

Das Painter-Manual

<http://fdit-www.cs.uni-dortmund.de/~peter/Haskellprogs/Painter.tgz>

Peter Padawitz

TU Dortmund

27. Mai 2017

Die Haskell-Datei `Painter.hs` dient der zweidimensionalen – auch animierten – Wiedergabe von Texten, Bäumen und anderen zweidimensionalen Figuren. Die Funktionen von `Painter.hs` erzeugen aus dem Inhalt einer als Parameter übergebenen Datei `svg`-Code. Die graphische Wiedergabe erfolgt automatisch beim Aufruf der `svg`-Datei durch einen Browser. Passt die darzustellende Figur nicht in dessen Fenster, dann wird das Fenster mit Scrollbars versehen, so dass sie vollständig sichtbar gemacht werden kann. Während der Bearbeitung einer einzelnen Figur sollte die zugehörige `svg`-Datei geöffnet bleiben, damit Reloads genügen, um zu sehen, wie neue Abstands- oder Hintergrundparameter (s.u.) die Figur verändern. Jede Bilddatei `file` kann mit

```
<iframe id="pic" src="file" width="600" height="600" frameborder="0"></iframe>
```

in Originalgröße – ggf. mit Scrollbars versehenen – Fenster oder mit

```

```

auf 600×600 Pixel skaliert in eine html-Datei eingebunden werden. Die Painter-Funktionen `drawTerms{C}`, `drawCS` und `movie` (s.u.) erzeugen oder verarbeiten eine Serie von Bildern und erstellen eine Webseite, auf der die Bilder einzeln oder als Diashow betrachtet werden können.

`drawText file` schreibt den Text in der Datei `file` in Spalten von jeweils 80 Zeichen und 37 Zeilen in ein Fenster.

`draw...I` startet eine Schleife, in der – durch Leerzeichen getrennt – ein horizontaler und ein vertikaler Skalierungsfaktor einzugeben sind. Ein dritter (optionaler) Parameter ist der Name einer `svg`-, `jpg`-, `png`-, `gif`- oder `pdf`-Datei im Verzeichnis `Pix`, die als Hintergrundbild interpretiert wird. Nach Drücken der return-Taste wird der jeweilige Befehl wie unten beschrieben ausgeführt. Die Schleife terminiert, wenn anstelle einer Parametereingabe die return-Taste gedrückt wird.

`drawTerm{C/Nodes}` führt `drawTerm{C}I` nur einmal mit horizontalem und vertikalem Skalierungsfaktor 10 bzw. 40 aus.

`draw{L}Graph` und `drawC{S}` führen `draw{L}GraphI` bzw. `drawC{S}I` nur einmal mit horizontalem und vertikalem Skalierungsfaktor 1 aus.

1 Typen und Farben

Zur internen Darstellung zweidimensionaler Figuren werden folgende elementare Typen verwendet:

<code>Point = (Float,Float)</code>	<code>LPoint = (String,Float,Float) (labelled point)</code>
<code>CPoint = (RGB,Float,Float) (colored point)</code>	
<code>Path = [Point]</code>	<code>LPath = [LPoint] (labelled path)</code>
<code>CPath = (RGB,Int,Path) (colored path)</code>	<code>CLPath = (RGB,LPath) (labelled and colored path)</code>

`RGB r g b` bezeichnet die Farbe mit (ganzzahligem) Rot-, Grün- und Blauwert `r`, `g` bzw. `b`. Stimmen mindestens zwei Werte mit 0 oder 255 überein, dann handelt es sich um eine von $6 * 255 = 1530$ Hue-Farben. Andere Farbwerte liefern hellere oder dunklere Versionen einer Hue-Farbe. Folgende Farbkonstanten sind vordefiniert: `red` (`r=255,g=0,b=0`), `magenta` (`r=b=255,g=0`), `blue` (`r=g=0,b=255`), `cyan` (`r=0,g=b=255`), `green` (`r=b=0,g=255`), `yellow` (`r=g=255,b=0`), `black` (`r=g=b=0`) und `white` (`r=g=b=255`). Die möglichen Werte der ganzzahligen `mode`-Komponente von `CPath` sind in Kapitel 3 beschrieben.

2 Terme zeichnen und transformieren

Für nichtleere Listen s zeichnet `drawList file draw s` für alle $1 \leq i \leq \text{length}(s)$ das i -te Element von s mit horizontalem und vertikalem Skalierungsfaktor 10 bzw. 40 unter Verwendung der Zeichenfunktion `draw(i)` in die Datei *Pix/file/i.svg*. `drawList file` erstellt außerdem die Webseite *Pix/file.html*, auf der durch die Elemente von s navigiert werden kann. *Pix* muss die JavaScript-Datei *Painter.js* enthalten, die den Steuerelementen von *Pix/file.html* Kommandos zuordnet.

`drawTerm{s}{C/Nodes}{I} file` erwartet in der Datei *file* einen (als) Baum t (darstellbaren Ausdruck) bzw. eine Liste ts davon, übersetzt t bzw. ts in svg-Code und schreibt diesen in die Datei *Pix/file.svg* bzw. die Dateien *Pix/file/i.svg*, $1 \leq i \leq \text{length}(ts)$. Öffnet man diese mit einem Browser, dann erscheint der jeweilige Baum als zweidimensionale Figur. Einzelheiten zur Syntax der Bäume entnehme man dem Programm *Painter.hs*. U.a. sind mehrzeilige Knoteneinträge möglich, wobei `%` als Zeilenumbruch interpretiert wird. `drawTerms{C}` erstellt außerdem (mit Hilfe von `drawList`; s.o.) die Webseite *Pix/file.html*, auf der durch die Elemente von ts navigiert werden kann.

`drawTermNodes file` erwartet neben einem Baum t eine Liste ps von Knoten in *file*, die bei der Wiedergabe von t in *Pix/file.svg* grün gezeichnet werden. t und ps müssen in *file* durch ein Komma getrennt werden.

`mkTree{C} file str` schreibt *str* in die Datei *file* und führt dann `drawTerm{C} file` aus.

`drawTerm{s}C` färbt die Knoten und Kanten eines Baumes entsprechend ihrem jeweiligen Abstand von der Wurzel des Baumes.

Der **Termparser** übersetzt den Inhalt der Eingabedatei in einen Baum der Instanz `Tree String` des polymorphen Typs

```
data Tree a = F a [Tree a] | V a
```

Ausdrücke `F"str" [e1, ..., en]` des Typs `Tree String` werden vom Parser erhalten, d.h. in einen Baum übersetzt, dessen Wurzel mit `str` markiert ist und dessen maximale Unterbäume diejenigen n Bäume sind, in die der Parser die Strings e_1, \dots, e_n übersetzt hat.

Als Variable erkannt wird jeder aus Buchstaben und Ziffern bestehende String, der mit `x`, `y`, `z`, `F`, `G` oder `H` beginnt. Jeder als Variable erkannte String `str` wird in das Blatt `V"str"` übersetzt.

Die Symbole `;; ; >> <> /\ \/ == = /= <= < >= > $: + - * / ^ ~ .` werden als Infix-Operatoren mit zum Teil beliebiger endlicher Stelligkeit erkannt (mit von links nach rechts zunehmender Priorität).

`reduce file [f1, ..., fn]` übersetzt den Inhalt der Datei *file* in eine Liste von Gleichungssystemen und verwendet diese als Reduktionsregeln. Es werden mit folgender Grammatik erzeugbare Strings verarbeitet:

$$\begin{array}{lcl} S & \rightarrow & \text{term} ; ; \text{eqss} \\ \text{eqss} & \rightarrow & \epsilon \mid \text{eqs} ; ; \text{eqss} \\ \text{eqs} & \rightarrow & \text{eq} \mid \text{eq} ; \text{eqs} \\ \text{eq} & \rightarrow & \text{term} = \text{term} \end{array}$$

Terme sind wie in Haskell aufgebaut, einschließlich über *field labels* (Attribute) definierter Elemente destruktiver Datentypen (“Records”; siehe §6, Beispiel `1isa2`), die (kanten-) markierten Bäumen entsprechen.

Laut obiger Grammatik muss der Inhalt von *file* ein einzelner Term sein oder die Form `t;eqs1;...;eqsk` haben. *t* ist der zu reduzierende Term. $\text{eqs}_1, \dots, \text{eqs}_k$ sind Listen von Gleichungen. Zunächst wird *t* mit Hilfe von `eval : Tree String → Maybe(Tree String)` partiell ausgewertet. *eval* interpretiert aussagenlogische und arithmetische Funktionen, Konditionale (*ite*), *sections* wie `(+5)`, das Symbol `suc` als Nachfolgerfunktion `(+1)`, das Infixsymbol `:` als den gleichnamigen Listenkonstruktor von Haskell, Attribute – wie in objektorientierten

Sprachen üblich – als Zugriffsfunktionen, λ -Applikationen (wie in Haskell, jedoch ohne backslash) sowie Haskell-Ausdrücke der Form $\text{case } t \text{ of } p_1 \rightarrow t_1 | \dots | p_n \rightarrow t_n$, die hier als Applikationen der Form $(p_1 \rightarrow t_1 | \dots | p_n \rightarrow t_n)(t)$ dargestellt sein müssen.

Nach der partiellen Auswertung von t mit eval werden die Gleichungen von eqs_1 in Präordnung (“leftmost-outermost”) auf t und die Unteräume von t sooft wie möglich angewendet, dann die Gleichungen von eqs_2 , dann die Gleichungen von eqs_3 , usw. Ist gar keine Gleichung anwendbar, dann wird die erste anwendbare Transformationsfunktion der Folge f_1, \dots, f_n – z.B. *distribute*, *flatten*, *impl* – angewendet (siehe Abschnitt REDUCER in Painter.hs).

Die Zwischenergebnisse werden (mit Hilfe von *drawList*; s.o.) im Verzeichnis *Pix/fileReduction* gespeichert, das über die Webseite *Pix/fileReduction.html* zugreifbar ist. Dabei wird jeder Baum zweimal gezeichnet, einmal mit dem (grün gefärbten) Redukt der Regelanwendung, die ihn erzeugt hat, dann mit dem (rot gefärbten) Redex der darauffolgenden Regelanwendung.

XCTL-Formeln beschreiben ein- bzw. zweistellige Relationen zwischen Baumknoten (siehe Kapitel 29 von Fix-points, Categories, and (Co)Algebraic Modeling) und Abschnitt TREE LOGICS in Painter.hs).

Sei form eine XCTL-Formel des Typs nodeSet , deren Semantik $\text{set}(t)$ jedem Baum t des Typs $\text{Tree}(\text{String})$ (s.u.) eine Menge von Knoten von t zuordnet.

drawNodeSet file form erwartet in der Datei *file* einen Baum t , zeichnet t mit horizontalem und vertikalem Skalierungsfaktor 10 bzw. 40 in die Datei *file_set.svg*. Dabei werden die Knoten von $\text{set}(t)$ grün gefärbt.

Sei form eine XCTL-Formel des Typs nodeRel , deren Semantik $\text{rel}(t)$ jedem Baum t des Typs $\text{Tree}(\text{String})$ und jedem Knoten w von t eine Menge von Knoten von t zuordnet. Sei $\text{ws} = [w \in \text{nodes}(t) \mid \text{rel}(t)(w) \neq \emptyset]$.

drawNodeRel file form erwartet in der Datei *file* einen Baum t , erstellt (mit Hilfe von *drawList*; s.o.) die Webseite *Pix/file.html* und zeichnet t mit horizontalem und vertikalem Skalierungsfaktor 10 bzw. 40 für alle $1 \leq i \leq \text{length}(\text{ws})$ in die Datei *Pix/file_rel/i.svg*. Dabei werden der Knoten $\text{ws}!!(i - 1)$ rot und die Knoten von $\text{rel}(t)(\text{ws}!!(i - 1))$ grün gefärbt.

