

Prinzipien von Programmiersprachen

Vorlesung

Wintersemester 2007 – 2026

Johannes Waldmann, HTWK Leipzig

18. Oktober 2026

Einleitung

Programme und Algorithmen

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

Deutsch als Programmiersprache

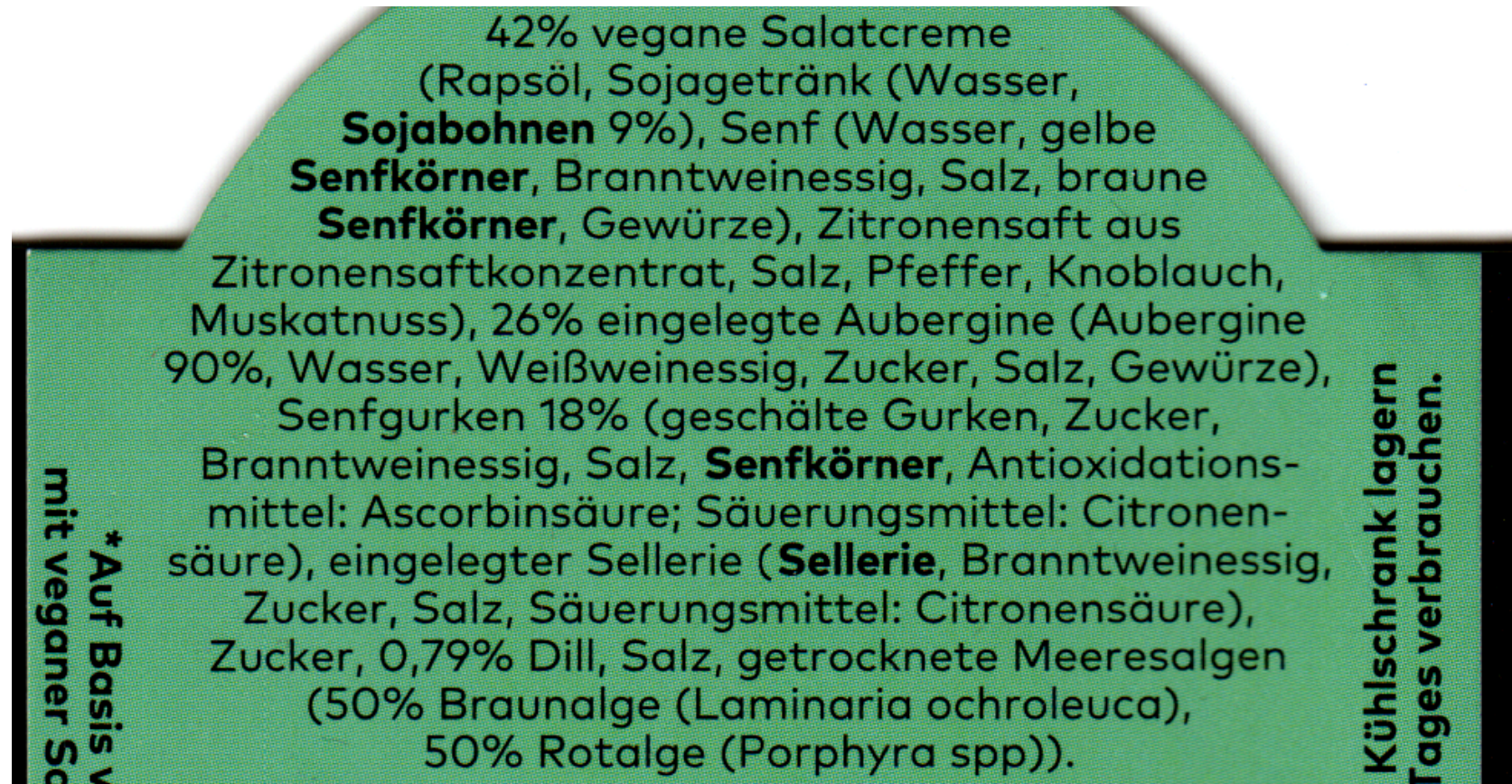
§6 (2) ... Der Zuteilungsdivisor ist so zu bestimmen, dass insgesamt so viele Sitze auf die Landeslisten entfallen, wie Sitze zu vergeben sind. Dazu wird zunächst die Gesamtzahl der Zweitstimmen aller zu berücksichtigenden Landeslisten durch die Zahl der jeweils nach Absatz 1 Satz 3 verbleibenden Sitze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

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

https://www.gesetze-im-internet.de/bwahlg/___6.html

Struktur durch Klammern, ist doch klar

- Wo ist die öffnende Klammer für die Muskatnuß?



