2

Los geht's: erste Graphen mit D3

Zu Anfang wollen wir einige Beispiele durcharbeiten, die die Möglichkeiten von D3 demonstrieren und Sie in die Lage versetzen, selbst Aufgaben aus der Praxis zu lösen – und sei es auch nur, indem Sie diese Beispiele entsprechend anpassen. Die ersten beiden Beispiele in diesem Kapitel zeigen, wie Sie aus Datendateien die gebräuchlichen *Streu-* und *XY-Diagramme* erstellen, und zwar komplett mit Achsen. Die Diagramme sehen nicht sehr hübsch aus, erfüllen aber ihren Zweck, und es ist auf einfache Weise möglich, sie zu verschönern und die Prinzipien auf andere Datenmengen zu übertragen. Das dritte Beispiel ist nicht vollständig, sondern dient mehr der Veranschaulichung, um Ihnen zu zeigen, wie einfach es ist, Ereignisbehandlung und Animationen in Ihre Dokumente aufzunehmen.

Erstes Beispiel: eine einzige Datenmenge

Um D3 kennenzulernen, betrachten wir die kleine, einfache Datenmenge aus Beispiel 2–1. Bei der grafischen Ausgabe dieser Daten mithilfe von D3 kommen wir schon mit vielen der Grundprinzipien in Berührung.

Beispiel 2–1: Eine einfache Datenmenge (examples-simple.tsv)

```
x       y1
100     50
200     100
300     150
400     200
500     250
```

Wie bereits in Kapitel 1 erwähnt, ist D3 eine JavaScript-Bibliothek zur Bearbeitung des DOM-Baums, um Informationen grafisch darzustellen. Das deutete schon darauf hin, dass jeder D3-Graph über mindestens *zwei* oder *drei* »bewegliche Teile« verfügt:

- Eine HTML-Datei mit dem DOM-Baum
- Eine JavaScript-Datei oder ein Skript mit den Befehlen zur Bearbeitung des DOM-Baums
- Häufig eine Datei oder eine andere Ressource mit der Datenmenge

Beispiel 2–2 zeigt die gesamte HTML-Datei.

Beispiel 2–2: Eine HTML-Datei, die ein SVG-Element definiert

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script src="d3.js"></script>    ❶
  <script src="examples-demo1.js"></script>    ❷
</head>

<body onload="makeDemo1()">    ❸
  <svg id="demo1" width="600" height="300"
       style="background: lightgrey" />    ❹
</body>
</html>
```

Ja, Sie sehen richtig – die HMTL-Datei ist im Grunde genommen leer! Alle Aktionen finden im JavaScript-Code statt. Sehen wir uns kurz die wenigen Dinge an, die in dem HTML-Dokument geschehen:

❶ Als Erstes lädt das Dokument *d3.js*, also die Bibliothek D3.

❷ Danach lädt das Dokument unsere JavaScript-Datei. Sie enthält alle Befehle zur Definition des Graphen, den wir anzeigen wollen.

❸ Das `<body>`-Tag definiert einen `onload`-Ereignishandler, der ausgelöst wird, wenn der Browser das `<body>`-Element komplett geladen hat. Die Ereignishand-

lerfunktion `makeDemo1()` ist in unserer JavaScript-Datei *examples-demo1.js* definiert.[1]

❹ Am Ende enthält das Dokument ein SVG-Element mit einer Größe von 600 × 300 Pixeln. Es hat einen hellgrauen Hintergrund, damit Sie es erkennen können, ist sonst aber leer.

Der dritte und letzte Bestandteil ist die JavaScript-Datei, die Sie in Beispiel 2–3 sehen.

Beispiel 2–3: Befehle für Abbildung 2–1