3 Figuren zeichnen

drawGraph file bzw. **drawLGraph file mode** übersetzt den Inhalt der Datei *file* in svg-Code für eine Figur des Typs CPath bzw. CLPath und schreibt den Code in die Datei *Pix/file.svg*. Die Figur besteht also aus Wegen des Typs CPath bzw. CLPath . Jeder Weg ist mit einem ganzzahligen *mode* attribuiert. *mode* gibt an, wie er gezeichnet wird (Färbung, Glättung, Markierungen, etc.):

Sei $(\text{color}, \text{mode}, \text{path}) \in \text{CPath}$ ein von **drawGraph file** zu zeichnender Weg. Ist $\text{mode} = 0$, dann werden der erste und zweite Punkt von *path* als Zentrum bzw. Radienpaar (a, b) einer mit *color* gefärbten Ellipse mit horizontalem Radius *a* und vertikalem Radius *b* interpretiert. Andernfalls werden die ersten fünf Ziffern z_1, z_2, z_3, z_4, z_5 von *mode* wie folgt interpretiert. (Fehlende Ziffern werden durch Einsen ersetzt.)

- $z_1 = 1$: Es werden keine Punkte von *path* gezeichnet.
- $z_1 = 2$: Jeder Punkt von *path* wird als unterschiedlich gefärbter Kreis gezeichnet.
- $z_1 = 3$: Jeder Punkt von *path* wird als unterschiedlich gefärbtes Quadrat gezeichnet.
- $z_1 = 4$: Jeder Punkt von *path* wird als unterschiedlich gefärbte Ellipse gezeichnet und mit seiner Position in der Wegliste, aus der er stammt, markiert.
- $z_2 = 1$: Es werden keine Linien von *path* gezeichnet.
- $z_2 = 2$: Jede Linie von *path* wird unterschiedlich gefärbt.
- $z_2 = 3$: Jede Linie von *path* wird zusammen mit Verbindungen ihrer Endpunkte zum Zentrum des von *path* aufgespannten Polygons zu einem Dreieck erweitert, das unterschiedlich gefärbt wird.

- $z_2 = 4$: $path$ wird als mit $color$ gefärbter Kantenzug gezeichnet.
- $z_2 = 5$: $path$ wird zu einem Polygon erweitert, das mit $color$ gefärbt wird.
- $z_2 \leq 3 \wedge z_3 = 1 \wedge z_4 \in \{1, 2, 3\}$: Die Komponenten von $path$ (Punkte oder Linien bzw. Dreiecke) werden gemäß ihrer jeweiligen Position in einer von z_5 (s.u.) bestimmten Permutation $perm$ von $path$ und eines gleichlangen Farbkreises unterschiedlich gefärbt: Die Farbe der i -ten Komponente von $path$ hat den Wert $hue(z_4)(|path|)(color)(perm(i))$.
 - Je zwei Nachbarn im Farbkreis $hue(1)$ sind bzgl. ihrer Hue-Werte gleich weit voneinander entfernt (Regenbogeneffekt).
 - Je zwei Nachbarn in den Farbkreisen $hue(2)$ und $hue(3)$ sind (fast) komplementär zueinander.
 - $z_5 = 1$: $perm = path$, d.h. der Abstand zum Anfang von $path$ bestimmen die Farbe einer Komponente von $path$.
 - $z_5 = 2$: Der Abstand zum Anfang bzw. Ende von $path$ bestimmen die Farbe einer Komponente von $path$, je nachdem, ob sie dem Anfang bzw. Ende näherliegt.
 - $z_5 = 3$: Die Winkelkoordinaten bzgl. des jeweiligen Vorgängers bestimmen die Farbe einer Komponente von $path$.
 - $z_5 = 4$: Die Steigung der Strecke zum jeweiligen Vorgänger bestimmen die Farbe einer Komponente von $path$.
- $z_2 > 2 \wedge z_3 = 2$: Linienzüge bzw. Polygone werden geglättet.
- $z_4 = 4$: Die Komponenten von $path$ werden mit $color$ gefärbt.

Sei $(color, lpath) \in CLPath$ ein von `drawLGraph file mode` zu zeichnender Weg. Jeder Punkt (lab, x, y) von $lpath$ wird als unterschiedlich gefärbte Ellipse gezeichnet und mit lab markiert. Jede Linie von $path$ wird so gefärbt wie ihr jeweiliger Startpunkt. $mode$ gibt an, wie die Punkte und Linien von $lpath$ gezeichnet werden. Die (vier) möglichen Werte von $mode$ entsprechen denen von z_4 (s.o.).

4 Kurven komponieren und zeichnen

Figuren vom Typ `[CPath]` können als Objekte des Typs

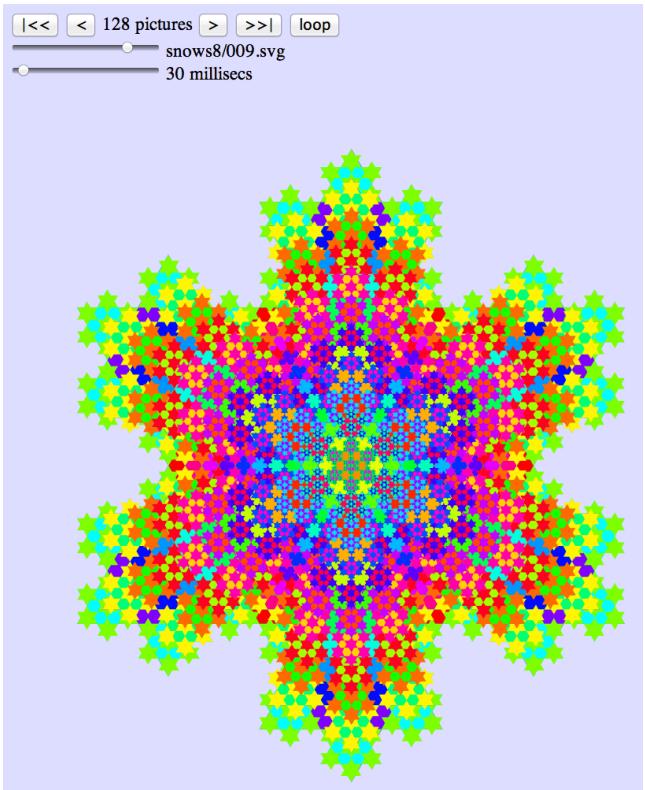
```
data Curve = C {file :: String, colors :: [RGB], modes :: [Int], paths :: [Path]}
```

dargestellt und in dieser Repräsentation verändert oder mit anderen Kurven kombiniert werden. Die Attribute einer Kurve c haben folgende Bedeutung: $file(c)$ ist die Datei, in der das Tripel $(paths(c), colors(c), modes(c))$ abgelegt wird. $paths(c)$ ist eine Zerlegung von c in zusammenhängende Wege. $colors(c)$ und $modes(c)$ ordnen jedem Pfad von c eine Farbe bzw. einen fünfstelligen Zahlencode $z_1 \dots z_5$ zu. Letzterer bestimmt die Art, wie er gezeichnet wird (s.o.).

`drawC c` trägt die Kurve c in die Datei $file(c)$ ein und schreibt svg-Code für c in die Datei $Pix/file(c).svg$.

`drawCS dir cs` schreibt für alle $1 \leq i \leq length(cs)$ svg-Code für $cs!!i$ in die Datei $Pix/dir/i.svg$. Außerdem erstellt `drawCS dir cs` die Webseite $Pix/dir.html$, auf der die Elemente von cs einzeln oder als Diashow betrachtet werden können. $Pix/dir.html$ greift auf die JavaScript-Datei `Painter.js` zu, die deshalb im Verzeichnis Pix enthalten sein sollte.

`movie dir frame` und `movies file frame dirs ps` erzeugen die Webseite $Pix/dir.html$ bzw. $Pix/file.html$, welche die Bilddateien des Verzeichnisses dir bzw. der Verzeichnisse von $dirs$ anzeigt. ps ist eine Liste von Paaren (i, j) von Indizes von $dirs$ bzw. $dirs!!i$, welche die Reihenfolge festgelegt, in der Bilddateien in Originalgröße ($frame=True$) bzw. auf 600×600 Pixel skaliert ($frame=False$) wiedergegeben werden.



Z.B. erzeugt der Aufruf

```
movies "snows10" False (map f [2..9]) ps
where f i = "snows"++show i
      ps = [(i,j) | i <- [0..7],
                  j <- [0..8]++[7,6..1]]
```

die links wiedergegebene Webseite

[Pix/snows10.html](#).

Sie zeigt die Bilddateien der Unterverzeichnisse `snows2, ..., snows9` von `Pix` in der durch `ps` festgelegten Reihenfolge an.

Mit dem oberen Slider wird die Datei ausgewählt, deren Inhalt angezeigt werden soll.

Mit dem unteren Slider stellt man die Bildfrequenz der drei Diashow-Modi `|<<, >>|` und `loop` ein. Diese werden durch Drücken des jeweiligen Knopfes aktiviert bzw. deaktiviert. `|<<` zeigt die Bilddateien von der aktuellen bis zur ersten, `>>|` von der aktuellen bis zur letzten und `loop` ebenfalls von der aktuellen bis zur letzten, beginnt dann aber wieder von vorn.

Die folgenden Funktionen erzeugen, verändern oder kombinieren Kurven vom Typ `Curve`:

`empty file` erzeugt eine leere Kurve in der Datei `file`.

`single file col mode ps` erzeugt aus dem Kantenzug `ps` in der Datei `file` eine Kurve, die bei Aufrufen von `drawC{S}` in der Farbe `col` und der durch den fünfstelligen Zahlencode `mode` bestimmten Weise gezeichnet wird (s.o.).

`updCol col c` und `updMod mode c` setzen alle Elemente von `colors(c)` bzw. `modes(c)` auf `col` bzw. `mode`.

`hueCol mode col c` passt die Farben der Kurve `c` an den Farbkreis `hue(mode)(|paths(c)|)` an: Für alle $1 \leq i \leq |paths(c)|$ wird `colors(c)!!i` auf `hue(mode)(|paths(c)|)(col)(i)` gesetzt. Die drei möglichen Werte von `mode` sind in Abschnitt 3 beschrieben.

`shift (a,b) c` verschiebt alle Punkte von `c` um `a` und `b` Pixel nach rechts bzw. unten.

`scale/hscale/vscale a c` multipliziert alle Punkte von `c` horizontal und/oder vertikal mit `a`.

`turn a c` dreht für um das Zentrum des von `concat(paths(c))` aufgespannten Polygons.

`turn0 a p c` dreht `c` `a` Grad um den Punkt `p`.

`shiftCol n c` verschiebt alle Farben von `colors(c)` bzgl. eines Farbkreises mit 1530 Farben um `n` Elemente.

`takeC n c` und `dropC n c` wenden `take(n)` bzw. `drop(n)` auf alle Pfade von `c` an.

`flipH c` und `flipV c` spiegeln `c` dann an der Horizontalen bzw. Vertikalen durch den Nullpunkt.

`transpose c` spiegelt alle Wege von `c` an der Diagonalen des Koordinatenkreuzes.

`toCenter c` verschiebt die Kurve `c` so, dass ihr Zentrum über dem Nullpunkt liegt.

`combine cs` vereinigt alle Kurven der Liste `cs` zu einer einzigen Kurve.

