ angewendet
Ü: Syntaxbäume zeichnen, `a` benutzen

Typinferenz in C# und Java

C#:

```
public class infer {
    public static void Main (string [] argv) {
        var arg = argv[0];
        var len = arg.Length;
        System.Console.WriteLine (len);
    }
}
```

Ü: das `var` in C# ist nicht das `var` aus Javascript.

Java:
für formale Parameter von anonymen Unterprogrammen

```
Function<Integer,Integer> f = (x) -> x;
```

Konstanten

= Variablen, an die genau einmal zugewiesen wird

- C: const (ist Attribut für Typ)
- Java: final (ist Attribut für Variable)

Vorsicht:

```
class C { int foo; }
static void g (final C x) { x.foo ++; }
```

Merksatz: alle Deklarationen so lokal und so konstant wie möglich!
(D. h. Attribute *immutable* usw.)

Lebensort und -Dauer von Name und Daten

- statisch (global, aber auch lokal:)

```
int f (int x) {
    static int y = 3; y++; return x+y;
}
```

- dynamisch
  - Stack `{ int x = ... }`
  - Heap
    - explizit (new/delete, malloc/free)
    - implizit

Beachte (in Java u.ä.) in `{ C x = new C(); }` ist `x` Stack-lokal, Inhalt ist Zeiger auf das Heap-globale Objekt.

Sichtbarkeit von Namen

= Bereich der Anweisungen/Deklarationen, in denen ein Name benutzt werden kann.

- global
- lokal: Block (und Unterblöcke)

Üblich ist: Sichtbarkeit beginnt nach Deklaration und endet am Ende des umgebenden Blockes

Tatsächlich (Java, C):
Sichtbarkeit beginnt schon in der Initalisierung

```
int x = sizeof(x); printf ("%d\n", x);
```

Ü: ähnliches Beispiel für Java? Vgl. JLS Kapitel 6.

Sichtbarkeit in JavaScript

Namen sind sichtbar

- global
- in Unterprogramm (Deklaration mit `var`)

```
(function() { { var x = 8; } return x; } ) ()
```

- in Block (Deklaration mit `let`)

```
(function() { { let x = 8; } return x; } ) ()
```

Ü: erkläre das Verhalten von

```
(function(){let x=8; {x=9} return x} )()
(function(){let x=8; {x=9;let x=10} return x} )()
```

durch die Sprachspezifikation
(und nicht durch Sekundärquellen)

Überdeckungen

Namen sind auch in inneren Blöcken sichtbar:

```
int x;
while (..) {
  int y;
  ... x + y ...
}
```

innere Deklarationen verdecken äußere:

```
int x;
while (..) {
  int x;
  ... x ...
}
```

Sichtbarkeit und Lebensdauer

. . . stimmen nicht immer überein:

- static-Variablen in C-Funktionen
  sichtbar: in Funktion, Leben: Programm

```
void u () { static int x; }
```

- lokale Variablen in Unterprogrammen
  sichtbar: innere Blöcke, Leben: bis Ende Unterpr.

```
void u () {
  int *p; { int x = 8; p = &x; }
  printf ("%d\n", *p);
}
```

Einleitung

- Ausdruck hat *Wert* (Zahl, Objekt, . . .)
  (Ausdruck wird *ausgewertet*)
- Anweisung hat *Wirkung* (Änderung des Speicher/Welt-Zustandes)
  (Anweisung wird *ausgeführt*)

Vgl. Trennung (in Pascal, Ada)

- Funktion (Aufruf ist Ausdruck)
- Prozedur (Aufruf ist Anweisung)

Ü: wie in Java ausgedrückt? wie stark getrennt?

Einleitung (II)

- in allen imperativen Sprachen gibt es Ausdrücke mit Nebenwirkungen (nämlich Unterprogramm-Aufrufe)
- in den rein funktionalen Sprachen gibt es keine (Neben-)Wirkungen, also keine Anweisungen (sondern nur Ausdrücke).
- in den C-ähnlichen Sprachen ist = ein Operator, (d. h. die Zuweisung ist syntaktisch ein Ausdruck, kann Teil von anderen Ausdrücken sein)

  ```
  int x = 3; int y = x + (x = 4);
  ```

- in den C-ähnlichen Sprachen: Ausdruck ist als Anweisung gestattet (z.B. in Block)

  ```
  { int x = 3; x++ ; System.out.println(x); }
  ```

Designfragen für Ausdrücke

- Syntax
  - Präzedenzen (Vorrang)
  - Assoziativitäten (Gruppierung)
  - kann Programmierer neue Operatoren definieren?
- statische Semantik
  - . . . vorhandene Operatornamen überladen?
  - Typen der Operatoren?
  - implizite, explizite Typumwandlungen?
- dynamische Semantik
  - Ausdrücke dürfen (Neben-)Wirkungen haben?
  - falls mehrere: in welcher Reihenfolge?

Syntax von Ausdrücken

- einfache Ausdrücke : Literale, (Variablen-)Namen
- zusammengesetzte Ausdrücke:
  - Operator-Symbol zwischen Argumenten
  - Funktions-Symbol vor Argument-Tupel

wichtige Spezialfälle für Operatoren:

- arithmetische (von Zahlen nach Zahl)
- relationale (von Zahlen nach Wahrheitswert)
- boolesche (von Wahrheitswerten nach Wahrheitsw.)

Wdhlg: Syntaxbaum, Präzedenz, Assoziativität.

Syntax von Literalen

- Was druckt diese Anweisung?

```
System.out.println ( 12345 + 54321 );
```

- dieses und einige der folgenden Beispiele aus: Joshua Bloch, Neil Gafter: *Java Puzzlers*, Addison-Wesley, 2005.

Der Plus-Operator in Java

- . . . addiert Zahlen und verkettet Strings.
- ```
  System.out.println ("foo" + 3 + 4);
  System.out.println (3 + 4 + "bar");
  ```
- Vorgehen für die Analyse:
  - abstrakten Syntaxbaum bestimmen
  - Typen (als Attribute der AST-Knoten) bestimmen,
  - dabei implizite Typ-Umwandlungen einfügen (in diesem Fall `Integer.toString()`)
  - Werte (als Attribute) bestimmen
  ```

Überladene Operatornamen

- Def: Name *n* ist *überladen*, falls *n* mehrere Bedeutungen hat
- aus praktischen Gründen sind arithmetische und relationale Operatornamen *überladen*
- Überladung wird statisch aufgelöst durch die Typen der Argumente.
- Beispiel:

```
int x = 3; int y = 4; ... x + y ...
double a;  double b; ... a + b ...
String p;  String q; ... p + q ...
```

Automatische Typanpassungen

- in vielen Sprachen postuliert man eine Hierarchie von Zahlbereichstypen:
  $\text{byte} \subseteq \text{int} \subseteq \text{float} \subseteq \text{double}$
  im allgemeinen ist das eine Halbordnung.
- Operator mit Argumenten verschiedener Typen:
  `(x :: int) + (y :: float)`
  beide Argumente werden zu kleinstem gemeinsamen Obertyp promoviert, falls dieser eindeutig ist (sonst statischer Typfehler)
  (Halbordnung $\rightarrow$ Halbverband)
- (das ist die richtige Benutzung von *promovieren*)

Implizite/Explizite Typumwandlungen

- Was druckt dieses Programm?

```
long x = 1000 * 1000 * 1000 * 1000;
long y = 1000 * 1000;
System.out.println ( x / y );
```

- Was druckt dieses Programm?

