Manual of the Painter tool

available here: https://fldit-www.cs.tu-dortmund.de/~peter/Haskellprogs/Painter.tgz

Peter Padawitz
TU Dortmund
15. Juli 2022

Der Haskell-Modul Painter.hs enthält Datentypen und Algorithmen zur zweidimensionalen – teils 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 mir 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

```
<img id="pic" src="file" width="600" height="600">
```

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... (0,0)` 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.

`drawTermsP` (zum Zeichnen von Tableau-Ableitungen; siehe Tableau.hs), `drawNodeSet` und `drawNodeRel` werden bei jedem Aufruf nur einmal mit horizontalem und vertikalem Skalierungsfaktor 10 bzw. 40 ausgeführt.

`drawC{S}` werden bei jedem Aufruf nur einmal mit horizontalem und vertikalem Skalierungsfaktor 1 ausgeführt.

1 Typen und Farben

Zur internen Darstellung zweidimensionaler Figuren werden folgende elementare Typen verwendet:

`Point = (Float,Float)`	`LPoint = (String,Float,Float)` (*labelled point*)
`CPoint = (RGB,Float,Float)` (*colored point*)	
`Path = [Point]`	`LPath = [LPoint]` (*labelled path*)
`CPath = (RGB,Int,Path)` (*colored path*)	`CLPath = (RGB,LPath)` (*labelled and colored path*)

`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=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 Komponente von `CPath` entsprechen denen der *mode*-Komponente z_4 (siehe Kapitel 3).

2 Terme zeichnen und transformieren

Einzelheiten zur Eingabesyntax von Termen entnehme man dem Programm Painter.hs. U.a. sind mehrzeilige Knoteneinträge möglich, wobei % als Zeilenumbruch interpretiert wird.

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

`drawTerm{s}{C/Nodes} file` erwartet in der Datei *file* einen (als) Baum t (darstellbaren Ausdruck) bzw. eine Liste ts davon (ohne eckige Klammern), übersetzt t bzw. ts in svg-Code und schreibt diesen in die Datei *Pix/file.svg* bzw. die Dateien *Pix/file/i.svg*, $1 \leq i \leq length(ts)$, so dass bei deren Öffnen die Bäume als zweidimensionale Figur erscheinen.

`drawTerms{C}` ruft `drawList` auf (s.o.), erstellt also zusätzlich die Webseite *Pix/file.html*, aus der durch die Bäume gescrollt werden kann.

`drawTerm{C}F file f` wendet die Funktion *f:Tree(String)→Maybe(Tree(String))* auf den Baum in der Datei *file* an und schreibt das Ergebnis in die Datei *Pix/fileResult.html*.

Für nichtleere Listen s zeichnet `drawList file draw s` für alle $1 \leq i \leq length(s)$ 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 interaktiv oder automatisch gescrollt werden kann. *Pix* muss die JavaScript-Datei *Painter.js* enthalten, die den Steuerelementen von *Pix/file.html* Kommandos zuordnet.

`drawTermNodes file` erwartet in *file* hinter einem Baum t und einem Komma eine Liste $ps \in \mathbb{N}^+$ von Knoten(positionen) , die bei der Wiedergabe von t in *Pix/file.svg* grün gezeichnet werden. t und ps müssen in *file* durch ein Komma getrennt werden.

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

Der eingebaute **Termparser** übersetzt den Inhalt der Eingabedatei in ein Objekt der Instanz *Tree(String)* des polymorphen Typs

```
data Tree a = F a [Tree a] | E a (Tree a) | V a
```

Die übliche Eingabe für ein Objekt der Form `F(a)`$[t_1,\ldots,t_n]$ ist der String `a`$(t_1,\ldots,t_n)$. Die Symbole `;; ; >> >< /\ \/ == = /= <= < >= > $ # : + - * / ^^ ^ . !! &` werden als Infix-Operatoren mit zum Teil beliebiger endlicher Stelligkeit erkannt (mit von links nach rechts zunehmender Priorität). So wird z.B. der String `a+b` in das Objekt `F(+)[a,b]` überführt. Listen- oder Tupelstrings $[t_1,\ldots,t_n]$ bzw. $(t_1,\ldots,t_n)$ werden in das Objekt `F[]`$[t_1,\ldots,t_n]$ bzw. `F()`$[t_1,\ldots,t_n]$ übersetzt. Geschaltelte Applikationen $f(t_1)\ldots(t_{n-1})(t_n)$ werden zu `F($)[F($)[...F($)[f,`t_1`],...,`t_{n-1}`],`t_n`]`.

Bzgl. der Baumstruktur von Objekten des Typs *Tree(String)* repräsentieren solche der Form `E(a)[`t`]` Kanten mit Zielobjekt t. a bezeichnet in diesem Fall ein Attribut, einen Destruktor, ein *field label* o.ä. So wird z.B. ein Haskell-Record `c`$\{a_1 = t_1,\ldots,a_n = t_n\}$ in den Baum `F(c)[E(`a_1`)(`t_1`),...,E(`a_n`)(`t_n`)]` überführt (siehe §6, Beispiel `lisa`). Objekte der Form `V(a)` repräsentieren Variablen. Der Termparser wird mit der Liste von Strings, die als Variablen interpretiert werden sollen, parametrisiert.

Teilstrings der Form `F"str"`$[e_1,\ldots,e_n]$, `E"str"`e oder `V"str"`, die obige Konstruktoren bereits enthalten, werden in die gleichlautenden Objekte vom Typ *Tree(String)* übersetzt.

`reduce file vars` $[f_1,\ldots,f_n]$ übersetzt den Inhalt der Datei *file* in eine Liste von Gleichungssystemen und verwendet diese als Reduktionsregeln. *vars* ist ein String, dessen durch Blanks voneinander getrennte Teilwörter die Variablen bilden, die im Term und den Gleichungen von *file* vorkommen. Der gesamte Inhalt von *file* wird gemäß folgender Grammatik parsiert:

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

Er muss also die Form $t;;eqs_1;;\ldots;;eqs_k$ haben, wobei t der zu reduzierende Term ist und $eqs_1,\ldots,eqs_k$ Listen von Gleichungen sind, die bei der Reduktion nach der partiellen Auswertung mit

$$eval : Tree\ String \to Maybe(Tree\ String)$$

auf t angewendet werden.

eval erkennt aussagenlogische und arithmetische Funktionen, Konditionale (*ite*), Sektionen wie z.B. (+5), die Nachfolgerfunktion (suc), die Listenkonstruktoren [] und (:) von Haskell, Funktionskomposition (.) und -applikation ($), λ-Abstraktionen $\lambda p.t$ (hier in der Form $(p \to t)$), Fallunterscheidungen

$$case\ t\ of\ p_1|b_1 \to t_1;\ \ldots;\ p_n|b_n \to t_n$$

(hier in der Form $(p_1|b_1 \to t_1,\ldots,p_n|b_n \to t_n)$) sowie Zugriffe auf Objektattribute (hier mit & anstelle des sonst üblichen Punktes). $b_1,\ldots,b_n$ bezeichnen Boolesche Ausdrücke, die ggf. entfallen.

Nach der partiellen Auswertung von t mit *eval* werden die Gleichungen von eqs_1 auf t und die Unterbäume 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 $f_1,\ldots,f_n$ – z.B. *distribute*, *flatten*, *impl* – angewendet (siehe Abschnitt REDUCER in Painter.hs). Die Redexauswahl in jedem einzelnen Reduktionsschritt folgt der *parallel-outermost*-Strategie: Alle maximalen reduzierbaren Teilausdrücke werden gleichzeitig durch ihre jeweiligen Redukte ersetzt.

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 einer Grünfärbung der Redukte des jeweils vorangegangenen Reduktionsschrittes und ein zweites Mal mit einer Rotfärbung der Redizes des aktuellen Reduktionsschrittes.

XCTL-Formeln beschreiben ein- oder zweistellige Relationen zwischen Knoten eines Baumes, im Painter dargestellt als Objekte des Typs *nodeSet* = *Node** bzw. *nodeRel* = *Node*→*Node**. Die Formeln selbst sind Terme über folgender Signatur und daraus abgeleiteten Operationen:

```
data XCTL nodeSet nodeRel =
     XCTL {true_,false_       :: nodeSet,
           atom               :: (String -> Bool) -> nodeSet,
           not_,ex,ax,af,eg   :: nodeSet -> nodeSet,
           and_,or_           :: nodeSet -> nodeSet -> nodeSet,
           exists,forall,div_ :: nodeRel -> nodeSet -> nodeSet,
           self,child,parent,next,prev,equal,equiv
                              :: nodeRel,
           closure,inv        :: nodeRel -> nodeRel,
           par,seq_           :: nodeRel -> nodeRel -> nodeRel,
           restrict           :: nodeRel -> nodeSet -> nodeRel }
```

Die Bedeutung der Operationen ergibt sich aus ihrer Implementierung im Abschnitt XCTL von *Painter.hs*. Eine weiterentwickelte Version findet sich im Abschnitt Σ-*formulas* von Fixpoints, Categories, and (Co)Algebraic Modeling, Kapitel 9.