```
function makeDemo1() {    ❶
    d3.tsv( "examples-simple.tsv" )    ❷
        .then( function( data ) {    ❸ ❹
            d3.select( "svg" )    ❺
                .selectAll( "circle" )    ❻
                .data( data )    ❼
                .enter()    ❽
                .append( "circle" )    ❾
                .attr( "r", 5 ).attr( "fill", "red" )    ❿
                .attr( "cx", function(d) { return d["x"] } )    ⓫
                .attr( "cy", function(d) { return d["y"] } );
        } );
}
```

Wenn Sie diese drei Dateien (die Datendatei, die HTML-Seite und die JavaScript-Datei) zusammen mit der Bibliotheksdatei *d3.js* in ein gemeinsames Verzeichnis stellen und die Seite im Browser laden, sollte der Browser den Graphen aus Abbildung 2–1[2] anzeigen.

Sehen wir uns nun nacheinander die einzelnen JavaScript-Befehle an:

❶ Das Skript definiert nur eine einzige Funktion, nämlich den Callback `makeDemo1()`, der aufgerufen wird, wenn die HTML-Seite vollständig geladen ist.

❷ Die Funktion lädt die Datendatei mithilfe der `Fetch`-Funktion `tsv()` (sie »ruft sie ab«, daher der Bezug zu »to fetch«). In D3 sind mehrere Funktionen definiert, um Dateiformate mit Trennzeichen zu lesen. `tsv()` ist für tabulatorgetrennte Dateien bestimmt.

---

1 Die Definition eines Ereignishandlers über das `onload`-Tag wird oft missbilligt, weil dadurch JavaScript-Code in HTML eingebettet wird. Moderne Alternativen finden Sie in Anhang A und C.

2 Sie sollten in der Lage sein, die Seite und die zugehörige JavaScript-Datei zu laden, indem Sie dem Browser das lokale Verzeichnis als Zieladresse angeben. Allerdings kann es sein, dass der Browser sich weigert, die Datendatei auf diese Weise zu laden. Daher ist es gewöhnlich notwendig, bei der Arbeit mit D3 einen Webserver auszuführen. Mehr darüber erfahren Sie in Anhang A.

❸ Wie alle Funktionen aus der JavaScript-API `Fetch` gibt auch `tsv()` ein `Promise`-Objekt zurück. Ein solches Objekt enthält ein Resultset und einen Callback und ruft Letzteren auf, wenn das Resultset vollständig und zur Verarbeitung bereit ist. Anschließend stellt das `Promise`-Objekt die Funktion `then()` bereit, um den gewünschten Callback zu registrieren. (Mehr über JavaScript-Promises erfahren Sie im Abschnitt »*Promises in JavaScript*« auf *S. 120*.)

❹ Der Callback, der nach dem Laden der Datei aufgerufen werden soll, wird als anonyme Funktion definiert, die den Inhalt der Datendatei als Argument erhält. Die Funktion `tsv()` gibt den Inhalt der tabulatorgetrennten Datei als Array von JavaScript-Objekten zurück. Jede Zeile in der Datei ergibt ein Objekt, wobei die Eigenschaftennamen in der Kopfzeile der Eingabedatei definiert sind.

❺ Wir wählen das `<svg>`-Element als die Position im DOM-Baum, die den Graphen enthalten soll. Die Funktionen `select()` und `selectAll()` nehmen einen CSS-Selektorstring entgegen (siehe »*CSS-Selektoren*« auf *S. 38* in Kapitel 3) und geben die übereinstimmenden Knoten zurück – `select()` nur die erste Übereinstimmung, `selectAll()` eine Sammlung aller Übereinstimmungen.

❻ Als Nächstes wählen wir *alle* `<circle>`-Elemente innerhalb des `<svg>`-Knotens aus. Das mag absurd erscheinen, denn schließlich gibt es darin gar keine `<circle>`-Elemente! Dieser merkwürdige Aufruf stellt jedoch kein Problem dar, da `selectAll("circle")` lediglich eine *leere* Sammlung (von `<circle>`-Elementen) zurückgibt, sondern erfüllt sogar eine wichtige Funktion, da er mit dieser leeren Sammlung einen *Platzhalter* erstellt, den wir anschließend füllen werden. Das ist ein gängiges D3-Idiom, um Graphen mit neuen Elementen zu versehen.

❼ Als Nächstes verknüpfen wir die Sammlung der `<circle>`-Elemente mit der Datenmenge. Dazu verwenden wir den Aufruf `data(data)`. Es ist wichtig, zu erkennen, dass die beiden Sammlungen (DOM-Elemente auf der einen Seite und Datenpunkte auf der anderen) nicht im Ganzen miteinander verbunden werden. Stattdessen versucht D3 eine 1:1-Beziehung zwischen den DOM-Elementen und den Datenpunkten herzustellen: *Jeder Datenpunkt wird durch ein eigenes DOM-Element* dargestellt, das wiederum seine Eigenschaften (wie Position, Farbe und Erscheinungsbild) aus den Informationen über den Datenpunkt bezieht. Der Aufbau und die Pflege dieser 1:1-Beziehungen zwischen einzelnen Datenpunkten und den zugehörigen DOM-Elementen ist ein grundlegendes Merkmal von D3. (In Kapitel 3 werden wir uns diesen Vorgang noch ausführlicher ansehen.)

❽ Die Funktion `data()` gibt die Sammlung der Elemente zurück, die mit den einzelnen Datenpunkten verknüpft wurden. Zurzeit kann D3 die einzelnen Datenpunkte nicht mit `<circle>`-Elementen verknüpfen, da es (noch) keine

gibt. Die zurückgegebene Sammlung ist daher leer. Allerdings bietet D3 mithilfe der (merkwürdig benannten) Funktion `enter()` auch Zugriff auf alle restlichen Datenpunkte, für die keine übereinstimmenden DOM-Elemente gefunden werden konnten. Die übrigen Befehle werden für die einzelnen Elemente in dieser »Restsammlung« aufgerufen.

❾ Als Erstes wird der Sammlung der `<circle>`-Elemente innerhalb des in Zeile 6 ausgewählten SVG-Elements ein `<circle>`-Element angehängt.

❿ Anschließend werden einige fixe (also nicht datenabhängige) Attribute und Formatierungen festgelegt, nämlich der Radius (das Attribut `r`) und die Füllfarbe.

⓫ Schließlich wird die Position der einzelnen Kreise aufgrund des Werts bestimmt, den der zugehörige Datenpunkt hat. Die Attribute `cx` und `cy` der einzelnen `<circle>`-Elemente werden jeweils auf der Grundlage der Einträge in der Datendatei festgelegt. Statt eines festen Werts stellen wir *Zugriffsfunktionen* bereit, die zu einem gegebenen Eintrag der Datendatei (also zu einem einzeiligen Datensatz) den entsprechenden Wert zurückgeben.

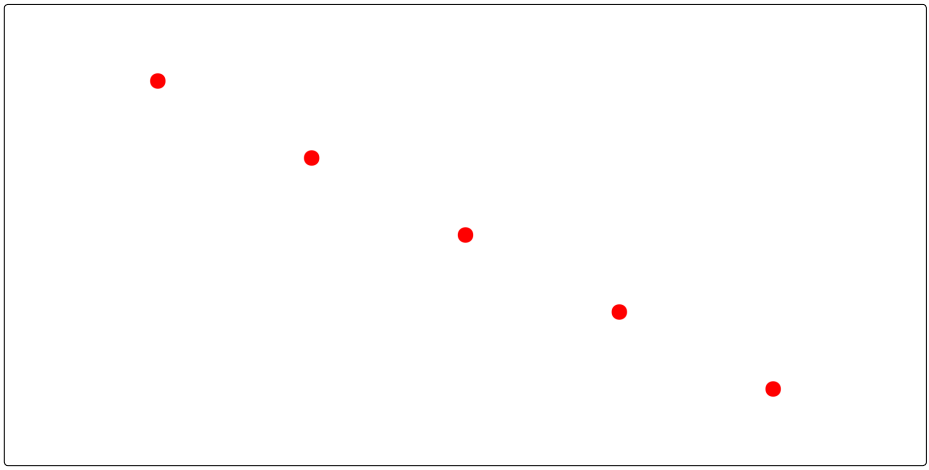

Abb. 2–1 *Grafische Darstellung einer einfachen Datenmenge (siehe Beispiel 2–3)*

Um ehrlich zu sein: Für einen so einfachen Graphen ist das ein erheblicher Aufwand! Hieran können Sie schon etwas erkennen, was im Laufe der Zeit noch deutlicher wird: D3 ist keine Grafikbibliothek und erst recht kein Werkzeug zur Diagrammerstellung, sondern eine Bibliothek, um den DOM-Baum zu bearbeiten und dadurch Informationen grafisch darzustellen. Sie werden die ganze Zeit damit beschäftigt sein, Operationen an Teilen des DOM-Baums vorzunehmen (die Sie mit Selections auswählen). Außerdem werden Sie feststellen, dass bei D3 nicht

gerade wenig Tipparbeit anfällt, da Sie Attributwerte bearbeiten und eine Zugriffsfunktion nach der anderen schreiben müssen. Andererseits ist der Code meiner Meinung nach zwar sehr wortreich, aber auch sauber und unkompliziert.

Wenn Sie ein wenig mit den Beispielen herumspielen, werden Sie womöglich noch einige weitere Überraschungen erleben. Beispielsweise ist die Funktion `tsv()` ziemlich wählerisch: Spalten *müssen* durch Tabulatoren getrennt sein, Weißraum wird *nicht* ignoriert, eine Kopfzeile *muss* vorhanden sein usw. Bei genauerer Untersuchung der Datenmenge und des Graphen werden Sie schließlich auch feststellen, dass das Diagramm nicht korrekt ist, sondern auf dem Kopf steht! Das liegt daran, dass SVG »grafische Koordinaten« verwendet, bei denen die horizontale Achse zwar wie üblich von links nach rechts, die vertikale aber von oben nach unten läuft.

Um diese ersten Eindrücke besser einordnen zu können, fahren wir mit einem zweiten Beispiel fort.

Zweites Beispiel: zwei Datenmengen

Für unser zweites Beispiel verwenden wir die Datenmenge aus Beispiel 2–4. Sie sieht zwar fast genauso harmlos aus wie die vorherige, aber bei genauerer Betrachtung werden einige zusätzliche Schwierigkeiten deutlich. Sie enthält nämlich nicht nur *zwei* Datenmengen (in den Spalten `y1` und `y2`), sondern umfasst auch Datenbereiche, die unserer Aufmerksamkeit bedürfen. Im vorherigen Beispiel konnten wir die Datenwerte unmittelbar als Pixelkoordinaten verwenden, doch die Werte der neuen Datenmenge erfordern eine Transformation, bevor sie als Bildschirmkoordinaten genutzt werden können. Wir müssen also etwas mehr Arbeit aufwenden.

Beispiel 2–4: Eine kompliziertere Datenmenge (examples-multiple.tsv)