```
System.out.println ((int) (char) (byte) -1);
```

- Moral: wenn man nicht auf den ersten Blick sieht, was ein Programm macht, dann macht es wahrscheinlich nicht das, was man will.

Explizite Typumwandlungen

sieht gleich aus und heißt gleich (cast), hat aber verschiedene Bedeutungen:

- Datum soll in anderen Typ gewandelt werden, Repräsentation ändert sich:

```
double x = (double) 2 / (double) 3;
```

- Programmierer weiß es besser (als der Compiler), Code für Typprüfung zur Laufzeit wird erzeugt, Repräsentation ändert sich nicht:

```
List books;
Book b = (Book) books.get (7);
```

Typ-Umwandlungen in Javascript

Gary Bernhardt: WAT (2012)

`https://www.destroyallsoftware.com/talks/wat`

Der Verzweigungs-Operator

Absicht: statt

```
if ( 0 == x % 2 ) {
  x = x / 2;
} else {
  x = 3 * x + 1;
}
```

lieber

```
x = if ( 0 == x % 2 ) {
      x / 2
    } else {
      3 * x + 1
    } ;
```

historische Notation dafür

```
x = ( 0 == x % 2 ) ? x / 2 : 3 * x + 1;
```

?/: ist *ternärer* Operator

Verzweigungs-Operator(II)

`(... ? ... : ... )` in C, C++, Java
Anwendung im Ziel einer Zuweisung (C++):

```
int main () {
    int a = 4; int b = 5; int c = 6;
    ( c < 7 ? a : b ) = 8;
}
```

Relationale Operatoren

kleiner, größer, gleich,...
Was tut dieses Programm (C? Java?)

```
int a = -4; int b = -3; int c = -2;
if (a < b < c) {
    printf ("aufsteigend");
}
```

Logische (Boolesche) Ausdrücke

- und `&&`, `||` oder, nicht `!` Negation
- nicht verwechseln mit Bit-Operationen `&`, `|`
  (in C gefährlich, in Java ungefährlich—warum?)
- verkürzte Auswertung?

```
int [] a = ...; int k = ...;
if ( k >= 0 && a[k] > 7 ) { ... }
```

  (Ü: wie sieht das in Ada aus?)
- Ü: welche relationalen Operatoren sind in Java für `boolean` überladen?

Noch mehr Quizfragen

- ```
  System.out.println ("H" + "a");
  System.out.println ('H' + 'a');
  ```
- ```
  char x = 'X'; int i = 0;
  System.out.print (true  ? x : 0);
  System.out.print (false ? i : x);
  ```

Erklären durch Verweis auf Java Language Spec.

Der Zuweisungs-Operator

Syntax:

- Algol, Pascal: Zuweisung `:=`, Vergleich `=`
- Fortran, C, Java: Zuweisung `=`, Vergleich `==`

Semantik der Zuweisung `a = b`:
Ausdrücke links und rechts werden verschieden behandelt:

- bestimme Adresse (lvalue) *p* von `a`
- bestimme Wert (rvalue) *v* von `b`
- schreibe *v* auf *p*

Weitere Formen der Zuweisung

(in C-ähnlichen Sprachen)

- verkürzte Zuweisung: `a += b`
  entsprechend für andere binäre Operatoren
  - lvalue *p* von *a* wird bestimmt (nur einmal)
  - rvalue *v* von *b* wird bestimmt
  - Wert auf Adresse *p* wird um *v* erhöht
- Inkrement/Dekrement
  - Präfix-Version `++i, --j`: Wert ist der geänderte
  - Suffix-Version `i++, j--`: Wert ist der vorherige

Ausdrücke mit Nebenwirkungen

(*side effect*; falsche Übersetzung: Seiteneffekt)

- in C-ähnlichen Sprachen: Zuweisungs-Operatoren bilden Ausdrücke, d. h. Zuweisungen sind Ausdrücke und können als Teile von Ausdrücken vorkommen.
- Wert einer Zuweisung ist der zugewiesene Wert

```
int a; int b; a = b = 5; // wie geklammert?
```

- Komma-Operator zur Verkettung von Ausdrücken (mit Nebenwirkungen)

```
for (... ; ... ; i++,j--) { ... }
```

Auswertungsreihenfolgen

Kritisch: wenn Wert des Ausdrucks von Auswertungsreihenfolge abhängt:

```
int a; int b = (a = 5) + (a = 6);
int d = 3; int e = (d++) - (++d);
int x = 3; int y = ++x + ++x + ++x;
```

- keine Nebenwirkungen: egal
- mit Nebenwirkungen:
  - C, C++: Reihenfolge nicht spezifiziert, wenn Wert davon abhängt, dann ist Verhalten *nicht definiert*
  - Java, C#: Reihenfolge genau spezifiziert (siehe JLS)

Auswertungsreihenfolge in C

Sprachstandard (C99, C++) benutzt Begriff *sequence point* (Meilenstein): bei Komma, Fragezeichen, `&&` und `||`

die Nebenwirkungen zwischen Meilensteinen müssen *unabhängig* sein (nicht die gleiche Speicherstelle betreffen), ansonsten ist das Verhalten *undefiniert*, d.h., der Compiler darf *beliebigen* Code erzeugen, z.B. solchen, der die Festplatte löscht oder Cthulhu heraufbeschwört. vgl. Aussagen zu

sequence points in `http://gcc.gnu.org/readings.html` und
Gurevich, Huggins: *Semantics of C*, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.6755`

Definition

Semantik: Ausführen einer Anweisung bewirkt Zustandsänderung
abstrakte Syntax:

- einfache Anweisung:
  - Zuweisung
  - leere Anweisung, `break`, `continue`, `return`, `throw`
  - Unterprogramm-Aufruf
- zusammengesetzte Anweisung:
  - Nacheinanderausführung (Block)
  - Verzweigung (zweifach: `if`, mehrfach: `switch`)
  - Wiederholung (Sprung, Schleife)

Programm-Ablauf-Steuerung

- Ausführen eines Programms im von-Neumann-Modell:
  - Was? (Operation)
  - Womit? (Operanden)
  - Wohin? (Resultat)
  - Wie weiter? (nächste Anweisung)
- Ablaufsteuerung durch strukturierte Programmierung:
  - Nacheinander
  - Verzweigung
  - Wiederholung
  - außer der Reihe (Sprung, Unterprogramm, Exception)

engl. *control flow*, falsche Übersetzung: Kontrollfluß;
to control = steuern, *to check* = kontrollieren/prüfen

Blöcke

Folge von (Deklarationen und) Anweisungen
Designfrage: Blöcke

- explizit (Klammern, begin/end)
- implizit (if . . . then . . . end if)

Designfrage/historische Entwicklung: Deklarationen . . .

- am Beginn des Programms (Maschinenp., COBOL, Fortran)
- am Beginn jedes Unter-Programms (Pascal)
- am Beginn jedes Blocks (C)
- an jeder Stelle jedes Blocks (C++, Java)

Verzweigungen (zweifach)

in den meisten Sprachen:

```
if Bedingung then Anweisung1
      [ else Anweisung2 ]
```

Designfragen:

- was ist als Bedingung gestattet (gibt es einen Typ für Wahrheitswerte?)
- dangling else
  - gelöst durch Festlegung (else gehört zu letztem if)
  - vermieden durch Block-Bildung (Perl, Ada)
  - tritt nicht auf, weil man else nie weglassen darf (vgl. ?/:) (Haskell)

Mehrfach-Verzweigung

Syntax:

```
switch (e) {
   case c1 : s1 ;
   case c2 : s2 ;
   [ default : sn; ]  }
```

Semantik

```
if (e == c1) s1
else if (e == c2) s2
   ... else sn
```

- Bezeichnung: der Ausdruck `e` heißt *Diskriminante*
- Vorsicht! Das ist *nicht* die Semantik in C(++), Java.
- welche Typen für `e`? (z.B.: Aufzählungstypen)
- Wertebereiche? (`case c1 .. c2 : ...`)
- was passiert, wenn mehrere Fälle zutreffen?
  (z.B.: statisch verhindert dadurch, daß `ci` verschiedene Literale sein müssen)

switch/break

