Algorithmen in Haskell für den 20. Bundeswettbewerb Informatik 2001/2002

Besondere Lernleistung

Eingereicht von
Heinrich-Gregor Zirnstein

Betreut durch
Herrn St. Müller
Anton-Philipp-Reclam-Schule

Leipzig, 17. Januar 2003

Inhaltsverzeichnis

1 Einleitung

Als im Herbst des Jahres 2001 die Aufgaben der ersten Runde des 20. Bundeswettbwerbs Informatik erschienen waren, gab sie mir mein Informatiklehrer, Herr St. Müller, im Wissen um die Vorteile von freiwilligem Lernpensum in der Schule. Ich durfte mich noch in derselben Unterrichtsstunde an der ersten Aufgabe versuchen und nutzte dafür die imperative Programmiersprache Delphi.

Zufällig wurde ich kurz danach in der Vorlesung "Kombinatorische Spieltheorie" von Herrn Dr. Waldmann an der Leipziger Universität, die ich im Rahmen der "Leipziger Schülergesellschaft für Mathematik" besuchte, auf die mir bisher unbekannte Programmiersprache Haskell aufmerksam: Herr Dr. Waldmann hatte die Aufgabe gestellt, eine gewisse Zahlenfolge zu berechnen. Für die Lösung benötigte ich ca. einer Seite Quellcode in der imperativen Programmiersprache C, Herr Dr. Waldmann jedoch nur eine Zeile ("zwei, wenn das Terminal klein ist.") in Haskell. Fasziniert von dieser Kürze beschloss ich, unverzüglich Haskell zu lernen, wozu freilich für gewöhnlich hierzulande ein Student der Informatik gegebenenfalls erst in höheren Semestern kommt.

Völlig selbstständig eignete ich mir nun die Grundlagen von Haskell an, und zwar durch selbstbestimmtes Lernen, wie ich es gewohnt bin und schon lange in den Unterrichtsstunden meines Mathematik- und Informatiklehrers, Herrn St. Müller, praktizieren durfte. Dies bedeutete zunächst, dass ich mir ein gutes Buch [Pep99] selbst suchte und darin las. Außer zusätzlichem Literaturstudium bei zunehmendem Schwierigkeitsgrad der Aufgabenstellungen brauchte ich keine weitere Hilfe und das erworbene Wissen setzte ich dann auch gleich am heimischen Computer beim Lösen der Aufgaben der ersten beiden Runden des Bundeswettbewerbs Informatik um.

Die vorliegende Arbeit soll diese besondere Lernleistung in überschaubarer Form dokumentieren. Dazu werde ich zuerst kurz auf den Ablauf des Bundeswettbewerbs Informatik und meine Teilnahme daran eingehen, anschließend Grundlegendes über Haskell darlegen, um dann exemplarisch meine Lösung zu nur einer Aufgabe aus der zweiten Runde zu präsentieren und zu besprechen, die meines Erachtens am deutlichsten zeigt, welche konkreten Konsequenzen und Vorteile das Programmieren in der funktionalen Programmiersprache Haskell hat.

Auf der als Anlage beigefügten Diskette sind neben dieser schriftlichen Arbeit hier schließlich auch meine natürlich selbstständig zu erstellenden Lösungen in Haskell der ersten beiden Wettbewerbsrunden verfügbar. Die Endrunde des Bundeswettbewerbs war mündlich zu absolvieren und es hat mich freundlicherweise Herr St. Müller begleitet. Außerdem möchte ich mich hier bei ihm und bei Herrn Dr. Waldmann für ihre Hinweise zur endgültigen Fassung dieser Dokumentation bedanken.

2 Rahmenbedingungen und Methodenwahl

2.1 Zum Ablauf des 20. Bundeswettbewerbs Informatik

Der Bundeswettbewerb Informatik ist in drei Runden aufgeteilt: In der ersten Runde werden fünf Aufgaben gestellt, von denen drei in Hausarbeit vollständig gelöst werden müssen, um in die zweite Runde zu gelangen. Ich habe drei von den fünf Aufgaben sehr gut, eine weitere vorsichtshalber gelöst und eine aus technischen Gründen ausgelassen, da mein HASKELL-System HUGS 98 auf dem Apple Macintosh nicht graphikfähig ist. Jede Lösung sollte dabei neben einem Programm auch eine Dokumentation der dazugehörigen Lösungsidee, eine Dokumentation des Programms selber und ein Ablaufprotokoll von Testbeispielen enthalten, so dass ich in der ersten Runde knapp 30 Seiten einsandte.

Dies gilt auch für die zweite Runde, in die ich also gelangte und in der es dann nur noch drei, deutlich schwerere Aufgaben gab, die nun alle gelöst werden mussten. Durch HASKELL hatte ich jedoch den enormen Vorteil des kurzen Quellcodes, so dass ich am Ende nur knapp 60 Seiten einzusenden hatte, wo manch anderer bis zu 160 Seiten zu Papier brachte. Dazu war freilich die Vertiefung meiner HASKELL-Kenntnisse durch Artikel aus dem Internet wie [Jon01] oder [Haskell98] nötig.

Die Jury war mit meinen Lösungen vollauf zufrieden und lud mich als einen von 24 Teilnehmern der Endrunde vom 29. Oktober bis zum 1. November 2002 nach Frankfurt am Main ein. Dort wurden mit jedem Teilnehmer zwei Einzelgespräche geführt. Ich wurde natürlich u.a. über HASKELL ausgefragt, da diese funktionale Programmiersprache beim Bundeswettbewerb Informatik von keinem Teilnehmer vor mir verwendet worden war. Ich hatte mich zuvor in die Literatur zur theoretischen Informatik und funktionalen Programmierung vertieft, z.B. in [BW92] oder [Erw99], und konnte insbesondere zu den theroretischen Grundlagen funktionaler Programmierung, wie dem Lambda-Kalkül, ausführlich Auskunft geben. Anschließend mussten je zweimal verschiedene Aufgaben im Team zu vier Teilnehmern theoretisch bearbeitet und über die Lösungen Vorträge gehalten werden. Die Themen dieser Gruppenaufgaben waren am ersten Prüfungstag der Entwurf und die Programmierung von Fußball spielenden Robotern und am zweiten Prüfungstag die möglichst effiziente Lösung NP-vollständiger Probleme, d.h. das Erreichen von möglichst kleinen Konstanten c in der typischen exponentiellen Laufzeit $\mathcal{O}(c^n)$ von Algorithmen für solche Probleme.

Zur Siegerehrung am letzten Wettbewerbstag wurde ich dann gleich mit vier Preisen ausgezeichnet: ich wurde einer der Bundessieger, erhielt außerdem einen “Preis für die beste Einzelleistung”, erhielt zusammen mit meiner Gruppe den “Preis für die beste Gruppenleistung” und schließlich gab es für mich als den jüngsten Teilnehmer den “Preis für den Besten unter den jüngeren Teilnehmern”.

2.2 Die funktionale Sprache HASKELL im Vergleich zu imperativen Programmiersprachen

Worin liegen nun die Ursachen für die Kürze der deswegen von mir bevorzugten Programmiersprache HASKELL? Sie sind vorrangig darin zu suchen, dass HASKELL als funktionale Sprache auf anderen Prinzipien beruht als die herkömmlichen imperativen Sprachen wie C oder PASCAL.

In konventionellen, "imperativ" genannten Programmiersprachen, wird eine Berechnung als das sequentielle Abarbeiten von Arbeitsschritten der Form "Erst tu dies, dann tu das" angesehen. In funktionale Programmiersprachen werden jedoch alle Rechnungen, wie der Name es schon vermuten lässt, als mathematische Funktionen angesehen, die aus gegebenen Parametern ein Funktionsergebnis machen. So können wir die Berechnung des Quadrates einer Zahl als Funktion $q : \mathbb{N} \to \mathbb{N}$ mit der Definition $q(n) = n \cdot n$ ansehen.

Eines der wichtigsten Merkmale solcher Funktionen ist nun, dass ihr Ergebnis nur von den Parametern, hier n, abhängt. In imperativen Programmiersprachen besteht das Resultat einer Funktion — meist werden sogar Prozeduren verwendet, die gar kein Ergebnis erst zurückliefern — hingegen darin, den gegebenen internen Zustand des Computers in Form des Arbeitsspeichers zu verändern. Ein Beispiel wäre eine Funktion f ohne Parameter, die einer globale Variable um eins erhöht und das erhaltene Ergebnis zurückliefert. Wird f zweimal hintereinander ausgeführt, so unterscheiden sich die beiden Ergebnisse trotz gleicher Parameter (es gibt nämlich keine!). Dies wird in funktionalen Sprachen untersagt.

Die Berechnung einer Funktion in funktionalen Sprachen entspricht nun der Auswertung eines Funktionsausdrucks. So wird z.B. das "Programm" $q(2 + 5)$ mit der oben definierten Quadratfunktion folgendermaßen ausgewertet ("$\Rightarrow$" heißt "wird umgeformt zu"):

$$q(2 + 5) \Rightarrow q(7) \Rightarrow 7 \cdot 7 \Rightarrow 49.$$

Denkbar wäre aber auch die Auswertung:

$$q(2 + 5) \Rightarrow (2 + 5) \cdot (2 + 5) \Rightarrow 7 \cdot (2 + 5) \Rightarrow 7 \cdot 7 \Rightarrow 49.$$

Die erste Auswertungsstrategie nennt man auch eager-evaluation, hier werden zuerst die Parameter der Funktion und dann deren Rumpf ausgewertet. Dies ist z.B. in einer anderen funktionalen Sprache namens ML realisiert. Nach der zweite Strategie hingegen wird erst erst der Rumpf durch seine Definition ersetzt und die Parameter der Funktion werden nur wenn unbedingt nötig ausgewertet. Es fällt auf, dass der Ausdruck $(2 + 5)$ hier unnötigerweise zweimal ausgewertet wird, was man umgeht, indem man die beiden Ausdrücken miteinander verknüpft, die ja ein und derselbe Parameter sind. In Haskell wird diese, lazy-evaluation genannte Form verwendet. Sie braucht stets weniger Reduktionsschritte und garantiert immer den Halt der Auswertung, sofern dieser möglich ist. Mit ihr lassen sich sogar (potentiell) unendliche Listen ohne weiteres realisieren!

