Ernst-Moritz-Arndt-Universität Greifswald
Fachrichtung Mathematik/Informatik

Lindenmayer-Systeme Algorithmen

Wissenschaftliche Arbeit zur Erlangung
des akademischen Grades

„Diplommathematiker“

an der Fachrichtung Mathematik/Informatik
der Ernst-Moritz-Arndt-Universität Greifswald

vorgelegt von
Volker Pohlers

im Auftrag von
Prof. Dr. habil. Ch. Bandt

Greifswald, den 22. Juli 1993

Haftungsausschluß:
Es wird keinerlei Haftung übernommen für irgendwelche Schäden, die aus der Benutzung der in dieser Arbeit beschriebenen Programme folgen.

Diese PDF-Datei wurde aus der ursprünglichen unter emTEX erstellten Diplomarbeit erzeugt. Dabei erfolgte eine Umstellung von LaTeX 2.09 auf LaTeX2e. Die Bilder mußten anders als im Original eingefügt werden: Statt der Konvertierung mit `bm2font` wurden die Bilder als PNG eingebunden, die Vektorgrafiken wurden statt der Verwendung von emTEX-Specials nach Postscript und dann nach PDF konvertiert und eingebunden. Das originale Paket `emsymb` für die Tastensymbole konnte nicht mehr verwendet werden, die Sondertasten im Anhang B.1 sehen leicht anders aus. Durch die Weiterentwicklung von TEX hat sich auch das Layout geringfügig geändert, die Seitennummerierung stimmt noch vollständig mit dem Original überein. Am Inhalt wurde nichts verändert.

Volker Pohlers, Januar 2002

Diese Diplomarbeit wurde mit dem Textsatzsystem TEX erstellt.

Inhaltsverzeichnis

1. Vorwort

Faszination Computergraphik — Bilder ungeahnter Schönheit, fantastischen Detailreichtums, mathematischer Strenge. Faszination Computergraphik — auch eine Herausforderung für Informatiker, Mathematiker, Programmierer. Hinter einer Computergraphik steckt ein nicht zu unterschätzender Aufwand: Modellierung der Bildkomponenten, Bildkomposition, Farbgebung, Perspektive, Raytracing. Die Berechnung von Computergraphiken stellt hohe Anforderungen an die Rechenleistung und an den Arbeitsspeicher der Computer. Der Versuch, auf einem heutigen PC fotorealistische Computergraphiken in erträglicher Zeit zu berechnen, ist damit von vornherein eingeschränkt. Dennoch kann man auch mit einem PC erstaunliche Resultate erzielen.

Ein interessantes Teilgebiet der Computergraphik ist das Zeichnen von Pflanzen. *Aristid Lindenmayer* schuf 1968 eine mathematische Beschreibung von biologischen Objekten wie Zellen, Pflanzen und deren Wachstumsprozessen. Dazu diente ihm eine eigene Beschreibungssprache, die eine kompakte und den Prozessen nahestehende Beschreibung zuließ. Diese nach ihm benannten *Lindenmayersysteme*, kurz L-Systeme, bestehen aus einem Startsymbol und mehreren Ersetzungsregeln für Symbole. Ähnlich wie bei formalen Sprachen werden die Ersetzungsregeln auf das Startsymbol und dann mehrfach wiederholt auf die sich ergebenden Ersetzungen angewandt. Am Ende dieses Prozesses ergibt sich ein String aus Symbolen.

1984 begann *Alvy Ray Smith* mit dem umgekehrten Prozeß, der Synthese realistischer Bilder von Pflanzen mittels L-Systemen durch eine *graphische Interpretation* des Symbolstrings.

Durch leistungsstarke Computer ist es möglich geworden, komplexe L-Systeme von Pflanzenmodellen in erstaunlicher Qualität zu visualisieren. Großen Anteil an dieser Entwicklung hat auch *Przemyslaw Prusinkiewicz*. In seinen Arbeiten [11, 12] finden sich Beispiele realitätsnaher L-Systeme.

Mit L-Systemen können auch auf einfache Weise klassische *Fraktale* wie die Kochkurve und die Drachenkurve erzeugt werden.[1] Weitere Beispiele solcher Fraktale, die ohne Aufwand als L-System formuliert werden können, sind in Mandelbrots ‘fraktaler Geometrie der Natur’ [6] zu finden.

[1] Es können sogar alle selbstähnlichen Mengen durch L-Systeme approximiert werden [4].

In dieser Arbeit werden nun Algorithmen zur Erzeugung und Interpretation von L-Systemen vorgestellt. Schwerpunkt ist dabei mein Programm LSC (Lindenmayer System Compiler), das das breite Spektrum kontextsensitiver parametrischer bedingter L-Systeme (ILS) generieren und dreidimensional graphisch darstellen kann. Speziell können mit diesem Programm *fast alle* Beispiele aus Prusinkiewicz' Werken durchgerechnet werden!

Mit einer im Textmodus arbeitende Variante des LSC können L-Systeme formal abstrakt untersucht werden. Einen Einblick gibt hierzu Prusinkiewicz in [13].

Aufgrund der Komplexität des Themas wird auf grundlegende theoretische Aspekte der L-Systeme in dieser Arbeit nicht eingegangen, sie sind aber für das Verständnis des hier beschriebenen unumgänglich. Deshalb ist diese Arbeit nur im Zusammenhang mit [4, 12] zu sehen. Begriffe wie Ableitung, kontextsensitives parametrisches bedingtes L-System[2], Turtleinterpretation und andere werden also vorausgesetzt und hier nicht nochmals definiert.

Zum Verständnis des Abschnitts 3.2 sind Grundbegriffe formaler Sprachen und des Compilerbaus, etwa Scanner und Parser, sowie die damit zusammenhängenden informationstheoretischen Prinzipien unumgänglich. Als Basis wird [1] empfohlen.

[2] Der Begriff des L-Systems wird in dieser Arbeit in mehrfacher Bedeutung verwendet. So steht er sowohl für eine Grammatik wie auch für die Interpretation der Ableitung (also im allgemeinen das graphische Bild). Gelegentlich wird mit L-System auch die Ableitung selbst bezeichnet. Die jeweilige Bedeutung wird aber aus dem umgebenden Text ersichtlich.

2. Erzeugung von L-Systemen

In diesem Kapitel werden Programme und die hinter ihnen stehenden grundlegenden Algorithmen zur Bildung von L-Systemen vorgestellt. Außerdem wird eine Methode dargelegt, die eine schnelle Erzeugung parametrischer L-Systeme ermöglicht.

2.1. Bekannte Programme zur Generierung von L-Systemen

Natürlich bin ich nicht der erste, der ein Programm zur Erzeugung und Interpretation von L-Systemen vorlegt. Auf IBM-kompatiblen PC's existiert mittlerweile eine Reihe von Programmen, die zur Generierung von L-Systemen genutzt werden können. Im folgenden werden die bekanntesten von ihnen zusammen mit einem Beispiel zur Demonstration ihrer Eingabesprache vorgestellt.

2.1.1. FractInt

FractInt ist *das* führende Fraktalprogramm auf dem PC-Sektor schlechthin. Es erzeugt eine fantastische Vielfalt von Fraktaltypen und überzeugt durch seine Geschwindigkeit. FractInt ist Freeware. In diesem Kapitel wird aber nur der Programmteil zur Erzeugung von L-Systemen betrachtet:

Version: WinFrac 17.26 vom 16. Juni 1992

Autor: Adrian Mariano, Nicolas Wilt[1]

Sprachumfang:

$+$	Drehung um Winkel δ
$-$	Drehung um Winkel $-\delta$
F	Zeichne Linie der Länge l
G	Bewegung der Länge l

[1] Sie sind nicht die einzigen Autoren von FractInt, haben aber die L-Systeme beigesteuert. Für FractInt zeigt sich ein ganzes Heer von Mitautoren verantwortlich.

|	Drehung um 180°
[	Aktuelle Turtleposition sichern
]	Turtleposition wiederherstellen
D	Draw vorwärts (zur allgemeineren Verwendung gegenüber F)
M	Move vorwärts (zur allgemeineren Verwendung gegenüber G)
$\backslash nn$	Erhöhe Winkel um $nn°$
$/nn$	Vermindere Winkel um $nn°$
$@nn$	Multipliziere Länge l mit nn. Zusätzlich kann noch ein I für Invers und Q für Quadratwurzel mit einbezogen werden, etwa *@IQ3* für $l \rightarrow \frac{1}{\sqrt{3}}l$.
Cnn	Stellt die Zeichenfarbe nn ein[2]
$>nn$	Erhöht Zeichenfarbe um nn
$<nn$	Vermindert die Zeichenfarbe um nn
!	Vertauscht die Bedeutung von $+$, $-$ sowie von \, /

Besonderheiten: Der Drehwinkel δ wird als $\frac{360°}{n}$ mit $0 < n < 50$ angegeben. Die bei einigen Befehlen auftretenden Zahlen nn sind in Dezimalschreibweise anzugeben. Bei den Befehlen wird zwischen Groß- und Kleinschreibung *nicht* unterschieden.

```
PentaPlexity {
; Manual construction by Roger Penrose
; as a prelude to his development of
; the famous Penrose tiles (the kites
; and darts) that tile the plane only
; non-periodically. Translated first
; to a "dragon curve" and finally to
; an L-System by Joe Saverino.
  Angle 10
  Axiom F++F++F++F++F
  F=F++F++F|F-F++F
  }
```

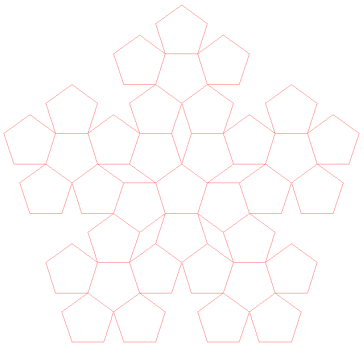


Abbildung 2.1.: PentaPlexity, mit FractInt erzeugt

Es können nur zweidimensionale L-Systeme generiert werden. In FractInt sind nicht beliebige Winkel wählbar. Durch die nachträgliche Veränderbarkeit sowohl von δ als auch von l kann dennoch eine große Zahl von L-Systemen generiert werden, etwa Penrose-Muster.

[2] Die Farbwerte werden als 24-bit RGB-Werte angesehen. So steht 000000h für schwarz, 0000FFh für rot, 00FF00h für gelb und FF0000h für blau.

2.1.2. Fractal Paint

Mit Fractal Paint steht ein Vektorgrafikprogramm zur Verfügung, was außer den 'normalen' Zeichenwerkzeugen wie Ellipse, Rechteck etc. auch L-Systeme verwendet. Diese können frei definiert und sehr komfortabel über einen 'Werkzeugkasten' genutzt werden. Fractal Paint ist Shareware.

Version: Fractal Paint 1.06 vom 12. Mai 1991

Autor: International Computer Smiths Canada

Sprachumfang:

$+$	Drehung um Winkel δ
$-$	Drehung um Winkel $-\delta$
F	Zeichne Linie der Länge l
G	Bewegung der Länge l
[	Aktuelle Turtleposition sichern
]	Turtleposition wiederherstellen
/	Halbierung der Länge l
@nn	Multipliziere Länge l mit nn. Zusätzlich kann noch ein I für Invers und Q für Quadratwurzel mit einbezogen werden, etwa *@IQ3* für $l \rightarrow \frac{1}{\sqrt{3}}l$.
Cnn	Stelle die Zeichenfarbe nn ein[3]
$>nn$	Erhöhe Zeichenfarbe um nn
$<nn$	Vermindere die Zeichenfarbe um nn
!	Vertausche die Bedeutung von $+$ und $-$

Besonderheiten: Die bei einigen Befehlen auftretenden Zahlen nn sind in Dezimalschreibweise anzugeben. Bei den Befehlen wird zwischen Groß- und Kleinschreibung *nicht* unterschieden.

Auch mit Fractal Paint können nur zweidimensionale L-Systeme erzeugt werden. Dank des fast gleichen Sprachumfangs zu FractInt können dessen Beispiele relativ leicht übernommen werden.

Fractal Paint ist beim Aufbau und Zeichnen der Fraktale sehr schnell. Durch Organisation des Programms als Vektorgrafikprogramm können bereits plazierte Fraktale sehr leicht verändert (verzerrt, verschoben ...) werden; die Druckbildauflösung ist wesentlich höher als die des Bildschirms. Leider erlaubt Fractal Paint *keinen* Export der Grafiken etwa als HPGL–File.

[3] Die Farbwerte werden als 24-bit RGB-Werte angesehen. So steht 000000h für schwarz, 0000FFh für rot, 00FF00h für gelb und FF0000h für blau.

```
O1:
LSYSDEF KOCH2 30 F---F---F---F 1 F -F+++F---F+
O2:
MOVETO 05437 06462
LSTYLE 0 0 0 0 0
TURNTO 60
LSYS KOCH2 2 219
O3:
MOVETO 07564 07780
LSTYLE 0 0 0 0 0
TURNTO 10
LSYS KOCH2 4 54
O4:
MOVETO 05094 08659
LSTYLE 0 0 0 0 0
PATTERN 1 0 0 1
RECT 09262 05187
O5:
MOVETO 07290 05757
TSTYLE 10 0 0 0 0 0 0 48 2 0 0 0 255 255 255 "Courier"
TEXT "Fractal Paint"
```

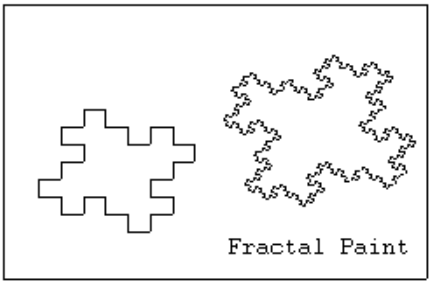


Abbildung 2.2.: Kochkurven, mit Fractal Paint erzeugt

2.1.3. L-System von R. Gerike

Roman Gerike stellte im Rahmen eines Crash-Kurses in C++ ein Programmpaket zur Generierung von L-Systemen vor [2]. Stellvertretend für eine Reihe kleinerer Programme soll dieses hier erwähnt werden, da es durch den Artikel in der c't einem größeren Interessentenkreis zugänglich wurde.

Version: L-SYSTEM.CPP 1.0 vom April 1992

Autor: Roman Gerike

Sprachumfang:

\	H-Drehung um Winkel δ
/	H-Drehung um Winkel $-\delta$
&	L-Drehung um Winkel δ
∧	L-Drehung um Winkel $-\delta$
+	U-Drehung um Winkel δ
−	U-Drehung um Winkel $-\delta$

|	U-Drehung um 180°
F, L, R	Zeichne Linie der Länge l
f	Bewegung der Länge l
[	Aktuelle Turtleposition sichern
]	Turtleposition wiederherstellen
\	Erhöhe Zeichenfarbe

Besonderheiten: L-System ist sehr effizient (und C-typisch sehr kryptisch) geschrieben. So wird der abgeleitete String nicht komplett im Speicher gehalten, sondern aus Pointern auf die gegebenen Regeln konstruiert. Bei langen Regeln ergibt sich dadurch eine Einsparung an Speicherplatz.[4]

Leider werden die Symbole R, L ebenfalls von der Turtlegrafik interpretiert, so daß bei der Übernahme anderer L-Systeme auf deren Verwendung besonders geachtet werden muß.

Die Größe des erzeugten L-Systems wird nicht automatisch an das Bildschirmformat angepaßt, der Nutzer hat hier selbst sinnvolle Werte für die Zeichenlänge l vorzugeben.

Dieses Programm erzeugt als einziges der uns bekannten dreidimensionale L-Systeme. Komplexere Strukturen, etwa parametrische OL-Systeme, können auch hier *nicht* generiert werden. Das ist aber auch nicht das Ziel dieses Programmierkurses.

2.1.4. OL von R. Scholl

Mit [17] wurde der Allgemeinheit erstmals ein Programmpaket vorgelegt, mit dem photorealistische Landschaften via Computer erzeugt werden können. Wohl als Zugabe findet sich hier ein Programm zur Erzeugung von L-Systemen, das bei weitem nicht so ausgereift ist wie die restlichen Programme. Da es mit dem Buch allerdings eine gewisse Verbreitung erreicht haben dürfte, soll es hier ebenfalls analysiert werden.

Version: OL.PAS V 2.1b vom 1. Februar 1991

Autor: Reinhard Scholl

Sprachumfang:

+	Drehung um Winkel δ
−	Drehung um Winkel $-\delta$

[4] Eine Einsparung erfolgt, wenn die Regeln im Durchschnitt länger als 4 Byte sind, da ein Pointer 4 Byte Speicher belegt und die vorletzte Ableitung aus Pointern bestehend den Zielstring ergibt.

F	Zeichne Linie der Länge l
f	Bewegung der Länge l
[	Aktuelle Turtleposition sichern
]	Turtleposition wiederherstellen

Besonderheiten: Um realistischere Pflanzenmodelle zu kreiern, kann zusätzlich zum Winkel δ ein Streuungswinkel γ angegeben werden. Der Winkel δ wird dann normalverteilt mit γ gestreut. Diese Art Zufall ist bei Prusinkiewicz *nicht* zu finden! Leider wurde im Programm die Standardersetzung einer nicht näher definierten Regel $(r \rightarrow r)$ unterschlagen.

Durch die Variation von δ werden mit einfachsten Mitteln natürlicher wirkende L-Systeme erzeugt. Der Algorithmus zur Berechnung und Interpretation des L-Systems ist identisch mit [9]. Durch rekursive Berechnung der Ableitung ist er nicht auf kontextsensitive L-Systeme erweiterbar.

2.1.5. pfg von Prusinkiewicz

In [11] wird eine einfache Programmimplementation des 'plant and fractal generators' für einen Macintosh-Computer vorgestellt. Mit dieser Version können bereits zweidimensionale kontextsensitive L-Systeme erzeugt und dargestellt werden.

Version: Plant and Fractal Generator pfg ©1988

Autor: Przemyslaw Prusinkiewicz

Sprachumfang:

$+$	Z-Drehung um Winkel δ
$-$	Z-Drehung um Winkel $-\delta$
F	Zeichne Linie der Länge l
f	Bewegung der Länge l
[	Aktuelle Turtleposition sichern
]	Turtleposition wiederherstellen
$<$	Abschluß des linken Kontexts einer Regel
$>$	Beginn des rechten Kontexts einer Regel
$*$	Leerer String

Besonderheiten: Der Drehwinkel δ wird als $\frac{360°}{n}$ mit $0 < n < 40$ angegeben. Die Eingabedatei muß streng gegliedert sein.

Das Programm pfg ist in Standard-C geschrieben und konnte leicht auf einem IBM-PC implementiert werden. Es verarbeitet als einziges der hier betrachteten Programme kontextsensitive L-Systeme und verwendet (daher) zur Generierung der Ableitung ein nichtrekursives Verfahren.

```
derivation length: 30
angle factor: 16
scale factor: 100
axiom: F1F1F1
ignore: +-F
0 < 0 > 0 --> 0
0 < 0 > 1 --> 1[-F1F1]
0 < 1 > 0 --> 1
0 < 1 > 1 --> 1
1 < 0 > 0 --> 0
1 < 0 > 1 --> 1F1
1 < 1 > 0 --> 1
1 < 1 > 1 --> 0
* < + > * --> -
* < - > * --> +
```

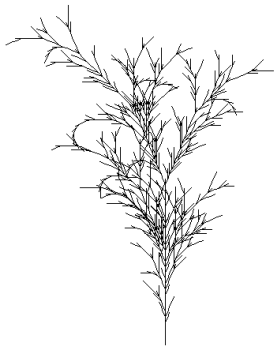


Abbildung 2.3.: kontextsensitives L-System, mit pfg erzeugt

2.1.6. Zusammenfassung

Alle betrachteten Programme erlauben — mehr oder weniger komfortabel — die Erzeugung einfacher zweidimensionaler L-Systeme. Dabei ist der abgeleitete String im allgemeinen dem Benutzer verborgen und tritt nur als Turtleinterpretation in Erscheinung. FractInt, OL von R.Scholl sowie pfg legen die Zeichenlänge l dabei so fest, daß das L-System den Bildschirmbereich voll ausnutzt. In Fractal Paint kann diese Vergrößerung interaktiv leicht selbst vorgenommen werden.

R. Gerikes Programm ist das einzige, welches auch dreidimensionale L-Systeme interpretieren kann; pfg kann als einziges kontextsensive L-Systeme erzeugen. Alle 5 Programme kennen *keine* wie in [11] allgemein definierten parametrischen L-Systeme. Ebenso fehlt bei all diesen Programmen die Erzeugung gefüllter Polygone und weiterer gestalterischer Elemente, um fotorealistische Bilder zu generieren. Auch ist die Bedienbarkeit aller Programme recht unterschiedlich.

Ein Programm, daß das breite Spektrum von L-System–Klassen abdeckt und eine leichte Untersuchung gegebener Beispiele ermöglicht, fehlte bisher und ist (hoffentlich) durch mein Programm LSC gegeben.

2.2. Von der Beschreibung zur Interpretation

Welche Prinzipien und Algorithmen wurden in den Programmen zur Erzeugung der Bilder verwendet? Dieser Abschnitt soll die Verarbeitung eines L-Systems von der Textdatei bis zur Bildschirmdarstellung anhand allgemeiner Algorithmen erläutern.[5]

Für die praktische Arbeit am Computer ist es zweckmäßig, wenn das zu interpretierende L-System in leicht editierbarer Form vorliegt, also etwa als Textdatei. In dieser stehen dann alle zur Berechnung und oft auch zur Interpretation nötigen Informationen:

- Axiom und Regeln sowie die Ableitungstiefe
- bei der Kontextsuche zu ignorierende Symbole

sowie eventuell

- die Winkeländerung δ
- Konstanten zur vereinfachten Variation von L-Systemen
- Interpretationsregeln von Symbolen (surfaces)
- Farbgebung, Linienstärke, Füllmuster und weitere graphische Gestaltungsmittel

Die Informationen dieser Datei müssen als erstes in eine programmäßig leichter zu handhabende Form gebracht werden (etwa eine Ablage der Ableitungstiefe in einer Variablen und der Regeln in Zeichenketten). Mit dieser Form kann das Programm die Ableitung berechnen (2. Schritt) und anschließend interpretieren (3. Schritt).

2.2.1. 1. Schritt: Erfassen der Beschreibung

Im einfachsten Fall ist es denkbar, daß die Beschreibung des L-Systems als typisierte Datei vom Programm verwaltet wird, also in PASCAL so definiert werden könnte:

[5] Die Algorithmen werden in einer Pseudo-Programmiersprache beschrieben, die sehr stark an PASCAL angelehnt ist und ohne weiteres verstanden werden dürfte.

```
TYPE
  rule    = String;
  LS_File = File of Record
                      depth : word;
                      axiom : rule;
                      rules : array[1..100] of rule;
                    end;
```

Dann kann allerdings eine so erzeugtes LS-File *nicht* von einer üblichen Textverarbeitung editiert werden, man ist damit an das Programm gebunden und auch eine Übernahme fremder LS-Files (auch ansonsten gleicher Syntax) gestaltet sich schwierig.

Man kann auch eine strukturierte Textdatei zugrunde legen. Das Lesen der Informationen mittels ReadLn muß dann in derselben Reihenfolge geschehen wie beim Schreiben mittels WriteLn. Es wird dadurch eine reine Textdatei erzeugt, die mit jedem Texteditor bearbeitet werden kann. Verändert man jedoch die Struktur dieser Datei, etwa nur durch Einfügen einer zusätzlichen Leerzeile oder eines Leerzeichens, kann dies verheerende Folgen haben: Das Programm 'verheddert' sich und überschreibt vielleicht Programmcode oder Daten oder stürzt ab. Es kann aber auch zu einer Fehlinterpretation kommen: Das Axiom wird zu einer Regel oder ähnliches. Im Programm sollte daher solchen Fehlern begegnet werden. Beispiele zur Arbeit mit strukturierten Textdateien sind die Programme aus [11, 17].