```
switch (index) {
  case 1  : odd  ++;
  case 2  : even ++;
  default :
    printf ("wrong index %d\n", index);
}
```

- Semantik in C, C++, Java ist nicht „führe den zum Wert der Diskriminante passenden Zweig aus“
- sondern „. . . passenden Zweig aus *sowie alle danach folgenden Zweige*“.
- C#: jeder Zweig *muß* mit `break` oder `goto` enden.

Kompilation

ein switch (mit vielen cases) wird übersetzt in:

- (naiv) eine lineare Folge von binären Verzweigungen (if, elsif)
- (semi-clever) einen balancierter Baum von binären Verzweigungen
- (clever) eine Sprungtabelle

Übung:

- einen langen Switch (1000 Fälle) erzeugen (durch ein Programm!)
- Assembler/Bytecode anschauen

Pattern Matching

- Fallunterscheidung nach dem Konstruktor
- Bindung von lokalen Namen

```
abstract class Term    // Scala
case class Constant (value : Int)
    extends Term
case class Plus (left: Term, right : Term)
    extends Term
def eval(t: Term): Int = {
  t match {
    case Constant(v) => v
    case Plus(l, r) => eval(l) + eval(r)
  } }
```

Wiederholungen

- Maschine, Assembler: (un-)bedingter Sprung
- strukturiert: Schleifen

Designfragen für Schleifen:

- wie wird Schleife gesteuert? (Bedingung, Zähler, Daten, Zustand)
- an welcher Stelle in der Schleife findet Steuerung statt (Anfang, Ende, dazwischen, evtl. mehreres)

Schleifen steuern durch. . .

- Zähler

```
for p in 1 .. 10 loop .. end loop;
```

- Daten

```
map (\x -> x*x) [1,2,3] ==> [1,4,9]
Collection<String> c
    = new LinkedList<String> ();
for (String s : c) { ... }
```

- Bedingung

```
while ( x > 0 ) { if ( ... ) { x = ... } ... }
```

- Zustand (Iterator, hasNext, next)

Zählschleifen

Idee: vor Beginn steht Anzahl der Durchläufe fest.

richtig realisiert ist das nur in Ada:

```
for p in 1 .. 10 loop ... end loop;
```

- Zähler `p` wird implizit deklariert
- Zähler ist im Schleifenkörper konstant

Vergleiche (beide Punkte) mit Java, C++, C

Termination

Satz: Jedes Programm aus

- Zuweisungen
- Verzweigungen
- Zählschleifen

terminiert (hält) für jede Eingabe.

Äquivalenter Begriff (für Bäume anstatt Zahlen): strukturelle Induktion (fold, Visitor, primitive Rekursion)

Satz: es gibt berechenbare Funktionen, die nicht primitiv rekursiv sind.
Beispiel: Interpreter für primitiv rekursive Programme.

Datengesteuerte Schleifen

Idee: führe für jeden Konstruktor eines algebraischen Datentyps (Liste, Baum) eine Rechnung/Aktion aus.

`foreach, Parallel.Foreach,...`

Zustandsgesteuerte Schleifen

So:

```
interface Iterator<T> {
  boolean hasNext(); T next ();  }
interface Iterable<T> {
   Iterator<T> iterator(); }
for (T x : ...) { ... }
```

Oder so:

```
public interface IEnumerator<T> : IEnumerator {
  bool MoveNext(); T Current { get; } }
interface IEnumerable<out T> : IEnumerable {
   IEnumerator<T> GetEnumerator() }
foreach (T x in ...) { ... }
```

(sieben Unterschiede . . .)

Implizite Iteratoren in C#

```
using System.Collections.Generic;
public class it {
    public static IEnumerable<int> Data () {
        yield return 3;
        yield return 1;
        yield return 4;
    }
    public static void Main () {
        foreach (int i in Data()) {
            System.Console.WriteLine (i);
}   }   }
```

Schleifen mit Bedingungen

das ist die allgemeinste Form, ergibt (partielle) rekursive Funktionen, die terminieren nicht notwendig für alle Argumente.
Steuerung

- am Anfang: `while (Bedingung) Anweisung`
- am Ende: `do Anweisung while (Bedingung)`

Weitere Änderung des Ablaufes:

- vorzeitiger Abbruch (break)
- vorzeitige Wiederholung (continue)
- beides auch nicht lokal

Abarbeitung von Schleifen

operationale Semantik durch Sprünge:

```
while (B) A;
==>
start : if (!B) goto end;
        A;
        goto start;
end   : skip;
```

(das ist auch die Notation der autotool-Aufgabe)

Ü: `do A while (B);`

vorzeitiges Verlassen

- ...der Schleife

```
while ( B1 ) {
    A1;
    if ( B2 ) break;
    A2;
}
```

- ...des Schleifenkörpers

```
while ( B1 ) {
    A1;
    if ( B2 ) continue;
    A2;
}
```

Geschachtelte Schleifen

manche Sprachen gestatten Markierungen (Labels) an Schleifen, auf die man sich in break beziehen kann:

```
foo : for (int i = ...) {
  bar : for (int j = ...) {

    if (...) break foo;

  }
}
```

Wie könnte man das simulieren?

Sprünge

- bedingte, unbedingte (mit bekanntem Ziel)
  - Maschinensprachen, Assembler, Java-Bytecode
  - Fortran, Basic: if Bedingung then Zeilennummer
  - Fortran: dreifach-Verzweigung (arithmetic-if)
- “computed goto” (Zeilennr. des Sprungziels ausrechnen)

Sprünge und Schleifen

- man kann jedes while-Programm in ein goto-Programm übersetzen
- und jedes goto-Programm in ein while-Programm ...
- ...das normalerweise besser zu verstehen ist.
- strukturierte Programmierung = jeder Programmbaustein hat genau einen Eingang und genau einen Ausgang
- aber: vorzeitiges Verlassen von Schleifen
- aber: Ausnahmen (Exceptions)

Sprünge und Schleifen (Beweis)

Satz: zu jedem goto-Programm gibt es ein äquivalentes while-Programm.
Beweis-Idee: `1 : A1, 2 : A2; .. 5: goto 7; ..` $\Rightarrow$

```
while (true) {
  switch (pc) {
    case 1 : A1 ; pc++ ; break; ...
    case 5 : pc = 7 ; break; ...
  }
}
```

Das nützt aber softwaretechnisch wenig, das übersetzte Programm ist genauso schwer zu warten wie das Original.

Schleifen und Unterprogramme

zu jedem while-Programm kann man ein äquivalentes angeben, das nur Verzweigungen (if) und Unterprogramme benutzt.

Beweis-Idee: `while (B) A;` $\Rightarrow$

```
void s () {
    if (B) { A; s (); }
}
```

Anwendung: C-Programme ohne Schlüsselwörter.

Was ist hier los?

```
class What {
    public static void main (String [] args) {
        System.out.println ("mozilla:open");
        http://haskell.org
        System.out.println ("mozilla:close");
    }
}
```

Denotationale Semantik (I)

vereinfachtes Modell, damit Eigenschaften entscheidbar werden (sind die Programme P_1, P_2 äquivalent?)
Syntax: Programme

- Aktionen,
- Zustandsprädikate (in Tests)
- Sequenz/Block, if, goto/while.

Beispiel:

```
while (B && !C) { P; if (C) Q; }
```

Denotationale Semantik (II)

Semantik des Programms P ist Menge der Spuren von P.

- *Spur* = eine Folge von Paaren von Zustand und Aktion,
- ein *Zustand* ist eine Belegung der Prädikatsymbole,
- jede Aktion zerstört alle Zustandsinformation.

Satz: Diese Spursprachen (von goto- und while-Programmen) sind *regulär*.
Beweis: Konstruktion über endlichen Automaten.

- Zustandsmenge = Prädikatbelegungen $\times$ Anweisungs-Nummer
- Transitionen? (Beispiele)

Damit ist Spur-Äquivalenz von Programmen entscheidbar.
Beziehung zu tatsächlicher Äquivalenz?

Grundsätzliches

- Ein Unterprogramm ist ein Block mit einer Schnittstelle.
- Funktion: liefert Wert, Aufruf ist Ausdruck
  Prozedur: liefert keinen Wert, Aufruf ist Anweisung
  (*benannt* oder *anonym* = Lambda-Ausdruck)
- Schnittstelle beschreibt Datentransport zwischen Aufrufer und Unterprogramm.
- zu Schnittstelle gehören
  - Deklarat. d. formalen Parameter (Name, Typ, Modus)
  - bei Funktionen: Deklaration des Resultattyps

Parameter-Übergabe (Semantik)

Datenaustausch zw. Aufrufer (caller) und Aufgerufenem (callee): über globalen Speicher