```
x       y1      y2
1.0     0.001   0.63
3.0     0.003   0.84
4.0     0.024   0.56
4.5     0.054   0.22
4.6     0.062   0.15
5.0     0.100   0.08
6.0     0.176   0.20
8.0     0.198   0.71
9.0     0.199   0.65
```

Symbole und Linien darstellen

Die HTML-Seite für dieses Beispiel ist im Großen und Ganzen identisch mit derjenigen, die wir zuvor benutzt haben (Beispiel 2–2). Allerdings müssen wir die folgende Zeile ersetzen:

```
<script src="examples-demo1.js"></script>
```

Für dieses Beispiel muss sie wie folgt lauten, damit sie auf das neue Skript verweist:

```
<script src="examples-demo2.js"></script>
```

Auch der `onload`-Ereignishandler muss den neuen Funktionsnamen angeben:

```
<body onload="makeDemo2()">
```

Das Skript sehen Sie in Beispiel 2–5, das resultierende Diagramm in Abbildung 2–2.

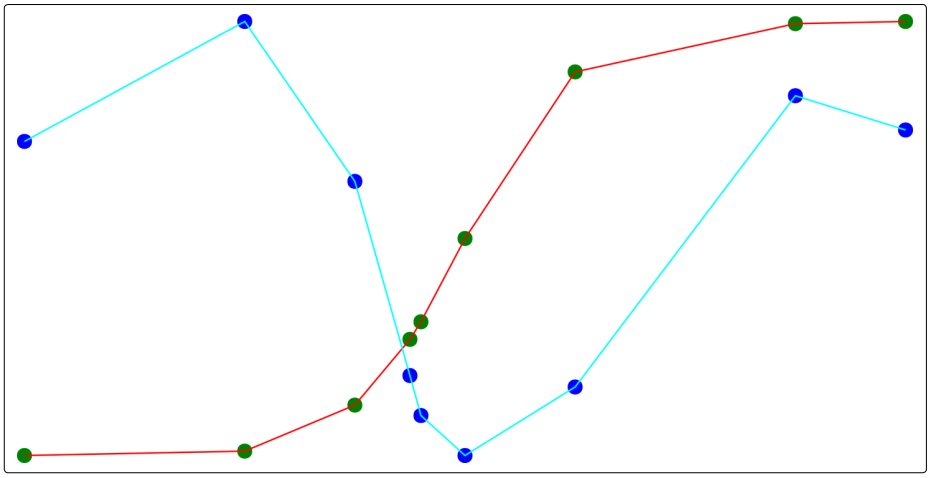

Abb. 2–2 *Einfaches Diagramm der Datenmenge aus Beispiel 2–4 (siehe auch Beispiel 2–5)*

Beispiel 2–5: Befehle für Abbildung 2–2

```
function makeDemo2() {
    d3.tsv( "examples-multiple.tsv" )
        .then( function( data ) {
            var pxX = 600, pxY = 300;    ❶

            var scX = d3.scaleLinear()    ❷
                .domain( d3.extent(data, d => d["x"] ) )    ❸
                .range( [0, pxX] );
```