Für den Programmnutzer am günstigsten — und von heutiger Software auch erwartet — ist eine formatfreie Textdatei, wie sie von Turbo Pascal und anderen Programmiersprachen verwendet wird. Allerdings ist die Verarbeitung solch freier Textdateien ungleich schwieriger als obige Methoden: das Programm muß hier selbst entscheiden, wo eine syntaktische Einheit (etwa eine Regel) abgeschlossen ist. Im allgemeinen werden bei formatfreien Textdateien sogenannte *Scanner* zum Erkennen der einzelnen syntaktischen Einheiten (*Token*) eingesetzt. Scanner basieren auf endlichen Automaten (siehe [1, 8]).

Unumgänglich wird der Einsatz eines komlexeren Scanners auch bei parametrischen L-Systemen: Es müssen Variablenbezeichner (mehrbuchstabig!), Rechenzeichen, Regelsymbole (eventuell auch mehrbuchstabig) und Sonderzeichen verarbeitet werden. Diese Token werden ja vom Programm einzeln verarbeitet und müssen deshalb irgendwann erkannt und voneinander getrennt werden. Für eine effiziente Berechnung der Ableitung reicht es in diesem Falle auch nicht aus, die Regeln als Strings abzuspeichern. Schließlich muß eine Berechnung der Bedingungen und mathematischen Ausdrücke erfolgen; wird dies durch einen Formelinterpreter erledigt[6], so hat dieser erstens die Formeln bei jedem Aufruf erneut zu interpretieren (zeitineffizient!) und zweitens erfolgt die Zerlegung der Eingabe in Token ebenfalls (innerhalb des Formelinterpreters). Effizienter ist die Benutzung eines Formelcompilers, wie er in meinem Programm LSC verwendet wird.

[6] Diese Formelinterpreter sind in verschiedenen Programmier-Toolboxen enthalten und verlangen als Eingabeparameter einen String. Oftmals wird allerdings nur eine freie Variable akzeptiert.

2.2.2. 2. Schritt: Berechnen der Ableitung

Wenn die Daten des L-Systems im Speicher bereitliegen, wird mit ihnen die Ableitung berechnet. Theoretisch geschieht dies, indem *alle* Symbole des Ableitungsstrings gleichzeitig, also *parallel* durch ihre Ersetzung ausgetauscht werden. Dieser Prozess wird sooft ausgeführt, wie es die Ableitungstiefe angibt. Als erster Ableitungsstring ist das Axiom zu wählen.

Da auf einem normalen PC leider nur ein Prozessor zur Verfügung steht, muß die parallele Ersetzung simuliert werden. Beim Austausch eines Symbols durch dessen Ersetzung bleiben die anderen Symbole des Ableitungsstrings unangetastet, daher kann für jeden Schritt der Austausch der Symbole auch *sequentiell* erfolgen:

Algorithmus 2.1 (Ableitung eines kontextsensitiven LS)

```
string1=axiom; t=0;
while t<Ableitungstiefe do
  strng2='';
  for i=1 to laenge(string1) do
    ermittle Ersetzung des i-ten Symbols von string1;
    append(string2, Ersetzung);
  endfor;
  string1=string2; inc(t);
endwhile;
derivation=string1;
```

Dieser Algorithmus ist universell nutzbar zur Ableitungsbestimmung. Auch im Falle kontextsensiver L-Systeme kann er zum Einsatz kommen, die Routine zur Ersetzungsermittlung muß lediglich kompletten Zugriff auf string1 haben. Leider hat dieser Algorithmus auch einen entscheidenden Nachteil: Es werden zwei Strings[7] im Speicher verarbeitet, die selbst im Falle einfacher L-Systeme (z.B. der Drachenkurve) Größenordnungen im Megabytebereich annehmen können. Daraus ergeben sich programmtechnische Probleme: werden die Strings komplett im Speicher gehalten, muß eine Speicherverwaltung benutzt werden, die Zusatzspeicher oder virtuellen Speicher nutzt und trotz Speichersegmentierung beim 80x86 hinreichend schnell arbeitet. Werden die Strings als Datei abgespeichert, spielt die Stringlänge keine entscheidende Rolle mehr, jedoch verlangsamt sich die Berechnung der Ableitung durch die langsamen Massenspeicher. Auch der Einsatz von Disk-Caches bringt keine wesentliche Verbesserung, hier ist ebenfalls ein zu großer Verwaltungsaufwand vonnöten.

Im Falle kontextfreier L-Systeme bietet sich daher ein rekursiver Algorithmus an (2.2), der auf das Speichern der Ableitung verzichtet und die Bildung der Ableitung mit der

[7] String ist hier kein PASCAL-String, sondern eine beliebig lange Abfolge von Symbolen.

Interpretation verknüpft. Leider können kontextsensitive L-Systeme so *prinzipiell nicht* berechnet werden, da ja der Kontext eines Symbols, also die benachbarten Symbole auf *einer* Ableitungsebene, überhaupt nicht gespeichert wird.

Algorithmus 2.2 (Ableitung eines kontextfreien LS)

```
procedure MakeDerivation(string);
  for i=1 to laenge(string) do
    inc(t);
    if t=Ableitungstiefe
      then interpretiere i-tes Symbol von string
      else makederive(Ersetzung des i-ten Symbols von string);
    dec(t);
  endfor;
endproc;

t:=0;
MakeDerivation(axiom);
```

Man kann dennoch die Ableitung als String erhalten, wenn man die Interpretation als Schreiben des Symbols in einen String aufgefaßt. Dieser stellt am Ende exakt die gewünschte Ableitung dar. Es entstehen aber wieder Speicherprobleme. Sinnvoller ist bei diesem Algorithmus die Interpretation als Turtlebefehl. Dies hat mehrere Vorteile:

- Der Speicher wird kaum belastet. Die Ableitung steht ja nie im Speicher, sondern wird auf dem Bildschirm in graphischer Form gespeichert. Eine Belastung des Speichers erfolgt hier nur im Returnstack des Programms wegen der rekursiven Aufrufe der Prozedur makederive.
- Wegen des wesentlich geringeren Speicherbedarfs ist dieser Algorithmus auch auf kleineren Computern leicht implementierbar.
- Es wird sofort mit dem Zeichnen am Bildschirm begonnen, da keine Wartezeiten durch die komplette Berechung von Zwischenableitungen entstehen.

Aufgrund dieser Vorteile wird dieser Algorithmus in allen mir bekannten Quellen für Programme zur Erzeugung kontextfreier L-Systeme mit leichten Variationen angewandt. Leider ist er für meinen LSC *nicht* zu gebrauchen, da dieser auch kontextsensitive L-Systeme bearbeiten soll.

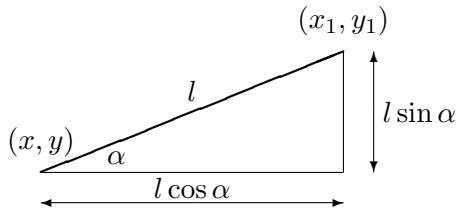
2.2.3. 3. Schritt: Interpretation der Ableitung

Zur Interpretation wird die Ableitung symbolweise ohne Zugriff auf Nachbarsymbole, also kontextfrei, abgearbeitet. Daher reicht es aus, daß die Ableitung symbolweise generiert wird, wie es in Algorithmus 2.2 geschieht.

Im Falle einer Interpretation des L-Systems als Turtlegraphik (siehe [4]), hier beschränkt auf die grundlegenden Befehle zur Bewegung der Turtle, kann die symbolweise Interpretation folgendermaßen generiert werden:

Algorithmus 2.3 (2D Turtleinterpretation)

```
case symbol of
  'F' : x1=x+l*cos(alpha); y1=y+l*sin(alpha);
        line(x,y,x1,y1); /* draw F */
        x=x1; y=y1;
  'f' : x=x+l*cos(alpha); y=y+l*sin(alpha);
  '+' : alpha=alpha+delta;
  '-' : alpha=alpha-delta;
endcase;
```



Die aktuellen Turtlekoordinaten sind dabei (x, y, α) mit x und y als 2D-Koordinaten (*turtleposition*) und α als Zeichenrichtung (*heading*). Die Winkeländerung δ ist vom L-System her vorgegeben; die Einheitslänge l sollte so festgelegt werden, daß das zu zeichnende L-System vollständig auf dem Bildschirm darstellbar ist. Ebenso muß auch die Startposition und Startrichtung der Turtle sinnvoll gewählt werden.

Die Ermittlung sinnvoller Werte für l und (x_0, y_0, α_0) kann das Programm selbst vornehmen, wenn die Ableitung *zweimal* interpretiert wird: einmal zur Ermittlung der Startwerte mit Algorithmus 2.4 und anschließend zum Zeichnen mit Algorithmus 2.3 basierend auf diesen Startwerten. Auch dabei muß nicht auf Algorithmus 2.2 und dessen Vorteile verzichtet werden, schließlich kann auch diese Ableitungsbildung *zweimal* ausgeführt werden.

Algorithmus 2.4 (Bestimmung der Startwerte)

```
xmin=0; xmax=0; ymin=0, ymax=0; x=0; y=0; alpha=0;
for i=1 to laenge(string) do
  case symbol of
    'F','f' : x=x+cos(alpha); y=y+sin(alpha);
              if x>xmax then xmax=x; if x<xmin then xmin=x;
```

```
                if y>ymax then ymax=y; if y>ymin then ymin=y;
      '+' : alpha=alpha+delta;
      '-' : alpha=alpha-delta;
    endcase;
  endfor;
  l=min(xsize/(xmax-xmin), ysize/(ymax-ymin));
  x0=-l*xmin; y0=-l*ymin; alpha=0;
```

Alle Zeichenoperationen liegen innerhalb des Rechtecks $(x_{min}, y_{min}, x_{max}, y_{max})$. Dieses Rechteck wird anschließend durch Skalierung an die Bildschirmgröße angepaßt ($xsize$ und $ysize$ sind die Bildschirmabmessungen).

Eine Interpretation mit einer dreidimensionalen Turtle erfolgt nicht wesentlich anders. In diesem Falle ist der Berechnungsaufwand der Turtleposition durch die notwendigen Matrizenmultiplikationen und die Überführung der 3D-Koordinaten in 2D-Koordinaten leider wesentlich höher. Darauf wird im Abschnitt 3.4.1 bei der Implementation der 3D-Turtle im Programm LSC näher eingegangen.

Es sei noch erwähnt, daß die Turtleinterpretation nicht die einzig mögliche ist. Zum Beispiel kann man die Ableitung in Noten umsetzen ([17, 11]) oder die Ableitung als String ausgeben, etwa bei Nutzung parametrischer L-Systeme als Parallelrechner ([13]). Auch muß die Ausgabe der Turtleinterpretation nicht unbedingt am Bildschirm erfolgen, es kann ebenso ein HPGL–Plotterfile oder eine PICTEX–Ausgabe generiert werden. Das Prinzip der sequentiellen Abarbeitung der Ableitung bleibt dabei stets erhalten.

2.3. Parametrische L-Systeme in PASCAL

Kontextfreie parametrische L-Systeme können mit Algorithmus 2.2 erzeugt werden. Wird ein Symbol als Prozedurname und die entsprechende Ersetzung als Prozedurkörper aufgefaßt, kann man kontextfreie L-Systeme schnell und einfach in PASCAL oder einer anderen höheren Programmiersprache erzeugen. Lediglich die Rekursionstiefenverwaltung muß explizit in jeder dieser Prozeduren auftauchen. Folgendes Beispiel[8] soll dies verdeutlichen:

```
PROGRAM LSystem2Pascal;
{$N+,E+}

{ OL-Systeme, besonders parametrische L-Systeme, koennen als Pascal- }
{ Programm formuliert werden. Als Beispiel wird hier ein System aus  }
{ [Prusinkiewicz 1990], Gl. 1.9 umgesetzt.                           }
{                                                                    }
```

[8] Dieses und ein weiteres Beispiel sowie die benötigen Units befinden sich auf der beigelegten Diskette.

```
{       #define R 1.456                                                  }
{       w : A(1)                                                         }
{       p1: A(s) --> F(s)[+A(s/R)][-A(s/R)]                              }
{                                                                        }
{       delta = 85                                                       }

USES
  uTurtle2D;

CONST
  R = 1.456;
  delta = 85;
  Length : float = 100;                              { Streckenlaenge }
  maxdepth : Integer = 12;                          { Ableitungstiefe }
  depth : Integer = 1;                          { akt. Rekursionstiefe }

VAR
  Turtle : tTurtle2D;

PROCEDURE A(s : float);
BEGIN
  IF depth < maxdepth THEN BEGIN
    Inc(depth);
  { A(s) --> }
    { F(s)   } Turtle.Draw(s*Length);       { es erfolgt eine 1-zu-1 }
    { [      } Turtle.PushState;          { Umsetzung der Symbole des }
    { +      } Turtle.TurnRight(delta);     { L-Systems in Prozedur- }
    { A(s/R) } A(s/R);                                    { aufrufe }
    { ]      } Turtle.PopState;
    { [      } Turtle.PushState;
    { -      } Turtle.TurnLeft(delta);
    { A(s/R) } A(s/R);
    { ]      } Turtle.PopState;
    Dec(depth);
  END;
END;

BEGIN
  Turtle.Init;
  { Axiom } A(1);
  Turtle.Done;
END.
```

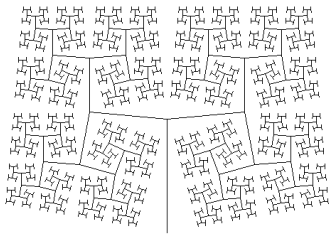


Abbildung 2.4.: Baum, mit Pascalprogramm erzeugt

Ich denke, daß dieses Programm und die Aufrufe der Turtlegraphik selbsterklärend sind und daher nicht weiter erläutert werden müssen.

Nachbemerkung. Inzwischen hat auch Prusinkiewicz in [13] analoge Gedanken geäußert und sogar Methoden zur Erzeugung kontextsensitiver L-Systeme in C angesprochen. Leider lassen sich diese Ideen nicht ohne weiteres auf PASCAL (aufgrund der strengen Syntax) übertragen. Außerdem wird die wichtige Frage, wie aus einem die Ableitung enthaltenden Textstring die zu den Symbolen gehörenden Prozeduren zur Laufzeit(!) ermittelt und inclusive Parameterübergabe aufgerufen werden sollen, nicht beantwortet.

3. LSC — ein Compiler für IL-Lindenmayer-Systeme

In diesem Kapitel wird mein Programm LSC zur Erzeugung kontextsensitiver parametrischer bedingter L-Systeme (parametric IL) vorgestellt.

Ein so umfangreiches Programm wie LSC soll im Rahmen dieser Arbeit nicht als Listing mit Kommentaren abgedruckt werden; dem detailinteressierten Leser stehen alle Quellen auf beiliegender Diskette zur Verfügung.[1] Vielmehr werden konkrete programmtechnische Prinzipien und Algorithmen des LSC-Compilers erläutert. Dabei wird besonders auf den ersten Schritt, das Erfassen der Beschreibung, eingegangen.

Zuvor wird aber noch ein kurzer Einblick in die Klasse der IL-Systeme gegeben, die LSC verarbeitet.

3.1. Kontextsensitive parametrische L-Systeme (IL)

Ein Programm, was über die in Kapitel 2.1 beschriebenen Programme hinausgehen soll, muß über einen soliden Sprachausbau verfügen. Der in meinem LSC (Lindenmayer System Compiler) gewählte Sprachumfang orientiert sich dabei an [12, Kapitel 1.10.2], geht jedoch bezüglich des Kontextes weiter: Jede Regel eines L-Systems darf einen linken und einen rechten Kontext, bestehend aus einer *beliebigen* Anzahl von Symbolen, je zu ersetzendem Symbol haben. Dabei können an die Symbole noch Parameter, ähnlich Prozeduraufrufen in höheren Programmiersprachen, angehängt werden. Mit Hilfe dieser Parameter sowie einiger spezieller Funktionen (etwa eine Zufallszahlerzeugung) können in jeder Regel mehrere logische Bedingungen erzeugt werden, die verschiedene Ersetzungen implizieren. Damit ist auch eine Realisierung stochastischer nichtdeterministischer L-Systeme gegeben: als Bedingung tritt dann z.B. $rnd < 0.33$ für die Warscheinlichkeit $\frac{1}{3}$ einer gewählten Ersetzung auf.[2] Ein Beispiel einer parametrischen bedingten kontextsensitiven Regel ist:

$$A(x) < B(y) > C(z) : x + y + z > 10 \rightarrow E((x + y)/2)F((y + z)/2)$$

[1] Die Quelltexte umfassen mehrere Tausend Zeilen!
[2] Das wird im Abschnitt 4.1 präzisiert.

$B(4)$ wird in

$$\cdots A(3)B(4)C(5)\cdots$$

durch $E(3.5)F(4.5)$ ersetzt, da sowohl der Kontext stimmt (links von B steht direkt ein A und rechts von B ein C) als auch die Bedingung $3 + 4 + 5 = 12 > 10$ erfüllt ist. Man sollte beachten, daß *nur* B ersetzt wird, auf A und C hat die Regel keinen Einfluß!

Hier soll keine exakte Definition einer Grammatik gegeben werden; diese ist mit [4] vorgelegt, vielmehr wird im Kapitel 3.2.2 die Syntax und Semantik der LSC-Sprache definiert, ebenso der Kontextbegriff und die Benutzung von Parametern.

3.2. Erfassung der Beschreibung

Der LSC besitzt eine komfortable Oberfläche, in der interaktiv IL-Systeme erstellt und graphisch dreidimensional dargestellt werden. Als Vorbild standen die integrierten Entwicklungsumgebungen der Borland–Compiler Pate. Das heißt, das L-System kann direkt in dem im LSC integrierten Editor eingegeben und auch korrigiert werden. Ebenso ist es möglich, L-Systeme von Diskette zu lesen oder sie dorthin abzuspeichern.

Werden beim Erfassen des L-Systems syntaktische oder semantische Fehler entdeckt, wird automatisch in den Editor gesprungen, eine entsprechende Fehlermeldung angezeigt und der Cursor an die Stelle im Text positioniert, wo der Fehler auftrat. Dies ist eine effektive Unterstützung des Programmnutzers, da dieser von der Aufgabe befreit wird, sein L-System absolut korrekt eingeben zu müssen.

Großer Wert wurde auch auf die formatfreie Struktur der Beschreibung gelegt. Das heißt, es können die einzelnen Token durch beliebig viele Kommentare, Leerzeichen oder Zeilenumbrüche getrennt werden; feste Positionen einzelner Sprachbestandteile wie etwa in FORTRAN oder strukturierten Textdateien (siehe auch Abschnitt 2.2.1) werden nicht erwartet.

Formatfreie Textdateien (die Programmbeschreibungen einer gewissen Sprache sind), deren Transformation in eine intern besser handhabbare Form (oft eine maschinennahe Sprache oder sogar der Maschinencode selbst), sowie die Fehlererkennung sind klassische Merkmale eines Compilers. Daher tun in allen Programmen für L-Systeme Bestandteile eines Compilers ihr Werk, auch wenn diese nicht als solche herausgestellt werden. So ist auch das Einlesen einer strukturierten Textdatei mittels ReadLn ein — wenn auch äußerst primitiver — Scanner, der die Textdatei zu Token zerpflückt. Das Ermitteln des i-ten Symbols eines Strings bei der Berechnung der Ableitung oder die Berechnung mathematischer Ausdrücke parametrischer L-Systeme führt ein Parser durch, der die gelesenen Token interpretiert.

Nun hätte für das vorliegende Programm LSC ein Compiler per Hand geschrieben werden können. Doch glücklicherweise gibt es CASE-Tools zur Compilererstellung, die den

Programmierer von lästiger Routineprogrammierung befreien und ihm so die Möglichkeit geben, sich auf die wesentlichen Punkte seiner Arbeit zu konzentrieren. Auch werden in diesen Tools moderne und komplizierte Algorithmen zur Optimierung der erzeugten Compiler verwendet, die ein Programmierer von sich aus vielleicht nicht genutzt hätte.

Diese sogenannten Compiler-Compiler erzeugen aus metasprachlichen Beschreibungen des lexikalischen und syntaktischen Aufbaus der Zielsprache sofort verwendbare Module in der Compiler-Implementationssprache. Diese beinhalten im allgemeinen auch gleich Fehlererkennungs- und -behandlungsroutinen für lexikalische und syntaktische Fehler. Abbildung 3.1 stellt das Zusammenwirken von Implementations- und Beschreibungssprache mit dem Compiler-Compiler und dem erzeugten Compiler mit seiner Quell- und Zielsprache dar.

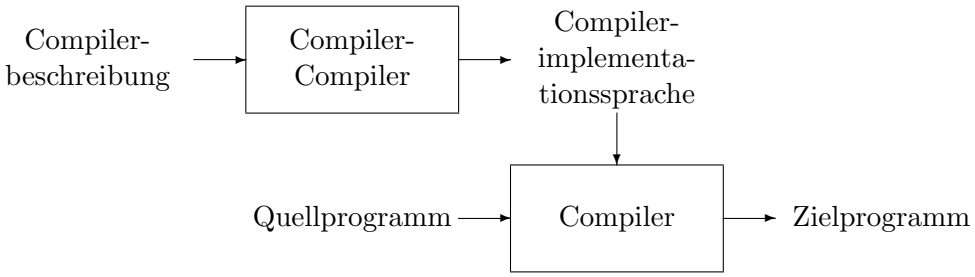


Abbildung 3.1.: Compiler-Compiler und erzeugter Compiler

3.2.1. Compilerbau mit ALEX und COCO

Von P.Rechenberg und H.Mössenböck wurde der moderne Compiler-Compiler COCO geschaffen ([14, 8]), der als Compiler-Implementationssprache Turbo-PASCAL verwendet. Mit seiner leicht verständlichen Beschreibungssprache Cocol, der guten Zusammenarbeit mit dem Scanner ALEX und seinen modernen Analyseverfahren ist COCO ein einfach zu erlernendes und zu benutzendes Softwarewerkzeug, das nicht nur zum Bau von Compilern verwendet werden kann. COCO generiert Compiler für LL1-Sprachen (linksrekursive Sprachen mit einem Lookahead von einem Token). Über LL1-Sprachen findet der interessierte Leser in [14, 1] Ausführungen.

3.2.1.1. ALEX – ein Scannergenerator

Ein mit ALEX konstruierter lexikalischer Scanner führt die erste Phase eines Compilers durch: die lexikalische Analyse. Dazu zerlegt er ein Quellprogramm in einzelne Teile, wie Namen, Zahlen oder Rechenzeichen, die sogenannten *Token.* Mit diesen können die syntaktischen Regeln einer Grammatik einfach dargestellt werden. Außerdem werden in der

lexikalischen Analyse bedeutungslose Zeichen wie Kommentare, Leerzeilen und überflüssige Leerzeichen entfernt.