```
#include <errno.h>
extern int errno;
```

oder über Parameter.
Datentransport (entspr. Schüsselwörtern in Ada)

- in: (Argumente) vom Aufrufer zum Aufgerufenen
- out: (Resultate) vom Aufgerufenen zum Aufrufer
- in out: in beide Richtungen

Parameter-Übergabe (Implementierungen)

- pass-by-value (Wert)
- copy in/copy out (Wert)
- pass-by-reference (Verweis)
  d.h. der formale Parameter im Unterprogramm bezeichnet
  die gleiche Speicherstelle
  wie das Argument beim Aufrufer
  (Argument-Ausdruck muß lvalue besitzen)
- pass-by-name (textuelle Substitution)
  selten . . . Algol68, CPP-Macros . . . Vorsicht!

Parameterübergabe

häufig benutzte Implementierungen:

- Pascal: by-value (default) oder by-reference (VAR)
- C: immer by-value (Verweise ggf. selbst herstellen)
- C++ by-value *oder* by-reference (durch `&`)

```
void p(int & x) { x++; } int y = 3; p(y);
```

- Java: immer by-value

  (beachte implizite Zeiger bei Verweistypen)
- C#: by-value (beachte implizite Zeiger bei Verweistypen, `class`, jedoch nicht bei `struct`)

  oder by-reference (mit Schlüsselwort `ref`)
- Scala: by-value oder by-name (Scala Lang Spec 6.6)

Call-by-value, call-by-reference (C#)

by value:

```
static void u (int x) { x = x + 1; }
int y = 3 ; u (y);
Console.WriteLine(y); // 3
```

by reference:

```
static void u (ref int x) { x = x + 1; }
int y = 3 ; u (ref y);
Console.WriteLine(y); // 4
```

Übung: ref/kein ref; struct (Werttyp)/class (Verweistyp)

```
class C { public int foo }
static void u (ref C x) { x.foo=4; x=new C{foo=5};}
C y = new C {foo=3} ; C z = y; u (ref y);
Console.WriteLine(y.foo + " " + z.foo);
```

Call-by-name

formaler Parameter wird durch Argument-Ausdruck ersetzt.
Algol(68): Jensen's device

```
int sum (int i, int n; int f) {
  int s = 0;
  for (i=0; i<n; i++) { s += f; }
  return s;
}
int [10][10] a; int k; sum (k, 10, a[k][k]);
```

moderne Lösung

```
int sum (int n; Func<int,int> f) {
   ...  { s += f (i); }
}
int [10][10] a; sum (10, (int k) => a[k][k]);
```

Call-by-name (Macros)

```
#define thrice(x) 3*x // gefährlich
thrice (4+y)  ==>  3*4+y
```

“the need for a preprocessor shows omissions in the language”

- fehlendes Modulsystem (Header-Includes)
- fehlende generische Polymorphie ($\Rightarrow$ Templates in C+)

weitere Argumente:

- mangelndes Vertrauen in optimierende Compiler (inlining)
- bedingte Übersetzung

Ü: was kann der Präprozessor in C# und was nicht? Warum?
(Wo ist der C#-Standard?
`http://stackoverflow.com/questions/13467103`)

Call-by-name in Scala

Parameter-Typ ist `=> T`, entspr. „eine Aktion, die ein `T` liefert“(in Haskell: `IO T`)
call-by-name

```
def F(b:Boolean,x: =>Int):Int =
    { if (b) x*x else 0 }
F(false,{print ("foo "); 3})
//      res5: Int = 0
F(true,{print ("foo "); 3})
//     foo foo res6: Int = 9
```

Man benötigt call-by-name zur Definition von Abstraktionen über den Programmablauf.
Übung: `If, While` als Scala-Unterprogramm

Bedarfsauswertung

- andere Namen: (call-by-need, lazy evaluation)
- Definition: das Argument wird bei seiner ersten Benutzung ausgewertet
- wenn es nicht benutzt wird, dann nicht ausgewertet;
  wenn mehrfach benutzt, dann nur einmal ausgewertet
- das ist der Standard-Auswertungsmodus in Haskell:
  alle Funktionen und Konstruktoren sind *lazy*
  da es keine Nebenwirkungen gibt, bemerkt man das zunächst nicht . . .
  . . . und kann es ausnutzen beim Rechnen mit unendlichen Datenstrukturen (Streams)

Beispiele f. Bedarfsauswertung (Haskell)

- ```
  [ error "foo" , 42 ] !! 0
  [ error "foo" , 42 ] !! 1
  length [ error "foo" , 42 ]
  let xs = "bar" : xs
  take 5 xs
  ```
- Fibonacci-Folge

  ```
  fib :: [ Integer ]
  fib = 0 : 1 : zipWith (+) fib ( tail fib )
  ```
- Primzahlen (Sieb des Eratosthenes)
- Papier-Falt-Folge

  ```
  let merge (x:xs) ys = x : merge ys xs
  let updown = 0 : 1 : updown
  let paper = merge updown paper
  take 15 paper
  ```

  vgl. `https://www.imn.htwk-leipzig.de/~waldmann/etc/stream/`
  ```

Beispiele f. Bedarfsauswertung (Scala)

Bedarfsauswertung für eine lokale Konstante (Schlüsselwort `lazy`)

```
def F(b:Boolean,x: =>Int):Int =
    { lazy val y = x; if (b) y*y else 0 }
F(true,{print ("foo "); 3})
//   foo res8: Int = 9
F(false,{print ("foo "); 3})
//   res9: Int = 0
```

Argumente/Parameter

- in der Deklaration benutzte Namen heißen (formale) *Parameter*,
- bei Aufruf benutzte Ausdrücke heißen *Argumente* (. . . nicht: aktuelle Parameter, denn engl. *actual* = dt. tatsächlich)

Designfragen bei Parameterzuordnung:

- über Position oder Namen? gemischt?
- defaults für fehlende Argumente?
- beliebig lange Argumentlisten?

Positionelle/benannte Argumente

Üblich ist Zuordnung über Position

```
void p (int height, String name) { ... }
p (8, "foo");
```

in Ada: Zuordnung über Namen möglich

```
procedure Paint (height : Float; width : Float);
Paint (width => 30, height => 40);
```

nach erstem benannten Argument keine positionellen mehr erlaubt

code smell: lange Parameterliste,
refactoring: Parameterobjekt einführen
allerdings fehlt (in Java) benannte Notation für Record-Konstanten.

