

Prinzipien von Programmiersprachen

Vorlesung

Wintersemester 2007 – 2015

Johannes Waldmann, HTWK Leipzig

25. Januar 2016

– Typeset by FoilTeX –

Einleitung

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

– Typeset by FoilTeX –

1

Sprache

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

– Typeset by FoilTeX –

2

Konzepte

- Hierarchien (baumartige Strukturen)
 - zusammengesetzte (arithmetische, logische) Ausdrücke
 - zusammengesetzte Anweisungen (Blöcke)
 - Klassen, Module
- Typen beschreiben Daten
- Namen stehen für Werte, Wiederverwendung
- Flexibilität durch Parameter (Unterprogramme, Polymorphie)

– Typeset by FoilTeX –

3

Paradigmen

- imperativ
Programm ist Folge von Befehlen (= Zustandsänderungen)
- 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)

– Typeset by FoilTeX –

4

Ziele der LV

Arbeitsweise: Methoden, Konzepte, Paradigmen

- isoliert beschreiben
 - an Beispielen in (bekannten und unbekannt) Sprachen wiedererkennen
- Ziel:
- verbessert die Organisation des vorhandenen Wissens
 - gestattet die Beurteilung und das Erlernen neuer Sprachen
 - hilft bei Entwurf eigener (anwendungsspezifischer) Sprachen

– Typeset by FoilTeX –

5

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

– Typeset by FoilTeX –

6

Organisation

- Vorlesung
- Übungen (alle in Z423)
Übungsgruppe wählen: <https://autotool.imn.htwk-leipzig.de/shib/cgi-bin/Super.cgi>
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Klausur: 120 min, ohne Hilfsmittel

– Typeset by FoilTeX –

7

Literatur

- <http://www.imn.htwk-leipzig.de/~waldmann/edu/ws13/pps/folien/main/>
- 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?tttype=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
find workspace/ -name "*.java" -exec grep
```

2. Maschinenmodelle (Bsp: Register, Turing, Stack, Funktion)

funktionales Programmieren in Haskell

(<http://www.haskell.org/>)

ghci

```
:set +t
length $ takeWhile (== '0') $ reverse $ show
Kellermaschine in PostScript.
```

```
42 42 scale 7 9 translate .07 setlinewidth .
setgray}def 1 0 0 42 1 0 c 0 1 1{0 3 3 90 27
arcn 270 90 c -2 2 4{-6 moveto 0 12 rlineto}
9 0 rlineto}for stroke 0 0 3 1 1 0 c 180 rot
```

Mit gv oder kghostview ansehen (Options: watch file).

Mit Editor Quelltext ändern. Finden Sie den Autor dieses Programms!

(Lösung: John Tromp, siehe auch

<http://www.iwriteiam.nl/SigProgPS.html>)

3. <http://99-bottles-of-beer.net/> (top rated ...)

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
gcc -o bar bar.o # linkt (lädt) Objektcode (
./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 auf kain, dann gcc wie oben)

gcc für Java (gcj):

```
gcj -c Foo.java # erzeugt Objektcode
gcj -o Foo Foo.o --main=Foo # linken, wie ob
```

- Assemblercode ansehen, vergleichen

```
gcj -S Foo.java # erzeugt Assemblercode (Fo
```

- Kompatibilität des Bytecodes ausprobieren zwischen Sun-Java und GCJ (beide Richtungen)