`overlay cs` vereinigt alle mit `toCenter` zum Nullpunkt verschobenen Kurven von `cs` zu einer einzigen Kurve.

inCenter f c verschiebt die Kurve c mit **toCenter** zum Nullpunkt, wendet dann die Transformation f auf c an und schiebt die transformierte Kurve zurück zu ihrem ursprünglichen Zentrum.

morphing mode n cs fügt zwischen je zwei aufeinanderfolgende Kurven c und c' der Liste cs n von einem Morphing-Algorithmus erzeugte äquidistante Kurven ein. Sie werden abhängig vom gewählten Farbkreis ($mode \in \{1, 2, 3\}$) unterschiedlich gefärbt. Im Fall $mode \notin \{1, 2, 3\}$ findet keine Farbverschiebung statt.

rainbow mode n c erzeugt n Kopien abnehmender Größe der Kurve c . Sie werden wie bei **morphing** abhängig vom gewählten Farbkreis ($mode \in \{1, 2, 3\}$) unterschiedlich gefärbt. Im Fall $mode \notin \{1, 2, 3\}$ findet keine Farbverschiebung statt. $rainbow(mode)(n)(c)$ entspricht $morphing(mode)(n)[c, scale(0)(c)]$.

shine mode n c erzeugt n Kopien abnehmender Größe der Kurve c . Die Kopien werden schrittweise aufgehellt ($mode = 1$) bzw. abgedunkelt ($mode = -1$).

In den folgenden Funktionsaufrufen stehen die Parameter col für die Startfarbe und $mode$ für den in Abschnitt 3 beschriebenen fünfstelligen Zahlencode, die bestimmen, wie die jeweilige Kurve gezeichnet wird.

rect col mode (b,h) erzeugt ein Rechteck der Breite b und der Höhe h .

arc col mode n rs k erzeugt ein Polygon mit $lg = n * |rs|$ Ecken. Für alle $1 \leq i \leq |rs|$ liegt die $(n * i)$ -te Ecke auf einem Kreis mit Radius $rs!!i$ um das Zentrum des Polygons. Es werden nur k benachbarte Kanten des Polygons gezeichnet. Die restlichen $lg - k$ Kanten werden durch eine einzige Kante ersetzt.

poly col mode n rs erzeugt ein Polygon mit $n * |rs|$ Ecken. Für alle $1 \leq i \leq |rs|$ liegt die $(i * n)$ -te Ecke auf einem Kreis mit Radius $rs!!i$ um das Zentrum des Polygons.

tria col mode r erzeugt ein gleichseitiges Dreieck, dessen Ecken auf einem Kreis mit Radius r liegen.

circ col mode r erzeugt einen Kreis mit Radius r .

elli col mode a b erzeugt eine Ellipse mit horizontalem Radius a und vertikalem Radius b .

piles col mode s erzeugt aus einer Liste s natürlicher Zahlen $|length(s)|$ Stapel von Quadraten, wobei der i -te Stapel aus $s(i)$ Quadraten besteht.

cant col mode n erzeugt eine absteigende Cantorsche Diagonalkurve durch eine $n * n$ -Matrix.

snake col mode n erzeugt eine horizontale Schlangenlinie durch eine $n * n$ -Matrix.

phylllo col mode n erzeugt eine Kurve, deren n Ecken in einer auf dem Goldenen Schnitt basierenden Weise angeordnet sind. Die Anordnung entspricht dem – Phyllotaxis genannten – Blattstand zahlreicher Pflanzen.

leaf col col' mode r k fügt die zwei Kreisbögen **arc col mode 1000 [r] k** und **arc col' mode [r] k** zu einem liegenden Blatt zusammen.

blosLeaf col mode double n ks erzeugt eine Blüte mit $lg = n * |ks|$ aus Aufrufen von **leaf** gebildeten Blättern. Für alle $1 \leq k \leq |ks|$ liegt das $(n * k)$ -te Blatt senkrecht zu einem Kreis mit Radius $ks!!k$ um das Zentrum der Blüte. Es entspricht $leaf(col_1)(col_2)(mode)(100)(k)$, wobei im Fall $double = True$ col_1 und col_2 die $(2 * n * k)$ -te bzw. $(2 * n * k + 1)$ -te Farbe im Farbkreis $hue(1)(col)(2 * lg)$ ist und im Fall $double = False$ $col_1 = col_2$ die $(n * k)$ -te Farbe im Farbkreis $hue(1)(col)(lg)$ ist.

blosCurve col n c erzeugt eine Blüte mit n aus der Kurve c gebildeten Blättern. Für alle $1 \leq k \leq n$ liegt das k -te Blatt senkrecht zu einem Kreis mit Radius $ks!!k$ um das Zentrum der Blüte. Es entspricht $c(col')$, wobei col' die k -te Farbe im Farbkreis $hue(1)(col)(n)$ ist.

kpath/ktour col mode n a b erzeugt einen jedes Feld eines $n * n$ -Schachbretts genau einmal besuchenden Springerweg bzw. -kreis mit Startpunkt (a, b) .

snow huemode mode d n k c bildet aus k^n Kopien der Kurve c eine die Kochsche Schneeflocke der Tiefe n verallgemeinernde Figur: Die Größen und Positionen der Hexagramme, aus denen die Schneeflocke zusammengesetzt ist.

gesetzt ist, entsprechen den Skalierungen bzw. Positionen der Kopien von c . Für alle $0 < i \leq n$ sind jeder Kopie von $scale(1/3^{i-1})(c)$ k kreisförmig angeordnete Kopien der Kurve $c_i = scale(1/3^i)(c)$ zugeordnet. Die reelle Zahl d bestimmt den Radius der Kreise und sollte im Intervall $[-2, 2]$ liegen. $mode \in \{1..6\}$ legt die Ausrichtung der Kopien fest: nach Norden, nach Süden, alternierend, zum Zentrum des Kreises hin, vom Zentrum weg, usw. Ist $mode$ gerade, dann wird eine weitere Kopie von c_i ins Zentrum jedes Kreises gezeichnet. Ist col die Startfarbe von c , dann ist im Fall $huemode < 4$ $hue(huemode)(col)(n)(i)$ die Startfarbe aller Kopien von c_i und im Fall $huemode > 4$ für alle $k^{i-1} < j \leq k^i$ $hue(huemode - 3)(col)(k^i)(k^{i-1} + j)$ die Startfarbe der j -ten Kopie von c_i .

5 Kurven aus Turtle-Aktionen erzeugen

`turtle mode acts` erzeugt aus einer Folge $acts$ von Aktionen des Typs

```
data Action = Turn Float | Move Float | Jump Float | Put Curve | Skip |
    Open RGB Float Bool | Close | Draw
```

die Kurve, die eine Schildkröte zeichnet, wenn sie $acts$ ausführt.

Turn(a) veranlasst sie zur Rechtsdrehung um a Grad. *Move(d)* lässt sie von ihrer gegenwärtigen Position aus eine d Pixel lange Linie zeichnen und an deren Endpunkt wandern. *Jump(d)* entspricht *Move(d)* ohne das Zeichnen der Linie. *Put(c)* lässt sie die Kurve c so zeichnen, dass deren Zentrum ihrer aktuellen Position entspricht. Zum jeweiligen Zustand der Schildkröte gehören neben ihrer Position auch ihre Orientierung (in Grad; Startwert 0) und ein Skalierungsfaktor (Startwert 1). Bevor sie c zeichnet, dreht sie c in ihre Richtung und multipliziert die Punkte von c mit ihrem aktuellen Skalierungsfaktor.

Die jeweils aktuelle Position der Schildkröte ist der letzte Punkt eines Pfades, der initialisiert wird, wenn sie *Open(col)(sc)(smooth)* ausführt; erweitert, wenn sie *Move(d)* ausführt; und zeichnet, wenn sie *Draw* oder *Close* ausführt. Im letzten Fall kehrt sie außerdem an den Anfang des Pfades zurück. Um eine solche Rekursion in beliebiger Tiefe realisieren zu können, verwaltet die Schildkröte einen Zustandskeller mit Einträgen der Form (a, sc, ps, col, b) vom Typ $(Float, Float, Path, RGB, Bool)$. Die fünf Komponenten eines Zustands sind die jeweilige Orientierung a der Schildkröte in Grad; der Skalierungsfaktor sc , mit dem sie zeichnet; der Kantenzug ps , den sie gerade aufbaut; die Farbe col , in der sie ihn zeichnet, und ein Boolescher Wert, der angibt, ob ps geglättet wird oder nicht.

Open und *Close* führen die push- bzw. pop-Befehle auf dem Keller aus, die anderen Aktionen verändern nur den aktuellen Zustand (= obersten Kellereintrag; siehe Haskell-Implementierung von *turtle*).

Die folgenden Funktionen erzeugen Kurven vom Typ *Curve* aus Aktionsfolgen vom Typ *[Action]*: Ihr Parameter *col* und *mode* sind wieder die Startfarbe bzw. der oben beschriebene fünfstellige Zahlencode, die bestimmen, wie die Kurve gezeichnet wird.

`hilb col mode n` erzeugt eine Hilbertkurve der Tiefe n .

`pytree col mode n` erzeugt einen Pythagoreischen Baum der Tiefe n .

`pytree` entspricht `growR[(0,0),(0,-90),(36,-135),(90,-90),(90,0),(0,0)][False,True,True]` (s.u.).

`gras/L col mode n` erzeugt einen nach rechts geneigten Grashalm der Tiefe n . *grasL* färbt seine Teile abhängig von ihrer jeweiligen Tiefe in einem von *mode* bestimmten Farbkreis (siehe Abschnitt 3).

`grasC col mode n c` erzeugt einen nach rechts geneigten Grashalm der Tiefe n und positioniert die Kurve c am Ende jedes Zweiges.

`fern/L col mode n` erzeugt einen nach rechts geneigten Farnzweig der Tiefe n . *fernL* färbt seine Teile abhängig von ihrer jeweiligen Tiefe in einem von *mode* bestimmten Farbkreis (siehe Abschnitt 3).

`fernU/UL col mode n` erzeugt einen senkrechten Farnzweig der Tiefe n . *fernUL* färbt seine Teile abhängig von ihrer jeweiligen Tiefe in einem von *mode* bestimmten Farbkreis (siehe Abschnitt 3).

bush/L col mode n erzeugt einen Busch der Tiefe n . *bushL* färbt seine Teile abhängig von ihrer jeweiligen Tiefe in einem von *mode* bestimmten Farbkreis (siehe Abschnitt 3).

dragon/F col mode n erzeugt eine Drachenkurve der Tiefe n .

fibo col mode n a b erzeugt einen Kantenzug *path* mit n Eckpunkten, die wie folgt mit dem Booleschen Fibonaccistrom *fib* korrelieren: *path*($i + 1$) hat bzgl. *path*(i) die Winkelkoordinate a bzw. $-b$, falls *fib*(i) den Wert 0 bzw. 1 hat. *fib* ist die erste Komponente der eindeutigen Lösung des Gleichungssystems

```
fib = concat fibs
fibs = [0]:zipWith (++) fibs ([1]:fibs)
```

dragon col mode n erzeugt einen Kantenzug *path* mit n Eckpunkten, die wie folgt mit dem Boolesche strom *drag* korrelieren: *path*($i + 1$) hat bzgl. *path*(i) die Winkelkoordinate a bzw. $-b$, falls *fib*(i) den Wert 0 bzw. 1 hat. *drag* ist die erste Komponente der eindeutigen Lösung des Gleichungssystems

```
drag = zip blink drag
blink = 0:1:blink
zip (x:s) t = x:zip t s
```

bunch n erzeugt einen vollständigen quintären Baum der Tiefe n mit dem Mode 12111 und der Startfarbe Rot, dessen Blätter Hexagramme sind.

grow c f cs wendet die Transformation f auf c an und lässt die Elemente der Kurvenliste cs wie Zweige aus den Kanten von c herauswachsen, wobei die Länge einer Kante von c die Skalierung und ihre Steigung die Ausrichtung der aus ihr herauswachsenden Kurve von cs bestimmt.