Default-Werte

C++:

```
void p (int x, int y, int z = 8);
p (3, 4, 5); p (3, 4);
```

Default-Parameter müssen in Deklaration am Ende der Liste stehen

Ada:

```
procedure P
     (X : Integer; Y : Integer := 8; Z : Integer);
P (4, Z => 7);
```

Beim Aufruf nach weggelassenem Argument nur noch benannte Notation

Variable Argumentanzahl (C)

wieso geht das eigentlich:

```
#include <stdio.h>
char * fmt = really_complicated();
printf (fmt, x, y, z);
```

Anzahl und Typ der weiteren Argumente werden überhaupt nicht geprüft:

```
extern int printf
    (__const char *__restrict __format, ...);
```

Variable Argumentanzahl (Java)

```
static void check (String x, int ... ys) {
    for (int y : ys) { System.out.println (y); }
}

check ("foo",1,2); check ("bar",1,2,3,4);
```

letzter formaler Parameter kann für beliebig viele des gleichen Typs stehen.

tatsächlich gilt `int [] ys`,
das ergibt leider Probleme bei generischen Typen

Aufgaben zu Parameter-Modi (I)

Erklären Sie den Unterschied zwischen (Ada)

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Check is
   procedure Sub (X: in out Integer;
                  Y: in out Integer;
                  Z: in out Integer) is
   begin
      Y := 8; Z := X;
   end;
   Foo: Integer := 9;   Bar: Integer := 7;
begin
   Sub (Foo,Foo,Bar);
   Put_Line (Integer'Image(Foo));
   Put_Line (Integer'Image(Bar));
end Check;
```

(in Datei `Check.adb` schreiben, kompilieren mit `gnatmake Check.adb`)

und (C++)

Aufgaben zu Parameter-Modi (II)

Durch welchen Aufruf kann man diese beiden Unterprogramme semantisch voneinander unterscheiden:
Funktion (C++): (call by reference)

```
void swap (int & x, int & y)
   { int h = x; x = y; y = h; }
```

Makro (C): (call by name)

```
#define swap(x, y) \
   { int h = x; x = y; y = h; }
```

Kann man jedes der beiden von copy-in/copy-out unterscheiden?

Aufgaben zu Parameter-Modi (III)

1. Die Fakultäts-Funktion in ECMA-Script

   ```
   function f(x) { return x==0 ? 1 : x * f(x-1) }
   f(4)
   ==> 24
   ```

2. Die Verzweigung als Funktion

   ```
   function ite(b,j,n) { return b ? j : n }
   ite(false,2,3)
   ==> 3
   ```

3. Ersetzen Sie `?:` in `f` durch `ite`, werten Sie `f(4)` aus, erklären Sie Ihre Beobachtung.
4. Simulation von call-by-name durch Unterprogramme als Argumente:

   ```
   function ite(b,j,n) { return b ? j() : n() }
   ```

   wie muß `ite(false,2,3)` jetzt aussehen?
5. passen Sie die Def. von `f` an und testen Sie

Lokale Unterprogramme

- Unterprogramme sind wichtiges Mittel zur Abstraktion, das möchte man überall einsetzen
- also sind auch lokale Unterprogramme wünschenswert (die Konzepte *Block* (mit lokalen Deklarationen) und *Unterprogramm* sollen *orthogonal* sein)

```
int f (int x) {
  int g (int y) { return y + 1; }
  return g (g (x));
}
```

UP und Sichtbarkeit von Namen

```
{ const x = 3;
  function step(y) { return x + y; }
  for (const z of [ 1,2,4 ]) {
    console.log(step(z+1)); } }
```

- was ist die Ausgabe dieses Programms?
- was ändert sich bei Umbenennung von `z` zu `x`?
- Antwort: nichts! — der Funktionskörper (`x+y`) wird in seiner Definitionsumgebung ausgewertet, nicht in seiner Aufruf-Umgebung.

vgl. Spezifikation: `https://tc39.github.io/ecma262/#sec-lexical-environments`, `https://www.ecma-international.org/ecma-262/7.0/`

Frames, Ketten, Indizes

Während ein Unterprogramm rechnet, stehen seine lokalen Daten in einem Aktivationsverbund (Frame).

- Jeder Frame hat zwei Vorgänger:
  - dynamischer Vorgänger:
    (Frame des *aufrufenden* UP) benutzt zum Rückkehren
  - statischer Vorgänger
    (Frame des *textuell umgebenden* UP)
    benutzt zum Zugriff auf “fremde” lokale Variablen
- Jeder Variablenzugriff hat *Index-Paar* (i, j):

  im i-ten statischen Vorgänger der Eintrag Nr. j

  lokale Variablen des aktuellen UP: Index $(0, j)$

Indizes werden *statisch* bestimmt, Frames zur Laufzeit.

Lokale Unterprogramme: Beispiel

```
with Ada.Text_Io; use Ada.Text_Io;
procedure Nested is
 function F (X: Integer; Y: Integer)
 return Integer is
  function G (Y: Integer) return Integer is
  begin
   if (Y > 0) then return 1 + G(Y-1);
   else return X; end if;
  end G;
 begin return G (Y); end F;
begin
 Put_Line (Integer'Image (F(3,2)));
end Nested;
```

Flache Unterprogramme (C)

Entwurfs-Entscheidung für C:

- jedes Unterprogramm ist global

Folgerung:

- leichte Implementierung:
  - dynamischer Vorgänger = der vorige Frame (auf dem Stack)
  - statischer Vorgänger: gibt es nicht
- softwaretechnische Nachteile:
  globale Abstraktionen machen Programm unübersichtlich.

Lokale Unterprogramme in C# und Java

- in funktionalen Programmiersprachen (LISP, ML, Haskell, JavaScript)

```
(function(f){return f(f(0))})
  (function(x){return x+1})
```

- C#, Java 8

```
int x = 3; Func<int,int> f = y => x + y;
Console.WriteLine (f(4));
```

```
int x = 3; Function<Integer,Integer> f = y -> x + y;
System.out.println (f.apply(4));
```

Unterprogramme als Argumente

```
static int d ( Func<int,int> g ) {
    return g(g(1));                }
static int p (int x) {
    Func<int,int> f = y => x + y;
    return d (f);                  }
```

Betrachte Aufruf $p(3)$.
Das innere Unterprogramm *f* muß auf den *p*-Frame zugreifen, um den richtigen Wert des *x* zu finden.
Dazu *Closure* konstruieren: *f* mit statischem Vorgänger.
Wenn Unterprogramme als Argumente übergeben werden, steht der statische Vorgänger im Stack.
(ansonsten muß man den Vorgänger-Frame auf andere Weise retten, siehe später)

Unterprogramme als Resultate

```
static int x = 3;
static Func<int,int> s (int y) {
    return z => x + y + z;
}
static void Main () {
    Func<int,int> p = s(4);
    Console.WriteLine (p(3));
}
```

Wenn die von $s(4)$ konstruierte Funktion p aufgerufen wird, dann wird der s-Frame benötigt, steht aber nicht mehr im Stack.
$\Rightarrow$ Die (Frames in den) Closures müssen im Heap verwaltet werden.

Lokale anonyme Unterprogramme

- ```
  int [] x = { 1,0,0,1,0 };
  Console.WriteLine
      (x.Aggregate (0, (a, b) => 2*a + b));
  ```
  http://code.msdn.microsoft.com/LINQ-Aggregate-Operators-c51b3869
- ```
  foldl ( \ a b -> 2*a + b) 0 [1,0,0,1,0]
  ```
  Haskell (http://haskell.org/)

historische Schreibweise: $\lambda ab.2a + b$
(Alonzo Church: The Calculi of Lambda Conversion, 1941)
vgl. Henk Barendregt: The Impact of the Lambda Calculus, 1997, ftp://ftp.cs.ru.nl/pub/CompMath.Found/church.ps

Lokale Klassen (Java)

- static nested class: dient lediglich zur Gruppierung

  `class C { static class D { .. } .. }`

- nested inner class:

  `class C { class D { .. } .. }`

  jedes D-Objekt hat einen Verweis auf ein C-Objekt ($\approx$ statische Kette) (bezeichnet durch `C.this`)

- local inner class: (Zugriff auf lokale Variablen in *m* nur, wenn diese final sind. Warum?)

  `class C { void m () { class D { .. } .. } }`

Lokale Funktionen in Java 8

```
interface Function<T,R> { R apply(T t); }
```

bisher (Java $\leqslant 7$):

```
Function<Integer,Integer> f =
    new Function<Integer,Integer> () {
        public Integer apply (Integer x) {
            return x*x;
    } } ;
System.out.println (f.apply(4));
```

jetzt (Java 8): verkürzte Notation (Lambda-Ausdruck) für Implementierung *funktionaler Interfaces*

