Prinzipien von Programmiersprachen
Vorlesung
Wintersemester 2007, 08, 09, 10, 11, 12

Johannes Waldmann, HTWK Leipzig

5. November 2012

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

Sprache

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

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)

Paradigmen

- imperativ

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

Ziele der LV

Arbeitsweise: Methoden, Konzepte, Paradigmen

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

Ziel:

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

Beziehungen zu anderen LV

- Grundlagen der Informatik, der Programmierung:

  strukturierte (imperative) Programmierung
- Softwaretechnik 1/2:

  objektorientierte Modellierung und Programmierung, funktionale Programmierung und OO-Entwurfsmuster
- Compilerbau: Implementierung von Syntax und Semantik

Sprachen für bestimmte Anwendungen, mit bestimmten Paradigmen:

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

Organisation

- Vorlesung Mo (u+g) 15:30 Li 211
- Übungen (alle in Z423)
  - Di (u+g) 9:30 (MIM)
  - Do(u) 11:15 + Do(g) 9:30 (INM)
  - Di(u) 13:45 + Di(g) 11:15

  Übungsgruppe wählen: `https://autotool.imn.htwk-leipzig.de/cgi-bin/Super.cgi`
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Klausur: 120 min, ohne Hilfsmittel

Literatur

- http://www.imn.htwk-leipzig.de/~waldmann/edu/ws12/pps/folien/pps/
- Robert W. Sebesta: Concepts of Programming Languages, Addison-Wesley 2004, ...

Zum Vergleich/als Hintergrund:

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

Inhalt

(nach Sebesta: Concepts of Programming Languages)

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

Übungen

1. Anwendungsgebiete von Programmiersprachen, wesentliche Vertreter

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

Lösungen:

```
find workspace/ -name "*.java" | xargs grep
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)co
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) (jeweils auf und zu)
- Operatoren (=, +, &&, ...)

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

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

Reguläre Sprachen/Ausdrücke

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

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

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?)

Bemerkung zu Reg. Ausdr.

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

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

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

Übungen Reg. Ausdr.

- $(\Sigma^*, \cdot, \epsilon)$ ist Monoid
- . . . aber keine Gruppe, weil man im Allgemeinen nicht dividieren kann. Welche Relation ergibt sich als „Teilbarkeit“: $u \mid w := \exists v : u \cdot v = w$
- Zeichne Hasse-Diagramme der Teilbarkeitsrelation
  - auf natürlichen Zahlen $\{0, 1, \ldots, 10\}$,
  - auf Wörtern $\{a, b\}^{\leq 2}$
- $(\mathrm{Pow}(\Sigma^*), \cup, \cdot, \ldots, \ldots)$ ist Halbring.

  Beispiel für Distributivgesetz?

  Welches sind jeweils die neutralen Elemente der Operationen?

(vgl. oben) Welche Relation auf Sprachen (Mengen) ergibt sich als „Teilbarkeit“ bzgl. $\cup$?

- Damit $a^{b+c} = a^b \cdot a^c$ immer gilt, muß man a^0 wie definieren?
- Block-Kommentare und weitere autotool-Aufgaben
- reguläre Ausdrücke für Tokenklassen in der Standard-Pascal-Definition
  `http://www.standardpascal.org/iso7185.html#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:
$u \to_R v \iff \exists x, z \in \Sigma^*, (l, r) \in R : u = x \cdot l \cdot z \wedge x \cdot r \cdot z = v.$
Beispiel: Bubble-Sort: $\{ba \to ab, ca \to ac, cb \to bc\}$
Beispiel: Potenzieren: $ab \to bba$
Aufgaben: gibt es unendlich lange Rechnungen für:
$R_1 = \{1000 \to 0001110\}, R_2 = \{aabb \to bbbaaa\}$?

Grammatiken

Grammatik G besteht aus:

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

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

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

Formale Sprachen: Chomsky-Hierarchie

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

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

Typ-3-Grammatiken

(= rechtslineare Grammatiken)
jede Regel hat die Form

- Variable → Terminal Variable
- Variable → Terminal
- Variable → ϵ

(vgl. lineares Gleichungssystem)
Beispiele

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

Sätze über reguläre Sprachen

Für jede Sprache L sind die folgenden Aussagen äquivalent:

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

Beweispläne:

- Grammatik ↔ Automat (Variable = Zustand)
- Ausdruck → Automat (Teilbaum = Zustand)
- Automat → 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 -> Bezeichner = Ausdruck
    | if Ausdruck then Anweisung else Anweis
Ausdruck -> Bezeichner | Literal
    | Ausdruck Operator Ausdruck
```

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

Klammer-Sprachen

Abstraktion von vollständig geklammerten Ausdrücke mit zweistelligen Operatoren

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

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

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

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 ($ab \to baa$, 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) ≈ kontextfreie Grammatik

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

Erweiterte BNF

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

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

kann in BNF übersetzt werden

Ableitungsbäume für CF-Sprachen

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

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

Ableitungsbäume (II)

Def: der *Rand* eines geordneten, markierten Baumes (T, m) ist die Folge aller Blatt-Markierungen (von links nach rechts).

