# Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen

Johannes Waldmann, HTWK Leipzig

19. Juni 2017

**Zusammenfassung**

Das Programmieren ist das Realisieren eines Algorithmus als Text in einer konkreten Programmiersprache. Ich zeige, wie das Verständnis der Algorithmen an sich gefördert und überprüft werden kann, ohne daß Studenten durch Eigenheiten konkreter Programmiersprachen abgelenkt werden. Ich verwende stattdessen stark eingeschränkte Sprachen mit trivialer Syntax und aufgabenspezifischer Semantik. Aufgaben-Erzeugung sowie -Bewertung werden im E-Learning/E-Assessment-System `autotool` technisch realisiert, das seit ca. 2000 an der Universität Leipzig, der HTWK Leipzig, sowie gelegentlich an weiteren Hochschulen eingesetzt wird.

## 1 Motivation

Ein klassischer Bestandteil der Lehre von Algorithmen und Datenstrukturen [THC09] ist das Thema „balancierte Suchbäume“, mit dem Beispiel AVL-Bäume. Der Inhalt sind Sätze über Algorithmen: Balance erzwingt logarithmische Höhe, Rotation erhält die Suchbaum-Eigenschaft und repariert die Balance lokal; sowie die zugrundeliegenden Beweismethoden (Induktion über die Höhe) sowie Programmierkonzepte (Rekursion, Iteration). Das Verständnis dieser Ideen kann dadurch unterstützt werden, daß Studenten die Algorithmen ausführen (auf dem Papier, an der Tafel) und implementieren (als Programmtext).

Ich beschreibe hier eine weitere Möglichkeit von unterstützenden Aufgaben sowie deren Automatisierung innerhalb des Systems `autotool`[RW02, Wal17]. Studenten sollen eine Folge $s$ von Einfüge- und Lösch-Operationen finden, die aus einem vorgegebenen AVL-Baum $t_1$ einen vorgegebenen Baum $t_2$ erzeugt. Für die Folge $s$ sind die Länge sowie einige Elemente vorgegeben und die restlichen zu finden. Die Vorgabe ist ein Folge mit Lücken $s_?$. Die Aufgabenstellung besteht dann aus $(t_1, s_?, t_2)$.

Studenten tragen einen Lösungsversuch (eine Folge $s$) in einem Web-Formular ein. Nach Absenden führt das `autotool`-System die Operationsfolge $s$ aus und gibt die Folge der dabei erzeugten Bäume aus. Schließlich wird das Endresultat mit der Vorgabe verglichen. Eine Lösung ist jede Folge $s$, die zu $s_?$ paßt und $t_1$ in $t_2$ überführt. Korrektheit ist also nicht definiert als Übereinstimmung mit einer Musterlösung, sondern als das Erfüllen der Spezifikation. Das hat auch den Vorteil, daß die Ablehnung einer falschen Lösung nachvollziehbar begründet werden kann, aber in der Begründung keine richtige Lösung verraten wird. Deswegen ist es unschädlich, und im Gegenteil sogar erwünscht, daß Studenten mehrere Lösungsversuche ausführen.

Der abschließende Test auf Korrektheit ist gar nicht der wesentliche Nutzen dieser Aufgabe. Vielmehr geht es um die ausführliche Beschreibung der Wirkung der einzelnen Operationen in der Ausgabe. Dabei können die Studenten sehen, wie die korrekte Implementierung

einzelner Operationen wirkt. Die Studenten können dabei auch selbständig und frei experimentieren und z.B. beobachten, was passiert, wenn im nächsten Schritt die Wurzel gelöscht wird u.ä., indem sie eine geeignete Folge $s$ eingeben und absenden. Diese löst dann zwar die vorgegebene Aufgabe nicht, aber die Anzahl der Fehlversuche spielt bei der Bewertung keine Rolle.