```
Function<Integer,Integer> g = x -> x*x;
System.out.println (g.apply(4));
```

Anwendung u.a. in `java.util.stream.Stream<T>`

Unterprogramme/Zusammenfassung

in prozeduralen Sprachen:

- falls alle UP global: dynamische Kette reicht
- lokale UP: benötigt auch statische Kette
- lokale UP as Daten: benötigt Closures
  $=$ (Code, statischer Link)
- UP als Argumente: Closures auf Stack
- UP als Resultate: Closures im Heap

in objektorientierten Sprachen: ähnliche Überlegungen bei lokalen (inner, nested) Klassen.

Übersicht

poly-morph = viel-gestaltig; ein Bezeichner (z. B. Unterprogramm-Name) mit mehreren Bedeutungen
Arten der Polymorphie:

- statische P.
  (Bedeutung wird zur Übersetzungszeit festgelegt):
  - ad-hoc: Überladen von Bezeichnern
  - generisch: Bezeichner mit Typ-Parametern
- dynamische P. (Bedeutung wird zur Laufzeit festgelegt):
  - Implementieren (Überschreiben) von Methoden

Ad-Hoc-Polymorphie

- ein Bezeichner ist *überladen*, wenn er mehrere (gleichzeitig sichtbare) Deklarationen hat
- bei jeder Benutzung des Bezeichners wird die Überladung dadurch *aufgelöst*, daß die Deklaration mit dem jeweils (ad-hoc) passenden Typ ausgewählt wird

Beispiel: Überladung im Argumenttyp:

```
static void p (int x, int    y) { ... }
static void p (int x, String y) { ... }
p (3, 4); p (3, "foo");
```

keine Überladung nur in Resultattyp, denn...

```
static int    f (boolean b) { ... }
static String f (boolean b) { ... }
```

Typhierarchie als Halbordnung

Durch `extends/implements` entsteht Halbordnung auf Typen, Bsp.
`class C;class D extends C;class E extends C`
definiert Relation $(\leqslant) = \{(C, C), (D, C), (D, D), (E, C), (E, E)\}$
auf $T = \{C, D, E\}$

dadurch entsteht Halbordnung auf Methoden-Signaturen (Tupel der Argument-Typen, *ohne* Resultat-Typ)
Bsp: Relation $\leqslant^2$ auf T^2:
$(t_1, t_2) \leqslant^2 (t_1', t_2') :\iff t_1 \leqslant t_1' \wedge t_2 \leqslant t_2'$
es gilt $(D, D) \leqslant^2 (C, C)$; $(D, D) \leqslant^2 (C, D)$;
$(C, D) \leqslant^2 (C, C)$; $(E, C) \leqslant^2 (C, C)$.

Ad-Hoc-Polymorphie und Typhierarchie

Auflösung von `p (new D(), new D())` bzgl.

```
static void p (C x, D y);
static void p (C x, C y);
static void p (E x, C y);
```

- bestimme die Menge P der zum Aufruf *passenden* Methoden
  (für diese gilt: statischer Typ der Argumente $\leqslant^n$ statischer Typ der formalen Parameter)
- bestimme die Menge M der minimalen Elemente von P
  (Def: m ist minimal falls $\neg\exists p \in P : p < m$)
- M muß eine Einermenge sein, sonst ist Überladung nicht auflösbar

Generische Polymorphie

parametrische Polymorphie:

- Klassen und Methoden können Typ-Parameter erhalten.
- innerhalb der Implementierung der Klasse/Methode wird der formale Typ-Parameter als (unbekannter) Typ behandelt
- bei der Benutzung der Klasse/Methode müssen alle Typ-Argumente angegeben werden
  (oder der Compiler inferiert diese in einigen Fällen)
- separate Kompilation (auch von generischen Klassen) mit statischer Typprüfung

Bsp: Generische Methode in C#

```
class C {
   static T id<T> (T x) { return x; }
}
```

beachte Position(en) von

- *Deklaration* des Typparameters
- *Benutzungen* des Typparameters

```
string foo = C.id<string> ("foo");
int    bar = C.id<int>    (42);
```

- *Instanziierung* des Typparameters

Bsp: Generische Klasse in Java

```
class Pair<A,B> {
  final A first; final B second;
  Pair(A a, B b)
    { this.first = a; this.second = b; }
}
Pair<String,Integer> p =
    new Pair<String,Integer>("foo", 42);
int x = p.second + 3;
```

vor allem für Container-Typen (Liste, Menge, Keller, Schlange, Baum, ...)

Bsp: Generische Methode in Java

- *Deklaration* des Typparameters
- *Benutzungen* des Typparameters

```
class C {
  static <A,B> Pair<B,A> swap (Pair<A,B> p) {
    return new Pair<B,A>(p.second, p.first); } }
```

- *Benutzungen* des Typparameters

```
Pair<String,Integer> p =
    new Pair<String,Integer>("foo", 42);
Pair<Integer,String> q =
    C.<String,Integer>swap(p);
```

Typargumente können auch *inferiert* werden:

```
Pair<Integer,String> q = C.swap(p);
```

Generische Fkt. höherer Ordg.

- Ziele:
  - Flexibilität (nachnutzbarer Code)
  - statische Typsicherheit
  - Effizienz (Laufzeit)
- wichtige Anwendung: Abstraktionen über den Programmablauf, z.B. für parallele Ausführung, Bsp:

```
public static TAccumulate Aggregate<TSource, TAccumul
  this ParallelQuery<TSource> source,
  TAccumulate seed,
  Func<TAccumulate,TSource,TAccumulate> func )
```

Bsp. Generische Fkt. höherer Ordg. (I)

Sortieren mit Vergleichsfunktion als Parameter

```
using System; class Bubble {
  static void Sort<T>
    (Func<T,T,bool> Less, T [] a) { ...
      if (Less (a[j+1],a[j])) { ... } }
  public static void Main (string [] argv) {
    int [] a = { 4,1,2,3 };
    Sort<int> ((int x, int y) => x <= y, a);
    foreach (var x in a) Console.Write (x);
} }
```

Ü: (allgemeinster) Typ und Implementierung einer Funktion `Flip`, die den Vergleich umkehrt:

```
Sort<int> (Flip( (x,y)=> x <= y ), a)
```

Bsp. Generische Fkt. höherer Ordg. (II)

„bulk operations“ auf Collections, z.B.

- Bibliothek `https://hackage.haskell.org/package/containers-0.5.9.1/docs/Data-Map-Strict.html`
- Beispiel:

```
intersectionWith
    :: Ord k
    => (a -> b -> c)
    -> Map k a -> Map k b -> Map k c
```

- ist effizienter als Iteration über alle Elemente eines Arguments

Übersicht

poly-morph = viel-gestaltig; ein Bezeichner (z. B. Unterprogramm-Name) mit mehreren Bedeutungen
Arten der Polymorphie:

- statische P.
  (Bedeutung wird zur Übersetzungszeit festgelegt):
  - ad-hoc: Überladen von Bezeichnern
  - generisch: Bezeichner mit Typ-Parametern
- dynamische P. (Bedeutung wird zur Laufzeit festgelegt):
  - Implementieren (Überschreiben) von Methoden

Objekte, Methoden

Motivation: Objekt = Daten + Verhalten.
Einfachste Implementierung:

- Objekt ist Record,
- einige Komponenten sind Unterprogramme.