Zwei Beispielformeln vom Typ *nodeSet* bzw. *nodeRel*:

```
EG((<=22)\/66)
(child >> (a /\ exists(descendant,d))) / child / (descendant >> c)
```

`descendant` is `closure(child)`, `>>` denotes `restrict`, `/` denotes `div_`. Sei *form* eine XCTL-Formel des Typs *nodeSet*, deren Semantik *set(t)* jedem Baum t des Typs *Tree(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 $set(t)$ grün gefärbt.

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

`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 $1 \leq i \leq length(ws)$ in die Datei *Pix/file_ rel/i.svg*. Dabei werden der Knoten $ws!!(i-1)$ rot und die Knoten von $rel(t)(ws!!(i-1))$ grün gefärbt.

SQL-Formeln beschreiben Tabellen im Sinne von Relationen mit Spaltennamen (Attributen), im Painter dargestellt als Objekte des Typs $Table(a) = Map(String)[a]$ oder als Liste von Zeilen des Typs $Row(a) = Map(String)(a)$, wobei *Map(a)(b)* der vom Haskell-Modul *Data.Map.Strict* zur Verfügung gestellte Typ endlicher Funktionen von a nach b ist. Die Formeln selbst sind Terme über folgender Signatur von Operationen zur Bildung bzw. Manipulation von Tabellen:

```
data SQL table row a =
     SQL {project         :: [String] -> table -> table,
          select          :: (row -> Bool) -> table -> table,
          prod,div_,njoin :: table -> table -> table,
          tjoin           :: (row -> Bool) -> table -> table -> table,
          ejoin           :: String -> String -> table -> table -> table,
          getAttrs        :: table -> [String],
          getVal          :: String -> row -> a,
          getTable        :: String -> IO table,
          getfromDB       :: String -> String -> IO table,
          pictTerm        :: table -> String -> String,
          putHere         :: table -> IO table,
          putPict         :: table -> String -> IO table,
          emptyTab        :: table,
          mkTab           :: [row] -> table,
          mkRows          :: table -> [row],
          getVal          :: String -> row -> a,
          readVal         :: Parser a,
          errVal          :: a,
          addVal          :: a -> a -> a}
```

Die Bedeutung dieser Operationen ist im Abschnitt SQL von *Painter.hs* als `SQL-Algebra` *sql* vom Typ

```
SQL (Map String [Val]) (Map String Val) Val
```

definiert, wobei `String` die Attribute und

```
data Val = N String | I Int | R Float deriving (Eq,Ord)
```

die Attributwerte (Tabelleneinträge) liefert. Der Typ `Map String [Val]` repräsentiert eine Tabelle *tab* als Funktion, die jedem Attribut die *Spalte* seiner Werte in der Matrixdarstellung von *tab* zuordnet.

Eine weiterentwickelte Version tabellenerzeugender und -manipulierender Formeln findet sich im Abschnitt Σ-*formulas* von Fixpoints, Categories, and (Co)Algebraic Modeling, Kapitel 9. Hier werden Attribute als *Labels*, Attributwerte als *Zustände* und Tabellen als Listen partieller Funktionen, die Labels Zustände zuordnen, modelliert, d.h. jede Zeile einer Tabelle entspricht einer solchen Funktion.

Diese Version wurde zusammen mit der o.g. Weiterentwicklung von XCTL in Expander2 implementiert, so dass entsprechende Modellierungen auf dort vorhandene graphische Darstellungsfeatures ohne den u.g. Umweg zugreifen können.

`compTable query` parsiert die SQL-Formel *query* und wertet sie – i.d.R. zu einer Tabelle – aus. Befindet sich z.B. der Term

```
db{table1 = tab{name = [Hilde,Inge,Robin],
                age = [81,85,72],
                children = [1,3,2],
                temp = [5.66,9.8,66.09]},
   table2 = tab{A = [1,4,7],
                B = [2,5,8],
                C = [3,6,9],
                D = [4,7,0]},
   table3  = tab{D = [3,6],
                 E = [1,2]}}
```

in der Datei *database*, dann wird durch den Aufruf `compTable("here$"++query++")")` die SQL-Formel

```
query = "tjoin(B<D,project([A,B,C],fromDB(table2,database)),fromDB(table3,database))"
```

zur Tabelle `tab{A = [1,1,4], B = [2,2,5], C = [3,3,6], D = [3,6,6], E = [1,2,2]}` ausgewertet und auf der Konsole ausgegeben. *tjoin(p)(t)(u)* liefert dieselbe Tabelle wie *select(p)(prod(t)(u))*.

Der Aufruf

```
compTable"pict(fromDB(table1,database),table1)"
```

erzeugt einen Term, der die Matrixdarstellung von *table1* repräsentiert, und speichert diesen in der Datei *ExpanderLib/table1*. Wird anschließend im *Solver*-Fenster von Expander2 der Interpreter *widgets* aktiviert und auf den Term `loadT$table1` (durch Drücken der *paint*-Taste) angewendet, dann zeichnet *widgets* die Matrix im *Painter*-Fenster von Expander2:

```
shelf(1)
[text$table1,white$circ$4,
 mat
 [(1,age,81),(1,children,1),
  (1,name,Hilde),(1,temp,5.66),
  (2,age,85),(2,children,3),
  (2,name,Inge),(2,temp,9.8),
  (3,age,72),(3,children,2),
  (3,name,Robin),(3,temp,66.09)]]
```

represents

table1

	age	name	temp	children
1	81	Hilde	5.66	1
2	85	Inge	9.8	3
3	72	Robin	66.09	2

`dbPict file` erzeugt eine Liste mit den Matrixdarstellungen aller Tabellen der Datenbank in *file* und speichert sie in der Datei *ExpanderLib/file*. Analog zur oben beschriebenen Erstellung der Ausgabe einer einzelnen Matrix werden durch Anwendung von *widgets* auf den Term `loadT$file` alle Matrizen im *Painter*-Fenster von Expander2 angezeigt.

3 Figuren zeichnen

Jeder Weg des Typs `[CPath]` oder `[CLPath]` einer von *drawC{S}* bzw. *drawGraph* zu zeichnenden Figur ist mit einem ganzzahligen *mode* attributiert, der angibt, wie er gezeichnet wird (Färbung, Glättung, Markierungen, etc.):

Sei $(color, mode, path) \in CPath$ ein von *drawC{S}* zu zeichnender Weg. Die ersten fünf Ziffern z_1 bis z_5 von *mode* werden wie folgt interpretiert. Modes mit weniger als fünf Ziffern werden um Einsen zu fünfstelligen Modes erweitert.

- $z_1 = 1$: Es werden keine Punkte von *path* gezeichnet.
- $z_1 \in \{2, 5\}$: Jeder Punkt von *path* wird als unterschiedlich gefärbter heller oder dunkler Kreis gezeichnet.
- $z_1 \in \{3, 6\}$: Jeder Punkt von *path* wird als unterschiedlich gefärbtes helles oder dunkles 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.
- $z_3 = 1$: Die Komponenten von *path* (Punkte oder Linien bzw. Dreiecke) werden gemäß ihrer jeweiligen Position in einer von z_5 bestimmten Permutation *perm* von *path* (s.u.) und eines gleichlangen Farbkreises unterschiedlich gefärbt: Die Farbe der i-ten Komponente von *path* hat den Wert $hue(z_4)(|path|)(color)(perm(i))$ (s.u.).
- $z_3 = 2$: Linienzüge bzw. Polygone werden vollständig geglättet, ansonsten wie $z_3 = 1$.
- $z_3 = 3$: Polygone werden außer dem Anfangspunkt geglättet, ansonsten wie $z_3 = 1$.
- $z_4 \in \{1, \ldots, 5\}$ bestimmt den Farbkreis, bezüglich dessen die Komponenten von *path* (s.o.) gefärbt werden. Bei $z_4 = 1$ sind die Hue-Werte der Farben von je zwei benachbarten Komponenten gleich weit voneinander entfernt (Regenbogeneffekt), bei $1 < z_4 < 5$ komplementär zueinander ubd bei $z_4 = 5$ identisch.
- $z_5 = 1$: Der Abstand zum Anfang von *path* bestimmt die Farbe einer Komponente von *path*.
- $z_5 = 2$: Der Abstand zum Anfang bzw. Ende von *path* bestimmt 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 bestimmt die Farbe einer Komponente von *path*.

