Prinzipien von Programmiersprachen
Vorlesung
Wintersemester 2007 – 2026

Johannes Waldmann, HTWK Leipzig

9. November 2026

– Typeset by FoilTEX –

Einleitung

Programme und Algorithmen

- Algorithmus (vgl. VL Alg. und Datenstr.)

  Vorschrift zur Lösung einer Aufgabe

- Programm (vgl. VL zu (Anwendungsorientierter) Progr.)

  Realisierung eines Algorithmus in konkreter Programmiersprache, zur Ausführung durch Maschine

- Programmiersprache

  bietet Ausdrucksmittel zur Realisierung von Algorithmen als Programme

Deutsch als Programmiersprache

§6 (2) . . . Der Zuteilungsdivisor ist so zu bestimmen, dass insgesamt so viele Sitze auf die Landeslisten entfallen, wie Sitze zu vergeben sind. Dazu wird zunächst die Gesamtzahl der Zweitstimmen aller zu berücksichtigenden Landeslisten durch die Zahl der jeweils nach Absatz 1 Satz 3 verbleibenden Sitze geteilt. Entfallen danach mehr Sitze auf die Landeslisten, als Sitze zu vergeben sind,. . .

§6 (5) Die Zahl der nach Absatz 1 Satz 3 verbleibenden Sitze wird so lange erhöht, bis jede Partei bei der zweiten Verteilung der Sitze nach Absatz 6 Satz 1 mindestens die bei der ersten Verteilung nach den Absätzen 2 und 3 für sie ermittelten zuzüglich der in den Wahlkreisen errungenen Sitze erhält, die nicht nach Absatz 4 Satz 1 von der Zahl der für die Landesliste ermittelten Sitze abgerechnet werden können.

`https://www.gesetze-im-internet.de/bwahlg/__6.html`

Struktur durch Klammern, ist doch klar

- Wo ist die öffnende Klammer für die Muskatnuß?



(HMF Food Production, Dortmund, 2022)

- vgl. Syntax von LISP (John McCarthy 1958), z.B. `https://research.scheme.org/lambda-papers/`

Beispiel: mehrsprachige Projekte

ein typisches Projekt besteht aus:

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

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

In / Into

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

  A good programmer can write LISP in any language.

Sprache

- wird benutzt, um Ideen festzuhalten/zu transportieren (Wort, Satz, Text, Kontext)

- wird beschrieben durch
  - Lexik
  - Syntax
  - Semantik
  - Pragmatik

- natürliche Sprachen / formale Sprachen

Wie unterschiedlich sind Sprachen?

- weitgehend übereinstimmende Konzepte.
  - LISP (1958) $=$ Perl $=$ PHP $=$ Python $=$ Ruby $=$ Javascript $=$ Clojure: imperativ, (funktional), nicht statisch typisiert (d.h., unsicher und ineffizient)
  - Algol (1958) $=$ Pascal $=$ C $=$ Java $=$ C# imperativ, statisch typisiert
  - ML (1973) $=$ Haskell: statisch typisiert, generische Polymorphie

- echte Unterschiede („Neuerungen“) gibt es auch
  - CSP (1977) $=$ Occam (1983) $=$ Go: Prozesse, Kanäle
  - Clean (1987) $\approx$ Rust (2012): Lineare Typen
  - Coq (1984) $=$ Agda (1999) $=$ Idris: dependent types

Konzepte

- Syntax: konkrete (für Zeichenfolgen), abstrakte (Bäume)
- Hierarchien (baumartige Strukturen)
  - einfache und zusammengesetzte (arithmetische, logische) Ausdrücke
  - einfache und zusammengesetzte Anweisungen (strukturierte Programme)
  - Komponenten (Klassen, Module, Pakete)
- Semantik: statische (vor Ausführung), dynamische
- Typen (Code-Prüfung und -Erzeugung vor Ausführung)
- Namen: stehen für Werte, gestatten Wiederverwendung
- flexible Wiederverwendung durch Parameter (Argumente)
  Unterprogramme: Daten, Polymorphie: Typen

Paradigmen

- imperativ

  Programm ist Folge von Befehlen

  (Befehl bewirkt Zustandsänderung)

- 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/Projekt objektorientierte Modellierung und Programmierung, funktionale Programmierung und OO-Entwurfsmuster
- Compilerbau: Implementierung v. Syntax u. Semantik
- (SS 22) Fortgeschrittene Konzepte in Progr.-Spr., (SS 26) Esoterische Progr.-Spr., ...

Anwendungsspezifische Sprachen und Paradigmen:

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

Organisation

- Vorlesung
- Hausaufgaben (ergibt Prüfungszulassung)
  mit diesen Aufgaben-Formen:
  - individuell online (autotool)
    `https://autotool.imn.htwk-leipzig.de/new/semester/102/vorlesungen`
  - in Gruppen (je 2 Personen)
    Präsentation und Bewertung in Übung (3 mal)
    Koordination: Projekt (Wiki, Issues) `https://gitlab.dit.htwk-leipzig.de/sws/pps-ws26`
- Klausur: 120 min, ohne Hilfsmittel

Literatur

- Skript, Aufgaben usw. erreichbar von

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

Zum Vergleich/als Hintergrund:

- Abelson, Sussman, Sussman: Structure and Interpretation of Computer Programs, MIT Press 1984

  `https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/index.html`

- Robert W. Sebesta: Concepts of Programming Languages, Addison-Wesley 2004, . . .
- Turbak, Gifford: Design Concepts of Programming Languages, MIT Press 2008

  `https://cs.wellesley.edu/~fturbak/`

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

Haus-Aufgaben

Bei Vorführung: *Pool-Rechner* benutzen (nicht mitgebrachte eigene) (Chancengleichheit, Reproduzierbarkeit), *große* (Control-Plus), *schwarze* Schrift auf weißem Grund

WS 26: 7, 5, 9 (Ersatz: 4)

1. Lesen Sie E. W. Dijkstra: *On the foolishness of natural language programming“*

   `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD667.html`

   und beantworten Sie

   - womit wird “einfaches Programmieren” fälschlicherweise gleichgesetzt?

- welche wesentliche Verbesserung brachten höhere Programmiersprachen, welche Eigenschaft der Maschinensprachen haben sie trotzdem noch?
- warum sollte eine Schnittstelle *narrow* sein?
- welche formalen Notationen von Vieta, Descartes, Leibniz, Boole sind gemeint? (jeweils: Wissenschaftsbereich, (heutige) Bezeichnung der Notation, Beispiele)
- warum können Schüler heute das lernen, wozu früher nur Genies in der Lage waren?
- Übersetzen Sie den Satz “the naturalness . . . obvious”.

Geben Sie dazu jeweils an:

- die Meinung des Autors, belegt durch konkrete Textstelle und zunächst wörtliche, dann sinngemäße

Übersetzung

- Beispiele aus Ihrer Erfahrung

2. zu John C. Reynolds: *Some Thoughts on Teaching Programming and Programming Languages* 2008, von *An additional reason for teaching programming languages. . .* bis Ende:

- Warum wird auf Turing-Vollständigkeit verwiesen?
- Geben Sie Beispiele aus Ihrer Erfahrung für problematische *input formats*, oder problemfreie.
- *partial list of the kind of capabilities. . .* : ordnen Sie die Listenelemente konkreten Lehrveranstaltungen zu (bereits absolvierte oder noch kommende)

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

   Lösungen:

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

   Das dient als Wiederholung zur Benutzung von Unix (GNU/Linux): führen Sie vor:

   - eine Shell öffnen
   - in der Manpage von `find` die Beschreibung von `-exec` anzeigen. Finden Sie (mit geeignetem

Shell-Kommandos) den Quelltext dieser Manpage, zeigen diesen an. (Wie benutzt man `man`? so: `man man`.)
(2024 – was war hier los? `https://git.savannah.gnu.org/cgit/man-db.git/commit/?id=b225d9e76fbb0a6a4539c0992fba88c83f0bd37e`

- was bedeutet der senkrechte Strich? in welcher Manpage steht das? in welcher Vorlesung war das dran?