Für alle Listen $bs = [b_1, \dots, b_k]$ Boolescher Werte wendet **growR ps bs f col mode n** die Funktion

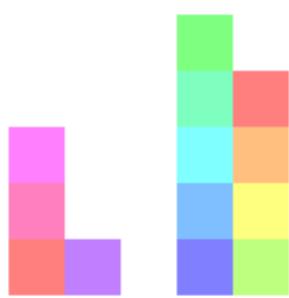
$$g = \lambda c. grow(c)(f)[c_1, \dots, c_k]$$

auf die Kurve $C([], [col], [mode], [ps])$ an, wobei für alle $1 \leq i \leq k$ aus $b_i = \text{True}$ $c_i = c$ folgt und im Fall $b_i = \text{False}$ c_i leer ist. Anschließend wird g auf alle vom ersten Aufruf von g erzeugten Kopien von c angewendet. Auf die von den neuen g -Aufrufen erzeugten Kopien von c wird wieder g angewendet, usw.

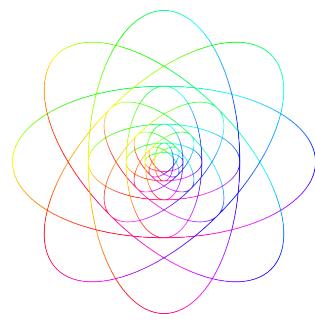
Dieser Vorgang wird n -mal wiederholt, so dass am Schluss eine aus m^n Kopien von c gebildete Figur entstanden ist, wobei m die Anzahl der Zahlen $1 \leq i \leq k$ mit $b_i = \text{True}$ ist.

base inner c verändert die Kurve c , wenn das Zentrum von c oder der Startpunkt des ersten Kantenzuges von c mit dem Nullpunkt übereinstimmt. Dann wird c so skaliert und ausgerichtet, dass die letzte Kante des ersten Kantenzuges von c mit der Linie vom Punkt $(0, 90)$ zum Nullpunkt übereinstimmt. Ein Aufruf der Form $grow(base(inner))(c))(f)(cs)$ lässt die Kurven von cs im Fall *inner=True* vom Zentrum von c weg und im Fall *inner=False* zu diesem hin aus aus c herauswachsen.

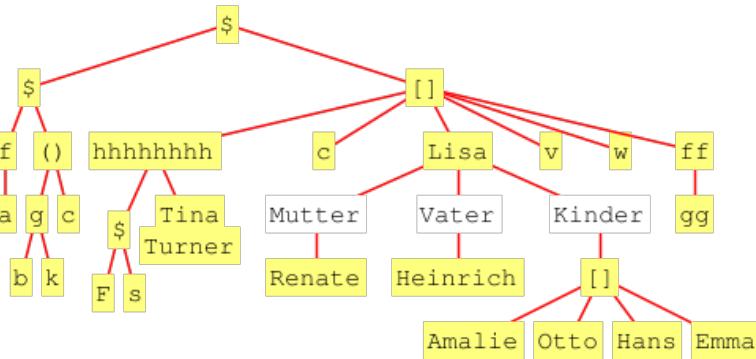
6 Beispiele



drawC1 \$ piles red 31111 [3,1,0,5,4]



drawC orbits (siehe Painter.hs)

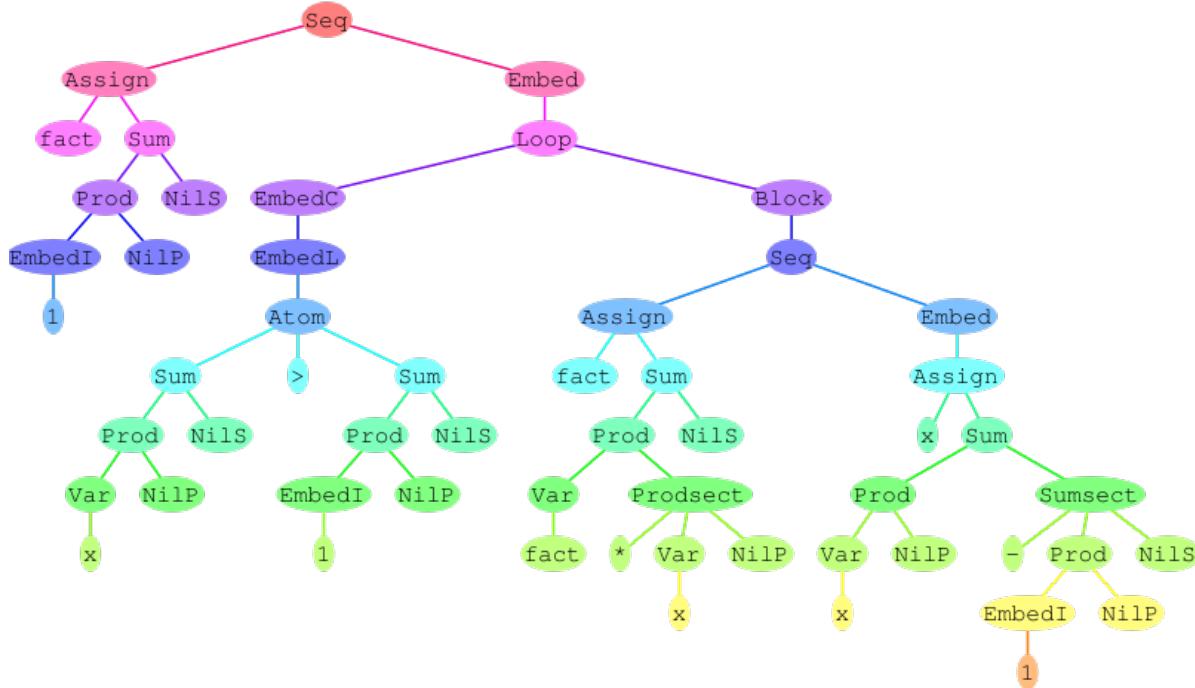


drawTerm1 "lisa2"

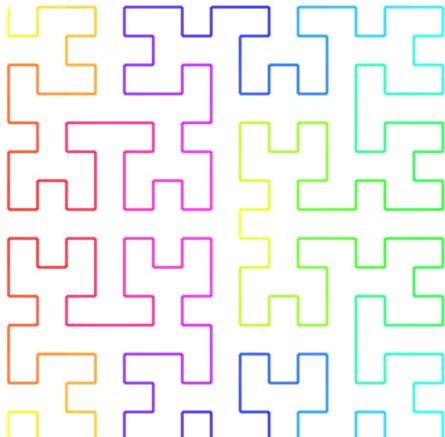
lisa2 enthält

$f\ a(g(b,k),c)[hhhhhhh(F\ s,Tina|Turner),c,$

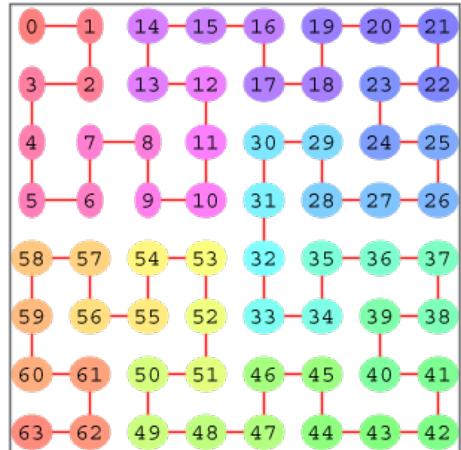
Lisa{Mutter=Renate,Vater=Heinrich,Kinder=[Amalie,Otto,Hans,Emma]},v,w,ff(gg)].



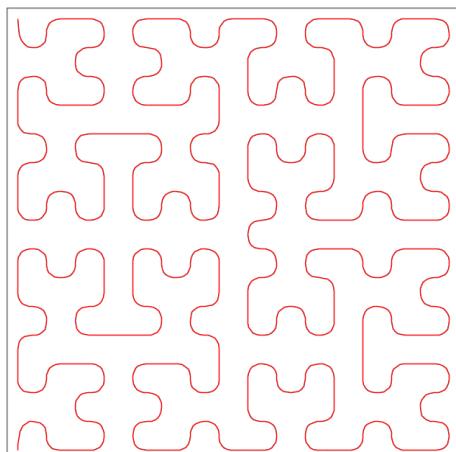
drawTermC1 "javaterm" (syntax tree of an iterative program for the factorial function)