```
gcj -C Foo.java # erzeugt Class-File (Foo.c
```

Syntax von Programmiersprachen

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 Knoten des Baums stehen Token,
- jedes Token hat einen Typ und einen Inhalt (eine Zeichenkette).
- diese Prinzip kommt aus der Mathematik (arithmetische Ausdrücke, logische Formeln)

Token-Typen

Token-Typen sind üblicherweise

- reservierte Wörter (if, while, class, ...)
- Bezeichner (foo, bar, ...)
- Literale für ganze Zahlen, Gleitkommazahlen, Strings, Zeichen
- Trennzeichen (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-Konstanten in \mathbb{C} .

Spezifikation formaler Sprachen

man kann eine formale Sprache beschreiben durch:

- *algebraisch* (Sprach-Operationen)
Bsp: reguläre Ausdrücke
- *Generieren* (Grammatik), Bsp: kontextfreie Grammatik,
- *Akzeptanz* (Automat), Bsp: Kellerautomat,
- *logisch* (Eigenschaften),
 $\left\{ w \mid \forall p, r : \left(\begin{array}{l} p < r \wedge w[p] = a \wedge w[r] = c \\ \Rightarrow \exists q : (p < q \wedge q < r \wedge w[q] = b) \end{array} \right) \right\}$

Sprach-Operationen

Aus Sprachen L_1, L_2 konstruiere:

- Mengenoperationen
 - Vereinigung $L_1 \cup L_2$,
 - Durchschnitt $L_1 \cap L_2$, Differenz $L_1 \setminus L_2$;
- 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) $L_1^* = \bigcup_{k \geq 0} L_1^k$

Def: Sprache *regulär* : \iff 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 $\emptyset \in E$ (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/regexpl.html>)

Bemerkung zu Reg. Ausdr.

Wie beweist man $w \in L(X)$?

(Wort w gehört zur Sprache eines regulären Ausdrucks X)

- wenn $X = X_1 + X_2$:
beweise $w \in L(X_1)$ *oder* beweise $w \in L(X_2)$
- wenn $X = X_1 \cdot X_2$:
zerlege $w = w_1 \cdot w_2$ *und* beweise $w_1 \in L(X_1)$ *und* beweise $w_2 \in L(X_2)$.
- wenn $X = X_1^*$:
wähle einen Exponenten $k \in \mathbb{N}$ *und* beweise $w \in L(X_1^k)$
(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 $\{0, 1, \dots, 10\}$,
 - auf Wörtern $\{a, b\}^{\leq 2}$
- $(\text{Pow}(\Sigma^*), \cup, \cdot, \dots)$ 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#6.1Lexicaltokens>

Welche Notation wird für unsere Operatoren $+$ und Stern benutzt? Was bedeuten die eckigen Klammern?

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:
 $R_1 = \{1000 \rightarrow 0001110\}, R_2 = \{aabb \rightarrow bbaaa\}$?

Grammatiken

Grammatik G besteht aus:

- Terminal-Alphabet Σ (üblich: Kleinbuchst., Ziffern)

Grammatik	{ terminale
	= mkSet "abc"
	, variablen
	= mkSet "SA"
	, start = 'S'
	, regeln = mkSet
	[("S", "abc")
	, ("ab", "aabbA")
	, ("Ab", "bA")
	, ("Ac", "cc")
]
 - Variablen-Alphabet V (üblich: Großbuchstaben)
 - Startsymbol $S \in V$
 - Regelmenge (Wort-Ersetzungs-System)

$R \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$
--
- von G erzeugte Sprache: $L(G) = \{w \mid S \rightarrow_R^* w \wedge w \in \Sigma^*\}.$

Formale Sprachen: Chomsky-Hierarchie

- (Typ 0) abzählbare 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 $L = L(X)$,
- es gibt eine Typ-3-Grammatik G mit $L = L(G)$,
- es gibt einen endlichen Automaten A mit $L = L(A)$.

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 $\leq q$.

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 \rightarrow Bezeichner = Ausdruck
 | if Ausdruck then Anweisung else Anweis
 Ausdruck \rightarrow Bezeichner | Literal
 | Ausdruck Operator Ausdruck

Bsp: korrekt geklammerte Ausdrücke:

$$G = (\{a, b\}, \{S\}, S, \{S \rightarrow aSbS, S \rightarrow \epsilon\}).$$

Bsp: Palindrome:

$$G = (\{a, b\}, \{S\}, S, \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \epsilon\}).$$

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

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

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

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

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

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

$D \subseteq L(G)$ Induktion über Wortlänge

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

$\langle \text{assignment} \rangle \rightarrow \langle \text{variable} \rangle = \langle \text{expression} \rangle$
 $\langle \text{number} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{number} \rangle \mid \langle \text{digit} \rangle$

Erweiterte BNF

- Wiederholungen (Stern, Plus) $\langle \text{digit} \rangle^+$
- Auslassungen
 $\text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ [else } \langle \text{stmt} \rangle \text{]}$

kann in BNF übersetzt werden

Ableitungsbäume für CF-Sprachen

Def: ein geordneter Baum T mit Markierung $m : T \rightarrow \Sigma \cup \{\epsilon\} \cup V$ 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, \dots, k_n gilt $(m(k), m(k_1)m(k_2)\dots m(k_n)) \in R(G)$ (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:

$\text{rand}(k) = \text{rand}(k_1) \text{rand}(k_2) \dots \text{rand}(k_n)$

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 $\{S \rightarrow aSb \mid SS\epsilon\}$ eindeutig?

(beachte: mehrere Ableitungen $S \rightarrow_R^* w$ sind erlaubt, und wg. Kontextfreiheit auch gar nicht zu vermeiden.)

Die naheliegende Grammatik für arith. Ausdr.

$\text{expr} \rightarrow \text{number} \mid \text{expr} + \text{expr} \mid \text{expr} * \text{expr}$
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 \rightarrow Zahl \mid Ausdruck + Ausdruck

ersetzen durch

Ausdruck \rightarrow Summe

Summe \rightarrow Summand \mid Summe + Summand

Summand \rightarrow Zahl

Präzedenzen

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

Grammatik-Regel

summand \rightarrow zahl

erweitern zu

summand \rightarrow zahl \mid produkt

produkt \rightarrow ...

(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

– Typeset by FoilTeX –

ld

Übung Operator/Grammatik

Übung:

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

– Typeset by FoilTeX –

ld

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>
```

– Typeset by FoilTeX –

ld

Semantik von Programmiersprachen

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

– Typeset by FoilTeX –

ld

Attributgrammatiken (I)

- Attribut: Annotation an Knoten des Syntaxbaums.

$A : \text{Knotenmenge} \rightarrow \text{Attributwerte}$ (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
 $(A(X_0), A(X_1), \dots, A(X_n))$
für Knoten X_0 mit Kindern $[X_1, \dots, X_n]$

$S \rightarrow mSS, A(X_0) + A(X_3) = A(X_2);$

$S \rightarrow e, A(X_0) = A(X_1);$

Terminale: $A(e) = 1, A(m) = 0$

– Typeset by FoilTeX –

ld

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)

– Typeset by FoilTeX –

ld

Donald E. Knuth

- The Art Of Computer Programming (1968, ...)
(Band 3: Sortieren und Suchen)

- T_EX, Metafont, Literate Programming (1983, ...)
(Leslie Lamport: L^AT_EX)

- Attribut-Grammatiken

- die Bezeichnung „NP-vollständig“

• ...