In diesem Zusammenhang möchte ich kurz anbringen, dass ich Haskell gegenüber ML oder gar älteren funktionalen Sprachen wie Lisp auch wegen der lazy-evaluation und den damit verbundenen unendlichen Listen bevorzuge. Aber auch die Syntax von Haskell erscheint mir intuitiver und mehr auf praktische Bedürfnisse zugeschnitten, im Gegensatz z.B. zu den "Klammerbergen" in Lisp. Ein weiteres Argument für die Programmiersprache Haskell ist ihre recht weitgehende Verbreitung.

Prinzipiell gesehen ist die Auswertungsreihenfolge aber egal, das Ergebnis der Funktion, nämlich 49, ändert sich nicht. In imperativen Sprachen spielt die Reihenfolge aber eine fundamentale Rolle. Es macht eben einen großen Unterschied, ob erst eine Zahl `x` quadriert und dann eins hinzugezählt, oder erst eins hinzugezählt und die Zahl dann quadriert wird. Beides kann allein durch die Anweisungen `x=x+1` und `x=x*x` erreicht werden, die dafür jedoch eine bestimmte Reihenfolge besitzen müssen. Ein weiteres wichtiges Prinzip der imperativen Programmierung sind Programmschleifen, die in funktionalen Sprachen durch die mächtigere und fundamentalere Rekursion ersetzt werden.

Dieses Prinzip der Berechnung mittels Funktionen führt zu mehr Transparenz des Programmcodes, d.h., man muss nicht auf eventuelle Feinheiten aufpassen, die sich durch das Verändern des internen Speichers, man denke nur an "wilde" Zeiger, und die Reihenfolge der Operationen ergeben. Auch lassen sich nun mathematische Methoden auf das Programm anwenden, dessen Korrektheit man so beweisen und welches sogar durch mathematische Überlegungen fast direkt erstellt werden kann.

Die eigentliche Macht funktionaler Sprachen wie HASKELL rührt aber noch von weiteren Eigenschaften her: zum einen sind Funktionen "first class citiziens", d.h. man kann mit ihnen umgehen, als wären es ganz gewöhnliche Werte wie Zahlen oder andere Daten. Man kann sie mittels anderer Funktionen "höherer Ordnung" auch zu neuen Funktionen durch Hintereinanderausführung verknüpfen etc. In imperativen Sprachen wie C oder PASCAL müssten dafür umständliche Zeiger auf Funktionen verwendet werden, mit denen man nicht weiter operieren kann.

Auch die Polymorphie und ein dazugehöriges strenges Typsystem sind ein grundlegender Bestandteil von HASKELL d.h., es ist z.B. möglich, den Datentyp einer Liste zu definieren, die beliebige gleichartige Daten enthalten kann. In C++ müsste man so etwas umständlich über sogenannte "Templates" lösen, in anderen Sprachen wäre das noch "schlimmer". Das Typklassensystem von HASKELL geht noch weiter. So ist es möglich, Sortierverfahren zu definieren, die zwar den genauen Typ (ganze Zahl, reelle Zahl, Liste o.ä.) ihrer Parameter nicht kennen, wohl aber wissen, dass auf ihnen eine Ordnungsrelation existiert, den beim Sortieren einzig entscheidenden Punkt.

Diese Sichtweise und Eigenschaften von HASKELL lenken die Konzentration des Programmierers von der konkreten Steuerung des Computers auf die inhaltlichen Aspekte eines Programmentwurfs und die abstrakte Implementierung von Algorithmen hin, was sich auch in der Kürze und Prägnanz des Quellcodes äußert. Der Programmierer kann seine Algorithmen in der Abstraktionsebene formulieren, in der er sie auch entwirft und umgeht damit irrelevante Details wie die Auswertungsreihenfolge und die konkrete Aufteilung des Speichers. Des Weiteren können die abstrakt formulierten Algorithmen vor allem durch Funktionen höherer Ordnung und das Typklassensystem allgemein gehalten und dementsprechend frei miteinander kombiniert werden, dass erst so wirklich modulare Programmierung ermöglicht wird, die softwaretechnisch sehr effizient und wünschenswert ist. Dies besagt, dass einzelne Programmteile in vielen Situationen zweckmäßig wieder verwendet werden können. In meiner noch folgenden Lösung zu einer Aufgabe des Bundeswettbewerbs Informatik zeigt sich dies sehr deutlich am Konzept der Halbgruppen.

Warum programmiert die Softwareindustrie dann immer noch in C und nicht in HASKELL? Funktionalen Sprachen wird nachgesagt, sie seien sehr langsam, was zum Teil richtig ist, denn funktionale Programme bedürfen öfters etwas ungewöhnlicher Optimierungen. Auch schafft die Persistenz aller Daten ein prinzipielles Problem, so ist es in einer reinen funktionalen Programmiersprache z.B. nicht möglich, ein Array mit Zugriffszeiten von $\mathcal{O}(1)$ zu implementieren. Näheres zur Problematik der Persistenz und der eng damit verbundenen Amortisation findet sich in [Oka99]. Die Compiler-Technik ist aber heutzutage so weit fortgeschritten, dass sie sich mit althergebrachten Methoden durchaus messen kann. Auch die Interaktion eines funktionalen Programms mit einem Benutzer, einem anderen Programm oder gar die Realisierung verschiedener Threads, was ja alles einen zeitlichen Ablauf erfordert, besitzt mittlerweile eine zufriedenstellende Lösung durch eine Erweiterung der Sprache um Funktionen mit Seiteneffekt; in HASKELL elegant über Monaden realisiert.

Es würde nun den Rahmen dieser Arbeit sprengen, ausführlicher darauf einzugehen,

wie die eben beschriebenen und weitere Konzepte in HASKELL umgesetzt werden. Mehr Informationen dazu finden sich z.B. auf der HASKELL-Website `http://www.haskell.org` und, speziell zur HASKELL-Syntax, in der Sprachdefinition [Haskell98] des derzeitigen Standarts HASKELL-98. Für das volle Verständnis meiner nun folgenden Lösung zu einer Aufgabe der zweiten Runde des Bundeswettbewerbs Informatik sind HASKELL-Vorkenntnisse erforderlich, aber die prinzipielle Lösungsidee sollte auch dem weniger versierten Leser zugänglich werden.

3 Lösung zu einer Aufgabe der zweite Runde des 20. Bundeswettbewerbs Informatik

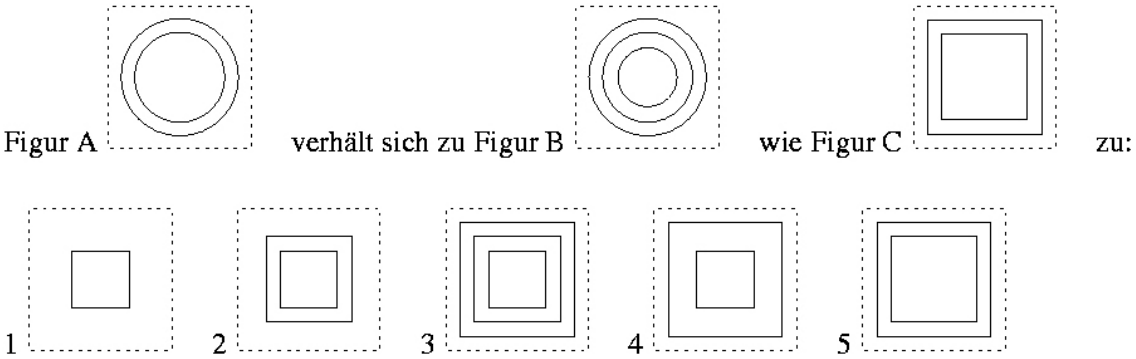
Im Folgenden möchte ich nur einen Teil meines Beitrags zum Bundeswettbewerb Informatik vorstellen, da mein gesamter Beitrag den geforderten Umfang dieser Arbeit bei weitem überschreitet. Ich habe mich für meine Lösung der dritten Aufgabe der zweite Runde entschieden, da ich der Ansicht bin, hier die Fähigkeiten von HASKELL am meisten ausgenutzt zu haben. Zuerst präsentiere ich die Aufgabenstellung und entwickle anschließend meine Lösung.

3.1 Aufgabenstellung

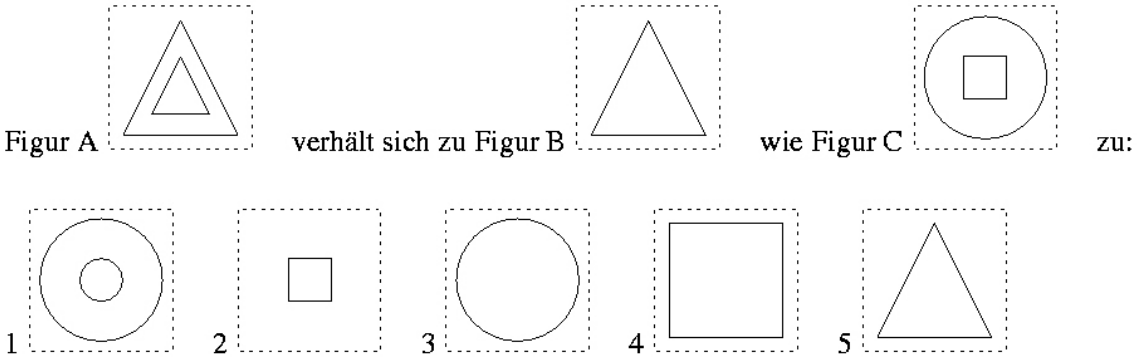
Die offizielle Aufgabenstellung hatte folgenden Wortlaut (siehe auch [BWINF20]):

"Gerda Bauch und ihrem Team vom Fernsehsender Contra 6 gehen langsam die Beispiele für ihre Spielshow um den Intelligenz-Quotienten aus. In ihrer Show zum Testen der 'allgemeinen Intelligenz' werden immer neue Beispiele gebraucht. Typisch sind für einige Tests Fragestellungen der folgenden Art: Figur A verhält sich zu Figur B wie Figur C zu einer von fünf anderen vorgegebenen Figuren. Du kannst dem Team sicher helfen, Tests dieses Typs zu erzeugen. Hier drei Beispiele:

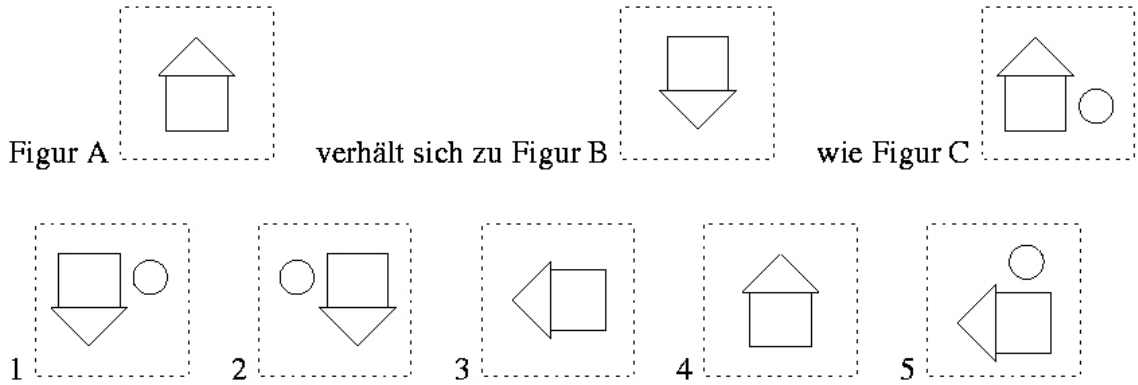
1. Beispiel



2. Beispiel



3. Beispiel



Aufgabe

1. Definiere einen Katalog einfacher Basisobjekte, aus denen Figuren für die Tests zusammengesetzt werden sollen. Beschreibe Regeln, nach denen Figuren aus den Basisobjekten aufgebaut werden können. Durch welche Transformationen werden die Figurenpaare ineinander übergeführt? Gib an, wie Ähnlichkeiten definiert werden können. Beschreibe darauf aufbauende Möglichkeiten zur programmtechnischen Umsetzung.

2. Entwickle ein Werkzeug, das den Entwurf beliebiger Beispiele für IQ-Tests obiger Art möglichst gut unterstützt. Es sollen jeweils zwei Paare von Figuren mit gleicher visueller Analogie und vier weitere, möglichst 'nahestehende' Figuren erzeugt und daraus Testaufgaben konstruiert werden. Demonstriere die Fähigkeiten dieses Werkzeugs, indem du mit seiner Hilfe mindestens drei weitere Tests erzeugst.

3. Welchen IQ kann eigentlich ein Computerprogramm haben? Entwickle ein Programm, das die oben beschriebenen und typische, erzeugbare Aufgaben auf der Grundlage der definierten Basisobjekte und der möglichen Transformationsregeln lösen kann."

3.2 Meine Lösung

Im Bundeswettbewerb Informatik wurde bekanntermaßen gefordert, zu jedem Programm eine Lösungsidee, eine Dokumentation des Programms selbst sowie ein Ablaufprotokoll einiger Testbeispiele anzufertigen. Ich verzichte hier auf diese strikte Trennung von Programm, Lösungsidee und Dokumentation und werde Lösungsidee und Quelltext im Folgenden zusammen entwickeln, so wie es in der Literatur zu Haskell durchaus üblich ist.

In Haskell gibt es nämlich das sogenannte "literate programming", bei dem der Programmierer beim Schreiben einer Quelltextdatei eigentlich nur Kommentar verfasst und

erst die Zeilen, denen jeweils ein > vorausgestellt ist, vom Interpreter als ausführbarer HASKELL-Code anerkannt werden. Ein Beispiel für diese "inverse comment convention" wäre

```
>   addiere_eins x = x + 1.
```

Der ungewöhnliche Aspekt dieser Vorgehensweise ist also, dass dieses Dokument, das wie gesagt auf der beigelegten Diskette auch in elektronischer Form vorliegt, damit auch von einem HASKELL-Interpreter ausführbar ist, z.B. mit dem für fast alle Betriebssysteme frei verfügbaren HUGS98 (`http://www.haskell.org/hugs`).

Ich biete hier also eine gegenüber meiner Originallösung veränderte Variante an, die ich auch noch inhaltlich, der Verständlichkeit dieser Dokumentation halber, ein wenig überarbeitet und zusätzlich kommentiert habe. Im laufenden Text zeige ich auch immer wieder die Vorteile der Programmierung in HASKELL auf.

Möglicherweise könnte diese Arbeit als Hilfsmittel zum selbstständigen Erlernen von HASKELL von Nutzen sein. Herr Dr. Waldmann schlug vor, diese Arbeit im Internet zu veröffentlichen, damit seine Studenten und (Schüler-)Praktikanten damit lernen können. In kleinerer Schrift habe ich deshalb noch technische Erläuterungen zu HASKELL-Sprachelementen und zu von mir verwendeten Bibliotheksfunktionen an einige Quelltextabschnitte angefügt.

3.2.1 Teilaufgabe 1 - Figuren, Basisobjekte, Transformationen

Zuerst müssen der **Programmkopf** definiert und einige zusätzliche HASKELL-Bibliotheken eingebunden werden. Die Standardbibliothek, das HASKELL-`Prelude`, wird automatisch eingebunden.

```
>   module IQ where
>
>   import List
>   import Random
>   import Maybe
>   import GraphicsUtils
```

Eine **Figur** soll nun einfach eine Menge von Basisobjekten sein, die sozusagen übereinandergelegt werden. In HASKELL führen wir dazu ein Typsynonym `Figur` ein, das nichts weiter als eine Liste von Basisobjekten ist:

```
>   type Figur = [BasisObjekt]
```

Schon allein hier zeigt sich eine Stärke von HASKELL nämlich dass beliebig lange Listen von Daten beliebiger Natur ohne weiteres verwendbar sind, die in Sprachen wie C oder PASCAL erst recht aufwändig mit Zeigern zu implementieren wären. Hier sind sie hingegen schon fast Bestandteil der Sprache.

Wir werden später noch einen Test auf Gleichheit zweier Figuren benötigen, der entscheidet, ob zwei Figuren die gleichen Basisobjekte enthalten:

```
>   eqFigur f1 f2 = length f1' == length f2' &&
>               and [b `elem` f2' | b <- f1']
>     where
```

```
>          f1' = nub f1
>          f2' = nub f2
```

Die in der HASKELL-Bibliothek `List` definierte Funktion `nub` sorgt dafür, dass alle doppelten Basisobjekte aus jeder Figur entfernt werden. Erst dann werden die Figuren auf Gleichheit hin überprüft werden, indem getestet wird, ob beide dieselbe Anzahl von Basisobjekten besitzen und die eine Figur `f1'` in der anderen Figur `f2'` enthalten ist.

Die `Prelude`-Funktion `length` liefert die Länge, also die Anzahl der Elemente in einer Liste zurück und `&&` bezeichnet das logische UND. Der Ausdruck in eckigen Klammern `[b `elem` f2' | b <- f1']` ist eine sogennannte "List-Comprehension" zum bequemen Konstruieren von Listen, die wie die mathematische Schreibweise für Mengen funktioniert. Rechts vom senkrechten Strich `|` steht hier, dass der Wert `b` alle Werte der Liste `f1'` durchlaufen soll. Er wird quasi von `f1'` "gefüttert", symbolisiert durch den Pfeil `<-`. Links von `|` steht dann, wie die Elemente der zu erzeugenden Liste berechnet werden. In diesem Fall sind es die Wahrheitswerte `b `elem` f2'`, die mittels der im `Prelude` definierten Funktion `elem` angeben, ob das Element `b` in der Liste `f2'` enthalten ist. Der Ausdruck `b `elem` f2'` ist im Übrigen gleichbedeutend mit `elem b f2'`, d.h. die Akzente ` dienen dazu, die Funktion der besseren Lesbarkeit wegen in die Mitte seiner beiden Parameter zu schreiben. Die ebenfalls im `Prelude` definierte Funktion `and` gibt nur dann `True` zurück, wenn alle Einträge in der eben konstruierten Liste auch `True` sind, also wenn jedes Element aus `f1'` auch in `f2'` enthalten ist und die erste Liste somit in der zweiten enthalten sein muss.

Die **Basisobjekte** sollen demnach aus einer einfachen Form und ihrer Lage in der Ebene bestehen. Ich habe also Form und Lage gleich zu einem Basisobjekt verschmolzen, obwohl die Bezeichnung "Basisobjekt" eher auf seine Form als seine Lage zutrifft.

```
>   data BasisObjekt = Obj {  lage    :: AehnlichkeitsAbb,
>                             form    :: Form
>                             } deriving (Eq, Ord, Show)
```

Ein `BasisObjekt` wird hier als Datentyp mit einem Konstruktor `Obj` definiert, der als Parameter die beiden `lage` und `form` genannten Werte erhält. Die Definition `data BasisObjekt = Obj AehnlichkeitsAbb Form` wäre auch denkbar gewesen, aber so kann man bequem über den Namen auf das jeweilige Feld zugreifen. Dieses Vorgehen ist also vergleichbar mit einem `record` in PASCAL oder einer `struct` in C. Der Teil mit dem HASKELL-Schlüsselwort `deriving` soll uns erst einmal nicht kümmern.

Auf die Darstellung der Lage eines Objektes komme ich später noch zu sprechen, ich will hier nur anmerken, dass sie auch die Größe des Objektes definiert.

Der **Katalog der Basisobjekte**, kurz Basisobjektliste, besteht aus Basisobjekten von vier Standardformen, nämlich Kreis, Quadrat, in etwa gleichseitiges Dreieck und eine Art Dach, die in verschiedenen Größen und Positionen auftreten. Zufällige Zusammenstellungen von Objekten der Basisobjektliste sollten halbwegs vernünftige aussehende Figuren erzeugen.

```
>   basis_objekte =  [[Obj {lage = pos . groesse, form = form}] |
>                          form <- [kreis, quadrat, dreieck, dach],
>                          groesse <- map strecken [1..3],
>                          px <- [-20,0,20], py <- [-20,0,20],
>                          let pos = verschieben (px,py)]
```

Hier handelt es sich um eine komplexere List-Comprehension, die aus mehreren Listen "gefüttert" wird, die hinter den vier Pfeilen `<-` stehen. Mit `let` können lokale Werte, wie hier `pos`, definiert werden. Die verwendeten Funktionen wie z.B. `strecken` werden später noch definiert.