4. funktionales Programmieren in Haskell

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

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

- zeigen Sie (den Studenten, die das noch nicht gesehen haben), wo die Software (hier `ghc`) im Pool installiert ist, und wie man sie benutzt und die Benutzung vereinfacht (`PATH`)
- Werten Sie den angegebenen Ausdruck aus sowie alle Teilausdrück (`[1..100]`, `product [1..100]`, usw.
- den Typ von `reverse` durch `ghci` anzeigen lassen
- nach diesem Typ in

`https://hoogle.haskell.org/` suchen. (Einschränken auf `package:base`) Die anderen (drei) Funktionen dieses Typs aufrufen.

- eine davon erzeugt unendliche Listen, wie werden die im Speicher repräsentiert, wie kann man sie benutzen? (Am Beispiel zeigen.)

5. 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 2
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 rc
```

In eine Text-Datei `what.ps` schreiben (vgl. Aufgabe 8) ansehen mit `gv what.ps` (im Menu: State → watch file).

Mit Editor Quelltext ändern, Wirkung betrachten.

- Ändern Sie die Strich-Stärke!
- wie funktioniert die Steuerung einer Zählschleife?
- warum ist PostScript: imperativ? strukturiert? prozedural?
- führen Sie wenigstens ein weiteres ähnliches PostScript-Programm vor (kurzer Text, aber nichttriviale Rechnung). Quelle angeben, Programmtext erklären!

6. In SICP 1.1 werden drei *Elemente der Programmierung* genannt. Illustrieren Sie diese Elemente durch Beispiele aus `https://web.archive.org/web/20241109065141/https://www.99-bottles-of-beer.net/`

Führen Sie nach Möglichkeit vor (im Pool, nicht in Web-Oberfläche von Dritt-Anbietern).

7. Stellen Sie Ihren Browser datenschutzgerecht ein (Wahl des Browsers, der Default-Suchmaschine, Blockieren von Schadsoftware.)

   In einem neuen Firefox-Profil (about:profiles) ausprobieren und diskutieren: Umatrix (dessen Log betrachten), Temporary Containers.

   Vgl. `https://restoreprivacy.com/firefox-privacy/` (hat selbst viele Tracker!) und weitere.

8. erklären Sie `https://xkcd.com/378/`.

   führen Sie die vier genannten Editoren vor, in dem Sie jeweils eine einzeilige Textdatei erzeugen.

9. durch (möglichst kurzes) Programm in einer Programmiersprache Ihrer Wahl: bestimmen Sie den längsten Satz aus Thomas Mann: Zauberberg (`https://www.gutenberg.org/ebooks/65661`). Drücken Sie dessen Struktur durch Klammern oder Einrückungen aus (wie bei einem Programmtext).

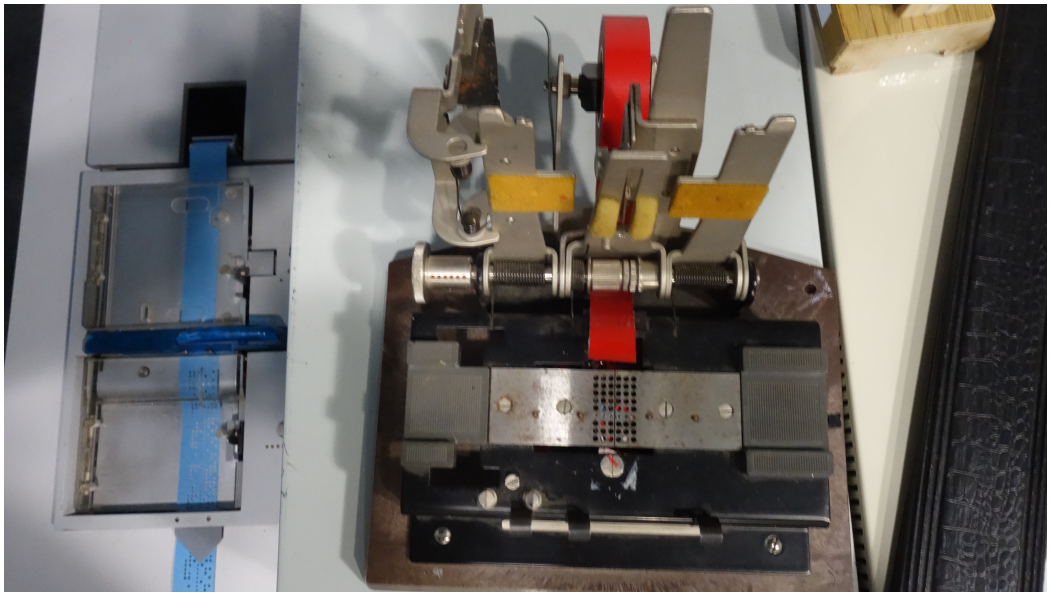
Syntax von Programmiersprachen

Konkrete und abstrakte Syntax

- für Programmtexte unterscheiden wir:
  - Syntax $=$ Form, Bsp.: Zeichenfolge, Diagramm (Graph)
  - Semantik $=$ Bedeutung, Bsp.: Typ, Wert, Wirkung
- bei Syntax unterscheiden wir:
  - konkrete Syntax (was wir sehen): Zeichenfolge
  - abstrakte Syntax (was wir meinen): Baum
- der abstrakte Syntaxbaum ermöglicht *syntaxgesteuerte Semantik* ($=$ Induktion über den Baum)
- konkrete Syntax begründet durch technische Entwicklung von Ein- und Ausgabegeräten (Fernschreiber)

Fernschreiber, Editor...





- (links) Ein Mitarbeiter überträgt einen Text auf einen Lochstreifen, Kieler Nachrichten 1973, Wikimedia CC-BY-SA 3.0

  (rechts) Lochband-Editiervorrichtung (Lochmaske, rot: Klebeband) (Technische Sammlungen Dresden)

Terminal (Konsole)

- Schreibmaschine für Ein- und Ausgabe (*Bediendrucker*)

  ist Vorbild für elektronische Text-Ein- und -Ausgabegeräte





(links) VT 100, 1978, DEC (Digital Equipment Corporation) (Wikimedia, CC BY 2.0)

(rechts) Gnome Terminal 3.56.3 for Gnome 48,

Ganz ganz ganz kurze Geschichte der (Programmiersprachen und) Betriebssysteme

- Betriebssystem $=$ Schnittstelle zwischen Anwenderprogramm und Hardware (Zugriff auf Prozessor, Hauptspeicher, Massenspeicher, E/A-Geräte; Dienstprogramme: Kommandozeilen-Intepreter, Editor)
- Geld verdienen mit Hardware, Software (OS, Compiler) war Zugabe, lief aber nur dort. (FORTRAN, IBM 1958)
- OS/360 (IBM, 1964) Brooks: *The Mythical Man-Month*, 75
- UNIX (Thomson, Ritchie, Bell Labs 1969) portabel (C)
- GNU (GNU's not Unix) (Stallmann 1984) freie Software Dienstprogramme (bash, emacs), separat: Kernel (Hurd)
- Linux (Torvalds 1991) Kernel für i368
- Debian (Murdock 1993) ist GNU/Linux-Distribution

Programme als Bäume

- ein Programmtext repräsentiert eine Hierarchie (einen Baum) von Teilprogrammen
- Die Semantik des Programmes wird durch Induktion über diesen Baum definiert.
- dieses Prinzip kommt aus der Mathematik (arithmetische Ausdrücke, logische Formeln, Beweise — sind Bäume)
- In den Blättern des Baums stehen *Token*,
- jedes Token hat einen *Inhalt* (eine Zeichenkette, Bsp `12.34E5`) und eine *Klasse* (Bsp Gleitkomma-Literal)

Token-Klassen

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

alle Token einer Klasse bilden eine *formale Sprache*.

Formale Sprachen

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

Beispiele:

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

Lexik (Bsp): numerische Literale

- Ada (2012) `http://www.ada-auth.org/standards/rm12_w_tc1/html/RM-2-4.html`
- Beispiele (Elemente der Literalmenge)

```
12      0     1E6     123_456     --  integer literals
12.0    0.0   0.456   3.14159_26  --  real literals
```

- formale Definition der Literalmenge