Sei $(color, path) \in CLPath$ ein von *drawGraph(sc)(file)(mode)* zu zeichnender Weg. Seine Linien werden mit *color* gefärbt. Ausgehend von *color* werden die Punkte von *path* als unterschiedlich gefärbte Ellipsen gezeichnet und mit *label* markiert, wobei sich alle benutzten Farben zum von $mode = z_4$ bestimmten Farbkreis ergänzen (s.o.).

4 Kurven komponieren und zeichnen

Listen gefärbter Kantenzüge können auch als Objekte des Typs

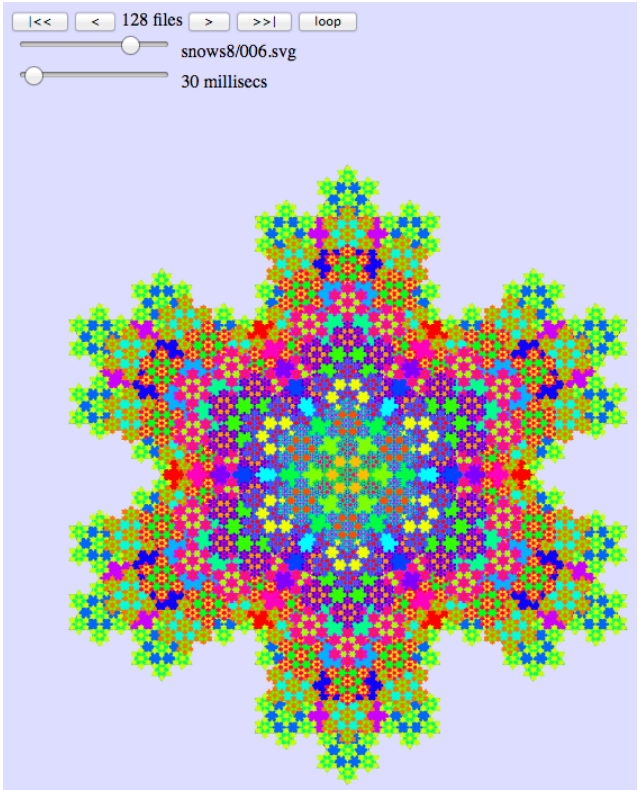
```
data Curve = C {file :: String, colors :: [RGB], modes :: [Int], paths :: [Path]}
```

dargestellt und in dieser Repräsentation verändert oder mit anderen Kantenzügen kombiniert werden. Die Attribute einer Kurve c haben folgende Bedeutung: *file(c)* ist die Datei, in die der svg-Code von c geschrieben werden soll. $paths(c)$ 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 sc c` schreibt svg-Code für die Kurve c in die Datei *Pix/file(c).svg*. Das Paar sc reeller Zahlen liefert die horizontalen bzw. vertikalen Skalierungsfaktoren der entsprechenden Zeichnung. Bei $sc = (0, 0)$ wird - wie bei *drawTerm* (siehe Abschnitt 2) - die Eingabe dieser Faktoren angefordert.

`drawCS frame 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* die Webseite *Pix/dir.html*, auf der die Kurven von cs einzeln oder als Diashow - in Originalgröße *(frame=True)* bzw. auf 600×600 Pixel skaliert *(frame=False)* - betrachtet werden können. *Pix/dir.html* greift auf die JavaScript-Datei `Painter.js` zu, die deshalb im Verzeichnis *Pix* enthalten sein sollte.

`moviePix frame dir` und `movies dir ps` erzeugen wie *drawCS* die Webseite *Pix/dir.html*. Bei *moviePix* sind dort die die Bilddateien des Verzeichnisses *dir* zugreifbar. Der Parameter *ps* von *movies* ist eine Liste von Paaren (d, i), wobei d ein Verzeichnis und i die i-te Bilddatei von d bezeichnet. Von *Pix/dir.html* aus kann vor- und rückwärts durch die Bilddateien von *ps* gescrollt werden.



Z.B. erzeugt der Aufruf

```
movies "snows1"
  [(i,j) | i <- map (("snows"++) . show)
                    [2..9],
           j <-  [0..8]++[7,6..1]]
```

die links wiedergegebene Webseite *Pix/snows1.html*, auf der vor- und rückwärts durch die Bilddateien der Unterverzeichnisse *snows2,...,snows9* von *Pix* gescrollt werden kann.

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`:

`emptyC 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 durch den fünfstelligen Zahlencode *mode* (s.o.) bestimmter Weise gezeichnet wird.

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

`hueCol m 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 $hue(m)(colors(c)!!i)(|paths(c)|)(i)$ gesetzt. Die möglichen Werte von m entsprechen denen der $mode$-Komponente z_4 (siehe Kapitel 3).

`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 m 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 ganzzahligen Wert m gewählten Farbkreis unterschiedlich gefärbt. Im Fall $m \in \{1, 2, 3, 4\}$ entspricht `m` der $mode$-Komponente z_4 (siehe Kapitel 3). Andernfalls findet keine Farbverschiebung statt.

`rainbow m n c` erzeugt n Kopien abnehmender Größe der Kurve c. Sie werden wie bei `morphing` abhängig vom vom ganzzahligen Wert m unterschiedlich gefärbt. Der Aufruf *rainbow(mode)(n)(c)* entspricht *morphing(mode)(n)[c,scale(0)(c)]*.

`shine m n c` erzeugt n Kopien abnehmender Größe der Kurve c. Bei $m = 1$ werden die Kopien schrittweise aufgehellt, bei $m = -1$ abgedunkelt.

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.

`phyllo 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 Planzen.

`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, 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

Datentyp für Turtle-Aktionen:

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

`turtle mode acts` erzeugt aus einer Folge *acts* von Turtle-Aktionen 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 Zustands*keller* 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 Boolscher 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.

`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.

6 Kurvenbäume zu Kurven auswerten

Datentyp für Kurvenbäume:

```
data CurveTree = CT Curve [CurveTree]
```

`buildCT c bs n` erzeugt einen aus Kurven der Form c zusammengesetzten Kurvenbaum mit Tiefe n. $bs = [b_1, \ldots, b_k]$ ist eine Liste Boolescher Werte, die angeben, ob ein Knoten auf der i-ten Ebene des Baumes ($i < n$) einen j-ten direkten Nachfolger hat ($b_j = \mathit{True}$) oder nicht ($b_j = \mathit{False}$).

`evalCT{C} {trans} t` wertet den Kurvenbaum $t = CT(c)[t_1, \ldots, t_n]$ rekursiv zu einer Kurve c_t aus, deren Kantenzüge aus den Knoten von t gebildete Polygone sind: c_t besteht aus einem "Wurzel"-Polygon p der Form c, an dessen n Kanten $k_1, \ldots, k_n$ die "Wurzel"-Polygone von $c_{t_1}, \ldots, c_{t_n}$ mit ihrer Kante $((0, 90), (0, 0))$ (s.u.) andocken, m.a.W.: $c_{t_1}, \ldots, c_{t_n}$ wachsen wie Zweige aus p heraus, wobei die Länge von k_i die Skalierung und Steigung von k_i die Ausrichtung von c_{t_i} bestimmen.

Zusätzlich färbt `evalCTC trans t` jedes Polygon p von t abhängig davon, auf welcher Ebene von t es sich jeweils befindet, und wendet anschließend die Kurventransformation *trans* auf p an.

`pytree{C} col mode trans n` erzeugt einen aus Pentagonen der Form *pytrunk* (siehe Painter.hs) zusammengesetzten Kurvenbaum t und wertet diesen zu einer Kurve aus, die einen Pythagoreischen Baum der Tiefe n darstellt. Dabei wird die Kurventransformation *trans* auf jedes Pentagon von t angewendet. Zusätzlich färbt `pytreeC` jedes Pentagon – vor der Anwendung von *trans* – abhängig davon, auf welcher Ebene von t es sich jeweils befindet.