Generell habe ich für die **Form** eines Basisobjektes nur die Arten Kreis und Polygon zugelassen, Quadrat, gleichseitiges Dreieck und Dach lassen sich durch Polygone modellieren. Ein `kreis` soll von der Größe her dann genau in das unten definierte `quadrat` passen.

```
>   data Form = Kreis | Polygon [Punkt]   deriving (Eq,Ord,Show)
>   kreis     = Kreis
>   quadrat   = Polygon [(-10,-10), (10,-10), (10,10), (-10,10)]
>   dreieck   = Polygon [(-10,10), (0,-10), (10,10)]
>   dach      = Polygon [(-10,10), (0,0), (10,10)]
```

Die **Lage eines Basisobjektes** ist nun einfach nur eine Ähnlichkeitsabbildung der Ebene auf sich selbst. Sie gibt an, dass das zu zeichnende Objekt gerade das Bild der zugrunde liegenden Form unter dieser Abbildung ist, wobei eine Form also am besten ein Polygon oder Kreis um den Koordinatenursprung ist, wie ich es oben definiert habe. Der Streckungsfaktor der Ähnlichkeitsabbildung ist damit ein Maß für die Größe des Objektes, die also in die Lage der Figur integriert worden ist.

Eine **Ähnlichkeitsabbildung** lässt sich bekanntlich durch die Abbildungsregel

$$\vec{x} \mapsto \lambda \cdot A\vec{x} + \vec{d}$$

beschreiben, wobei A eine orthogonale Matrix, λ der Streckungsfaktor, $\vec{d}$ der zusätzliche Verschiebungsvektor und $\vec{x}$ der abzubildende Vektor ist. Ich habe mir nicht die Mühe gemacht, eine Ähnlichkeitsabbildung in dieser Form in Haskell zu implementieren, ich habe sie einfach als generelle Abbildung der Ebene auf sich selbst definiert, also von Punkten auf Punkten, die durch ihre Koordinaten repräsentiert werden:

```
>   type Koordinate       = Int
>   type Punkt            = (Koordinate, Koordinate)
>   type AehnlichkeitsAbb = Punkt -> Punkt
```

Eine solche Abbildung ist in Haskell nun ein "first class citizien", d.h. man kann mit ihr umgehen, als wäre sie ein ganz normaler Wert. Man kann Abbildungen so z.B. auch über den Operator "." miteinander verketten, d.h. eine Funktion kreieren, die gerade der Hintereinanderausführung der beiden ursprünglichen Funktionen entspricht. Es gilt also `(f . g) x = f (g x)`. In C oder Pascal wäre das Verknüpfen von Funktionen zu neuen gar nicht möglich, da dies eine weit weniger einfache Speicherverwaltung erfordern würde.

Dies birgt natürlich die Gefahr, dass sich eine "böse", sprich beliebige Abbildung der Ebene auf sich, unter dem Namen "Ähnlichkeitsabbildung" einschleichen kann, was in meinem Programm aber nicht auftritt. Für die Gleichheit zweier Basisobjekte muss aber die Gleichheit von zwei Ähnlichkeitsabbildungen geprüft werden, wozu die oben angeführte Charakterisierung unabdinglich ist. Durch Einsetzen von drei nichtkollinearen Punkten, am besten von `(0,0)`, `(1,0)` und `(0,1)`, kann dann zuverlässig geprüft werden, ob zwei **Ähnlichkeitsabbildungen übereinstimmen**.

```
>   instance Eq AehnlichkeitsAbb where
>       abb1 == abb2 =   abb1 (0,0) == abb2 (0,0)
>                     && abb1 (1,0) == abb2 (1,0)
>                     && abb1 (0,1) == abb2 (0,1)
```

Hier wird der Typ `AehnlichkeitsAbb` zu einer Instanz der in Haskell vordefinierten Typklasse `Eq` gemacht, die alle Typen umfasst, für die der Gleichheitstest `==` definiert ist. Der Gleichheitstest für

Ähnlichkeitsabbildungen besteht hier eben darin, zu testen, ob die Bilder der oben genannten Punkte gleich sind. Damit der Interpreter HUGS diese Definition anerkennt, muss er übrigens mit der Option `-98` gestartet werden, da der HASKELL98-Standard vom zu instanziierenden Typ eine strengere Form erwartet.

Das Typklassensystem offenbart hier eine weitere Stärke von HASKELL: der Gleichheitstest zwischen Ähnlichkeitsabbildungen ist durch diese Definition allen weiteren Funktionen bekannt, so z.B. auch den Bibliotheksfunktionen zum Überprüfen des Enthalten-Seins in einer Liste. Das von mir schon weiter oben verwendete Schlüsselwort `deriving` impliziert gar die automatische Zuordnung zu entsprechenden Typklassen, die in HASKELL der Bequemlichkeit halber bereitgestellt wird. Den oben besprochenen Gleichheitstest von Figuren habe ich nicht in Typklassen eingebunden, da HASKELL die Gleichheit von Listen automatisch aus der Gleichheit der Listenelemente ableitet, aber nicht im Sinne von Mengengleichheit, so wie es hier benötigt wird. Ein nochmalige Definition wird aber vom Interpreter (ohne Zusatzoption) nicht akzeptiert, da ja dann zwei Definitionen vorhanden wären.

Ebenso lässt sich ein **Vergleich von Ähnlichkeitsabbildungen** bezüglich ihres Streckungsfaktors definieren. Dies ermöglicht es dann, Figuren nach ihrer Größe zu vergleichen. Die hier angegebene Definition erfüllt aber nicht die Bedingung, dass aus `(t1 <= t2 && t2 <= t1) == True` folgt `(t1 == t2) == True`, d.h. die so definierte Ordnungsrelation ist nicht antisymmetrisch, weshalb man etwas vorsichtiger bei deren Verwendung sein sollte.

```
>   instance Ord AehnlichkeitsAbb where
>      f1 <= f2 = streckFaktor f1 <= streckFaktor f2
```

Um Basisobjekte in Textnotation darzustellen, was auf HUGS-Systemen ohne Graphikausgabe nötig ist, müssen Ähnlichkeitsabbildungen noch zu einer Instanz der Typklasse `Show` gemacht werden:

```
>   instance Show AehnlichkeitsAbb where
>     show abb = "[Aehnlichkeitsabbildung]"
```

Zum **späteren Zeichnen der Objekte**, z.B. eines Polygons, muss man dann nur noch die Ähnlichkeitsabbildung auf die Eckpunkte des Polygons anwenden und das Ergebnis zeichnen. Schwierig wird dies jedoch beim Kreis, den ich ja ohne Polygone implementiert habe. Da er genau in ein Quadrat passt, könnte man einfach die Abbildung auf Eckpunkte des Quadrates anwenden und dann die dahinein passende Ellipse zeichnen, da Ähnlichkeitsabbildungen winkeltreu sind und Kreise wieder in Kreise überführen. Ich habe mich in meiner Originallösung für die weniger elegante Variante entschieden, den Verschiebungsanteil der Ähnlichkeitsabbildung, der dem Bild von `(0,0)` entspricht, und ihren Streckungsfaktor über die folgende Funktion direkt herauszufiltern:

```
>   streckFaktor :: AehnlichkeitsAbb -> Double
>   streckFaktor abb = laenge (abb (1,0) `vminus` (dx,dy))
>     where
>         (dx,dy) = abb (0,0)
```

Dabei bezeichnet `vminus` die Vektorsubtraktion, die ich zusammen mit der Vektoraddition `vplus` und der Länge eines Vektors definiert habe:

```
>   vplus, vminus :: (Num a) => (a,a) -> (a,a) -> (a,a)
```

```
>   vplus  (x,y) (x2,y2) = (x+x2,y+y2)
>   vminus (x,y) (x2,y2) = (x-x2,y-y2)
>
>   laenge :: Punkt -> Double
>   laenge (x,y) = sqrt (fromInt (x^2 + y^2))
```

Die Vektoraddition und -subtraktion für zweidimensionale Vektoren sind hier polymorph im Typ `a` der Koordinaten. Die Typklassenannahme `(Num a) =>` besagt hier aber, dass er zur im `Prelude` vordefinierten Typklassen `Num` gehören muss, man ihn also wie eine Zahl behandeln kann. Insbesondere brauchen wir hier die Addition und die Subtraktion der Koordinaten.

Von der Korrektheit des Herausfilterns des Streckungsfaktors überzeugt man sich leicht durch Benutzen der oben angegebenen Charakterisierung von Ähnlichkeitsabbildungen. Auf das Zeichnen komme ich später noch einmal zurück.

Ich habe nun die **grundlegenden Ähnlichkeitsabbildungen** Streckung, Verschiebung, Spiegelung und Drehung bereit gestellt. Aus diesen werden dann alle anderen Ähnlichkeitsabbildungen durch Verkettung zusammengesetzt.

```
>   verschieben :: Punkt -> AehnlichkeitsAbb
>   verschieben = vplus
>
>   strecken :: Koordinate -> AehnlichkeitsAbb
>   strecken l (x,y) = (l*x, l*y)
>
>   spiegelnX, spiegelnY :: AehnlichkeitsAbb
>   spiegelnX (x,y) = (x,-y)  -- an x-Achse
>   spiegelnY (x,y) = (-x,y)  -- an y-Achse
>
>   drehen90, drehen180, drehen270 :: AehnlichkeitsAbb
>   drehen90  (x,y) = (y,-x)
>   drehen180 (x,y) = (-x,-y)
>   drehen270 (x,y) = (-y,x)
```

Die **Transformationen**, mit denen Figuren verändert werden können, sind wieder Abbildungen von Figuren auf Figuren.

```
>   type Transformation   = Figur -> Figur
```

Es sind im wesentlichen Spiegelung, Drehungen in 90°-Schritten, Entfernen des größten oder kleinsten Objektes aus der Figur sowie das Umwandeln von Formen in andere. Die ersten entsprechen den Ähnlichkeitsabbildungen, die ich mit folgender Funktion in Transformationen umwandeln kann:

```
>   liftAehnAbb :: AehnlichkeitsAbb -> Transformation
>   liftAehnAbb abb xs = map f xs
>     where
>         f (Obj{form = fo, lage = la}) = Obj{form = fo, lage = abb . la}
```

Diese Definition bedarf einiger Erläuterung. Zunächst ist `map` eine im `Prelude` definierte und häufig auftretende Funktion, die eine Funktion `f` auf jedes Element einer Liste, hier `xs`, anwendet. Die Anweisung `map f xs` ist so gleichbedeutend mit `[f x | x <- xs]`.