```
numeric_literal ::= decimal_literal | based_literal
decimal_literal ::= numeral [.numeral] [exponent]
numeral ::= digit {[underline] digit}
exponent ::= E [+] numeral | E - numeral
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

benutzt eine Notation f. reguläre Ausdrücke

Spezifikation formaler Sprachen

man kann eine formale Sprache beschreiben:

- *algebraisch* (Sprach-Operationen)

  Bsp: reguläre Ausdrücke

- *generativ* (Grammatik), Bsp: kontextfreie Grammatik,

- durch *Akzeptanz* (Automat), Bsp: Kellerautomat,

- *logisch* (Eigenschaften),

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

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: $*$ oder weglassen)
  - (Vereinigung) $A + B \in E$ (autotool: $+$)
  - (Stern, Hülle) $A^* \in E$ (autotool: $\hat{}*$)

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: $\Sigma^* \setminus (\Sigma^* ab \Sigma^*)^2$
   ähnlich in Konfiguration der autotool-Aufgaben

2. zusätzliche nicht-reguläre Operatoren
   Beispiel: exakte Wiederholungen $L^{\boxed{k}} := \{w^k \mid w \in L\}$
   Bsp.: $(ab^*)^{\boxed{2}} = \{aa, abab, abbabb, ab^3ab^3, \ldots\} \notin$ REG

3. Markierung von Teilwörtern, definiert (evtl. nicht-reguläre)
   Menge von Wörtern mit Positionen darin

Implementierung regulärer Ausdrücke

- die richtige Methode ist Kompilation des RE in einen endlichen Automaten

  Ken Thompson: *Regular expression search algorithm*, Communications of the ACM 11(6) (June 1968)

- wenn nicht-reguläre Sprachen entstehen können (durch erweiterte RE), ist keine effiziente Verarbeitung (mit endlichen Automaten) möglich.

- auch reguläre Operatoren werden gern schlecht implementiert.

  Russ Cox: *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)*, 2007 `https://swtch.com/~rsc/regexp/regexp1.html`

Bemerkung zu Reg. Ausdr.

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

(Wort w gehört zur Sprache eines regulären Ausdrucks X)

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

Beispiel: $w = abba, X = (ab^*)^*$.

$w = abb \cdot a = ab^2 \cdot ab^0 \in ab^* \cdot ab^* \subseteq (ab^*)^2 \subseteq (ab^*)^*$.

Übungen zu Lexik (Testfragen)

(ohne Wertung, zur Wiederholung und Unterhaltung)

- was ist jeweils Eingabe und Ausgabe für: lexikalische Analyse, syntaktische Analyse?
- warum werden reguläre Ausdrücke zur Beschreibung von Tokenmengen verwendet? (was wäre die einfachste Alternative? für welche Tokentypen funktioniert diese?)
- $(\Sigma^*, \cdot, \epsilon)$ ist Monoid, aber keine Gruppe
- $(\mathrm{Pow}(\Sigma^*), \cup, \cdot, \ldots, \ldots)$ ist Halbring (ergänzen Sie die neutralen Elemente)
- In jedem Monoid: Damit $a^{b+c} = a^b \cdot a^c$ immer gilt, muß

man a^0 wie definieren?

Aufgaben zu regulären Ausdrücken: autotool. Das ist Wiederholung aus VL Theoretische Informatik—Automaten und Formale Sprachen.

Hausaufgaben

WS 26: 1, (3 oder 4 oder 5), 9.

1. Für jedes Monoid $M = (D, \cdot, 1)$ definieren wir die Teilbarkeits-Relation $u \mid w := \exists v : u \cdot v = w$

   Geben Sie Beispiele $u \mid w$, $\neg(u \mid w)$ an in den Monoiden

   - $(\mathbb{N}, +, 0)$
   - $(\mathbb{Z}, +, 0)$
   - $(\mathbb{N}, \cdot, 1)$
   - $(\{a, b\}^*, \cdot, \epsilon)$
   - $(2^{\mathbb{N}}, \cup, \emptyset)$

   Zeigen Sie (nicht für ein spezielles Monoid, sondern allgemein): die Relation $\mid$ ist reflexiv und transitiv.

   Ist sie antisymmetrisch? (Beweis oder Gegenbeispiel.)

NB: Beziehung zur Softwaretechnik:

- „Monoid“ ist die Schnittstelle (API, abstrakter Datentyp),
- $(\mathbb{N}, 0, +)$ ist eine Implementierung (konkreter Datentyp).
- „allgemein zeigen“ bedeutet: nur die in den Axiomen des ADT (API-Beschreibung) genannten Eigenschaften benutzen

2. Zeichnen Sie jeweils das Hasse-Diagramm dieser Teilbarkeitsrelation

- für $(\mathbb{N}, +, 0)$, eingeschränkt auf $\{0, 1, \ldots, 4\}$
- für $(\mathbb{N}, \cdot, 1)$, eingeschränkt auf $\{0, 1, \ldots, 10\}$
- für $(2^{\{p,q,r\}}, \cup, \emptyset)$
- für $(\{a, b\}^*, \cdot, \epsilon)$ auf $\{a, b\}^{\leq 2}$

Geben Sie eine Halbordnung auf $\{0, 1, 2\}^2$ an, deren

Hasse-Diagramm ein auf der Spitze stehendes Quadratnetz ist.

Diese Halbordnung soll *intensional* angegeben werden (durch eine Formel), nicht *extensional* (durch Aufzählen aller Elemente).

3. Führen Sie vor (auf Rechner im Pool Z430, vorher von außen einloggen und probieren)

   Editieren, Kompilieren, Ausführen eines kurzen (maximal 3 Zeilen) Pascal-Programms

   Der Compiler `fpc` (`https://www.freepascal.org/`) ist installiert (`/usr/local/waldmann/opt/fpc/latest`).

   (Zweck dieser Teilaufgabe ist nicht, daß Sie Pascal

lernen, sondern der Benutzung von ssh, evtl. tmux, Kommandozeile (PATH), Text-Editor wiederholen)

Zu regulären Ausdrücke für Tokenklassen in der Standard-Pascal-Definition `https://archive.org/details/iso-iec-7185-1990-Pascal/`

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

In Ihrem Beispiel-Programm: erproben Sie mehrere (korrekte und fehlerhafte) Varianten für Gleitkomma-Literale. Vergleichen Sie Spezifikation (geben Sie den passenden Abschnitt der Sprachdefinition an) und Verhalten des Compilers.

Dieser Compiler (fpc) ist in Pascal geschrieben. Was

bedeutet das für: Installation des Compilers, Entwicklung des Compilers?

4. Führen Sie vor (wie und warum: siehe Bemerkungen vorige Aufgabe): Editieren, Kompilieren (`javac`), Ausführen (`java`) eines kurzen (maximal 3 Zeilen) Java-Programms.

   Suchen und buchmarken Sie die *Java Language Specification* (Primärquelle in der aktuellen Version) Beantworten Sie *damit* (und nicht mit Hausaufgabenwebseiten und anderen Sekundärquellen):

   gehören in Java

   - `null`
   - Namen für Elemente von Aufzählungstypen

zur Tokenklasse Literal, reserviertes Wort (Schlüsselwort), Bezeichner (oder evtl. anderen)?

Wo stehen die Token-Definitionen im `javac`-Compiler? `https://hg.openjdk.java.net/jdk/jdk15/file/` (bzw. aktuelle Version)

In Ihrem Beispiel-Programm: erproben Sie verschiedene Varianten von Ganzzahl-Literalen (siehe vorige Aufgabe)

5. Führen Sie vor (wie vorige Aufgaben): Kompilation und Ausführung eines sehr kurzen Ada-Programms

```
with Ada.Text_IO; use Ada.Text_IO;
procedure floating is
begin put_line (float'image( 2)); -- fehler
```

```
end floating;
```

Verwenden Sie den *GNU Ada Translator*, ist Teil von GCC (GNU Compiler Collection).

Ist im Pool installliert, siehe `https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/`

Aufrufen mit `gnatmake floating.adb` (kompilieren und linken), ausführen mit `./floating`.

