"00" 01" 02" 03" 04" 05" 06" 07" 08" 09" 0A" 0B" 0C" 0D" 0E" 0F"

"4C" 4D" 4E" 4F" 50" 51" 52" 53" 54" 55" 56" 57" 58" 59" 5A" 5B"

"4A" 71" 4B" 79

Prinzipien von Programmiersprachen
Vorlesung
Wintersemester 2007 – 2024

Johannes Waldmann, HTWK Leipzig

13. Dezember 2024

Einleitung

Organisatorisches

- nur für die erste Übung (vor der ersten Vorlesung),
- danach erscheint hier (Woche für Woche jeweils nach der VL) das reguläre Skript.
- alle Informationen zu meinen LV erreichen Sie von `https://www.imn.htwk-leipzig.de/~waldmann/`
- Lehrmaterial (z.B. Quelltexte) und Einteilung und Diskussion von Hausaufgaben: `https://gitlab.dit.htwk-leipzig.de/johannes.waldmann/pps-ws24`
- nächste Folie: Beispiel-Hausaufgaben

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
- (SS 22) Fortgeschrittene Konzepte in Progr.-Spr.
- Compilerbau: Implementierung v. Syntax u. Semantik

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/96/vorlesungen`
  – in Gruppen (je 3 Personen)
  Präsentation und Bewertung in Übung
  Koordination: Projekt (Wiki, Issues) `https://git.imn.htwk-leipzig.de/waldmann/pps-ws23`

- Klausur: 120 min, ohne Hilfsmittel

Literatur

- Skript, Aufgaben usw. erreichbar von `https://www.imn.htwk-leipzig.de/~waldmann/edu/`

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

WS 24: 7, 6, (4 oder 3)

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 of . . . 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?
- erklären Sie `https://xkcd.com/378/`, führen Sie die vier genannten Editoren vor, in dem Sie jeweils eine einzeilige Textdatei erzeugen.

Bei Vorführung (Screen-Sharing/Projektion):

große (Control-Plus), *schwarze* Schrift auf weißem Grund

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 3) 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!
- nennen Sie einige Aspekte von PS, die in PDF übernommen wurden (Beantworten Sie anhand der Original-Dokumentation.)
- Warum sollte man niemals “online und ganz umsonst PS to PDF converter” benutzen?

6. In SICP 1.1 werden drei *Elemente der Programmierung* genannt. Illustrieren Sie diese Elemente durch Beispiele aus `https://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.

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.
- 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: `^*`)

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. Fragen dazu notfalls im Git.lmn-Tracker.

Hausaufgaben

WS 24 (Ü KW 44) 2, (3 oder 4 oder 5), 7, (optional 8)

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).

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 Grammatiken)

jede Regel hat die Form

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

(vgl. lineares Gleichungssystem)

Beispiele

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

Sätze über reguläre Sprachen

Für jede Sprache L sind die folgenden Aussagen äquivalent:

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

Beweispläne:

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

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

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

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 24 (Ü in KW 45) 2, (3 oder 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`,

Attributgrammatiken mit SableCC

- Etienne Gagnon, 1998–, `https://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
```

- Bespiel: siehe Verzeichis `pps-ws23/rechner`

  Benutzung: `make ; make test ; make clean`

- 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/*`

Bemerkungen (häufige/nicht offensichtliche Fehlerquellen)

- `Redefinition of ...` : nicht so:
  `foo -> bar ; foo -> baz;` sondern so:
  `foo -> {eins} bar | {zwei} baz;`

Regeln mit gleicher linker Seite zusammenfassen,

die rechten Seiten durch Label (`{eins},{zwei}`) unterscheiden

- `... conflict ...`:

  die Grammatik ist nicht eindeutig (genauer: wird von Sablecc nicht als eindeutig erkannt)

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

Kombinator-Parser:

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

Auswertung arithmetischer Ausdrücke

(das ist ungefähr die erste VL Compilerbau)

- abstrakter Syntaxbaum (AST)

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

- Auswertung (Rekursion über AST, ist ein Fold)

```
value :: Exp -> Integer
value e = case e of
  Literal i -> i
  Plus  l r -> value l + value r
  Times l r -> value l * value r
```

Kombinator-Parser f. arith. Ausdrücke

- Daan Leijen: *Parsec, a fast combinator parser*, 2001,
  `https://web.archive.org/web/20140528151730/http://legacy.cs.uu.nl/daan/download/parsec/parsec.pdf`
- `https://hackage.haskell.org/package/parsec-3.1.14.0/docs/Text-Parsec-Expr.html`

```
expr    = buildExpressionParser table term
term    =  parens expr  <|> natural
table   = [ [ binary "*" (*) AssocLeft
            , binary "/" (div) AssocLeft ]
          , [ binary "+" (+) AssocLeft
            , binary "-" (-)   AssocLeft ] ]
binary  name fun assoc =
  Infix (reservedOp name >> return fun) assoc
```

- ist *embedded* (in Haskell) DSL

Hausaufgaben

WS 24: Aufgabe 1, 2

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. zum angegebenen Beispiel Sablecc

- Test vorführen.
- das dabei verwendete Makefile erklären.
  Was ist die Semantik der Ziele und Regeln eines Makefiles? Was ist bei der Syntax zu beachten? (Hinweis: ein besonderer Whitespace)
- Grammatik ergänzen: Multiplikation.

Eindeutigkeit der Grammatik und semantisch korrekte Auswertung vorführen und begründen.

- (in der Übung, jeder selbst) Subtraktion, Klammern.

3. (Zusatz) Generalized Algebraic Data Types (ein Thema aus OS FKPS SS22)

   Verwenden/ergänzen Sie diesen AST-Typ

```
{-# language GADTs #-}
data Exp a where
  Literal :: Integer -> Exp Integer
  Plus ::
    Exp Integer -> Exp Integer -> Exp Integer
  Greater ::
    Exp Integer -> Exp Integer -> Exp Bool
  Ifthenelse :: Exp Bool -> ...
```

Erklären Sie den Fehler in

```
Ifthenelse (Literal 0) (Literal 1) (Literal
```

Rufen Sie `Ifthenelse` typkorrekt auf.

Passen Sie den Interpreter (die Funktion `value`) an.

4. (Zusatz) Kombinatorparser (ein Thema aus VL Compilerbau SS22)

   einfache Beispiele vorführen und erklären (elementare Parser `char`, `eof`; Kombinatoren `(>>)`, `many`, `sepBy`; ggf. `buildExpressionParser`)