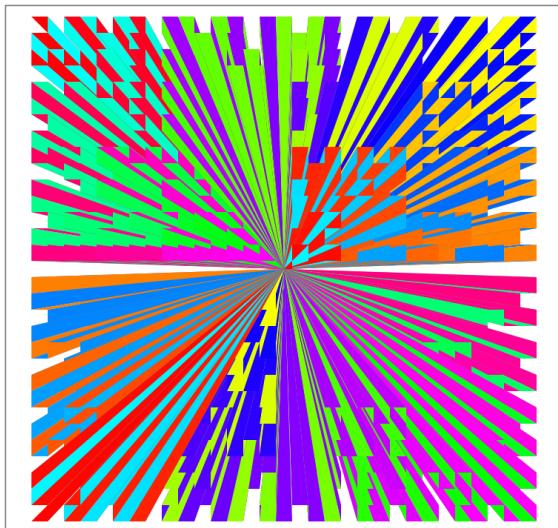
`drawC $ hilb yellow 12112 4`



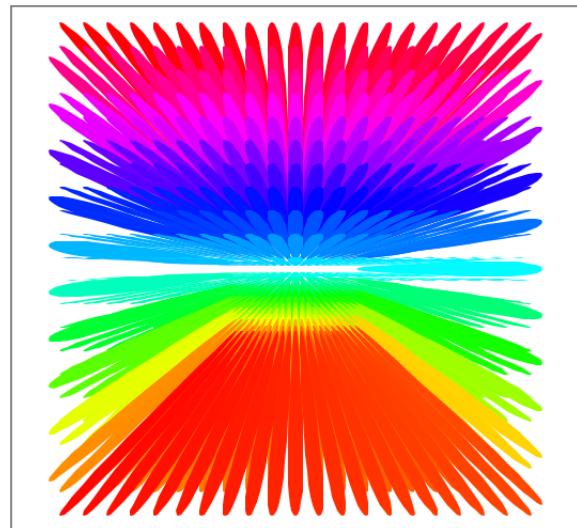
`drawC $ hilb red 42111 3`



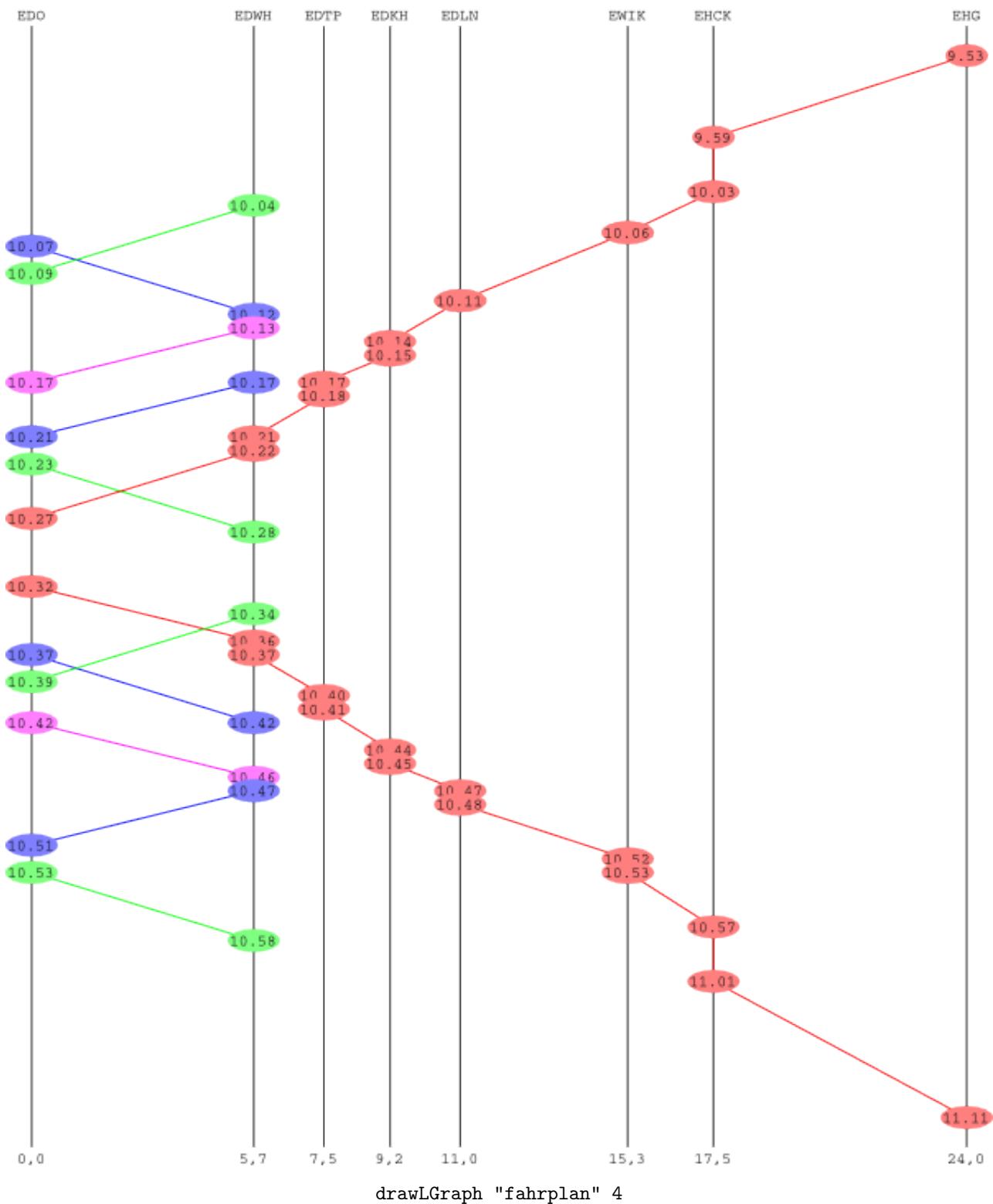
`drawC $ hilb 14211 4`

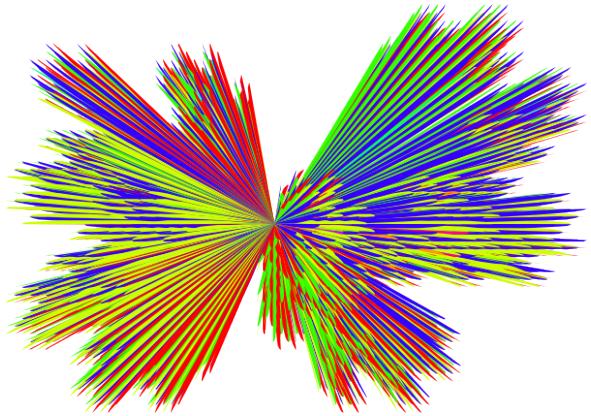
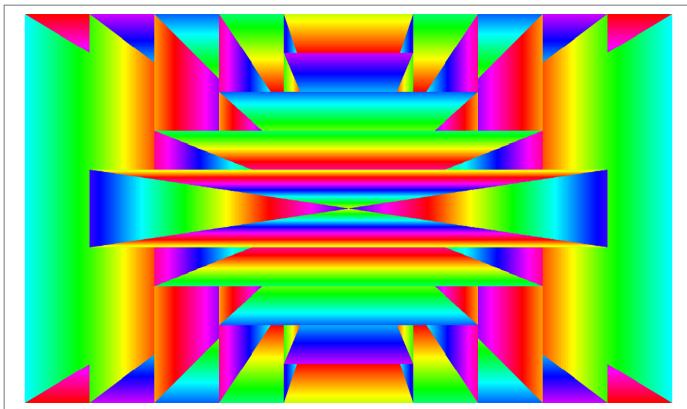


`drawC $ hilb red 13121 5`

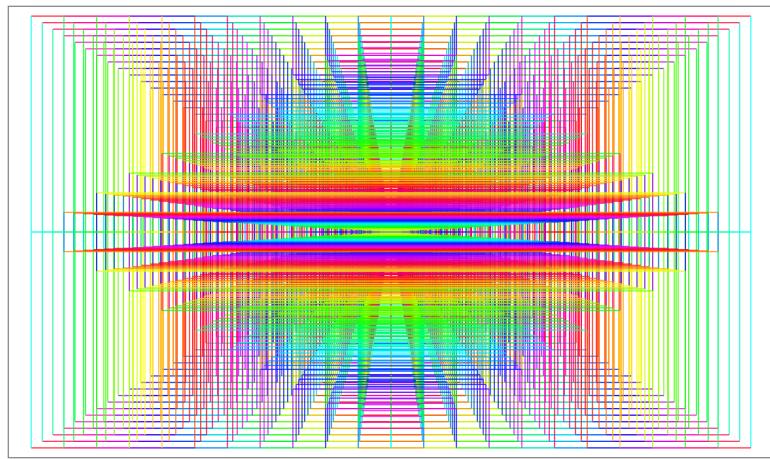
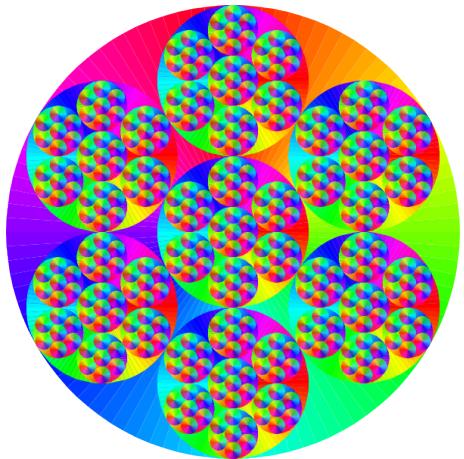


`drawC $ snake red 13211 22`

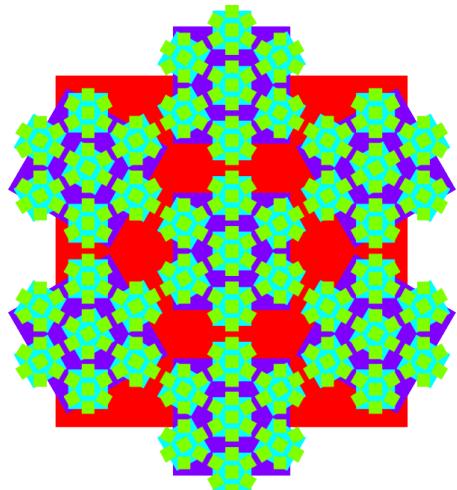
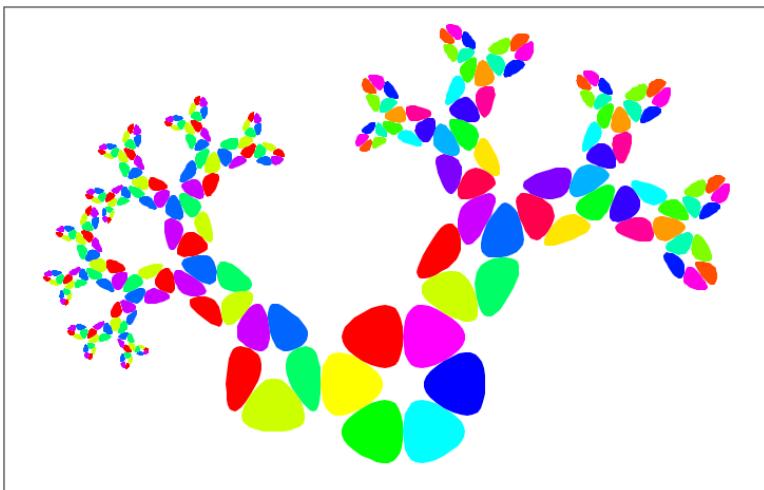




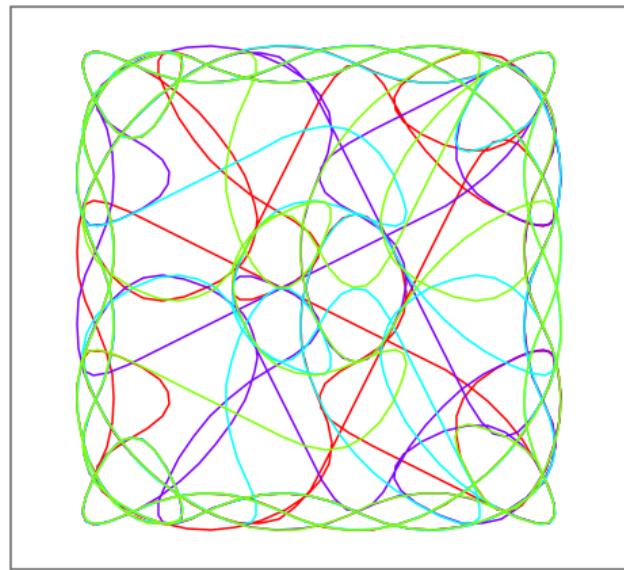
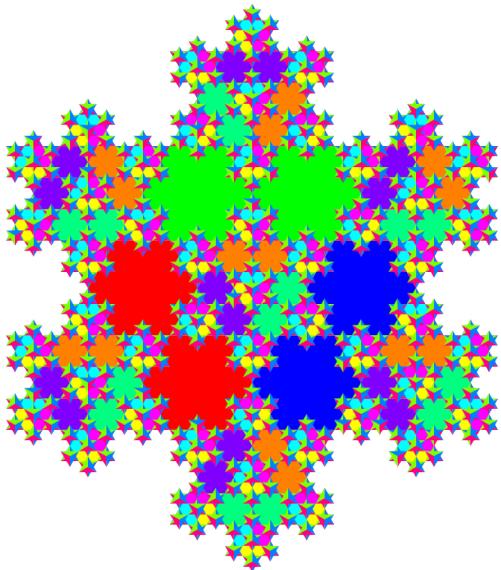
```
drawC $ morphing 1 5 $ map (rainbow 1 33 . rect cyan 13114) [(100,60),(0,60),(100,0)]
drawC $ dragonF red 13213 3333
```



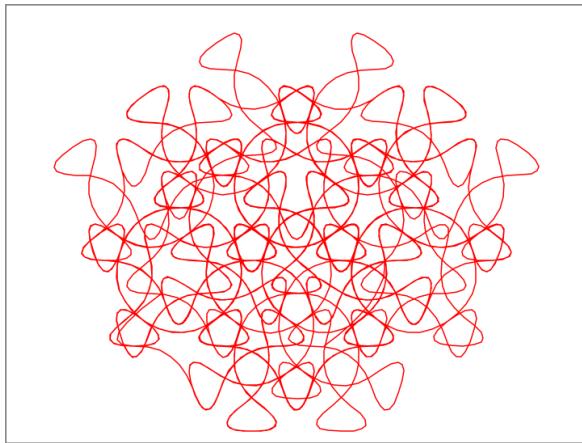
```
drawC $ snow 1 4 1 4 6 $ circ red 13113 222
drawC $ morphing 1 11 $ map (rainbow 1 33 . rect cyan 12114) [(100,60),(0,60),(100,0)]
```