Erläutern Sie die Fehlermeldung durch Verweis auf den Sprachstandard. Setzen Sie passende Literale ein (ändern Sie den Rest des Programms nicht). Probieren Sie dabei alle Zweige und Optionen in den regulären Ausdrücken des Standards (2.4.1).

6. Im WS22 hatten Teilnehmer dieser LV diese Fehler im GNU Ada Translator (gnat, Teil von gcc) gefunden:

   - excessive compilation time for decimal literal—that should be rejected as type-error `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=107392`
   - decimal literal with long exponent: Constraint Error `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=107391`

   Untersuchen Sie ähnliches für Compiler für andere Sprachen.

7. Suchen und diskutieren Sie „Wadler's law (of language design)".

   Am Entwurf welcher Programmiersprachen war der Autor

beteiligt? Welche Sprache hat er in einem aktuellen Lehrbuch benutzt?

Untersuchen Sie für (wenigstens) Java und Haskell, ob Block-Kommentare geschachtelt werden können. Belegen Sie durch

- Sprachstandard (exakte Definition von Kommentaren)
- und eigene Beispiele (einfachste Programme, die vom Compiler akzeptiert oder abgelehnt werden)

8. Gelten die Aussagen von Cox (2007) („but it's slow in. . .“) jetzt immer noch? Überprüfen Sie das praktisch (die Testfälle aus dem zitierten Paper oder ähnliche).

9. Ein Sprach-Standard (Rust) definiert die Tokenklasse:

```
DEC_LITERAL → DEC_DIGIT ( DEC_DIGIT | _ )*
DEC_DIGIT → [0-9]
```

Ein anderer Sprach-Standard (ECMA-Script, vereinfacht) definiert diese Tokenklasse:

```
DecimalDigits ::
  DecimalDigit
  DecimalDigits DecimalDigit
  DecimalDigits NumericLiteralSeparator DecimalDigi
DecimalDigit                :: one of     0 1 2 3 4 5 6
NumericLiteralSeparator ::  _
```

- Begründen Sie durch ein konkretes Beispiel: die Mengen `DEC_LITERAL` und `DecimalDigits` stimmen nicht überein.
- Welche Teilmengenbeziehung gilt und warum?

- Geben Sie die Original-Quellen (Sprach-Standards) an.
- Geben Sie die entsprechende Tokenklasse aus einem anderen Sprachstandard an und vergleichen Sie.

Syntaxbäume

Wort-Ersetzungs-Systeme

Berechnungs-Modell (Markov-Algorithmen)

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

Syntax: Programm ist Regelmenge $R \subseteq \Sigma^* \times \Sigma^*$,
Semantik: die 1-Schritt-Ableitungsrelation $\to_R$, Hülle $\to_R^*$

$$u \to_R v \iff \exists x, z \in \Sigma^*, (l, r) \in R : u = x \cdot l \cdot z \wedge x \cdot r \cdot z = v.$$

- Bubble-Sort: $\{ba \to ab, ca \to ac, cb \to bc\}$
- Potenzieren: $ab \to bba$ (Details: Übung)
- gibt es unendlich lange Ableitungen 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: $L(G) = \{w \mid S \rightarrow_R^* w \wedge w \in \Sigma^*\}$.

Formale Sprachen: Chomsky-Hierarchie

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

Tokenklassen sind meist reguläre Sprachen.
Syntax von Programmiersprachen meist kontextfrei.
Zusatzbedingungen (Bsp: Benutzung von Bezeichnern nur nach Deklaration) meist Teil der statischen Semantik (Menge der stat. korrekten Programme ist nicht kontextfrei)

Typ-3-Grammatiken

(= rechtslineare Gr.) jede Regel hat die Form

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

(vgl. lineares Gleichungssystem), Beispiel

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

Anwendung: Kommentare in Java `https://docs.oracle.com/javase/specs/jls/se25/html/jls-3.html#jls-3.7`

```
TraditionalComment:   |  CommentTail:
  / * CommentTail     |     * CommentTailStar
                      |     NotStar CommentTail
```

Sätze über reguläre Sprachen

Für jede Sprache L sind die folgenden Aussagen äquivalent:

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

Beweispläne:

- Grammatik $\leftrightarrow$ Automat (Variable $=$ Zustand)
- Ausdruck $\rightarrow$ Automat (Teilbaum $=$ Zustand)
- Automat $\rightarrow$ Ausdruck (dynamische Programmierung)

  $L_A(p, q, r) =$ alle Pfade von p nach r über Zustände $\leq q$.

Kontextfreie Sprachen

Def (Wdhlg): G ist kontextfrei (Typ-2), falls
$\forall(l,r) \in R(G) : l \in V^1$
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))` $\Rightarrow$ `(()())` $\Rightarrow aababb$

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

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

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

(erweiterte) Backus-Naur-Form

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

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

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

Erweiterte BNF

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

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

kann in BNF übersetzt werden

Ableitungsbäume für CF-Sprachen

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

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

Ableitungsbäume (II)

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

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

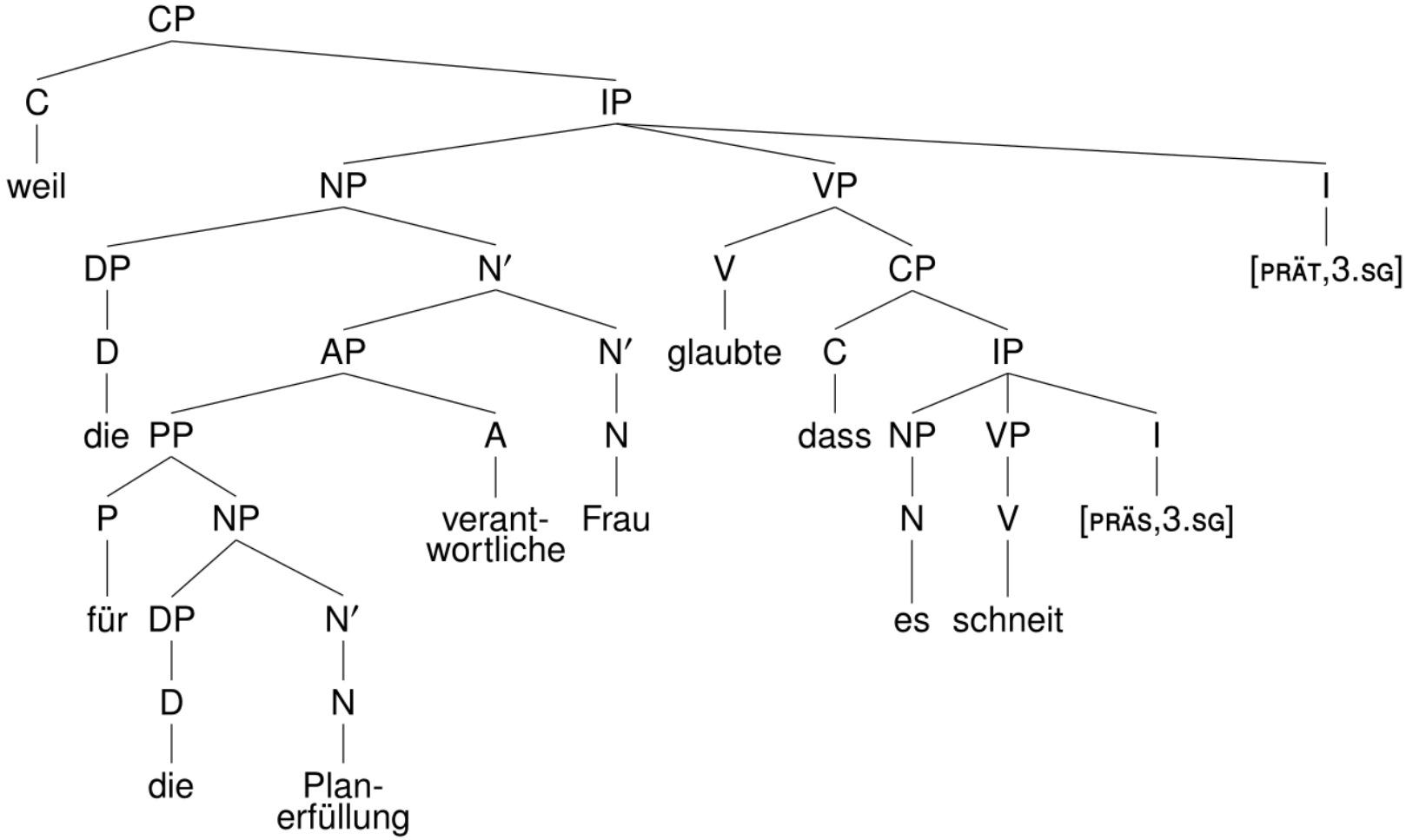
- Für Blätter: $\mathrm{rand}(b) = m(b)$,

- 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 $\mathrm{rand}(T, m) = w$.