```
cabal install --lib parsec
ghci
import Text.Parsec
parseTest (many (char 'f') >> many (char 'o')) "fool
```

Typen

Der Nutzen der statische Typisierung

- Typ ist Menge von Werten mit Operationen
  für jede eigene Menge von Werten aus dem
  Anwendungsbereich benutze einen eigenen Typ
- statische Typisierung gibt
  - *Sicherheit*: findet Entwurfsfehler im Programm
  - *Effizienz*: verringert Platz (Typ-Angaben) und Arbeit (Typ-Prüfung) zur Laufzeit
- mit ausdrucksstarken Typsystemen kann man weitere Laufzeit-Arbeiten in die Übersetzungszeit verschieben, Konstruktion von Wörterbüchern f. Typklassen in Haskell

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

- einfache (primitive) 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

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.
- 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 ;;
```

- `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 $\left|B\right|^{|A|} = \left|B^A\right|$
- 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 24: (1 oder 2), (3 oder 4), Zusatz: (5 oder 6 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#

Bezeichner, Bindungen, Bereiche

Variablen

- vereinfacht: Variable bezeichnet eine Speicherstelle
- genauer: Variable besitzt Attribute
  - Name
  - Adresse
  - Wert
  - Typ
  - Lebensdauer
  - Sichtbarkeitsbereich
- Festlegung 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

Konkrete Syntax von Namen

- . . . wird definiert durch die Tokenklasse *Bezeichner*
- 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 = ...`

Deklaration und Definition

- Bsp: `int x = 8;`
  `int x` ist Deklaration, `= 8` ist Definition
- Bsp: `static int f(int y) { return y+1; }`
  `static int f(int y)` ist Deklaration,
  `(int y) { return y+1; }` ist Definition.
- Deklaration:
  - statische Semantik: der Name ist ab hier sichtbar
  - dynamische S.: dem Namen ist Speicherplatz zugeordnet
- Definition:
  - dynamische Semantik: dem Namen ist Wert zugeordnet
  - statische S.: (siehe „garantierte Initialisierung“ später)

Typen für Namen

- dynamisch (Wert hat Typ)
- statisch (Name hat Typ)
  - deklariert (durch Programmierer)
  - inferiert (durch Übersetzer)
    z. B. `var` in C#
- Vor/Nachteile: Lesbarkeit, Sicherheit, Kosten

  der Typ eines Namens ist seine beste Dokumentation

  (weil sie maschinell überprüft wird - bei statischer Typisierung)

Dynamisch typisierte Sprachen

- Daten sind typisiert, Namen sind nicht typisiert.
- LISP, Clojure, PHP, Python, Perl, Javascript, ...
- ```
  let foo = function(x) {return 3*x;};
  foo(1);
  foo = "bar";
  foo(1);
  ```
- Ü: zum Vergleich: dieses Beispiel in Typescript (statisch typisiert)
  ```

Statisch typisierte Sprachen

- Namen sind typisiert, Daten sind typisiert (? siehe unten)
- Invariante:

  zur Laufzeit ist der *dynamische Typ* des Namens (der Typ des Datums auf der durch den Namen bezeichneten Speicherstelle)

  immer gleich dem *statischen Typ* des Namens
- woher kommt der statische Typ?
  - Programmierer deklariert Typen von Namen (C, Java)
  - Compiler inferiert Typen von Namen (ML, C# (var))
- dynamischer Typ muß zur Laufzeit nicht repräsentiert werden (das spart Platz u. Zeit): Compiler erzeugt Code, der das Resultat der Laufzeittypprüfung vorwegnimmt.

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

- für lokale Variablen in C#, Java: `var`

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

- Ü: dieses `var` ist nicht das `var` aus Javascript.

- für formale Parameter von anonymen Unterprogrammen

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

- Typ von `f` wird nicht inferiert: `var f = (x)-> x`

Code-Inferenz

- in vielen einfachen Sprachen dienen Typen tatsächlich „nur“ zur Spezifikation und Dokumentation

  . . . man könnte sie also doch weglassen, wenn man nur die Implementierung selbst braucht?

- moderne, ausdrucksstarke Typsysteme nützen deutlich mehr und tragen auch zur *Code-Erzeugung* bei.

  Sandy Maguire: `https://thinkingwithtypes.com/`

- Anwendungen/Beispiele (u.a. in autotool)
  - typgesteuerte Testdatenerzeugung, Rudy Matela, 2017, `https://hackage.haskell.org/package/leancheck`
  - *Type-Level Web APIs with Servant*, Alp Mestanogullari et al., 2015, `https://www.servant.dev/`

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 ++; }
```

- alle Deklarationen so lokal und so konstant wie möglich!

  (d. h., Attribute *immutable* usw.)

  denn das verringert den Umfang der Dinge, über die man nachdenken muß, um das Programm zu verstehen

Lebensort und -Dauer von Name und Daten

- statisch (auf statisch zugeordneter Adresse im Hauptspeicherbereich)

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

- dynamisch (auf zur Laufzeit bestimmter Adresse)
  - Stack (Speicherbereich für Unterprogramm-Aufruf)

    `{ int x = ... }`
  - Heap (Hauptspeicherbereich)
    - explizit (new/delete, malloc/free)
    - implizit (kein delete, sondern automatische Freigabe)

- Beachte (in Java, C#) in `{ C x = new C(); }` ist `x` stack-lokal, Inhalt ist Zeiger auf das heap-globale Objekt.

Sichtbarkeit von Namen

- eine Deklaration ist sichtbar, wenn die Verwendung des Namens ein Bezug auf die deklarierte Variable ist
- üblich ist: Sichtbarkeit beginnt nach Deklaration und endet am Ende des umgebenden Blockes.
- Import-Deklarationen machen Namen aus anderen Namensbereichen sichtbar
- (Java) ohne Import-D. besteht *qualifizierte* Sichtbarkeit
- (C): Sichtbarkeit beginnt in der Initalisierung

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

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

Verdeckung von Deklarationen

- 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 in JavaScript

- Namen sind sichtbar
  - Deklaration mit `var`: im (gesamten!) Unterprogramm

    ```
    (function() { { var x = 8; } return x; } )
    ```
  - Deklaration mit `let`: im (gesamten!) Block

    ```
    (function() { { let x = 8; } return x; } )
    ```
- Ü: erkläre (durch Verweis auf Sprachspezifikation)

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

Hausaufgaben

1. Beobachten und erklären Sie die Ausgabe von

```
#include <stdio.h>
int main (int argc, char **argv) {
  int x = 3;
  { printf ("%d\n", x);
    int x = 4;
    printf ("%d\n", x);
  }
  printf ("%d\n", x);
}
```

schreiben Sie ein entsprechendes Java-Programm und vergleichen Sie (statische und dynamische Semantik:

experimentell und mit Sprachspezifikation)

2. Sichtbarkeit von Deklarationen in Javascript.

   Siehe Folie, Original-Dokumentation zeigen und Beispiele vorführen (node), ergänzen durch weitere Beispiele mit nicht offensichtlicher Semantik, Bsp: Variablen-Deklaration in einem Zweig einer Verzweigung.

   nur Sichtbarkeiten — Programmablaufsteuerung soll trival sein (keine Schleifen, keine Unterprogramme)

3. frühere Folie *Verweis- und Wertsemantik in C#*: den angegebenen Testfall durchführen (mit Mono C# Shell, `csharp`), Lebensort (Stack, Heap) der Daten angeben, dann `class` durch `struct` ersetzen.