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

– Typeset by FoilTeX –

ld

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.

– Typeset by FoilTeX –

ld

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

```
String x = "foo"; String y = "bar";
Boolean.toString(x.length() < y.length());
(Curry-Howard-Isomorphie)
```

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 { -> attr
```

- Quelltexte:

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

Benutzung:

```
cd pps-ws15/rechner ; make ; make test ; ma
(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 Attributierung der AST-Knoten durch Zahlen
- Hauptprogramm in `Interpreter.java`
- bauen, testen, aufräumen: siehe `Makefile`

– generierte Dateien in `rechner/*`

- Aufgaben:

Multiplikation, Subtraktion, Klammern, Potenzen

Kommentar: in Java fehlen: algebraische Datentypen, Pattern Matching, Funktionen höherer Ordnung. Deswegen muß SableCC das simulieren — das sieht nicht schön aus. Die „richtige“ Lösung sehen Sie später im Compilerbau.

Abstrakter Syntaxbaum, Interpreter:

```
http://www.imn.htwk-leipzig.de/~waldmann/
edu/ws11/cb/folien/main/node12.html,
```

Kombinator-Parser:

```
http://www.imn.htwk-leipzig.de/~waldmann/
edu/ws11/cb/folien/main/node70.html
```

Dynamische Semantik

- operational:

beschreibt Wirkung von Anweisungen durch Änderung des Programmzustandes

- 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); \circ ;
- Ausführung eines binären Operators \circ :
 $x \leftarrow \text{pop}; y \leftarrow \text{pop}; \text{push } (x \circ y)$;

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 Speicherzustand nach Speicherzustand

Vorteile denotationaler Semantik:

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

Vorteil deklarativer Programmierung:

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 Programmzustände:

$\{ V \} A \{ N \}$

- für jeden Zustand s , in dem Vorbedingung V gilt:
- wenn Anweisung A ausgeführt wird,
- und Zustand t erreicht wird, dann gilt dort Nachbedingung N

Beispiel:

$\{ x \geq 5 \} y := x + 3 \{ y \geq 7 \}$

Gültigkeit solcher Aussagen kann man

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

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 $V' \Rightarrow V$ und $N \Rightarrow N'$
dann $\{ V' \} A \{ N' \}$
- wenn $\{ V$ und $B \} C \{ N \}$
und $\{ V$ und not $B \} D \{ N \}$
dann $\{ V \} \text{if } (B) \text{ then } C \text{ else } D \{ N \}$
- 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  
2: iload_1  
3: ifle          17  
6: iload_2  
7: iload_0  
8: imul  
9: istore_2  
10: iload_1  
11: iconst_1  
12: isub  
13: istore_1
```

```
14: goto          2  
17: iload_2  
18: ireturn
```

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
```

Typen

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 int)
- vorgegebene Typen (Fortran: Integer, Real, Arrays)
- benutzerdefinierte Typen
(algebraische Datentypen;
Spezialfälle: enum, struct, class)
- abstrakte Datentypen (interface)

Überblick

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

Aufzählungstypen

können einer Teilmenge ganzer Zahlen zugeordnet werden

- vorgegeben: int, char, boolean
- nutzerdefiniert (enum)

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

Designfragen:

- automatisch nach int umgewandelt?
- automatisch von int umgewandelt?
- eine Konstante in mehreren Aufzählungen möglich?

Keine Aufzählungstypen

das ist nett gemeint, aber vergeblich:

```
#define Mon 0
#define Tue 1
...
#define Sun 6

typedef int day;

int main () {
    day x = Sat;
    day y = x * x;
}
```

Aufzählungstypen in C

im wesentlichen genauso nutzlos:

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

int main () {
    day x = Sat;
    day y = x * x;
}
```

Übung: was ist in C++ besser?

Aufzählungstypen in Java

```
enum Day {
    Mon, Tue, Wed, Thu, Fri, Sat, Sun;

    public static void main (String [] argv)
        for (Day d : Day.values ()) {
            System.out.println (d);
        }
}
```

verhält sich wie Klasse

(genauer: Schnittstelle mit 7 Implementierungen)

siehe Übung (jetzt oder bei Objekten)

Teilbereichstypen in Ada

```
with Ada.Text_IO;
procedure Day is
    type Day is ( Mon, Tue, Thu, Fri, Sat, Sun );
    subtype Weekday is Day range Mon .. Fri;
    X, Y : Day;
begin
    X := Fri;      Ada.Text_IO.Put (Day'Image(X));
    Y := Day'Succ(X); Ada.Text_IO.Put (Day'Image(Y));
end Day;
```

mit Bereichsprüfung bei jeder Zuweisung.

einige Tests können aber vom Compiler statisch ausgeführt werden!

Abgeleitete Typen in Ada