(HMF Food Production, Dortmund, 2022)

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

Beispiel: mehrsprachige Projekte

ein typisches Projekt besteht aus:

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

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

In / Into

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

Sprache

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

Wie unterschiedlich sind Sprachen?

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

Konzepte

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

Paradigmen

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

Ziele der LV

Arbeitsweise: Methoden, Konzepte, Paradigmen

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

Ziel:

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

Beziehungen zu anderen LV

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

Anwendungsspezifische Sprachen und Paradigmen:

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

Organisation

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

Literatur

- Skript, Aufgaben usw. erreichbar von

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

Zum Vergleich/als Hintergrund:

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

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

- Robert W. Sebesta: Concepts of Programming Languages, Addison-Wesley 2004, ...

- Turbak, Gifford: Design Concepts of Programming Languages, MIT Press 2008

<https://cs.wellesley.edu/~fturbak/>

Inhalt

(nach Sebesta: Concepts of Programming Languages)

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

Haus-Aufgaben

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

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

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

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

und beantworten Sie

- womit wird “einfaches Programmieren” fälschlicherweise gleichgesetzt?

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

Geben Sie dazu jeweils an:

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

Übersetzung

- Beispiele aus Ihrer Erfahrung

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

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

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

Lösungen:

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

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

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

Shell-Kommandos) den Quelltext dieser Manpage, zeigen diesen an. (Wie benutzt man `man`? so: `man man`.)

(2024 – was war hier los? `https:`

`//git.savannah.gnu.org/cgiit/man-db.git/commit/?id=b225d9e76fbb0a6a4539c0992fba88c83f0bd37e`

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

4. funktionales Programmieren in Haskell

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

```
ghci
```

```
:set +t
```

```
length $ takeWhile (== '0') $ reverse $ sho
```

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

`https://hoogle.haskell.org/` suchen.

(Einschränken auf `package:base`) Die anderen (drei) Funktionen dieses Typs aufrufen.

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

5. PostScript

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

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

Mit Editor Quelltext ändern, Wirkung betrachten.

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

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

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

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

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

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

8. erklären Sie <https://xkcd.com/378/>.

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

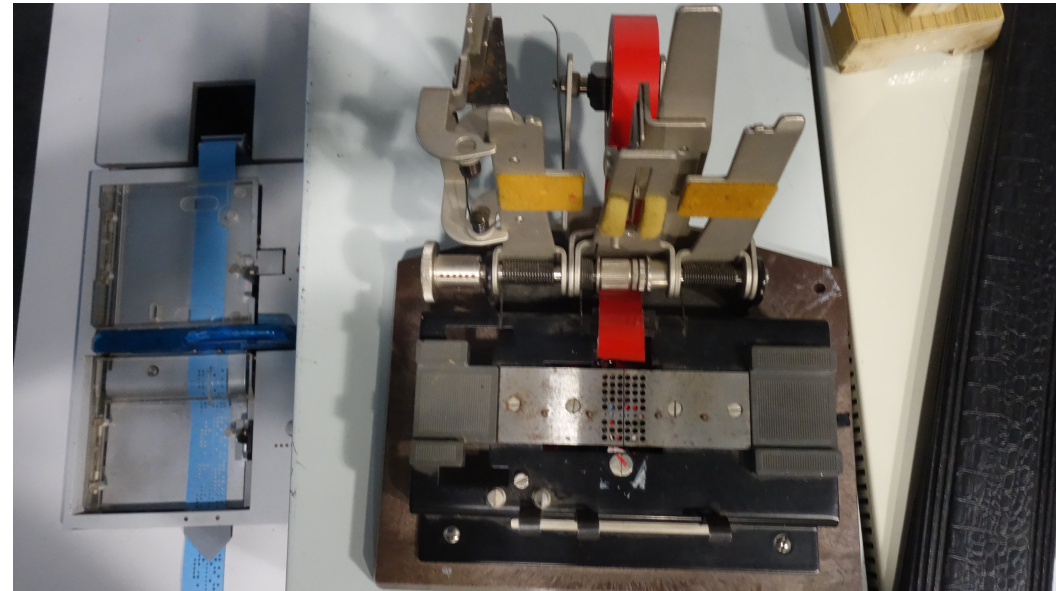
9. durch (möglichst kurzes) Programm in einer Programmiersprache Ihrer Wahl: bestimmen Sie den längsten Satz aus Thomas Mann: Zauberberg (<https://www.gutenberg.org/ebooks/65661>). Drücken Sie dessen Struktur durch Klammern oder Einrückungen aus (wie bei einem Programmtext).

Syntax von Programmiersprachen

Konkrete und abstrakte Syntax

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

Fernschreiber, Editor...