Für Aufgaben des beschriebenen Typs könne automatisch Instanzen erzeugt werden: das System würfelt einen Baum $t_1$ sowie eine Operationsfolge $s_0$ und bestimmt daraus eine Folge $s_?$ durch Ersetzen von Elemente an zufällig gewählten Positionen durch Markierungen `?`. Der Aufgabensteller konfigurierte den Würfel durch Vorgabe der Größe der Größe von $t_1$, der Länge von $s_?$ sowie der Anzahl der Fragezeichen in $s_?$.

Durch den Würfel erhält jeder Student eine andere Aufgaben-Instanz, damit sind Plagiate von Lösungen unmöglich.

Alle hier beschriebenen Aufgaben können unter dieser Adresse ausprobiert werden: `https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=240`.

## 2 Aufgaben zu Suchbäumen

Suchbäume $S(K)$ mit Schlüsseln einer Menge $K$ realisieren den abstrakten Datentyp Menge mit folgender Schnittstelle. Die Semantik $[\![t]\!]$ eines Baumes ist die Menge seiner Schlüssel.

- $\mathsf{empty} : S(K)$ mit $[\![\mathsf{empty}]\!] = \emptyset$, $\mathsf{null} : S(K) \to \mathbb{B}$ mit $[\![\mathsf{null}(t)]\!] = (\emptyset = [\![t]\!])$
- $\mathsf{contains} : K \times S(K) \to \mathbb{B}$ mit $[\![\mathsf{contains}(x,t)]\!] = (x \in [\![t]\!])$
- $\mathsf{insert} : K \times S(K) \to S(K)$ mit $[\![\mathsf{insert}(x,t)]\!] = \{x\} \cup [\![t]\!]$
- $\mathsf{delete} : K \times S(K) \to S(K)$ mit $[\![\mathsf{delete}(x,t)]\!] = [\![t]\!] \setminus \{x\}$

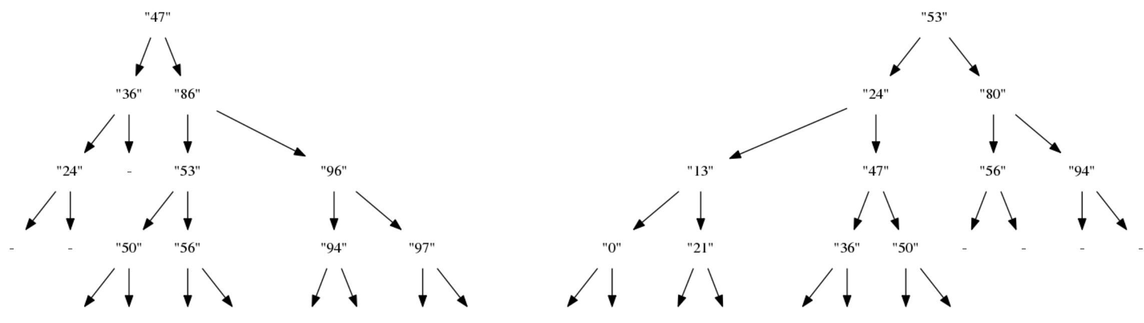
Für die Aufgaben verwenden wir Zahlen als Schlüssel. Jeder implementierende Typ $S(K)$ besitzt eine Funktion zur grafischen Darstellung. Wir verwenden `dot` [G+17].

Derzeit existieren diese Implementierungen

- nicht balancierte binäre Bäume, AVL-Bäume, Rot-Schwarz-Bäume, B-Bäume.

Ein Beispiel für eine Aufgabenstellung für AVL-Bäume ist:

```
Auf den Baum (linkes Bild)
sollen diese Operationen angewendet werden
(wobei Sie  Any  geeignet ersetzen sollen):
   [ Any , Any , Any , Delete 86 , Any, Any , Insert 13 , Insert 21 ]
so daß dieser Baum (rechtes Bild) entsteht:
```



Eine korrekte Lösung ist

```
[ Insert 53, Insert 80, Delete 97, Delete 86, Delete 96, Insert 0, Insert 13, Insert 21 ]
```