Bsp. Ableitungsbaum (Deutsch)

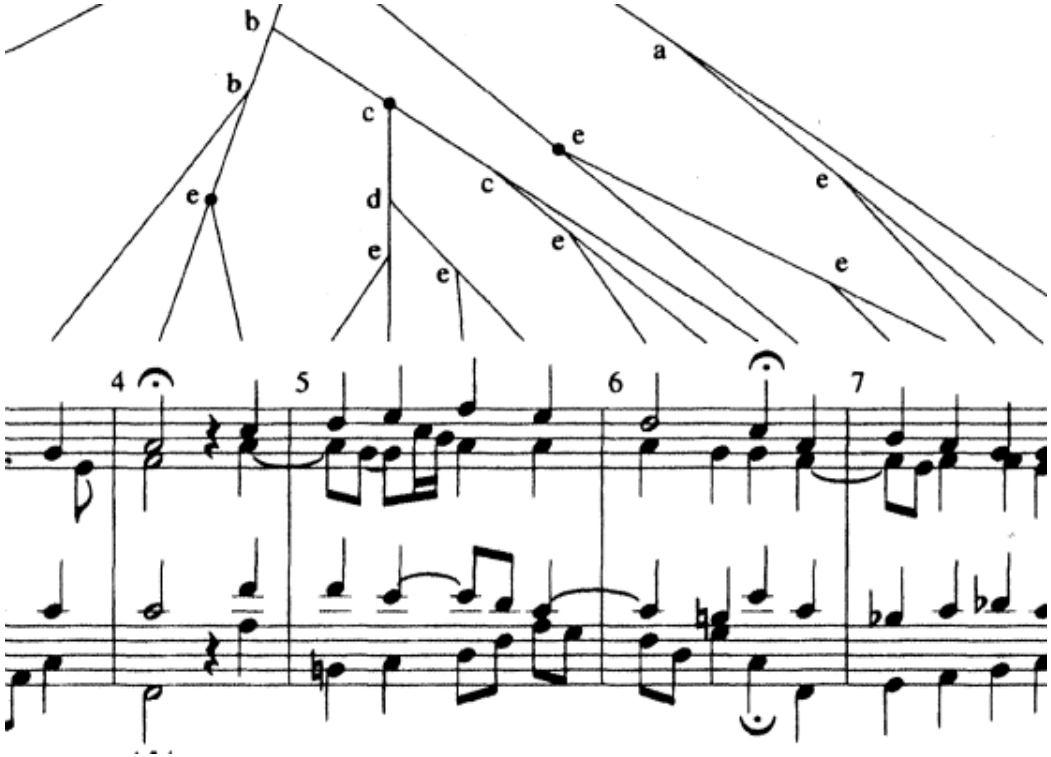
bezüglich einer Phrasenstrukturgrammatik des Deutschen



G. Müller, Linguistische Grundlagen (VL 8), 2009 `https://home.uni-leipzig.de/muellerg/1001/grundlagen.html`

Bsp. Ableitungsbaum (Musik)

Fred Lerdahl and Ray Jackendoff: *A Generative Theory of Tonal Music*, 1983



vgl. Hansen: *The Legacy of GTTM*, `http://www.dym.dk/dym_pdf_files/volume_38/volume_38_033_055.pdf` 2010

Eindeutigkeit

- Def: G heißt *eindeutig* : $\forall w \in L(G)\ \exists$ *genau ein* Ableitungsbaum (T, m) für G mit $\mathrm{rand}(T, m) = w$.

  Bsp: $(\{a, b\}, \{S\}, S, \{S \to aSb|SS|\epsilon\})$ ist mehrdeutig.

  (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

- (Wdhlg.) Definition: Operation ist *assoziativ*
- 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

  (d.h., eine äquivalente eindeutige Grammatik konstruieren, deren Ableitungsbäume die gewünschte Struktur haben)

Assoziativität (II)

- `X1 - X2 + X3` auffassen als `(X1 - X2) + X3`
- Grammatik-Regeln

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

- ersetzen durch

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

Präzedenzen

- Beispiel

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

- Grammatik-Regel

```
summand -> zahl
```

- erweitern zu

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

(Assoziativität beachten)

Zusammenfassung Operator/Grammatik

Ziele:

- Klammern einsparen
- trotzdem eindeutig bestimmter Syntaxbaum

Festlegung:

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

Realisierung in CFG:

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

Hausaufgaben

WS 26: 2, 4, 6

1. Definition: für ein Wortersetzungssystem R:

   Die Menge der R-Normalformen eines Wortes x ist:
   $\text{Nf}(R, x) := \{y \mid x \to_R^* y \wedge \neg\exists z : y \to_R z\}$

   Für das $R = \{ab \to baa\}$ über $\Sigma = \{a, b\}$:

   bestimmen Sie die R-Normalformen von

   - a^3b, allgemein a^kb,
   - ab^3, allgemein ab^k,

   die allgemeinen Aussagen exakt formulieren, für $k = 3$ überprüfen, durch vollständige Induktion beweisen.

2. Für Alphabet $\Sigma = \{a, b\}$, Sprache
$E = \{w : w \in \Sigma^* \wedge |w|_a = |w|_b\}$, Grammatik
$G = (\Sigma, \{S\}, S, \{S \to \epsilon, S \to SS, S \to aSb, S \to bSa\})$:

- Geben Sie ein $w \in E$ mit $|w| = 8$ an mit zwei verschiedenen G-Ableitungsbäumen.
- Beweisen Sie $L(G) \subseteq E$ durch strukturelle Induktion über Ableitungsbäume.
- Beweisen Sie $E \subseteq L(G)$ durch Induktion über Wortlänge. Benutzen Sie im Induktionsschritt diese Fallunterscheidung für $w \in E$: hat w einen nicht leeren echten Präfix u mit $u \in E$? Wenn ja, dann beginnt eine Ableitung für w mit $S \to SS$. Wenn nein, dann mit welcher Regel?

3. Vergleichen sie Definitionen und Bezeichnungen für *phrase structure grammars* Noam Chomsky: *Three Models for the Description of Language*, 1956, Abschnitt 3, `https://chomsky.info/articles/`, mit den heute üblichen (kontextfreie Grammatik, Ableitung, erzeugte Sprache, Ableitungsbaum)

   Erläutern Sie *The rule (34) . . . cannot be incorporated. . .* (Ende Abschnitt 4.1)

4. vergleichen Sie die Syntax-Definitionen von Fortran (John Backus 1956) und Algol (Peter Naur 1960),

   Quellen: *Historic Documents in Computer Science*, collected by Karl Kleine, `http://web.eah-jena.de/~kleine/history/`

(benutze Wayback Machine
`https://web.archive.org/`)

Führen Sie Kompilation und Ausführen eines Fortran-Programms vor (im Pool ist `gfortran` installiert, als Teil von GCC (GNU Compiler Collection))

Verwenden Sie dabei nur einfache Arithmetik und einfache Programmablaufsteuerung.

Geben Sie den Assembler-Code aus (Option `-S`). Vergleichen Sie mit Assembler-Code des entsprechenden C-Programms.