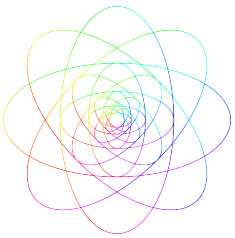
`base c` transformiert die Kurve c in eine *based curve*, falls das Zentrum von c oder der Startpunkt des ersten Kantenzuges von c mit dem Nullpunkt übereinstimmt. c wird so skaliert und ausgerichtet, dass die letzte Kante des ersten Kantenzuges von c mit $((0, 90), (0, 0))$ übereinstimmt, so dass c im Rahmen der Ausführung eines Aufrufs `evalCT t` (s.o.) an ihre Vorgänger-Kurve andocken kann.

pytrunk (s.o.) ist eine solche *based curve*. Painter.hs enthält weitere *based curves*, z.B. *hexa*, ein regelmäßiges Hexagon. Jede von ihnen kann iteriert und analog zu *pytree* (s.o) mit *buildCT* und *evalCT* zu (zirkulären) Bäumen zusammengesetzt werden, z.B. wie folgt:

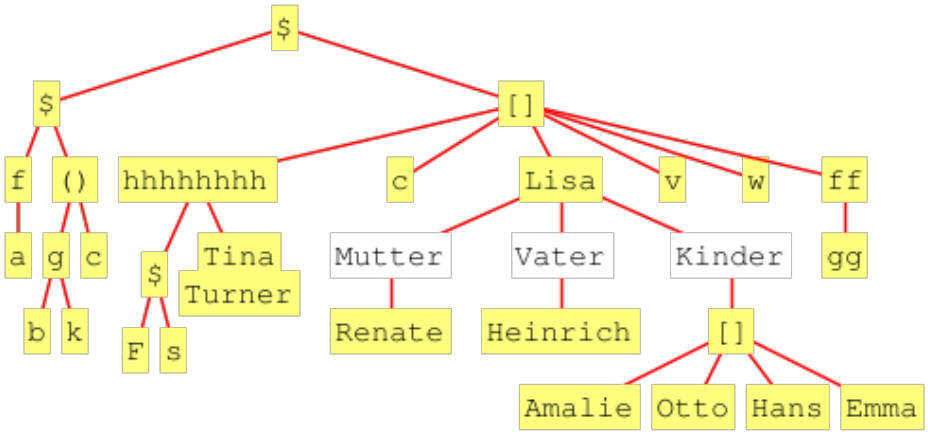
```
hexas c mode trans  = evalCTC trans . buildCT (hexa c mode) (replicate 6 True)
pytree c mode trans = evalCT . buildCT (trans $ pytrunk c mode) [False,True,True]
```

7 Beispiele Die Bäume werden mit drawTerm oder drawTermC gezeichnet, der Fahrplan mit drawGraph und alle anderen Figuren mit drawC. Weitere Beispiele finden sich im Abschnitt SAMPLE PICTURES von Painter.hs definiert. Das jeweils erzeugte Bild steht in einer svg-Datei des *Pix*-Verzeichnisses (s.o.).

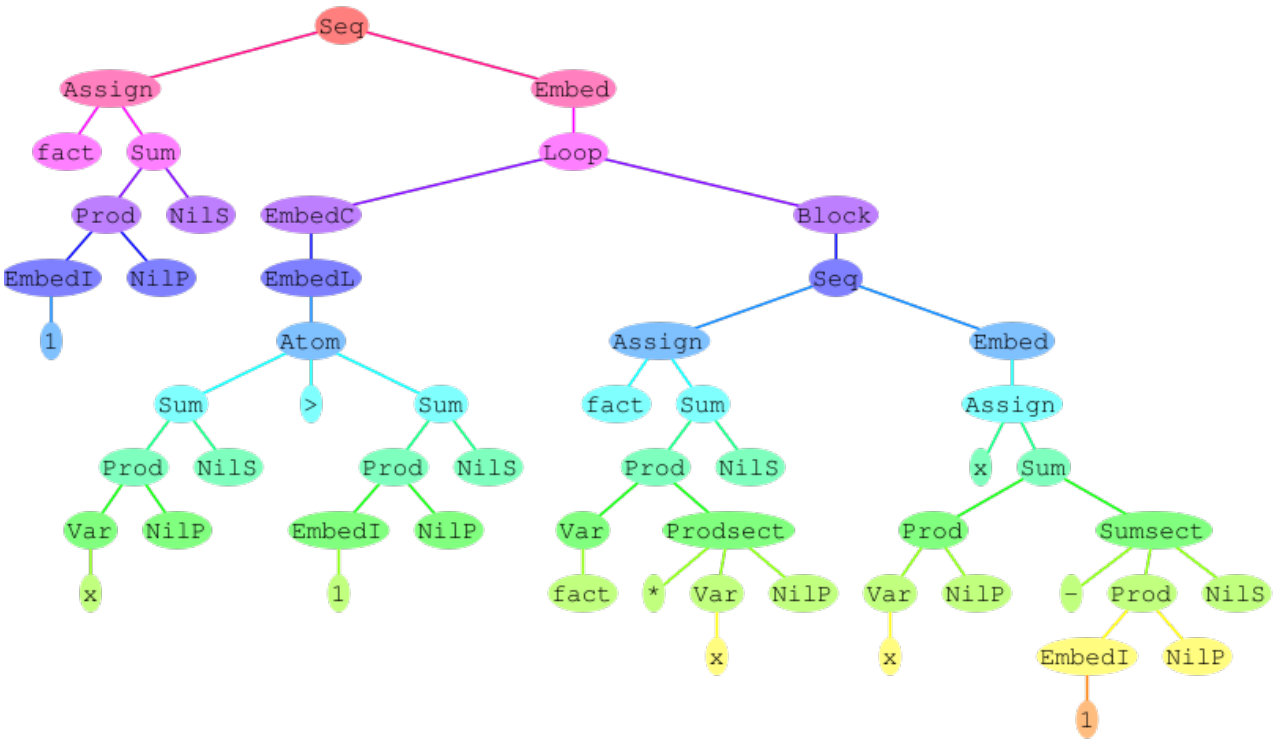
`piles red 31111 [3,1,0,5,4]`



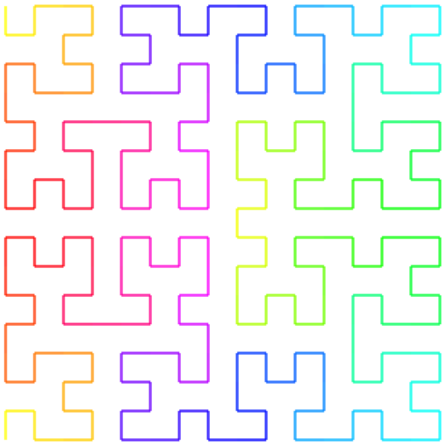
`orbits`



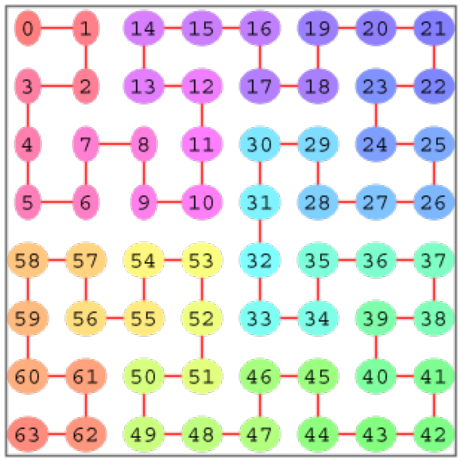
`Datei "lisa"` (Kanten sind zum Teil mit Attributen markiert.)



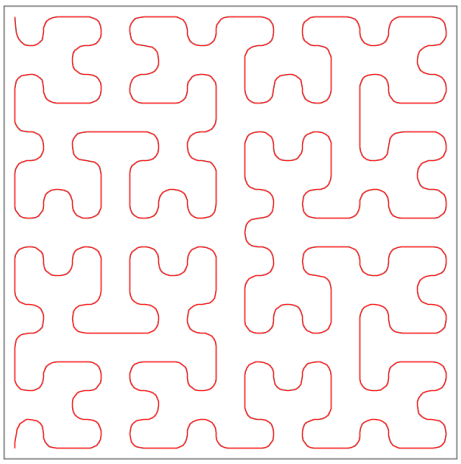
`Datei "javaterm"` (Syntaxbaum eines iterativen Programms für die Fakultätsfunktion)