Für die Erzeugung zufälliger Aufgabeninstanzen werden, wie in der Einleitung beschrieben, $t_1$ und $s_0$ ausgewürfelt und daraus $t_2$ bestimmt. Damit interessante, d.h., nicht trivial lösbare, Instanzen entstehen, werden jeweils mehrere Kandidaten-Instanzen bestimmt und dann diejenige verwendet, die den Hamming- oder Editier-Abstand von $t_1$ zu $t_2$ maximiert. Die Wurzeln von $t_1$ und $t_2$ sollen sich unterscheiden und die Wurzeln der nächsten Teilbäume möglichst auch, usw., damit die Aufgabe nicht durch Betrachtung unabhängiger Teilbäume einfach lösbar ist. Unterschiedliche Wurzeln erreicht man bei nicht balancierten Bäumen durch delete-Operationen, bei anderen auch als Folge von Balance-Reparaturen.

# 3 Aufgaben zu heap-geordneten Bäumen

Heaps $H(K)$ mit Schlüsseln (Prioritäten) einer total geordneten Menge $K$ realisieren den abstrakten Datentyp Prioritätswarteschlange mit folgenden Operationen. Die Semantik $[\![h]\!]$ eines Heaps $h$ ist die (Multi)menge der Schlüssel. Die insert-Operation liefert einen Verweis $p \in \mathsf{Ref}(K)$, der bei decrease verwendet wird.

- $\mathsf{empty} : H(K)$, $\mathsf{null} : H(K) \to \mathbb{B}$,
- $\mathsf{extractMin} : H(K) \to K \times H(K)$ mit $[\![\mathsf{extractMin}(h)]\!] = (x, h')$ wobei $x = \min[\![h]\!], [\![h']\!] = [\![h]\!] \setminus \{x\}$
- $\mathsf{insert} : H(K) \to \mathsf{Ref}(K) \times H(K)$ mit $[\![\mathsf{insert}(x, h)]\!] = (p, h')$ wobei $[\![h']\!] = [\![h]\!] \cup \{x\}$ und $p$ ist ein Verweis in $h'$, so daß $h'[p] = x$.
- $\mathsf{decrease} : \mathsf{Ref}(K) \times K \times H(K) \to H(K)$ mit $[\![\mathsf{decrease}(p, y, h)]\!] = h'$ wobei $h' = h[p := y]$

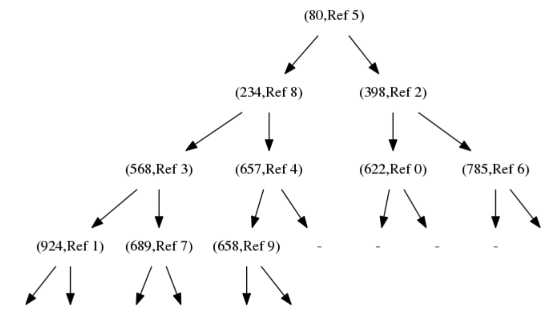
Implementierung besitzen zusätzlich eine Funktion zur grafischen Darstellung. Wir verwenden wieder Zahlen Schlüssel (Prioritäten). Die Verweise sind ebenfalls Zahlen. Die Ziele der Verweise erscheinen in der grafischen Darstellung. Für jeden Knoten werden der Schlüssel sowie der Name des Verweises auf diesen Knoten gezeichnet.

Derzeit wird die Schnittstelle implementiert durch

- fast vollständig balancierte binäre Heaps.

Dabei sind alle Blätter in Tiefe $h$ oder, rechts davon, in Tiefe $h - 1$. Solche Heaps könne implizit durch Arrays repräsentiert werden, was in dieser Aufgabe jedoch keine Rolle spielt.

Eine Beispiel-Aufgabenstellung enthält den Baum



Grundsätzlich treffen Bemerkungen zu Suchbäumen zu. Es zeigte sich, daß Aufgaben zu Heaps schwieriger waren, erkennbar an einer höheren Anzahl von Fehlversuchen. Ich nehme an, daß die decrease-Operation die Ursache ist. Dadurch wird eine direkte Zuordnung zwischen den Schlüsseln in der Eingabe $t_1$ und denen in der Ausgabe $t_2$ erschwert.