Die Funktion `f` nimmt hier als Parameter ein Basisobjekt. Das Pattern-Matching bei Datentypen mit benannten Feldern, wie hier dem `BasisObjekt`, funktioniert wie in dieser Definition angegeben, d.h. den Bezeichnern `fo` und `la` werden die Werte der Felder `form` und `lage` zugewiesen. Auf der rechten Seite der Definition von `f` wird dann ein neues Basisobjekt mit der Form `fo` und der der Ähnlichkeitsabbildung `abb` unterworfenen Lage `la` konstruiert.

Das Entfernen von Objekten habe ich folgendermaßen implementiert, wobei dem Vergleich von Objekten nach ihrer Größe hier jetzt seine besondere Rolle zukommt:

```
>   entfernen_klein, entfernen_gross :: Transformation
>   entfernen_klein []    = []
>   entfernen_klein fig   = tail (sort fig)
>   entfernen_gross []    = []
>   entfernen_gross fig   = init (sort fig)
```

Die `Prelude`-Funktion `sort` sortiert eine Liste aufsteigend, sofern auf den Elementen eine Ordnungsrelation definiert ist, sie also zur Typklass `Ord` gehören. Die ebenfalls im `Prelude`definierte Funktion `tail` liefert von einer Liste den Teil ohne das erste Element zurück, wirft also das erste Element fort. Die `Prelude`-Funktion `init` hingegen gibt von einer Liste den Teil ohne das letzte Element zurück.

Das Tauschen von Formen habe ich schließlich so implementiert:

```
>   formen_tauschen :: [Form] -> Figur -> Figur
>   formen_tauschen [kreis2,quadrat2,dreieck2,dach2] fs =
>     map (\b@(Obj{form = fo}) -> b{form = tausche fo}) fs
>     where
>         tausche form
>             | form == kreis   = kreis2
>             | form == quadrat = quadrat2
>             | form == dreieck = dreieck2
>             | form == dach    = dach2
>             | otherwise       = form   -- sollte nicht auftreten
```

Die dritte Zeile dieser Definition ist eine kompakte Formulierung ähnlich zur Definition von `liftAehnAbb`. Der erste Parameter von `map` muss, wie schon gesagt, eine Funktion sein, die hier keinen Namen besitzt, sondern über eine sogenannte "Lambda-Abstraktion" definiert wird. Alles zwischen `\` (soll ein Lambda λ darstellen) und dem Pfeil `->` definiert die Parameter der namenlosen Funktion, und alles auf der rechten Seite von `->` ist dann das Ergebnis der Funktion.

Der Parameter ist hier ein Basisobjekt von der Form `Obj{form = fo}`, dem Bezeichner `fo` wird also der Wert des Feldes `form` zugeordnet. Über das `@`-Zeichen wird der ganze Parameter noch dem Bezeichner `b` zugeordnet. Mit `b` können wir jetzt also auf den ganzen Parameter der namenlosen Funktion und mit `fo` auf seine Form zugreifen. Das `@`-Zeichen lässt sich auch bei Funktionen mit Namen zum selben Zweck anwenden.

Auf der rechten Seite von `->` steht ein "Update" der Felder von `b`: der Ausdruck `b{form = tausche fo}` liefert ein neues Basisobjekt, das sich vom ursprünglichen Basisobjekt `b` darin unterscheidet, dass das Feld `form` durch den Wert `tausche fo` ersetzt worden ist. Der Wert von `b` ändert sich dadurch aber nicht.

Insgesamt wird mit diesem Ausdruck also die Form eines jeden Basisobjektes in der Liste `fs` der Funktion `tausche` unterworfen.

Der Test, welche Form vorliegt, ist hier weniger effizient gestaltet, da im Falle eines Polygons die Listen der Punkte durchgegangen und verglichen werden müssen. Dies ist bei den gegebenen Datengrößen aber nicht weiter problematisch. Um den Zeitaufwand ggf.

weiter zu reduzieren, hätte man die Formen Quadrat, Dreieck und Dach wie den Kreis auch als Konstruktor definieren und ihnen erst beim Zeichnen ein konkretes Polygon zuordnen können.

Diese **Basistransformationen** habe ich nun wie die Basisobjekte in einer Liste zusammengefasst.

```
>   basis_transform :: [Transformation]
>   basis_transform =
>     map liftAehnAbb
>         [spiegelnX, spiegelnY, drehen90, drehen180, drehen270]
>     ++ [entfernen_klein, entfernen_gross,
>         formen_tauschen [quadrat, kreis, dreieck, dach],
>         formen_tauschen [kreis, quadrat, dach, dreieck],
>         formen_tauschen [kreis, dreieck, quadrat, dach]]
```

Hier werden einige Ähnlichkeitsabbildungen nun konkret mittels `liftAehnAbb` zu Transformationen verallgemeinert und mit anderen Transformationen in der Liste der Basistransformationen festgehalten. Wieder taucht `map` auf und die `Prelude`-Funktion `++` dient zum Zusammenhängen zweier Listen.

Ähnlichkeiten zwischen Figurenpaaren lassen sich aufbauend auf diesen Transformationen definieren. Wenn sich Figur X zur Figur Y irgendwie verhält, dann besagt das, dass es eine Transformation gibt, die X auf Y abbildet und die eine Verkettung der Basistransformationen ist. Zwei Figurenpaare (Figur A, Figur B) und (Figur C, Figur D) heißen nun ähnlich, wenn die Transformation von Figur A auf Figur B auch die Figur C auf die Figur D abbildet. Dies ist gerade der Inhalt des Satzes "Figur A verhält sich zu Figur B wie Figur C zu Figur D".

3.2.2 Teilaufgabe 2 - IQ-Test erzeugen

Bei der Erzeugung eines IQ-Tests habe ich mich mehr auf die zufällige Erzeugung konzentriert. Mein Haskell-System für meinen Apple Macintosh gab mir nicht die Möglichkeit, eine graphische Benutzeroberfläche zu gestalten, die den Benutzer durch die Erzeugung eines IQ-Test geleiten könnte, die Graphikausgabe in HUGS für Windows ist eine Notlösung. Da mein Programm für die Verwendung in einem interaktiven Haskell-Interpreter wie etwa Hugs gedacht ist, kann man sich natürlich aus den vordefinierten Funktionen leicht einen IQ-Test durch Zusammensetzen der Figuren und Transformationen "von Hand" oder durch die nun folgenden Funktionen zur zufälligen Erzeugung selbst zusammenbasteln. Dies ist zwar nicht sonderlich nutzerfreundlich, aber ich habe die zufällige Erzeugung eines ganzen IQ-Tests in eine einzige Funktion integriert, so dass der Nutzer ganz einfach vollständige IQ-Tests erzeugen kann.

Um das zufällige Erzeugen und auch das anschließende Lösen eines IQ-Tests gleich allgemein in den Griff zu kriegen, habe ich den Begriff der **Halbgruppe** (mit Einselement) eingeführt, den ich in Haskell als Typklasse implementiere:

```
>   infixr 7 °
>   class Halbgruppe a where
>     e   :: a                -- Einselement
>     (°) :: a -> a -> a      -- Multiplikation
```

Beim Instanziieren der Typklasse sollte ° assoziativ sein und die Eigenschaft `e°x=x°e=x` für alle Gruppenelemente `x` gelten.

Über die Festsetzung einer Präzedenz mittels `infixr 7 °` kann das Symbol ° nun auch als Name einer Funktion mit zwei Parametern verwendet werden. Dabei wird es, wie z.B. das Pluszeichen `+` auch, gleich zwischen die beiden Parameter geschrieben.

Die möglichen Figuren und Transformationen werden dann als Halbgruppe aufgefasst, die von den Elementen der jeweiligen Basislisten erzeugt wird. Das Einselement in der Halbgruppe der Figuren ist die leere Figur und die Verknüpfung ist gerade das Übereinanderlegen von Figuren. Die Transformationen bilden die gewohnte Abbildungshalbgruppe, wobei hier aber Kürzungsregeln auftreten, die in diesem allgemeinen Rahmen jedoch leider nicht berücksichtigt werden können.

```
>   instance Halbgruppe (a -> a) where    -- Abbildungshalbgruppe
>     e     = id
>     x ° y = x . y
>   instance Halbgruppe [a] where -- Wörter über dem Alphabet a
>     e     = []                  -- leeres Wort
>     x ° y = x ++ y              -- Konkatenation
```

Zum **zufälligen Erzeugen** einer Figur oder Transformation werden zufällig Elemente aus der gegebenen Basisliste gewählt und miteinander verknüpft. Dieses Vorgehen habe ich also gleich allgemein für Halbgruppen implementiert, so dass hier eben Spezialfälle davon verwendet werden.

```
>   zufalls_element :: Halbgruppe a => Int -> (Int, Int) -> [a] -> a
>   zufalls_element seed (low, upp) basis = foldr1 (°) randomElems
>     where
>         -- Zufallsgenerator erzeugen und gleich Zahl ziehen
>         (anzahl, g) = randomR (low, upp) (mkStdGen seed)
>         -- Zufallszahlen von 0 bis (Anzahl Basiselemente-1)
>         rlist       = randomRs (0,(length basis)-1) g
>         -- Die zu verknüpfenden Elemente aus der Basis wählen
>         randomElems = map (basis !!) (take anzahl rlist)
```

Die Funktionen zur Erzeugung von Zufallszahlen kommen aus der HASKELL-Bibliothek `Random`. Dabei erzeugt `mkStdGen` einen Zufallsgenerator und initialisiert in mit einem `seed`. Die Funktion `randomR` liefert nun eine Zufallszahl im Bereich von `low` bis `upp`, angegeben durch das geordnete Paar `(low, upp)`, und einen neuen Zufallsgenerator `g`, der beim nächsten Mal eine andere Zufallszahl ausgibt. Mittels `randomRs` kann dann gleich eine unendliche Liste von Zufallszahlen erzeugt werden. Auf diese weitere Stärke von HASKELLkomme ich aber später noch zu sprechen. Die `Prelude`-Funktion `take` nimmt nun die ersten `anzahl` Elemente dieser Liste.