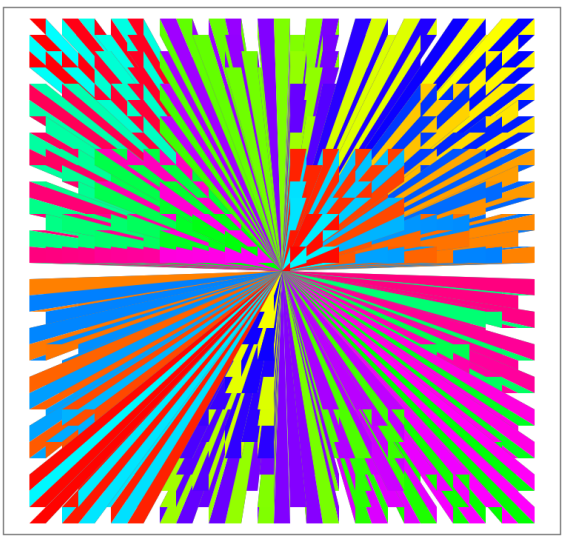
hilb yellow 12112 4



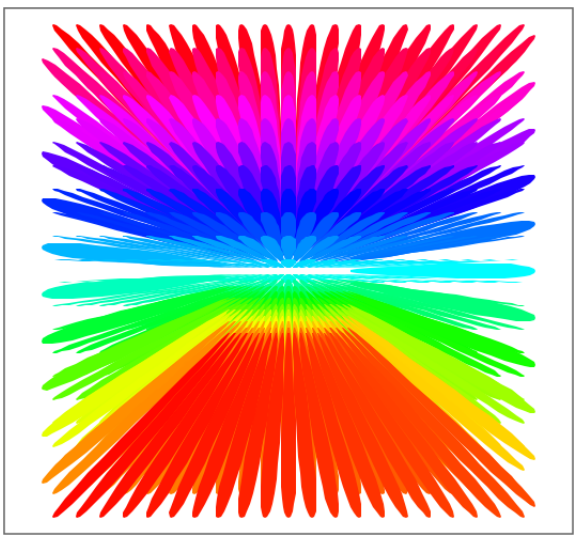
hilb red 42111 3



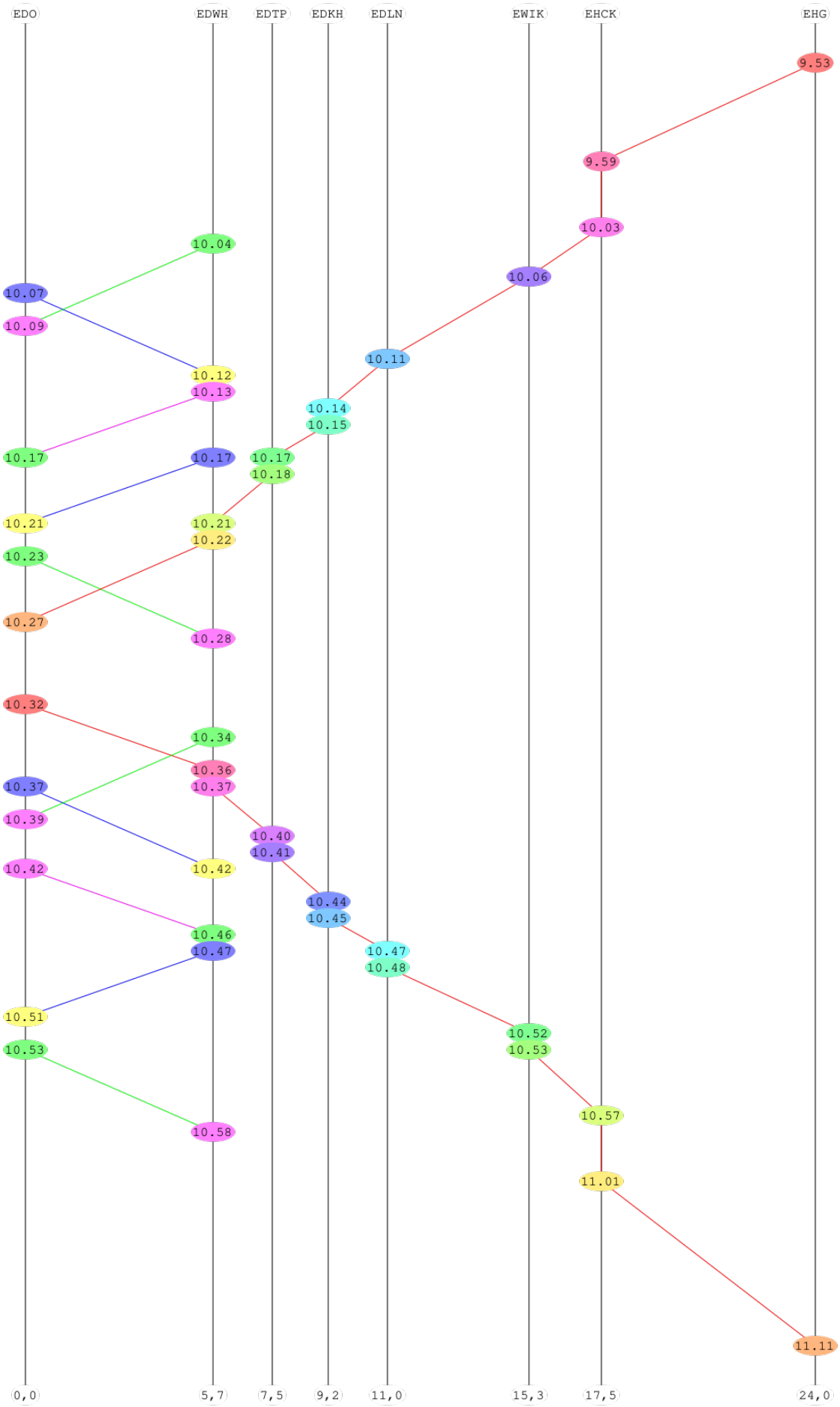
hilb 14211 4



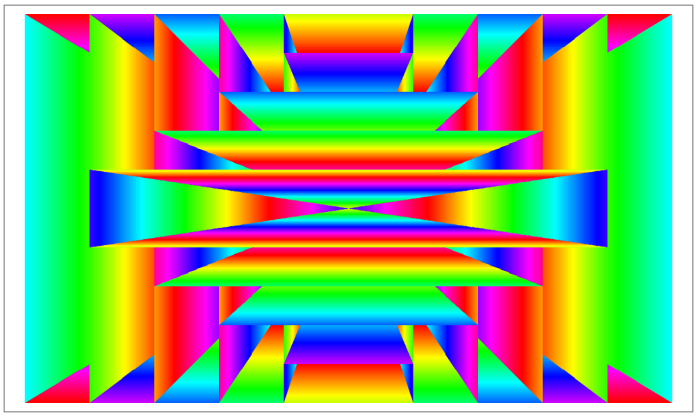
hilb red 13121 5

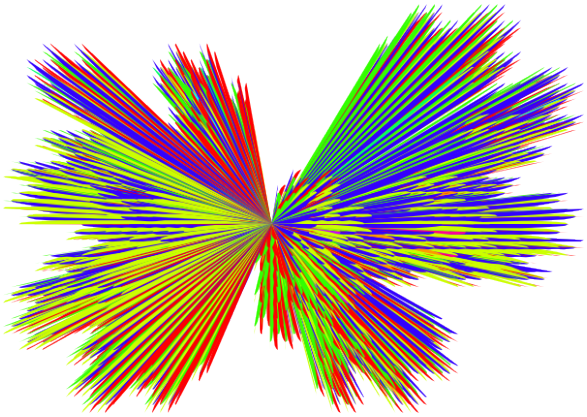


snake red 13211 22

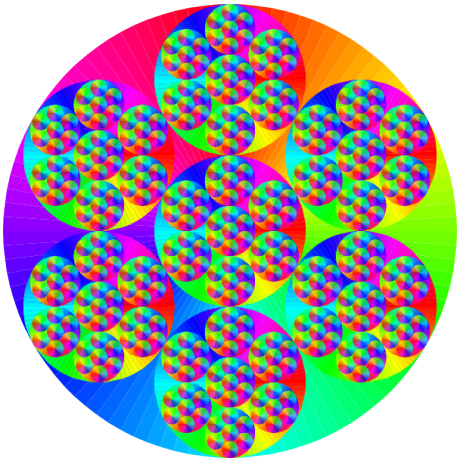

EDO
EDWH
EDTP
EDKH
EDLN
EWIK
EHCK
EHG
9.53
9.59
10.03
10.04
10.06
10.07
10.09
10.11
10.12
10.13
10.14
10.15
10.17
10.17
10.17
10.18
10.21
10.21
10.22
10.23
10.27
10.28
10.32
10.34
10.36
10.37
10.37
10.39
10.40
10.41
10.42
10.42
10.44
10.45
10.46
10.47
10.47
10.48
10.51
10.52
10.53
10.53
10.57
10.58
11.01
11.11
0,0
5,7
7,5
9,2
11,0
15,3
17,5
24,0