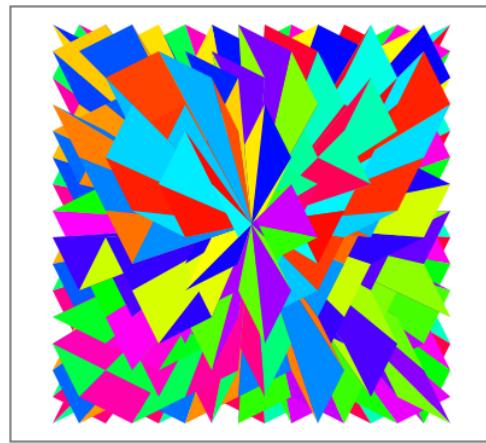
```
drawC $ grow1 13211 (siehe Painter.hs)      drawC $ snow 1 6 1 4 6 $ rect red 15111 (222,222)
```



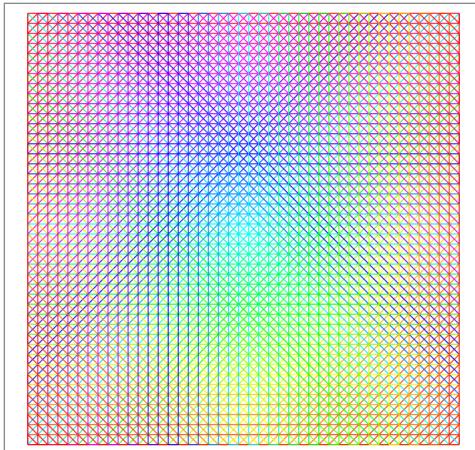
```
drawC $ snow 1 6 1 4 6 $ combine $ map (tria red 13113) [222,-222]
drawC $ hueCol 1 red $ overlay $
concatMap ((\c -> [c,flipH c]) . (\n -> ktour red 14211 8 n n)) [1,8]
```



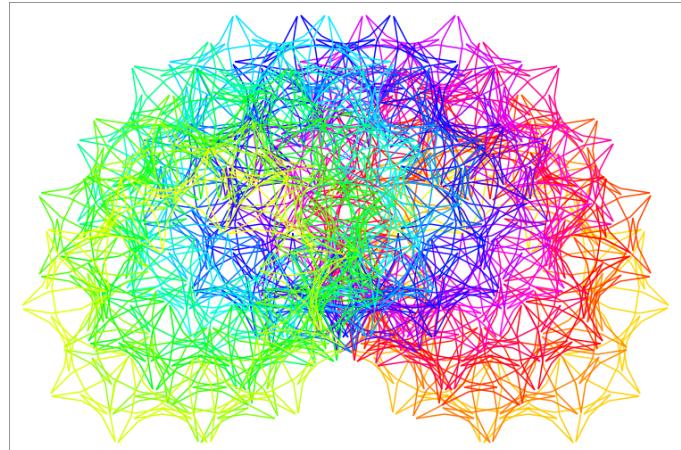
```
drawC $ flipH $ turn 72 $ fibo red 14211 500 144 72
```

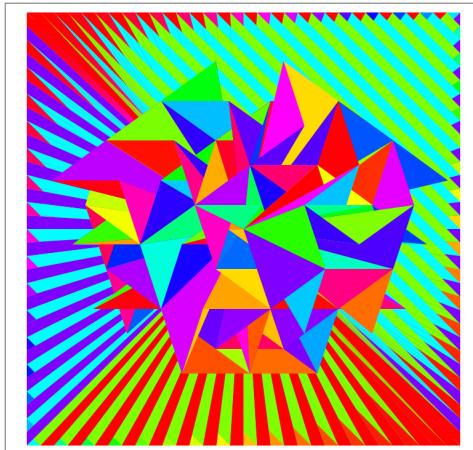
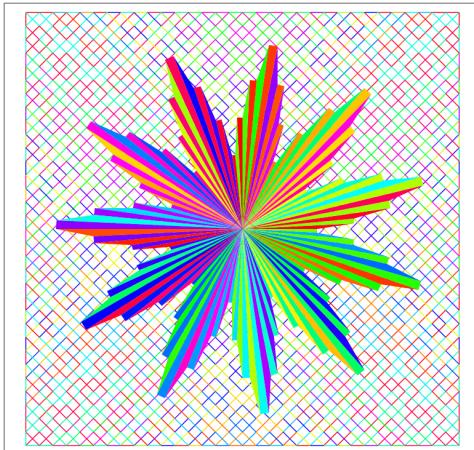


```
drawC $ ktour red 13121 16 8 8
```

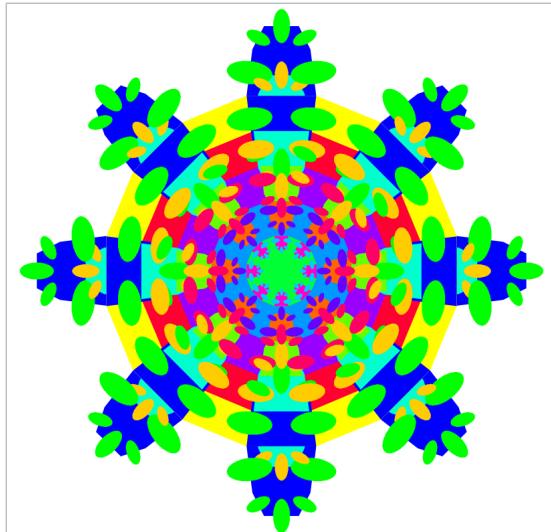
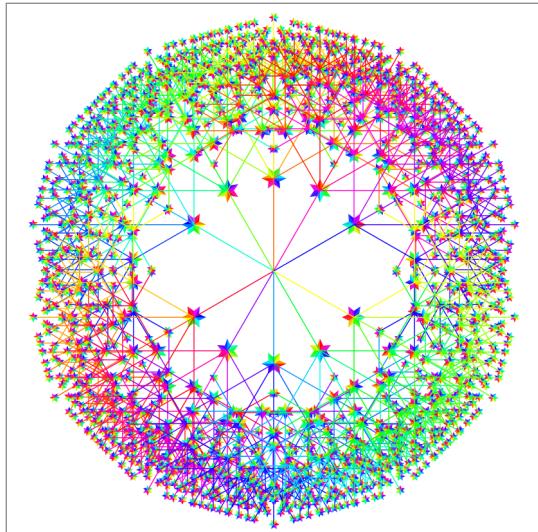


```
drawC $ overlay [f cant,flipV $ f cant,f snake,transpose $ f snake] - f g = g red 12111 44
drawC $ turn 225 $ fibo yellow 12111 4150 18 162
```

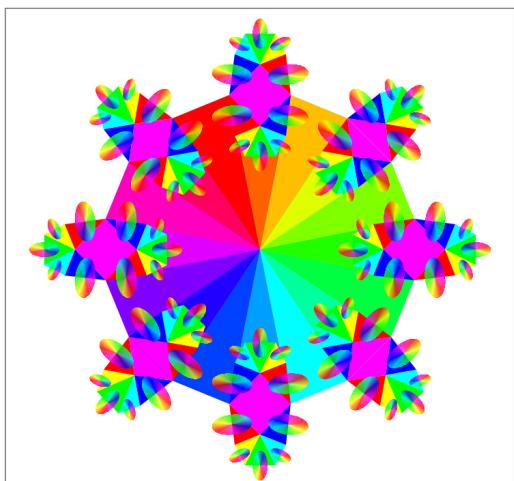




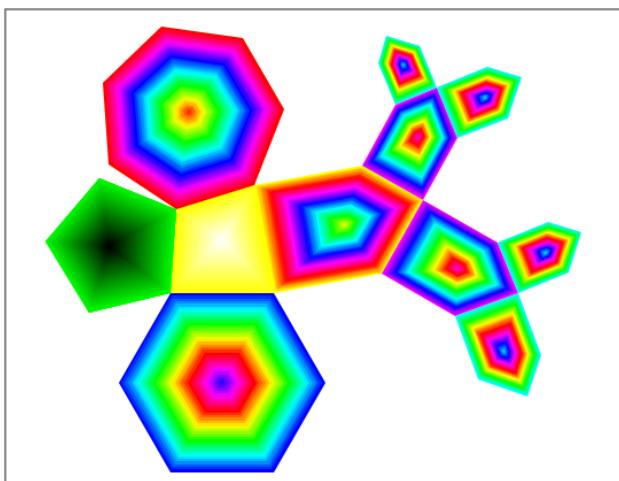
```
drawC $ overlay [cant red 12121 33,flipV $ cant red 12121 33,scale 0.25 $ poly red 13123 11
$ map (11*) [2,2,3,3,4,4,5,5,4,4,3,3]]
drawC $ overlay [cant red 13111 33,flipV $ cant red 13113 33,flipH $ turn 72
$ fibo red 13124 500 144 72]
```



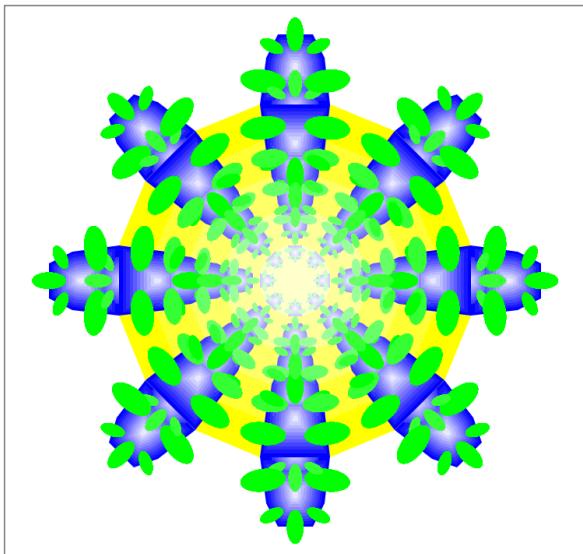
```
drawC $ combine [f blue,flipH $ f yellow] where f col = hueCol 1 col $ bunch 4
drawC $ grow7 (rainbow 1 5) id id 15111 (siehe Painter.hs)
```