## 4 Aufgaben zu Hashtabellen

Eine Hashtabelle $T(K)$ repräsentiert den abstrakten Datentyp Menge, die Implementierung verwendet direkten Zugriff mittels Hashfunktionen.

Gegenüber den vorher beschriebenen Aufgaben zu Bäumen enthalten Aufgaben zu Hashing nicht nur eine Beschreibung der vorgegebenen Start- und Endzustände, hier: Belegung der Hashtabelle, sondern zusätzlich eine Spezifikation des verwendeten Hash-Verfahrens sowie seiner Parameter, insbesondere seiner Hashfunktion(en).

Form und Koeffizienten von Hashfunktionen können teilweise versteckt werden. Damit wird die Angabe der Hashfunktion zu einem Teil der Aufgabe.

Derzeit gibt es Aufgaben zu diesen Hashverfahren:

- Hashing mit Verkettung: jede Zelle der Hashtabelle enthält eine Liste von Elementen.
- offenes Hashing mit linearem Sondieren: jede Zelle der Hashtabelle enthält kein oder ein Element, die Sondierungsreihenfolge für $x$ mit $\text{hash}(x) = p$ ist $p, p+1, p+2, \ldots$ (modulo Tabellengröße).
- doppeltes Hashing: jede Zelle der Hashtabelle enthält kein oder ein Element, die Sondierungsreihenfolge für $x$ mit $\text{hash}_1(x) = p$ und $\text{hash}_2(x) = q$ ist $p, p+q, p+2q, \ldots$.

Die Hashfunktion kann vorgegeben werden als lineare Funktion mit unbekannten Koeffizienten $h(x) = ?+?\cdot x$. Bei Hashing mit Verkettung sind die Koeffizienten leicht zu rekonstruieren, da man Paare $(x, h(x))$ direkt aus der Tabelle ablesen kann. Bei offenem Hashing ist das schwieriger, da nicht klar ist, nach wieviel Sondierungs-Schritten ein Element eingefügt wurde. Das ist gut so, denn dann wird genau das beim Bearbeiten der Aufgabe geübt.

Eine Beispiel-Aufgabenstellung für doppeltes Hashing ist

```
Replace each hole (_) in the configuration
    Config
        { size = 10 , hash1 = \ x -> _ + _ * x , hash2 = \ x -> _ }
and in the sequence of operations
    [ Insert _ , Insert _ , Insert _ , Insert 40 , Insert _ , Insert _ , Insert 22 ]
with a numerical expression
such that the sequence of operations transforms
    Table (listToFM
              [ (2,12),(9,41)])
to
    Table (listToFM
              [ (0,11),(1,25),(2,12),(3,89),(4,22),(6,40),(7,97),(8,64),(9,41) ])
```

Eine Lösung ist

```
Solution
    { config = Config { size = 10 , hash1 = \ x -> 6 + 3 * x , hash2 = \ x -> 1 }
```

```
    , ops = [ Insert 25 , Insert 11 , Insert 89 , Insert 40
            , Insert 97 , Insert 64 , Insert 22
            ]
    }
```

Ein Ausschnitt aus der Antwort des System ist

```
Table (listToFM [ ( 1, 25) , ( 2, 12) , ( 9, 41) ])

execute Insert 11
    insert(x,S)    x = 11
    hash1(x) = 9 (mod  10)    hash2(x) = 1 (mod  10)
    probe at address 9      contains 41
    probe at address 0      is empty, insert  x  here

result: Table (listToFM [ ( 0, 11) , ( 1, 25) , ( 2, 12) , ( 9, 41) ])
```

Die Aufgaben zum doppelten Hashing erwiesen sich beim Einsatz im Sommersemester 2017 tatsächlich als schwierig, mit einer hohen Anzahl von Fehlversuchen.