```
            var scY1 = d3.scaleLinear()   ❹
                .domain(d3.extent(data, d => d["y1"] ) )
                .range( [pxY, 0] );   ❺
            var scY2 = d3.scaleLinear()
                .domain( d3.extent(data, d => d["y2"] ) )
                .range( [pxY, 0] );

            d3.select( "svg" )   ❻
                .append( "g" ).attr( "id", "ds1" )   ❼
                .selectAll( "circle" )   ❽
                .data(data).enter().append("circle")
                .attr( "r", 5 ).attr( "fill", "green" )   ❾
                .attr( "cx", d => scX(d["x"]) )   ❿
                .attr( "cy", d => scY1(d["y1"]) );   ⓫

            d3.select( "svg" )   ⓬
                .append( "g" ).attr( "id", "ds2" )
                .attr( "fill", "blue" )   ⓭
                .selectAll( "circle" )   ⓮
                .data(data).enter().append("circle")
                .attr( "r", 5 )
                .attr( "cx", d => scX(d["x"]) )
                .attr( "cy", d => scY2(d["y2"]) );   ⓯

            var lineMaker = d3.line()   ⓰
                .x( d => scX( d["x"] ) )   ⓱
                .y( d => scY1( d["y1"] ) );

            d3.select( "#ds1" )   ⓲
                .append( "path" )   ⓳
                .attr( "fill", "none" ).attr( "stroke", "red" )
                .attr( "d", lineMaker(data) );   ⓴

            lineMaker.y( d => scY2( d["y2"] ) );   ㉑

            d3.select( "#ds2" )   ㉒
                .append( "path" )
                .attr( "fill", "none" ).attr( "stroke", "cyan" )
                .attr( "d", lineMaker(data) );

            // d3.select( "#ds2" ).attr( "fill", "red" );   ㉓
        } );
}
```

❶ Wir weisen die Abmessungen des eingebetteten SVG-Bereichs Variablen zu (`px` für Pixel), um später darauf verweisen zu können. Es ist natürlich auch möglich, die Größenangabe aus dem HTML-Dokument herauszunehmen und sie stattdessen über JavaScript festzulegen. (Probieren Sie es aus!)

❷ In D3 gibt es *Skalierungsobjekte*, die den eingegebenen *Definitionsbereich* auf den ausgegebenen *Wertebereich* abbilden. Hier verwenden wir lineare Skalierungen, um die Datenwerte aus ihrem natürlichen Definitionsbereich in den Wertebereich der Pixel im Graphen zu übertragen. Die Bibliothek enthält jedoch auch logarithmische und exponentielle Skalierungen und sogar solche, die die Zahlenbereiche auf Farben abbilden, um Falschfarbendiagramme und Heatmaps zu erstellen (siehe Kapitel 7 und 8). Skalierungen sind Funktionsobjekte: Sie werden mit einem Wert des Definitionsbereichs aufgerufen und geben den skalierten Wert zurück.

❸ Sowohl der Definitions- als auch der Wertebereich werden als zweielementiges Array angegeben. Bei `d3.extent()` handelt es sich um eine Komfortfunktion, die ein Array (von Objekten) entgegennimmt und den größten und den kleinsten Wert als zweielementiges Array zurückgibt (siehe Kapitel 10). Um den gewünschten Wert aus den Objekten im Eingabearray zu entnehmen, müssen wir eine Zugriffsfunktion bereitstellen (ähnlich wie im letzten Schritt des vorherigen Beispiels). Um uns Tipparbeit zu ersparen, nutzen wir hier (und in den meisten folgenden Beispielen!) die *Pfeilfunktionen* von JavaScript (siehe Anhang C).

❹ Da die drei Spalten in der Datenmenge unterschiedliche Wertebereiche aufweisen, brauchen wir drei Skalierungsobjekte, eines für jede Spalte.

❺ Um die auf dem Kopf stehende Ausrichtung des SVG-Koordinatensystems auszugleichen, kehren wir für die vertikale Achse den Ausgabebereich in der Definition des Skalierungsobjekts um.

❻ Wir wählen das `<svg>`-Element aus, um Symbole für die erste Datenmenge hinzuzufügen.

❼ Dieser Vorgang ist neu: Bevor wir irgendwelche Elemente des Graphen hinzufügen, hängen wir ein `<g>`-Element an und geben ihm einen eindeutigen Bezeichner mit. Das endgültige Element sieht wie folgt aus:

```
<g id="ds1">...</g>
```

Das SVG-Element `<g>` sorgt für eine logische Gruppierung, sodass wir auf *alle* Symbole für die erste Datenmenge gleichzeitig verweisen und sie von denen der zweiten Datenmenge unterscheiden können (siehe Anhang B und den Kasten »*Das praktische <g>-Element*« auf *S. 115*).

❽ Wie zuvor erstellen wir mit `selectAll( "circle" )` eine leere Platzhaltersammlung. Die `<circle>`-Elemente werden als Kinder des gerade hinzugefügten `<g>`-Elements erstellt.

❾ Feste Formatierungen werden unmittelbar auf die einzelnen `<circle>`-Elemente angewendet.

❿ Eine Zugriffsfunktion wählt die geeignete Spalte für die horizontale Achse aus. Beachten Sie, dass der Skalierungsoperator auf die Daten angewendet wird, bevor sie zurückgegeben werden.

⓫ Eine Zugriffsfunktion wählt die geeignete Spalte für die erste Datenmenge aus. Auch hier erfolgt wieder eine passende Skalierung.

⓬ Das schon bekannte Verfahren wird erneut angewandt, um Elemente für die zweite Datenmenge hinzuzufügen. Beachten Sie aber die Unterschiede!

⓭ Für die zweite Datenmenge wird die Füllfarbe mit dem Attribut `fill` des `<g>`-Elements angegeben. Dieses Erscheinungsbild werden die Kinder erben. Durch die Definition im Elternelement können wir das Aussehen aller Kinder später in einem Rutsch ändern.

⓮ Abermals wird eine leere Sammlung für die neu hinzugefügten `<circle>`-Elemente erstellt. Hier ist das `<g>`-Element zu mehr als nur zu unserer Bequemlichkeit da: Wenn wir an dieser Stelle `selectAll( "circle" )` für das `<svg>`-Element aufriefen, würden wir keine leere Sammlung, sondern die `<circle>`-Elemente der *ersten* Datenmenge erhalten. Anstatt neue Elemente hinzuzufügen, würden wir die vorhandenen bearbeiten und die erste Datenmenge mit der zweiten überschreiben. Das `<g>`-Element erlaubt es uns, klar zwischen den Elementen und ihrer Zuordnung zu Datenmengen zu unterscheiden. (Das wird noch deutlicher, wenn wir uns in Kapitel 3 eingehender mit der `Selection`-API von D3 beschäftigen.)

⓯ Die Zugriffsfunktion wählt jetzt eine geeignete Spalte für die zweite Datenmenge aus.

⓰ Um die beiden Datenmengen besser unterscheiden zu können, wollen wir die Symbole, die zu einer Menge gehören, jeweils durch gerade Linien verbinden. Diese Linien sind komplizierter als die Symbole, da jedes Liniensegment von *zwei* aufeinanderfolgenden Datenpunkten abhängt. D3 kommt uns dabei aber zu Hilfe: Die Factory-Funktion `d3.line()` gibt ein Funktionsobjekt zurück, das bei Übergabe einer Datenmenge einen für das Attribut `d` des SVG-Elements `<path>` geeigneten String produziert. (Mehr über das Element `<path>` und seine Syntax lernen Sie in Anhang B.)

⓱ Der Liniengenerator erfordert eine Zugriffsfunktion, um die horizontalen und vertikalen Koordinaten für jeden Datenpunkt zu entnehmen.

⓲ Das `<g>`-Element der ersten Datenmenge wird anhand des Werts seines `id`-Attributs ausgewählt. Ein ID-Selektorstring besteht aus dem Nummernzeichen (`#`), gefolgt von dem Attributwert.

⓳ Ein `<path>`-Element wird als Kind der Gruppe `<g>` der ersten Datenmenge hinzugefügt ...

⓴ ... und sein Attribut `d` wird durch den Aufruf des Liniengenerators für die Datenmenge festgelegt.

㉑ Anstatt einen komplett neuen Liniengenerator zu erstellen, verwenden wir den vorhandenen weiter. Dabei geben wir eine neue Zugriffsfunktion an, diesmal für die zweite Datenmenge.

㉒ Ein `<path>`-Element für die zweite Datenmenge wird an der passenden Stelle im SVG-Baum angehängt und gefüllt.

㉓ Da der Füllungsstil für die Symbole der zweiten Datenmenge im Elternelement definiert wurde (nicht in den einzelnen `<circle>`-Elementen), ist es möglich, ihn in einer einzigen Operation zu ändern. Wenn Sie die Auskommentierung dieser Zeile aufheben, werden alle Kreise für die zweite Datenmenge rot dargestellt. Nur Optionen für das Erscheinungsbild werden vom Elternelement geerbt. Dagegen ist es nicht möglich, etwa den Radius aller Kreise auf diese Weise zu ändern oder die Kreise in Rechtecke umzuwandeln. Für solche Operationen müssen alle betroffenen Elemente einzeln angefasst werden.

Inzwischen haben Sie wahrscheinlich schon einen Eindruck von der Arbeit mit D3 gewonnen. Der Code ist zwar sehr wortreich, aber der ganze Vorgang hat auch etwas von einem Baukastensystem, bei dem sie einfach vorgefertigte Elemente zusammenstecken. Insbesondere die Methodenkettung kann wie die Zusammenstellung einer Unix-Pipeline wirken (oder wie das Bauen mit LEGO-Steinen). Bei den Komponenten selbst liegt der Schwerpunkt gewöhnlich mehr auf den Mechanismen als auf der Strategie. Dadurch können Sie für viele verschiedene Zwecke wiederverwendet werden, allerdings werden Programmierer oder Designer zusätzlich mit der Verantwortung belastet, semantisch sinnvolle grafische Konstruktionen zusammenzustellen. Als letzten Gesichtspunkt möchte ich noch hervorheben, dass D3 Entscheidungen zugunsten einer »späten Bindung« verzögert. So werden beispielsweise gewöhnlich Zugriffsfunktionen als Argumente übergeben, anstatt zu verlangen, dass die passenden Spalten vor der Übergabe an das Rendering-Framework aus der ursprünglichen Datenmenge extrahiert werden.

Diagrammelemente mit wiederverwendbaren Komponenten hinzufügen

Der Graph aus Abbildung 2–2 ist ziemlich spartanisch: Er zeigt nur die Daten und nichts sonst. Insbesondere gibt er die Skalen nicht an, was hier besonders wichtig wäre, da für die beiden Datensätze unterschiedliche Wertebereiche gelten. Ohne Skalen oder Achsen ist es nicht möglich, quantitative Informationen aus diesem

Graph (oder überhaupt aus irgendeinem Diagramm) abzulesen. Daher müssen wir ihm Achsen hinzufügen.

Aufgrund der Teilstriche und Teilstrichbeschriftungen sind Achsen komplexe grafische Elemente. Zum Glück gibt es in D3 eine Funktionalität zum Anlegen von Achsen, die bei Übergabe eines Skalierungsobjekts (das den Definitions- und den Wertebereich sowie die Abbildung zwischen ihnen definiert) alle erforderlichen grafischen Elemente erstellt und zeichnet. Da die grafische Achsenkomponente aus vielen einzelnen SVG-Elementen besteht (für die Teilstriche und Beschriftungen), sollte sie immer in ihrem eigenen `<g>`-Container erstellt werden. Die auf dieses Elternelement angewandten Formatierungen und Transformationen werden von allen Teilen der Achse geerbt. Das ist wichtig, da sich alle Achsen zu Anfang im Ursprung befinden (in der oberen linken Ecke) und mithilfe des Attributs `transform` zu ihrer gewünschten Position verschoben werden müssen. (Genauere Erläuterungen zu Achsen folgen in Kapitel 7.)

Neben dem Hinzufügen von Achsen und der bis jetzt noch nicht gezeigten D3-Funktionalität zum Erstellen von Kurven demonstriert Beispiel 2–6 auch eine andere Arbeitsweise mit D3. Der Code in Beispiel 2–5 war geradlinig, aber umfangreich, und enthielt eine Menge Wiederholungen. Beispielsweise war der Code zum Erstellen der drei verschiedenen Skalierungsobjekte fast identisch. Auch der Code für die Symbole und Linien wurde für die zweite Datenmenge größtenteils dupliziert. Die Vorteile dieses Stils sind Einfachheit und ein linearer logischer Verlauf, was allerdings mit einer größeren Wortfülle erkauft wird.

In Beispiel 2–6 gibt es weniger redundanten Code, da die mehrfach verwendeten Anweisungen in lokale Funktionen gepackt wurden. Da diese Funktionen innerhalb von `makeDemo3()` definiert sind, haben sie Zugriff auf die Variablen in diesem Gültigkeitsbereich. Das hilft dabei, die Anzahl der erforderlichen Argumente für die lokalen Hilfsfunktionen gering zu halten. Dieses Beispiel stellt auch *Komponenten* als Grundbausteine der Kapselung und Wiederverwendung vor und zeigt »synthetische« Funktionsaufrufe – lauter wichtige Techniken für die Arbeit mit D3.

Beispiel 2–6: Befehle für Abbildung 2–3