Beispielsweise sieht die — etwas vereinfachte — Deklaration von Variablen in PASCAL in EBNF[3] geschrieben wie folgt aus:

```
VAR identifier { ',' identifier } ':' identifier ';'
```

Dabei sind `VAR, ',', ':'` und `';'` einzelne Token, während `identifier` für eine Tokenklasse steht: hier für eine in PASCAL für Bezeichner zulässige Kombination von Buchstaben und Ziffern. Ohne dieses Token wäre eine Beschreibung wie die obige in dieser kurzen Form nicht möglich; es hätten ja anstelle von `identifier` alle erlaubten Bezeichner aufgezählt werden müssen.

Von ALEX verarbeitete Tokenarten sind:

- Schlüsselworte (*keywords*) wie `IF, END` usw.
- Tokenklassen (*token classes*). Dies umfaßt Namen, Zahlen, Strings und ähnliches. ALEX liefert dann nicht die Zeichenfolge selbst, sondern das Token für diese Klasse.
- einzelne Zeichen und Verbundzeichen (*single tokens*) wie `','` und `'<='`.

ALEX liefert für jedes erkannte Token eine korrespondierende Nummer. War das Token eine Tokenklasse, legt ALEX die Zeichenkette intern ab und liefert in diesem Fall noch einen numerischen Wert, den *spelling index*, mit dem im Parser über die Prozedur `GetVal` die aktuelle Zeichenfolge erhalten werden kann.

Die Beschreibung von Tokenklassen erfolgt in EBNF, ein Beispiel dafür ist die Deklaration von `ident` und `number` in Abschnitt 3.2.2. Diese Deklarationen müssen keine LL1-Sprachdefinition ergeben, ALEX versucht von sich aus, die Deklarationen zusammenzufassen (siehe Beispiel eines Modula-2 Scanners mit dem zugehörigen erzeugten Automaten in [8]).

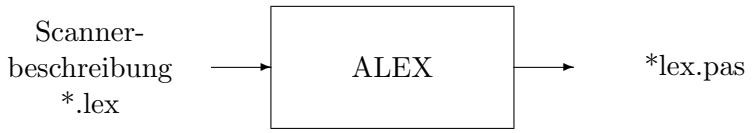


Abbildung 3.2.: der lexikalische Scanner-Generator ALEX

[3] Extended Backus Naur Form, von N.Wirth empfohlene Metasprache zur Beschreibung von Grammatiken. Dabei steht $\{a\}$ für eine beliebige Anzahl von Wiederholungen von a, auch null mal; $[a]$ für ein null- oder einmaliges Auftreten von a und $a|b$ für entweder a oder b.

Für die praktische Arbeit mit ALEX reicht obiges Wissen und die Kenntnis der Scannerbeschreibungssprache *Alexis*[4] nach [8] vollkommen aus, die Zusammenarbeit mit dem von COCO erzeugten Parser erfolgt innerhalb der von COCO generierten Units; der Nutzer hat keine Notwendigkeit, hier einzugreifen.

3.2.1.2. COCO – ein Compiler-Compiler

Der Compiler-Compiler COCO erzeugt Programmteile für die zweite Phase eines Compilers: die syntaktische und semantische Analyse. Die Analyse liefert

- die Feststellung, ob das Quellprogramm korrekt ist; wenn nicht, wird eine Liste der gefundenen Fehler zur späteren Auswertung angelegt.
- die für die Abarbeitung des Quellprogramms notwendige speicherinterne Darstellung in Form der *Zwischensprache* bzw. des *p-Codes*. [5]
- speicherinterne Tabellen, die für die Verarbeitung der Zwischensprache notwendig sind.

Abbildung 3.3.: der Compiler-Compiler COCO

Beschreibungssprache. Die Compilerbeschreibungssprache *Cocol* beruht auf folgender in EBNF deklarierten kontextfreien Grammatik:

```
coco      = GRAMMAR ident
            [ SEMANTIC DECLARATIONS { any } ]
            [ MACROS { macrodef } ]
            TERMINALS { symbol [ attr ] [ aliasname ] }
            [ PRAGMAS { symbol [ attr ] [ semaction ] } ]
            NONTERMINALS { ident [ attr ] [ aliasname ] }
            RULES { ident [ attr ] '=' expr '.' } ]
            ENDGRAM .
expr      = term {'|' term } .
```

[4] Die Sprache Alexis ist so einfach gehalten, daß sie aus wenigen Beispielen heraus begriffen werden kann. So dürfte der in Abschnitt 3.2.2 vorgestellte Parser meines LSC vollständig zu verstehen sein.

[5] Wird kein Zwischencode erzeugt, sondern sofort Aktionen ausgeführt, entartet der generierte Compiler zu einem Interpreter.

```
term      = fact { fact } .
fact      = ( symbol [ attr ]
              | 'eps'
              | 'any'
              | semaction
              | '(' expr ')'
              | '{' expr '}'
              | '[' expr ']' .
attr      = '<' ( outattr | inattr [ ';' outattr ] ) '>' .
inattr    = 'in' ':' ( ident | number ) { ',' ( ident | number ) } .
outattr   = 'out' ':' ident { ',' ident } .
semaction = 'sem' ( '(' ident ')' | { any } ) 'endsem' .
macrodef  = 'sem' ':' ident ':' { any } 'endsem' .
symbol    = ident | string .
aliasname = ALIAS symbol .
```

Es müssen alle Terminal- und Nichtterminalsymbole vor ihrem ersten Aufruf deklariert werden. Jedes Symbol kann mit beliebig vielen Attributen versehen werden (siehe [14]). In den Syntaxregeln *semaction* dürfen beliebige PASCAL-Konstruktionen verwendet werden, in der Regel stehen hier Programmfragmente zur sematischen Analyse (etwa Typ- und Bereichsüberpüfung) und Zwischencodeerzeugung.

Im allgemeinen wird bei der Grammatikbeschreibung links die Syntax geschrieben und rechts davon — wenn an dieser Stelle notwendig — zwischen *sem* und *endsem* die PASCAL-Fragmente. Man kann diesen Text dann so interpretieren, daß nach der Erkennung eines Symbols der zugehörige PASCAL-Code ausgeführt werden wird. Folgender kurzer Ausschnitt aus der Definition der LSC-Sprache, die Deklaration der Ableitungstiefe, soll dies verdeutlichen:

```
...
depthDecl =
  depth
  number<out:valu>        sem depth:=Str2Float(valu);
                              IF depth<>Int(depth) THEN SError(9);
                          endsem
  .
...
```

Es wird das Schlüsselwort `'depth'` erwartet und danach eine Zahl. Da der von ALEX erzeugte Scanner nur Strings bei Tokenklassen liefert[6], wird mit der Funktion `Str2Float`

[6] Genauer ist `valu` ein numerischer Wert, ein Hashtable-Pointer, über den auf den eigentlichen Zahlenstring zugeriffen wird. Dies geschieht alles in der Funktion `Str2Float`.

dieser in einen numerischen Wert umgewandelt und in der globalen Variable `depth` festgehalten (das entspricht hier der Erzeugung des Zwischencodes). Anschließend wird ein Test auf den Zahlentyp durchgeführt, ist `depth` keine Integervariable, so wird eine entsprechende Fehlermeldung generiert (also ein Semantiktest durchgeführt).

Bestandteile der erzeugten Compilers. COCO erzeugt aus der Compilerbeschreibung einen Syntaxanalysator und ein Semantikauswerteprogramm. Der Nutzer muß nur noch ein Hauptprogramm und die semantischen Module hinzufügen. Diese sind bei ähnlichen Aufgaben (hier die Konstruktion prozeduraler Sprachen) einander sehr ähnlich, so daß auch hier auf bereits bestehende Units zurückgegriffen werden kann. Im allgemeinen müssen aber noch Units für die Abarbeitung der Zwischensprache — deren Interpretation bzw. Umwandlung in eine weitere systemnähere Sprache, etwa Maschinencode — vom Nutzer entwickelt und dem Projekt hinzugefügt werden.

3.2.2. Die Sprache LSC

Entsprechend der Zielsetzung in Abschnitt 3.1 orientiert sich der Sprachumfang und (L-System-) Programmaufbau stark an [11, 12]. So können die Beispiele aus P. Prusinkiewicz' Werken fast ohne Änderungen übernommen und gestestet werden.

Allgemeiner Aufbau eines LSC-Programms. Ein LSC-Programm beginnt mit den Deklarationen von Konstanten, zu ignorierenden Symbolen und der Ableitungstiefe. Anschließend folgt die Deklaration des Axioms und der einzelnen Regeln. Es schließen sich für die Interpretation nötige Informationen an.[7] In EBNF sieht dieses folgendermaßen aus:

```
ls               = { constDecl | ignoreDecl | depthDecl }
                   axiomDecl
                   [ rulesDecl ] .
constDecl        = CONST ConstDeclaration { ConstDeclaration } .
ConstDeclaration = ident ['='] simExpr .
ignoreDecl       = ignore procFormal { procFormal } .
depthDecl        = depth number .
axiomDecl        = axiom procCall { procCall } .
rulesDecl        = rules { rule } .
```

Eine Regel wiederum besteht aus dem linken Kontext, dem Regelnamen, dem rechten Kontext und mehreren Bedingungen und dazugehörige Ersetzungen des Symbols. Kontext, Bedingungen und Ersetzungen sind optional.

[7] Um eine strikte Trennung in Ableitungsbildung und Ableitungsinterpretation zu erhalten, werden diese Deklarationen erst im Abschnitt 3.4 behandelt. Die im Compiler notwendigen Erweiterungen gegenüber der hier beschriebenen Versionen sind alle von der Art der oben beschriebenen Tiefendeklaration und haben *keinen* Einfluß auf die Ableitungsbildung.

```
rule             = [ ':' context '<' ]
                   procFormal
                   [ '>' context ]
                   ruleBody { ruleBody } ';' .
context          = procFormal { procFormal } .
ruleBody         = [ ':' Expression ] '-->' { procCall } .
```

Die Sonderzeichen `':'` und `';'` treten bei Prusinkiewicz nicht auf, wurden aber eingeführt, um die LSC-Sprache LL1-konform zu halten: Ohne diese Sonderzeichen könnte die LSC-Sprache nur in 2 Pässen oder aber mit einem unbegrenzten Lookahead gelesen werden (Ist das erste Symbol einer Regel linker Kontext oder der Regelname? Dies läßt sich in LL1-Sprachen *nicht* beantworten). `':'` beginnt den linken Kontext, wenn dieser vorhanden ist, sowie den Anfang einer logischen Bedingung (`Expression`). Das Semikolon `';'` definiert das Ende einer Regel und sollte nie vergessen werden! Folgendes Beispiel zeigt, warum:

```
A(x) : x > 0       --> A(x/2)
B                  --> A(1)B;
```

wird aufgefaßt als

```
A(x) : x > 0       --> A(x/2)B
       /* else */  --> A(1)B;
```

und ist damit eine sprachkonforme Schreibweise, die aus A eine Regel mit zwei Ersetzungen macht. Auch der Compiler erkennt *nicht*, daß für B keine Regel definiert wurde; bei L-Systemen werden ja Symbole ohne eigene Regel bei der Ableitungsbildung einfach übernommen![8]

Ein einzelnes Symbol — in der Grammatik mit `procName` bezeichnet — tritt sowohl im Bedingungsteil einer Regel links von '→' auf als auch im Ersetzungsteil rechts von '→'. Im Bedingungsteil einer Regel dürfen *nur* Variablen als Parameter verwendet werden (diese werden an dieser Stelle initialisiert); im Ersetzungsteil dürfen als Parameter *beliebige* mathematische Ausdrücke (`SimExpr`) gebildet aus ebendiesen Variablen und globalen Konstanten auftreten.

```
procName         = ident | lsIdent | '+' | '-' | '/' | '^' .
procFormal       = procName [ '(' formalParameters ')' ] .
formalParameters = ident { ',' ident } .
procCall         = procName [ '(' actualParameters ')' ] .
actualParameters = SimExpr { ',' SimExpr } .
```

[8] Durch eine striktere Defininition der Regeln, etwa geklammert in `begin` und `end`, kann auch dieses Problem gelöst werden. Doch zeichnen sich L-Systeme auch durch ihre kompakte Schreibweise aus, die dann nicht mehr vorhanden ist.

Die Tokenklassen `ident` und `lsIdent` werden im Scanner deklariert.

Die algebraischen und logischen Ausdrücke stimmen mit der üblichen mathematischen Schreibweise voll überein, die Rangfolge der Operatoren entspricht der gewohnten mathematischen Auffassung (siehe auch Abschnitt 3.2.3.2).

```
Expression      = SimExpr [RO SimExpr] .
SimExpr         = Term {AO Term} .
Term            = PFactor {MO PFactor} .
PFactor         = Factor [ '^' Factor ] .
Factor          = ident
                  | number
                  | '-' Factor
                  | 'NOT' Factor
                  | FO '(' Expression ')'
                  | rnd.
FO              = int | sin | eps .
AO              = '+' | '-' | 'OR' .
MO              = '*' | '/' | 'AND' .
RO              = '=' | '<' | '>' | '<=' | '>=' | '<>' .
```

Der LSC-Parser wird als komplettes Listing wiedergegeben. Wie aus dem Listing ersichtlich ist, dürfen Symbolnamen auch aus *mehreren* Buchstaben und Ziffern bestehen! Dies trägt zu einer besseren Lesbarkeit komplexerer L-Systeme gegenüber einbuchstabigen Symbolen bei. Ein Beispiel dafür ist `plant.lsc` aus [11, figure 3.4] (auch auf der Diskette enthalten). Kommentare werden wie bei Prusinkiewicz (und wie in C) in `/*` und `*/` eingeschlossen.

```
SCANNER ls
------------------------------------------------------------------------

CHARACTER SETS
  letter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_";
  digit     = "0123456789";
  lsletter  = "|\&!.''[]{}~";          -- spezielle LS-Symbole
  endfile   = CHR(26);
  endline   = CHR(13);

TOKEN CLASSES
   1 = letter {letter | digit} EXCEPT KEYWORDS.  -- ident
   2 = digit {digit} [ "." digit {digit} ].      -- number
   3 = lsletter.                                 -- lsIdent
```

```
ERROR TOKEN = 4

KEYWORDS
   5 = "and"        6 = "or"         7 = "not"
   8 = "depth"      9 = "axiom"     10 = "rules"

SINGLE TOKENS
  11 = "-->".      12 = ":".        13 = "<".        14 = ">".
  15 = "(".        16 = ")".        17 = "+".        18 = "-".
  19 = "*".        20 = "/".        21 = "=".        22 = "<=".
  23 = ">=".       24 = "<>".       25 = ",".        26 = ";".
  27 = "^".

KEYWORDS
  28 = "ignore"    29 = "const"     30 = "rnd"       31 = "int"
  32 = "sin"
  -- hier erfolgen Eintraege fuer die Interpretation

SINGLE TOKENS -- pragmas
   0 = endfile.    33 = endline.

COMMENTS FROM "/*"  TO "*/"

END
```

Die semantische Beschreibung ist mit der Angabe der Grammatik leider noch nicht gegeben. Sie wird daher verbal nachgereicht:

- Für mathematische Konstrukte gelten die von PASCAL gewohnten Vereinbarungen: eine Variable oder Konstante kann erst dann im Ersetzungsteil einer Regel verwendet werden, wenn diese vorher deklariert wurde (andernfalls gibt der LSC-Compiler eine entsprechende Fehlermeldung aus).

- Jedes Symbol muß das ganze L-System hindurch eine konstante Anzahl von Parametern haben; das erste Auftreten eines Symbols wird als dessen Deklaration gewertet.

- Tritt ein Symbol erstmals im `ignore`-Teil auf, müssen daher auch hier symbolische Parameterbezeichner benutzt werden; diese können frei gewählt werden und haben keinerlei Einfluß auf die folgenden Deklarationen.

- Variablen (Parameterbezeichner) sind nur lokal in der betreffenden Regel gültig.

- Wird in einer Regel eine Variablenname verwendet, der vorher bereits als Konstante deklariert wurde, kann die Konstante in dieser Regel nicht mehr genutzt werden (Prinzip der Lokalität von Bezeichnern, analog PASCAL).

- Sind mehrere Regeln auf ein Symbol anwendbar, wird von diesen die im Quelltext an erster Stelle stehende angewandt.

Ein Beispiel möge die LSC-Sprache verdeutlichen:

```
/*********************************************************************/
/* wenig sinnvolles L-System mit vielen in LSC moeglichen Konstrukten */
/*********************************************************************/

const  z=10           /* diese Konstanten koennen auch ohne '=' */
       t=z+2.55       /* deklariert werden */
ignore +(i,i)-(j)     /* bei der Kontextsuche zu ignorierende Symbole */
                      /* i,j sind dabei formale Platzhalter */
depth  5              /* Ableitungstiefe */
axiom  G(3)F(1,5)     /* Startsymbol */
rules
: G(x) < F(z,y)         : x+z<=10         --> G(z)F(x+y,0.3)-(3)f(y);
         G(x)  > F(a,b) : (x>0) or (x<10) --> F(z,b)~D(a)C(a,x);
         G(x)                             --> C(x,1);
         C(a,s)         : a=0  -->    /* leere Ableitung */
                        : a<10 --> C(a-1,s+t)
                  /* else */  --> C(s,0);
```

3.2.3. Der LSC-Compiler

Leider reicht die alleinige Angabe einer Grammatik wie obige für LSC nicht aus, um einen lauffähigen Compiler zu erhalten. Wie schon angesprochen, fehlt noch der PASCAL-Code der semantischen Beschreibung zu dieser Grammantik. Hierbei werden die verschiedensten Konstrukte vereinigt; es gilt, Problemkreise wie Fehlerbehandlung, Erzeugung des Zwischencodes, Verwaltung der Bezeichner (etwa Prozedur- und Variablennamen) und ähnliches zusammenzubringen und zu koordinieren. Abbildung 3.4 verdeutlicht die Komplexität des Compilers von LSC. Die wichtigste vom LSC-Compiler exportierte Prozedur ist `parse` aus der Unit `lsSyn.pas`. Durch ihren Aufruf im Hauptprogramm wird der gesamte Compilierungsvorgang gestartet. Im fehlerfreien Fall ist nach Beendigung von `parse` der Zwischencode sowie die LSI-Tabelle (siehe 3.2.3.1) erzeugt.

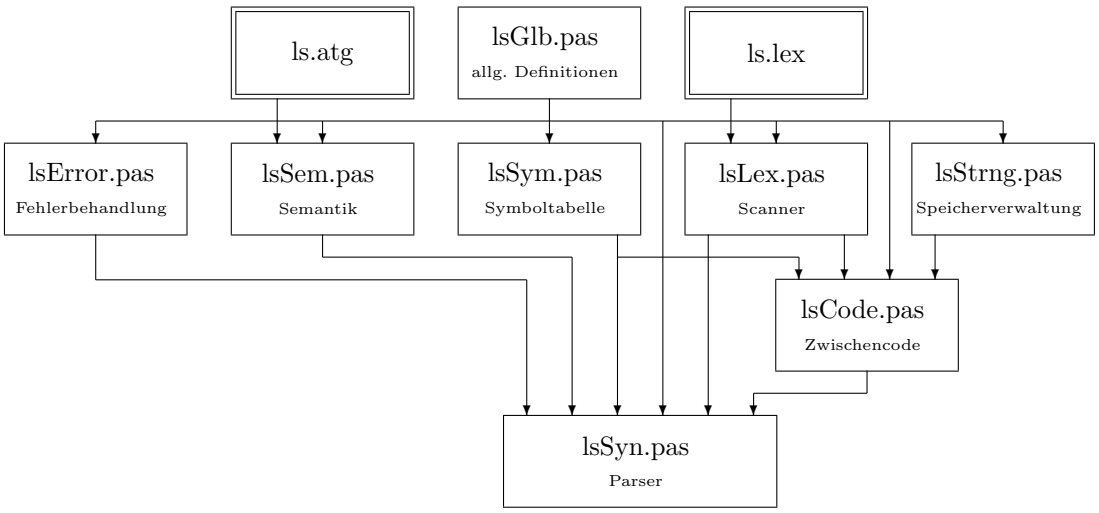


Abbildung 3.4.: Zusammenspiel der am eigentlichen Compiler beteiligten Units

3.2.3.1. Semantische Aktionen

Einfache Aufgaben, wie das Übernehmen der Ableitungstiefe in eine globale Variable (siehe Seite 26), bereiten keine Schwierigkeiten beim Aufstellen der zugehörigen Semantikaktionen. Komplexere Aufgaben müssen dagegen gut vorbereitet sein, um die Semantik überschaubar und leicht erweiterbar zu halten. Beim LSC-Compiler betrifft dies die Bereiche:

- Verwaltung der Symbole des L-Systems
- Verwaltung der Regelbezeichner und der zu den Regeln gehörenden Variablen.
- Umsetzung algebraischer und Boolescher Ausdrücke
- Umsetzung von Konstantendeklarationen
- Speicherung der Regeln
- Umsetzung des linken und rechten Kontexts

Die Verwaltung der Symbole eines L-Systems wird mit der in der Unit `lsGlb.pas` definierten *LSI-Tabelle* vollzogen. Anstelle der Symbolbezeichner bzw. deren spelling index' werden innerhalb von LSC nur ein Byte große Nummern geführt. Das sichert eine schnellere Abarbeitung bei der Ableitungsbildung und trägt zur Speicherminimierung bei.

```
LSI: ARRAY[1..255] OF RECORD
```

```
    Spix : Word;       { Spelling Index fuer Name       }
    Addr : Word;       { StartAdresse                   }
    Param : Word;      { Anzahl der Parameter * Size    }
    ignore : Boolean;  { fuer Kontextsuche              }
    Interpret : Word;  { fuer Interpretation reserviert }
END;
```

Mit der LSI-Tabelle kann auf alle notwendigen Informationen zu einem Symbol zugegriffen werden: über den spelling index `spix` auf den Symbolbezeichner, mit `Param` auf die Anzahl der Parameter des Symbols[9], ob das Symbol bei der Kontextsuche zu ignorieren ist (`ignore`) sowie auf die Startadresse `addr` des zugehörigen Zwischencodes (für die Ableitungsbildung).

Aufgrund der Beschränkung auf Bytes ist die Anzahl der Symbole eines zu generierenden L-Systems auf 255 beschränkt.[10] Für die meisten L-Systeme dürfte diese Zahl jedoch voll ausreichen; selbst komplexe Beispiele aus [12] benötigen selten mehr als zwanzig bis dreißig Symbole.

Regelbezeichner und Variablen werden in der Unit `lsSym.pas` in einer ziemlich komplexen Struktur verwaltet. Mit ihrer Hilfe kann der Compiler leicht die Übersicht über die in einer Regel aktuellen Variablen und globaler Konstanten behalten. Es kann nach gleichem Namen gesucht werden (Wurde diese Variable bereits deklariert?) sowie nach globaleren Bezeichnern (Gibt es bereits Regeln für ein Symbol? Ist dieser Bezeichner eine Konstante?). `lsSym.pas` übernimmt auch die Vergabe von Speicherplatz für Variablen. Bei Konstanten wird ihr Wert zurückgeliefert, sonst ein Verweis auf die Speicheradresse des Objekts.

Die Unit `lsSym.pas` basiert auf `tastesym.mod` von H.Mössenböck, der Symbolverwaltung eines kleinen Modula-ähnlichen Beispiel-Compilers zu ALEX/COCO. Um diese an die Erfordernisse von LSC anzupassen, waren umfangreiche Änderungen und Erweiterungen notwendig. So dürfen ja in Modula — wie auch in PASCAL — Prozeduren erst dann aufgerufen werden, nachdem sie deklariert wurden. Im Gegensatz dazu darf in L-Systemen ein Symbol auch ohne Regel genutzt werden. Hier waren erweiterte Suchroutinen gefragt. Auch die dynamische Speicherverwaltung mußte an die interaktive Arbeit von LSC angepaßt werden.