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

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

Terminal (Konsole)

- Schreibmaschine für Ein- und Ausgabe (*Bediendrucker*)
ist Vorbild für elektronische Text-Ein- und -Ausgabegeräte



```
File Edit View Search Terminal Help
$ uname -a
Linux 16tons 6.17.1-300.fc43.x86_64
TC 2025 x86_64 GNU/Linux
$
```

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

(rechts) Gnome Terminal 3.56.3 for Gnome 48,

Ganz ganz ganz kurze Geschichte der (Programmiersprachen und) Betriebssysteme

- Betriebssystem = Schnittstelle zwischen Anwenderprogramm und Hardware (Zugriff auf Prozessor, Hauptspeicher, Massenspeicher, E/A-Geräte; Dienstprogramme: Kommandozeilen-Interpreter, Editor)
- Geld verdienen mit Hardware, Software (OS, Compiler) war Zugabe, lief aber nur dort. (FORTRAN, IBM 1958)
- OS/360 (IBM, 1964) Brooks: *The Mythical Man-Month*, 75
- UNIX (Thomson, Ritchie, Bell Labs 1969) portabel (C)
- GNU (GNU's not Unix) (Stallmann 1984) freie Software Dienstprogramme (bash, emacs), separat: Kernel (Hurd)
- Linux (Torvalds 1991) Kernel für i386
- Debian (Murdock 1993) ist GNU/Linux-Distribution

Programme als Bäume

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

Token-Klassen

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

alle Token einer Klasse bilden eine *formale Sprache*.

Formale Sprachen

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

Beispiele:

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

Lexik (Bsp): numerische Literale

- Ada (2012) http://www.ada-auth.org/standards/rm12_w_tcl/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),
$$\left\{ w \mid \forall p, r : \left(\begin{array}{l} (p < r \wedge w[p] = a \wedge w[r] = c) \\ \Rightarrow \exists q : (p < q \wedge q < r \wedge w[q] = b) \end{array} \right) \right\}$$

Sprach-Operationen

Aus Sprachen L_1, L_2 konstruiere:

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

Def: Sprache *regulär* : $\iff$ kann durch diese Operationen aus endlichen Sprachen konstruiert werden.

Satz: Durchschnitt und Differenz braucht man dabei nicht.

Reguläre Sprachen/Ausdrücke

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

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

Jeder solche Ausdruck beschreibt eine *reguläre Sprache*.

Beispiele/Aufgaben zu regulären Ausdrücken

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

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

Erweiterte reguläre Ausdrücke

1. zusätzliche Operatoren (Durchschnitt, Differenz, Potenz), die trotzdem nur reguläre Sprachen erzeugen

Beispiel: $\Sigma^* \setminus (\Sigma^* ab \Sigma^*)^2$

ähnlich in Konfiguration der autotool-Aufgaben

2. zusätzliche nicht-reguläre Operatoren

Beispiel: exakte Wiederholungen $L^{\boxed{k}} := \{w^k \mid w \in L\}$

Bsp.: $(ab^*)^{\boxed{2}} = \{aa, abab, abbabb, ab^3ab^3, \dots\} \notin \text{REG}$

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 $w \in L(X)$?

(Wort w gehört zur Sprache eines regulären Ausdrucks X)

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

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

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

Übungen 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
- $(\text{Pow}(\Sigma^*), \cup, \cdot, \dots, \dots)$ ist Halbring (ergänzen Sie die neutralen Elemente)
- In jedem Monoid: Damit $a^{b+c} = a^b \cdot a^c$ immer gilt, muß

man a^0 wie definieren?

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

Hausaufgaben

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

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

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

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

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

Ist sie antisymmetrisch? (Beweis oder Gegenbeispiel.)

NB: Beziehung zur Softwaretechnik:

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

2. Zeichnen Sie jeweils das Hasse-Diagramm dieser Teilbarkeitsrelation

- für $(\mathbb{N}, +, 0)$, eingeschränkt auf $\{0, 1, \dots, 4\}$
- für $(\mathbb{N}, \cdot, 1)$, eingeschränkt auf $\{0, 1, \dots, 10\}$
- 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 installiert, siehe <https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/>

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

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

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

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

Untersuchen Sie ähnliches für Compiler für andere Sprachen.

7. Suchen und diskutieren Sie „Wadler’s law (of language design)“.

Am Entwurf welcher Programmiersprachen war der Autor

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

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

Belegen Sie durch

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

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

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

`DEC_LITERAL` $\rightarrow$ `DEC_DIGIT` (`DEC_DIGIT` | `_`) *

`DEC_DIGIT` $\rightarrow$ `[0-9]`

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

`DecimalDigits` ::

`DecimalDigit`

`DecimalDigits` `DecimalDigit`

`DecimalDigits` `NumericLiteralSeparator` `DecimalDigit`

`DecimalDigit` :: one of 0 1 2 3 4 5 6

`NumericLiteralSeparator` :: `_`

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

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