Geogebra in JSXGraph

Schriftliche Hausarbeit

gem. §30 LPO 1

eingereicht bei

Dr. Matthias Ehmann

Lehrstuhl für Mathematik und ihre Didaktik

Didaktik der Informatik

Fakultät 1

Universität Bayreuth

von

Peter Wilfahrt

Wichernstraße 17

95447 Bayreuth

Matr. Nr. 1093553

Lehramt Realschule

Informatik/Wirtschaftswissenschaften

eingereicht am

23.12.2009

im WS 2009/10

1. JSXGraph

Sucht man mit einer der großen Suchmaschinen im Internet nach JSXGraph[1], wird diese Definition präsentiert:

> *„JSXGraph is a cross-browser library for interactive geometry, function plotting, charting, and data visualization in a web browser“*[2]

Das Ergebnis erläutert treffend was JSXGraph ist, klärt den Leser aber nicht darüber auf, was dessen Besonderheiten sind.

Warum ist es wichtig, dass JSXGraph „Open Source“ ist?

Unter diesem Begriff versteht man im Allgemeinen freie Software, die ohne Kosten nutz- und für Programmierer im Quelltext verfügbar ist.

Im Rahmen dieser Arbeit möchte ich Sie für die folgende Interpretation des Begriffs „Open Source“ inspirieren:

„Open“ als: offen für alle Plattformen[3], offen für alle Webbrowser[4], offen und zugänglich für alle Sprachen und Länder, offen und erweiterbar für jeden Programmierer[5], offen und kombinierbar mit vielen Programmen.

„Source“ kann neben der allgemeinen Bedeutung als Quelltext auch für Datenquelle stehen. So ist JSXGraph nicht nur mit eigenen Funktionen programmierbar, sondern bietet neben einer graphischen Oberfläche[6] auch die Möglichkeit, Daten aus anderen Programmen zu importieren und bewegt sich so wieder zirkular auf den Begriff „Open“ zu.

Dieser Aspekt sowie die motivierende Integration im JSXGraph-Entwicklungsteam[7] haben mich dazu bewogen, den im Folgenden vorgestellten GeogebraReader[8] zu programmieren. Er stellt eine Importschnittstelle für Geogebra[9]-Dateien zur Verwendung in JSXGraph dar. Damit wird es nun für viele Schüler und Lehrer möglich, die Vorteile von JSXGraph und Geogebra, der dynamischen Mathematiksoftware, zu kombinieren und Synergieeffekte zu nutzen.

---

[1] vgl. http://www.google.de/search?q=jsxgraph

[2] vgl. http://www.jsxgraph.org/

[3] Unabhängig ob Linux, Mac OS X, Windows oder ein mobiles Endgerät

[4] Mozilla Firefox, Opera, Safari, Google Chrome und Microsoft Internet Explorer

[5] vgl. http://jsxgraph.uni-bayreuth.de/docs

[6] Die GUI befindet sich gerade in der Entwicklung.

[7] vgl. http://jsxgraph.uni-bayreuth.de/wp/documentation/the-team/

[8] vgl. angehängte CD-Rom 6.2.

[9] vgl. http://www.geogebra.org/

2. Geogebra in JSXGraph

Versucht man sich der Problemstellung, wie in den einleitenden Sätzen beschrieben, anzunehmen, muss man sich zuerst Gedanken über die vorhandenen Eingabequellen machen, die verarbeitet werden sollen.

Geogebra-Konstruktionen werden beim Speichern in der Software Geogebra zunächst in ein Archiv gepackt. Dieses enthält neben eventuell hinzugefügten Bildern die Datei `geogebra.xml`. In selbiger werden alle Konstruktionsanweisungen und Elemente aufgelistet, so dass wir hier unsere Verarbeitungsanweisungen finden.

Die erste Schwierigkeit besteht zunächst darin, dass der Dateiinhalt entpackt und an die auslesende Funktion weitergegeben wird, bevor diese alle nachfolgenden, notwendigen Schritte vollzieht.

Anschließend werden die XML-Daten in die jeweiligen geometrischen Konstruktionen umgesetzt und zuletzt werden die mathematischen Funktionen aus der Datei interpretiert.

Um diese Prozesse nachzuvollziehen, erkläre ich nun die dafür notwendigen Bestandteile.

Auf der folgenden Seite ist das Ablaufdiagramm für den gesamten Ausleseprozess aufgeführt. Die einzelnen Abschnitte werden in den Unterpunkten dieses Kapitels erläutert.