```
procedure Fruit is
    subtype Natural is
        Integer range 0 .. Integer'Last;
    type Apples is new Natural;
    type Oranges is new Natural;
    A : Apples; O : Oranges; I : Integer;
begin -- nicht alles korrekt:
    A := 4; O := A + 1; I := A * A;
end Fruit;
```

Natural, Äpfel und Orangen sind isomorph, aber nicht zuweisungskompatibel.

Sonderfall: Zahlenkonstanten gehören zu jedem abgeleiteten Typ.

Zusammengesetzte Typen

Typ = Menge, Zusammensetzung = Mengenoperation:

- Produkt (record, struct)
- Summe (union, case class)
- Rekursion
- Potenz (Funktion)

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 (≤ 1960)

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;
}
```

– Typeset by FoilTeX –

ld

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() }
```

– Typeset by FoilTeX –

ld

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>

– Typeset by FoilTeX –

ld

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)

– Typeset by FoilTeX –

ld

Potenz-Typen

$B^A := \{f : A \rightarrow B\}$ (Menge aller Funktionen von A nach B)

ist sinnvolle Notation, denn $|B|^{|A|} = |B^A|$

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.

– Typeset by FoilTeX –

ld

Felder (Arrays)

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?

– Typeset by FoilTeX –

ld

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))
```

– Typeset by FoilTeX –

ld

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.

– Typeset by FoilTeX –

ld

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 [][].)
```

Dynamische Feldgrößen

Designfrage: kann ein Feld (auch: String) seine Größe ändern?

(C: wird sowieso nicht geprüft, Java: nein, Perl: ja)

in Java: wenn man das will, dann will man statt Array eine LinkedList, statt String einen StringBuffer.

wenn man mit Strings arbeitet, dann ist es meist ein Fehler:

benutze Strings *zwischen* Programmen,
aber niemals *innerhalb* eines Programms.

ein einem Programm: benutze immer
anwendungsspezifische Datentypen.

... deren externe Syntax spiel überhaupt keine Rolle

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

Übung Typen

- Teilbereichstypen und abgeleitete Typen in Ada (Vergleich mit dimensionierten Typen in F#)
- Arrays in C (Assemblercode anschauen)
- rechteckige und geschachtelte Arrays in C#
- Wert/Verweis (struct/class) in C#

Bezeichner, Bindungen, Bereiche

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

$$e = \sum_{n \geq 0} \frac{1}{n!}, \quad \sin = (x \mapsto \sum_{n \geq 0} (-1)^n \frac{x^{2n+1}}{(2n+1)!})$$

- 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 Initialisierung

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

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

Ausdrücke

Einleitung

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

Vgl. Trennung (in Pascal, Ada)

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

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)

Designfragen für Ausdrücke

- Präzedenzen (Vorrang)
- Assoziativitäten (Gruppierung)
- Ausdrücke dürfen (Neben-)Wirkungen haben?
- in welcher Reihenfolge treten die auf?
- welche impliziten Typumwandlungen?
- explizite Typumwandlungen (cast)?
- kann Programmierer Operatoren definieren? überladen?

– Typeset by FoilTeX –

ld

Syntax von Ausdrücken

- einfache Ausdrücke : Konstante, Variable
- zusammengesetzte Ausdrücke:
 - Operator-Symbol zwischen Argumenten
 - Funktions-Symbol vor Argument-Tupel

wichtige Spezialfälle für Operatoren:

- arithmetische, relationale, boolesche

Wdhlg: Syntaxbaum, Präzedenz, Assoziativität.

– Typeset by FoilTeX –

ld

Syntax von Konstanten

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.

– Typeset by FoilTeX –

ld

Der Plus-Operator in Java

... addiert Zahlen und verkettet Strings.

```
System.out.println ("foo" + 3 + 4);
```

```
System.out.println (3 + 4 + "bar");
```

– Typeset by FoilTeX –

ld

Überladene Operatornamen

aus praktischen Gründen sind arithmetische und relationale Operatornamen *überladen*

(d. h.: ein Name für mehrere Bedeutungen)

Überladung wird aufgelöst durch die Typen der Argumente.

```
int x = 3; int y = 4; ... x + y ...
```

```
double a; double b; ... a + b ...
```

```
String p; String q; ... p + q ...
```

– Typeset by FoilTeX –

ld

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:

$(x :: \text{int}) + (y :: \text{float})$

beide Argumente werden zu kleinstem gemeinsamen Obertyp promoviert, falls dieser eindeutig ist (sonst statischer Typfehler)

(Halbordnung \rightarrow Halbverband)

– Typeset by FoilTeX –

ld

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.

– Typeset by FoilTeX –

ld

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), Repräsentation ändert sich nicht:

```
List books;
```

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

... kommt nur vor, wenn man die falsche Programmiersprache benutzt (nämlich Java vor 1.5)

– Typeset by FoilTeX –

ld

Typ-Umwandlungen in Javascript

Gary Bernhardt: WAT (2012)

<https://www.destroyallsoftware.com/talks/wat>

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

– Typeset by FoilTeX –

ld

Der Verzweigungs-Operator

Absicht: statt

lieber

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

– Typeset by FoilTeX –

ld

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;  
}
```

– Typeset by FoilTeX –

ld

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");  
}
```

– Typeset by FoilTeX –

ld

Logische (Boolesche) Ausdrücke

- und &&, || oder, nicht !, gleich, ungleich, kleiner, ...
- 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?)

– Typeset by FoilTeX –

ld

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.

– Typeset by FoilTeX –

ld

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

– Typeset by FoilTeX –

ld

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

– Typeset by FoilTeX –

ld

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);
```