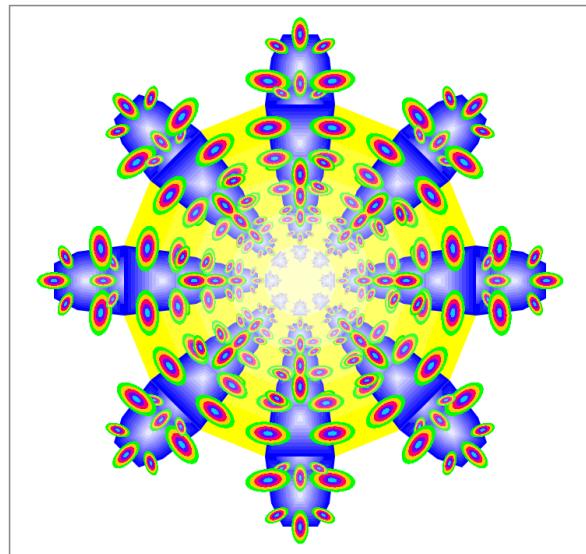
drawC \$ grow7 id id id 13111



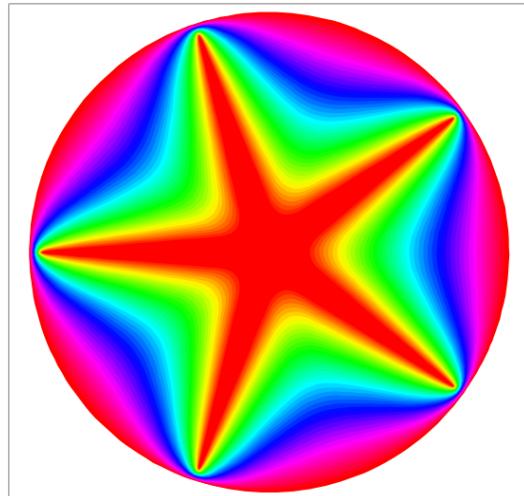
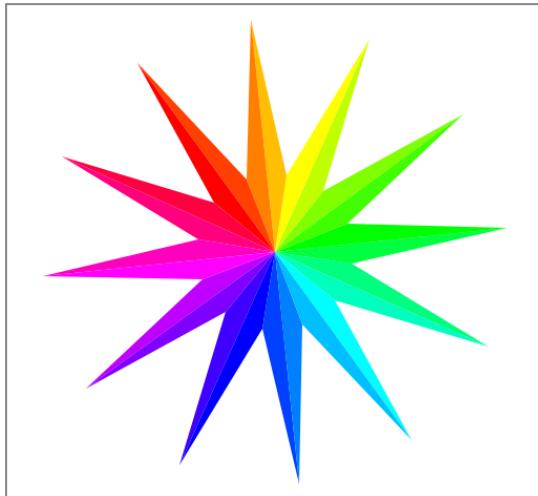
drawC \$ grow2 15111 (siehe Painter.hs)



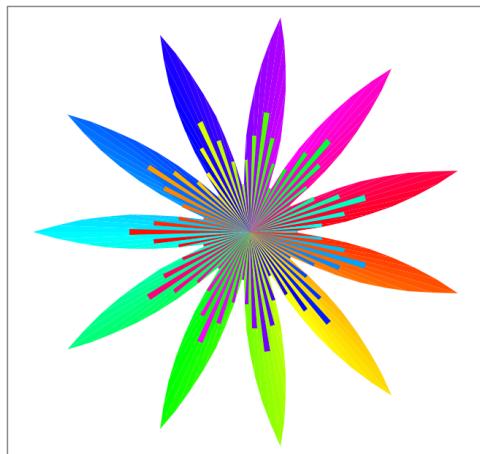
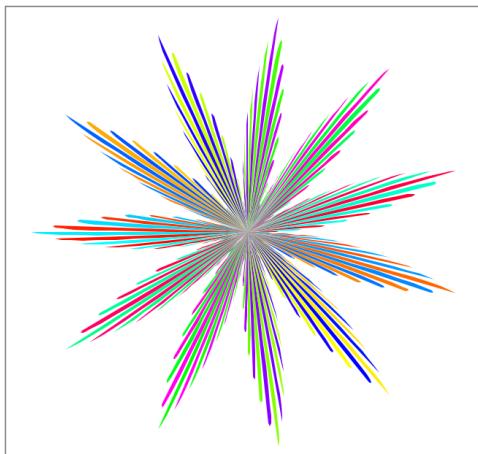
`drawC $ f id 15111`



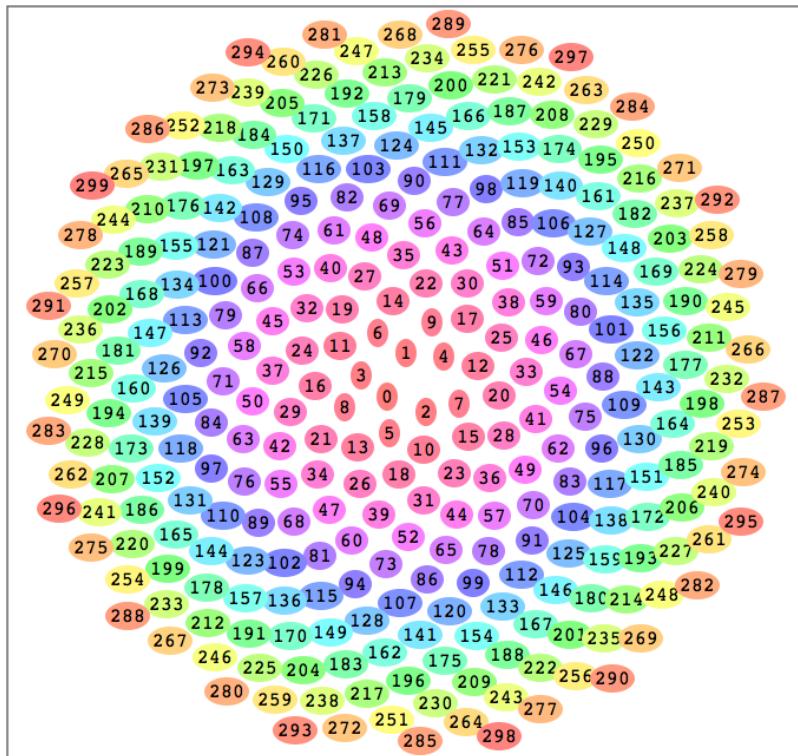
`drawC $ f (rainbow 1 5) 15111 where f = grow7 (shine 1 5) (shine 1 11)`



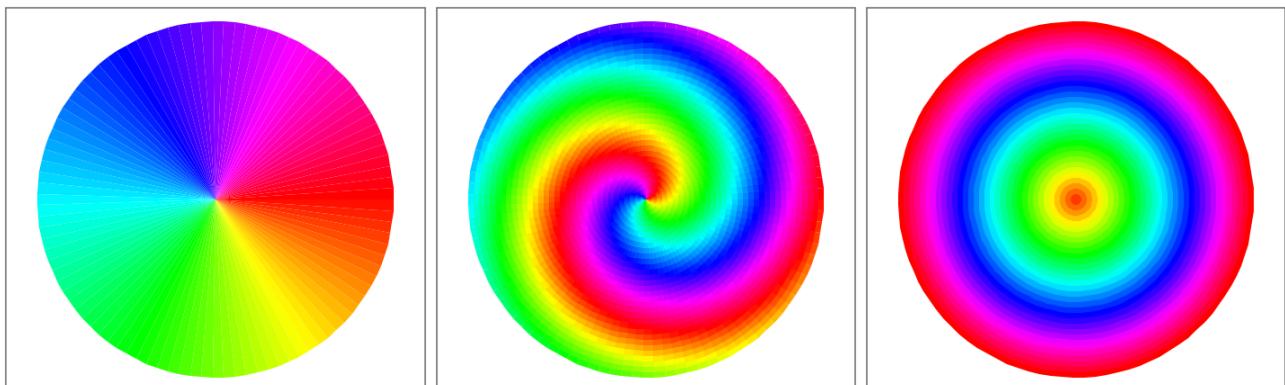
`drawC $ turn 54 $ poly cyan 13111 12 [111,37]
drawC $ morphing 1 44 [poly red 15211 30 [40],poly red 15211 5 [5,40]]`



`drawC $ poly red 13221 11 rs
drawC $ combine [poly red 13111 11 rs, turn (-1.5) $ poly 13121 11 rs']
where rs = [1..9]++reverse [2..8]; rs' = [2,2,3,3,4,4,5,5,4,4,3,3]`



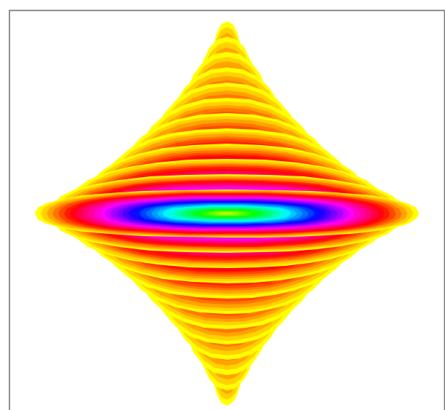
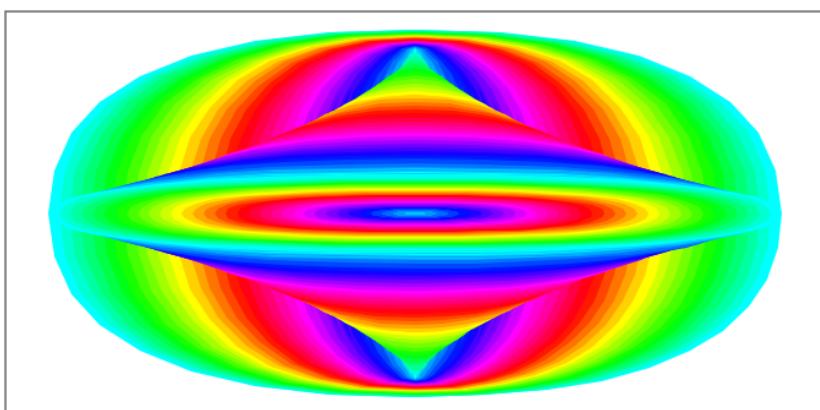
```
drawC $ phyllo red 41111 300
```



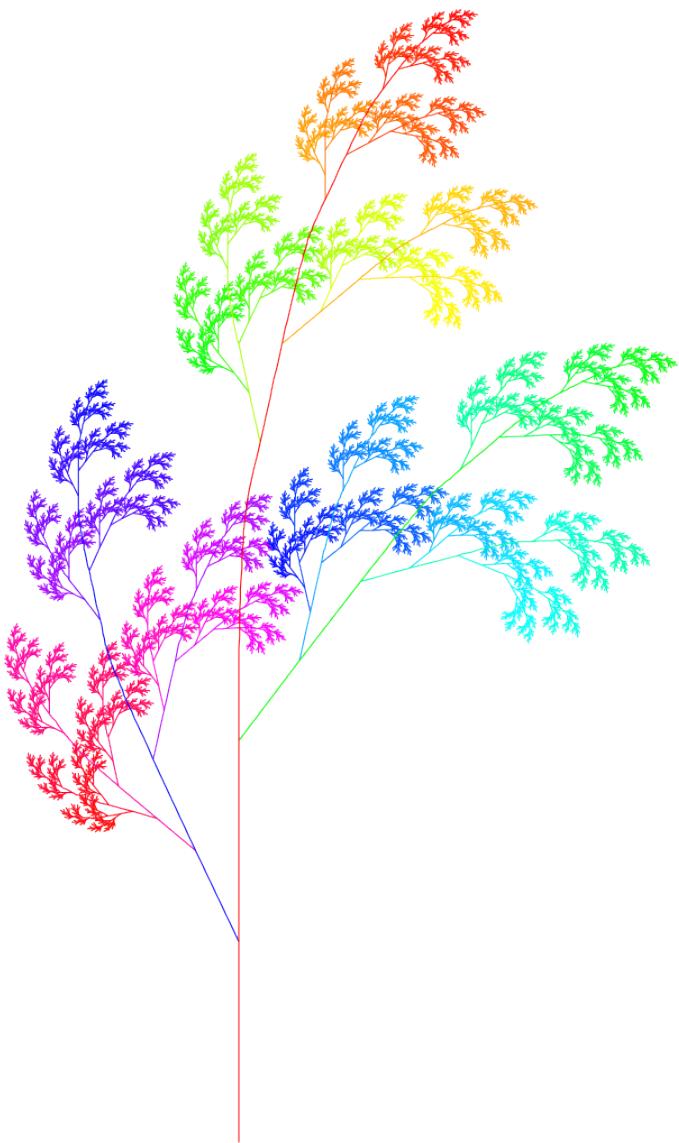
```
drawC $ circ red 13111 33
```

```
drawC $ rainbow 1 33 $ circ red 13111 33
```