```
function makeDemo3() {
    d3.tsv( "examples-multiple.tsv" )
        .then( function( data ) {
            var svg = d3.select( "svg" );   ❶

            var pxX = svg.attr( "width" );   ❷
            var pxY = svg.attr( "height" );
```

```
        var makeScale = function( accessor, range ) {    ❸
            return d3.scaleLinear()
                .domain( d3.extent( data, accessor ) )
                .range( range ).nice();
        }
        var scX = makeScale( d => d["x"], [0, pxX] );
        var scY1 = makeScale( d => d["y1"], [pxY, 0] );
        var scY2 = makeScale( d => d["y2"], [pxY, 0] );

        var drawData = function( g, accessor, curve ) {    ❹
            // Zeichnet Kreise
            g.selectAll( "circle" ).data(data).enter()
                .append("circle")
                .attr( "r", 5 )
                .attr( "cx", d => scX(d["x"]) )
                .attr( "cy", accessor );

            // Zeichnet Linien
            var lnMkr = d3.line().curve( curve )    ❺
                .x( d=>scX(d["x"]) ).y( accessor );

            g.append( "path" ).attr( "fill", "none" )
                .attr( "d", lnMkr( data ) );
        }

        var g1 = svg.append( "g" );    ❻
        var g2 = svg.append( "g" );

        drawData( g1, d => scY1(d["y1"]), d3.curveStep );    ❼
        drawData( g2, d => scY2(d["y2"]), d3.curveNatural );

        g1.selectAll( "circle" ).attr( "fill", "green" );    ❽
        g1.selectAll( "path" ).attr( "stroke", "cyan" );

        g2.selectAll( "circle" ).attr( "fill", "blue" );
        g2.selectAll( "path" ).attr( "stroke", "red" );

        var axMkr = d3.axisRight( scY1 );    ❾
        axMkr( svg.append("g") );    ❿

        axMkr = d3.axisLeft( scY2 );

        svg.append( "g" )
            .attr( "transform", "translate(" + pxX + ",0)" )    ⓫
            .call( axMkr );    ⓬