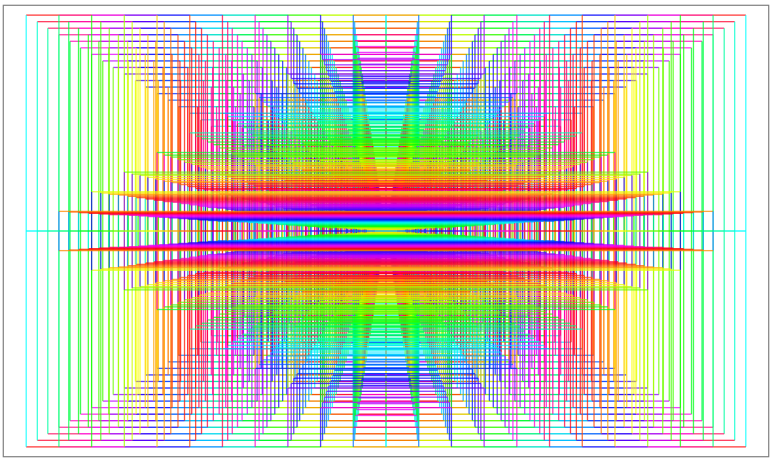
Datei "fahrplan"



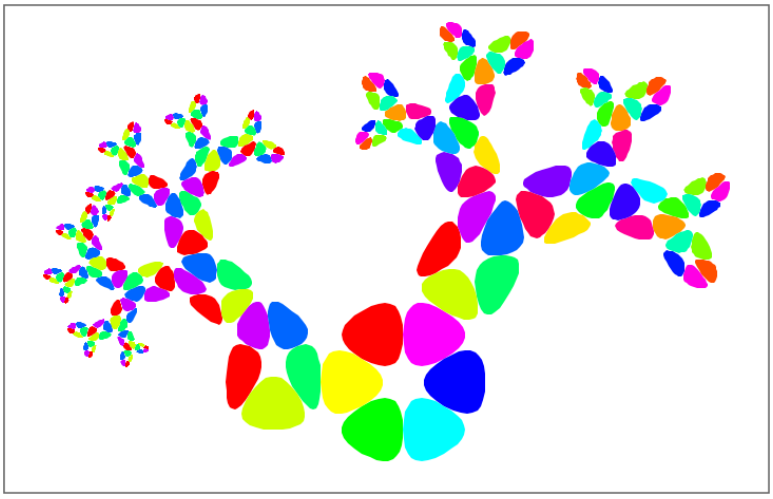


morphing 1 5 $ map (rainbow 1 33 . rect cyan 13114) [(100,60),(0,60),(100,0)]
dragonF red 13213 3333

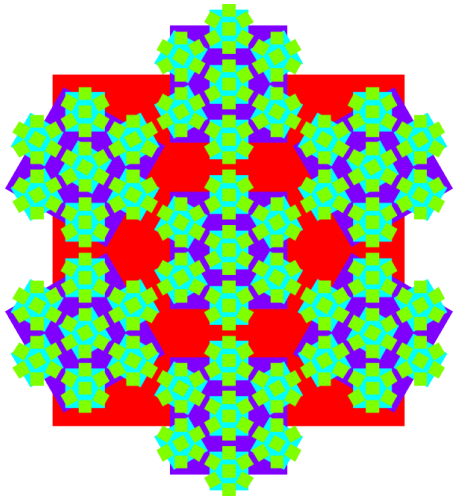




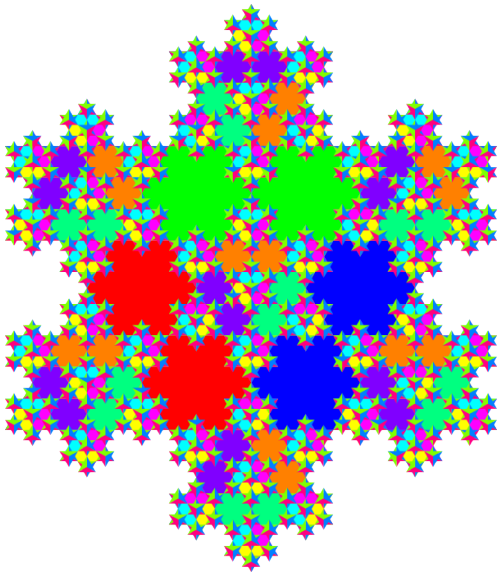
snow 1 4 1 4 6 $ circ red 13113 222
morphing 1 11 $ map (rainbow 1 33 . rect cyan 12114) [(100,60),(0,60),(100,0)]

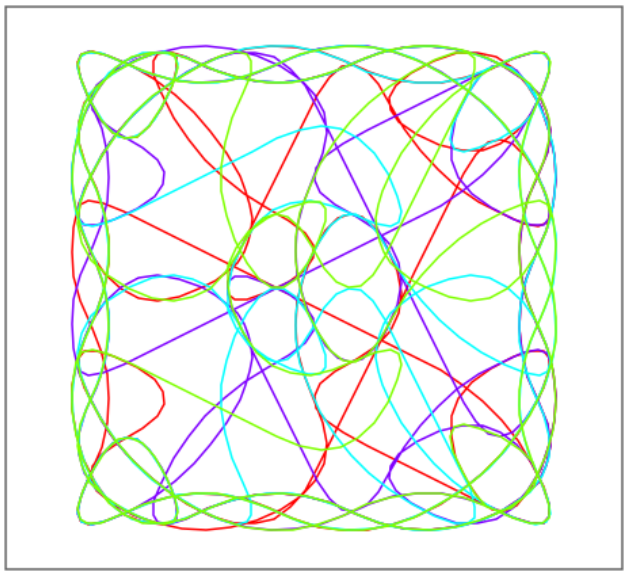


grow1 13211

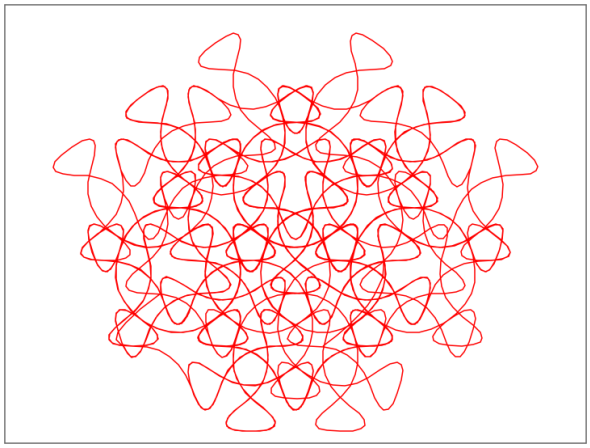


snow 1 6 1 4 6 $ rect red 15111 (222,222)

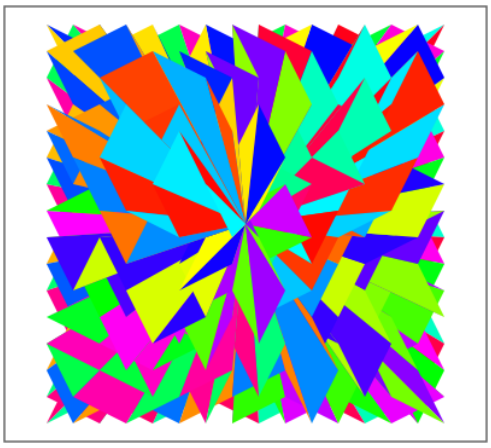




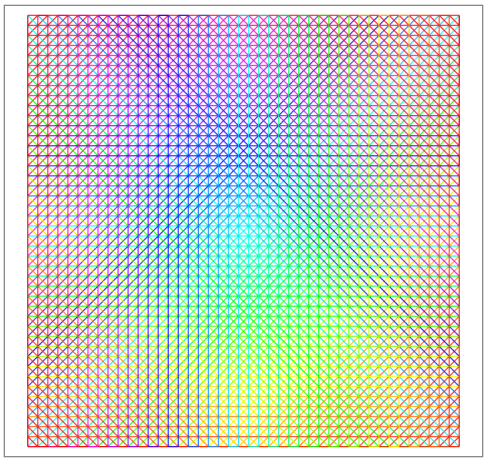
```
snow 1 6 1 4 6 $ combine $ map (tria red 13113) [222,-222]
hueCol 1 $ overlay $
concatMap ((\c -> [c,flipH c]) . (\n -> ktour red 14211 8 n n)) [1,8]
```

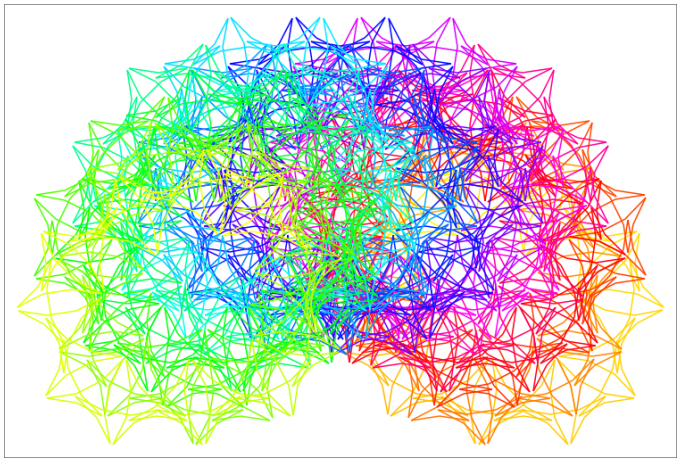


```
flipH $ turn 72 $ fibo red 14211 500 144 72
```