# 5 Algorithmen auf Graphen

Die Breitensuche wird durch folgenden Aufgabentyp getestet: eine Aufgaben-Instanz besteht aus einem Muster $G_M$ für die Adjazenz-Listen-Darstellung eines Graphen, einem Startknoten $s$ und einem Muster $T_M$ für einen Baum. Eine Lösung ist eine Adjazenz-Listen-Darstellung $G$ eines Graphen. Die Lösung ist korrekt, wenn $G$ zu $G_M$ paßt und der Breitensuchbaum $\mathsf{BFS}(G)$, der von `autotool` berechnet wird, zu $T_M$ paßt.

Eine Beispiel-Aufgabenstellung ist

```
Replace the holes (_) in the adjacency list of the graph G
Adjacency_List [ ( _ , [ _, 4, 7] ) , ( _, [ 5, _]) , ( 4, [ _, _])
       , ( _, [ _, _, 4, 5]) , ( _, [ 4, _, 1, 3]) , ( 3, _) , ( 6, [ _, 1, _]) ]
so that the breadth first search tree of G, starting at 3,
matches the pattern
  Branch 3 [ Branch _ [ Branch 6 _
                      , Branch 5 [ Branch 2 [ ] , Branch _ [ ] ] ] ]
```

Die Konfiguration des Aufgabengenerators ist

```
Config  { graph_generator = Graph_Generator
           { directed = True , vertices = 7 , out_degree_range = ( 1, 4) }
    , handle_graph = Show 0.7 , handle_bfs_tree = Show 0.7 , candidates = 100
    }
```

Es werden 100 Graphen $G$ erzeugt und für jeden $T = \mathsf{BFS}(G)$ bestimmt. In $G$ und $T$ werden zufällig ausgewählte Teilstrukturen versteckt, d.h., durch `_` ersetzt, so daß der Anteil der sichtbaren Funktionssymbole möglichst nahe bei 70 Prozent liegt.

Eine ähnliche Aufgabe behandelt Tiefensuch-Wälder.

## 6 Technische Details

Das System `autotool` besteht aus

- zustandsloser Semantik-Service zur Generierung von Aufgabeninstanzen und zur Bewertung von Lösungsversuchen, implementiert als XML-RPC-Service mit haxr
- Persistenz-Schicht zur Speicherung von Aufgaben, Einschreibungen, Punkten u.ä., implementiert als mysql (mariabdb) mit persistent [Sno17]
- Web-Oberfläche [Sie16], implementiert mit yesod [Sno12].
  - für Student: Bearbeiten von Aufgaben, Einsehen der Bewertung
  - für Dozent: Konfiguration, Test, Verwaltung von Aufgaben, Punkten, u.ä.

Die hier beschriebenen Aufgaben sind als Module im Semantik-Service realisiert. Die gemeinsame Modulschnittstelle ist durch diese mehrparametrige Relation (Typklasse) für $p$ (Aufgabentyp), $i$ (Typ der Instanz), $b$ (Typ der Einsendung) beschrieben:

- describe: $p \times i \to \mathsf{Text}$ beschreibt Aufgabenstellung,
- initial: $p \times i \to b$ berechnet einen syntaktisch korrekten, aber semantisch falschen Lösungsvorschlag,
- eval: $p \times i \times b \to \mathbb{B} \times \mathsf{Text}$ bewertet die Einsendung, liefert Resultat richtig/falsch und Begründung

Das Würfeln von Aufgabeninstanzen nach einer Konfiguration $c$ wird beschrieben durch

- generate: $p \times c \times \text{Seed} \to i$

wobei Seed aus der Matrikelnummer (oder einem anderen Kennzeichen des Studenten) bestimmt wird und der Initialisierung eine deterministischen Pseudozufallsgenerators dient.