        svg.append( "g" ).call( d3.axisTop( scX ) )
            .attr( "transform", "translate(0,"+pxY+")" );    ⓭
    } );
}
```

❶ Das <svg>-Element wird gezeichnet und einer Variablen zugewiesen, sodass es später verwendet werden kann, ohne `select()` aufrufen zu müssen.

❷ Als Nächstes wird die Größe des <svg>-Elements abgefragt. Viele D3-Funktionen können sowohl als Set- als auch als Get-Methoden dienen: Wird ein zweites Argument angegeben, so wird die genannte Eigenschaft auf diesen Wert gesetzt; anderenfalls wird der aktuelle Wert der Eigenschaft zurückgegeben. Hier nutzen wir diesen Umstand, um die Größe des <svg>-Elements zu ermitteln.

❸ Bei der Funktion `makeScale()` handelt es sich lediglich um einen komfortablen Wrapper, um D3-Funktionsaufrufe knapper formulieren zu können. Skalierungsobjekte kennen Sie bereits aus dem vorhergehenden Listing (Beispiel 2–5). Wird die Funktion `nice()` für ein Skalierungsobjekt aufgerufen, so erweitert sie den Wertebereich auf den nächsten »runden« Wert.

❹ Die Funktion `drawData()` kombiniert alle erforderlichen Befehle, um eine einzige Datenmenge auszugeben: Sie erstellt sowohl die Kreise für die einzelnen Datenpunkte als auch die verbindenden Linien. Beim ersten Argument von `drawData()` muss es sich um eine Selection handeln, gewöhnlich ist das ein <g>-Element als Behälter für alle grafischen Elemente zur Darstellung einer einzelnen Datenmenge. Funktionen, die eine Selection als erstes Argument übernehmen und dann Elemente zu ihr hinzufügen, werden als *Komponenten* bezeichnet und bilden einen wichtigen Mechanismus für Kapselung und Codewiederverwendung in D3. Dies ist das erste Beispiel dafür. Das Anlegen der Achsen weiter hinten in diesem Code ist ein weiteres. (Eine ausführliche Darstellung finden Sie in Kapitel 5.)

❺ Die Factory `d3.line()` kennen Sie bereits aus Beispiel 2–5. Sie kann einen Algorithmus entgegennehmen, der definiert, mit welcher Art von Kurve aufeinanderfolgende Punkte verbunden werden sollen. Als Standard werden gerade Linien verwendet, aber D3 definiert noch viele andere Algorithmen. Außerdem können Sie Ihre eigenen schreiben. (Wie das geht, erfahren Sie in Kapitel 5.)

❻ Hier werden die beiden <g>-Container für die beiden Datenmengen erstellt.

❼ Bei ihrem Aufruf werden der Funktion `drawData()` eines der Containerelemente, eine Zugriffsfunktion, die die Datenmenge angibt, und die gewünschte Kurvenform übergeben. Nur um zu zeigen, was alles möglich ist, verwenden wir für die beiden Datenmengen hier zwei unterschiedliche Arten von Kurven, nämlich eine mit Stufenfunktionen und die andere mit natürlichen kubischen Splines. Die Funktion `drawData()` fügt die erforderlichen <circle>- und <path>-Elemente zum <g>-Behälter hinzu.

❽ Für jeden Behälter werden die gewünschten grafischen Elemente ausgewählt, um ihre Farbe festzulegen. Obwohl es sehr einfach möglich gewesen wäre, wurde die Auswahl der Farbe absichtlich nicht in die Funktion `drawData()` aufgenommen. Das steht im Einklang mit einem üblichen D3-Idiom: Das Erstellen von DOM-Elementen wird von der Konfiguration ihres Erscheinungsbilds getrennt gehalten.

❾ Die Achse für die erste Datenmenge wird an der linken Seite des Graphen gezeichnet. Denken Sie daran, dass alle Achsen standardmäßig im Ursprung dargestellt werden. Das Objekt `axisRight` zeichnet Teilstrichbeschriftungen *rechts* neben die Achse, sodass sie sich außerhalb des Graphen befinden, wenn die Achse an dessen rechter Seite platziert wird. Hier dagegen geben wir die Achse links aus, sodass sich die Teilstrichbeschriftungen innerhalb des Diagramms befinden.

❿ Die Factory-Funktion `d3.axisRight( scale )` gibt ein Funktionsobjekt zurück, das die Achse mit all ihren Teilen generiert. Sie erfordert einen SVG-Container (gewöhnlich ein `<g>`-Element) als Argument und erstellt alle Elemente der Achse als Kinder dieses Containerelements. Mit anderen Worten, es handelt sich dabei um eine *Komponente* nach der weiter vorn gegebenen Definition. (Mehr darüber erfahren Sie in Kapitel 7.)

⓫ Für die Achse auf der rechten Seite des Graphen muss das Containerelement an die gewünschte Stelle verlegt werden. Dazu wird das SVG-Attribut `transform` verwendet.

⓬ Hier geschieht etwas Neues: Anstatt die Funktion `asMkr` explizit mit dem einschließenden `<g>`-Element als Argument aufzurufen, übergeben wir sie als Argument an die Funktion `call()`, die zur API `Selection` gehört (siehe Kapitel 3) und nach einer ähnlichen Funktionalität in der Sprache JavaScript gestaltet wurde. Sie ruft ihr Argument (bei dem es sich um eine Funktion handeln muss) auf und übergibt ihr die aktuelle Selection als Argument. Diese Art von »synthetischem« Funktionsaufruf ist in JavaScript sehr gebräuchlich, weshalb es sinnvoll ist, sich daran zu gewöhnen. Ein Vorteil dieses Programmierstils besteht darin, dass er eine Verkettung von Methoden ermöglicht, wie Sie in der nächsten Zeile sehen.

⓭ Hier fügen wir die horizontale Achse am unteren Rand des Graphen hinzu. Nur um zu demonstrieren, was alles möglich ist, wurde die Reihenfolge der Funktionsaufrufe umgekehrt: Die Achsenkomponente wird zuerst aufgerufen, und erst danach wird die Transformation angewendet. An dieser Stelle brau-

chen wir auch die Hilfsvariable `axMkr` nicht mehr. Dies ist wahrscheinlich die idiomatischste Weise, um diesen Code zu schreiben.[3]

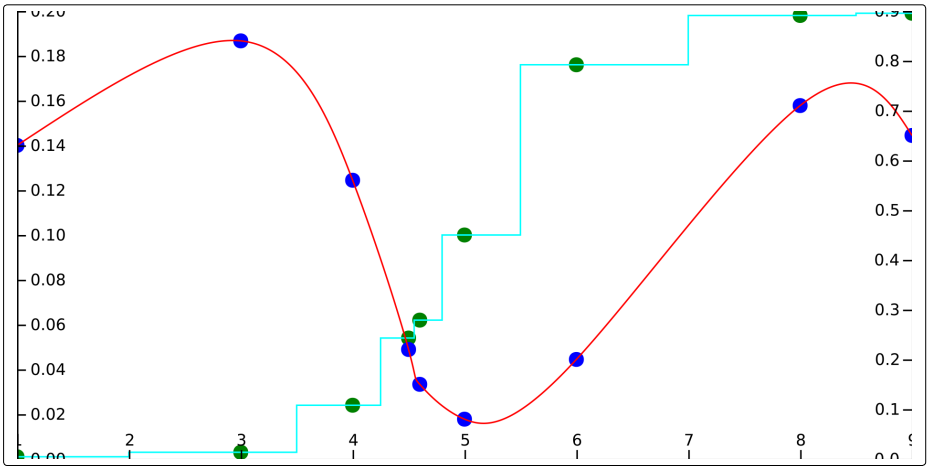

Abb. 2–3 *Ein verbessertes Diagramm mit Koordinatenachsen und verschiedenen Arten von Kurven (vgl. Abb. 2–2 und Beispiel 2–6)*

Den resultierenden Graphen sehen Sie in Abbildung 2–3. Er ist nicht schön, aber er erfüllt seinen Zweck und zeigt die beiden Datenmengen jeweils mit den zugehörigen Maßstäben. Um das Aussehen des Graphen zu verbessern, könnte in einem ersten Schritt um die eigentlichen Daten des SVG-Bereichs herum noch etwas Platz für die Achsen und Teilstrichbeschriftungen geschaffen werden. (Probieren Sie es aus!)

Drittes Beispiel: Listeneinträge animieren

Unser drittes und letztes Beispiel mag etwas seltsamer sein als die beiden ersten, veranschaulicht aber zwei wichtige Dinge:

- D3 ist nicht darauf beschränkt, SVG-Grafiken zu generieren, sondern kann *alle* Teile des DOM-Baums bearbeiten. In diesem Beispiel wollen wir damit herkömmliche HTML-Listenelemente gestalten.
- Mit D3 ist es einfach, reaktive und animierte Dokumente zu erstellen, die auf Benutzerereignisse (wie Mausklicks) reagieren bzw. ihr Erscheinungsbild mit der Zeit ändern. Hier zeige ich nur einen winzigen Ausschnitt der erstaunlichen Möglichkeiten. Ausführlicher werden wir dieses Thema noch in Kapitel 4 behandeln.

3 Auch die Funktion `drawData()` wird gewöhnlich auf diese Weise aufgerufen: `g1.call( drawData, d => scY1(d["y1"]), d3.curveStep ).`

HTML-Elemente mit D3 erstellen

Zur Abwechslung – und weil das Skript sehr kurz ist – sind die JavaScript-Befehle hier unmittelbar in der HTML-Seite und nicht in einer eigenen Datei enthalten. Die vollständige Seite für das aktuelle Beispiel einschließlich der D3-Befehle finden Sie in Beispiel 2–7 (siehe auch Abb. 2–4).

Beispiel 2–7: D3 für nicht grafische HTML-Bearbeitung verwenden (siehe Abb. 2–4)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script src="d3.js"></script>
  <script>
function makeList() {
    var vs = [ "From East", "to West,", "at Home", "is Best" ];   ❶

    d3.select( "body" )   ❷
        .append( "ul" ).selectAll( "li" )   ❸
        .data(vs).enter()   ❹
        .append("li").text( d => d );   ❺
}
  </script>
</head>

<body onload="makeList()" />
</html>
```