- 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 machen, was er will)

```
int x = 3; int y = ++x + ++x + ++x;
```

vgl. Aussagen zu *sequence points* in
<http://gcc.gnu.org/readings.html>
Gurevich, Huggins: *Semantics of C*,

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.6755>

Anweisungen(I)

Definition

Semantik: Anweisung hat *Wirkung* (Zustandsänderung), die bei Ausführung eintritt.

abstrakte Syntax:

- einfache Anweisung:
 - Zuweisung
 - 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)
strukturierte Programmierung:

- Nacheinander
- außer der Reihe (Sprung, Unterprogramm, Exception)
- Verzweigung
- Wiederholung

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: Deklarationen gestattet
- am Beginn des (Unter-)Programms (Pascal)
 - am Beginn des Blocks (C)
 - an jeder Stelle des 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

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

Designfragen:

- welche Typen für e ?
- welche Werte für c_i ?
- Wertebereiche?
- was passiert, wenn mehrere Fälle zutreffen?
- was passiert, wenn kein Fall zutrifft (default?)
- (effiziente Kompilation?)

– Typeset by FoilTeX –

ld

Switch/break

das macht eben in C, C++, Java nicht das, was man denkt:

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

C#: jeder Fall *muß* mit break (oder goto) enden.

– Typeset by FoilTeX –

ld

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

– Typeset by FoilTeX –

ld

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)
  }
}
```

– Typeset by FoilTeX –

ld

Anweisungen(II)

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)

– Typeset by FoilTeX –

ld

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)

– Typeset by FoilTeX –

ld

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

– Typeset by FoilTeX –

ld

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.

– Typeset by FoilTeX –

ld

Datengesteuerte Schleifen

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

– Typeset by FoilTeX –

ld

Zustandgesteuerte 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 ...) { ... }
```

(siehe Unterschiede ...)

– Typeset by FoilTeX –

ld

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);
        } } }
```

– Typeset by FoilTeX –

ld

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

– Typeset by FoilTeX –

ld

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);

– Typeset by FoilTeX –

ld

vorzeitiges Verlassen

- ... der Schleife

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

- ... des Schleifenkörpers

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

– Typeset by FoilTeX –

ld

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?

– Typeset by FoilTeX –

ld

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)

– Typeset by FoilTeX –

ld

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; ..

⇒

```
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; ⇒

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

Anwendung: C-Programme ohne Schlüsselwörter.

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 × Anweisungs-Nummer
- Transitionen? (Beispiele)

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

Auswertung der Umfrage

<http://www.imn.htwk-leipzig.de/~waldmann/edu/ws13/pps/umfrage/>

Unterprogramme

Grundsätzliches

Ein Unterprogramm ist ein benannter Block mit einer Schnittstelle. Diese beschreibt den Datentransport zwischen Aufrufer und Unterprogramm.

- Funktion
 - liefert Wert
 - Aufruf ist Ausdruck
- Prozedur
 - hat Wirkung, liefert keinen Wert (void)
 - Aufruf ist Anweisung

Parameter-Übergabe (Semantik)

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

```
#include <errno.h>
```

```
extern int errno;
```

oder über Parameter.

Datentransport (entspr. Schlüsselwörtern 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!

– Typeset by FoilTeX –

ld

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)

– Typeset by FoilTeX –

ld

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{f
C y = new C {foo=3} ; C z = y; u (ref y);
Console.WriteLine(y.foo + " " + z.foo);
```

– Typeset by FoilTeX –

ld

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])
```

– Typeset by FoilTeX –

ld

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>

– Typeset by FoilTeX –

ld

Call-by-name in Scala

Parameter-Typ ist \Rightarrow 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

– Typeset by FoilTeX –

ld

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)

– Typeset by FoilTeX –

ld

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

– Typeset by FoilTeX –

ld

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

vgl. <http://mathworld.wolfram.com/DragonCurve.html>

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

nach erstem benanntem 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 : Inte
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++)
#include <iostream>

void sub (int & x, int & y, int & z) {
    y = 8;
    z = x;
}

int main () {
    int foo = 9;

```

– Typeset by FoilTeX –

ld

```

int bar = 7;

sub (foo, foo, bar);
std::cout << foo << std::endl;
std::cout << bar << std::endl;
}

```

– Typeset by FoilTeX –

ld

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?

– Typeset by FoilTeX –

ld

Lokale Unterprogramme

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

```

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

```

– Typeset by FoilTeX –

ld

Statische und dynamische Sichtbarkeit

Zugriff auf nichtlokale Variablen? (Bsp: Zugriff auf X in F)

```

with Ada.Text_IO; use Ada.Text_IO;
procedure Nest is
    X : Integer := 4;
    function F (Y: Integer) return Integer is
    begin return X + Y; end F;
    function G (X : Integer) return Integer is
    begin return F(3 * X); end G;
begin Put_Line (Integer' Image (G(5)));
end Nest;

```

- statische Sichtbarkeit: textuell umgebender Block (Pascal, Ada, Scheme-LISP, Haskell ...)
- dynamische Sichtbarkeit: Aufruf-Reihenfolge ((Common-LISP), (Perl))

– Typeset by FoilTeX –

ld

Frames, Ketten

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 $(0, j)$

– Typeset by FoilTeX –

ld

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;

```

– Typeset by FoilTeX –

ld

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.