Der Student gibt eine textuelle Repräsentation der Lösung (vom Typ $b$) ein. Das Texteingabefeld wird mit initial$(p, i)$ initialisiert. Mit Absicht gibt es keine grafische Eingabemöglichkeit, um Studenten an die wissenschaftliche Notation (durch Terme in einer geeigenten Signatur) zu gewöhnen.

Bei den allermeisten Aufgaben ist die konkrete Syntax für $b$ die Daten-Syntax von Haskell [has10], mit den Ausdrucksmitteln

- Literale für primitive Daten: `123`, `False`
- Paare, Tupel: `(124,False)`
- (typreine) Listen: `[(123,False),(456,True)]`
- Konstruktoren algebraischer Datentypen, mit Argumenten: `Just (123,False)`
- Records mit benannten Komponenten: `Config{size=10}`

Pretty-Printer und Parser werden automatisch generiert. Gelegentlich wird eine aufgabenspezifische konkrete Syntax verwendet, z.B. für arithmetische Ausdrücke in Hashfunktionen.

Die Eingabe-Parser [LM16] erzeugen nützliche Fehlermeldungen. Die fehlerhafte Stelle wird markiert und mögliche Fortsetzungen werden ausgegeben. Jedes Eingabefeld (für den

Studenten: das für den Lösungsversuch, für den Dozenten: das für die Konfiguration der Instanz oder des Instanzengenerators) ist typisiert und wird automatisch mit einem Verweis auf die API-Dokumentation des jeweiligen Typs versehen.

Die hier gezeigten Aufgabe verwenden das Prinzip, daß Teile einer Datenstruktur (Bsp: Koeffizienten der Hashfunktion, Elemente der Liste von Operationen) in der Aufgabenstellung versteckt werden und zur Lösung zu ergänzen sind. Ein Typ $p$ dient in diesem Sinne als Vorlagentyp, wenn für ihn diese Eigenschaften deklariert werden:

- der zugehörige Basistyp: `type (Base p)`
- der Mustervergleich $\text{match} : p \times \text{Base}(p) \to \mathbb{B} \times \mathsf{Text}$
- die Injektion inject: $\text{Base}(p) \to p$
- das pseudozufällige Verstecken von Bestandteilen $\text{obfuscate} : \text{Base}(p) \times \text{Seed} \to p$.

## 7 Diskussion

Der `autotool`-Semantik-Service wird ebenfalls vom autolat-Plugin [FS10] für OpenOLAT [Fre17] unterstützt und an der Universität Leipzig angewendet. Ich möchte `autotool` gern in weitere Systeme integrieren. Eine Verwendung der QTI-Schnittstelle [IMS15] ist naheliegend, scheint aber derzeit nicht möglich, da im Standard weder das externe Generieren von Aufgaben noch das externe Bewerten von Einsendungen vorgesehen sind, sondern höchsten Zeichenkettenvergleiche, die von QTI-Player selbst ausgeführt werden. Das ist für eine semantische Bewertung mit sinnvoller Rückmeldung völlig unzureichend.

Die hier beschriebenen Aufgaben habe ich für die Vorlesung Algorithmen und Datenstrukturen im Sommersemester 2017, die ich vertretungsweise übernommen habe, überarbeitet bzw. neu entwickelt. Dabei haben ca. 80 Studenten über ein Semester hinweg jede Woche 1 bis 2 `autotool`-Aufgaben bearbeitet, mit ca. 10 Einsendungen pro Aufgabe (teilweise aber deutlich mehr).

Ich habe damit die eigentlich vorgesehenen Java-Programmier-Übungen ersetzt. Die `autotool`-Aufgaben schienen mir dem didaktischen Ziel des Verständnisses der Algorithmen nützlicher. Der Aufwand zum Lösen und Korrigieren von Java-Programmen ist für Student und Dozent deutlich höher, so daß Programmieraufgaben tatsächlich nur dreimal während des Semesters gestellt wurden. Tatsächlich habe ich wohl die Arbeit verschoben: vom (z.T. händischen) Korrigieren der Java-Aufgaben zum Programmieren der `autotool`-Aufgaben. Die Studenten haben deswegen in dieser Vorlesung keine Programme selbst geschrieben und (außer Pseudocode im Skript und `autotool`-Quelltexten) kaum welche gesehen — das finde ich nicht so tragisch, zumal es parallel eine Vorlesung zur anwendungsorientierten Java-Programmierung gibt und vorher eine zu Grundlagen der Programmierung.