```
typedef struct {
    int x; int y; // Daten
    void (*print) (FILE *fp); // Verhalten
} point;
point *p; ... ; (*(p->print))(stdout);
```

Anwendung: Datei-Objekte in UNIX (seit 1970)
(Merksatz 1: all the world is a file) (Merksatz 2: those who do not know UNIX are doomed to re-invent it, poorly)

Objektbasierte Sprachen (JavaScript)

(d. h. objektorientiert, aber ohne Klassen)
Objekte, Attribute, Methoden:

```
var o = { a : 3,
  m : function (x) { return x + this.a; } };
```

Vererbung zwischen Objekten:

```
var p = { __proto__ : o };
```

Attribut (/Methode) im Objekt nicht gefunden $\Rightarrow$ weitersuchen im Prototyp $\Rightarrow$ . . . Prototyp des Prototyps . . .
Übung: Überschreiben

```
p.m = function (x) { return x + 2*this.a }
var q = { __proto__ : p }
q.a = 4
alert (q.m(5))
```

Klassenbasierte Sprachen

gemeinsame Datenform und Verhalten von Objekten

```
typedef struct { int (*method[5])(); } cls;
typedef struct {
    cls * c;
} obj;
obj *o; ... (*(o->c->method[3]))();
```

allgemein: Klasse:

- Deklaration von Daten (Attributen)
- Deklaration und Implementierung von Methoden

Objekt:

- tatsächliche Daten (Attribute)
- Verweis auf Klasse (Methodentabelle)

this

Motivation: Methode soll wissen, für welches Argument sie gerufen wurde

```
typedef struct { int (*method[5])(obj *o);
} cls;
typedef struct {
    int data [3]; // Daten des Objekts
    cls *c; // Zeiger auf Klasse
} obj;
obj *o; ... (*(o->c->method[3]))(o);
int sum (obj *this) {
    return this->data[0] + this->data[1]; }
```

jede Methode bekommt *this* als erstes Argument
(in Java, C# geschieht das implizit)

Klassen in ECMA-Script

- syntaktische Hilfen zur Notation der objekt(prototyp)-basierten Vererbung, seit Version 6 (2015)
- ```
  class C {
    constructor(x) { this.x=x }
    m (y) { return this.x + y } }
  let p = new C(8)
  p.m(3)
  ```
- Definition siehe `https://www.ecma-international.org/ecma-262/7.0/#sec-class-definitions`
  ```

Vererbung

Def: Klasse *D ist abgeleitet von* Klasses *C*:

- *D* kann Menge der Attribute- und Methodendeklarationen von *C* erweitern (aber nicht verkleinern oder ändern)
- *D* kann Implementierungen von in *C* deklarierten Methoden übernehmen oder eigene festlegen (überschreiben).

Anwendung: dynamische Polymorphie

- Wo ein Objekt der Basisklasse erwartet wird (der *statische Typ* eines Bezeichners ist *C*),
- kann ein Objekt einer abgeleiteten Klasse (*D*) benutzt werden (der *dynamische Typ* des Wertes ist *D*).

Dynamische Polymorphie (Beispiel)

```
class C {
  int x = 2; int p () { return this.x + 3; }
}
C x = new C() ; int y = x.p ();
```

Überschreiben:

```
class E extends C {
  int p () { return this.x + 4; }
}
C x =              // statischer  Typ: C
      new E() ; // dynamischer Typ: E
int y = x.p ();
```

Vererbung bricht Kapselung

```
class C {
   void p () { ... q(); ...  };
   void q () { .. };
}
```

Jetzt wird `q` überschrieben (evtl. auch unabsichtlich—in Java), dadurch ändert sich das Verhalten von `p`.

```
class D extends C {
   void q () { ... }
}
```

Korrektheit von `D` abhängig von *Implementierung* von `C`
$\Rightarrow$ object-orientation is, by its very nature, anti-modular . . .
`http://existentialtype.wordpress.com/2011/03/15/teaching-fp-to-freshmen/`

Überschreiben und Überladen

- Überschreiben:
  zwei Klassen, je eine Methode mit gleichem Typ
- Überladen:
  eine Klasse, mehrere Methoden mit versch. Typen

- C++: Methoden, die man überschrieben darf, `virtual` deklarieren
- C#: Überschreiben durch `override` angezeigen,
- Java: alle Methoden sind virtual, deswegen ist Überschreiben von Überladen schlecht zu unterscheiden:
  Quelle von Programmierfehlern
- Java-IDEs unterstützen Annotation `@overrides`

Equals richtig implementieren

```
class C {
  final int x; final int y;
  C (int x, int y) { this.x = x; this.y = y; }
  int hashCode () { return this.x + 31 * this.y; }
}
```

nicht so:

```
  public boolean equals (C that) {
    return this.x == that.x && this.y == that.y;
  }
```

Equals richtig implementieren (II)

...sondern so:

```
public boolean equals (Object o) {
  if (! (o instanceof C)) return false;
  C that = (C) o;
  return this.x == that.x && this.y == that.y;
}
```

Die Methode `boolean equals(Object o)` wird aus HashSet aufgerufen.
Sie muß deswegen *überschrieben* werden.
Das `boolean equals (C that)` hat den Methodenamen nur *überladen*.

Statische Attribute und Methoden

für diese findet *kein* dynamischer Dispatch statt.
(Beispiele—Puzzle 48, 54)
Damit das klar ist, wird dieser Schreibstil empfohlen:

- dynamisch: immer mit Objektnamen qualifiziert, auch wenn dieser `this` lautet,
- statisch: immer mit Klassennamen qualifiziert (niemals mit Objektnamen)

Vererbung und generische Polym.

- mit Sprachkonzepte *Vererbung* ist Erweiterung des Sprachkonzeptes *Generizität* wünschenswert:
- beim Definition der Passung von parametrischen Typen sollte die Vererbungsrelation $\leqslant$ auf Typen berücksichtigt werden.

- Ansatz: wenn `E` $\leqslant$ `C`, dann auch `List<E>` $\leqslant$ `List<C>`
- ist *nicht* typsicher, siehe folgendes Beispiel
- Modifikation: ko- und kontravariante Typparameter

Generics und Subtypen

Warum geht das nicht:

```
class C { }

class E extends C { void m () { } }

List<E> x = new LinkedList<E>();

List<C> y = x; // Typfehler
```

Antwort: wenn das erlaubt wäre, dann:

Obere Schranken für Typparameter

- Java: `class<T extends S> { ... }`,
  C#: `class <T> where T : S { ... }`
  als Argument ist jeder Typ *T* erlaubt, der *S* implementiert

```
interface Comparable<T>
    { int compareTo(T x); }
static <T extends Comparable<T>>
    T max (Collection<T> c) { .. }
```

Untere Schranken für Typparameter

- Java: `<S super T>`
  Als Argument ist jeder Typ *S* erlaubt, der Obertyp von *T* ist.

```
static <T> int binarySearch
    (List<? extends T> list, T key,
    Comparator<? super T> c)
```

variante generische Interfaces (C#)

Kontravarianz (`in P`), Kovarianz (`out P`)

```
interface I<in P> { // kontravariante Dekl.
  // P get (); kovariante Benutzung (verboten)
  void set (P x); // kontravariante Benutzung
}
class K<P> : I<P> { public void set (P x) {} }
class C {} class E : C {} // E <= C
I<C> x = new K<C>();
I<E> y = x; // erlaubt, I<C> <= I<E>
```

- kontravariant: $E \leqslant C \Rightarrow I(E) \geqslant I(C)$
- kovariant: $E \leqslant C \Rightarrow I(E) \leqslant I(C)$
- invariant: $E \neq C \Rightarrow I(E) \not\leqslant I(C)$

Vergleich: Varianz und Schranken

Unterscheidung:

- bei Schranken geht es um die Instantiierung (die Wahl der Typargumente)
- bei Varianz um den erzeugten Typ (seine Zuweisungskompatibilität)

Generics und Arrays

das gibt keinen Typfehler:

```
class C { }
class E extends C { void m () { } }
E [] x = { new E (), new E () }; C [] y = x;
y [0] = new C (); x [0].m();
```

warum ist die Typprüfung für Arrays schwächer als für Collections?
Historische Gründe. Das sollte gehen:

```
void fill (Object[] a, Object x) { .. }
String [] a = new String [3];
fill (a, "foo");
```

Das sieht aber mit Generics besser so aus: . . .