– Typeset by FoilTeX –

ld

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));
```

– Typeset by FoilTeX –

ld

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)

– Typeset by FoilTeX –

ld

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.

⇒ Die (Frames in den) Closures müssen im Heap verwaltet werden.

– Typeset by FoilTeX –

ld

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: $\lambda a.b.2a + b$

(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](http://ftp.cs.ru.nl/pub/CompMath.Found/church.ps)

– Typeset by FoilTeX –

ld

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 { .. } .. } }
```

– Typeset by FoilTeX –

ld

Lokale Funktionen in Java 8

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

bisher (Java ≤ 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>`

– Typeset by FoilTeX –

ld

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.

– Typeset by FoilTeX –

ld

Polymorphie

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

– Typeset by FoilTeX –

ld

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 eine Halbordnung auf Typen

Bsp.

```
class C; class D extends C; class E extends C
```

definiert Relation

$(\leq) = \{(C, C), (D, C), (D, D), (E, C), (E, E)\}$ auf $T = \{C, D, E\}$

Relation \leq^2 auf T^2 :

$(t_1, t_2) \leq^2 (t'_1, t'_2) : \iff t_1 \leq t'_1 \wedge t_2 \leq t'_2$

es gilt $(D, D) \leq^2 (C, C)$; $(D, D) \leq^2 (C, D)$; $(C, D) \leq^2 (C, C)$; $(E, C) \leq^2 (C, C)$.

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 \leq^n 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; }
}

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

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

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

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);
```

Dynamische Polymorphie (Objektorientierung)

Definitionen

ein Bezeichner mit mehreren Bedeutungen

poly-morph = viel-gestaltig. Formen der Polymorphie:

- ad-hoc:
einfaches Überladen von Bezeichnern
- parametrisch (und statisch):
Typparameter für generische Klassen und Methoden
- dynamisch:
Auswahl der Methoden-Implementierung durch Laufzeittyp des Objektes

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)

– Typeset by FoilTeX –

ld

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 ⇒

weitsuchen im Prototyp ⇒ ... 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))
```

– Typeset by FoilTeX –

ld

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)

– Typeset by FoilTeX –

ld

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)

– Typeset by FoilTeX –

ld

Vererbung

Def: Klasse *D* ist abgeleitet von Klasse *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*).

– Typeset by FoilTeX –

ld

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 ();
```

– Typeset by FoilTeX –

ld

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*

⇒ object-orientation is, by its very nature, anti-modular ...

<http://existentialtype.wordpress.com/2011/03/15/teaching-fp-to-freshmen/>

– Typeset by FoilTeX –

ld

Überschreiben und Überladen

• Überschreiben:

zwei Klassen, je eine Methode mit gleichem Typ

• Überladen:

eine Klasse, mehrere Methoden mit versch. Typen

• C++: Methoden, die man überschreiben darf, *virtual* deklarieren

• C#: Überschreiben durch *override* anzeigen,

• Java: alle Methoden sind *virtual*, deswegen ist Überschreiben von Überladen schlecht zu unterscheiden: Quelle von Programmierfehlern

• Java-IDEs unterstützen Annotation *@overrides*

– Typeset by FoilTeX –

ld

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 * thi
}
```

nicht so:

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

– Typeset by FoilTeX –

ld

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.
}
```

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*.

– Typeset by FoilTeX –

ld

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)

– Typeset by FoilTeX –

ld

Generische Fkt. höherer Ordg.

Anwendung: 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)
```

– Typeset by FoilTeX –

ld

Anonyme Typen (Wildcards)

Wenn man einen generischen Typparameter nur einmal braucht, dann kann er `?` heißen.

```
List<?> x = Arrays.asList
    (new String[] { "foo", "bar" });
Collections.reverse(x);
System.out.println(x);
```

jedes Fragezeichen bezeichnet einen anderen (neuen) Typ:

```
List<?> x = Arrays.asList
    (new String[] { "foo", "bar" });
List<?> y = x;
y.add(x.get(0));
```

– Typeset by FoilTeX –

ld

Vererbung und generische Polym. (I)

- Vererbung: jedes Objekt bringt seine eigene Implementierung mit
- Generizität: (gemeinsame) Implementierung wird durch (Typ/Funktions)-Parameter festgelegt

```
interface I { void P (); }
static void Q (IList<I> xs)
    { foreach (I x in xs) { x.P(); } }
static void R<C> (Action<C> S, IList<C> xs)
    { foreach (C x in xs) { S(x); } }
```

für gleichzeitige Behandlung mehrerer Objekte ist Vererbungspolymorphie meist ungeeignet
(z. B. `Object.equals(Object o)` falsch, `Comparable<T>.compareTo(T o)` richtig)

– Typeset by FoilTeX –

ld

Vererbung und generische Polym. (II)

- mit Sprachkonzept *Vererbung* ist Erweiterung des Sprachkonzeptes *Generizität* wünschenswert:
- beim Definition der Passung von parametrischen Typen sollte die Vererbungsrelation \leq auf Typen berücksichtigt werden.
- Ansatz: wenn $E \leq C$, dann auch $List<E> \leq List<C>$
- ist *nicht* typsicher, siehe folgendes Beispiel
- Modifikation: ko- und kontravariante Typparameter

– Typeset by FoilTeX –

ld

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:

– Typeset by FoilTeX –

ld

Statische Sicherheit

- für Java und C#: wenn Programm P statisch korrekt ist, dann sind zur Laufzeit alle Methodenaufrufe erfolgreich
- d.h. für jedes $o.m()$: wenn $o \neq \text{null}$, dann hat o eine Implementierung von m
- ... aber $o \neq \text{null}$ ist nicht garantiert.
- ... weil zwischen T und $\text{Maybe } T$ (bzw. $\text{Nullable}\langle T \rangle$) nicht unterschieden wird

– Typeset by FoilTeX –

ld

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) { .. }
```

– Typeset by FoilTeX –

ld

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)
```

– Typeset by FoilTeX –

ld

Wildcards und Bounds

```
List<? extends Number> z =
Arrays.asList(new Double[]{1.0, 2.0});
z.add(new Double(3.0));
```

– Typeset by FoilTeX –

ld

variante generische Interfaces (C#)

Kontravarianz (in P), Kovarianz (out P)

```
interface I<in P> { // kontravariante Dekl.
// P get (); kovariante Benutzung (verbote
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: $E \leq C \Rightarrow I(E) \geq I(C)$
- kovariant: $E \leq C \Rightarrow I(E) \leq I(C)$
- invariant: $E \neq C \Rightarrow I(E) \not\leq I(C)$

– Typeset by FoilTeX –

ld

Vergleich: Varianz und Schranken

Unterscheidung:

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

– Typeset by FoilTeX –

ld

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();
aber ... (Übung)
```

– Typeset by FoilTeX –

ld

Generics und Arrays (II)

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: ...

– Typeset by FoilTeX –

ld

Übung Polymorphie

- Aufgabe zu Funktion `Flip` (C#)
- `binarySearch` aufrufen (Java), so daß beide ? von T verschieden sind
- Typescript
<http://www.typescriptlang.org/Handbook>
Benutzung: Quelltext in Datei `C.ts`, dann

```
tsc C.ts  
node C.js
```

Ergänzungen

Statisch typisiert ⇒ 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 Jahrzehnten (Algol 1960, ML 1970, C++ 1980, Java 1990 usw.)
Was dagegen?

- Typsystem ist ausdruckschwach:
(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.)

„wichtige“ „falsche“ Sprachen: JS

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

- Motivation: Software soll beim Kunden laufen
- technisches Problem: Kunde versteht/beherrscht seinen Computer/Betriebssystem nicht
(z.B. kann oder will keine JRE)
- stattdessen zwingt man Kunden zu Flash-Plugin oder
- 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)

„wichtige“ „falsche“ Sprachen: PHP

- Facebook ist in PHP implementiert
- deswegen steckt Facebook viel Geld in diese Sprache aus ebenfalls verständlichen Gründen :
 - für Kunden: Reaktionszeit der Webseite
 - für Betreiber: Entwicklungs- und Betriebskosten

Aktuelle Entwicklungen: JS

... was ist mit Microsoft? Die haben auch viel Geld und clevere Leute? — Ja:

- ECMA-Script 6 übernimmt viele Konzepte moderner (funktionaler) Programmierung, u.a.
 - `let` (block scope), `const` (single assignment)
 - `destructuring` (pattern matching)
 - `tail calls` (ohne Stack)<https://github.com/lukehoban/es6features>
- <http://www.typescriptlang.org/>
TypeScript adds *optional types*, *classes*, and *modules* to JavaScript.

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

Aktuelle Entwicklungen: PHP

- HHVM: Hip Hop Virtual Machine
<https://github.com/facebook/hhvm/blob/master/hphp/doc/bytocode.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>

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 ($\&\text{mut } T$).

<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", $(x, y) \Rightarrow x+y$, Typen: bool, int
 - Kombinationen (Funktionen):
 - Datum \rightarrow Datum, Bsp. $(x, y) \Rightarrow x+y$
 - Typ \rightarrow Typ, Bsp. `List<T>`
 - Typ \rightarrow Datum, Bsp. `Collections.<String>sort()`
 - Datum \rightarrow 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

Zusammenfassung

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.*

autotool-Auswertung

```
117 : 6*76* : 4 1 2 1
77 : 6*25* : 1 1 2 1 1 1 1 1
67 : 6*23* : 3 1 1
55 :
```

Bemerkungen

- $L(X_1) = L(X_2)$ für $X_i \in \text{REGEX}$ ist entscheidbar (aber PSPACE-vollständig)
- $L(G_1) = L(G_2)$ für $G_i \in \text{CFG}$ ist nicht entscheidbar
- für $G \in \text{CFG}$: „ G ist eindeutig“ ist nicht entscheidbar
- für $G \in \text{CFG}$: $\Sigma^* \setminus L(G)$ ist nicht notwendig CF
- Hanoi-* geht besser: man kann den Suchraum komplett durchlaufen und damit das Minimum exakt bestimmen.