Algebraische und Boolesche Ausdrücke werden in die sogenannte *umgekehrte polnische Notation* überführt (siehe Abschnitt 3.2.3.2). Steht ein solcher Ausdruck bei einer **Konstantendefinition**, wird der für diesen Ausdruck erzeugte Zwischencode abgearbeitet und nur das Ergebnis der Symbolverwaltung mitgeteilt. Dieser Zwischencode kann verworfen werden.

[9] Es wird nicht die Anzahl der Parameter gespeichert, sondern der von den Parametern belegte Speicherplatz: In der Ableitungsbildung wird dieses Produkt häufiger als die Parameteranzahl selbst gebraucht. Damit führt das Merken des Speicherbedarfs zu einer höheren Effizienz der Ableitungsbildung.

[10] Die Null wird im Compiler als Abschlußsymbol benötigt

```
ConstDeclaration =
  ident<out:spix>          sem Declare(spix,iconst,ruleobj);
                                        { neue Konstante deklarieren }
                               ruleobj^.typ:=ifloat; { vom Typ float }
                               Push(pc);                { sichere PC }
                           endsem
  ["="]
  Expression<out:typ>      sem Pop(fix);       { Startadresse p-Code }
                               IF typ<>1 THEN BEGIN { falls falscher }
                                 SError(8); typ:=1;  { Typ --> Error }
                               END;
                               EmitOp(iRETC);
                               GetNumberOfErrors(anz,lanz);
                                      { traten Fehler im p-Code auf? }
                               IF anz+lanz=0 THEN       { nein, dann }
                               Interpret(fix,ConstOk)    { berechnen }
                               IF not ConstOk THEN SError(12);
                               pc := fix;         { PC zuruecksetzen }
                               ruleobj^.valu := ConstFloat;   { Wert }
                           endsem .
```

Obige Deklaration einer Konstanten zeigt diesen Ablauf.

Regeln werden als Prozeduren aufgefaßt (vergleiche Abschnitt 2.3) und in übliche Zwischencodestrukturen umgesetzt [1]. Es wird also eine Sequenz von Regelaufrufen erzeugt, gepaart mit Code zur Kontextsuche und Code für Bedingungen. Bedingungen entsprechen einer '`if not ... then`'-Struktur und werden mit einem `FJMP`-Befehl[11] realisiert. Wurde eine Regel erfolgreich abgearbeitet, so wird die Interpretation mit `RET` abgeschlossen, andernfalls weitere zu diesem Symbol gehörende Regeln bzw. weitere Regelkörper (in der LSC-Grammatik mit `ruleBody` bezeichnet) interpretiert. Verlief auch dies erfolglos, wird die Interpretation mit einem `FRET`-Befehl beendet. Die zu einem Symbol gehörenden Regeln und Regelkörper werden als lineare verkettete Liste im Zwischencode gehalten.

Der linke bzw. rechte Kontext eines Symbols wird mit den Befehlen `LCTX` bzw. `RCTX` eingeleitet und als Sequenz von Regelaufrufen abgespeichert. Das Ende des Kontextes wird mit dem Zwischencode-Befehl `CEND` angezeigt.

Die zum linken Kontext gehörenden Symbole werden dabei in umgekehrter Reihenfolge bezüglich ihrer Deklaration im Quellprogramm abgespeichert. Damit ist ein einfaches und analoges Vorgehen beim Kontextvergleich sowohl beim linken als auch beim rechten Kontext möglich: Es wird im Ableitungsstring ein Symbol nach links bzw. rechts geschritten und mit dem nächsten Symbol des Ableitungskontextes verglichen. Details der Kontextsuche werden im Abschnitt 3.3.3 besprochen.

[11] Eine genaue Beschreibung des Zwischencodes und ein Beipiel eines übersetzten L-Systems folgt in Abschnitt 3.2.3.4

Abschließend wird die — stark vereinfachte — Semantik der Regeldeklaration wiedergegeben:[12]

```
rule =                      sem NewObj(9999,procs,ruleobj);
                                ...    { neue Regel und Scope anlegen }
                            EmitOp(iADNX);     { verkettete Liste }
                            Emit(dummyaddr);{ der Regel erweitern }
                            endsem
    [ ":"                   sem EmitOp(iLCTX);       { linker Kontext }
                            endsem
    procFormal                 { siehe ls.atg fuer Umkehr der Reihen- }
    { procFormal }                                { folge der Symbole }
    "<"                     sem EmitOp(iCEND);     { Ende li. Kontext }
                            endsem
    ]
    procName<out:rulepos>   sem FindLocalFather(LSI[rulepos].spix,obj);
                                  { ex. schon Regeln fuer das Symbol? }
                            IF obj<>NIL THEN { ja --> verk. Liste }
                              FixUp(LSI[rulepos].Interpret,
                              ruleobj^.start)         { erweitern }
                            ELSE       { sonst neue Liste anlegen }
                              LSI[rulepos].Addr:=ruleobj^.start;
                            endsem
    [ "(" formalParameters<out:anz> ")" ]
                            sem ...       { Parameteranzahl behandeln }
                            endsem
    [ ">"                   sem EmitOp(iRCTX);      { rechter Kontext }
                            endsem
      procFormal
      { procFormal }        sem EmitOp(iCEND);          { Kontextende }
                            endsem
    ]
    ruleBody { ruleBody }                          { die Regelkoerper }
    ";"                     sem EmitOp(iFRET);  { Regelende b. Fehler }
                            LeaveScope;          { Scope freigeben }
                            endsem .
------------------------------------------------------------------------
ruleBody =
  [ ":" Expression<out:typ> sem IF typ<>2 THEN SError(6);{ Boolesch }
```

[12] Es soll an dieser Stelle nochmals betont werden, daß die hier wiedergegebenen Programmauszüge nur der allgemeinen Darstellung dienen und nicht bis ins letzte Detail erläutert werden. Der interessierte Leser möge die Quelltexte von beiliegender Diskette für Details zu Rate ziehen.

```
                              EmitOp(iFJMP);     { verkettete Liste }
                              Emit(addr);               { erweitern }
                            endsem
  ] "-->" ( procCall { procCall } | eps )
                            sem EmitOp(iRET);    { normales Regelende }
                            endsem .
-----------------------------------------------------------------------
procFormal =
  procName<out:Pos>         sem anz:=0;             { Parameteranzahl }
                              EmitOp(iCBYT); EmitByte(Pos);
                            endsem
  [ "(" formalParameters<out:anz> ")" ]
                            sem ...       { Parameteranzahl behandeln }
                            endsem .
-----------------------------------------------------------------------
formalParameters<out:anz> =
  ident<out:spix>           sem Declare(spix,vars,obj);   { neue Var. }
                              obj^.typ:=ifloat;         { Typ float }
                              anz:=1;              { Parameteranzahl }
                              EmitOp(iSTO);       { Speicheradresse }
                              Emit(obj^.Addr);      { der Parameter }
                            endsem
  { "," ident<out:spix>     sem Declare(spix,vars,obj);     { weitere }
                              obj^.typ:=ifloat;         { Parameter }
                              inc(anz);
                            endsem
  } .
-----------------------------------------------------------------------
procCall                    sem ... endsem             { siehe ls.atg }
actualParameters            sem ... endsem             { siehe ls.atg }
```

3.2.3.2. Der mathematische Parser

Mathematische Ausdrücke lassen sich leicht berechnen, wenn die übliche Infix-Notation in die umgekehrte polnische Notation (Postfix-Notation) überführt und mit Hilfe einer 2-Stapel-Maschine abgearbeitet wird (siehe [1]). Ein Stapel dient zum Ablegen der Zahlen und Rechenergebnisse, der andere ist der übliche Programmstack. Bei der Umwandlung der Infix- in Postfixausdrücke werden die Prioritäten der Operatoren untereinander sowie Klammerausdrücke aufgelöst, es entsteht ein Ausdruck, der linear von links nach rechts abgearbeitet werden kann. Zum Beispiel wird

$$\sin(34\pi) * (5 - (45 * 3/(8 + 2) - 3))$$

zu

$$34\ \pi * \sin 5\ 45\ 3 *\ 8\ 2 + /\ 3 - - *$$

Dies sieht auf den ersten Blick recht ungewohnt aus, Liebhaber der HP-Taschenrechner oder FORTH-Freunde sind diese Notation jedoch gewohnt. Und ein Computer hat ohnehin keine Schwierigkeiten damit (wenn er dafür programmiert wurde). Alle Rechenoperationen laufen *nur* über den Stack. Einzelne Zahlen werden auf den Stack gelegt. Ein Operator nimmt die benötigte Anzahl Operanden vom Stack, verknüpft sie und legt das Ergebnis auf dem Stack ab. So nimmt '+' zwei Zahlen vom Stack und legt deren Summe zurück. Alle mathematischen Operationen werden direkt in Zwischencodebefehle umgesetzt und im Abschnitt 3.2.3.4 weiter erläutert.

Wie aber erfolgt die Umsetzung von Infix- in Postfix-Notation? Nun, die wichtigste Aufgabe dabei, das Auflösen der Operatorprioritäten und der Klammerungen übernimmt der Parser, genauer gesagt, die Definition der Grammatik sorgt bereits dafür! Um beispielsweise ein Produkt zu parsen, das in einer Grammatik als `term = factor * factor.` deklariert worden ist, hat ein COCO-generierter Compiler bereits beide Faktoren analysiert und dabei auch deren sematische Aktionen, speziell die Zwischencodeerzeugung, ausgeführt! Es muß daher nur dafür gesorgt werden, daß die Operatoren *nach* den Operanden abgespeichert werden. Durch diesen Effekt des Compilers konnte der eigenlich recht komplizierte mathematische Parser sehr einfach gehalten werden (siehe `ls.atg`).

Derzeit sind in LSC die in Tabelle 3.1 aufgezeigten Operatoren implementiert.

Priorität	Operatoren
1	=, <, >, <=, >=, <>
2	+, −, OR
3	*, /, AND
4	∧(Potenz)
5	negatives Vorzeichen, NOT
6	rnd, int, sin, Klammerungen

Tabelle 3.1.: mathematische Operatoren in LSC

Sollte es nötig sein, den Parser mit weiteren mathematischen Funktionen auszustatten, ist dazu der gewünschte Funktionsname im Scanner `ls.lex` zu deklarieren sowie in der Grammatik `ls.atg` an folgenden Stellen einzuarbeiten: bei den Semantic Declarations, in die Deklaration von `F0` sowie in die Deklaration von `Factor`. Dabei sollte man sich an bereits deklarierten Funktionen wie `sin` orientieren. Anschließend ist noch der Zwischencode um einen neuen Befehl zu erweitern (Unit `lsCode.pas`). Es muß natürlich auch die Zwischencode-Abarbeitung dieses Befehls implementiert werden. Der Aufwand ist geringer, als es diese Aufzählung warscheinlich vermuten läßt!

3.2.3.3. Fehlerbehandlung

Ein Compiler kann nicht davon ausgehen, daß das ihm zur Übersetzung angebotene Quellprogramm fehlerfrei ist. Es ist im Gegenteil häufig mit Fehlern aller Art gespickt. Wichtige Fehlerklassen sind:

- Syntaxfehler. Es werden Symbole vergessen oder zuviel geschieben (meist Flüchtigkeitsfehler).
- Ein Bezeichner wird unterschiedlich geschrieben, etwa bei der Deklaration einer Variablen und deren Nutzung, bzw. mehrfach deklariert (Semantikfehler).
- Ein Ausdruck ist vom falschen Typ, etwa wenn eine als BOOLEAN deklarierte Variable innerhalb eines mathematischen Ausdrucks auftritt (Semantikfehler).
- Restriktionen. Ein Konstrukt wie der zum Fehler führende ist in der Sprache zwar erlaubt, aber mit vorliegendem Compiler nicht realisierbar. Beispiele hierfür sind zuviele Bezeichner (Überlauf interner Tabellen) sowie Zahlenbereichsüberschreitungen bei mathematischen Ausdrücken, soweit sie bereits beim Compilieren erkannt werden.

Fehler der ersten Art, also Syntaxfehler, werden in von COCO erzeugten Compilern automatisch erkannt: Daß ein Quellprogramm einen Syntaxfehler enthält, ist durch einen Vergleich mit der Grammatik leicht feststellbar. Schwieriger ist schon die Bestimmung der Fehlerursache: Wurden ein oder mehrere Token zuviel geschrieben oder aber ausgelassen? COCO verwendet einen intelligenten Fehlererkennungsmechanismus, der mit hoher Warscheinlichkeit die richtige Fehlerursache ermittelt (siehe [14, Seite 59 ff]). Es wird ein Fehler der Art *'... expected'* oder — wenn die Erkennung fehlschlug — ein *'syntax error'* gemeldet. Anschließend versucht der mit COCO erzeugte Compiler, die Fehlerstelle zu übergehen[13] und sich wieder mit dem Quellprogramm zu synchronisieren, um so in einer Compilierung möglichst viele Fehler im Quellprogramm aufzuspüren.

Alle erkannten Fehler werden in einer Liste von COCO gesammelt und können nach der Compilierung ausgewertet werden. In den Originalgrogrammen zu COCO wurde im Fehlerfall ein Listing des Quellprogramms mit eingestreuten Fehlermeldungen erzeugt. Dieses Verhalten ist bei einem interaktiven Entwicklungssystem wie Turbo-PASCAL oder LSC nicht erwünscht. Beim Auftreten eines Fehlers sollte der Editor aktiviert werden, der Cursor auf die Stelle im Quellprogramm, wo der Fehler erkannt wurde, plaziert werden sowie eine Beschreibung der Fehlerart am Editorrand angezeigt werden. Um diesen Effekt zu erreichen, waren einige diffizile Eingriffe in den von COCO erzeugten Compiler (`alexfram, cocosemf, cocosynf, lsError.pas`) sowie den Editor (`gTed.pas`) nötig.

[13] Auch das Übergehen von Fehlerstellen ist nicht trivial. Folgefehler sollen ausgeschlossen werden und der an der Fehlerstelle stehende Quelltext durch den von Programmautor warscheinlich gemeinten Konstrukt ersetzt werden, um eine sinnvolle Weitercompilierung zu ermöglichen.

Fehler der zweiten und dritten Art, Semanikfehler, muß der Programmierer im erzeugten Compiler selbst behandeln. Hilfreiche Routinen zum Aufsammeln der Fehler bis zum Ende des Compilierens und deren Auswertung sind in `lsError.pas` bereits enthalten. Man kann sich damit auf die Typ- und Bezeichnerüberwachung beschränken. So wird mit Hilfe von Grammatikattributen ([14]) der Typ von Bezeichnern überwacht. Als Beispiel folgt die (vereinfachte) Deklaration von Termen in LSC:

```
...
Term<out:typ2> =
  PFactor<out:typ2>
  { MO<out:mulop>
    PFactor<out:typ3>     sem IF (typ2<>typ3) THEN SError(7);
                              CASE mulop of
                                '*' : BEGIN
                                        IF typ2<>1 THEN SError(8);
                                        EmitOp(iMUL);
                                      END;
                                '/' : ...
                                'AND' : BEGIN
                                        IF typ2<>2 THEN SError(6);
                                        EmitOp(iAND);
                                      END;
                              END
                            endsem
  } .
...
```

Es werden die Typen der beiden Faktoren miteinander verglichen. Sind diese ungleich, wird ein Sematikerror *'Operand types dont match'* generiert. Bei der Auswertung des Multiplikationsoperators `mulop` erfolgt eine weitere Typüberprüfung. Bei algebraischen Ausdrücken müssen beide Operanden vom Typ 1 sein (float), bei Booleschen Ausdrücken vom Typ 2. Im Fehlerfalle wird wieder eine entsprechende Meldung erzeugt (*'Float type expected'* bzw. *'Boolean type expected'*).

Damit der Compiler weiterarbeiten kann, sollte auch ein Semantikfehler restauriert werden. Dies betrifft z.B. eine explizite Typzuweisung an die Attribute der Grammatik. Im obigen Beispiel ist das nicht notwendig; ein Term erhält bei dieser Definition automatisch den Typ des ersten Operanden. In der Deklaration von `Factor` findet man mehrere Beispiele zur Typkorrektur.

Falsch geschriebene oder noch nicht definierte Bezeichner werden als spezielles Objekt `undefobj` weitergeführt, doppelt deklarierte Bezeichner nur einmal in die Semantiktabelle aufgenommen (Prozedur `Declare` in `ls.atg`). Ein Auszug aus der Deklaration von `Factor` zeigt die typischen Pascal-Konstrukte für solche Typtests.

```
...
Factor<out:typ4> =
  ident<out:spix>          sem Find(spix, obj);   { search object spix }
                               IF obj=NIL THEN BEGIN    { if not found }
                                 SError(2); obj := undefobj; END
                               ELSE IF obj^.kind = iconst  { Konstante }
                                 THEN ...
                               ELSE IF obj^.typ = ifloat    { Variable }
                                 THEN ...
                               ELSE SError(3); typ4:=1; { unknown type }
                           endsem
...
```

In der vorliegenden Fassung werden von LSC die in Tabelle 3.2 aufgezeigten Semantikfehler gemeldet.

Nr.	Semantikfehler
1	*Identifier declared twice*
2	*Undeclared identifier*
3	*Variable expected*
4	*Types in assignment dont match*
5	*Procedure name expected*
6	*Boolean type expected*
7	*Operand types dont match*
8	*Float type expected*
9	*Integer type expected*
10	*LSI table full*
11	*Illegal number of parameters*
12	*Error in constant expression*

Tabelle 3.2.: Semantikfehler des LSC

Fehler der vierten Art, Restriktionen, führen im allgemeinen zu einem sofortigen Abbruch eines COCO-generierten Compilers. Da der Nutzer des Compilers kaum Möglichkeiten hat, diese Einschränkungen des Compilers zu umgehen, etwa in Form einer Fehlerkorrektur wie bei Syntax- und Semantikfehlern, erfolgt eine Fehlermeldung in einem speziellen Fenster. Der Nutzer kann jedoch versuchen, sein Quellprogramm so zu vereinfachen, daß eine Restriktion nicht mehr auftritt.[14]

[14] Natürlich können auch innerhalb des Compilers Eingriffe vorgenommen werden (Vergrößerung der Tabellen, größerer Zahlenbereich etc.), so daß die Ursache der Restriktion verschwindet. Doch treten Restriktionen beim Compilieren in LSC höchst selten auf. Häufiger anzutreffen sind Restriktionen bei der Berechnung der Ableitung. Hier kann der Nutzer von LSC durch Verringern der Ableitungstiefe versuchen, diese Restriktion zu umgehen.

Für den interessierten Leser sei noch bemerkt, daß die Fortsetzung eines Programms nach Auftreten eines Fehlers nicht problemlos ist. Vielfach wird ein Fehler irgendwo in den Tiefen des Parsers erzeugt. Um eine ordnungsgemäße Weiterarbeit des LSC zu garantieren, ist es notwendig, den Programmstack und andere für den Programmablauf wichtige Register zu restaurieren. Ein wertvolles Hilfsmittel hierzu sind die 'Far Jumps' aus [19, Kapitel 8].

3.2.3.4. Der erzeugte Zwischencode

Die gesamte Verwaltung des Zwischencodes sowie dessen Interpretation ist in der Unit `lsCode.pas` enthalten. Die hier implementierte virtuelle 2-Stack-Maschine nutzt dabei folgende Datenstrukturen:

Data : ARRAY[0..50] OF float	Datenspeicher für (lokale) Variablen
Code : ARRAY[1..30000] OF Char[15]	Codespeicher für den Zwischencode
pc : Word	program counter
stack : ARRAY[0..50] OF float	Stack für den mathematischen Parser

Die Instruktionen des Zwischencodes haben unterschiedliche Länge, der Operationscode belegt dabei ein Byte, ein Operand ein (symb), zwei (addr) oder SizeOf(float)[16] Bytes. Boolesche Werte werden als Floatzahlen gehandelt, wobei 0 für FALSE und 1 für TRUE steht. Die derzeit implementierten Instruktionen sind in folgender Tabelle aufgelistet, es steht dabei EingString für die letzte Ableitung und AusgString für die zu erzeugende Ableitung.

LOAD	addr	holt float aus data[addr] und legt sie auf Stack
STO	addr	kopiert alle Parameter aus EingString und bringt sie nach data[addr]
LITF	float	packt float auf Stack
ADD		addiert oberste zwei Stackwerte
SUB		subtrahiert oberste zwei Stackwerte
DIV		dividiert oberste zwei Stackwerte
MUL		multipliziert oberste zwei Stackwerte
NEG		negiert obersten Stackwert
AND		logisches AND der obersten zwei Stackwerte
OR		logisches OR der obersten zwei Stackwerte

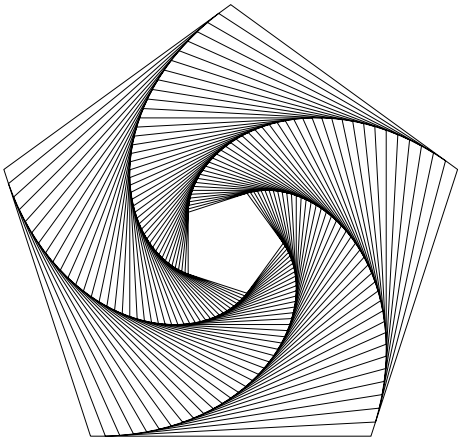
[15] Die Benutzung von Char anstelle von Byte im Zwischencode ist ein Überbleibsel aus früheren Fassungen des Programms. Dort wurden als Symbole eines L-Systems lediglich einbuchstabige Bezeichner zugelassen; diese wurden dann direkt (ohne Umweg über die LSI-Tabelle) in den Zwischencode integriert.

[16] Um LSC an verschieden ausgestattete PC's anzupassen, wurde der Zahlentyp *float* benutzt, der für Single, Double, Extended oder Real stehen kann. Demzufolge belegt eine Float-Zahl vier, acht, zehn oder sechs Byte Speicher.