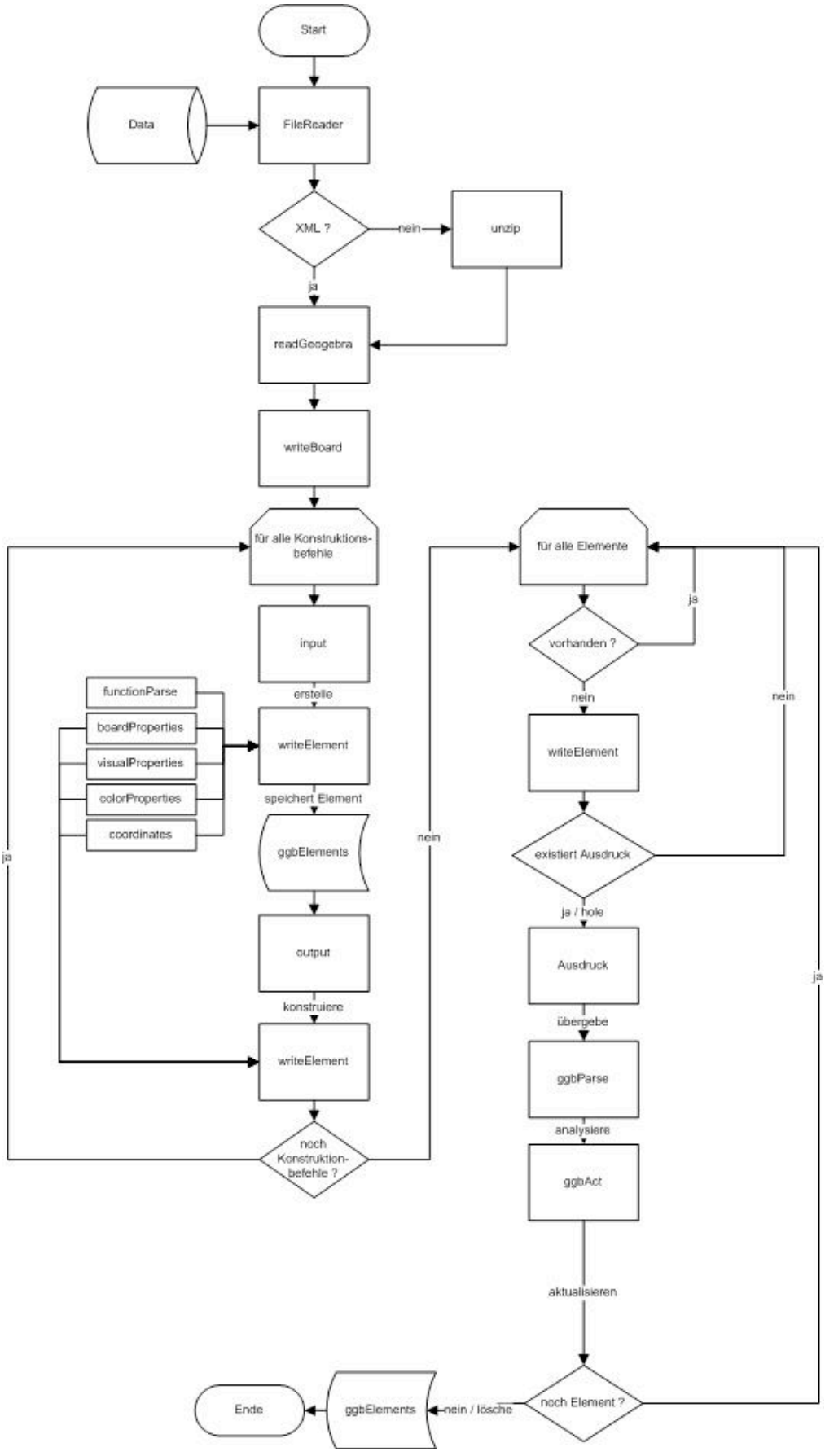

Abbildung 1: Ablaufdiagramm GeogebraReader

2.1. Geogebra XML-Schema

Die Konstruktionsbeschreibung und die Liste aller vorhandenen Elemente werden beim Speichern von Geogebra in die Datei `geogebra.xml` geschrieben. Ein Punkt wird mit diesen Zeilen beschrieben:

```
<element type="point" label="A">
        <show object="true" label="true"/>
        <objColor r="0" g="0" b="255" alpha="0.0"/>
        <labelMode val="0"/>
        <fixed val="false"/>
        <breakpoint val="false"/>
        <coords x="1.0" y="1.0" z="1.0"/>
        <coordStyle style="cartesian"/>
        <pointSize val="3"/>
</element>
```

Um an diesem XML-Code zu arbeiten, muss er an den GeogebraReader weitergereicht werden. Dazu ist es in den meisten Fällen notwendig das Archiv zu entpacken und den reinen Code weiterzugeben. Dazu wird die Datei oder der Dateiinhalt als String durch den FileReader verarbeitet und mit dem von Dr. Matthias Ehmann[10] implementierten Unzip[11] entpackt und als XML-Baum an den Reader gegeben. Eine entpackte Beispieldatei ist in der angehängten CD-Rom als Punkt 6.1. aufgeführt. Aus obiger Auflistung wird ersichtlich, dass die einzelnen Attribute, wie zum Beispiel die Koordinaten im XML-Tag `coords` die entsprechenden Werte mitbringen. So liegt dieser Punkt bei x = 1 und y = 1.

Die komplette Konstruktionsbeschreibung ist nach dem Geogebra XML File Format[12] aufgebaut und kann anhand dessen ausgelesen werden. In Abbildung 1 sind für das Auslesen der XML-Daten folgende Elemente notwendig:

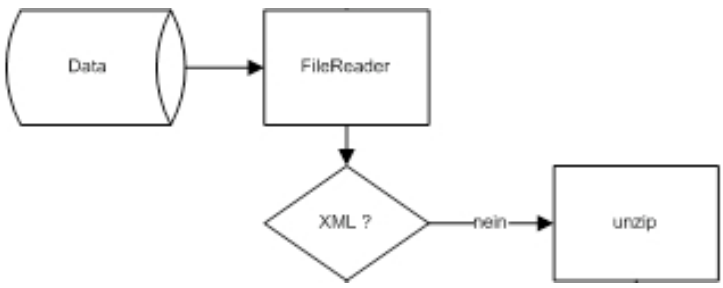

Abbildung 2: Auslesen der XML-Daten

[10] vgl. http://did.mat.uni-bayreuth.de/~matthias/

[11] vgl. http://jsxgraph.uni-bayreuth.de/docs/symbols/JXG.Util.html#.Unzip

[12] http://www.geogebra.org/source/program/xml_format/GeoGebra_XML_File_Format_2.5.htm