Ausdrücke

Definition, Abgrenzung

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

Vgl. Trennung (in Pascal, Ada)

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

Ü: wie in Java ausgedrückt? wie stark getrennt?

Syntax von Ausdrücken

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

wichtige Spezialfälle für Operatoren:

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

Wdhlg: Syntaxbaum, Präzedenz, Assoziativität.

Designfragen für Ausdrücke

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

Beziehungen zw. Ausdruck und Anweisung

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

```
int x = 3; int y = 4; int z = x + (y = 5);
```

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

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

Überladene Operatornamen

- Def: Name n *überladen*, falls n mehrere Bedeutungen (gleichzeitig sichtbare Deklarationen) hat
- in vielen Sprachen sind arithmetische und relationale Operatornamen überladen . . .

  weil das Typsystem keine flexiblere Lösung gestattet, wie z.B. `class Num a where (+) :: a -> a -> a`
- Überladung wird statisch aufgelöst (vom Compiler, anhand der Typen der Argument-Ausdrücke)
- ```
  int x = 3; int y = 4; ... x + y ...
  double a;  double b; ... a + b ...
  String p;  String q; ... p + q ...
  ```
  ```

Automatische Typanpassungen

- in vielen Sprachen postuliert man eine Hierarchie von Zahlbereichstypen:

  $\text{byte} \subseteq \text{int} \subseteq \text{float} \subseteq \text{double}$

  im allgemeinen ist das eine Halbordnung.

- Operator mit Argumenten verschiedener Typen:

  ```
  (x :: int) + (y :: float)
  ```

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

  (Halbordnung $\rightarrow$ Halbverband)

- (das ist die richtige Benutzung von *promovieren*)

Wahrheitswerte in C, C++

- der Typ der Wahrheitswerte ist `bool`

  (in C: `#include <stdbool.h>`)

- mit impliziter Konversion:
  - zu `int`: false $\rightarrow 0$, true $\rightarrow$ 1
  - von `int`: 0 $\rightarrow$ false, alles andere $\rightarrow$ true

- ```
  bool x = false; bool y = true; bool z = true;
  int a = x + y + z;
  int b = x || (y + z);
  ```
  ```

Der Plus-Operator in Java

- hat diese Überladungen: Addition von `int`, Addition von `double`, ..., Verkettung von `String`

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

- Vorgehen für die Analyse:
  - abstrakten Syntaxbaum bestimmen
  - Typen (als Attribute der AST-Knoten) bestimmen,
  - dabei implizite Typ-Umwandlungen einfügen (in diesem Fall `Integer.toString()`)
  - Werte (als Attribute) bestimmen
  ```

Explizite Typumwandlungen

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

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

```
int x = 4; double y = (double) x / 5;
/* Ü : */ double z = (double) (x / 5) ;
```

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

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

Typumwandlungen in Haskell

- Joachim Breitner et al.: *Safe zero-cost coercions for Haskell*, JFP 2016, `https://dblp.org/rec/journals/jfp/BreitnerEJW16.html`

  Umwandlung zwischen Basistyp und abgeleitetem Typ mit gleicher Laufzeit-Repräsentation

  – sicher: durch Compiler bewiesen
  – kostenlos: keine Laufzeitkosten

```
newtype Foo = Foo Int
data Bar a = Bar Bool a
xs = replicate 10 (Bar True (Foo 3)) :: [Bar Foo]
ys = Data.Coerce.coerce xs :: [Bar Int]
```

Der Zuweisungs-Operator

- Syntax:
  - Algol, Pascal: Zuweisung $:=$, Vergleich $=$
  - Fortran, C, Java: Zuweisung $=$, Vergleich $==$
- Semantik der Zuweisung `a = b`:
  - bestimme Adresse (lvalue) p von `a`
  - bestimme Wert (rvalue) v von `b`
  - schreibe v auf p
- diese Ausdrücke haben einen lvalue:
  - Variablen
  - `a[i]`, mit: rvalue von a ist Array, rvalue von i ist Index
  - `o.a`, mit: rvalue von o ist Objekt mit Attribut a

  Bsp: `foo()[bar()]`

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

- Ü: experimentell bestätigen, daß lvalue des Zuweisungsziels nur einmal ausgewertet wird

Teil-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) – vgl. C mit Java

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

Auswertungsreihenfolgen

- zusammengesetzte Programme können mehrere Bestandteile mit Nebenwirkungen haben.

  In welcher Reihenfolge finden diese statt?

  – Anweisungen: explizite Programm-Ablauf-Steuerung

  ```
  { int a = 5; a = 6; int b = a + a; }
  ```

  – Ausdrücke?

  ```
  { int a; int b = (a = 5) + (a = 6); }
  ```

- C, C++: Reihenfolge nicht spezifiziert, wenn Wert davon abhängt, dann ist Verhalten *nicht definiert*

- Java, C#: Reihenfolge genau spezifiziert (siehe JLS)

Ausdrucks-Semantik von C

- Sprachstandard benutzt Begriff *sequence point* (Meilenstein): bei Komma, Fragezeichen, `&&` und `||`
- Nebenwirkungen zwischen Meilensteinen müssen *unabhängig* sein (nicht die gleiche Speicherstelle betreffen),
- ansonsten ist das Verhalten *undefiniert*, d.h., der Compiler darf *beliebigen* Code erzeugen, z.B. solchen, der die Festplatte löscht oder Cthulhu heraufbeschwört.
- vgl. Aussagen zu sequence points in `https://gcc.gnu.org/readings.html` und Gurevich, Huggins: *Semantics of C*, `https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.6755`

Logische (Boolesche) Ausdrücke

- Konjunktion `&&`, Disjunktion `||`, Negation `!`

  Äquivalenz, Antivalenz

- *verkürzte Auswertung* für Konjunktion und Disjunktion: wenn nach Auswertung des linken Arguments das Resultat feststeht, denn werte rechtes nicht aus

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

  dann tritt dessen Nebenwirkung (o. Exception) nicht auf

- warum keine verkürzte Auswertung für Äquiv., Antiv.?

Der ternäre Verzweigungs-Operator ?:

- ```
  if ( 0 == x % 2 ) { x = x / 2; }
  else {  x = 3 * x + 1; }
  ```
- die Zuweisung herausfaktorisieren:

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

- historische Notation dafür benutzt ternären Operator `?:`

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

- `(x && y)` $\equiv$ `(x ? y : false)`, `(x || y)` $\equiv$ ...
- Verzweigungs-Operator auf lvalues (C++):

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

Übungen

1. Gary Bernhardt: WAT (2012) `https://www.destroyallsoftware.com/talks/wat`
2. Wiederholung Operator-Syntax:
   - ist die Mengendifferenz assoziativ?
   - vgl. `https://gitlab.haskell.org/ghc/ghc/issues/15892` „the fix was to add a pair of parentheses“
3. Was spricht dafür und dagegen, daß in einem Programmtext neue Operatoren definiert werden?

   In C++ darf man keine neuen Operatoren deklarieren, aber vorhandene Operatoren neu implementieren. Begründen Sie diese Design-Entscheidung.

Hausaufgaben

WS 24: 5, 3, 4

1. Konversion von `int` nach `float` in Java:

(a) Es gilt nicht int $\subseteq$ float, denn:
- beide Mengen sind gleich groß (wie groß?)
- und es gibt (viele) $y \in$ float $\setminus$ int (welche?)

(b) Geben Sie ein $x \in$ int $\setminus$ float explizit an.
(eine ganze Zahl, die keine exakte Darstellung als `float` besitzt)

(c) Wo ist diese Konversion in der Sprachspezifikation (JLS) beschrieben?

(d) desgleichen für `long` zu `double`

(e) Gilt int $\subseteq$ double? (nach JLS, nach IEEE-Standard)

2. durch Verweis auf JLS erklären:

- ```
  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);
  ```
- ```
  long x = 1000 * 1000 * 1000 * 1000;
  long y = 1000 * 1000;
  System.out.println ( x / y );
  ```
- ```
  System.out.println
      ((int) (char) (byte) -1);
  ```

  Quelle: Joshua Bloch, Neil Gafter: *Java Puzzlers*, Addison-Wesley, 2005.

3. statische Semantik (Typisierung) und dynamische

Semantik (Auswertung) dieses Programms (in C, in Java)

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

dazu den AST für `a < b < c` zeichnen und annotieren.

4. UB (undefined behaviour) für C-Ausdrücke mit abhängigen Teilausdrücken zwischen Sequence Points:

(a) Finden Sie C- oder C++- Programme, bei denen

- verschiedene Compiler (gcc, clang, g++, clang++)
- ein Compiler bei verschiedenen Optionen (`-O0`, `-O3`)
- verschiedene Versionen eines Compilers (im Pool: verschiedene gcc sind installiert)

unterschiedliche Semantik realisieren. Beispiel:

```
int y = 1; int z = (y=2) + (y=3);
```

(b) Wo ist dieses (undefined) Verhalten im (draft) C++-Standard beschrieben?
(`http://www.open-std.org/jtc1/sc22/wg21/`)

(c) Vergleichen Sie mit Semantik des entsprechenden Java-Programms. (Ausführen, Bytecode ansehen, Sprachspezifikation)

(d) Wer ist Cthulhu, wo wohnt er (derzeit), was hat er vor? Seine Beziehung zu Semantik von C-Programmen ist Folklore (kann nur durch Internet-Quellen belegt werden).

5. Verkürzte Auswertung bei logischen Operatoren in Java und JS (Tests mit `jshell`, `node`)
   (a) einen Testfall angeben, der die verkürzte Auswertung bei `||` zeigt.
   (b) Der Operator `|` verknüpft Zahlen bitweise. (Testfall angeben) Es gibt `|` auch für `boolean`. Worin besteht der Unterschied zu `||` ? (Testfall angeben)
   (c) desgl. für `&`
   (d) das gleiche für JS oder TS untersuchen
6. Verkürzte Auswertung bei logischen Operatoren in Ada: Sprachstandard und Vorführung. Benutze GNAT (GNU Ada Translator) als Teil von GCC (GNU Compiler Collection), ist im Pool installiert

Anweisungen

Definition (Wiederholung)

- abstrakte *Syntax*:
  - einfache Anweisung:
    - leere Anweisung (`skip`), Zuweisung (`l:=r`),
    - Sprung `goto`, `break`, `continue`, `return`, `throw`
    - Unterprogramm-Aufruf (`p(a,b)`)
  - zusammengesetzte Anweisung:
    - Nacheinanderausführung (Block)
    - Verzweigung (zweifach: `if`, mehrfach: `switch`)
    - Wiederholung (Schleife)
- *Semantik*: Ausführen einer Anweisung bewirkt Zustandsänderung

Blöcke

- Def: Folge von (Deklarationen und) Anweisungen
- Designfrage/historische Entwicklung: Deklarationen . . .
  - am Beginn des Progr. (Assembler, COBOL, Fortran)
  - am Beginn jedes Unter-Programms (Pascal)
  - am Beginn jedes Blocks (C)
  - an jeder Stelle jedes Blocks (C++, Java)
- Designfrage für Syntax: Blöcke
  - explizit (Klammern, `begin`/`end`)
  - implizit (`if` ... `then` ... `end if` , d.h., ohne `begin`)

Verzweigungen (zweifach)

- in vielen Sprachen:

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

- Designfrage (Syntax und Semantik): Bedingung ist . . .
  - beliebiger Ausdruck mit Typ Wahrheitswert
  - nur Vergleich zwischen Ausdrücken vom Typ Zahl
- Designfrage Syntax: Mehrdeutigkeit der Grammatik
  - gelöst durch Festlegung (else gehört zu letztem if)
  - vermieden durch Block-Bildung (Ada)
  - tritt nicht auf, weil man `else` nie weglassen darf, weil beide Zweige einen Wert liefern (`?:`, Haskell)

Mehrfach-Verzweigung

Syntax:

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

Semantik

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

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

switch/break

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

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

Verzweigungen in Ausdrücken

- zweifach-Verzweigung in C-ähnlichen Sprachen:

  Ausdruck vom Typ `int` mit Wert `11`:

  ```
  false ? 12 : 11
  ```

- Mehrfach-Verzweigung (*switch expression*) in Java (21)

  Ausdruck vom Typ `int` mit Wert `1`:

  ```
  switch (3) {case 0 -> 0; default -> 1;}
  ```

- warum? *nur* nebenwirkungsfreie Programme sind leicht zu spezifieren, zu testen, zu komponieren, zu parallelisieren $\Rightarrow$ Ausdrücke, nicht Anweisungen $\Rightarrow$ funktionale Programmierung

Kompilation der Mehrfachverzweigung

ein switch (mit vielen cases) wird übersetzt in:

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

Übung:

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

Pattern Matching (Def., Bsp. Scala)

- Fallunterscheidung nach Konstruktor (Bsp: `Const`,`Plus`) und Bindung von lokalen Namen (im Bsp: `l`,`r`)

- ```
  data Term = Constant Int | Plus Term Term -- Haskel
  eval :: Term -> Int
  eval t = case t of
    Constant i -> i
    Plus l r -> eval l + eval r
  ```

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

Pattern Matching (Bsp. Java)

- ein Muster, dessen lokale Variable `s` hat Typ `String` und ist im Ja-Zweig sichtbar:

```
Object o = "foo";
(o instanceof String s) ? s.length() : 42
```

- mehrere Muster, Mustervariable in jeweiligem Zweig *und* (vorher schon) Bedingung sichtbar

```
switch (o) { case String s when s.length() > 2 -> 4
  case Integer i -> 0; default -> 2;}
```

- Benennung von Record-Komponenten

```
record R (int x, String y) {}
switch (new R(2,"foo")) {
  case R(int x, String y ) -> x; ...}
```

Wiederholungen (Schleifen)

- Programmablaufsteuerung von Wiederholungen:
  - von-Neumann-Modell (Maschine, Assembler): unbedingter, bedingter Sprung
  - strukturierte Programmierung: Schleifen
- wie beweist man Programm-Eigenschaften?
  - partielle Korrektheit: mit Invariante
  - Termination: mit Maßfunktion
- Designfragen für Schleifen:
  - wie wird Schleife gesteuert? Bedingung, Zähler, Zustand (Iterator), Daten (Collection)
  - an welcher Stelle in der Schleife findet Steuerung statt (Anfang, Ende, dazwischen, evtl. mehrere Stellen)

Schleifen steuern durch...

- Zähler

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

- Daten

```
map (\x -> x*x) [1,2,3] ==> [1,4,9]

Collection<String> c
    = new LinkedList<String> ();
for (String s : c) { ... }
```

- Bedingung

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

- Zustand (Iterator, hasNext, next)

Zählschleifen

- Idee: vor Beginn steht Anzahl der Durchläufe fest.
  Maßfunktion $=$ Abstand des akt. Zählerwertes zum Ende
- richtig realisiert ist das nur in Ada:

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

  - Zähler `p` wird implizit deklariert
  - Zähler ist nur im Schleifenkörper sichtbar
  - Zähler ist im Schleifenkörper konstant
  - Zählerstand nur implizit d. Schleifensteuerung geändert
  - Ausdrücke für Bereichsgrenzen werden nur einmal (vor Betreten der Schleife) ausgewertet
- Vergleiche (beide Punkte) mit Java, C++, C

Datengesteuerte Schleifen

- die Zählschleife ist schon ein *code smell* (Anzeichen für unzweckmäßige Programmierung),
- der eigentliche *smell* ist die Verwendung von Zahlen (!)
- weil man (wegen verfrühter Optimierung) über Indizes spricht statt über die indizierten Werte

```
T [] a; for (int i = ...) { ... a[i] ... }
```

- Notation zur Vermeidung von Indizes bei Verarbeitung *aller* Elemente einer Datenstruktur

```
for (T x : a) { ... x ... }
```

mit Indizes, die gar nicht dastehen, kann man auch keine Indexfehler machen (z.B. off-by-one)

Zustandsgesteuerte Schleifen

- Iterator repräsentiert Strom von Daten
  (Stream = Liste mit bedarfsweiser Auswertung)

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

- Iterator-Objekt ist oft Index(Variable) mit Verweis auf zugrundeliegende Struktur. Das vermeidet Risiken wie:
  `int i; int j; int [] a; int [] b; .. a[j]`
- Iterator ist hier implizit (über den Wert einer Variablen, die gar nicht dasteht, muß man nicht nachdenken)

```
Iterable <T> c; for (T x : c) { ... }
```

Implizite Iteratoren in C#

- durch diese Konstruktion wird ein Iterator angelegt:

```
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);  } } }
```

- der markierte Block ist eine Co-Routine, seine Ausführung ist mit der des Hauptprogramms verschränkt.
- Coroutinen in Simula (1967), siehe: Ole-Johan Dahl, C. A. R. Hoare: *Hierarchical Program Structures*, 1972

  `https://dl.acm.org/doi/book/10.5555/1243380`

Bedingungsgesteuerte Schleifen

- das ist die allgemeinste Form, ergibt (partielle) rekursive Funktionen,

  (zum Vergleich: Programme mit Zählschleifen $=$ primitiv rekursive Funktionen)

- 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

Dynamische Semantik von Schleifen

- operationale Semantik durch Sprünge (autotool-Aufgabe)

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

- Ü: `do A while (B);`
- diese Programme sind semantisch äquivalent:

  `while (B) A;` und `if (B) { A; while (B) A }`
  - das definiert auch die Semantik (durch Transformation)
  - Compiler machen das tatsächlich (loop unrolling)

  Vergleiche: $(B_1 A)^* B_0 = B_0 \cup B_1(A \cdot (B_1 A)^* B_0)$

vorzeitiges Verlassen

- ...des Schleifenkörpers

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

- ...der Schleife

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

- operationale Semantik: äquivalentes Goto-Programm

```
start: if (B1) goto next else goto end;
next : A1; if (B2) goto ... ; A2; goto star
end  : skip;
```

- äquivalentes Programm mit Standard-Schleife?
  - für `continue`: einfach
  - für `break`: nur mit Boolescher Hilfsvariablen

Geschachtelte Schleifen

- manche Sprachen gestatten Markierungen (Labels) an Schleifen als Ziele für `break`, `continue`:

```
foo : for (int i = ...) {
  bar : for (int j = ...) {
    ... ; if (...) break foo; ...  } }
```

- deswegen (und nur deswegen) gibt es Marken (Labels) in Java, diese sind syntaktisch vor *jeder* Anweisung erlaubt

  . . . und das ist ein gültiges Programm:

```
void m () {  https://haskell.org/
             return; }
```

  Ü: warum zwei Zeilen?

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)
- zur Geschichte der Verzweigung in Programmiersprachen (mit vielen Original-Dokumenten)

  Eric Fischer: *if-then-else had to be invented*, !!Con West 2019 `https://github.com/ericfischer/if-then-else/blob/master/if-then-else.md`

  `http://bangbangcon.com/west/2019/speakers/`

Sprünge und Schleifen

- Goto und While: softwaretechnisch (pragmatisch) sehr unterschiedlich, aber semantisch gleich ausdrucksstark
- Satz: zu jedem While-P. gibt es ein äquivalentes Goto-P.
- Satz: zu jedem Goto-P. gibt es ein äquivalentes While-P.

  Beweis durch Kompilation:

  übesetze `1: A1; 2: A2; .. 5: goto 7;` zu

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

- das beweist: . . . äquivalentes While-P. mit ≤ 1 Schleife
- softwaretechnisch nützt das gar nichts

Schleifen und Unterprogramme

- Zu jedem While-P. gibt es ein äquivalentes P. ohne Schleifen: nur mit Verzweigungen und (rekursiven) Unterprogrammen
- Beweis-Idee: `while (B) A;` wird übersetzt in

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

- Anwendung: C-Programme ohne Schlüsselwörter.

  (Wiederholung: wie entfernt man `if`?)
- Anwendung: International Obfuscated C Code Contest `https://www.ioccc.org/`

Garantierte Initialisierung in Java

- JSL Kap. 16: For every *access* of a local variable x ..., x must be *definitely assigned* before the access, or a compile-time error occurs.

  For every *assignment* to a blank *final* variable, the variable must be *definitely unassigned* before the assignment, or a compile-time error occurs.

- Beispiel: A Java compiler recognizes that k is definitely assigned before its access (as an argument of a method invocation) in the code:

```
int k; // deklariert ohne Initialisierung
if (v > 0 && (k = System.in.read()) >= 0)
    System.out.println(k);
```

Analyse des Programmablaufes lt. JLS

- die genannten Bedingungen werden statisch geprüft:

  ...takes into account the structure of statements and expressions; it also provides a special treatment of the expression operators `&&`, `||`, `!`, and `? :`, and of boolean-valued constant expressions.

  Except for the special treatment of the conditional boolean operators `&&`, `||`, `!`, and `? :` and of boolean-valued constant expressions, the values of expressions are not taken into account ...