5. für die Java-Grammatik (nach JLS in aktueller Version)

- es werden tatsächlich zwei Grammatiken benutzt (lexikalische, syntaktische), zeigen Sie deren Zusammenwirken an einem einfachen Beispiel (eine Ableitung, bei der in jeder Grammatik nur wenige Regeln benutzt werden)
- bestimmen Sie den Ableitungsbaum (bzgl. der syntaktischen Grammatik) für das übliche `hello world`-Programm,
- Beispiele in `jshell` vorführen. Wie lautet die Grammatik für die dort erlaubten Eingaben? Ist das Teil der JLS? Wenn nein, finden Sie eine andere Primärquelle.

6. bzgl. der eindeutigen Grammatik für arithmetische Ausdrücke (aus diesem Skript):

- Ableitungsbaum für `1*2-3*4`
- Grammatik erweitern für geklammerte Ausdrücke, Eindeutigkeit begründen, Ableitungsbaum für `1*(2-3)*4` angeben

arithmetische Ausdrücke in Java:

- welche Variable der Java-Grammatik erzeugt arithmetische Ausdrücke?
- Ableitungsbaum für `1*(2-3)*4` von dieser Variablen aus angeben (und live vorführen durch Verfolgung der URLs der Grammatik-Variablen)
- Beziehung herstellen zu den Regeln auf Folie

„Zusammenfassung Operator/Grammatik“.

Semantik von Programmiersprachen

Statische und dynamische Semantik

- Definition:
  - Semantik $=$ Bedeutung
  - (vgl. Syntax $=$ Form)
- dynamische S. (beschreibt Ausführung des Programms)
  Beschr.-Methoden: operational, axiomatisch, denotational
- statische Semantik
  (Vorhersage der dyn. Semantik zur Übersetzungszeit)
  Beispiele (in C, Java, . . .)
  - Typ-Korrektheit von Ausdrücken,
  - deklarationsgemäße Benutzung von Bezeichnern

Bsp statische/dynamische Semantik

Benutzung eines nicht deklarierten Namens:

- ECMA-Script (Javascript): Programm wird ausgeführt, dynamische Semantik ist „Exception wird ausgelöst“

```
> {console.log("foo"); console.log(x);}
foo
Thrown:
ReferenceError: x is not defined
```

- Java: verhindert durch statische Semantik-Prüfung (Programm ist statisch falsch, wird nicht in Bytecode übersetzt, nicht ausgeführt, hat *keine* dyn. Sem.)

```
{ System.out.print("foo"); System.out.println(x);}
|  Error:  cannot find symbol
|    symbol:   variable x
```

Attributgrammatiken (I)

- Attribut: Annotation an Knoten des Syntaxbaums.

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

- Attributgrammatik besteht aus:
  - kontextfreier Grammatik G, Bsp: $(\ldots, \{S \rightarrow \epsilon \mid aSbS\})$
  - für jeden Knotentyp (Terminal + Regel)
    eine Menge (Relation) E von erlaubten Attribut-Tupeln
    $(A(X_0), A(X_1), \ldots, A(X_n))$
    für Knoten X_0 mit Kindern $[X_1, \ldots, X_n]$

- Beispiel: Terminale: $A(\epsilon) = A(a) = A(b) = 0$

  innere Knoten: $S \rightarrow \epsilon$, $A(X_0) = A(X_1)$;
  $S \rightarrow aSbS$, $A(X_0) = \max(1 + A(X_2), A(X_4))$;

Attributgrammatiken (II)

ein Ableitungsbaum T mit Annotationen A ist
korrekt bezüglich einer Attributgrammatik (G, E), wenn

- T ein Ableitungsbaum für G ist
- in jedem Knoten X_0 mit Kindern $[X_1, \ldots, X_n]$
  gilt $(A(X_0), A(X_1), \ldots, A(X_n)) \in E$.

Plan:

- Baum beschreibt Syntax, Attribute beschreiben Semantik

Ursprung: Donald Knuth: Semantics of Context-Free Languages, (Math. Systems Theory 2, 1968)
technische Schwierigkeit: Existenz und effiziente Berechnung der Attributwerte

Donald E. Knuth

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

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

Arten von Attributen

- synthetisiertes Attribut:

  hängt nur von Attributwerten in Kindknoten ab

  Bsp: Typ von Ausdrücken, Wert von Ausdrücken
- ererbtes (inherited) Attribut:

  hängt nur von Attributwerten in Elternknoten und (linken) Geschwisterknoten ab Bsp: deklarierte Typen für Namen
- Wenn Abhängigkeiten bekannt sind, kann man Attributwerte durch Werkzeuge bestimmen lassen.

  (Bransen et al.: *Linearly Ordered Attribute Grammar Scheduling . . .*, TACAS 2015

  `https://doi.org/10.1007/978-3-662-46681-0_24`)
- wir betrachten jetzt nur synthetisierte Attribute.

Attributgrammatiken–Beispiele

- Auswertung arithmetischer Ausdrücke (dynamisch)
  jedes Attribut ist eine Zahl
- Typprüfung (statisch)
  jedes Attribut ist ein Typ-Ausdruck
- Kompilation (für Kellermaschine) (statisch)
  jedes Attribute ist eine Befehlsfolge
- Bestimmung des abstrakten Syntaxbaumes
  jedes Attribut ist ein Baum
- alles diese Attr. sind synthetisiert, können durch Induktion über den Ableitungsbaum (d.h., von Blättern zu Wurzel) berechnet werden

Konkrete und abstrakte Syntax

- konkreter Syntaxbaum $=$ der Ableitungsbaum
- abstrakter Syntaxbaum (AST) $=$ wesentliche Teile des konkreten Baumes
- unwesentlich sind Knoten, die zu Hilfsvariablen gehören, die eingeführt wurden, damit Grammatik eindeutig ist
- abstrakter Syntaxbaum ist synthetisiertes Attribut:

```
E -> E + P  ;  E.abs = new Plus(E.abs, P.abs)
E -> P  ;  E.abs = P.abs  // kein neuer AST-Knoten
```

Typisierung von Funktionsaufrufen

- – Funktion f hat Typ $A \to B$
  – Ausdruck X hat Typ A
  – dann hat Ausdruck $f(X)$ den Typ B
- Notation als *Inferenz-Regel* $$\frac{f : A \to B \quad X : A}{f(X) : B}$$
- Beispiel

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

Bsp. Operationale Semantik: Keller

- Kellerspeicher
  - Zustand ist Zahlenfolge $s \in \mathbb{Z}^*$, Empty $= []$
  - Operationen:
    - Push(x), Semantik: $[s_1, \ldots, s_n] \rightarrow [x, s_1, \ldots, s_n]$
    - $y :=$ Pop$()$, Semantik: $[y, s_1, \ldots, s_n] \rightarrow [s_1, \ldots, s_n]$

- Realisierung zweistelliger Verknüpfungen: Argumente vom Keller holen, Resultat auf Keller schreiben, z.B.

  Plus $\equiv \{a :=$ Pop$(); b :=$ Pop$();$ Push$(a + b)\}$

- benutzt in Prog.-Spr. Forth (1970), PostScript (1982),

  JVM (Java Virtual Machine, 1994), Bsp: 6.5 `iadd`

Kompilation für Kellermaschine

- Spezifikation:
  - Eingabe: Java-Ausdruck A, Bsp. $3 * x + 1$
  - Ausgabe: JVM-Programm P, Bsp:

    `push 3; push x; imul; push 1; iadd;`
  - Zusammenhang: $[] \xrightarrow{P} [\text{Wert}(A)]$
  - dann gilt auch $\forall k \in \mathbb{Z}^* : k \xrightarrow{P} ([\text{Wert}(A)] \circ k)$

- Realisierung (Kompilation):
  - Code für Konstante/Variable c : `push c;`
  - Code für Ausdruck $x \circ y$: `code(x); code(y); o;`

  der so erzeugte Code ist synthetisiertes Attribut

- JVM-Programm (Bytecode) ansehen mit `javap -c`,

Hausaufgaben

WS 26: Aufgaben 1, 2, 3

1. arithmetische Ausdrücke (keine Programmablaufsteuerung), Beispiel

```
class C { static int f (int x) {return 3*x;
```

von Java nach Java-Bytecode übersetzen mit `javac` und Resultat betrachten mit `javap -c`.
Zeigen Sie durch ähnnliche Beispiele, daß richtig behandelt werden:

- Links-Assoziativität der Subtraktion
- Punkt- vor Strich-Rechnung

Vergleichen Sie den Bytecode mit dem Verfahren aus VL.
Schlagen Sie für einige der vorkommenden

Bytecode-Befehle die Semantik in der JVM-Spezifikation (aktuelle Version) nach.
Erläutern Sie die JVM-Befehle `dup`, `pop`. Geben Sie Java-Programme an, in dessen Bytecode diese vorkommen.