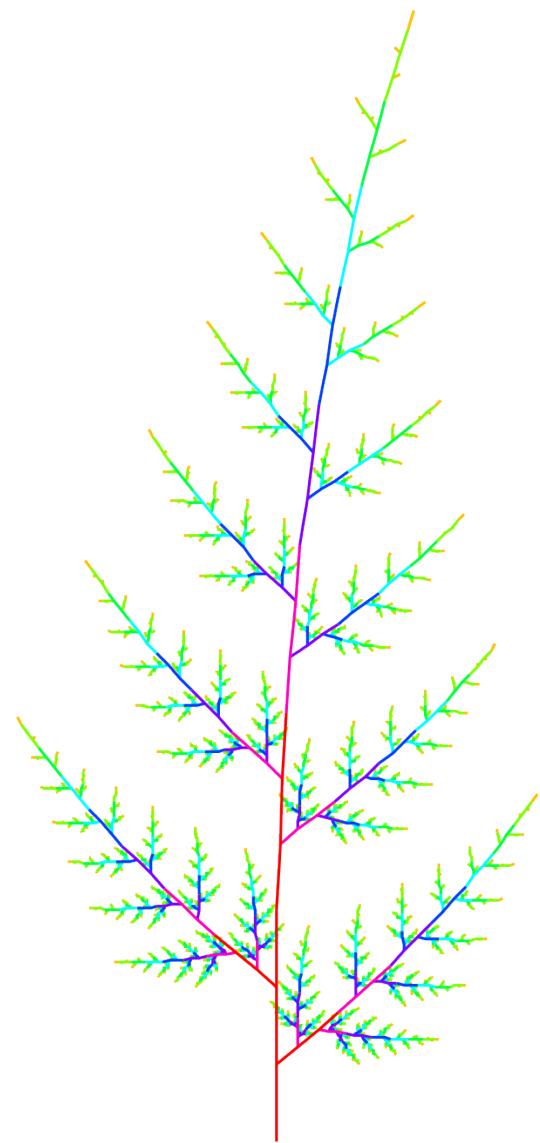
```
drawC $ rainbow 1 33 $ circ red 15111 33
```



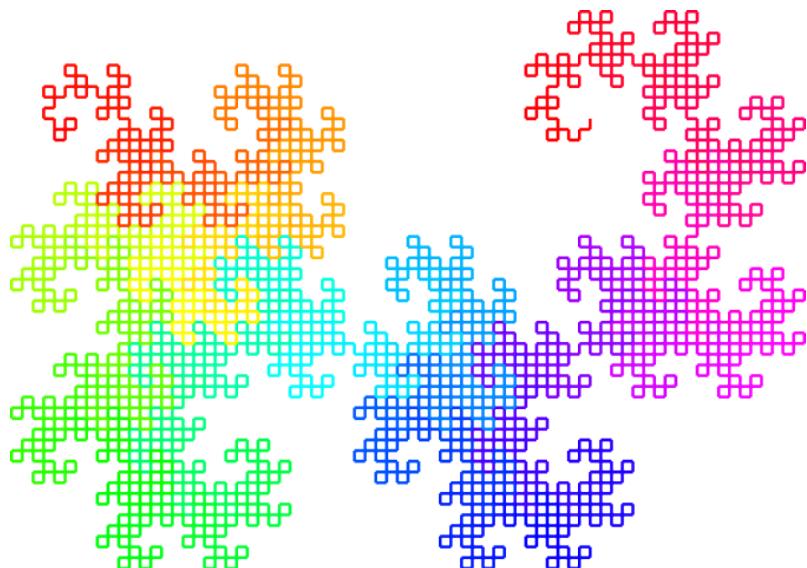
```
morphing 1 33 $ map (rainbow 1 33 . rect cyan 15211) [(100,50),(2,45),(100,10)]  
morphing 4 11 $ map (rainbow 1 33 . rect yellow 15211) [(11,88),(88,11)]
```



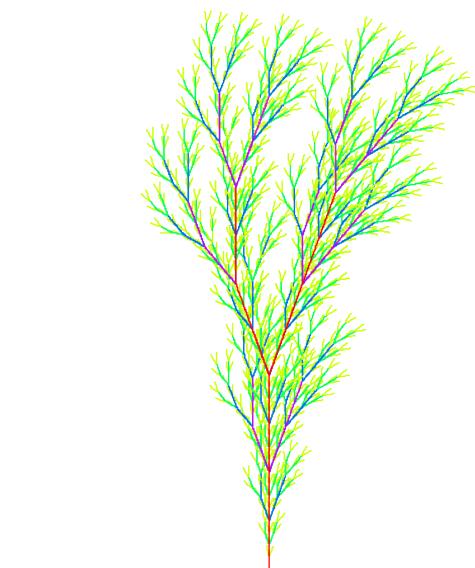
`drawC $ hueCol 1 $ gras red 14111 8`



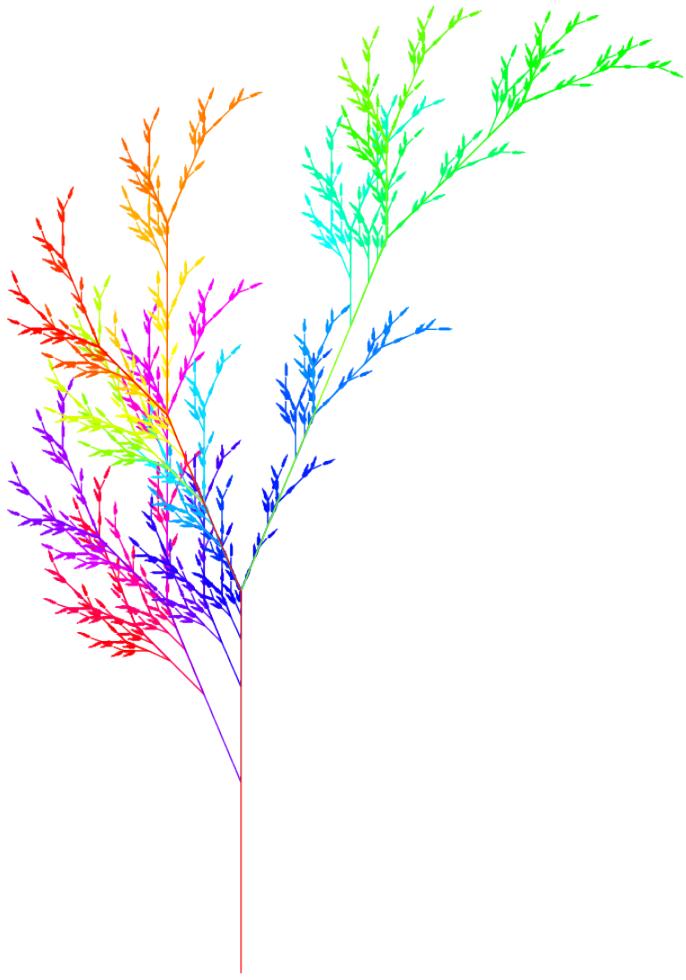
`drawC $ fernL red 14111 8`



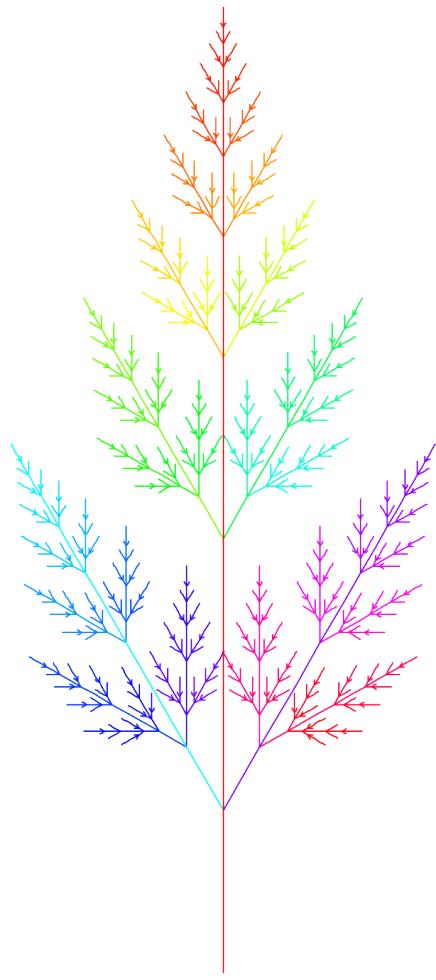
`drawC $ dragonF red 12111 3333`



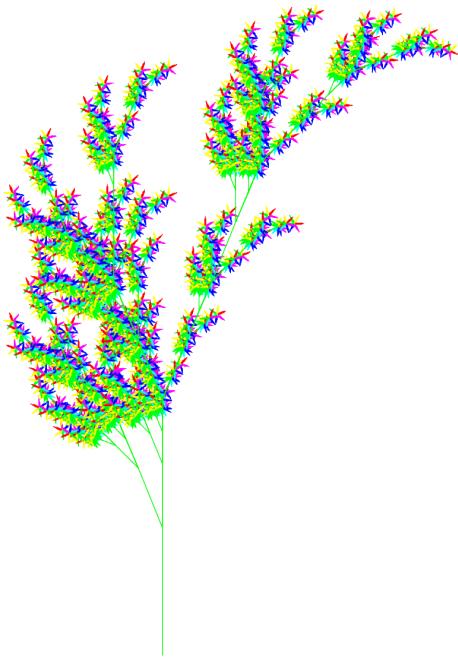
`drawC $ bushL red 14111 5`



`drawC $ hueCol 1 $ gras red 14111 8`



`drawC $ fernL red 14111 8`

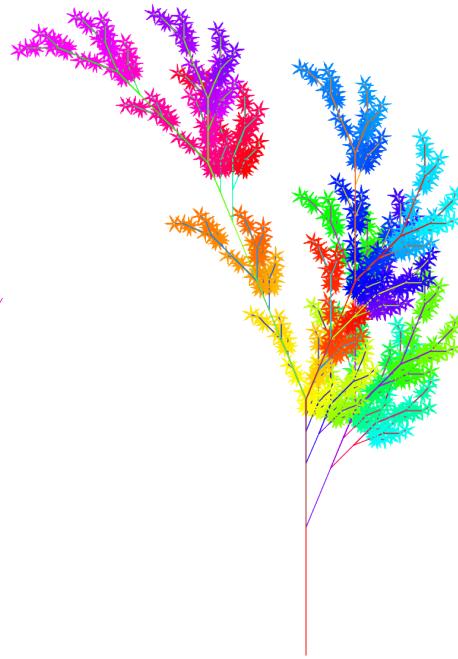
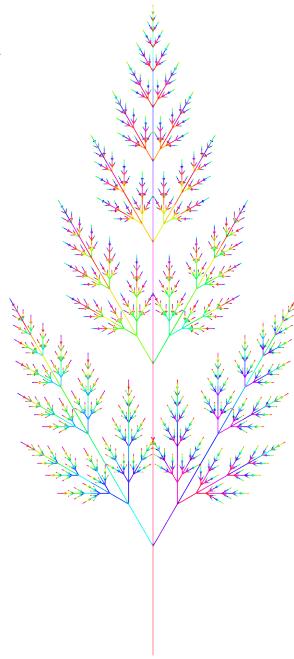


`drawC $ grasC green 14111 5 $ blosR red`

`drawC $ flipH $ hueCol 1 $ grasC red 14111 5 $ hueCol 1 $ blosR green`

`where blosR col = blosCurve col 6 $ \col -> C [col] [15211]`

`[(0,0),(3,-2),(16,0),(3,2),(0,0)]`



`drawC $ hueCol 1 $ fernU red 12111 12`