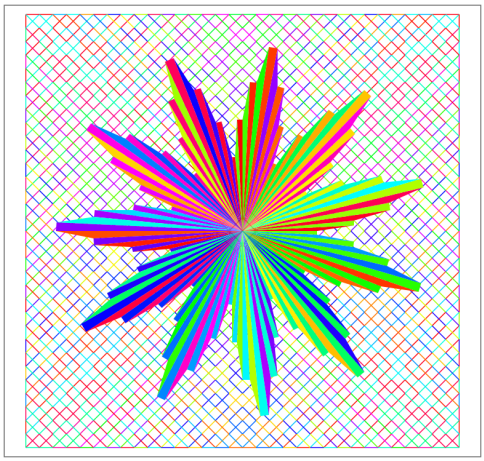


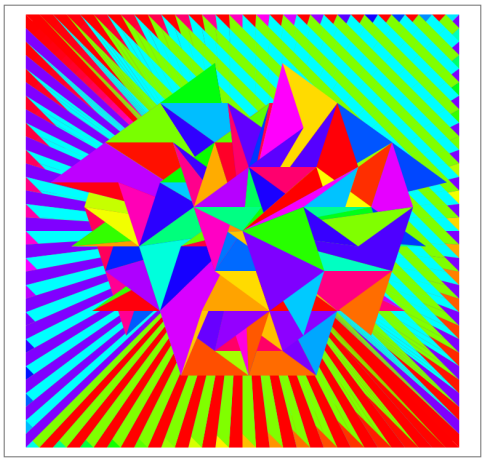
```
ktour red 13121 16 8 8
```



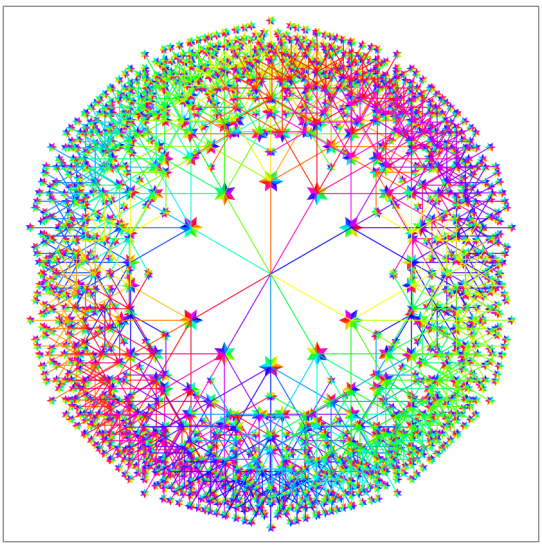


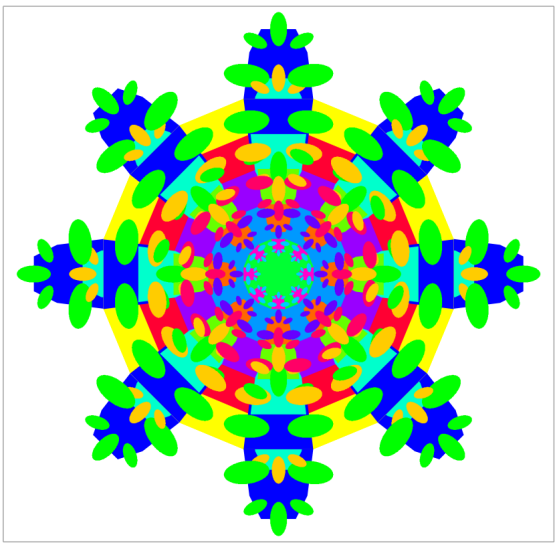
```
overlay [f cant,flipV $ f cant,f snake,transpose $ f snake] - f g = g red 12111 44
turn 225 $ fibo yellow 12111 4150 18 162
```