Übung Polymorphie

- Aufgabe zu Funktion `Flip` (C#),
  Quelltext: `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws16/pps/code/kw55/`
- `binarySearch` aufrufen (Java), so daß beide `?` von `T` verschieden sind

Statisch typisiert $\Rightarrow$ sicher und effizient

- Programmtext soll *Absicht* des Programmierers ausdrücken.
- dazu gehören Annahmen über *Daten*, formuliert mittels *Typen* (`foo::Book`)
  ...alternative Formulierung:
  Namen (`fooBook`, Kommentar `foo // Book`)
- nur durch statische Typisierung kann man Absichten/Annahmen maschinell umfassend prüfen
  ...alternative Prüfung: Tests
- ist nützlich für Wiederverwendbarkeit
- ist nützlich für sichere und effiziente Ausführung

Statische Typisierung: für und wider

Für statische Typisierung spricht vieles.
Es funktioniert auch seit Jahrtzehnten (Algol 1960, ML 1970, C++ 1980, Java 1990 usw.)
Was dagegen?

- Typsystem ist ausdrucksschwach:
  (Bsp: keine polymorphen Container in C)
  Programmierer kann Absicht nicht ausdrücken
- Typsystem ist ausdrucksstark:
  (Bsp: kontravariante Typargumente in Java,C#)
  Programmierer muß Sprachstandard lesen und verstehen und dazu Konzepte (z.B. aus Vorlesung) kennen

Fachmännisches Programmieren

- Hardware: wer Flugzeug/Brücke/Staudamm/. . . baut, kann (und darf) das auch nicht allein nach etwas Selbststudium und mit Werkzeug aus dem Baumarkt
- Software: der (Bastel-)Prototyp wird oft zum Produkt, der Bastler zum selbsternannten Programmierer

so auch bei Programmiersprachen:
entworfen *von* oder *für* Leute ohne (viel) Fachwissen

- BASIC (1964) (Kemeny, Kurtz) to enable students in fields other than science and math. to use computers
- Perl (1987) (Larry Wall: Chemie, Musik, Linguistik)
- PHP (1994) (Rasmus Lerdorf) Personal Home Page Tools (like Perl but . . . simpler, more limited, less consistent.)

Legacy-Sprachen: ECMA-Script (Javascript)

semantisch ist das LISP (z.B. Funktionen als Daten),
syntaktisch ist es Java

- Motivation: Software soll beim Kunden laufen
- technisches Problem: Gerätebenutzer versteht/beherrscht seinen Computer/Betriebssystem nicht (z.B. kann oder will keine JRE installieren)
- stattdessen zwingt man Kunden auf Browser mit Javascript-Engine (der Browser ist das OS)
- das steckt z.B. Google viel Geld hinein:
  `https://code.google.com/p/v8/`

  aus verständlichen Gründen (Anzeige von Werbung)

Aktuelle Entwicklungen: JS

- ECMA-Script 6 übernimmt viele Konzepte moderner (funktionaler) Programmierung, u.a.
  - let (block scope), const (single assignment)
  - desctructuring (pattern matching)
  - tail calls (ohne Stack)

  `https://github.com/lukehoban/es6features`
- . . . was ist mit Microsoft? Die haben auch viel Geld und clevere Leute? — Ja:

  `http://www.typescriptlang.org/`

  TypeScript adds *optional types*, classes, and modules to JavaScript.

Personen: Luke Hoban, Anders Hejlsberg, Erik Meijer, . . .

Legacy-Sprachen: PHP

- Facebook ist (ursprünglich) in PHP implementiert
- deswegen steckt Facebook viel Geld in Technologien (a.k.a. Work-Arounds) für diese Sprache aus ebenfalls verständlichen Gründen :
  - für Kunden (d.h. Werbekunden): Antwortzeiten der Webserver
  - für Betreiber: Entwicklungs- und Betriebskosten

Aktuelle Entwicklungen: PHP

- HHVM: Hip Hop Virtual Machine
  `https://github.com/facebook/hhvm/blob/master/hphp/doc/bytecode.specification`
- Hack `http://hacklang.org/` Type Annotations, Generics, Nullable types, Collections, Lambdas, ...

Julien Verlaguet: *Facebook: Analyzing PHP statically*, 2013, `http://cufp.org/2013/julien-verlaguet-facebook-analyzing-php-statically.html`
vgl. Neil Savage: *Gradual Evolution*, Communications of the ACM, Vol. 57 No. 10, Pages 16-18, `http://cacm.acm.org/magazines/2014/10/178775-gradual-evolution/fulltext`

Aktuelly: Web Assembly

- a new portable, size- and load-time-efficient format suitable for compilation to the web.

  `http://webassembly.org/`
- d.h., Programme in vernünftigen (d.h. typsicheren) Sprachen schreiben (statt JS),

  nach WASM kompilieren und im Browser ausführen
- das gabe es alles schon? Natürlich:
  - Java → Bytecode (class files) (1996),
  - Pascal → P(ortable)-Code (1973)
- formale Spezifikation (typsichere Kellermaschine)

  `https://webassembly.github.io/spec/core/index.html`

Die Zukunft: Typen für Ressourcen

`https://www.rust-lang.org/`

. . . a systems programming language that . . . prevents segfaults and guarantees thread safety.

- jedes Datum hat genau einen *Eigentümer*,
  man kann Daten *übernehmen* und *ausborgen*,
- *statisch* garantiert: für jedes Datum `x:T` gibt es
  - one or more references (`&T`) to a resource,
  - exactly one mutable reference (`&mut T`).

  `https://doc.rust-lang.org/book/references-and-borrowing.html#the-rules`

Die Zukunft: Datenabhängige Typen

`https://idris-lang.org/` ...aspects of a program's behaviour can be specified *precisely* in the type.

- elementare Bausteine:
  - Daten: `42`, `"foo"`, `(x,y)=>x+y`, Typen: `bool`, `int`
- Kombinationen (Funktionen):
  - Datum → Datum, Bsp. `(x,y)=>x+y`
  - Typ → Typ, Bsp. `List<T>`
  - Typ → Datum, Bsp. `Collections.<String>sort()`
  - Datum → Type, *(data-)dependent type*, Bsp. Vektoren
    `data Vec : Nat -> Type -> Type`

```
(++) : Vec p a -> Vec q a -> Vec (p+q) a
head : Vect (S p) a -> a    --  S = Nachfolger
```

Nicht reguläre Typen

- ein rekursiver polymorpher Typ heißt *regulär*, wenn die Typ-Argumente bei Rekursion gleich bleiben
  `data List a = Nil | Cons a (List a)`,
  ...sonst *nicht regulär*
- `data List a b = Nil | Cons a (List b a)`
- `data Tree a = Leaf a | Branch (Tree (a,a))`
- Anwendung: Implementierung von
  `containers:Data.Sequence` `https://hackage.haskell.org/package/containers-0.5.9.1/docs/Data-Sequence.html#t:Seq`

Themen

- Methoden zur Beschreibung der
  - Syntax: reguläre Ausdrücke, kontextfreie Grammatiken
  - Semantik: operational, denotational, axiomatisch
- Konzepte:
  - Typen,
  - Namen (Deklarationen), Blöcke (Sichtbarkeitsbereiche)
  - Ausdrücke und Anweisungen (Wert und Wirkung),
  - Unterprogramme (als Daten)
  - Polymorphie (statisch, dynamisch)
- Wechselwirkungen der Konzepte
- Paradigmen: imperativ, funktional, objektorientiert

Sprachen kommen und gehen, Konzepte bleiben.

Wie weiter? (LV)

Anwendung und Vertiefung von Themen PPS z.B. in VL

- Programmverifikation
  u.a. axiomatische Semantik imperativer Programme
- Compilerbau
  - Realisierung der Semantik durch
    - Interpretation
    - Transformation
  - abstrakte und konkrete Syntax (Parser)
- Constraint-Programmierung
- Fortgeschrittene Konzepte von Programmiersprachen (OS)
- Symbolisches Rechnen (Transform. v. Syntaxbäumen)