2.2. Auslesen und Konstruktionsbefehle

Nachdem die XML-Daten, wie in 2.1. beschrieben, zur Verfügung stehen, wird damit zuerst das Zeichenblatt erzeugt, bevor die einzelnen Konstruktionsbefehle durchlaufen und ausgeführt werden. Die mit dem XML-Tag `command` aufgelisteten Befehle werden, wie rechts schematisch dargestellt, der Reihe nach abgearbeitet, indem die jeweiligen Eingabeelemente (bei einer Linie zum Beispiel zwei Punkte) zuerst erzeugt und dann über die Funktion `writeElement` zu einer Linie verarbeitet werden. Der XML-Code lautet beispielsweise für ein Dreieck:

```
<command name="Polygon">
  <input a0="C" a1="D" a2="B"/>
  <output a0="poly1" a1="b"
          a2="c" a3="d"/>
</command>
```

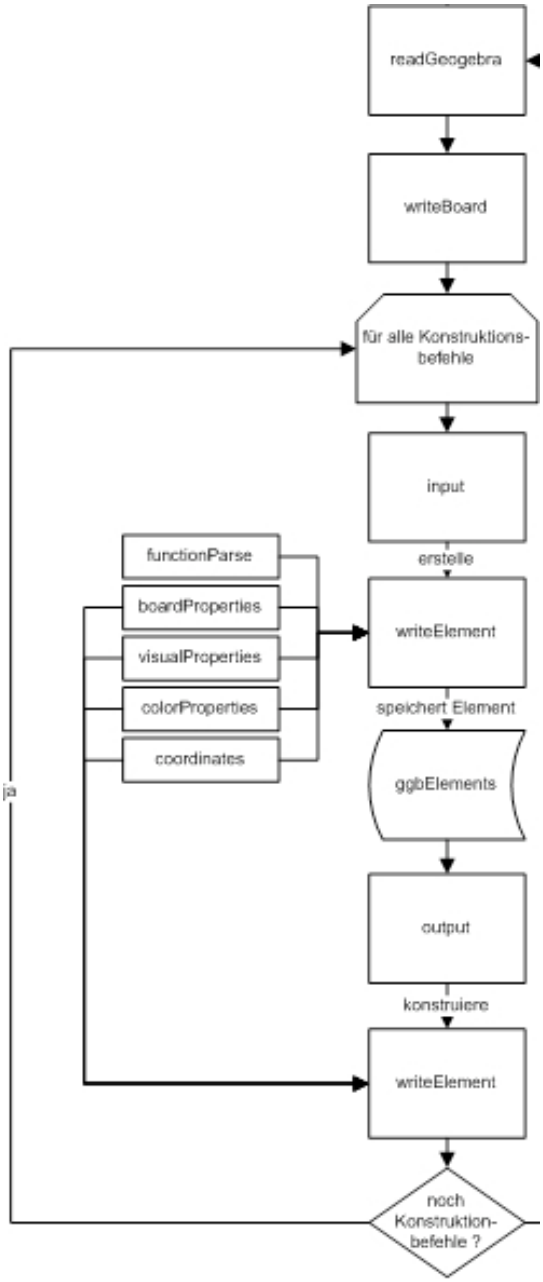

Abbildung 3: readGeogebra [1]

Die konstruierten Objekte werden bis zum Ende des Auslesevorgangs in `ggbElements` gespeichert, um eine Namensauflösung zu gewährleisten und eine wiederholte Erzeugung zu vermeiden.

Die Eigenschaften, wie zum Beispiel Farben, Koordinaten, Anzeigeoptionen und mehr, werden über die im Diagramm links aufgeführten Funktionen bewerkstelligt und sind ausgelagert, da sie für jedes Objekt benötigt werden. Gibt es bei einem Element mehrere Ausgabeelemente, so werden deren Attributwerte über die an `writeElement` optionalen `output`-Elemente übergebenen. Diese werden nachträglich wie alle anderen Elemente in `ggbElements` gespeichert, um auch hier die Namensauflösung und Weiterverwendung zu ermöglichen.

2.3. Konstruktion der Elemente

`writeElement` unterscheidet zwischen allen geometrischen Objekten und konstruiert im Bedarfsfall weitere Hilfselemente oder übergibt alle notwendigen Parameter an die JSXGraph-interne Funktion `createElement`.