```
|  Error:
|  variable x might not have been initialized
|  {final int x;if(false)x=8;System.out.println(x);
|                                                 ^
|
```

Aufgaben zur Programm-Äquivalenz

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

- Syntax: Programme
  - Aktionen,
  - Zustandsprädikate (in Tests)
  - Sequenz/Block, Verzweigung (if)
  - Sprünge: Label, goto,
  - Schleifen: while, break, continue
  - Boolesche Variablen und Operatoren

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

Approximierte Spur-Semantik v. Programmen

- 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 *effektiv regulär*.

  Beweis: Konstruktion über endlichen Automaten.
  - Zustandsmenge $=$ Prädikatbelegungen $\times$ Anweisungs-Nummer
  - Transitionen? (Beispiele)

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

Hausaufgaben

WS 24: 1, 3, 5

1. Syntax If-Then-Else

(a) (Wdhlg) Ergänzen: das Problem des *dangling else* ist die Mehrdeutigkeit der Grammatik mit den Regeln . . .

(b) (Wdhlg) Geben Sie ein Bespielprogramm P mit 2 Ableitungsbäumen bzgl. einer solchen Grammatik an.

(c) Suchen Sie die entsprechenden Regeln der Java-Grammatik,

(d) geben Sie den Ableitungsbaum für voriges P bzgl. dieser Grammatik an, begründen Sie, daß diese Grammatik eindeutig ist.

(e) Suchen Sie die entsprechenden Regeln in der Grammatik der Programmiersprache Ada,

(f) wie muß P geändert werden, damit es durch diese Grammatik erzeugt werden kann?

2. Kompilation für Mehrfachverzweigung

(a) Schreiben Sie ein Programm, das einen C-Programmtext dieser Form ausgibt

```
void p(int x) {
  switch (x) {
    case   0 : q0(); break;
    case   1 : q1(); break;
    ...
    case 999 : q999(); break;     } }
```

Unterprogramme q_i nicht definieren, es geht nur um Kompilation (zu Objektfile, ohne Linking)

(b) Betrachten Sie den Assemblercode, der dafür von

`gcc -O2 -S` erzeugt wird.

(c) Ändern Sie das Programm zu

```
case     0 : ...
case   100 :
...
case 99900 :
```

beobachten und erklären Sie (ggf. weiter Abstände ausprobieren)

3. pattern matching (bzw. Pläne dafür) in JS, TS, C# zeigen (mit Primärquellen)

4. ein (kurzes) (Gewinner-)Programm eines IOCCC vorführen und erläutern, bei dem Programmablaufsteuerung nicht offensichtlich ist

5. mit JLS 14.21, 14.22 erklären:

```
switch(0) { case 0 -> 0; default -> { int y  = 0; y
```

weitere Beispiele zeigen für dort genannte Bedingungen

Semantik (Teil 2)

Dynamische Semantik

- Methoden zur Beschreibung der Semantik:
  - *operational*: beschreibt Wirkung einzelner Anweisungen durch Änderung des Speicherbelegung
  - *denotational*: ordnet jedem (Teil-)Programm einen Wert zu, Bsp: eine Funktion (höherer Ordnung). Beweis von Programmeigenschaften durch Term-Umformungen
  - *axiomatisch* (Bsp: Hoare-Kalkül): Schlußregeln zum Beweis von Aussagen über Programme
- Anwendung: die dynamische S. von *zusammengesetzen* Anweisungen beschreibt Programm-Ablauf-Steuerung

Anweisungen: Definition

- abstrakte *Syntax*:
  - einfache Anweisung:
    - leere Anweisung (`skip`), Zuweisung (`l:=r`),
    - Sprung `goto`, `break`, `continue`, `return`, `throw`
    - Unterprogramm-Aufruf (`p(a,b)`)
  - zusammengesetzte Anweisung:
    - Nacheinanderausführung (Block)
    - Verzweigung (zweifach: `if`, mehrfach: `switch`)
    - Wiederholung (Schleife)
- *Semantik*: Ausführen einer Anweisung
  bewirkt Zustandsänderung
  (evtl. mit mehreren Zwischenzuständen)
  Zustand: Speicherbelegung und Außenwelt (über OS)

Programm-Ablauf-Steuerung

- engl. *control flow*, falsche Übersetzung: Kontrollfluß;
  to control = steuern, *to check* = kontrollieren/prüfen
- von-Neumann-Modell: jede Anweisung beschreibt
  - Was? (Operation) Womit? (Operanden) Wohin? (Resultat)
  - Wie weiter? (nächste Anweisung)

  Programm-Ablauf dabei also durch Sprünge gesteuert
- Ablaufsteuerung durch strukturierte Programmierung:
  jedes Teilprogramm (Teilbaum des AST)
  hat genau einen Eingang und genau einen Ausgang
- vorzeitiges Verlassen eines Teilprogrogramms:
  Schleife (break, continue), UP (return), throw/catch

Operationale Semantik: Sprünge

- Maschinenmodell:

  Variable PC (program counter) enthält Adresse des nächsten auszuführenden Befehls

- Semantik von Goto(z) ist: PC $:= z$

  Semantik der Nicht-Sprungbefehle: $\ldots$, PC $:=$ PC $+ 1$

- andere Varianten der Programmablaufsteuerung können in Goto-Programme übersetzt werden

  Bsp: Schleife `while (B) A` $\Rightarrow$ `if (B) ...`

  das findet bei Kompilation von Java nach JVM statt

Axiomatische Semantik

Notation für f. Aussagen über Speicherbelegungen:

Hoare-Tripel: `{ V } A { N }`

- für jede Belegung s, in der Vorbedingung V gilt:
- *wenn* Anweisung A ausgeführt wird

  und Belegung t erreicht wird,
- *dann* gilt dort Nachbedingung N

Beispiel: `{ x >= 5 } y := x + 3 { y >= 7 }`

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

Gültigkeit solcher Aussagen kann man

- (vor Programm-Ausführung) beweisen (mit Hoare-Kalkül)
- (während Programm-Ausführung) überprüfen (assert)

Eiffel

Bertrand Meyer, `https://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

Sprachstandard: *Eiffel: Analysis, design and programming language* ECMA-367 (2nd edition, June 2006)

Hoare-Kalkül: Überblick

zu jedem Knotentyp in abstrakten Syntaxbäumen von strukturierten imperativen Programmen ein Axiom-Schema

- elementare Anweisung:
  - leere Anweisung `{ N } skip { N }`
  - Zuweisung `{ N[x/E] } x := E { N }`
- zusammengesetzte Anweisungen:
  - ```
    wenn { V } C { Z }  und  { Z } D { N }
    dann  { V } C; D { N }
    ```
  - ```
    wenn { V und     B } C { N }
    und  { V und not B } D { N }
    dann { V } if (B) then C else D { N }
    ```
  - ```
    wenn  { I and B } A { I },
    dann  { I } while (B) do A { I and not B }
    ```
    ```