Durch die `autotool`-Aufgaben haben sich die Studenten auch ohne Programmieren mit den Algorithmen beschäftigt, indem sie ihre Ausführung durch das `autotool` beobachtet und durch geeignete Eingaben zielgerichtet gesteuert haben. Für die Übungen habe ich damit Zeit gewonnen für die Diskussion von Aufgaben zu Eigenschaften von Algorithmen. (Die Prüfungszulassung ergibt sich aus beiden Anteilen.) Eine wissenschaftliche fundierte Einschätzung des tatsächlichen Nutzens (und Aufwandes) von `autotool`-Aufgaben scheint mir hier nicht möglich, da zu viele weitere Parameter variieren: nicht nur Änderung der Übungs-Aufgaben, sondern auch im Vortragsstil, im Skript, in Klausuraufgaben.

Berichten von Kollegen sowie eigene Erfahrung mit `autotool` in anderen Vorlesungen [Wal15, Wal14] bestätigen, daß Studenten das System schnell verstehen und gern benutzen und insbesondere die sofortigen, ausführlichen und objektiven Antworten begrüßen.

## Literatur


[Fre17] Frentix GmbH. LMS OpenOLAT. `https://www.openolat.com/`, 2017.

[FS10] Bertram Felgenhauer and Klemens Schölhorn. Autolat: autotool integration for OpenOLAT. `https://github.com/klemens/openolat/wiki#autolat`, 2010.

[G+17] Emden Gansner et al. Graphviz - Graph Visualization Software. `http://www.graphviz.org/`, 2017.

[has10] Haskell 2010 Language Report. `https://www.haskell.org/onlinereport/haskell2010/`, 2010.

[IMS15] IMS Global Learning Consortium. IMS question and test interoperability v2.2 final specification. `https://www.imsglobal.org/question/index.html`, 2015.

[LM16] Daan Leijen and Paolo Martini. parsec: Monadic Parser Combinators. `https://hackage.haskell.org/package/parsec`, 2016.

[RW02] Mirko Rahn and Johannes Waldmann. The Leipzig autotool System for Grading Student Homework. In *Intl. Workshop on Functional and Declarative Programming in Education (FDPE 2002)*. Technical Report 0210, University of Kiel, 2002.

[Sie16] Marcellus Siegburg. *REST-orientierte Refaktorisierung des E-Learning-Systems Autotool*. master's thesis, HTWK Leipzig, 2016. `http://www-ps.informatik.uni-kiel.de/~msi/thesis/thesis.pdf`.

[Sno12] Michael Snoyman. Developing Web Applications with Haskell and Yesod, 2012.

[Sno17] Michael Snoyman. persistent: Type-safe, multi-backend data serialization. `https://hackage.haskell.org/package/persistent`, 2017.

[THC09] Ronald Rivest Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms*. MIT Press, 2009.

[Wal14] Johannes Waldmann. Automated Exercises for Constraint Programming. In *28th Workshop on (Constraint) Logic Programming (WLP 2014), Wittenberg*, 2014. `http://www.imn.htwk-leipzig.de/%7Ewaldmann/talk/14/wlp/auto/`.

[Wal15] Johannes Waldmann. Automatisierte Bewertung und Erzeugung von Übungsaufgaben zu Prinzipien von Programmiersprachen. In *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung KPS 2015, Pörtschach*, 2015. `http://www.imn.htwk-leipzig.de/%7Ewaldmann/talk/15/kps/paper/`.

[Wal17] Johannes Waldmann. Leipzig autotool. `https://gitlab.imn.htwk-leipzig.de/autotool/all0`, 2017.