Im Quellcode des GeogebraReaders sieht der dafür benötigte Code folgendermaßen aus:

```
try {
  if(typeof input != 'undefined')
    p = board.create('glider', [gxtEl.x, gxtEl.y, input[0]], attr);
  else
    p = board.create('point', [gxtEl.x, gxtEl.y], attr);

    JXG.GeogebraReader.debug("* <b>Point ("+ p.id +"):</b> "+ attr.name + "("+ gxtEl.x +", "+ gxtEl.y +")<br>\n");
    return p;
} catch(e) {
    JXG.GeogebraReader.debug("* <b>Err:</b> Point " + attr.name +"<br>\n");
    return false;
}
```

Abbildung 4: Quelltextauszug writeElement (point)

Bei den verschiedenen Konstruktionsbefehlen wird der notwendige Code erweitert, sodass das Ausgabeelement korrekt funktioniert. Im Fall einer Tangente sieht der wesentliche Quelltext dann im Vergleich zu obigem „einfachen“ Punkt so aus:

```
try {
  JXG.GeogebraReader.debug("* <b>Tangent:</b> First: " + input[0].name + ", Sec.: "+ input[1].name +"("+ input[1].type +")<br>\n");
  switch(input[1].type) {
    case 1330923344: // graph
      input[0].makeGlider(input[1]);
      t = board.create('tangent', [input[0]], attr);
      return t;
    break;
    case 1330922316: // circle
      var m = function(circ) {
      return [[circ.midpoint.X()*circ.midpoint.X()+circ.midpoint.Y()*circ.midpoint.Y()-circ.getRadius()*circ.getRadius(),
                  -circ.midpoint.X(),-circ.midpoint.Y()],
                [-circ.midpoint.X(),1,0],
                [-circ.midpoint.Y(),0,1]
               ];
          };

      var t = board.create('line', [
                function(){ return JXG.Math.matVecMult(m(input[1]), input[0].coords.usrCoords)[0]; },
                function(){ return JXG.Math.matVecMult(m(input[1]), input[0].coords.usrCoords)[1]; },
                function(){ return JXG.Math.matVecMult(m(input[1]), input[0].coords.usrCoords)[2]; }
              ], {visible: false});

      var i1 = board.create('intersection', [input[1], t, 0], {visible: false});
      var i2 = board.create('intersection', [input[1], t, 1], {visible: false});
      var t1 = board.create('line', [input[0], i1]);
      var t2 = board.create('line', [input[0], i2]);
    break;
  }
} catch(e) {
  JXG.GeogebraReader.debug("* <b>Err:</b> Tangent " + attr.name +"<br>\n");
  return false;
}
```

Abbildung 5: Quelltextauszug writeElement (tangent)

2.4. Analyse und Interpretation von Ausdrücken

Das Ablaufdiagramm in 2.2. zeigt ausschließlich die Befehlskonstruktion, die mit der Konstruktion der Elemente Hand in Hand läuft. Der Unterschied und Grund der Trennung der beiden Funktionen liegt darin begraben, dass nicht jedes Element einem Befehl zugeordnet ist.

Um sicherzustellen, dass jedes Element nach dem Auslesen vorhanden ist, wird abschließend jedes Element, egal ob vorher konstruiert oder nicht, nochmals betrachtet. Ist dieses schon vorhanden, wird es übersprungen. Falls nicht, wird es konstruiert und überprüft, ob es einen Ausdruck (XML-Tag: `expression`) dazu gibt, wie z. B.:

```
<expression label ="f"
            exp="f(x) = sin(x)"/>
```

Sollte dies der Fall sein, muss der Ausdruck in eine JavaScript-Funktion übersetzt werden.

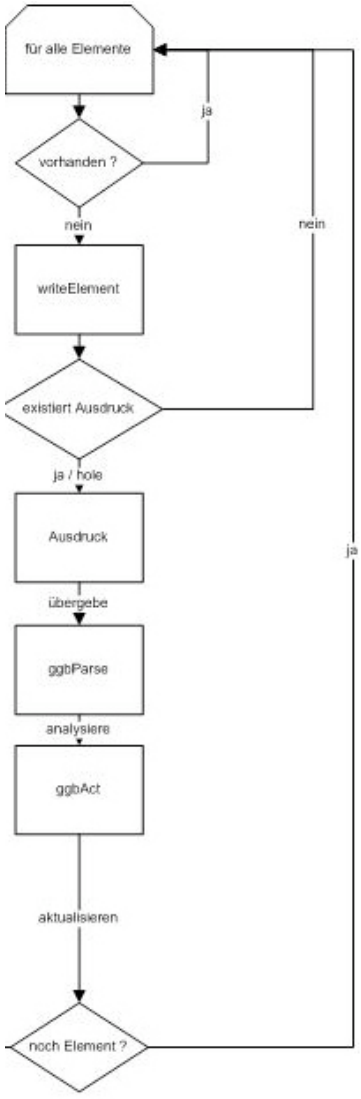

Abbildung 6: Ablaufdiagramm Elemente

Bevor der Funktionsausdruck an den in Punkt 2.5. ausführlich erläuterten LALR(1)-Parser übergeben wird, muss der String vorbehandelt werden. Das im Mathematischen gebräuchliche `'a b'` für `'a*b'` wird von Parser und JavaScript nicht erkannt und muss somit vorbehandelt werden. Dies ist bei Dateien im Geogebraformat bis Version 3.02 nötig, in der letzteren Version werden die Terme bereits korrekt abgespeichert. Wie die elementaren Mathematikfunktion `sin(x)` in das zugehörige `Math.sin(x)` ersetzt werden muss, gilt dies analog für `cos(x)` und viele mehr. In der GeogebraReader-Methode `functionParse` wird der Funktionsterm für die Weiterverwendung vorbereitet und, mit regulären Ausdrücken bearbeitet, sodass es eindeutige Funktionsparameter und ein für den Parser korrekter Term zurückgegeben wird.

Damit wird ein Ausdruck wie: `(sin(2 a), a^(2))`

zum Parsen vorbereitet in: `(sin(2*a), a^(2))`

Wie diese Zeichenkette als Koordinate verarbeitet wird behandelt das Kapitel 2.5..