Axiom für Zuweisung

- Axiom (-Schema): `{ N[x/E] } x := E { N }`
- dabei bedeutet $N[x/E]$:

  der Ausdruck N, wobei jedes Vorkommen des Namens x durch den Ausdruck E ersetzt wird

  Bsp: $(y \geq 7)[y/x+3] \;=\; (x+3 \geq 7) \;=\; (x \geq 4)$
- Bsp: Anwendung `{ ... } y := x+3 { y >= 7 }`
- Übung: welche Vorbedingung ergibt sich für

  ```
  a := a + b; b := a - b ; a := a - b;
  ```

  und Nachbedingung $a = X \wedge b = Y$?

  Dabei auch Axiom für Nacheinanderausführung benutzen

Simultan-Zuweisung

- Anweisung $(v_1, v_2) := (e_1, e_2)$ für $v_1 \neq v_2$
- axiomatische Semantik:

  $\{N[v_1/e_1, v_2/e_2]\}\ (v_1, v_2) := (e_1, e_2)\ \{N\}$

  verwendet links simultane Ersetzung
- Bsp: $\{\dots\}\ (a, b) := (b, a)\ \{a = 2 \wedge b = 5\}$

  Bsp: $\{\dots\}\ (x, y) := (x + y, x - y)\ \{x = 7 \wedge y \geq 3\}$
- realisiert in der Sprache CPL 1963
- in JS als *destructuring assignment*, ECMA 262: 12.15.5

```
[a,b]=[8,9]; [a,b]=[b,a]
```

Logische Axiome

- logische Umformungen (Programm A bleibt erhalten)
  - Verschärfen einer Vorbedingung (von V zu V')
  - Abschwächen einer Nachbedingung (von N zu N')

```
wenn { V } A { N } und V' => V und N => N'
dann { V' } A { N' }
```

- Anwendung: beweise `{x < 1} x := 5-x { x > 2 }`
  - Zuweisungs-Axiom ergibt `{5-x>2} x:=5-x {x>2}`
  - äquivalent umgeformt zu `{x<3} x:=5-x {x>2}`
  - dann o.g. Axiom anwenden mit

    `V=(x<3), V'=(x<1), N=N'=(x>2)`

Axiom für Verzweigung

- das Axiom:

```
wenn { V und     B } C { N }
und  { V und not B } D { N }
dann { V } if (B) then C else D { N }
```

- Anwendung: beweisen Sie

```
{ x > 9 }
if (x > y) then a := x - 2 else a := y + 2
{ a > 7 }
```

Axiom für Verzweigung (Rechnung)

- wir müssen `{x>9 und x>y} a:=x-2 {a>7}`

  und `{x>9 und x<=y} a:=y+2 {a>7}` zeigen,

  um das Axiom-Schema anwenden zu können

- Zuweisungs-Axiom ergibt `{x-2>7} a:=x-2 {a>7}`

  äquivalent umgeformt `{x>9} a:=x-2 {a>7}`

  Axiom-Schema zum Verschärfen der Vorbedingung
  ($V' = (x > 9) \wedge (x > y), V = (x > 9)$) ergibt erstes Teilziel

- Zuweisungs-Axiom ergibt `{y+2>7} a:=y+2 {a>7}`

  äquivalent umgeformt `{y>5} a:=y+2 {a>7}`

  Axiom-Schema zu Verschärfen der Vorbedingung
  ist anwendbar für $V' = (x > 9 \wedge x \leq y), V = (y > 5)$ wegen $V' \Rightarrow V$,
  ergibt zweites Teilziel

Axiom für Schleifen

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

- ```
  Eingabe int p, q;   // p = P und q = Q
  int c = 0;
  // inv: p * q + c = P * Q
  while (q > 0) {
     ??? := ???; q := q - 1;
  }
  // c = P * Q
  ```

- Invariante muß: 1. vor der Schleife gelten, 2. im S.-Körper invariant bleiben, 3. nach der Schleife nützlich sein

- erst Spezifikation (hier: Invariante), dann Implementierung. (sonst: *cart before the horse*, EWD 1305)
  ```

Erweiterter Euklidische Algorithmus (Spezif.)

- Def: $\gcd(x, y)$ ist das Infimum (größte untere Schranke) von x und y in der Teilbarkeits-Halbordnung, d.h.,
  $\gcd(x, y)|x \wedge \gcd(x, y)|y \wedge \forall h : \ldots$
- Erweiterter Euklidischer Algorithmus, Spezifikation:
  - Eingabe: $x, y \in \mathbb{N}$
  - Ausgabe: $a, b \in \mathbb{Z}$ mit $g = a \cdot x + b \cdot y \wedge g|x \wedge g|y$
- Ü: diese Spez. erfüllen für Eingabe $x = 60, y = 35$
- Satz: $g = \gcd(x, y)$.
- Beweis des Satzes:
  1. $g|x \wedge g|y$ nach Spezifikation,
  2. $\ldots$

Erweiterter Euklid — imperative Impl.

- Ansatz: verwende $x, y, a, b, p, q \geq 0$ mit Invariante

  $$\gcd(x, y) = \gcd(x_{\text{in}}, y_{\text{in}}) \wedge x = a \cdot x_{\text{in}} + b \cdot y_{\text{in}}, y = p \cdot x_{\text{in}} + q \cdot y_{\text{in}}$$

- ```
  // X = x, Y = y, x >= 0, y >= 0
  (a,b,p,q) := ...
  // Inv:  gcd (x,y) = gcd (X,Y) ,
  //       x = a X + b Y , y = p X + q Y
  while ( y > 0 ) {
     (x,y,a,b,p,q) := (y, x mod y, ... )
  }
  // gcd(X,Y) = a X + b Y
  ```
  ```