Ein Aufruf des im `Prelude` definierten Operators `!!` der Form `xs !! i` wählt das `i`-te Element aus der Liste `xs` aus. Wenn wir nur die Liste, wie es in `(basis !!)` der Fall ist, als Parameter angeben, dann ist sozusagen der erste Teil der Typsignatur `[a] -> Int -> a` des Operators `!!` verbraucht und übrig bleibt `Int -> a`, so dass der Ausdruck `(basis !!)` also eine Funktion darstellt, die aus einem Parameter `i` gerade das `i`-te Element der Liste `basis` macht. Diese können wir dann an `map` geben, so dass wir mit den Zufallszahlen einige Elemente der Liste `basis` auswählen.

Die `Prelude`-Funktion `foldr1` gehört zu den wichtigen "Faltungs"-funktionen. Sie macht nichts weiter, als alle Elemente der gegebenen Liste `randomElems` über den °-Operator miteinander zu verknüpfen, oder anders ausgedrückt: `foldr1 (°) [a1,a2,a3,...] = a1°(a2°(a3°(...)))`. Durch die zufällige Zu-

sammenstellung und anschließende Verknüpfung verschiedener Basiselemente erhalten wir also ein zufälliges Element der Halbgruppe.

Durch meine Wahl der Basisobjekte sieht eine so entstandene Figur in den meisten Fällen auch relativ vernünftig aus. Alle diese zufällig ausgewählten Basiselemente sind aber nicht notwendigerweise verschieden, was bei Transformationen vielleicht reichhaltigere (eine Transformation mehrmals), bei Figuren eher triviale (doppelte Basisobjekte sehen natürlich aus wie ein einziges) Strukturen erzeugt. Kürzungsregeln bei den Transformationen können zur zufälligen Erzeugung der für einen IQ-Test wirklich langweiligen Identität führen. Ich habe entschieden, in dieser Hinsicht keine Vorkehrungen zu treffen, triviale IQ-Tests müssen vom Programmbenutzer ausgesiebt werden.

Durch diese Verallgemeinerung auf Halbgruppen wird der Quelltext zur zufälligen Erzeugung also wiederverwendet, was erst dank HASKELLs polymorphen Typsystem und seinem Typklassensystem möglich wird. Dieses wichtige Element moderner Softwarekonstruktion ist in anderen Sprachen wie C++ zwar realisiert, aber eben nur recht dürftig und umständlich. Funktionale Sprachen bieten hier einen wesentlich eleganteren und schnelleren Zugang.

Einen **IQ-Test** habe ich nun folgendermaßen definiert:

```
>   data IQTest = IQTest (Figur,Figur) Figur [Figur] deriving (Show)
```

Das erste Figurenpaar besteht aus den Figuren A und B, die nächste Figur ist Figur C und die folgende Liste von Figuren stellt die Auswahl der Figuren dar.

Man kann nun einen **IQ-Test** über die folgende Funktion **kreieren**:

```
>   iqTest :: Int -> Figur -> Figur -> IQTest
>   iqTest seed figur1 figur2 =
>     IQTest (figur1, transform figur1) figur2 figurS5
>     where
```

Die beiden als Parameter übergebenen Figuren entsprechen also den Figuren A und C, aber die maßgebliche Transformation für den IQ-Test von A nach B wird durch das zufällige Verketten von zwei Basistransformationen bestimmt:

```
>         transform =
>             zufalls_element seed (2,2) basis_transform
```

Um die vier weiteren Möglichkeiten für Figur D zu bekommen, werden einfach zufällige weitere Transformationen auf Figur D angewandt, bis vier weitere unterschiedliche Figuren erhalten worden sind, die der Figur D ziemlich ähnlich sehen sollten. Ich habe hier Figur D mit `figur2'` bezeichnet.

```
>         figur2' = transform figur2
>
>         -- 1..2 Transformationen für weitere Transformationen
>         zufallTrf seed  =
>             zufalls_element seed (1,2) basis_transform
>
>         -- die ersten 5 ungleichen Figuren auswählen,
>         -- wobei die erste stets die richtige Figur D ist
```

```
>         figurS5' = chooseNotEqBy (eqFigur) 5
>             ([figur2']++
>             [(t . transform) figur2 |
>                 d<-[1..], let t = zufallTrf (seed+d)])
```

Es sei hier noch bemerkt, dass die fünf Figuren aus einer unendlichen Liste nacheinander erzeugter zufälliger Figuren ausgewählt werden, auch eine Besonderheit von HASKELL. Der "Trick" ist, dass eine unendliche Liste nicht vollständig ausgewertet werden darf. Dies könnte allerdings doch passieren, wenn die Generierung der Figuren nicht in der Lage ist, fünf verschiedene Elemente zu erzeugen, dann würde das Programm in eine Endlosschleife geraten. Das ist allerdings bei der hier gegebenen genügend großen Anzahl von möglichen Transformationen und Basisobjekten auszuschließen.

Bei den in `figurS5'` abgelegten fünf Figuren ist die erste aber stets die richtige Figur D. Wir teilen die Liste also an einer zufällige Position namens `index` auf und legen dort die richtige Figur D hin.

```
>         (index, _) = randomR (0,4) (mkStdGen seed)
>         (figs1,figs2) = splitAt index (tail figurS5')
>         figurS5 = figs1 ++ (figur2':figs2)
```

Die `Prelude`-Funktion `splitAt` teilt eine Liste an der Position `index` in den Teil davor und den dahinter auf und gibt das Ergebnis als geordnetes Paar zurück.

Damit ist die Funktion zur Erstellung eines IQ-Tests mit zufälliger Transformation fertig.

Die oben verwendete Hilfsfunktion zum Auswählen verschiedener Elemente habe ich folgendermaßen implementiert:

```
>   chooseNotEqBy :: (a -> a -> Bool) -> Int -> [a] -> [a]
>   chooseNotEqBy eq n list
>     | n <= 0    = []
>     | otherwise = reverse (choose list [])
>     where
>         -- x `elementvon` xs <=> x ist in xs enthalten (bzgl. eq)
>         elementvon x xs = any (x `eq`) xs
>         choose (x:xs) cs
>             | length cs == n    = cs
>             | x `elementvon` cs = choose xs cs
>             | otherwise         = choose xs (x:cs)
```

Die `Prelude`-Funktion `reverse` kehrt eine Liste um und die ebenfalls im `Prelude` definierte Funktion `any` testet hier für die Liste `xs`, ob die Funktion `(x `eq`)` für irgend eines ihrer Elemente den Wert `True` zurückliefert, also ob eines gleich `x` ist.

Um dem Benutzer noch mehr Arbeit abzunehmen, und auch die **Figuren automatisch** zu **erzeugen**, habe ich die Funktion `iqTestRand` bereitgestellt, die die vollautomatische zufällige Generierung eines IQ-Tests bewirkt, gesteuert durch eine Initialisierung des Zufallsgenerators:

```
>   iqTestRand :: Int -> IQTest
>   iqTestRand seed = iqTest seed figur1 figur2
```

```
>      where
>         figur1 = nub (zufalls_element seed (3,3) basis_objekte)
>         figur2 = nub (zufalls_element (seed+1) (3,3) basis_objekte)
```

Dabei werden die Figuren aus je genau drei Basisobjekten erzeugt, was gute Ergebnisse liefert.

3.2.3 Teilaufgabe 3 - IQ-Test lösen

Um einen **IQ-Test** zu **lösen**, muss nun eine Transformation t mit einer bestimmten Eigenschaft gefunden werden, nämlich der, dass sie die Figur A in die Figur B überführt, und dass das Bild von Figur C unter t gerade in der Menge der Möglichkeiten für die Figur D liegt. Die gesuchte Figur D ist dann gerade t(Figur C), dies war ja die Definition der Ähnlichkeit zwischen Figurenpaaren. Es handelt sich also um das Suchen eines Elementes in einer von "Basiselementen" erzeugten Halbgruppe, hier der Transformationshalbgruppe, welches eine bestimmte Eigenschaft p erfüllen soll.

Diese Suche habe ich ganz allgemein als Breitensuche implementiert: es wird erst das Einselement auf p getestet, dann alle Wörter über dem Alphabet Σ der Basiselemente der Länge 1, dann alle der Länge 2 und so weiter. Um die Suche zu begrenzen, muss eine Suchtiefe angegeben werden, d.h. die maximale Länge der zu durchsuchenden Wörter. Die folgende Funktion führt diese Suche durch:

```
>   suche_element ::
>     Halbgruppe a => Int -> (a -> Bool) -> [a] -> Maybe a
>   suche_element tiefe p basis = suche_element_ tiefe p basis [e]
```

Die eigentliche Arbeit wird von der Funktion `suche_element_` erledigt, die in der übergebenen Liste bereits erzeugter Wörter nach solchen sucht, die die Eigenschaft p erfüllen und anschließend alle Wörter einer Länge größer durch Multiplikation von links mit den Basiselementen erzeugt. Diese werden ggf. weiter durchsucht.

```
>   suche_element_ ::
>     Halbgruppe a => Int -> (a -> Bool) -> [a] -> [a] -> Maybe a
>   suche_element_ tiefe p basis erzeugte =
>     case gefunden of
>         Nothing -> if tiefe == 0 then Nothing else
>                 suche_element_ (tiefe-1) p basis neu_erzeugte
>         _     -> gefunden
>
>     where
>         gefunden        = find p erzeugte
>         neu_erzeugte    = [b°el | el <- erzeugte, b <- basis]
```

Die eben definierte Funktion `suche_element_` liefert das gesuchte Element als `Maybe` "verpackt" zurück. Der `Maybe`-Typ ist im `Prelude` definiert und dient dazu, Situationen zu meistern, in denen es etwas nicht geben könnte oder nicht gefunden wurde, ausgedrückt durch den `Maybe`-Wert `Nothing`. Wenn aber ein vorhandener Wert `wert` als `Maybe` verpackt werden soll, wird dies über `Just wert` realisiert. Unsere Funktion liefert also `Nothing` zurück, wenn kein Element mit der Eigenschaft `p` gefunden wurde, und sie liefert im Erfolgsfalle `Just element` zurück, wobei der Wert `element` die Eigenschaft `p` erfüllt.