Um später das Ersetzen von Elementnamen und Variablen[13] unterscheiden zu können, wird ein Term wie `f(x) = [...]` in eine anonyme Funktion[14] `function(__x)` umgewandelt. So wird sichergestellt, dass später alle Variablen mit dem Präfix `'__'` nicht betrachtet werden.

Das ist zum Beispiel bei folgender Funktion wichtig:

```
f(x) = x^(3) - 3 x^(2) + a
```

Hier gilt `'x'` als Funktionsparameter, während `'a'` den Wert eines Sliders[15] darstellt. Für die Funktion darf nur `'x'` entsprechend präpariert werden, alle anderen Variablen aber nicht.

Nach dem Vorbereiten wird mit dieser Funktion weitergearbeitet:

```
function(__x) = __x^(3) - 3*__x^(2) + a
```

JavaScript kann damit noch nicht korrekt rechnen, daher muss der Funktionsrumpf noch weiterbehandelt werden. Insbesondere muss die Rechenanweisung `'^'`[16] und der Elementzugriff mittels `'a'` noch umgesetzt werden in das korrekte `Math.pow()` sowie `'a'` als Referenz auf das Element mit dem Namensattribut `'a'`.

In älteren Dateiformaten wird seitens Geogebra gestattet, dass Unicodezeichen wie ² oder ³ als Exponenten oder `'≟'` statt dem JavaScript-Äquivalent `'=='` verwendet werden. Diese und viele andere Operanden werden vor dem Parsen in JavaScript-Syntax umgesetzt, indem überprüft wird, ob das Zeichen vorhanden ist und dann ersetzt wird:

```
exp = (exp.match(/\u225F/)) ?
        exp.replace(/\u225F/, '==') : exp;
```

Im folgenden Kapitel wird erläutert, wie die mathematischen Ausdrücke, wie obige Potenzen, umgesetzt werden und Kapitel 2.6.2. erklärt, wie eine mathematische Funktion registriert wird.

---

[13] Variablen können hier Elemente oder Funktionen sein, z. B. Punkt N oder Funktion sin().

[14] vgl. Kapitel 2.6.1

[15] Ein Slider ist ein Schieberegler, an dessen numerischen Wert wir interessiert sind.

[16] Das Zeichen `'^'` steht in JavaScript für das exklusive Oder anstatt der Exponentialfunktion.

2.5. Grammatikalische Analyse der Funktionen

In der Mathematik sind oft neben ‚einfachen' Zahlenwerten auch kompliziertere, verschachtelte Ausdrücke von Nöten um das gewünschte Ergebnis zu visualisieren. Will man zum Beispiel einen Punkt anhand einer Funktion positionieren, kann ein Ausdruck wie dieser entstehen:

```
A = (sin(2 a), a^(2));
```

JavaScript akzeptiert die Angabe eines solchen Werts nicht. Daher muss dieser Ausdruck für JavaScript so umgesetzt werden, dass dieser am Ende folgendermaßen aussieht:

```
board.ggbElements['A'].x =
    function(){ return Math.sin(2*board.ggbElements['a']); };
board.ggbElements['A'].y =
    function(){ return Math.pow(board.ggbElements['a'], 2); };
```

Um dies zu bewerkstelligen wird im der Methode `ggbParse` des Geogebra-Readers ein mit JS/CC[17] erzeugter Parser eingesetzt.

Dieses mächtige Tool verbindet eine lexikalische Analyse mit einer kontextfreien Grammatik und erlaubt damit reguläre Ausdrücke wie zum Beispiel

```
'[0-9]+\.[0-9]*|[0-9]*\.[0-9]+'
```

für eine Floatzahl in einen Funktionsaufruf wie:

```
JXG.GeogebraReader.ggbMatch('float', %match);
```

umzusetzen, sodass JavaScript der Wert und passende Datentyp geliefert wird. So kann auch problemlos, die eigentlich schwierige Klammerung

```
'(' e ')'
```

in das Verwendbare

```
JXG.GeogebraReader.ggbAct('bra', %2);
```

formiert werden, sodass die innere Expression, übergeben mit `'%2'`, mathematisch ausgewertet werden kann. Am Ende wird das Ergebnis korrekt dargestellt und zur Registrierung zurückgegeben. Die aufgerufene Methode `ggbAct` unterscheidet je nach Fall[18], welcher Wert zurückgegeben oder interpretiert wird.

---

[17] vgl. http://jscc.jmksf.com/

[18] Der Fall wird anhand der zuerst übergebenen Parameters unterschieden.