Partielle und totale Korrektheit

- Hoare-Tripel `{V} A {N}` beschreibt *partielle* Korrektheit:
  wenn vorher `V` gilt *und* `A` hält, *dann* gilt danach `N`
  Bsp: `{true} while (true); {x=42}` ist wahr
- stärker (und nützlicher) ist *totale* Korrektheit:
  partielle Korrektheit *und* `A` hält tatsächlich (*Termination*)
- Beweis-Verfahren (für Schleifen `while (B) do A`)
  - partielle Korrektheit: Invariante
  - Termination: Maßfunktion (Schrittfunktion)
    m: Speicherbelegung $\rightarrow \mathbb{N}$ mit `{m = M} A { m < M}`
  Bsp: eine Maßf. für `while(x>0){ ...; x=x/2;}` ist `x`
- es gilt: $m($aktueller Zustand$) \geq$ verbleibende Anzahl der Schleifendurchläufe (Schritte)

Beispiele für Maßfunktionen

- `while (x > 0) { x = x - 1; } // MF: x`

- ```
  z = 1; while (x > 0) {
    y = x; x--; while (y > 0) { y--; z++; }
  }
  ```

  MF$(x, y) = 2x^2 + y$, denn für $x \geq 1$:

  MF$(x, y) \geq 2x^2 + 0 >$ MF$(x - 1, x) = 2(x - 1)^2 + x = 2x^2 - 3x + 2$

- ```
  z = 1; while (x > 0) {
    y = z; x--; while (y > 0) { y--; z++; }
  } // MF: ?
  ```

Wie findet man die Maßfunktion?

- die richtige Antwort ist (wie für die Invariante, siehe EWD): das ist die falsche Frage.
- das softwaretechnisch richtige Vorgehen ist:
  1. (Entwurf) Invariante und Maßfunktion hinschreiben,
  2. (Implementierung) Schleife so ausprogrammieren, daß diese Behauptungen stimmen.
- in der Praxis wünscht man eine Teil-Automatisierung:

  maschinelles Finden von einfachen („offensichtlichen“) Maßfunktionen und Beweisen dafür
- vgl. Typisierung von Namen:
  - Deklaration (durch Programmierer),
  - Inferenz (durch Compiler) (z.B. `var` in C#, `auto` in C++)

Automatische Laufzeitanalyse

- Martin Hofman, Jan Hoffman, et al.: *Resource Aware ML*, 2010–, `https://www.raml.co/publications/`

  Namen sind statisch typisiert, Typ enthält Komplexität

  Typ wird inferiert (Koeffizienten des Laufzeitpolynoms durch Constraint-Solver bestimmt)
- Intl. Workshop on Termination (seit 1993),
  Intl. Termination and Complexity Competition (seit 2003),

  `https://www.termination-portal.org/`
- Geser, Hofbauer, Waldmann: SRS Termin. Analysis

  `https://gitlab.imn.htwk-leipzig.de/waldmann/pure-matchbox#srs-nontermination-analysis`
- automatische Analyse ist nützlich, denn . . .

  `https://accidentallyquadratic.tumblr.com/`

Hausaufgaben

Für alle Programme: Diskussion der Eigenschaften (Hoare-Tripel, Invarianten) im Pseudocode. Geben Sie zusätzlich eine Implementierung in einer Programmiersprache Ihrer Wahl an, die dem Pseudocode optisch nahe kommt. Für Java: verwenden Sie `assert` (JLS 14.10)

WS 24: 2, 4, 5

1. zur Folie „Zuweisungs-Axiom“:

   - bestimmen Sie die Vorbedingung zu `a := a + b; ...`, aus den Axiomen für Zuweisung und Nacheinanderausführung.
   - Geben Sie ein ähnliches Verfahren an, das mit

`a := a XOR b` beginnt, wobei `XOR` die bitweise Antivalenz bezeichnet.

- Für das C++-Programm

```
#include <iostream>

void sub (int & a, int & b) {
  a = a + b; b = a - b; a = a - b;
}

int main () {
  int p = 3; int q = 4;
  sub (p, q);  // (*)
  using namespace std;
  cout << p << q << endl;
```

```
}
```

Kompilieren Sie mit `-O3`,
betrachten Sie den erzeugten Assemblercode:
für `sub`: wieviele Register werden benutzt?
für `main`: vergleichen Sie mit der Variante, bei welcher der markierte Unterprogrammaufruf auskommentiert wird (beide Varianten abspeichern, z.B. `g++ -O3 -S -o prog.s prog.cc`, `diff` benutzen)

2. `https://www.imn.htwk-leipzig.de/~waldmann/edu/ws21/inf/folien/#(114)` (Parteien A, B, C).

   Vgl. auch die Folien davor (20 und 21 Kugeln)

   `https://www.imn.htwk-leipzig.de/~waldmann/edu/ws21/inf/folien/#(9)` (91 Atome)

3. Ergänzen Sie das folgende Programm, so daß die Spezifikation (das Potenzieren) erfüllt wird:

```
Eingabe: natürliche Zahlen a, b;
// a = A und b = B
int p := 1; int c := ???;
// Invariante:  c^b * p = A^B
while (b /= 0) {
    if (b ist ungerade)
      then (c,p) := ...
      else (c,p) := ...
    //  Z
    b := abrunden (b/2);
}
Ausgabe: p; // p = A^B
```

- Initialisieren Sie $\mathtt{c}$ so, daß die Invariante gilt.
- Wieso folgt aus der Invariante bei Verlassen der Schleife die Korrektheit der Ausgabe?
- Bestimmen Sie eine geeignete Aussage $\mathtt{Z}$ als Vorbedingung der nachfolgenden Anweisung bezüglich der Invariante.
- Bestimmen Sie daraus die Lücken $(\ldots)$

4. Für das Programm

```
Eingabe: positive natürliche Zahlen A, B;
(a,b,c,d) := (A,B,B,A)
while (a /= b) {
  if (a > b) then (a,d) := (a-b,c+d)
             else (b,c) := (b-a,d+c)   }
Ausgabe: (a+b)/2 , (c+d)/2
```

- zeigen Sie, daß die erste Ausgabe gleich `gcd(A,B)` ist. Zeigen Sie dazu: `gcd(a,b) = gcd(A,B)` ist invariant. Welche Eigenschaften des `gcd` werden benötigt?
- was ist die zweite Ausgabe? Geben Sie eine Vermutung an und beweisen Sie mit einer geeigneten Invariante.
- wozu ist die Bedingung „positiv“ notwendig?

5. Es können zusätzlich Aufgaben aus dem Math+-Adventskalender `https://www.mathekalender.de/wp/de/kalender/` bearbeitet werden—wenn eine Beziehung zur Vorlesung hergestellt wird, z.B. Verwendung einer Methode aus dem Skript oder einer esoterischen Programmiersprache.

   Zum Lesen der Aufgaben ist keine Registrierung erforderlich.

   In unserem Issue-Tracker diskutieren, pro Woche max. eine Aufgabe zur Präsentation in der Übung, sofern Zeit ist. Aufgaben müssen nicht vollständig gelöst werden.