Die Struktur dieses Beispiels ist fast identisch mit derjenigen des Beispiels am Anfang dieses Kapitels (siehe insbesondere Beispiel 2–3), allerdings gibt es auch einige Abweichungen im Detail:

❶ Die Datenmenge wird nicht aus einer externen Datei geladen, sondern im Code selbst definiert.

❷ Als äußerster Container wird das HTML-Element `<body>` ausgewählt.

❸ Der Code hängt ein `<ul>`-Element an und erstellt dann (mit `selectAll( "li" )`) einen leeren Platzhalter für die Listeneinträge.

❹ Wie zuvor wird die Datenmenge an die Selection gebunden und die Sammlung der Datenpunkte ohne übereinstimmende DOM-Elemente abgerufen.

❺ Schließlich wird für jeden Datenpunkt ein Listeneintrag angehängt, dessen Inhalt (in diesem Beispiel der Text des Listeneintrags) jeweils mit Werten aus der Datenmenge gefüllt wird.

All das ähnelt sehr stark dem, was wir schon zuvor getan haben. Der Unterschied besteht darin, dass das Ergebnis einfaches HTML in Textform ist. D3 ist also auch ein hervorragendes Werkzeug für die allgemeine (also nicht nur die grafische) DOM-Bearbeitung.

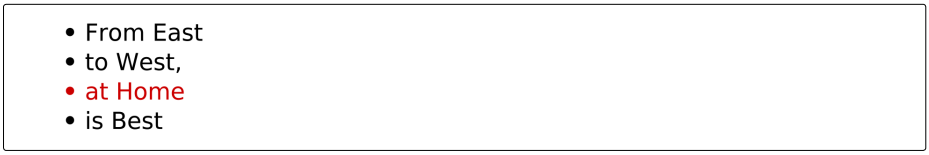

Abb. 2–4 *Eine Liste mit Aufzählpunkten in HTML (siehe Beispiele 2–7 und 2–8)*

Eine einfache Animation erstellen

Es kostet nicht viel Mühe, dafür zu sorgen, dass dieses Dokument auf Benutzerereignisse reagiert. Wenn Sie die Funktion `makeList()` in Beispiel 2–7 durch diejenige aus Beispiel 2–8 ersetzen, können Sie die Textfarbe von Schwarz in Rot und zurück ändern, indem Sie auf ein Listenelement klicken. Die Änderung erfolgt allerdings nicht sofort, sondern zieht sich über einige Sekunden hin.

Beispiel 2–8: Animation als Reaktion auf Benutzerereignisse

```
function makeList() {
    var vs = [ "From East", "to West,", "at Home", "is Best" ];

    d3.select( "body" )
        .append( "ul" ).selectAll( "li" )
        .data(vs).enter()
        .append("li").text( d => d )    ❶
        .on( "click", function () {    ❷
            this.toggleState = !this.toggleState;    ❸ ❹
            d3.select( this )    ❺
                .transition().duration(2000)    ❻
                .style("color", this.toggleState?"red":"black");    ❼
        } );
}
```

❶ Bis zu diesem Punkt ist die Funktion identisch mit der aus Beispiel 2–7.

❷ Die Funktion `on()` registriert einen Callback für den genannten Ereignistyp (in diesem Fall `"click"`) mit dem aktuellen Element als DOM-Ereignisziel (`EventTarget`). Jeder Listeneintrag kann jetzt Klickereignisse empfangen und übergibt sie an den bereitgestellten Callback.

❸ Wir müssen uns den jeweils aktuellen Zustand der einzelnen Listeneinträge merken. Wo sonst können wir diese Information ablegen als in dem Element selbst? Die sehr liberale Haltung von JavaScript macht das extrem einfach: Wir fügen einfach einen neuen Member zu dem Element hinzu! Vor dem Aufruf des Callbacks weist D3 das aktive DOM-Element `this` zu und ermöglicht so den Zugriff auf dieses Element.

❹ Diese Zeile nutzt die liberale Haltung von JavaScript auch noch für einen anderen Zweck. Beim ersten Aufruf des Callbacks (für jeden Listeneintrag) ist `toggleState` noch nicht zugewiesen und hat daher den besonderen Wert `undefined`, der in einem booleschen Kontext zu `false` ausgewertet wird. Daher ist es nicht nötig, die Variable zu initialisieren.

❺ Um mithilfe der Methodenverkettung auf dem aktuellen Knoten arbeiten zu können, müssen wir ihn in eine Selection einschließen.

❻ Die Methode `transition()` interpoliert nahtlos zwischen dem aktuellen Status des ausgewählten Elements (in diesem Fall dem aktuellen Listeneintrag) und dessen gewünschtem endgültigem Erscheinungsbild. Für das Intervall für den Übergang sind 2000 ms festgelegt. D3 kann zwischen Farben (über deren numerische Darstellung) und vielen anderen Größen interpolieren. (Die Interpolation wird in Kapitel 7 behandelt.)

❼ Schließlich wird die neue Textfarbe auf der Grundlage des aktuellen Zustands der Statusvariablen ausgewählt.