In der gesamten Grammatik[19] finden sich dadurch die Hauptteile für lexikalische Analyse. Hier werden die unterschiedlichen Datentypen unterschieden:

```
'\('
'\)'
'[0-9]+'                        INT
'[0-9]+\.[0-9]*|[0-9]*\.[0-9]+' FLOAT
'\_\_[a-zA-Z0-9]+'              PARAM
'[a-zA-Z]+(\_*[a-zA-Z0-9]+)*'   VAR
'\&[a-zA-Z]+\;'                 HTML
'\"[a-zA-Z0-9äüöß\=\.\ ]*\"'    STRING
;
```

Abbildung 7: Parsergrammatik (non-associative tokens)

Daneben stellt die kontextfreie Grammatik dem Parser die Operationen mit den obigen Datentypen:

```
e:        '(' e ',' e ')'       [* %% = JXG.GeogebraReader.ggbAct('coord', %2, %4, element); *]
        | e '<=' e              [* %% = JXG.GeogebraReader.ggbAct('le', %1, %3); *]
        | e '>=' e              [* %% = JXG.GeogebraReader.ggbAct('ge', %1, %3); *]
        | e '==' e              [* %% = JXG.GeogebraReader.ggbAct('eq', %1, %3); *]
        | e '!=' e              [* %% = JXG.GeogebraReader.ggbAct('neq', %1, %3); *]
        | e '<' e               [* %% = JXG.GeogebraReader.ggbAct('lt', %1, %3); *]
        | e '>' e               [* %% = JXG.GeogebraReader.ggbAct('gt', %1, %3); *]
        | e '+' e               [* %% = JXG.GeogebraReader.ggbAct('add', %1, %3); *]
        | e '-' e               [* %% = JXG.GeogebraReader.ggbAct('sub', %1, %3); *]
        | '!' e                 [* %% = JXG.GeogebraReader.ggbAct('neg', %2); *]
        | e '^' e               [* %% = JXG.GeogebraReader.ggbAct('pow', %1, %3); *]
        | e '||' e              [* %% = JXG.GeogebraReader.ggbAct('or', %1, %3); *]
        | e '&&' e              [* %% = JXG.GeogebraReader.ggbAct('and', %1, %3); *]
        | e '*' e               [* %% = JXG.GeogebraReader.ggbAct('mul', %1, %3); *]
        | e '/' e               [* %% = JXG.GeogebraReader.ggbAct('div', %1, %3); *]
        | '-' e &'*'            [* %% = JXG.GeogebraReader.ggbAct('negmult', %2); *]
        | '(' e ')'             [* %% = JXG.GeogebraReader.ggbAct('bra', %2); *]
        | STRING '+' e          [* %% = JXG.GeogebraReader.ggbAct('string', %1, %3); *]
        | INT                   [* %% = JXG.GeogebraReader.ggbAct('int', %1); *]
        | FLOAT                 [* %% = JXG.GeogebraReader.ggbAct('float', %1); *]
        | PARAM                 [* %% = JXG.GeogebraReader.ggbAct('param', %1); *]
        | HTML                  [* %% = JXG.GeogebraReader.ggbAct('html', %1); *]
        | STRING                [* %% = JXG.GeogebraReader.ggbAct('string', %1); *]
        | VAR '(' e ')'         [* %% = JXG.GeogebraReader.ggbAct('var', %1, %3); *]
        | VAR                   [* %% = JXG.GeogebraReader.ggbAct('var', %1); *]
        ;
```

Abbildung 8: Parsergrammatik (grammar specification)

Dabei ist wichtig zu erwähnen, dass die Operatoren anhand ihrer Auflistungsreihenfolge absteigend höhere Priorität bekommen. Das ist vor allem bei so genannten Punkt-vor-Strich-Rechnungen und äquivalenter mathematischer Gewichtung nötig:

```
< '\+'  < '\*'  < '^';   < '\,'; < '<=' < '!';  < '\|\|'
  '\-';   '/';                     '>='           '\&\&';
                                   '=='
                                   '!='
                                   '<'
                                   '>';
```

Abbildung 9: Parsergrammatik (Operanden)

[19] vgl. angehängte CD-Rom 6.3.

Die Ausdrücke der Grammatik werden vom Parser betrachtet und im jeweils passenden Fall angewandt. So wird die Suche verfeinert und eine ganze Zahl, zum Beispiel 3, wird nicht durch ein nicht treffendes Float, wie zum Beispiel die Zahl '3.2', sondern durch den Ausdruck für Integer gefunden.

Nach diesem Prinzip werden die kompliziertesten Funktionen in immer kleinere, zur Grammatik passende, Teile zerlegt und können schnell ausgelesen werden. Die in 2.4. angesprochene Funktion

```
function(__x) = __x^(3) - 3*__x^(2) + a
```

wird mit mit `ggbParse` geparset und ergibt nach dem Ersetzen der einzelnen Bestandteile in der JXG.GeogebraReader.ggbAct-Funktion, nun:

```
function(__x) = Math.pow(__x, 3) - 3*Math.pow(__x, 2) + a
```

Im Laufe des Ausleseprozesses wird für jeden Bestandteil ein String erzeugt, der bei Rückgabe in den umschließenden Ausdruck zurückgegeben wird. Damit ergibt sich für: `x^2 / 6 - 2 + 8 * sin(x)` folgender „Auslesebaum“:

*Abbildung 10: Expressiontree für x^2 / 6 - 2 + 8 * sin(x)*

Um eine reibungsfreie Integration zu ermöglichen, gibt es unter anderem die Besonderheit, dass Elemente unterschiedlichen Datentyps addiert, subtrahiert, multipliziert oder dividiert werden müssen.
Das wird im GeogebraReader so bewerkstelligt, dass eine Überprüfung des Datentyps stattfindet und entsprechend reagiert wird.
Mit folgendem Codeausschnitt kann eine Addition beliebiger Typen stattfinden:

```
case 'add':
  if(   ((typeof v1 == 'string' || typeof v1 == 'number') && (typeof v2 == 'string' || typeof v2 == 'number'))
        &&
        (v1.match(/JXG/) && v2.match(/JXG/))
    ) {
    return [v1+'.X() + '+v2+'.X()', v1+'.Y() + '+v2+'.Y()'];
  } else if((typeof v1 == 'string' || typeof v1 == 'number') && typeof v2 == 'object') {
    return [v1+'.X() + '+v2[0], v1+'.Y() + '+v2[1]];
  } else if(typeof v1 == 'object' && (typeof v2 == 'string' || typeof v2 == 'number')) {
    return [v1[0]+' + '+v2+'.X()', v1[1]+' + '+v2+'.Y()'];
  } else if(typeof v1 == 'object' && typeof v2 == 'object') {
    return [v1[0]+' + '+v2[0], v1[1]+' + '+v2[1]];
  } else {
    return v1 +'+'+ v2;
  }
break;
```