NOT		logisches NOT des obersten Stackwertes
EQU		= Test der obersten zwei Stackwerte
LSS		< Test der obersten zwei Stackwerte
GTR		> Test der obersten zwei Stackwerte
LEQ		<= Test der obersten zwei Stackwerte
GEQ		>= Test der obersten zwei Stackwerte
UEQ		<> Test der obersten zwei Stackwerte
GBYT	symb	generiert Symbol im AusgString
GFLOT		holt float vom Stapel und schreibt ihn in AusgString
RET		Regelende bei erfolgter Ersetzung
FRET		Regelende bei Mißerfolg
ADNX	addr	Adresse der nächsten Regel bei Mißerfolg
FJMP	addr	holt bool vom Stapel und Sprung zu addr, wenn falsch
LCTX		Beginn linker Kontext
RCTX		Beginn rechter Kontext
CEND		Kontextende
CBYT	symb	Test, ob Symbol symb in EingString folgt (Kontexttest)
RND		packt Zufallszahl auf Stapel ($0 \leq rnd < 1$)
POWR		potenziert oberste zwei Stackwerte
INT		ganzzahliger Teil des obersten Stackwertes (wie in PASCAL)
SIN		Sinus des obersten Stackwertes (in Grad!)
RETC		Ende bei Konstantenberechnung

Abschließend soll der erzeugte Code für folgendes L-System aufgeführt werden:

```
/*****************************************************************/
/* verschachtelte Polygone (n-Eck, um a Grad ineinander gedreht)   */
/*****************************************************(c)*vp'93**/
const n=5    a=3
depth 300
axiom P(n,1)
rules P(n,x)          --> P1(n,x,n);
      P1(n,x,d) :d>0 --> F(x)+(360/n)P1(n,x,d-1)
                     --> f(x*sin(a)/(sin(a)+sin(360/n-a)))+(a)
                         P(n,x*sin(180-360/n)/(sin(a)+sin(360/n-a)));
```



```
LSI-Tabelle
;  1 P          addr:   22 parameter:  2 ignore: FALSE
;  2 P1         addr:   46 parameter:  3 ignore: FALSE
;  3 F          addr:    0 parameter:  1 ignore: FALSE
;  4 +          addr:    0 parameter:  1 ignore: FALSE
;  5 f          addr:    0 parameter:  1 ignore: FALSE

erzeugter Zwischencode
;axiom
                         { P(5,1) }
     1: GBYT       1     {    Ausgabe P }
     3: LITF    5.00     {    Ausgabe 5, 5 ist als Konstante def. }
    10: GFLOT
    11: LITF    1.00     {    Ausgabe 1 }
    18: GFLOT
    19: GBYT       1     {    Ausgabe P }
    21: RET              { Ende des Axioms }
;rule P
    22: ADNX       0     { nur eine Regel fuer P }
    25: STO        0     { Variablen n,x speichern }
                         { P1(n,x,n) }
    28: GBYT       2     {    Ausgabe P1 }
    30: LOAD       0
    33: GFLOT            {    Ausgabe n }
    34: LOAD       1
    37: GFLOT            {    Ausgabe x }
    38: LOAD       0
    41: GFLOT            {    Ausgabe n }
    42: GBYT       2     {    Ausgabe P1 }
```

```
    44: RET              { Ende der Regel }
    45: FRET
;rule P1
    46: ADNX       0     { nur eine Regel fuer P1 }
    49: STO        0     { Variablen n,x,d speichern }
                         { Bedingung d>0 ? }
    52: LOAD       2     {    d }
    55: LITF    0.00     {    0 }
    62: GTR              {    > }
    63: FJMP     115     {    nein --> weiter bei 115 }
                         {    ja   --> erster Regelkoerper }
                         { F(x) }
    66: GBYT       3     {    Ausgabe F }
    68: LOAD       1
    71: GFLOT            {    Ausgabe x }
    72: GBYT       3     {    Ausgabe F }
                         { +(360/n) }
    74: GBYT       4     {    Ausgabe + }
    76: LITF  360.00     {    360 }
    83: LOAD       0     {    n }
    86: DIV              {    / }
    87: GFLOT            {    Ausgabe Ergebnis }
    88: GBYT       4     {    Ausgabe + }
                         { P1(n,x,d-1) }
    90: GBYT       2     {    Ausgabe P1 }
    92: LOAD       0
    95: GFLOT            {    Ausgabe n }
    96: LOAD       1
    99: GFLOT            {    Ausgabe x }
   100: LOAD       2     {    d }
   103: LITF    1.00     {    1 }
   110: SUB              {    - }
   111: GFLOT            {    Ausgabe Ergebnis }
   112: GBYT       2     {    Ausgabe P1 }
   114: RET              { Ende Regelkoerper }
   115: ...              { Code des zweiten Regelkoerpers fuer P1 }
                         { ist analog aufgebaut, lediglich die }
                         { mathematischen Operationen sind }
                         { umfangreicher }
   237: RET
   238: FRET
;
```

3.3. Bildung der Ableitung

Im zweiten Schritt der Erzeugung eines L-Systems wird mit Hilfe der internen Darstellung einer L-System–Beschreibung, des Zwischencodes, die Ableitung des L-Systems berechnet. Dazu wird Algorithmus 2.1 aus Abschnitt 2.2.2 verfeinert:

- Es muß einer konkreten Speicherverwaltung Rechnung getragen werden.
- Es müssen Algorithmen zur Kontextsuche eingebaut werden.

Umgesetzt wurden diese Algorithmen in den Units `lsStrng.pas` (Speicherverwaltung) sowie `lsCode.pas` (Ableitungsbildung).

3.3.1. Speicherverwaltung

Leider kann – wie in Abschnitt 2.2.2 bereits erwähnt – die Ableitung eines L-Systems Größenordnungen im Megabytebereich annehmen. Ein Programm, das L-Systeme mit Algorithmus 2.1 generiert, sollte demzufolge mit Strings dieser Größenordnung hantieren können. Doch das veraltete Segmentierungsmodell eines 80x86-Prozessors mit seiner Beschränkung auf Segmente einer maximalen Größe von 64 KByte stellt hierzu hohe Anforderungen an das Programm; besonders, wenn eine solches Speicherverwaltung in der Implementationssprache (also Turbo-PASCAL) *nicht* vorgesehen ist. In ersten Versionen von LSC wurde daher mit 64 KByte großen Arrays gearbeitet. Erst in der neuesten Version von Turbo PASCAL, *Borland PASCAL with Objects 7.0*, existiert ein Objekt zur Nutzung großer Speichermengen: Der Typ `TMemoryStream` ist zur Ableitungsbildung bei L-Systemen ideal geeignet. Leider sind über dieses Objekt kaum Informationen erhältlich, lediglich eine kurze Erwähnung im Turbo-Visions–Handbuch war zu finden. So war ein Verständnis für `TMemoryStream` nur durch Experimente zu erwerben; aus Zeitgründen konnte die Unit `lsStrng.pas` nicht weiter optimiert werden. Dennoch sind mit dieser Version Stringlängen von mehreren Megabyte möglich, die Stringlänge wird nur durch den verfügbaren Speicher begrenzt!

Ein String wird als lineare Liste von Symbolen, repräsentiert durch ihre Positionsnummern in der LSI-Tabelle, und ihren Parametern, *float*-Zahlen, verwaltet. Ein Zugriff auf die Elemente der Liste erfolgt sequentiell, intern wird aber zur Kontextsuche ein wahlfreier Zugriff benötigt.

Ein Symbol ohne Parameter wird dabei einfach durch seine Nummer repräsentiert. Bei einem Symbol mit Parametern werden die Symbolnummer, dann die einzelnen Parameter und anschließend nochmals die Symbolnummer (zwecks Symmetrie für die Kontextsuche) geschrieben. Anfang und Ende eines Strings werden durch die Symbolnummer 0 gekennzeichnet. Ein kleines Beispiel möge dies verdeutlichen:

$$F + G(1.37,-12)\ F \ldots$$

wird im String wie folgt abgelegt:

0	**F**	+	**G**	1.37	-12	**G**	**F**	···	**0**

Die Strings werden in der Unit `lsStrng.pas` als Objekte des Typs `TStrng` deklariert, ihre wichtigsten in LSC genutzten Methoden sind:

`Open`	setzt Zeiger auf Stringanfang
`Close`	schreibt Endesymbol 0
`Reset2ActSymbol`	setzt Zeiger auf aktuelles Symbol zurück
`Set2NextSymbol`	setzt Zeiger auf nächstes Symbol
`GetSymbol : Byte`	liest nächstes Symbol aus String
`GetFloat : Float`	liest nächsten Parameter aus String
`GetAllFloats(VAR Dest)`	liest alle Parameter nach Dest
`WriteSymbol(aByte : Byte)`	schreibt Symbol in String
`WriteFloat(aFloat : Float)`	schreibt Parameter in String
`Copyfrom(VAR aStrng : TStrng; ByteAnz : Word)`	kopiert ByteAnz Bytes aus aStrng
`ContextSearch(aByte : Byte)`	Kontextsuche nach aByte.

3.3.2. Abarbeitung des Zwischencodes

Die Ableitung wird in der Prozedur `MakeDerivation` analog dem Algorithmus 2.1 gebildet, `Strng1` ist dabei die letzte Zwischenableitung und `Strng2` die nächste zu bildende Ableitung.

```
PROCEDURE MakeDerivation;
BEGIN
  actDepth := 1;                              { aktuelle Rekursionstiefe }
  Strng1^.Open; Strng2^.Open;
  Interpret(1, ok);                             { Axiom uebernehmen }
  Strng2^.Close;

  WHILE actDepth <= Options.Depth DO BEGIN
    swap(Strng1, Strng2);                       { Strings vertauschen }
    WITH Strng1^ DO BEGIN
      Open; Strng2^.Open;
      aSymbol := GetSymbol;                     { erstes Symbol holen }
```

```
      WHILE aSymbol <> 0 DO BEGIN { solange Stringende nicht erreicht }
        IF LSI[aSymbol].Addr <> 0 THEN BEGIN   { wenn SymbolRegel ex. }
          Interpret(LSI[aSymbol].Addr, ok);   { dann diese abarbeiten }
          WHILE (NOT ok) AND (nextpc<>0) DO BEGIN
            Reset2ActSymbol;                 { sonst Zuruecksetzen und }
            Interpret(nextpc, ok);   { weitere passende Regeln nehmen }
          END;
          IF NOT ok THEN CopySymbol;   { ansonsten direkte Uebernahme }
        END ELSE CopySymbol;              { auch wenn keine Regel ex. }
        Set2NextSymbol;
        aSymbol := GetSymbol;               { naechstes Symbol holen }
      END;                                     { usw. bis Stringende }
      Strng2^.Close;
      Inc(actDepth);                   { akt. Ableitungstiefe erhoehen }
    END;                                { und erneute Ableitung bilden }
  END;                      { bis gewuenschte Ableitungstiefe erreicht }
END;
```

Der Algorithmus 2.1 wird hier um die Verarbeitung *mehrerer* möglicher Regeln für ein Symbol erweitert: Bei der Interpretation des Zwischencodes für ein Symbol mit `Interpret` wird die Boolesche Variable `ok` bei einer erfolgreichen Abarbeitung der Regel auf `TRUE` gesetzt; die Variable `nextpc` erhält die Startadresse des Zwischencodes der nächsten zu diesem Symbol gehörenden Regel (s.a. Abschnitt 3.2.3.1). Dadurch können in der inneren WHILE–Schleife *alle* zu einem Symbol gehörenden Regeln zum Einsatz kommen (natürlich nur, falls die vorhergehenden Regeln nicht erfolgreich verwendet werden konnten). Wurde keine bzw. keine passende Regel gefunden, so wird mit `CopySymbol` das aktuelle Symbol mitsamt seinen Parametern in den neuen Ableitungsstring `Strng2` kopiert.

Die Funktion Interpret übernimmt dabei die Abarbeitung des Zwischencodes einer zum aktuellen Symbol gehörenden Regel, die zugehörige Adresse wird in der Variablen `pc` übergeben.

Mathematische und Boolesche Befehle werden in `Interpret` – wie in Abschnitt 3.2.3.2 erläutert – mit einer virtuellen Stackmaschine abgearbeitet. Steuerbefehle (RET, FJMP, ADNX u.a.) bewirken eine Änderung in der sonst linearen Abarbeitung des Zwischencodes. Weitere Befehle dienen zum Variablentransport (LOAD, STO) oder zur Nutzung von Konstanten (LITF). Die Befehle zur Kontextverarbeitung (LCTX, RCTX, CEND, CBYT) greifen auf Methoden der Strings zurück und werden im folgenden Abschnitt weiter erläutert.

```
PROCEDURE Interpret(pc: Word; VAR ok:Boolean);
BEGIN
  ok:=FALSE;
  Strng1^.ContextSearchDirection := noContext; { keine Kontextsuche }
```

```
  WHILE TRUE DO BEGIN
    CASE Instruction(Next) OF        { naechsten Zwischencode holen }

    { math. Operationen, analog  SUB, DIV, MUL, POWR, NEG, INT, RND }
        SIN:   Push(Sin(Pop*Pi/180));  { Push und Pop beziehen sich }
        ADD:   Push(Pop+Pop);                  { auf den Float-Stack }
    { logische Operationen und Vergleiche, analog aufgebaut sind }
    { OR, NOT, EQU, LSS, GTR, LEQ, GEQ, UEQ }
        AND:   IF (Pop=1) AND (Pop=1) THEN Push(1) ELSE Push(0);
    { false jump, die IF NOT ... - Bedingung }
        FJMP:  adr:=NextWord; IF Pop=0 THEN pc:=adr;
    { packe die naechste Zahl (Konstante) auf Stack }
        LITF:  Push(float(Addr(code[pc])^)); Inc(pc,SizeOf(float));
    { generiere Zahl bzw. Symbolname in neuer Ableitung }
        GFLOT: Strng2^.WriteFLoat(Pop);
        GBYT:  Strng2^.WriteSymbol(Byte(code[pc])); Inc(pc);
    { Ende des Zwischencodes fuer ein Symbol }
        RET:   ok:=TRUE; Exit;      { bei erfolgreicher Abarbeitung }
        FRET:  Exit;                                        { sonst }
    { Adresse der naechsten Regel fuer das aktuelle Symbol }
        ADNX:  nextpc:=NextWord;
    { holt Variable und legt sie auf den Stack }
        LOAD:  Push(data[NextWord]);
    { zu Anfang aufrufen: kopiert alle Parameter des Symbols in die }
    { lokalen Speicher }
        STO:   Strng1^.GetAllFloats(data[NextWord]);
    { Kontextsuche }
        LCTX:  Strng1^.ContextSearchDirection := left;
        RCTX:  Strng1^.ContextSearchDirection := right;
        CEND:  Strng1^.ContextSearchDirection := noContext;
               Reset2ActSymbol;
        CBYT:  IF NOT Strng1^.ContextSearch(Byte(Next)) THEN Exit;
    { Kontantenberechnung beim Compilieren }
        RETC:  ConstFloat := Pop; ok := True; Exit;
    END;
  END;
END;
```

Die hierbei aufgerufenen Funktionen `Next` und `NextWord` holen aus dem Zwischencodespeicher ein Byte bzw. ein Word. Bei ihrem Aufruf wird gleichzeitig der *program counter* `pc` entsprechend erhöht; damit zeigt dieser wie bei einer realen Registermaschine stets auf den nächsten auszuführenden Zwischencodebefehl.

3.3.3. Kontexttest

Kontext ist bei L-Systemen der links beziehungsweise rechts vom aktuellen System stehende String in einer (Zwischen-)Ableitung. Der Kontextbegriff wird bei Prusinkiewicz auf *'bracked LS'*, L-Systeme mit Verzweigungssymbolen [und], erweitert ([12, Seite 32]):

- Im Kontext zu einem Symbol sind in '[' und ']' eingeschlossene Strings zu überlesen. Man beachte dabei auch eine mögliche Schachtelung der Klammern.
- Ein rechter Kontext endet an einer überzähligen rechten Klammer ']', der linke Kontext wird auch über eine überzählige linke Klammer ']' hinaus gesucht.
- Im L-System als *ignore* deklarierte Symbole sind bei der Kontextsuche zu übergehen.
- Aufgrund der speziellen Bedeutung der Klammern ist eine Deklaration von ihnen als *ignore* wirkungslos. Sie werden stets als Verzweigungssymbole interpretiert.

Außerdem gelten noch folgende Vereinbarungen:

- Ein linker bzw. rechter Kontext kann aus einer beliebigen Anzahl von Symbolen bestehen.
- Ein nicht vorhandender oder zu kurzer Kontext, etwa am Anfang des Strings, wird als fehlerhafter Kontext gewertet.

Auf der Basis all dieser Vereinbarungen wurde Algorithmus 3.1 entwickelt und in LSC durch `ContextSearch` zum Testen *eines* im Kontext erwarteten Symbols implementiert.

Algorithmus 3.1 (Kontexttest)

```
{ Ueberlesen von geklammerten Symbolen bzw. eines ignore-Symbols }
function UeberlesenRechts : Boolean;
begin
  UeberlesenRechts := true;
  if Symbol = '[' then                  { Verzweigungen ueberlesen }
    tiefe := 1;
    repeat
      GetNextSymbolRight;      { hole naechestes Symbol aus String }
      if Symbol = '[' then Inc(tiefe) endif;
      if Symbol = ']' then Dec(tiefe) endif;
    until (tiefe=0) or (Symbol = 0);
    if Symbol = 0 then UeberlesenRechts := false
      else GetNextSymbolRight;             { letzte ']' uebergehen }
```

```
    endif;
    if LSI[symbol].ignore then
      GetNextSymbolRight;       { zu ignorierende Symbole ueberlesen }
    endif;
    UeberlesenRechts := false;     { sonst war nix zum Ueberlesen da }
  end;

  { Test, ob aByte im Kontext folgt ... }
  function ContextSearch(aByte : Byte) : Boolean;
  begin
    if ContextSearchDirection = right then
      GetNextSymbolRight;                    { hole rechten Kontext }
      while UeberlesenRechts do       { Uebergehen zu ignor. Symbole }
      if (Symbol = 0) OR (Symbol = ']') then
        ContextSearch := false;
      else ContextSearch := Symbol = aByte;    { Test auf Gleichheit }
    endif;
    if ContextSearchDirection = left then
    ...                   { analog rechtem Kontext mit Funktionen wie }
                               { GetNextSymbolLeft und UeberlesenLeft }
    endif;
    { Der Zeiger des Strings steht nun auf der richtigen Position,um }
    { bei wiederholten Aufrufen dieser Funktion den gesamten Kontext }
    { zu testen }
  end;
```

Der Test eines Kontexts, der länger als ein Symbol ist, erfolgt durch wiederholte Abarbeitung des Zwischencodebefehls CBYT und der sich daraus ergebenden mehrfachen Aufrufe von ContextSearch.

Der linke Kontexttest wurde durch die Umordnung der zu suchenden Symbole bestens vorbereitet (siehe Abschnitt 3.2.3.1) und wird damit in völlig analoger Weise wie oben beschrieben abgearbeitet.

Bei der praktischen Arbeit mit LSC hat es sich gezeigt, daß eine effektive Abarbeitung des Kontexttests unbedingt notwendig ist, speziell das Übergehen von geklammertem Ausdrücken sollte zeitsparend implementiert werden. Leider war dies bei Streams und den in LSC möglichen parametrischen Symbolen in der Kürze der Zeit nicht so leicht zu verwirklichen; hier sollte das Programm bei Bedarf optimiert werden.[17]

[17] Der Leser möge bei Bedarf die Programmausführungszeiten des Programms PFG von P. Prusinkiewicz, optimiert für einen 80386 kompiliert, und die Zeiten von LSC miteinander vergleichen: LSC benötigt ein Vielfaches der Rechenzeit wie PFG. Dies liegt zum einen an der allgemeineren Syntax parametrischer L-Systeme und zum anderen an dem Zeitoverhead, den eine Speicherverwaltung für Megabytestrings mit sich bringt (PFG arbeitet nur mit nichtparametrischen L-Systemen und einer maximalen Stringlänge von 32 KByte).

3.4. Interpretation der Ableitung

Nach der Berechnung der Ableitung steht diese im Speicher zur Interpretation bereit. Die einfachste Interpretation ist die Ausgabe der Ableitung als Text: Die Symbolnamen werden zusammen mit ihren Parametern im Klartext in ein File geschrieben, dies kann dann beliebig verwendet werden. Sinnvoll ist so etwas bei der Betrachtung der L-Systeme vom Standpunkt formaler Grammatiken aus, etwa zur Darstellung paralleler Algorithmen [13]. Es existiert daher auch eine Textvariante LSC-TXT meines Programms, welches ebenfalls die Zwischenableitungen mit ausgibt.

Die am häufigsten anzutreffende Interpretationsart ist jedoch die graphische Darstellung mit Hilfe der Turtlegraphik. Diese wird nun in einer komfortablen dreidimensionalen Version beschrieben.

3.4.1. 3D-Turtlegraphik

3.4.1.1. Theoretische Grundlagen

Eine zweidimensionale Turtle, beschrieben durch ihre Turtlekoordinaten (x, y, α) mit x und y als 2D-Koordinaten (*turtleposition*) und α als Zeichenrichtung (*heading*), kann auch als Vektor $\vec{H} = (\cos\alpha, \sin\alpha)$, also als Einheitsvektor, angeheftet im Punkt (x, y), aufgefaßt werden. Diese Betrachtung erlaubt eine sinnvolle *Erweiterung* der zweidimensionalen Turtle um eine weitere Dimension [11]: An die Stelle *eines* Vektors tritt ein rechtwinkliges Koordinatensystem aus *drei* orthonormalen Vektoren $\vec{H}, \vec{L}, \vec{U}$. Sie stehen für die Bewegungsrichtung der Turtle (*heading*), die Ausrichtung nach links (*left*), und die Ausrichtung nach oben (*up*). Es gilt $\vec{H} \times \vec{L} = \vec{U}$. Die Rotation der Turtle kann nun mit folgender Formel berechnet werden:

$$\left[\vec{H}'\ \vec{L}'\ \vec{U}'\right] = \left[\vec{H}\ \vec{L}\ \vec{U}\right]\ \mathbf{R}$$

Dabei ist $\mathbf{R}$ eine 3x3-Rotationsmatrix. Die speziellen Drehungen um einen Winkel α um die Vektoren $\vec{H}, \vec{L}, \vec{U}$ werden durch folgende Matrizen beschrieben:

$$\mathbf{R}_U(\alpha) = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_L(\alpha) = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

$$\mathbf{R}_H(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

Die folgenden Turtlesymbole steuern die Drehung der Turtle im Raum (Abbildung 3.5):

+	Drehung nach links um Winkel δ. Die Rotationsmatrix ist $\mathbf{R}_U(\delta)$.
−	Drehung nach rechts um Winkel δ. Die Rotationsmatrix ist $\mathbf{R}_U(-\delta)$.
&	Senkung um Winkel δ. Die Rotationsmatrix ist $\mathbf{R}_L(\delta)$.
∧	Anhebung um Winkel δ. Die Rotationsmatrix ist $\mathbf{R}_L(-\delta)$.
\	Rollen nach links um Winkel δ. Die Rotationsmatrix ist $\mathbf{R}_H(\delta)$.
/	Rollen nach rechts um Winkel δ. Die Rotationsmatrix ist $\mathbf{R}_H(-\delta)$.
|	Drehung in entgegengesetzte Richtung. Die Rotationsmatrix ist $\mathbf{R}_U(180°)$.

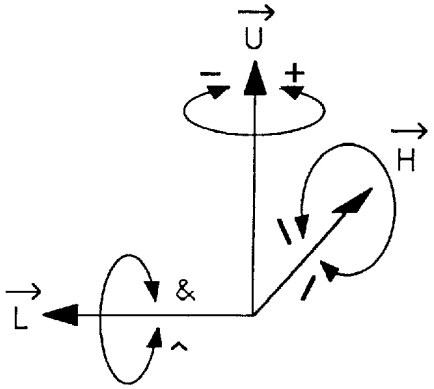


Abbildung 3.5.: Steuerung der Turtle im Raum

Bewegt wird die Turtle bei F und f in Richtung des Vektors $\vec{H}$.

Durch obige Konstruktionen ist eine Abwärtskompatibilität zur normalen zweidimensionalen Turtle gegeben, das heißt, *alle* zweidimensionalen L-Systeme ergeben bei dreidimensionaler Interpretation (und geeigneter Projektion) dasselbe Bild wie bei zweidimensionaler Interpretation.

Zur Konstruktion von Pflanzen ist eine Erweiterung der Turtlebefehle um den Befehl $ sinnvoll [12, Seite 57]. Das Symbol $ dreht die Turtle um ihre Achse $\vec{H}$ derart, daß der Vektor $\vec{L}$ in eine waagerechte Position gebracht wird. Dies wird durch folgende Gleichungen erreicht:

$$\begin{aligned} \vec{L} &= \frac{\vec{V} \times \vec{H}}{|\vec{V} \times \vec{H}|} \\ \vec{U} &= \vec{H} \times \vec{L} \end{aligned}$$

Dabei ist $\vec{V}$ ein der Schwerkraft entgegengerichteter Vektor.