Die Funktion `find` stammt aus der Bibliothek `List` und liefert hier ein Element aus der Liste `erzeugte`, welches die Eigenschaft `p` erfüllt, ebenfalls als `Maybe` verpackt.

Mit Hilfe dieser Funktion kann nun das Lösen eines gegebenen IQ-Tests implementiert werden. Die folgende Funktion `solve_iqTest` gibt die Position der richtigen Figur D in der Auswahlliste an:

```
>   solve_iqTest :: IQTest -> Int
>   solve_iqTest (IQTest (figurA, figurB) figurC auswahl) =
>     case figAzuB of
>         Nothing     -> error "IQ-Test ist unloesbar"
>         Just trf    ->
>             let
>                 figD        = trf figurC
>                 -- Stelle in der Liste finden
>                 Just idx    = findIndex
>                     (\fig-> fig `eqFigur` figD) auswahl
>             in idx+1
>     where
```

Die Funktion `findIndex` stammt ebenfalls aus der Bibliothek `List` und liefert hier nicht ein Element aus einer Liste, sondern dessen Position in der Liste zurück, wieder als `Maybe` verpackt.

Die gesucht Transformation von Figur A zu Figur B wird also in `figAzuB` hinterlegt. Sie wird mit einer Suchtiefe von 5 gesucht, das dürfte auf Grund von Kürzungsregeln vernünftig sein, zumal der selbst generierte IQ-Test nur 2 Transformationen verwendet.

```
>         figAzuB = suche_element 5
>                 isValidTransformation basis_transform
```

Die Funktion gibt gerade an, ob eine Transformation die gesuchte ist, d.h. ob sie Figur A in Figur B überführt und ihre Anwendung auf Figur C in der Liste der Möglichkeiten für Figur D liegt.

```
>         isValidTransformation trf =
>                 figurB `eqFigur` trf figurA &&
>                 (figD `elementVon` auswahl)
>             where
>                 figD = trf figurC
>                 elementVon fig fs = any (fig `eqFigur`) fs
```

Dies genügt, um einen IQ-Test zu lösen.

Allerdings kann es durchaus mehrere Transformationen t geben, die die genannten Bedingungen erfüllen, und die nicht die gleiche Figur D liefern, d.h. der IQ-Test wäre mehrdeutig. In meiner Orginiallösung habe ich dies nicht weiter behandelt, da es ja genügt, mindestens eine Lösung für einen IQ-Test zu finden, um ihn zu bestehen.

Dennoch wäre es bei der Erstellung von IQ-Tests für Gerda Bauch vielleicht wünschenswert zu erfahren, ob der gegebene IQ-Test mehrdeutig ist. Dazu ist es zweckmäßig, nach jeder IQ-Test Erzeugung den Test auch gleich mit der eben beschriebenen Funktion zu prüfen, die nun auf Mehrdeutigkeit erweitert werden müsste. Diese könnte

man entscheiden, indem man die gefundenen Transformationen nach ihrer Kompliziertheit anordnen, d.h. der Anzahl der Basistransformationen, aus denen die Transformation zusammengesetzt ist. Wenn es eine einfache, aber verschiedene wesentlich kompliziertere Transformationen gibt, dann kann der IQ-Test als eindeutig gelten, da man davon ausgehen kann, dass der Proband stets die einfachste Transformation zuerst entdeckt. Trotzdem könnte das Programm den Benutzer vor dieser Situation warnen. Wenn es aber mehrere einfache Transformationen gibt, dann sollte das Programm ausgeben, dass der IQ-Test mehrdeutig ist und warnt Gerda Bauch somit vor einer Verwendung. Ein neuer lässt sich ja schnell kreieren.

3.2.4 Weiteres - IQ-Test zeichnen

Zum Schluss kommen wir noch zum **Zeichnen eines IQ-Testes**, damit er von Gerda Bauch auch tatsächlich eingesetzt werden kann. Ich habe hierbei wie gesagt die `Graphics`-Bibliothek von Hugs für Windows verwendet.

Wie schon in Teilaufgabe 1 angegeben, muss man zum Zeichnen eines Polygons nur die Ähnlichkeitsabbildung auf alle Eckpunkte anwenden. Die Koordinaten und Größe eines Kreises habe ich jedoch "von Hand" bestimmt. Die folgende Funktion `draw_basis_objekt` zeichnet ein Basisobjekt in ein Quadrat der linken oberen Ecke `(dx,dy)`, welches die Seitenlänge `2*offset` besitzt:

```
>   offset = 70 :: Int
>
>   draw_basis_objekt :: (Int, Int) -> BasisObjekt -> Graphic
>   draw_basis_objekt (dx,dy) (Obj {form=form, lage=lage}) = draw form
>     where
>         toScreen p = p `vplus` (offset+dx, offset+dy)
>         g          = round $ 10*(streckFaktor lage)
>
>         draw Kreis = regionToGraphic $ subtractRegion
>                 (ellipseRegion (x-g,y-g) (x+g,y+g))
>                 (ellipseRegion (x-g+1,y-g+1) (x+g-1,y+g-1))
>             where
>                 (x,y) = toScreen (lage (0,0))
>
>         draw (Polygon (p:ps)) = polyline $
>                         map (positioniere) (p:ps++[p])
>             where
>                 positioniere (x,y) = toScreen (lage (x,y))
```

Eine Figur kann nun mit Hilfe seiner Basisobjekte und mit einem Rahmen gezeichnet werden:

```
>   draw_figur :: (Int, Int) -> Figur -> Graphic
>   draw_figur (dx,dy) fig = overGraphics $
>     map (draw_basis_objekt (dx,dy)) fig
>     ++ [rechteck (dx,dy) (dx+2*offset,dy+2*offset)]
>
>   rechteck :: (Int, Int) -> (Int, Int) -> Graphic
```

```
>   rechteck (x1,y1) (x2,y2) =
>     polyline [(x1,y1),(x2,y1),(x2,y2),(x1,y2),(x1,y1)]
```

Ein ganzer IQ-Test wird nun noch schön in ein Fenster gezeichnet, das der Benutzer durch Klicken einer beliebigen Taste wieder wegbekommt:

```
>   draw_iqTest :: IQTest -> IO ()
>   draw_iqTest (IQTest (figurA, figurB) figurC auswahl) =
>     putStrLn
>      "Oben : Figur A, B; Figur C \n Unten: Auswahl fuer Figur D" >>
>     runGraphics(
>         do  w <- openWindow "" (700,280)
>             drawInWindow w fs_graph
>             getKey w
>             return ()
>             closeWindow w
>     )
>     where
>         fss = [[figurA,figurB,[],figurC],auswahl]
>         fs_graph = overGraphics
>             [draw_figur (2*offset*i2, 2*offset*i1) f |
>                     (fs,i1) <- (zip fss [0..]),
>                     (f,i2) <- (zip fs [0..])]
```

Die Lösung kann gleich mit ausgegeben werden:

```
>   show_iqTest :: IQTest -> IO ()
>   show_iqTest test =
>     do
>         draw_iqTest test
>         putStr ("Loesung = Figur No " ++ show (solve_iqTest test))
```

3.2.5 Programmbeispiele

Die Abbildungen 1,2 und 3 zeigen drei durch mein Programm erzeugte IQ-Tests. Die Lösungen zu diesen Beispielen sind die Figuren 5,5 und 1. Ein IQ-Test wird am besten über `show_iqTest $ iqTestRand seed` erzeugt und gleich ausgegeben, wobei `seed` eine ganze Zahl für die Initialisierung des Zufallsgenerators ist. Ein Beispiel:

```
? show_iqTest $ iqTestRand 23409

Oben : Figur A, B; Figur C
Unten: Auswahl fuer Figur D
```

Anschließend erscheint das Fenster, in dem die Figuren dargestellt sind. Die beiden Figuren links oben sind die Figuren A und B, die rechts oben ist Figur C und die Figuren in der zweiten Zeile sind die Möglichkeiten für Figur D. Nachdem das Fenster durch den Druck einer beliebigen Taste geschlossen wurde, wird die vom Computer berechnete Lösung des IQ-Tests ausgegeben: `Loesung = Figur No 5`.