Abbildung 11: ggbAct (add)

Somit kann der Parser die beiden Summanden übergeben ohne sich um den jeweiligen Typ kümmern zu müssen. Das ermöglicht eine Addition zu Punkt N, mit dem Koordinatenpaar (x(T), 0), und (0, k) wobei k hier für eine berechnete Steigung steht.
Als Ergebnis bekommt der neue Punkt die Koordinaten:

((x(T) + 0), (0 + k))

Einen weiteren Spezialfall stellt das Auslesen von Variablen dar. In der Grammatik sind dafür zwei Fälle vorgesehen.

```
1.VAR
2.VAR '(' e ')'
```

Der Fall 1 trifft auf alle Elementzugriffe, zum Beispiel `'N'` oder `'a'`, zu und gibt die jeweilige Referenz auf das Element zurück, hier:

```
JXG.getReference(board, 'N')
```

Besondere Objekte wie Slider, Polygone oder Label werden berücksichtigt und im Parser jeweils auf die entsprechende Methode verwiesen um an den gewünschten Wert zu kommen, wie zum Beispiel:

```
JXG.getReference(board, 'a').Value()
```

Im Fall 2 steht die Variable für die gewünschte Funktion und muss anders als in Fall 1 behandelt werden. Der erste Parameter symbolisiert den Funktionsnamen, während der zweite den Funktionsparameter beinhaltet. Dies ist wichtig für die Auflösung von beispielsweise `y(T)` oder der Umsetzung von `sin(x)` in `Math.sin(x)`.

Der dazugehörige Codeabschnitt findet sich im GeogebraReader in der Funktion `ggbAct` im Fall '`var`':

```
case 'var':
  if(v2) {
    switch(v1.toLowerCase()) {
        case 'x':
            return v2 +'.X()';
        break;
        case 'y':
            return v2 +'.Y()';
        break;
        default:
            return 'Math.'+v1.toLowerCase()+'('+ v2 +')';
        break;
    }
  } else {
    if(typeof JXG.GeogebraReader.board.ggbElements[v1] == 'undefined' || JXG.GeogebraReader.board.ggbElements[v1] == '') {
        var input = JXG.GeogebraReader.getElement(v1);
        JXG.GeogebraReader.board.ggbElements[v1] = JXG.GeogebraReader.writeElement(JXG.GeogebraReader.board, input);
        JXG.GeogebraReader.debug("regged: "+ v1 +" (id: "+ JXG.GeogebraReader.board.ggbElements[v1].id +")");
    }

    var a = JXG.GeogebraReader.board.ggbElements[v1];
    if(typeof a.Value != 'undefined') {
      return 'JXG.getReference(JXG.GeogebraReader.board, "'+ v1 +'").Value()';
    } else if (typeof a.Area != 'undefined') {
      return 'JXG.getReference(JXG.GeogebraReader.board, "'+ v1 +'").Area();';
    } else if (typeof a.plaintextStr != 'undefined') {
      return '1.0*JXG.getReference(JXG.GeogebraReader.board, "'+ v1 +'").plaintextStr;';
    } else {
      return 'JXG.getReference(JXG.GeogebraReader.board, "'+ v1 +'")';
    }
  }
break;
```

Abbildung 12: ggbAct (var)

2.6. JavaScript-Besonderheit

2.6.1. Anonyme Funktionen

Eine der größten Schwierigkeiten bei dem Auslesen, der Verwaltung und Aktualisierung der Elemente ist der Zugriff auf die aktuellen numerischen Werte einer Funktion. Um Speicherzugriffsprobleme und keine oder falsche Aktualisierungen zu vermeiden, wird im GeogebraReader stark auf das Konzept der anonymen Funktionen gesetzt. Diese Funktionen sind alleinig bei JavaScript implementiert und in anderen Programmiersprachen fremd. Das mächtige Konzept erlaubt es, dass Funktionen wie normale Werte gespeichert werden. Der Unterschied ist, dass nicht der aktuelle Wert dort hinterlegt wird, sondern der jeweils aktuelle Wert geliefert wird. Das Prinzip wird in folgendem Beispiel verdeutlicht:

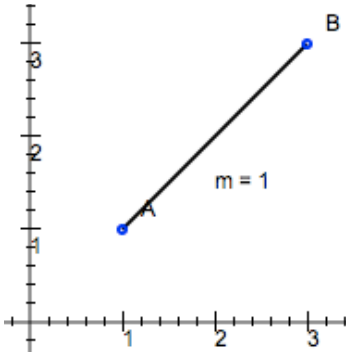

Abbildung 13: Steigung

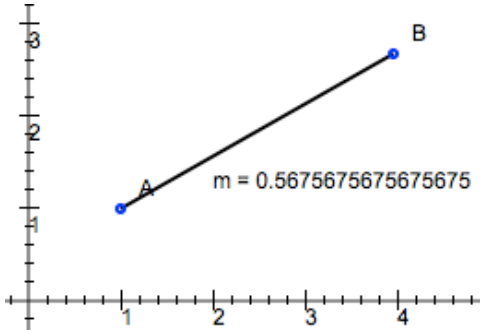

Abbildung 14: Steigung

```
var slope = board.createElement('text', [
            function(){ return m.X(); }, function() { return m.Y(); },
            function(){ return "m = "+ function() { return i.Y()-l1.point1.Y();}(); }
            ]);
```

Abbildung 15: writeElement (slope)

Das Konzept der anonymen Funktionen und deren Verwendung wird in Abbildung 11 deutlich. Würde man anstatt der Funktionen die jeweiligen Koordinaten des Mittelpunkts direkt verwenden, also

```
board.createElement('text',
      [m.X(), m.Y(), "m = "+ i.Y()-l1.point1.Y()]);
```

dann würden beim Erzeugen die jeweils aktuellen Zahlenwerte geschrieben werden. Von Abbildung 9 zu 10 wäre dann bei dem String 'm = 1' keine Änderung zu sehen, da die Koordinaten und Berechnung des Steigungswerts anhand der ursprünglichen, statischen Werte berechnet werden. Mit dem JavaScript-Funktionskonzept wird diesem Problem Abhilfe geschaffen und es

erlaubt den Zugriff auf die aktuell gültigen Werte und somit die Interaktion mit dem Benutzer. Wird einer der Punkte aktualisiert, gibt die Funktion den entsprechenden neuen Wert zurück anstatt den statischen Zahlenwert zum Zeitpunkt der Erzeugung zu verwenden.

2.6.2. Dynamische Funktionen

Neben den gerade beleuchteten anonymen Funktionen gibt es in JavaScript auch die Möglichkeit dynamische Funktionen zu erzeugen. Damit kann je nach Bedarf eine neue Funktion registriert werden deren Funktionsrumpf und Parameteranzahl verändert werden kann.
Für den GeogebraReader ist das wichtig, da bei mathematischen Funktionen mehr als ein Funktionsparameter definiert werden kann, im Vorhinein aber nicht klar ist, wie viele Parameter nötig sind.
Daher wird mit folgendem Code eine jeweils passende Funktion mit der benötigten Parameteranzahl erzeugt.

```
var l = func.length - 1;
if (l==1)
  f = board.createElement('functiongraph', [Function(func[0], func[1])]);
else if (l==2)
  f = board.createElement('functiongraph', [Function(func[0], func[1], func[2])]);
else if (l==3)
  f = board.createElement('functiongraph', [Function(func[0], func[1], func[2], func[3])]);
else if (l==4)
  f = board.createElement('functiongraph', [Function(func[0], func[1], func[2], func[3], func[4])]);
```

Abbildung 16: writeElement (func)

Dabei steht der letzte Parameter für den Funktionsrumpf und bringt den Code als String mit sich. In Abbildung 12 könnte also auch vereinfacht folgende Exponentialfunktion stehen:

```
f = board.createElement('functiongraph',
      [ Function("x", "y", "Math.pow(x, y)"); ]);
```

Das in 2.4. und 2.5. betrachtete Beispiel

```
function(__x) = Math.pow(__x, 3) - 3*Math.pow(__x, 2) + a
```

ergibt dann hier letztendlich den folgenden Aufruf:

```
f = board.createElement('functiongraph', [
  Function("__x", "Math.pow(__x, 3) - 3*Math.pow(__x, 2) +
                JXG.getReference(board, 'a').Value()");
]);
```

4. Zusammenfassung und Ausblick

Abschließend zu dieser Arbeit und rückblickend auf die Entwicklungsphase des GeogebraReaders komme ich zu dem Schluss, dass sich die Zeit und Mühen mehr als gelohnt haben. Ich durfte sowohl die Einfachheit, gepaart mit der mächtigen Speicherverwaltung, die selbst den höchsten Anforderungen stand hält, kennenlernen als auch erfahren, wie viel Spaß man in einem Open Source Projekt haben kann. Die meisten Entwickler freier Software stehen mit Rat und Tat zur Seite, wenn man mal sprichwörtlich auf dem Schlauch steht und einen kleinen Tipp benötigt oder eine Inspiration braucht. Das hat mich persönlich weitergebracht und so konnte ich viel Erfahrung sammeln. Dafür bedanke ich mich bei jedem der Beteiligten.
Die Arbeit hat Früchte getragen, was sich auch dadurch zeigt, dass ich weiterhin an der Entwicklung des GeogebraReaders mithelfen will und zusätzlich noch weitere Plugins zur einfachen Verwendung von JSXGraph schreiben werde.

5. Erklärung nach §30 Abs 6 LPO 1

Ich versichere, dass ich die schriftliche Hausarbeit selbständig verfasst und keine anderen Hilftsmittel als die angegebenen benutzt habe. Die Stellen der Arbeti, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, wurden von mir unter Angabe der Quelle als Entlehnung kenntlich gemacht. Dies gilt ebenso für Zeichnungen und bildliche Darstellungen.

_____

Bayreuth, den 23. Dezember 2009

6. Anhang

6.1. Geogebra-XML-Datei

6.2. JXG.GeogebraReader

6.3. Parsergrammatik