3.4.1.2. Implementation

Die dreidimensionale Turtle wurde in der Unit `Turtle3D.pas` realisiert. Sie umfaßt neben Routinen zur Drehung und Bewegung der Turtle im Raum auch Routinen zum Aufbau gefüllter Polygone und die Verwaltung des Turtlestacks. Dabei ist diese Unit *unabhängig* von L-Systemen; kann also auch für andere Zwecke als nur zur Interpretation von L-Systemen eingesetzt werden.

Der Turtlestatus einer Turtle umfaßt neben ihrer aktuellen Position im Raum und ihrer aktuellen Ausrichtung, bestimmt durch die Matrix $\left[\vec{H}\ \vec{L}\ \vec{U}\right]$, auch noch eine Reihe weiterer Informationen:

- Die Zeichenfarbe `Color`.
- Die Farbe zum Ausfüllen von Polygonen `FillColor`.
- Die Linienstärke `LineWitdh`.
- Die Bildschirmkoordinaten der Turtle, also die entsprechenden projizierten 3D-Koordinaten `sx, sy`.
- Der aktuelle Drehwinkel `alpha`.

Dieser Turtlestatus wird in einem *Stack* mit den Routinen

`TurtlePushState`	aktuellen TurtleStatus sichern
`TurtlePopState`	Turtlestatus restaurieren

verwaltet. Der aktuelle Turtlestatus steht in der Recordvariablen `Turtle` zur Verfügung.

Die Rotation der Turtle im Raum erfolgt mit folgenden Prozeduren:

`TurtleSetAngle(w : Float)`	Drehwinkel setzen
`TurtleTurnRightH`	Drehung um Drehwinkel um Achse H
`TurtleTurnRightL`	Drehung um Drehwinkel um Achse L
`TurtleTurnRightU`	Drehung um Drehwinkel um Achse U
`TurtleTurnLeftH`	Drehung um negativen Winkel um Achse H
`TurtleTurnLeftL`	Drehung um negativen Winkel um Achse L
`TurtleTurnLeftU`	Drehung um negativen Winkel um Achse U
`TurtleTurnHorizontal`	Drehung der Achse $\vec{L}$ ins Waagerechte

In diesen Routinen sind die Matrixmultiplikationen entsprechend Abschnitt 3.4.1 umgesetzt. Dabei wurde, um die 3D-Graphik hinreichend schnell zu gestalten, die rechenaufwendige Bestimmung von sin und cos aus den Matrixmultiplikationen heraus in die Routine `TurtleSetAngle` verlagert.[18]

Das Zeichnen gefüllter Polygone geschieht mit

`TurtleBeginFill`	Polygonmodus einschalten
`TurtleFill`	Polygon zeichen
`TurtlePushFill`	Setzen eines Polygonpunktes

Ist der Polygonmodus eingeschaltet, werden *der aktuelle und alle weiteren* von der Turtle angesteuerten Punkte in einem Array gesammelt. Ein Punkt kann auch durch expliziten Aufruf von `TurtlePushFill` diesem Array hinzugefügt werden. Mit `TurtleFill` werden alle Punkte dieses Arrays in ihrer Abfolge miteinander in der Zeichenfarbe `Color` verbunden, das so gezeichnete Polygon mit der Farbe `FillColor` ausgemalt und der Polygonmodus wieder verlassen.

Die Bewegung der Turtle erfolgt durch

`TurtleMove(s : Float)`	Verschieben der Turtle um s in Richtung $\vec{H}$
`TurtleDraw(s : Float)`	Zeichnen einer Linie der Länge s
`TurtleGDraw(s : Float)`	Zeichnen einer Linie der Länge s *ohne* eine Polygonpunkt zu markieren

Eine Linie wird dabei in der Zeichenfarbe `Color` und der Linienstärke `LineWitdh` gezeichnet.

Weitere Routinen der Unit `Turtle3D.pas` sind:

`TurtleReset`	Anfangswerte der Turtle setzen
`TurtleColor(Color : Byte)`	Zeichenfarbe setzen
`TurtleFillColor(fColor : Byte)`	Füllfarbe setzen

Mit `TurtleReset` wird ein definierter Turtlestatus gesetzt: die Turtle wird im Koordinatenursprung plaziert, die Matrix $\left[\vec{H}\ \vec{L}\ \vec{U}\right]$ und der Stack für den TurtleStatus initialisiert, der Polygonmodus abgeschaltet, der Drehwinkel ist 90°, Füll- und Zeichenfarbe sind weiß, die Linienstärke eins.

[18] Weitere Optimierungen der 3D-Graphik erfolgen durch die Ablage der Matrix $\left[\vec{H}\ \vec{L}\ \vec{U}\right]$ als *lineares Array* (Turbo PASCAL organisiert den Zugriff auf eine lineares Feld wesentlich schneller als den Zugriff auf ein zwei- oder mehrdimensionales Array) und die explizite Ausrechnung der einzelnen Matrixmultiplikationen gegenüber einer allgemeinen Multiplikationsroutine (vorteilhaft wegen der spärlichen Besetzung der Rotationsmatrizen $\mathbf{R}_U(\alpha), \mathbf{R}_L(\alpha), \mathbf{R}_H(\alpha)$).

Die Projektion der dreidimensionalen Turtlekoordinaten erfolgt in der Unit `Sys3D.pas`, die leicht abgeändert aus [17] entnommen wurde. In dieser sind allgemeine Routinen zur Parallel- und Punktprojektion eines rechwinkligen 3D-Koordinatensystems auf den Bildschirm enthalten. Die Koordinaten eines Punktes sind als INTEGER-Werte (-32768 bis 32767) anzugeben. Der Betrachterstandpunkt wird mit `SetCamera(zoom, projection, angleX, angleY, angleZ, dx, dy)` festgelegt.[19]

`zoom` ist der Vergrößerungsfaktor in Prozent. Der Wert 100 bedeutet, daß der gesamte INTEGER-Raum gerade so auf den Bildschirm paßt.

`projection` legt die Projektionsart fest: 0 bedeutet Parallelprojektion, Werte zwischen 1000 und 10000 aktivieren eine Punktprojektion. Der Wert legt dann die relative Entfernung des Betrachterstandpunktes zur Graphik fest.

`angleX, angleY, angleZ` sind die Drehwinkel. Bleiben diese auf 0, so betrachtet man die 3D-Graphik von vorn aus z-Richtung. Andere Werte drehen das Koordinatensystem um die entsprechenden Achsen. Dabei wird immer zuerst um die z-, dann um die y- und schließlich um die x-Achse gedreht.

`dx,dy` verschieben die Ausgabe am Bildschirm horizontal bzw. vertikal.

Die Projektion eines Punktes erfolgt mit `ProjVoxel`.

Leider wird in dieser Unit das Sichtbarkeitsproblem nicht angegangen; es wird zwar bei `ProjVoxel` eine Tiefeninformation berechnet, diese aber nicht weiter genutzt (etwa in einem sogenannten *z-Buffer*). Der Polygonmodus arbeitet mit Turbo PASCAL's `Graph.FillPoly` und berücksichtigt ebenfalls nicht verdeckte Graphikelemente. Man sollte daher dreidimensionale L-Systeme so konstruieren, daß sie 'von hinten nach vorn' gezeichnet werden.

3.4.1.3. Übersetzung der Ableitung

Die Ableitung wird in drei Stufen in eine Graphik übersetzt:

1. Durchforsten der LSI-Tabelle mit den im L-System verwendeten Symbolen nach solchen Symbolen, die von der Turtlegraphik interpretiert werden können.[20]

2. Erste Interpretation der Ableitung, um die Größe der 3D-Graphik zu berechnen und den Skalierungsfaktor zu bestimmen.

[19] In LSC können diese Werte interaktiv verändert sowie aktuelle Einstellungen in das L-System übernommen werden.

[20] Dieser Schritt dient zur Effizienzsteigerung der Interpretation. An die Stelle der zeitaufwendigen Stringvergleiche in den nächsten Schritten treten schnelle Zahlenvergleiche.

3. Zweite Interpretation zum Zeichnen der Graphik. Soll die Graphik mehrmals hintereinander von verschiedenen Betrachterstandpunkten aus gezeichnet werden, ist lediglich dieser dritte Punkt wiederholt aufzurufen.

All dies erfolgt in der Unit `Ls3DInterpret`.

Bei der Skalierung reicht es, die lageverändernden und weitere relevante Turtlebefehle zu berücksichtigen: Das Ändern der Zeichenfarbe oder das Füllen eines Polygons haben keinen Einfluß auf die Größe des zu zeichnenden L-Systems. Damit reicht folgende Routine:

```
PROCEDURE SetAngle;
BEGIN
  With Strng2^ DO
    IF noparam THEN BEGIN             { Wenn kein Parameter folgt, }
      IF (turtle.angle <> Options.angle) { ist der Winkel aktuell? }
      THEN TurtleSetAngle(Options.angle)     { nein --> neu setzen }
    END ELSE TurtleSetAngle(GetFloat);    { sonst Parameter nehmen }
END;

PROCEDURE GetCurveSize;          { Groesse des L-Systems ermitteln }
BEGIN
  TurtleReset; TurtleSetAngle(Options.angle); tlinewidth := 1;
  maxx := 0; maxy := 0; maxz := 0; minx := 0; miny := 0; minz := 0;
  With Strng2^ DO BEGIN
    Open;
    WHILE GetSymbol <> 0 DO BEGIN        { solange kein StringEnde }
      CASE TurtleKdo(LSI[aSymbol].Interpret) OF
        GDraw, Draw, Move : BEGIN                 { Turtle bewegen }
                        IF noparam THEN TurtlemoveEval(1)
                        ELSE TurtlemoveEval(GetFloat);
                        with turtle do begin          { Bildgroesse }
                          IF minx>x then minx:=x;     { korrigieren }
                          IF miny>y then miny:=y;
                          IF minz>z then minz:=z;
                          IF maxx<x then maxx:=x;
                          IF maxy<y then maxy:=y;
                          IF maxz<z then maxz:=z;
                        END
                      END;
                                              { Drehung der Turtle }
        TurnRightU : BEGIN SetAngle; TurtleturnRightU; END;
        TurnLeftU  : BEGIN SetAngle; TurtleturnLeftU;  END;
```

```
        TurnRightL : BEGIN SetAngle; TurtleturnRightL; END;
        TurnLeftL  : BEGIN SetAngle; TurtleturnLeftL;  END;
        TurnRightH : BEGIN SetAngle; TurtleturnRightH; END;
        TurnLeftH  : BEGIN SetAngle; TurtleturnLeftH;  END;
        TurnAround : BEGIN TurtleSetAngle(180);
                           TurtleTurnRightU; END;
        TurnHorizontal : TurtleTurnHorizontal;
        PushState  : TurtlePushState;         { TurtleStackbefehle }
        PopState   : TurtlePopState;
        DecDiam    : BEGIN IF noparam              { Linienstaerke }
                           THEN turtle.LineWidth:=turtle.LineWidth+1
                           ELSE turtle.LineWidth:=GetFloat;
                         IF Turtle.Linewidth>tlinewidth THEN
                           tlineWidth := Turtle.Linewidth;
                       END            { max. Linienstaerke ermitteln }
      END;
      Set2NextSymbol;                 { Pointer auf naechstes Symbol }
    END;
  END;
  { umrechnen auf Bildschirmkoordinaten }
  length := 32000/max(maxx-minx, maxy-miny, maxz-minz, 1)
END;
```

Beim Zeichnen des L-Systems müssen natürlich alle Turtlebefehle benutzt werden. Die Routine `PlotCurve` ist analog zu `GetCurveSize` aufgebaut und wird hier nicht extra wiedergegeben. Details, wie z.B. das Skalieren von Kreisen, können dem Quelltext von beiliegender Diskette entnommen werden.

3.4.2. TEX-Ausgabe

Außer der Darstellung am Bildschirm ist ein Ausdrucken des erzeugten L-Systems wünschenswert. Dies kann mit einer Hardcopyroutine geschehen — allerdings ist die Qualität aufgrund der relativ geringen Bildschirmauflösung gegenüber der Druckerauflösung gering. Da klassische L-Systeme nur aus Linien bestehen, ist eine Plotterausgabe des L-Systems etwa als HPGL-File denkbar. Eine weitere Variante ist das Erzeugen eines TEX–Input-Files: LATEX bietet mit der `picture`-Umgebung ebenfalls eine geräteunabhängige Vektorgraphik. Leider lassen sich mit den Standard–LATEX-Befehlen nur Linien betimmter Neigung zeichnen. Allgemeine Linien sind mit dem Zusatzpaket PICTEX konstruierbar; erfordern aber einen unverhältnismäßig hohen Zeitaufwand zu ihrer Berechnung. Eine weitere Möglichkeit zum Zeichnen von Linien bieten viele TEX-Gerätetreiber an, so auch die Treiber des emTEX-Paketes von Eberhard Mattes für IBM-PC's. Diese bieten u.a. auch den Vorteil, bereits bei einer Bildschirmausgabe die volle Graphik darzustellen.

In meinem Programm LSC werden die drei Befehle

```
\special{em:moveto},
\special{em:lineto},
\special{em:linewidth #1},
```

beschrieben in [7, Seite 41], verwendet. Es wird ein `input`-File mit einer eigenen `picture`-Umgebung von folgendem Aufbau generiert:

```
% TeX-file created with vp's LSC
% picture scaled 100 x max100
% POLYGON.TEX
\special{em:linewidth 0.4pt}
\begin{picture}(100,96)
\ifx\undefined\noLSC
  \put(19.0983005,0.0000000){\special{em:moveto}}
  \put(80.9016994,0.0000000){\special{em:lineto}}
  ...
  \put(61.5916695,42.5325402){\special{em:lineto}}
\else
  \put(0,0){\framebox(100,96){\jobname}}
\fi
\end{picture}
```

Das erzeugte Bild hat stets eine relative Breite von 100 und kann beliebig in TEX eingebunden werden. Eine Größenanpassung des Bildes erfolgt TEX-üblich mit `\unitlength` *x* `mm`. So wurde das Bild auf Seite 43 mit

```
\begin{center}
\unitlength0.6mm
\input polygon
\end{center}
```

auf eine Breite von 6 cm skaliert und horizontal zentriert dargestellt. Weitere Hinweise zum Einbinden der Bilder in TEX-Dokumente sind in der Programmbeschreibung von LSC zu finden.

Leider ist bei den Treibern des emTEX-Paketes die Anzahl der maximal auf einer Seite darstellbaren Linien auf 2730 beschränkt. Dadurch ist sowohl die Anzahl der L-Systeme pro Blatt eingeschränkt als auch die Komplexität eines einzelnen darstellbaren L-Systems.[21]

[21] Eine Optimierung durch Einsparung von `em:lines` erfolgt in LSC durch 'Kürzen' mehrfach hintereinander auftretender ']', siehe Unit `ls2DInterpret.pas`.

4. Spezielle Lindenmayer-Systeme

In diesem Kapitel werden einige mit LSC berechnete L-Systeme vorgestellt.

4.1. Stochastische Lindenmayer-Systeme

Bei stochastischen L-Systemen, wie sie in [12, Seite 28] eingeführt werden, gibt es zu einem Symbol *mehrere* Ersetzungsregeln. Bei jedem Aufruf des Symbols wird mit einer gewissen Warscheinlichkeit eine dieser Regeln *zufällig* ausgewählt.

Ein einfaches Beispiel aus [12] ist

$$\begin{aligned} \omega &: F \\ p_1 &: F \xrightarrow{.33} F[+F]F[-F]F \\ p_2 &: F \xrightarrow{.33} F[+F]F \\ p_3 &: F \xrightarrow{.34} F[-F]F \end{aligned}$$

Die Zahl über '→' ist hier die Warscheinlichkeit der Regel.

Eine solche Warscheinlichkeit ist in LSC nicht vorgesehen, kann aber über eine Bedingung leicht simuliert werden:

```
depth  5    angle  22.5
axiom  F
rules  F : rnd < 0.33 --> F[+F]F[-F]F;
       F : rnd < 0.5  --> F[+F]F;
       F  /* else */  --> F[-F]F;
```

Dieses L-System liefert dieselben Ergebnisse wie obiges L-System! Die Zahlenwerte ergeben sich aus der inneren Abarbeitungsstruktur von LSC: Konnte eine Regel nicht erfolgreich abgearbeitet werden, wird zur nächsten passenden Regel gegangen. In unserem Beispiel wurde die erste Regel mit einer Warscheinlichkeit von $1 - \frac{1}{3}$ *nicht* abgearbeitet, zusammen mit der Warscheinlichkeit $\frac{1}{2}$ ergibt dies eine Gesamtwarscheinlichkeit von $\left(1 - \frac{1}{3}\right) * \frac{1}{2} = \frac{1}{3}$ der zweiten Regel. Für die dritte Regel ergibt sich analog $\left(1 - \frac{1}{3}\right) * \left(1 - \frac{1}{2}\right) * 1 = \frac{1}{3}$ als Gesamtwarscheinlichkeit.

Bild 4.1 vermittelt einen Eindruck dieses L-Systems. Gleichzeitig wurde gegenüber obiger Version noch eine Abänderung vorgenommen und alle Regeln für F in einer Regel vereint. Eine solche Schreibweise ist meines Erachtens sinnvoller, da zusammengehörende Regeln sofort als solche erkannt werden. Außerdem kann LSC solche Konstrukte im allgemeinen schneller abarbeiten.

```
/* stochastic flower [Pru90, pp28] */
depth 5    angle 25
axiom fl fl fl fl fl
rules
  fl --> [turn(90)F]f(20);
  F : rnd<0.33 --> F[+F]F[-F]F
    : rnd<0.5  --> F[+F]F
               --> F[-F]F;
```

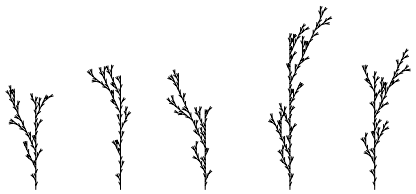


Abbildung 4.1.: stochastische Pflanzenstrukturen

4.2. Pflanzensimulationen

In Prusinkiewicz' Arbeiten [11, 12] sind zahlreiche Beispiele für Pflanzen angegeben. Ein solches Beispiel ist folgender in Abbildung 4.2 gezeigter Busch:

```
/* bush                                  [Pru90, pp.26] */
axiom
        color(17) A
rules
        A --> [&F L!A]/////'[&F L!A]///////'[&F L!A];
        F --> S/////F;
        S --> F L;
        L --> ['''^^{-f+f+f-|-f+f+f}];
options
        depth 6
        angle 22.5
        camera 210, -5, 31, -90, 0, -70
```

Ein weiteres Beispiel für eine Pflanzensimulation ist der von Michael F. Barnsley entwickelte Farn [9, Kapitel 5]. Die Idee, mittels *Collagen* über ein *iteriertes Funktionensystem IFS* Pflanzen zu erzeugen, läßt sich leicht auf L-Systeme übertragen.

Der Farn in Abbildung 4.3 wurde mit der in Abbildung 4.4 aufgezeigten Collage generiert, die Umsetzung in ein L-System war relativ einfach. Deutlich sind die einzelnen Teile der Collage im L-System–Text zu erkennen. Die Bedingung $x > 0.03$ sorgt im Zusammenhang

Abbildung 4.2.: ein einfacher dreidimensionaler Busch

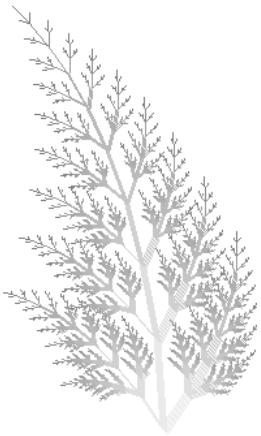


Abbildung 4.3.: ein Farn nach Barnsley

```
/* Farn                          vp'93 */
axiom
      turn(90) Farn(1)
rules
      Farn(x) : x>0.03 --> '!F(x)
                  [+(35) Farn(0.4*x)]
                  [-(35) Farn(0.4*x)]
                  [+(5) Farn(0.85*x)]
              --> F(x);
options
      camera 190, 0, 0, 0, 0, -90
      depth  10
```

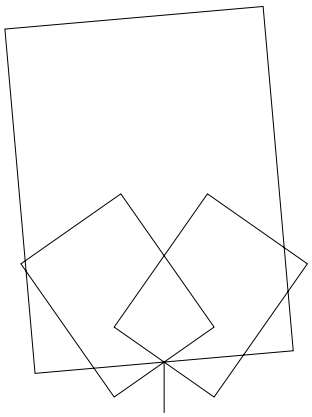


Abbildung 4.4.: L-System und Collage des Farns

mit dem `else`-Zweig der Regel für ein effizientes Zeichnen: Wird der aktuelle berechnete Teilfarn kleiner als ca. zwei Bildschirmpixel, wird die Rekursion für diesen Teilfarn abgebrochen und nur ein kleiner Strich gezeichnet. Optisch ergibt sich kein Unterschied, die Berechnung des Farns läuft jedoch wesentlich schneller als ohne diesen Trick.

4.3. Verschachtelte Polygone

Auf Seite 43 wurde ein weiteres L-System zur Erläuterung des Zwischencodes verwandt. Dieses L-System wird nun näher mit seinem mathematischen Hintergrund vorgestellt.

Ziel dieses L-Systems ist es, ein verkleinertes Abbild des regelmäßigen n-Ecks so in das äußere Polygon einzupassen, daß alle Ecken des inneren Polygons auf den Kanten des äußeren liegen. Dazu muß das innere n-Eck um einen (in Grenzen frei wählbaren) Winkel α gegenüber dem äußeren Polygon gedreht werden. Aus den Werten n und α und der Seitenlänge x des äußeren Polygons sind die Seitenlänge l des inneren Polygons und der Abstand a zwischen zwei zusammengehörenden Ecken des inneren und äußeren Polygons zu bestimmen.