2. Stack-Operationen in der Programmiersprache FORTH (C. H. Moore 1970) (Interpreter `gforth-1.0.1` ist im Pool installiert)

   (a) Bestimmen Sie den Wert eines arithmetischen Ausdrucks (der dann in der Übung vorgegeben wird, Bsp. `(1 + 2)* 3`)
   durch FORTH-Befehlsfolge (Bsp. `1 2 + 3 * .`)

   (b) hier wird eine Befehlsfolge bezeichnet und danach mehrfach ausgeführt:

```
: f dup rot + ;
0 1 f f f f f f f f f f .
```

welche Zahl wird berechnet?

(c) Die Spezifikation von `rot` steht in `https:`
`//forth-standard.org/standard/core/ROT`
Erläutern Sie die dort vorgeschlagene reference implementation `: ROT >R swap R> swap ;`

3. Arithmetische Ausdrücke: abstrakter Syntaxbaum, Auswertung, Kombinator-Parser

```
data E = Literal Integer | Plus E E | Times E E

wert :: E -> Integer
wert e = case e of
  Literal i -> i
  Plus x y -> wert x + wert y
```

```
summe, produkt, faktor, zahl :: Parser E
summe = foldl Plus <$> produkt <*> many (char '+' *
produkt = faktor
faktor = zahl
zahl = (Literal . read) <$> many1 (satisfy isDigit)
```

vollständiger Quelltext (30 Zeilen) im Projekt-Repo

(a) Kompilieren, Testen (Addition)

(b) Multiplikation hinzufügen

(c) Klammern hinzufügen

Werbung: in Vorlesung Compilerbau (SS27):

- Interpretation für weitere Sprach-Elemente (Anweisungen, Unterprogramme)
- Implementierung des Parsers
- Kompilation (zu maschinennahem Code)

Typen

Das Ziel: statische Typisierung

- Typ ist Menge von Werten mit Operationen
  für jede eigene Menge von Werten aus dem
  Anwendungsbereich benutze einen eigenen Typ
- statische Typisierung: für jedes Teilprogramm
  den Typ *vor* (d.h., *ohne*) dessen Ausführung bestimmen
  - *Sicherheit*: findet Entwurfsfehler im Programm
  - *Effizienz*: verringert Platz (Typ-Angaben) und Arbeit
    (Typ-Prüfung) zur Laufzeit
- wir diskutieren Typisierung von Namen und Programmen
  später. Heute: Motivation und Typen von Daten.

Ausdrucksstarke Typsysteme: Anwendungen

- mit ausdrucksstarken Typsystemen kann man weitere Laufzeit-Arbeiten in die Übersetzungszeit verschieben,

- `https://git.imn.htwk-leipzig.de/waldmann/autotool`
  - Back-End: typsichere REST-API mit Servant, Mestanogullari et al., 2015 `https://www.servant.dev/`
  - Front-End mit Yesod, Snoyman et al.

    `https://www.yesodweb.com/`
    * statisch geprüftes Routing
    * compile-time templates (HTML, CSS)
    * statisch sicherer DB-Zugriff (ORM) mit

      `https://hackage.haskell.org/package/persistent`,

      `https://hackage.haskell.org/package/esqueleto`

Typ-Information und Laufzeitdaten

- jedes Datum wird zur Laufzeit des Programms im zugeordneten Speicher binär repräsentiert (als Bitfolge)
- richtige Verarbeitung ist nur möglich, wenn bekannt ist, welcher Typ dort repräsentiert wird
- dynamische Typisierung: der Typ steht in der Speicherstelle selbst (z.B. OO: jedes Objekt enthält Verweis auf seine Klasse)
- statische Typisierung: der Typ steht nicht im Speicher, sondern im Quelltext
- Mischform (in statisch typisierten OO Sprachen) keine Laufzeitrepräsentation von statischen Methoden

Historische Entwicklung

- keine Typen (nur ein Typ: alles ist Maschinenwort)
- vorgegebene Typen (Fortran: Integer, Real, Arrays)
- benutzerdefinierte Typen
  (algebraische Datentypen;
  Spezialfälle: enum, struct, class)
- abstrakte Datentypen (interface)
- polymorphe Typen (z.B. `List<E>`, Arrays, Zeiger)
- (data) dependent types (z.B. in Agda, Idris)

Überblick Datentypen

- einfache (primitive, nicht zusammengesetzte) Typen
  - Zahlen, Wahrheitswerte, Zeichen
  - benutzerdefinierte Aufzählungstypen
  - Teilbereiche
- zusammengesetzte (strukturierte) Typen
  - Produkt (record)
  - Summe (union) (Spezialfall: Aufzählungen)
  - rekursive Typen (Anwendung: Listen, Bäume)
  - Potenz (Funktionen): Unterprogramme, Arrays, (Tree/Hash-)Maps
    Verweistypen (Zeiger) als Speziallfall von Arrays
- Gleichheit (Zuweisungskompatibilität) von Typen: *nominell* (gleicher Name) oder *strukturell* (gleiche Form)

Zahlenbereiche

- Maschinenzahlen (oft im Sprachstandard festgelegt)
  - ganze Zahlen (in binärem Zweierkomplement)
  - gebrochene Zahlen (in binärer Gleitkommadarstellung) Goldberg 1991: *What Every Computer Scientist Should Know About Floating-Point Arithmetic*

    `https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`

- Abstraktionen (oft in Bibliotheken, Bsp. `https://gmplib.org//manual/`)
  - beliebig große Zahlen
  - exakte rationale Zahlen

Aufzählungstypen

können einer Teilmenge ganzer Zahlen zugeordnet werden

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

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

```
data Day = Mon | Tue | Wed | Thu | Fri | Sa
```

Ü: `enum` in Java

Designfragen:

- automatische oder manuelle Konversion zw. Aufzählungstyp und zugrundeliegendem Zahltyp

Maßeinheiten in F#

- physikalische Größe $=$ Maßzahl $\times$ Einheit.
- teure (Software)fehler durch Ignorieren der Einheiten.
  vgl. *Some Famous Unit Conversion Errors*, NASA,
  `https://spacemath.gsfc.nasa.gov/weekly/6Page53.pdf`
- in F# (Syme, 200?), aufbauend auf ML (Milner, 197?)

```
$ dotnet fsi
[<Measure>] type m ;; let x = 1.2<m> ;;
x * x ;;
[<Measure>] type s ;;  let y = 2.5<s> ;;
x * y ;; x + y ;;
```

- `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure`

Zeichen und Zeichenketten

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

  das ist historisch begründet (US-amerikanische Hardware-Hersteller, lateinisches Alphabet)

- das umfassende Modell ist `https://www.unicode.org/versions/Unicode14.0.0/` (insbes. Kapitel 2)

  jedes Zeichen wird durch *encoding scheme* (z.B. UTF8) auf *Folge* von code units (z.B. Bytes) abgebildet.