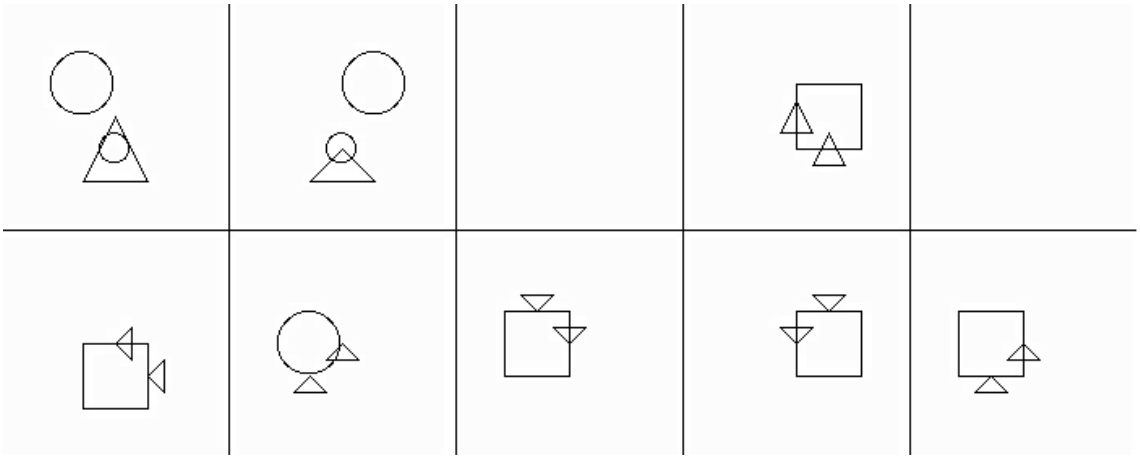


Abbildung 1: Test 71891, besonders ästhetisch

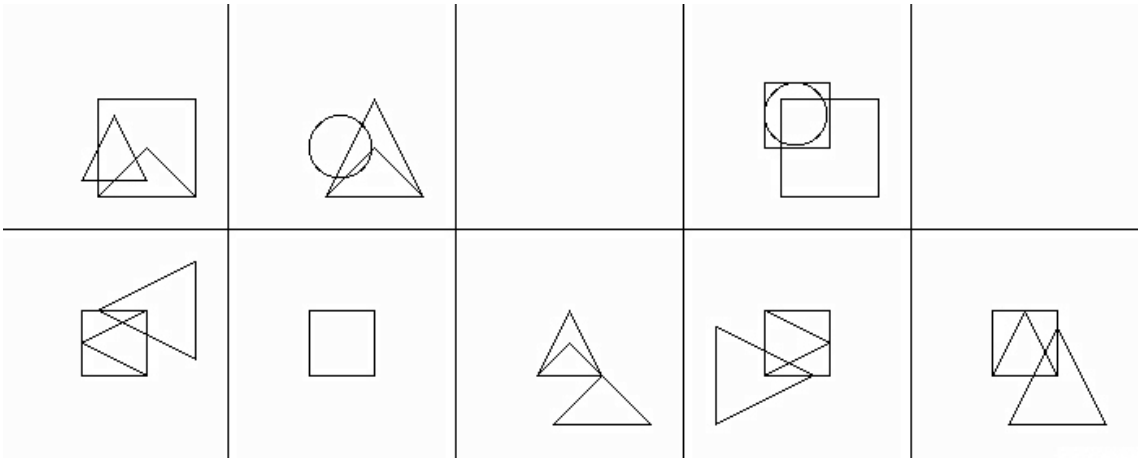


Abbildung 2: Test 23409, interessant

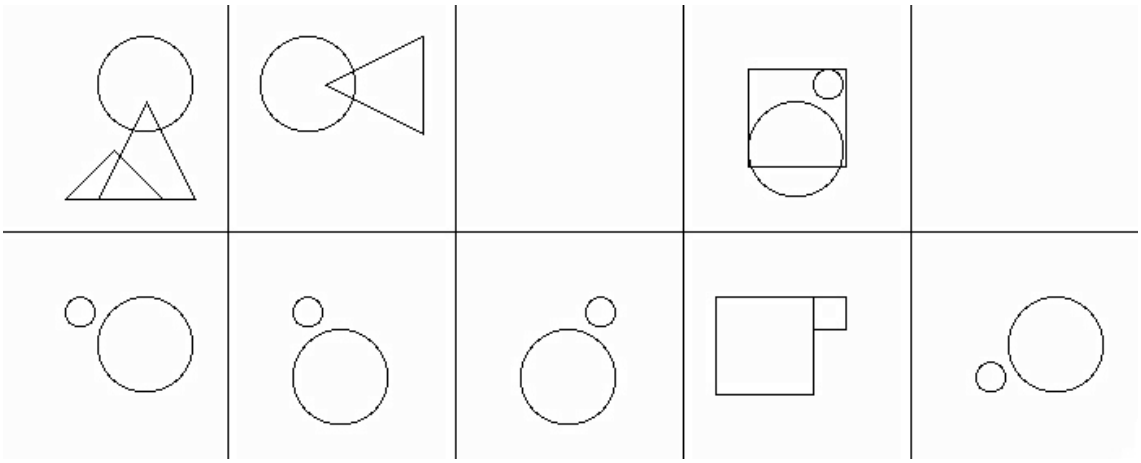


Abbildung 3: Test 7654102, ziemlich schwer

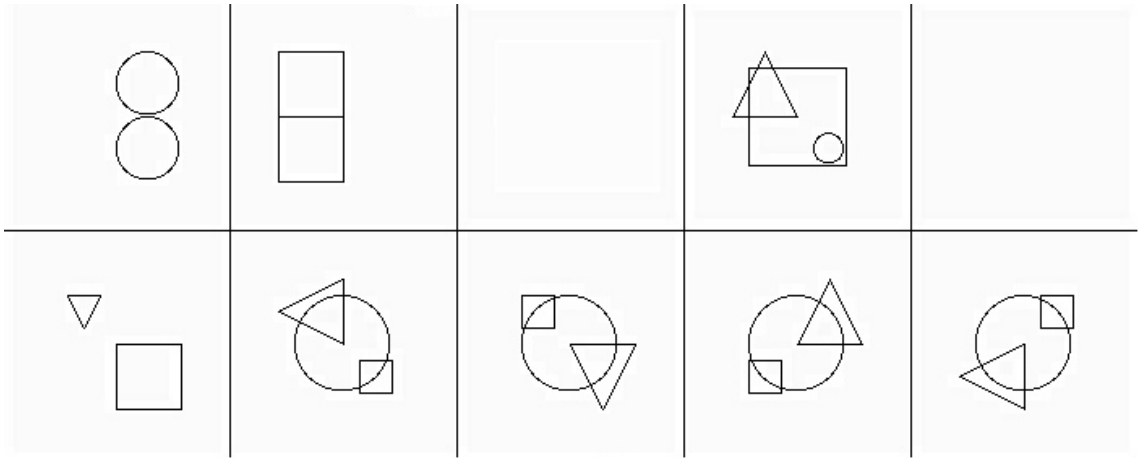


Abbildung 4: Test 8766, nur zwei Basisobjekte

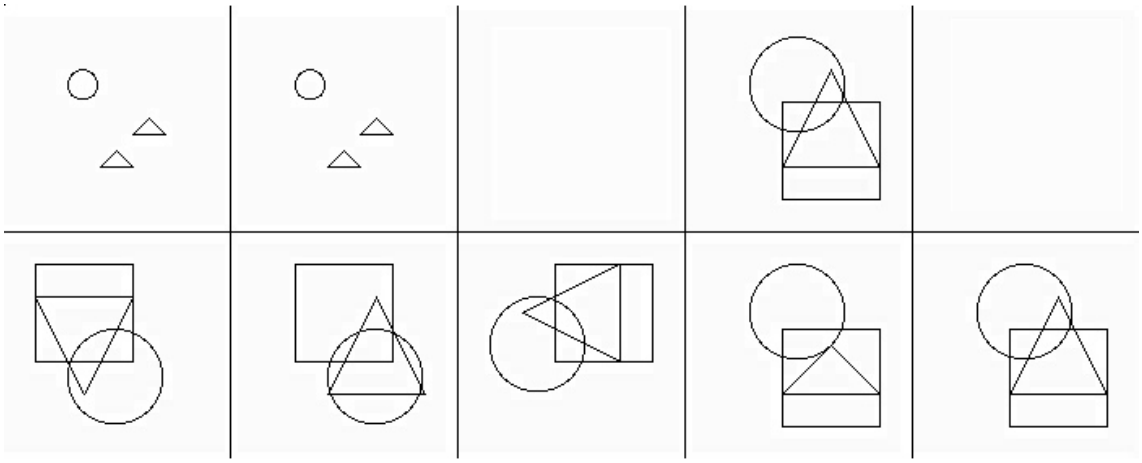


Abbildung 5: Test 73026, trivial

Ich habe davon gesprochen, dass langweilige IQ-Tests entstehen können. Die Abbildungen 4 und 5 zeigen zwei Beispiele: Im Test 8766 sind nur zwei Basisobjekte in den Figuren A und B zu unterscheiden und im Test 73026 finden wir die Identität als Transformation zwischen der Figur A und Figur B.

4 Schlusswort

In dieser Arbeit ging es mir darum zu zeigen, dass das Programmieren in Haskell Ausdrucksmöglichkeiten eröffnet, die die Softwareentwicklung wesentlich vereinfachen sowie auf die inhaltlichen Konzepte konzentrieren, und die nicht in herkömmlichen Programmiersprachen zu finden sind. Häufig stellt eine Verallgemeinerung, wie hier in Form der Halbgruppen, den inhaltlichen Kernpunkt heraus und reduziert zusammengehörende Probleme auf ein einziges. Auch dies trägt neben der kompakten Formulierung zur drastischen Verkürzung des Quellcodes bei. Für mich bedeutete dies also weniger Schreibarbeit und dafür mehr Denkarbeit. Das hat mir wohl auch geholfen, hervorragend im Bundeswettbewerb Informatik abzuschneiden.

Mit der Fertigstellung dieser Arbeit ist mein Interesse an Haskell natürlich erst recht geweckt worden. Ich möchte unterdessen gerne mehr darüber wissen, wie funktionale Programme in Maschinensprache übersetzt werden. Damit ist auch die Frage verbunden, wie man in Haskell besonders effiziente Algorithmen und Datenstrukturen implementieren kann. So ist es auf den ersten Blick z.B. nicht ersichtlich, wie man denn die eigentlich ganz einfache Datenstruktur der Queue in Haskell realisieren würde.

5 Quellenverzeichnis

[BW92] Richard Bird und Philip Wadler. *Einführung in die funktionale Programmierung.* –Coed.– München, Wien: Hanser; London: Prentice-Hall Internat, 1992.

[BWINF20] Website des 20. Bundeswettbewerbs Informatik. `http://www.bwinf.de/archiv/bwi20/bwinf20.php`

[Erw99] Martin Erwig. *Grundlagen funktionaler Programmierung.* München, Wien: Oldenbourg, 1999.

[HasWeb] Offizielle Haskell-Website. `http://www.haskell.org`

[HugsWeb] Website des Haskell-Interpreters Hugs. `http://www.haskell.org/hugs`

[Haskell98] Simon Peyton Jones et al. *Haskell 98: A Non-strict, Purely Functional Language.* February 1999. `http://haskell.org/report`.

[Jon01] Simon Peyton Jones. *Tackling the Awkward Squad: monadic input/output, concurrency, exceptions and foreign-language calls in Haskell.* June 2001. `http://research.microsoft.com/users/simonpj`.

[Oka99] Chirs Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[Pep99] Peter Pepper. *Funktionale Programmierung in Opal, ML, Haskell und Gofer*. Berlin: Springer, 1999.

6 Eigenständigkeitserklärung

Ich erkläre, dass ich diese Arbeit selbständig angefertigt und alle benutzten Quellen und Hilfen angegeben habe.