Beachte: die Blatt-Markierungen sind $\in \{\epsilon\} \cup \Sigma$, d. h. Terminalwörter der Länge 0 oder 1.

Für Blätter: $\mathrm{rand}(b) = m(b)$, für innere Knoten: $\mathrm{rand}(k) = \mathrm{rand}(k_1)\,\mathrm{rand}(k_2)\ldots\mathrm{rand}(k_n)$

Satz: $w \in L(G) \iff$ existiert Ableitungsbaum (T, m) für G mit $\mathrm{rand}(T, m) = w$.

Eindeutigkeit

Def: G heißt *eindeutig*, falls $\forall w \in L(G)$ *genau ein* Ableitungsbaum (T, m) existiert.

Bsp: ist $\{S \to aSb|SS|\epsilon\}$ eindeutig?

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

Die naheliegende Grammatik für arith. Ausdr.

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

ist mehrdeutig (aus *zwei* Gründen!)

Auswege:

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

Assoziativität

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

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

- . . . und dann die Grammatik entsprechend einrichten

Assoziativität (II)

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

Grammatik-Regeln

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

ersetzen durch

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


Präzedenzen

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

Grammatik-Regel

```
summand -> zahl
```

erweitern zu

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

(Assoziativität beachten)

Zusammenfassung Operator/Grammatik

Ziele:

- Klammern einsparen
- trotzdem eindeutig bestimmter Syntaxbaum

Festlegung:

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

Realisierung in CFG:

- Links/Rechts-Assoziativität ⇒ Links/Rechts-Rekursion
- verschiedene Präzedenzen ⇒ verschiedene Variablen

Übung Operator/Grammatik

Übung:

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

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

Attributgrammatiken (I)

- Attribut: Annotation an Knoten des Syntaxbaums.

  A : Knotenmenge $\to$ Attributwerte (Bsp: $\mathbb{N}$)

- Attributgrammatik besteht aus:

  – kontextfreier Grammatik G (Bsp: $\{S \to e \mid mSS\}$)
  – für jeden Knotentyp (Terminal + Regel)
  eine Menge (Relation) von erlaubten Attribut-Tupeln
  $(A(X_0), A(X_1), \ldots, A(X_n))$
  für Knoten X_0 mit Kindern $[X_1, \ldots, X_n]$

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

Attributgrammatiken (II)

ein Ableitungsbaum mit Annotationen ist
korrekt bezüglich einer Attributgrammatik, wenn

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

Plan:

- Baum beschreibt Syntax, Attribute beschreiben Semantik

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

Donald E. Knuth

- The Art Of Computer Programming (1968, . . .)

  (Band 3: Sortieren und Suchen)

- TEX, Metafont, Literate Programming (1983, . . .)

  (Leslie Lamport: LATEX)

- Attribut-Grammatiken
- die Bezeichnung „NP-vollständig“
- . . .

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

Arten von Attributen

- synthetisiert:

  hängt nur von Attributwerten in Kindknoten ab

- ererbt (inherited)

  hängt nur von Attributwerten in Elternknoten und (linken) Geschwisterknoten ab

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

Attributgrammatiken–Beispiele

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

Konkrete und abstrakte Syntax

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

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

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

Regeln zur Typprüfung

. . . bei geschachtelten Funktionsaufrufen

- Funktion f hat Typ $A \to B$
- 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)

Ausdrücke → Kellermaschine

Beispiel:

$3 * x + 1 \Rightarrow$ push 3, push x, mal, push 1, plus

- Code für Konstante/Variable c : `push c;`
- Code für Ausdruck x op y: `code(x); code(y); op;`
- Ausführung eines Operators:

  holt beide Argumente vom Stack, schiebt Resultat auf Stack

Der erzeugte Code ist synthetisiertes Attribut!
Beispiele: Java-Bytecode (javac, javap),
CIL (gmcs, monodis)

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
```