Für das kleine Dreieck in Abbildung 4.5 gilt nach dem Sinussatz

$$\frac{\sin\alpha}{a} = \frac{\sin\gamma}{x-a} \tag{4.1}$$

$$\frac{\sin\alpha}{a} = \frac{\sin\beta}{l} \tag{4.2}$$

Im großen Dreieck gilt für die Innenwinkel

$$\frac{360^\circ}{n} + \frac{\beta}{2} + \frac{\beta}{2} = 180^\circ$$

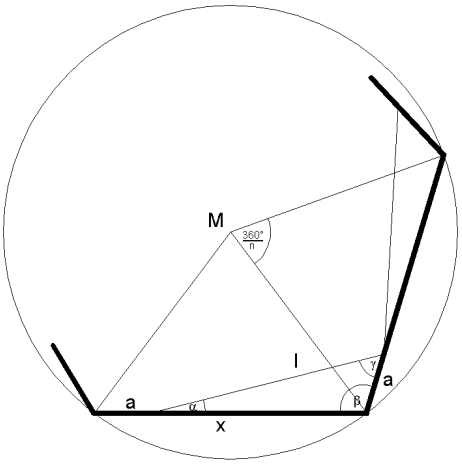


Abbildung 4.5.: Verschachtelte Polygone

Das ergibt

$$\beta = 180° - \frac{360°}{n} \tag{4.3}$$

Im kleinen Dreieck gilt für die Innenwinkel

$$\alpha + \beta + \gamma = 180°$$

Daraus folgt mit 4.3

$$\gamma = \frac{360°}{n} - \alpha \tag{4.4}$$

Aus 4.2 und 4.4 folgt für a

$$\begin{aligned} a &= \frac{\sin\alpha}{\sin\alpha + \sin\gamma}\, x \\ &= \frac{\sin\alpha}{\sin\alpha + \sin\left(\frac{360°}{n} - \alpha\right)}\, x \end{aligned} \tag{4.5}$$

Aus 4.2, 4.3 und 4.5 folgt für l

$$\begin{aligned} l &= \frac{\sin\beta}{\sin\alpha} a \\ &= \frac{\sin\beta}{\sin\alpha} \frac{\sin\alpha}{\sin\alpha + \sin\left(\frac{360°}{n} - \alpha\right)}\, x \\ &= \frac{\sin\left(180° - \frac{360°}{n}\right)}{\sin\alpha + \sin\left(\frac{360°}{n} - \alpha\right)}\, x \end{aligned} \tag{4.6}$$

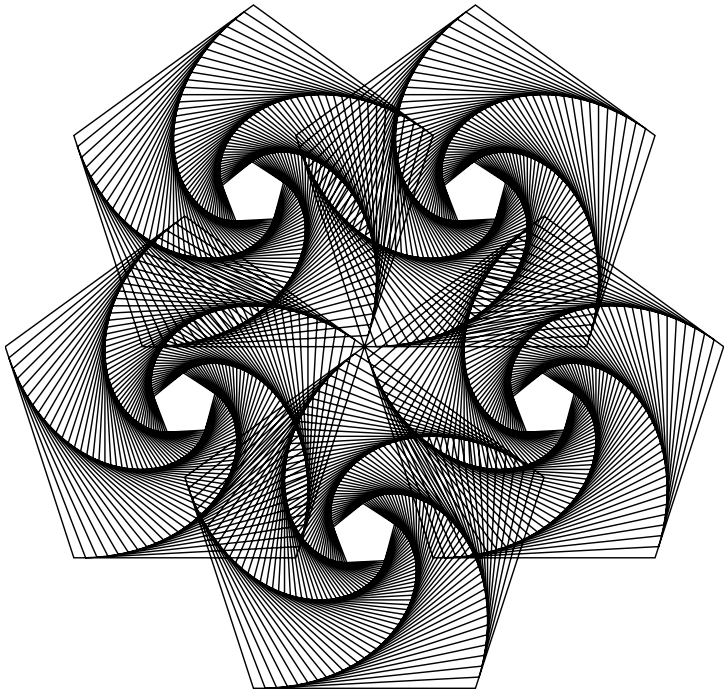


Abbildung 4.6.: eine Polygoncollage

Die erste Regel für `P` (...$d > 0$ ...) zeichnet ein regelmäßiges n-Eck, die zweite Regel dreht ein kleineres n-Eck in dieses hinein. Abbildung 4.6 zeigt eine Collage aus fünf verschachtelten Fünfecken, generiert mit dem L-System `Polygon3.lsc`.

```
/*******************************************************/
/* geschachtelte Polygone Version 3               vp'93 */
/*******************************************************/

const n = 5     /* Anzahl der Ecken */
      alpha = 3 /* Polygone um a Grad ineinander gedreht */

const n360  = 360/n
      a     = sin(alpha)/(sin(alpha)+sin(n360-alpha))
      l     = sin(180-n360)/(sin(alpha)+sin(n360-alpha))

depth 300
axiom P1 P1 P1 P1 P1
rules P1             --> [P(n,1,n)]+(n360);
      P(n,x,d) : d>0 --> F(x)+(n360)P(n,x,d-1)
           /* d=0 */ --> f(a*x)+(alpha)P(n,l*x,n);
```

5. Ausblick

Weder diese Diplomarbeit noch mein Programm LSC sind als 'Endprodukte' zu verstehen. Ich hoffe vielmehr, mit dieser Arbeit eine grundlegende Voraussetzung zur weiteren *praktischen* Untersuchung der L-Systeme geschaffen zu haben. Die parallel zu dieser Arbeit entstandene Diplomarbeit [4] geht bereits in diese Richtung.

Folgende Aspekte der L-Systeme scheinen mir eine weiterführende Untersuchung wert:

- Stochastische L-Systeme und ihr Verhältnis zu stochastischen Fraktalen.
- Parallele Algorithmen und ihre Umsetzung in L-Systeme.
- Programmieren mit L-Systemen.
- Modelle natürlicher oder technischer Objekte als L-Systeme formulieren.

Auch das Programm LSC sollte weiterentwickelt werden:

- Erweiterung der Turtlegraphik um einen Polygonstack.
- Erweiterung der 3D-Graphik zur realitätsnahen Darstellung (verdeckte Objekte durch *z-Buffer* überwachen, Raytracing, Rendering).
- Erweiterung der Grammatik um *Surfaces*.
- Beschleunigung der zeitaufwendigen Algorithmen zur Ableitungsbildung und Graphikgrößenberechnung; evtl. in Assembler formulieren.
- Routinen zum Bilderspeichern.
- Weitere Ausgabeformate: HPGL, Postscript, PICTEX.
- Modernere Oberfläche, Anpassung an Windows.
- Neufassung des Programms in C und Transport auf andere Plattformen (UNIX, NEXT, ...).

Schließlich wäre auch noch eine Erweiterung des Begriffs des L-Systems selbst denkbar: Anstelle einer linearen Ableitung wird ein Graph verwendet. Der Kontext eines Symbols ist dann ein Teilgraph. Folgende Fragen sollten beantwortet werden:

- Gibt es eine geeignete graphische Interpretation der erzeugten Ableitung? Oder eine anderweitige sinnvolle Interpretation?
- Welcher Zusammenhang besteht zwischen diesen L-Systemen und anderen bekannten mathematischen Systemen?
- Lassen sich mit solchen L-Systemen vielleicht Graphenprobleme sehr einfach lösen?
- ...

A. Inhalt der Diskette

Auf beiliegender HD-Diskette sind sämtliche Quelltexte meines Programms LSC, Beispieldateien, Dokumentation sowie weitere, teilweise in Kapitel 2 vorgestellte Programme zur graphischen Umsetzung von L-Systemen zu finden. Dabei wurden sämtliche Dateien aufgrund des Gesamtumfangs (mehr als 1,8 MByte Daten) in selbstextrahierende Archive gepackt. Die Programme können also *nicht* direkt von der Diskette aus gestartet werden!

Zur Installation ist das gewählte Archiv auf die Festplatte zu kopieren und dort aufzurufen. Nach dem Entpacken stehen die Programme und weitere zugehörige Dateien im aktuellen Verzeichnis auf der Festplatte und können dort wie gewohnt aufgerufen werden. In einigen Archiven sind weitere Archive verpackt; diese müssen ebenfalls expandiert werden.

Benötigte Hard- und Software. Es wird mindestens ein 286er mit 640 KByte RAM erwartet. Sinnvoll arbeiten läßt es sich mit den Programmen aber erst auf einem schnellen 386er oder einem 486er PC mit genügend freiem Speicher und VESA-VGA-Graphikkarte. LSC unterstützt den Einsatz eines Coprozessors. Zum Kompilieren der Programmquellen wird Turbo PASCAL 7.0 (mit Einschränkungen ist auch Turbo PASCAL 6.0 möglich) sowie Turbo C 2.0 benötigt. Die Dokumentation zu LSC liegt als emTEX-File vor und kann damit nur auf einem emTEX-System ausgedruckt werden.

Diskетteninhalt

Verzeichnis KAPITEL1

`!LSYSTEM.EXE` L-System von R. Gerike

`FRAME.CPP`	Graphikfensterverwaltung
`FRAME.H`	dazugehörige Headerdatei
`TURTLE3D.CPP`	3D-Turtlegraphik
`TURTLE2.H`	dazugehörige Headerdatei
`L-SYSTEM.CPP`	Hauptprogramm
`LS.EXE`	kompilierte Version (386er PC)
`*.LS`	Beispieldateien aus [2]

`!PFG.EXE` plant and fractal generator aus [11]

`!ORIGINA.EXE`	Archiv der originalen Quelltexte
`*.LS`	Beipieldateien
`GENERATE.C`	Ableitungsbildung
`GENERATE.H`	dazugehörige Headerdatei
`INTERPRE.C`	2D-Turtlegraphik
`INTERPRE.H`	dazugehörige Headerdatei
`PFG.EXE`	kompilierte Version (386er PC)
`GENERATE.EXE`	Text-Version

Verzeichnis KAPITEL2

`!LPASCAL.EXE` Umsetzung von L-Systemen in PASCAL-Programme

`LS2P*.PAS`	Beispiele
`UTURTLE2.PAS`	Turtlegraphik

Verzeichnis KAPITEL3

`!LSC-SRC.EXE` Quelltexte von LSC

`ALEXFRAM`	Hilfsdatei von ALEX
`COCOSEMF`	Hilfsdatei von COCO
`COCOSYNF`	Hilfsdatei von COCO
`LS.ATG`	Grammatik von LSC
`MAKELSC.BAT`	Batchdatei zum Compilieren der verschieden Versionen von LSC
`LSC-TXT.HLP`	Hilfstexte für LSC-Textversion
`LSC.HLP`	Hilfstexte für LSC
`ALEX.EXE`	Scannergenerator
`COCO.EXE`	Parsergenerator
`LS.LEX`	lexikalischer Scanner von LSC
`LS2DINTE.PAS`	2D-Turtleinterpretation
`LS3DINTE.PAS`	3D-Turtleinterpretation
`LSC-PM.PAS`	Hauptfile für *protected mode*-Version
`LSC-TXT.PAS`	Hauptfile für Textversion
`LSC.PAS`	Hauptfile

`LSC2D.PAS`	Hauptfile für Batchversion
`LSCODE.PAS`	Zwischencodeverwaltung
`LSERROR.PAS`	Fehlerbehandlung
`LSGLB.PAS`	allgemeine Definitionen
`LSINTERP.PAS`	Textinterpretation
`LSLEX.PAS`	lexikalischer Scanner, mit ALEX erzeugt
`LSSEM.PAS`	Sematikausführung, mit COCO erzeugt
`LSSTRNG.PAS`	Stringverwaltung mit Streams
`LSSTRNG1.PAS`	Stringverwaltung mit 64K-Arrays
`LSSYM.PAS`	Symboltabellenverwaltung
`LSSYN.PAS`	Parser, von COCO erzeugt
`MAUS.PAS`	Mausroutinen
`GTED.PAS`	Editor
`SYS3D.PAS`	3D-Graphikroutinen
`TURTLE3D.PAS`	3D-Turtlegraphik
`USHOWPAL.PAS`	Palettenanzeige
`UTURTLE2.PAS`	2d-Turtlegraphik
`!CTOOLS.EXE`	Archiv mit den für die Programmoberfläche benötigten Units aus [18]. Die Rechtsbestimmungen des Vogel-Verlags sind bei anderweitiger Nutzung der Units zu beachten!
`FARBEN.PRM`	Farbdatei
`*.BGI`	256-Farben–BGI-Treiber

!LSC-DOC.EXE Dokumentation

`MAKEDVI.BAT`	Batchdatei zur Erzeugung des TEX-Files
`!SCRIPT.EXE`	Script-Style–Familie
`BM2FONT.EXE`	Programm zur Einbindung von Bildern in TEX
`LSC-DOC.DVI`	die Dokumentation (als compiliertes TEX-File)
`LSC-DOC.TEX`	Quelltext
`LSCDOC.TEX`	Include-File
`*.STY`	weitere spezielle Style-Files
`*.GIF`	Bilder

Verzeichnis LSC

`!LSC-TXT.EXE` Textversion von LSC

`COLINST.EXE`	Programm zum Farbeinstellung, generiert FARBEN.PRM
`FARBEN.PRM`	Farbdatei; bei Hercules-Karte bitte löschen!
`LSC-TXT.EXE`	das Programm
`LSC-TXT.HLP`	Hilfetexte
`*.LSC`	Beispieldateien

`!LSC` 3D-Graphikversion von LSC

`COLINST.EXE`	Programm zum Farbeinstellung, generiert FARBEN.PRM
`FARBEN.PRM`	Farbdatei; bei Hercules-Karte bitte löschen!
`*.BGI`	BGI-Treiber
`LSC.EXE`	das Programm
`LSC.HLP`	Hilfetexte
`*.LSC`	Beispieldateien

Verzeichnis ZUSATZ. Hier sind einige zusätzliche Programme zu finden. Für diese wird keinerlei Support geleistet; sie sind reine 'add on'-Zugaben.

`!LSVIEW.EXE` Testversion des LSC-Compilers. Mit diesem Programm kann die Arbeit des Scanners und des mathematischen Parsers verfolgt werden. Achtung: Die hier verwendete Grammatik ist weder vollständig noch fehlerfrei.

`!OLS.EXE` Programm zur graphischen Darstellung zweidimensionaler 0L-Systeme. In gewisser Weise der Vorläufer von LSC.

`OLS.EXE`	Programm
`OLS.HLP`	Hilfetexte
`*.OLS`	Beispiele

B. Programmdokumentation

B.1. Die Programmoberfläche

LSC arbeitet mit einer fensterorientierten Oberfläche und pull-down-Menüs, wie man es von modernen Programmen gewohnt ist. Die Bedienung des Programms erfolgt mittels Cursortasten, Hotkeys und der Maus wie üblich; solange Sie sich in der Menüzeile oder in den Menüs befinden, erhalten Sie mit F1 eine kurze kontextsensitive Hilfe. Abbildung B.1 gibt eine Ansicht des Hauptbildschirms von LSC, man kann die Menüleiste und das Editierfenster sehen.

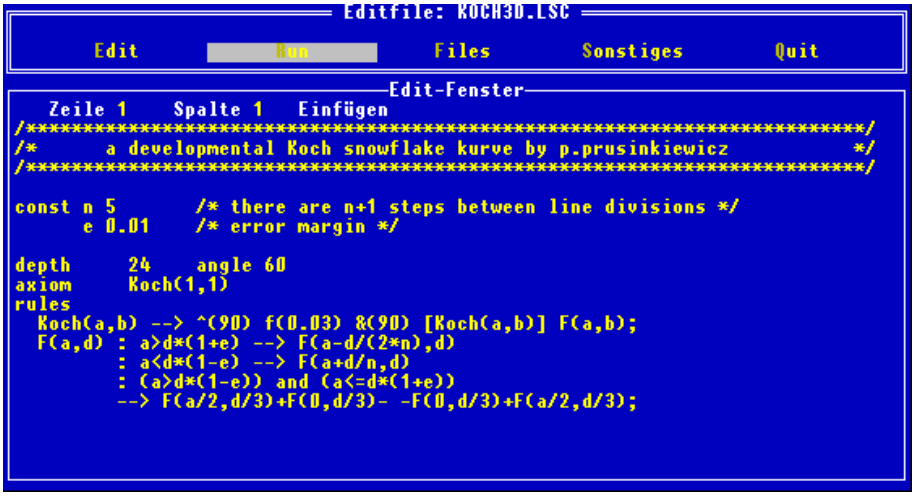


Abbildung B.1.: Die LSC-Oberfläche

Dateioperationen

Bei Aktivierung des Menüpunktes *Files* öffnet sich ein Menü mit Funktionen zum Einlesen, Abspeichern und Löschen von L-Systemen und dem Wechsel des Verzeichnisses.

Einlesen. Nach Bestätigung dieses Menüpunktes öffnet sich ein Fenster, in dem der gewünschte Dateiname eingegeben werden kann. Als Voreinstellung erscheint eine Wildcard-Spezifikation: *.LSC. Wenn Sie diese mit ⏎ einfach annehmen, erscheint ein

Fenster mit allen Dateien, auf die diese Spezifikation paßt. Hier können Sie mit den Cursortasten herumwandern und mit ⏎ eine Datei auswählen. Sie können aber auch die Spezifikation ändern. Geben Sie dabei einen Namen an, zu dem kein passendes File gefunden wird, kreiert LSC eine neue Datei. Ein Ändern auf ein anderes Laufwerk oder Subdirectory ist nicht möglich, dafür wählen Sie bitte den Punkt *Verzeichnis*.

Alternativ zum Einlesen eines L-Systems über das Menü können Sie dieses als Parameter beim Start von LSC übergeben.

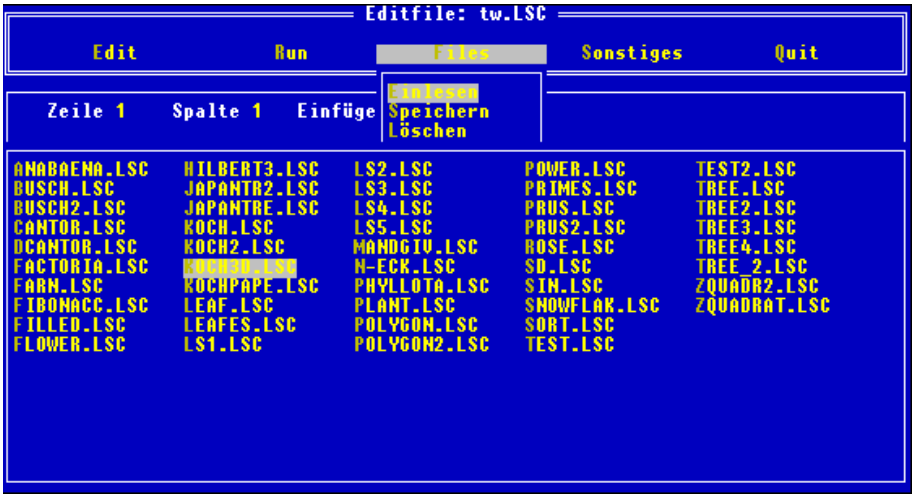


Abbildung B.2.: Die Dateiauswahl

Speichern. Mit diesem Menüpunkt wird die aktuelle Datei gespeichert. Normalerweise werden Sie die bearbeitete Datei unter dem Namen speichern wollen, den Sie beim Aufruf mit *Einlesen* angegeben haben. Dieser erscheint auch als Voreinstellung in dem Dateinamenfenster. Sie können die Datei aber auch unter einem anderen Namen abspeichern, wenn Sie die Voreinstellung entsprechend abändern. Wenn Sie dabei Wildcards verwenden, wird ein Fenster mit den auf die Wildcards zutreffenden Dateien geöffnet. Wenn Sie eine Datei davon auswählen, wird diese mit dem gerade bearbeiteten Text überschrieben. Soll die Datei in einem anderen Verzeichnis als dem aktuellen abgespeichert werden, wählen Sie bitte den Punkt *Verzeichnis* und nach Verzeichniswechsel erneut den Punkt *Speichern*.

Löschen. Hier können Sie einzelne Dateien löschen. Geben Sie die gewünschte Datei im Dateinamenfenster an. Wildcards führen hier nicht zum Löschen von Dateigruppen. Die auf die Wildcards passenden Dateien werden vielmehr in einem Fenster zur Auswahl angeboten, nur die gewählte Datei wird gelöscht. Selbstverständlich erfolgt vor dem Löschen noch eine Sicherheitsabfrage.

Verzeichniswechsel. Hier können Sie das aktuelle Verzeichnis (Subdirectory) und/oder

das angemeldete Laufwerk wechseln. Bei den Dateioperationen wird das momentan angemeldete Verzeichnis oben im Fensterrahmen angegeben.

Editieren

LSC verfügt über einen leistungsfähigen Editor mit Wordstar-kompatibler Tastenbelegung. Wenn Sie bereits mit Turbo-PASCAL oder ähnlichen Programmen gearbeitet haben, werden Ihnen viele Kommandos bekannt vorkommen.

[←], [Ctrl][S]	Bewegt den Cursor ein Zeichen nach links.
[→], [Ctrl][D]	Bewegt den Cursor ein Zeichen nach rechts.
[˄], [Ctrl][E]	Bewegt den Cursor ein Zeichen nach oben.
[˅], [Ctrl][X]	Bewegt den Cursor ein Zeichen nach unten.
[Ctrl][←], [Ctrl][A]	Bewegt den Cursor ein Wort zurück.
[Ctrl][→], [Ctrl][F]	Bewegt den Cursor ein Wort vor.
[PageUp], [Ctrl][R]	Blättert eine Seite zurück.
[PageDown], [Ctrl][C]	Blättert eine Seite vor.
[<——], [Ctrl][H]	Löscht das Zeichen links vom Cursor.
[Del], [Ctrl][G]	Löscht das Zeichen unter dem Cursor.
[Ctrl][T]	Löscht das nächste Wort.
[Ctrl][Y]	Löscht die ganze Zeile.
[<—┘], [Ctrl][M]	Fügt eine neue Zeile ein, der Cursor geht in die neue Zeile (nur im INSERT-Modus).
[Ctrl][N]	Fügt eine neue Zeile ein, der Cursor bleibt aber in der aktuellen Zeile.
[Ctrl][W]	Rollt den Bildschirm eine Zeile nach unten.
[Ctrl][Z]	Rollt den Bildschirm eine Zeile nach oben.
[INS], [Ctrl][V]	Wechselt zwischen INSERT- und OVERWRITE-Modus.
[ESC], [Ctrl][[]	Beendet das Editieren.
[Home], [Ctrl][Q]+[S]	Bewegt den Cursor an den Zeilenanfang.

End, Ctrl Q+D	Bewegt den Cursor an das Zeilenende.
Ctrl+Home, Ctrl Q+E	Bewegt den Cursor an den Seitenanfang.
Ctrl+End, Ctrl Q+X	Bewegt den Cursor an das Seitenende.
Ctrl+PageUp, Ctrl Q+R	Bewegt den Cursor an den Textanfang.
Ctrl+PageDown, Ctrl Q+C	Bewegt den Cursor an das Textende.
Ctrl Q+Y	Löscht bis zum Zeilenende.
Ctrl Q+F	Sucht eine Zeichenfolge im Text. Optionen: u: (upper) Zwischen Groß- und Kleinbuchstaben wird nicht unterschieden. w: (word) Findet nur komplette Wörter. b: (back) Durchsucht die Datei rückwärts.
Ctrl Q+A	Sucht und ersetzt eine Zeichenfolge durch eine andere. Optionen: g: (global) Sucht im gesamten Text. n: (no) Ersetzt eine gefundene Zeichenfolge ohne Rückfrage. u: (upper) Zwischen Groß- und Kleinbuchstaben wird nicht unterschieden. w: (word) Findet nur komplette Wörter. b: (back) Durchsucht die Datei rückwärts.
Ctrl K+R	Liest eine Datei von der Diskette und fügt sie an der aktuellen Cursorposition in den Text ein.
Ctrl K+W	Schreibt einen Block in eine Datei.
Ctrl K+Y	Löscht einen Block.
Ctrl K+C	Kopiert einen Block.
Ctrl K+V	Verschiebt einen Block.
Ctrl U	Abbruch einer Blockoperation.
Ctrl K+O	Fügt aktuelle Optionen des L-Systems in den Text ein.

Die Blockoperationen (mit Ausnahme des Blocklesens) sind abweichend vom Wordstar-Standard implementiert. Wenn Sie beispielsweise das Kommando zum Blockverschieben eingeben, fragt der Editor erst nach dem Blockanfang, dann nach dem Blockende und schließlich nach der neuen Position. Zur Bestimmung der Position können sämtliche Cursorsteuerfunktionen verwendet werden, einschließlich Suchen!