Zusammengesetzte Typen

Typ = Menge, Zusammensetzung = Mengenoperation:

- Produkt (record, struct), z.B.

```
data C = C { real :: Double, imag :: Double }
```

- disjunkte Summe (union, case class, enum), z.B.

```
data Ordering = LT | EQ | GT
```

- Rekursion, z.B.

```
data List a = Nil | Cons a (List a)
```

- Potenz (Funktion), z.B.

```
type Sorter a = (List a -> List a)
```

Produkttypen (Records)

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

```
typedef struct {
    A foo; B bar; C baz;
} R;
R x; ...  B y = x.bar; ...
```

- erstmalig in COBOL (1960) (Bromberg et al. 1960), basiert auf Flow-Matic (Hopper, 1959),

  https://archive.computerhistory.org/resources/text/Oral_History/Hopper_Grace/102702026.05.01.pdf)

Summen-Typen

- $R = A \cup B \cup C$
- disjunkte (diskriminierte) Vereinigung (Pascal, Niklas Wirth 1970)

```
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, Dennis Ritchie 1972):

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

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 (M. Odersky, 2004, `https://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()  }
```

Rekursive algebraische Datentypen

- Haskell (Simon Peyton Jones et al, 1990,

```
data Tree a = Leaf a
            | Branch ( Tree a ) ( Tree a )
```

- Java (James Gosling, 1995)

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

- `Tree a` ist ein *algebraischer Datentyp*:
  - die *Signatur* der Alg.: die Konstruktoren (Leaf, Branch)
  - die *Elemente* der Algebra sind Terme (Bäume)

Potenz-Typen

- $B^A := \{f : A \to B\}$ (Menge aller Funkt. von A nach B)
- Potenz ist sinnvolle Notation, denn $|B|^{|A|} = \left|B^A\right|$
- Realisierungen:
  - Funktionen (Unterprogramme)
  - Wertetabellen (Funktion mit endlichem Definitionsbereich) (Assoziative Felder, Hashmaps)
  - Felder (Definitionsbereich ist Aufzählungstyp) (Arrays)
  - Zeiger (Hauptspeicher als Array)
  - Zeichenketten (Strings)
- die unterschiedliche Notation dafür ist bedauerlich.

```
f(42);  f.get(42); f[42]; *(f+42); f.charAt(42)
```

Felder (Arrays)

- Realisierung einer Abbildung, Definitionsbereich ist Intervall von Zahlen, Wertebereich ist benutzerdefiniert.
- Motivation: Zugriff auf beliebiges Element in konstanter Zeit (unabhängig von Intervallgröße)

```
a[i] = * (a + w * i)
```

- Design-Entscheidungen:
  - welche Index-Typen erlaubt? (Zahlen? Aufzählungen?)
  - Bereichsprüfungen bei Indizierungen? (C:nein, Java:ja)
  - Allokation statisch oder dynamisch?
  - Index-Bereiche statisch oder dynamisch?
  - mehrdimensionale Felder (gemischt oder rechteckig)?

Felder in C

```
int main () {
    int a [10][10];
    a[3][2] = 8;
    a[2][12] = 5;
    printf ("%d\n", a[3][2]);
}
```

- statische Dimensionierung,
- dynamische Allokation,
- keine Bereichsprüfungen.
- Form: rechteckig, Adress-Rechnung:

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

Felder in Javascript

- die Notation `a[i]` wird verwendet für Felder (Zugriff über Index) *und* (Hash)Maps (Zugriff über Schlüssel).

- durch das Fehlen statischer Typisierung sowie implizite Umwandlung zwischen Zahl und Zeichenkette wird absurdes Verhalten spezifiziert, vgl. (2017) `https://news.ycombinator.com/item?id=14675706`

```
var arr1 = []; arr1[4294967296]=1;
  // arr1.length == 0
var arr2 = []; arr2[2147483647]=1;
  // arr2.length == 2147483648
var arr3 = []; arr3[-1]=1;
  // arr3.length == 0
```

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.
- Arrays sind immer eindimensional, aber man kann diese schachteln. (Kosten?)

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.
- moderne 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.

  `https://www.imn.htwk-leipzig.de/~waldmann/etc/safe-speed/`

Felder in C#

Übung: Unterschiede zwischen

- `int [][] a` geschachtelt (wie in Java)
- `int [,]  a` mehrdimensional rechteckig

in

- Benutzung (Zugriff)
- Konstruktion/Initialisierung

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

  `int x = 2 ; int *p = &x; ... *p + 3`
- implizite Verweise: Java:

  alle nicht primitiven Typen sind Verweistypen,
  De-Referenzierung ist implizit

  `Object a = ...; Object b = a;` kopiert Verweis
- C#: class ist Verweistyp, struct ist Werttyp

Verweis- und Wertsemantik in C#

- für Ausdrücke, deren Typ `class ...` ist:

  Verweis-Semantik, implizite Verweise (wie in Java)

- für Ausdrücke, deren Typ `struct ...` ist:

  Wert-Semantik, keine Verweise

- Testfall (hier `class` durch `struct` ersetzen)

```
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);
```

- ähnlicher Plan: `value class` für Java (JEP 401)

Algebraische Datentypen in Pascal, C

Rekursion unter Verwendung von Verweistypen

Pascal:

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

C: ähnlich, benutze typedef

Null-Zeiger: der Milliarden-Dollar-Fehler

- Tony Hoare (2009): [The null reference] has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

  (`https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-`

- Das Problem sind nicht die Zeiger selbst, sondern daß (in vielen Sprachen) der Wert `null` zu jedem Zeigertyp gehört — obwohl er gar kein Zeiger ist.

  Das ist die Verwechslung zwischen `t` und `Maybe t`.

  (`data Maybe t = Nothing | Just t`)

Hausaufgaben Typen

WS 26: (1 oder 2), (3 oder 4), Zusatz: (5 oder 7)

1. für Mengen

   $A = \emptyset, B = \{0\}, C = \{1, 2\}, D = \{3, 4, 5\}, E = \{6, 7, 8, 9\}$,

   geben Sie an:

   - alle Elemente von
     $A \times C, B \times D, A \cup B, B^A, A^B, C^B, B^C, C^D$
   - ein Element aus $(C \times D)^E$
   - die Kardinalitäten von $(C \times D)^E, C^{D \cup E}$

   ähnliche Aufgabenstellungen vorbereiten, die Sie dann in der Übung den anderen Studenten stellen.

2. Geben Sie eine Isomorphie zwischen den Mengen $(A^B)^C$ und $A^{(B \times C)}$ an.

   Illustrieren Sie das durch konkrete kleine endliche Mengen A, B, C.

   Diskutieren Sie auch die Fälle, daß A, B, C leer sind.

   Diese Isomorphie wird in Haskell durch die Funktion `curry` realisiert. Zeigen Sie das in ghci. Verwenden Sie dabei für A, B, C paarweise verschiedene Typen. Wie lautet die Umkehrfunktion?

   Begründen Sie, daß $(A^B)^C$ nicht immer isomorph ist zu $A^{(B^C)}$. In welchen Fällen besteht Isomorphie?

3. zur Folie „Felder in C“:

   Programm kompilieren, ausführen.

   Assembler-Code ausgeben und erklären (`gcc -S` oder `clang -S`)

   Unterschiede zwischen `-O0` und `-O3`?

4. zu Folie „Felder in Javascript“:

   das zitierte Beispiel vorführen (node), mit Verweisen auf Sprachstandard erklären.

   Untersuchen Sie die Aussage eines Kommentators: „Typescript prevents all of these errors“. (Lokal im Pool: `npm install typescript; npx tsc`, auch `ts-node` ist nützlich. Keine Online-Dienste verwenden.)

5. Erläutern und variieren Sie das Verhalten von

```
#include <stdio.h>
typedef union { double foo; long int bar; } U;
int main ()
  { U x;
    x.bar =   0; printf ("%ld\n", x.bar);
    x.foo = 7.0; printf ("%ld\n", x.bar);
  }
```

Wiederholen Sie dabei die Gleitkomma-Darstellung (genau — welche Bits bedeutet was?)

Fügen Sie zu der Vereinigung einen weiteren Typ der gleichen Länge hinzu, z.B. Array von Bytes;

sowie einen Typ anderer Länge, z.B. `float`.

6. zu Folie „Kosten der Bereichsprüfungen“ und dort angegebener Quelle:

   führen Sie den Testfall vor, analysieren Sie die Ausgabe des Disassemblers (im Pool installiert). Vergleichen Sie verschiedene JIT/JVM-Versionen.

   Schreiben Sie das äquivalente Matrix-Multiplikationsprogramm in C, betrachten Sie den Assembler-Code, vergleichen Sie.

7. Beispiele vorführen, Spezifikation zeigen (Primärquellen) zum Vergleich von

   - `Data.Maybe` (Haskell),
   - `java.util.Optional`,
   - nullable in C#