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 (Masterverteidigungen Mi, 27. 1., 15:00 und 16:15 Z 417)
- Fortgeschrittene Konzepte der Programmierung (OS)
- Symbolisches Rechnen (Transform. v. Syntaxbäumen)

Wie weiter? (Masterprojekt/Arbeit, WHK)

- Allgemeines und Aktuelles: <http://www.imn.htwk-leipzig.de/~waldmann/edu/diplom/>
 - Termersetzung/Symbolisches Rechnen,
 - Constraint-Programmierung
 - E-Learning, insb. für Semantik von Programmiersprachen
 - für Einsteiger: u.a. Reparatur/Erweiterung autotool
 - ersetze XML-RPC durch JSON-RPC
 - Datenbank-Refactoring
 - besseres Pretty-Printing
- dafür Vertrag als wiss. Hilfskraft möglich (auch sofort)

Die neue autotool-Oberfläche

<https://autotool-test.imn.htwk-leipzig.de/semester/81/vorlesungen>

- Testbetrieb! Daten am 22. 1. 2016 von Produktionsbetrieb übernommen (werden *nicht* synchronisiert)
- Quelltexte: <https://gitlab.imn.htwk-leipzig.de/autotool/yesod-tool/tree/yesod/yesod>
- realisiert mit <http://www.yesodweb.com/> (Framework) und <https://haskell.org/> (Sprache)
- Fehlermeldungen, Verbesserungsvorschläge: <https://gitlab.imn.htwk-leipzig.de/groups/autotool/issues> (mit HRZ-Account über LDAP)

Testfragen

Die folgende Grammatik G über dem Alphabet $\Sigma = \{w, f, u, i\}$ soll Ausdrücke mit den Konstanten w, f und den binären Operatoren u, i beschreiben:

$$G = (\Sigma, \{E\}, \{E \rightarrow w \mid f \mid EiE \mid EuE\}, E).$$

Begründen Sie, daß G mehrdeutig ist.

Gesucht ist eine zu G äquivalente eindeutige kontextfreie Grammatik G' , für deren Ableitungsbäume gilt: der Operator u ist linksassoziativ, der Operator i ist rechtsassoziativ, der Operator u bindet stärker als der

Operator i .

Wie sieht unter diesen Bedingungen der abstrakte Syntaxbaum für $fiwufiw$ aus?

Untersuchen Sie, ob G_1, G_2, G_3 die gewünschten Eigenschaften erfüllen. (Falls nein: begründen, falls ja: konkreten Syntaxbaum für $fiwufiw$ angeben.)

- $G_1 = (\Sigma, \{E, A\}, \{E \rightarrow A \mid AiE \mid EuA, A \rightarrow w \mid f\}, E)$.

- $G_2 = (\Sigma, \{E, A, B\}, \{E \rightarrow B \mid BiE, B \rightarrow A \mid BuA, A \rightarrow w \mid f\}, E)$.

- $G_3 = (\Sigma, \{E, A, B\}, \{E \rightarrow B \mid BuE, B \rightarrow A \mid BiA, A \rightarrow w \mid f\}, E)$.

- Welches ist die Bedeutung der Aussageform $\{V\}P\{N\}$ im Hoare-Kalkül?

Geben Sie eine wahre und eine falsche Aussage dieser Form an.

- Wodurch wird eine kontextfreie Grammatik zu einer Attributgrammatik erweitert?

- Geben Sie einen regulären Ausdruck für die Spursprache dieses Programms an.

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

Das Spur-Alphabet ist $\{A, B, C, P_0, P_1, Q_0, Q_1\}$, dabei bedeuten

A : die Anweisung A wird ausgeführt,

P_0 (bzw. P_1): der Ausdruck P wird ausgewertet und ergibt

falsch (bzw. *wahr*).

Nach welcher Regel bestimmt man, ob ein Ausdruck $f(x)$ korrekt getypt ist? (Ohne Berücksichtigung von Vererbung oder Generizität.)

- Wenn f den Typ ... hat

- und x den Typ ... hat,

- dann ist der Typ von $f(x)$

Wie werden die folgenden Operationen für Typen in Programmiersprachen realisiert?

- Vereinigung:

- Kreuzprodukt:

- Potenz (vier verschiedene Realisierungen)

–

–

–

–

In Java gibt es keine direkte Realisierung der Vereinigung, was wird stattdessen empfohlen?

Für das Ada-Programm:

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Main is
```

```
  X : Integer := 3; Y : Integer := 2;
```

```
  procedure P (X : Integer) is
```

```
    procedure Q (Y : Integer) is
```

```
      procedure R (X : Integer) is
```

```
        begin Put_Line (Integer'Image (X+Y))
```

```
        begin if Y > 0 then P(X-1); else R(X+Y);
```

```
        begin if X > 0 then Q(X-1); else P(X-Y);
```

```
      begin P (X-1); end Main;
```

Zeichnen Sie die Frames mit allen Einträgen und

Verweisen zu dem Zeitpunkt direkt vor dem ersten Aufruf von `Put_Line`.

Wie wird auf die Werte von x und y zugegriffen, die in `Integer'Image (X+Y)` benötigt werden?

Für folgende Deklaration:

```
int a [] = { 1,2,0 }; void p (int x, int y)
```

betrachten wir den Aufruf `p(a[0], a[1])`.

Geben Sie die Ausführungsschritte sowie die resultierende Speicherbelegung an, falls zur Parameterübergabe benutzt wird:

- Wertübergabe

- Verweis-Übergabe

Für die Deklarationen:

```
class C { }    class D extends C { }  
static void p (Object x, C      y) { System.  
static void p (D      x, C      y) { System.  
static void p (C      x, Object y) { System.
```

Beschreiben Sie, wie die Überladung für die folgenden Aufrufe aufgelöst wird:

- `p (new D(), new D());`

- `p (new C(), new C());`