Run

Hiermit wird die Berechnung eines L-Systems gestartet. Zuerst wird die *Ableitung* dieses Systems berechnet. Sie können in einem Fenster die Berechnungsschritte verfolgen. Dann schaltet LSC in den Graphikmodus um und beginnt die Größe der L-System–Graphik zu berechnen. Aus technischen Gründen bleibt der Bildschirm dabei dunkel! Also keine Panik, falls die Berechnung etwas länger dauert.

Anschließend erscheint auf dem Bildschirm ein Koordinatensystem, je nach Graphik mit zwei oder drei sichtbaren Koordinatenachsen, siehe Abbildung B.3. In Bild B.4 sehen Sie als Beipiel ein zweidimensionales L-System (*phyllotaxis*, nach [12, Kapitel 4]). Mit

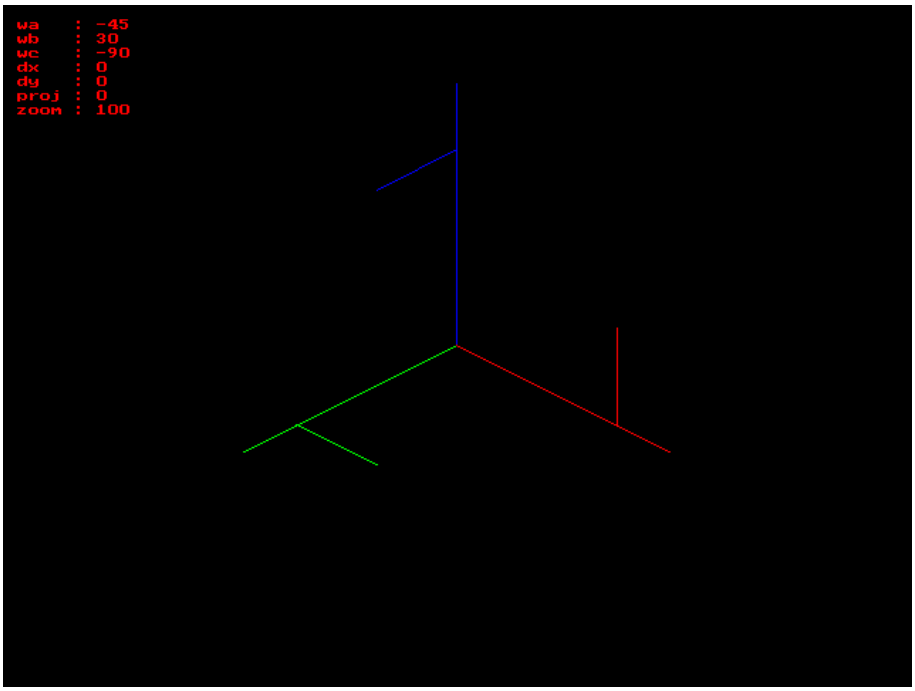


Abbildung B.3.: Das 3D-Koordinatensystem

⏎ wird das L-System gezeichnet. Dabei wird es vom Programm so skaliert, daß es vollständig in den von den Koordinatenachsen angezeigten Raum paßt.

Mit folgenden Tasten können Sie Lage und Größe des Koordinatensystems verändern (es sollte dabei vorrangig der abgesetzte Ziffernblock einer AT-Tastatur bei ausgeschalteter Num-Anzeige genutzt werden):

⏎	Zeichnen des L-Systems
S	Zeichnen des L-Systems, anschließend Warten auf beliebigen Tastendruck. Zum ‘Grabben’ des Bildes.

[+]	Vergrößern
[-]	Verkleinern
[↓], [↑], [←], [→]	Verschieben des Koordinatensystems
[*], [/]	Drehen des Bildes (Winkel γ ändern)
[PageUp], [PageDown]	Drehen des Koordinatensystems um die blaue Achse (Winkel α ändern)
[Home], [End]	Drehen des Koordinatensystems senkrecht zur blauen Achse (Winkel β ändern)
[Z], [H]	Ändern der Projektionsart (siehe weiter unten)

Links oben am Bildschirm werden die aktuellen Werte α, β, γ, dx, dy, $proj$ und $zoom$ angezeigt. Die Winkel zeigen Ihnen die Verdrehung des Koordinatensystems, die Werte dx und dy die Verschiebung des Koordinatensystems auf dem Bildschirm in horizontaler bzw. vertikaler Richtung. 100 bedeutet dabei unabhängig von der Bildauflösung der äußere Rand. *zoom* ist der Vergrößerungsfaktor in Prozent. Der Wert 100 bedeutet, daß das gesamte Koordinatensystem gerade so auf den Bildschirm paßt. *proj* legt die Projektionsart fest: 0 bedeutet Parallelprojektion, Werte zwischen 1000 und 10000 aktivieren eine Punktprojektion. Der Wert legt dann die relative Entfernung des Betrachterstandpunktes zur Graphik fest.

Im Editor können Sie mit [Ctrl][K]+[O] die aktuellen Einstellungen des Koordinatensystems in den Text einfügen.

Sonstiges

Unter diesem Menüpunkt finden Sie Funktionen zum Erzeugen eines Listings, zum Anzeigen der generierten Ableitung in Textform, zur Ausgabe des Bildes als TEX-File, statistische Informationen sowie eine Funktion zum Anzeigen der aktuellen Palette.

Listing. Diese Funktion werden Sie warscheinlich nie brauchen. Sie können sich hier ein Textfile des *Zwischencodes* erzeugen. Informationen zum Zwischencode werden in meiner Diplomarbeit gegeben.

Show. Hier können Sie sich die Ableitung des L-Systems im 'Klartext', d.h. als Textstring, ansehen. Zum Durchrollen des Textes verwenden Sie dieselben Tastaturkommandos wie im Editor. Sie können aber keine Änderungen an der Ausgabe vornehmen.

TEX. Hier erfolgt eine emTEX-Ausgabe des Bildes. Nach Aktivierung dieses Menüpunktes erscheint als Voreinstellung der aktuelle Filenamen im Dateinamenfenster. Sie können das TEX-File auch unter einem anderen Namen abspeichern, wenn Sie die Voreinstellung

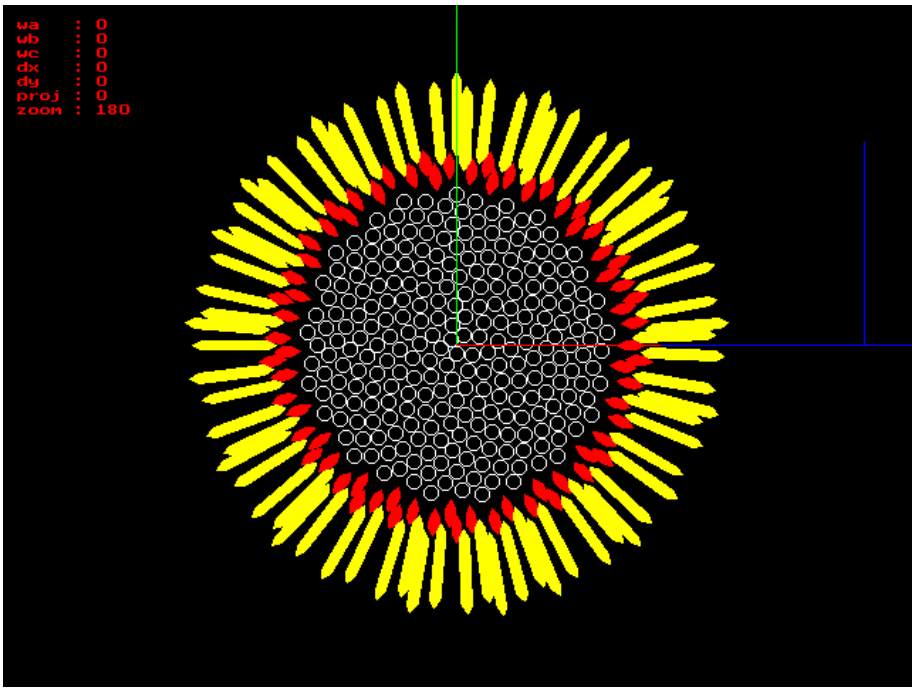


Abbildung B.4.: Sonnenblume, mit LSC erzeugt

entsprechend abändern. Wenn Sie dabei Wildcards verwenden, wird ein Fenster mit den auf die Wildcards zutreffenden Dateien geöffnet. Wenn Sie eine Datei davon auswählen, wird diese mit dem TEX-File überschrieben.

Es wird ein emTEX-include-File erzeugt, das aus einer picture-Umgebung der relativen Breite 100 und der maximalen relativen Höhe 100 sowie speziellen emTEX-Treiber-Befehlen besteht. Dieses File können Sie etwa mit

```
\unitlength0.6mm      % Bild wird 6cm breit eingebunden
\vspace{1ex}          % Abstand zum Text davor
\input <filename> \\
```

in eigene TEX-Files übernehmen. Ein Ausdrucken ist wegen den genutzten speziellen Treiber-Befehlen nur mit emTEX möglich! Beachten Sie auch folgendes:

- Damit emTEX die Files noch verarbeiten kann, dürfen nicht zu viele Linien im File enthalten sein. emTEX verarbeitet auf einer Seite maximal 2730 Linien.
- Die Berechnung der Bilder in TEX kann einige Zeit dauern. Deshalb können Sie mit einer globalen Definition `\def\noLSC{}` die Berechnung verhindern. Es wird dann nur ein Rahmen entsprechend der Bildgröße erzeugt.

- Die Größe des eingebundenen Bildes wird TEX-üblich durch `\unitlength`x`mm` festgelegt. $100 * x$ ist dann die Bildbreite in mm.

Info. Hier können Sie einem Fenster die aktuelle und die maximale Länge der Ableitung entnehmen. Je mehr freien Speicher Sie haben, desto größer kann die Ableitung maximal werden.

Palette. Es wird die aktuelle Farbpalette angezeigt. Mit einer beliebigen Taste beenden Sie diese Anzeige.

B.2. Die Sprache LSC

Die Syntax

Die Syntax eines L-Systems sowie der Umfang der interpretierten Symbole orientiert sich an [11, 12]. In der folgenden Beschreibung wird eine allgemeine Kenntnis der L-Systeme vorausgesetzt.

Prinzipiell besteht ein L-System aus einem Axiom und mehreren Ersetzungsregeln. In *einem* Schritt werden nun alle Symbole des Axioms durch ihre Ersetzung ersetzt: es wird eine Zwischenableitung gebildet. Mit dieser Zwischenableitung anstelle des Axioms wird der Ersetzungsschritt so oft durchgeführt, wie es die *Ableitungstiefe* angibt. Die letzte erhaltene Zwischenableitung heißt *Ableitung*.

Ein L-System hat in LSC folgenden allgemeinen Aufbau:

```
Optionen
Axiom
Regeln
Optionen
```

Axiom. Das Axiom ist die Startsymbolfolge eines L-Systems. Hinter dem Schlüsselwort `axiom` ist eine Folge von Symbolen zu schreiben. Ein *Symbol* besteht aus einem *Namen*[1] und *Parametern*[2], die in Klammern, durch Kommas voneinander getrennt, dem Symbolnamen folgen. Ein Symbol ist also syntaktisch gleich einem Prozeduraufruf in Turbo PASCAL.

Regeln. Nach dem Schlüsselwort `rules` folgen die einzelnen Regeln des L-Systems. Kontextfreie Regeln werden als

[1] Ein Symbolname besteht aus einer beliebigen Anzahl von Buchstaben und Ziffern, beginnend mit einem Buchstaben, oder ist eins der folgenden Zeichen: `[]{}$ /\^&+-|~ .!'%`

[2] Ein Parameter ist ein zulässiger mathematischer Ausdruck aus (bereits definierten) Konstanten, (an dieser Stelle gültigen) Variablen und den Funktionen *sin, rnd, int*.

```
Symbol '-->' Symbolfolge ';'
```

geschrieben. Beim Symbol dürfen nur Variablen als Parameter geschrieben werden, die dann an dieser Stelle deklariert werden. Bei den Symbolen rechts von '`->`' können diese Variablen in den Parameterkonstrukten verwendet werden.

Bei kontextsensitiven L-Systemen wird der linke Kontext links vom Regelsymbol, der rechte Kontext rechts vom Regelsymbol geschrieben. Außerdem kann eine *Bedingung*[3] angegeben werden. Ist diese wahr, so wird das Regelsymbol durch die Symbolfolge ersetzt, andernfalls wird zur nächsten passenden Regel übergegangen. Konnte keine Regel angewendet werden, bleibt das Symbol erhalten.

```
':' linker Kontext '<' Symbol '>' rechter Kontext
                        ':' Bedingung '-->' Symbolfolge ';'
```

Bedingung, linker und rechter Kontext können auch entfallen (dann werden natürlich auch die Trennzeichen nicht geschrieben).

Zu jedem Symbol (incl. Kontext) können mehrere Ersetzungen in einer Regel zusammengefaßt werden:

```
':' linker Kontext '<' Symbol '>' rechter Kontext
                        ':' Bedingung '-->' Symbolfolge
                        ':' Bedingung '-->' Symbolfolge
                              ...
                        ':' Bedingung '-->' Symbolfolge ';'
```

Optionen. Das Schlüsselwort `options` kann, muß aber nicht geschrieben werden. Hier erfolgen die Vereinbarung und Deklaration von Konstanten, die Festlegung der Ableitungstiefe und des *Drehwinkels*, die Deklarierung von Symbolen als 'ignore' sowie die Einstellungen des Koordinatensystems (siehe Seite 75). All diese Optionen können in beliebiger Reihenfolge und auch mehrfach auftreten. Es wird die letzte Deklaration als gültig angesehen.

Konstanten. Eine Deklaration von Konstanten erfolgt durch

```
const Name '=' Parameter
      Name '=' Parameter
      ...
      Name '=' Parameter
```

[3] Eine Bedingung ist ein Boolescher Ausdruck, wie er z.B. bei der `if`-Konstruktion in Turbo PASCAL auftritt.

Das Gleichheitszeichen kann entfallen.

Ableitungstiefe und Drehwinkel. Diese werden durch

```
depth Zahl
angle Zahl
```

deklariert. Die Ableitungstiefe ist eine natürliche Zahl, der Winkel wird in Grad angegeben.

ignore. Nach dem Schlüsselwort `ignore` sind all die Symbole (mit beliebigen symbolischen Bezeichnern als Parameter) anzugeben, die bei der *Kontextsuche* ignoriert werden sollen.

Koordinateneinstellung. Die Ausrichtung des Koordinatensystems erfolgt mit

```
camera   zoom ',' alpha ',' beta ',' gamma ',' dx ',' dy
```

Die Bedeutung von *zoom*, α, β, γ, *dx* und *dy* ist auf Seite 76 beschrieben.

Hinweise

- Es wird keine feste Textstruktur erwartet. Wie in Turbo PASCAL können in den Text beliebig viele Kommentare, Leerzeichen und Zeilenvorschübe eingefügt werden.
- Jedes Symbol muß das ganze L-System hindurch eine konstante Anzahl von Parametern haben; das erste Auftreten eines Symbols wird als dessen Deklaration gewertet.
- Tritt ein Symbol erstmals im `ignore`-Teil auf, müssen daher auch hier symbolische Parameterbezeichner benutzt werden; diese können frei gewählt werden und haben keinerlei Einfluß auf die folgenden Deklarationen.
- Variablen (Parameterbezeichner) sind nur lokal in der betreffenden Regel gültig.
- Wird in einer Regel eine Variablenname verwendet, der vorher bereits als Konstante deklariert wurde, kann die Konstante in dieser Regel nicht mehr genutzt werden (Prinzip der Lokalität von Bezeichnern, analog Turbo PASCAL).
- Sind mehrere Regeln auf ein Symbol anwendbar, wird von diesen die im Quelltext an erster Stelle stehende angewandt.
- Kommentare sind in `/*` und `*/` einzuschließen und dürfen nicht verschachtelt werden.

- Das Semikolon am Ende einer Regel darf auf keinen Fall fehlen! Sonst kommt es zu Fehlinterpretationen der Regeln.
- Zwischen ein mit einem Buchstaben endendes Symbol ohne Parameter und ein mit einem Buchstaben beginnendes Symbol ist ein Leerzeichen einzufügen. Sonst werden beide Symbole als eins betrachtet! Z.B. ist `F F+F` anstelle von `FF+F` zu schreiben.

Als Beispiel wird das L-System zu Abbildung B.4 angebeben.

```
/************************************************************/
/* phyllotaxis                                      [Pru90] */
/************************************************************/

const a 137.5 /* divergence angle */

axiom
      A(0)
rules
      A(n) : n < 260 --> turn(a)[f(n^0.5)circle(0.8)]A(n+1)
           : n < 330 --> turn(a)[f(n^0.5)
                             '(6){-F+F+F-|-F+F+F-}]A(n+1)
                     --> turn(a)[f(n^0.5)
                             '(14){-F+f(7)+F-|-F+f(7)+F-}]A(n+1);
options
        depth 400
        angle 30
        camera 150, 0, 0, 0, 0, 0
```

Die Turtlebefehle

LSC kann folgende Symbole verarbeiten:

F	Bewegung in Zeichenrichtung und Zeichnen einer Linie.
f	Bewegung in Zeichenrichtung ohne zu zeichnen.
turn	Drehung nach links.
$+$	Drehung nach links.
$-$	Drehung nach rechts.
&	Senkung der Turtle.

∧	Anhebung der Turtle.
\	Rollen nach links.
/	Rollen nach rechts.
|	Drehung in entgegengesetzte Richtung.
$	Drehung der Turtle in die Senkrechte.
[	Beginn einer Verzweigung.
]	Ende einer Verzweigung.
{	Beginn eines Polygons.
G	Bewegung in Zeichenrichtung und Zeichnen einer Linie. Dabei wird kein Punkt zum Polygon hinzugefügt.
.	Nimmt aktuellen Punkt zum Polygon hinzu.
}	Ende eines Polygons. Dieses wird mit der aktuellen Zeichenfarbe, die *vor* { galt, gefüllt.
!	Vermindern der Linienstärke.
’	Erhöhen der Zeichenfarbe.
color	Erhöhen der Zeichenfarbe.
circle	Zeichen eine Kreises.

Falls ein Symbol Parameter besitzt, wird der erste Parameter genutzt, ansonsten gelten programminterne Werte: Bei Drehungen kann ein Winkel angegeben werden, fehlt dieser, wird der *Drehwinkel* `alpha` genommen. Bei color und ’ wird auf die als Parameter übergebene Farbe umgeschalten, sonst die in der Palette folgende Farbe genommen. Ebenso wird bei ‘!’ mit einem Parameter dieser als neue Linienstärke angesehen, sonst die aktuelle Linienstärke um eins vermindert. Bei Bewegungen der Turtle gibt ein Parameter die Streckenlänge an, fehlt dieser, bewegt sich die Turtle um eine *Einheitslänge* vorwärts.[4]

Die Symbole turn und color wurden eingeführt, um einen Startwinkel und Anfangszeichenfarben festzulegen, wenn im L-System die Symbole + und ’ parameterfrei verwendet werden (sonst müsste bei diesen im gesamten L-System stets ein Parameter mitgeführt werden, dies würde zu Lasten der Effizienz von LSC und zu Lasten der Klarheit des L-Systems gehen).

Weitere Hinweise und viele Beipiele zur Benutzung obiger Turtlebefehle sind in den Werken von Prusinkiewicz zu finden.

[4] Diese Einheitslänge wird programmintern so berechnet, daß das zu zeichnende L-System vollständig im Koordinatensystem darstellbar ist.

B.3. Entwicklung

Version 1.0 September 1991 im Rahmen eines Seminarvortrages entstanden. Es konnten nur 0L-Systeme mit maximal drei Regeln berechnet werden. Eine automatische Skalierung auf Bildschirmgröße fehlte.

Version 2.x Februar bis August 1992. Immer noch 0L-Systeme, aber bereits Farbgebung, ca. 60 mögliche Regeln, Zeichnen von Polygonen, automatische Skalierung. Außerdem völlig neue graphische Oberfläche.

Version 3.0 März 1993. Völlig neu geschrieben, Einsatz eines internen Compilers zum Bearbeiten von IL-Systemen. Neuer Algorithmus zur Ableitungsbildung; die maximale Ableitungslänge beträgt 64 KByte. Text-Version.

Version 3.01 Juli 1993. Die Oberfläche ist fertig, ebenso eine 3D-Turtlegraphik. Jetzt auch im protected mode compilierbar.

Version 3.02 Juli 1993. Neue String-Unit. Jetzt sind Ableitungen beliebiger Länge, nur durch den freien Speicher beschränkt, möglich.

Literaturverzeichnis

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Compilerbau (in zwei Bänden)*. Addison-Wesley Verlag, Bonn, 1988

[2] Roman Gerike. *Programmierte Gewächse. Simulation von Pflanzenwachstum in C++*. c't 4/92

[3] Klöppel, Paul, Rauch, Ruhland. *Compilerbau. Am Beispiel der Programmiersprache SIMPL*. Vogel-Verlag, Würzburg, 1991

[4] Frank Kriese. *L-Systeme. Theorie*. Diplomarbeit, Universität Greifswald, Fachbereich Mathematik und Informatik, Greifswald, 1993

[5] Thomas Kuschel. *Erzeugung von Fraktalen durch Igelgeometrie*. Diplomarbeit, Universität Greifswald, Fachbereich Mathematik und Informatik, Greifswald, 1989

[6] Benoît B. Mandelbrot. *Die fraktale Geometrie der Natur*. Birkhäuser Verlag, Basel, 1991

[7] Eberhard Mattes. *dvidrv 1.4d*. Manual, Teil des emTEX-Paketes, 1990

[8] H. Mössenböck. Alex — A Simple and Efficient Scanner Generator. SIGPLAN Notices, Vol. 21 #5, Mai 1986

[9] H.-O. Peitgen D. Saupe. *The Science of Fractal Images*. Springer-Verlag, New York, 1988

[10] H.-O. Peitgen, H. Jürgens, D. Saupe. *Fractals for the Classroom, Part Two*. Springer-Verlag, New York, 1992

[11] Przemyslaw Prusinkiewicz, James Hanan. *Lindenmayer Systems, Fractals, and Plants*. Springer-Verlag, New York, 1989

[12] Przemyslaw Prusinkiewicz, Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990

[13] Przemyslaw Prusinkiewicz, James Hanan. *L-systems: from formalism to programming languages*. Artikel aus [15, Seite 193 ff.]

[14] P. Rechenberg, H. Mössenböck. *Ein Compiler-Generator für Mikrocomputer*. Carl Hanser Verlag, München Wien, 1988

[15] G. Rozenberg, A. Salomaa (Eds.). *Lindenmayer Systems. Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*. Springer-Verlag, New York, 1992

[16] Michael Schmidt. *Generierung effizienter Übersetzer*. Dissertation, Universität Paderborn, Dissertationsdruck Darmstadt, 1990 (?)

[17] Reinhard Scholl, Olaf Pfeiffer. *Natur als fraktale Grafik: stochastische Fraktale und L-Systeme programmiert mit Turbo-Pascal*. Markt&Technik Verlag, Haar bei München, 1991

[18] Till S. Schwalm. *Toolbox für Turbo-PASCAL, 3. Auflage*. Vogel Verlag, Würzburg, 1993

[19] Michael Tischer. *Turbo-PASCAL intern, 3. Auflage*. Data Becker Verlag, Düsseldorf, 1991

Selbständigkeitserkärung

Hiermit erkläre ich, die vorliegende Arbeit eigenständig und nur die angegebenen Quellen benutzend angefertigt zu haben.

Greifswald, den 22. Juli 1993 Volker Pohlers