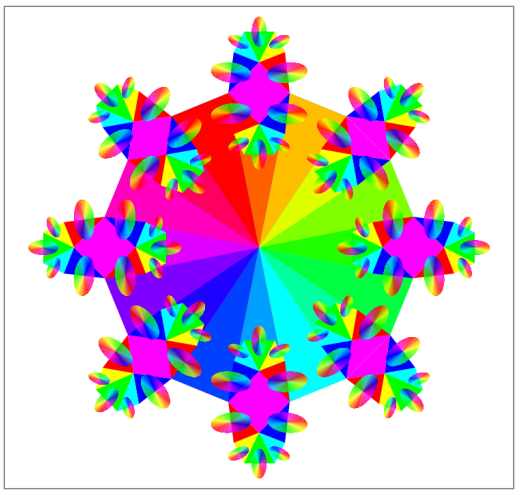


```
overlay [cant red 12121 33,flipV $ cant red 12121 33,scale 0.25 $ poly red 13123 11
                                $ map (11*) [2,2,3,3,4,4,5,5,4,4,3,3]]
        overlay [cant red 13111 33,flipV $ cant red 13113 33,flipH $ turn 72
                                         $ fibo red 13124 500 144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

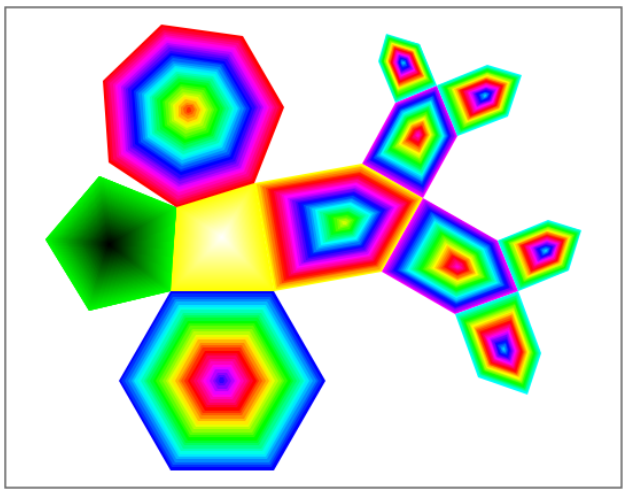




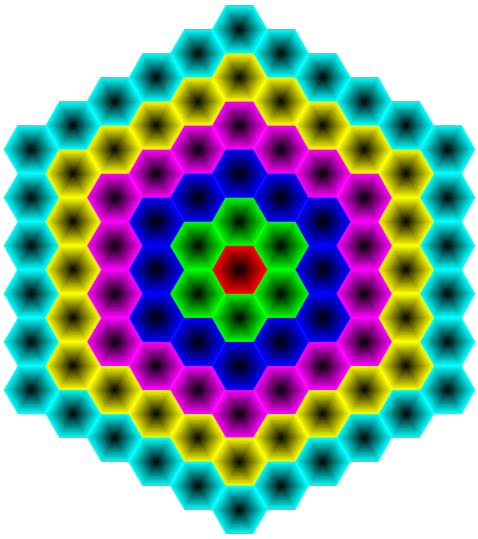
```
combine [f blue,flipH $ f yellow] where f col = hueCol 1 $ updCol col $ bunch 4
grow7 (rainbow 1 5) id id 15111
```



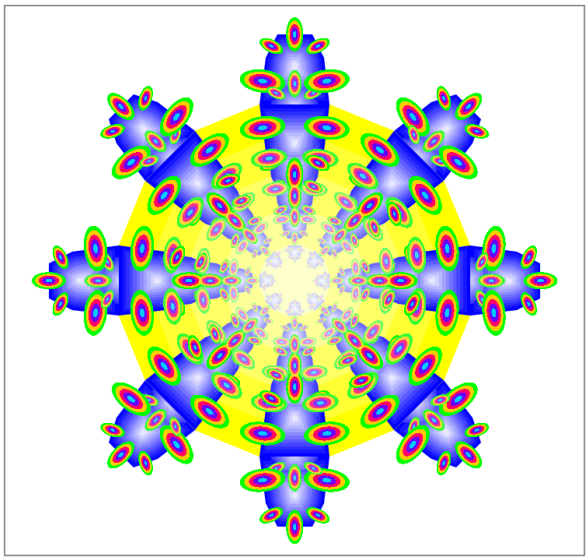
```
grow7 id id id 13111
```



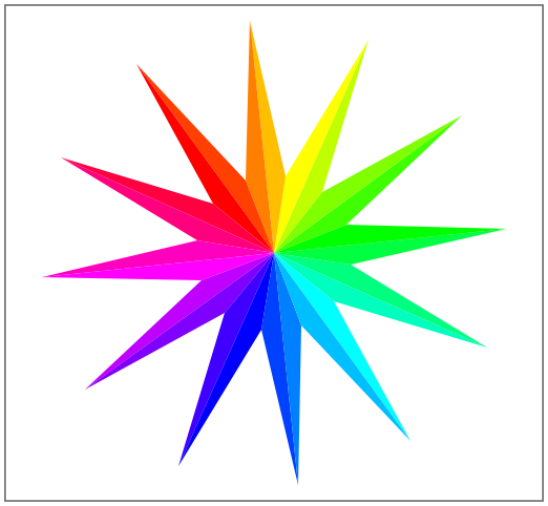
```
grow2 15111
```

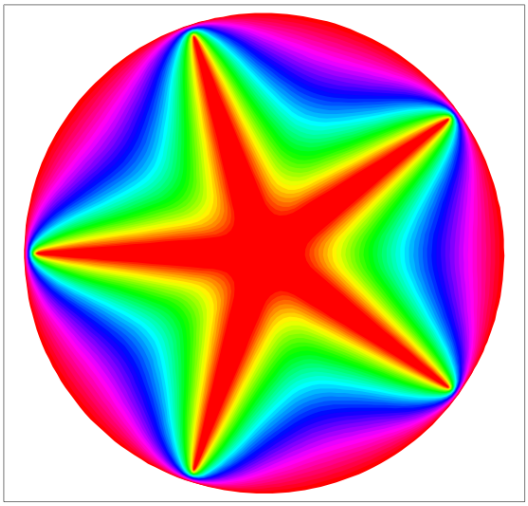


drawC hexas6d



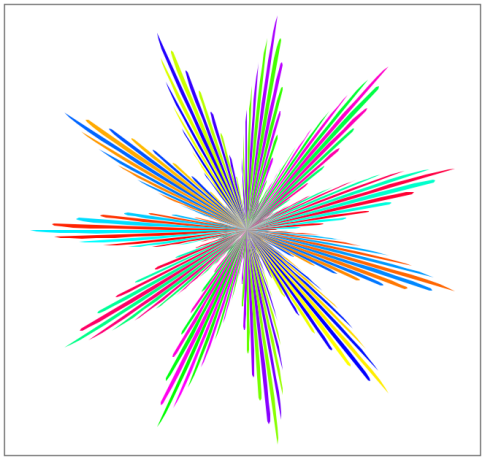
grow7 (shine 1 5) (shine 1 11) (rainbow 1 5) 15111

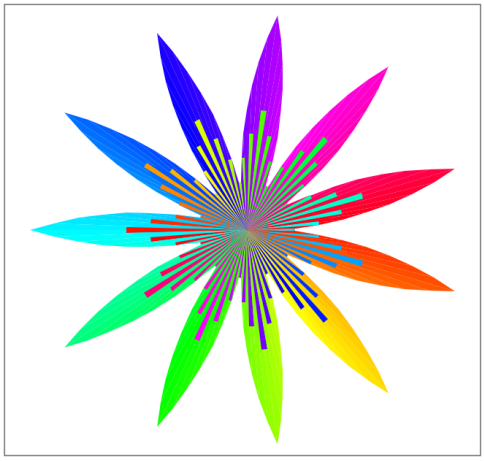




turn 54 $ poly cyan 13111 12 [111,37]

morphing 1 44 [poly red 15211 30 [40],poly red 15211 5 [5,40]]

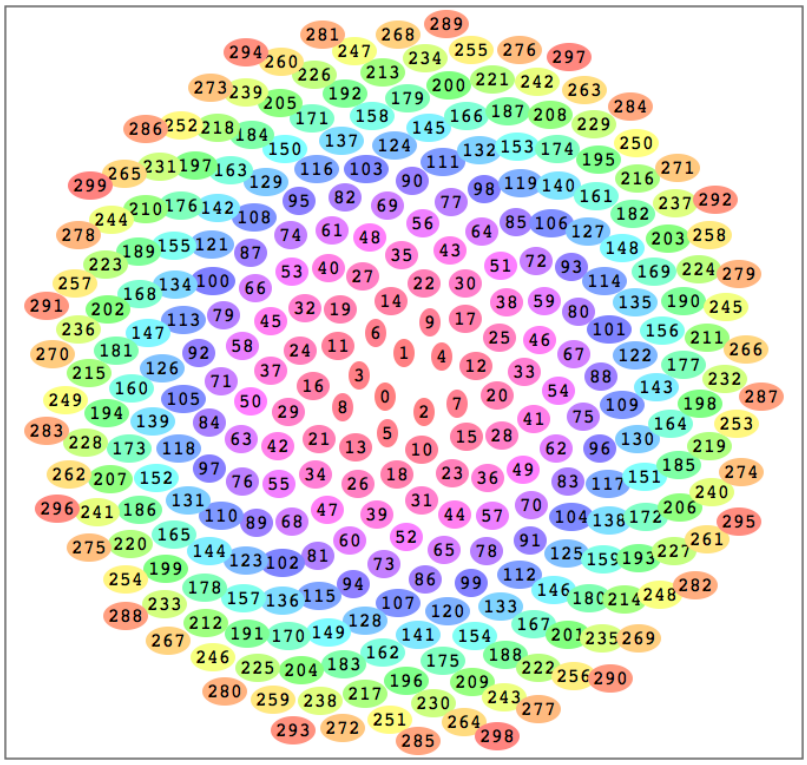




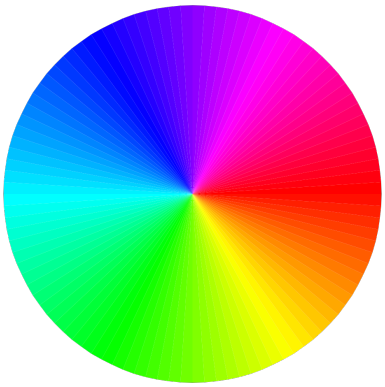
poly red 13221 11 rs

combine [poly red 13111 11 rs, turn (-1.5) $ poly 13121 11 rs']

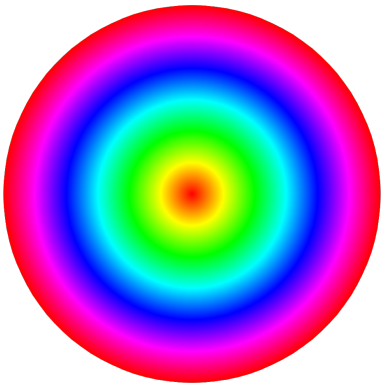
where rs = [1..9]++reverse [2..8]; rs' = [2,2,3,3,4,4,5,5,4,4,3,3]



phyllo red 41111 300

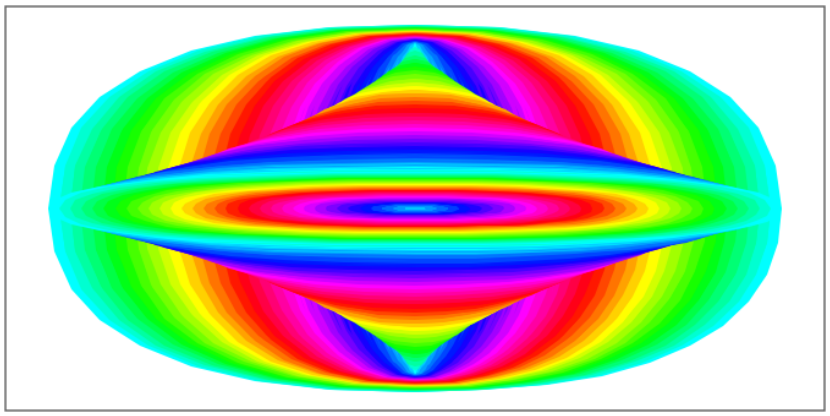


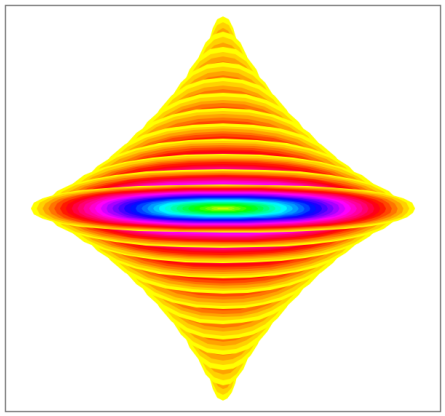
circ red 13111 1530



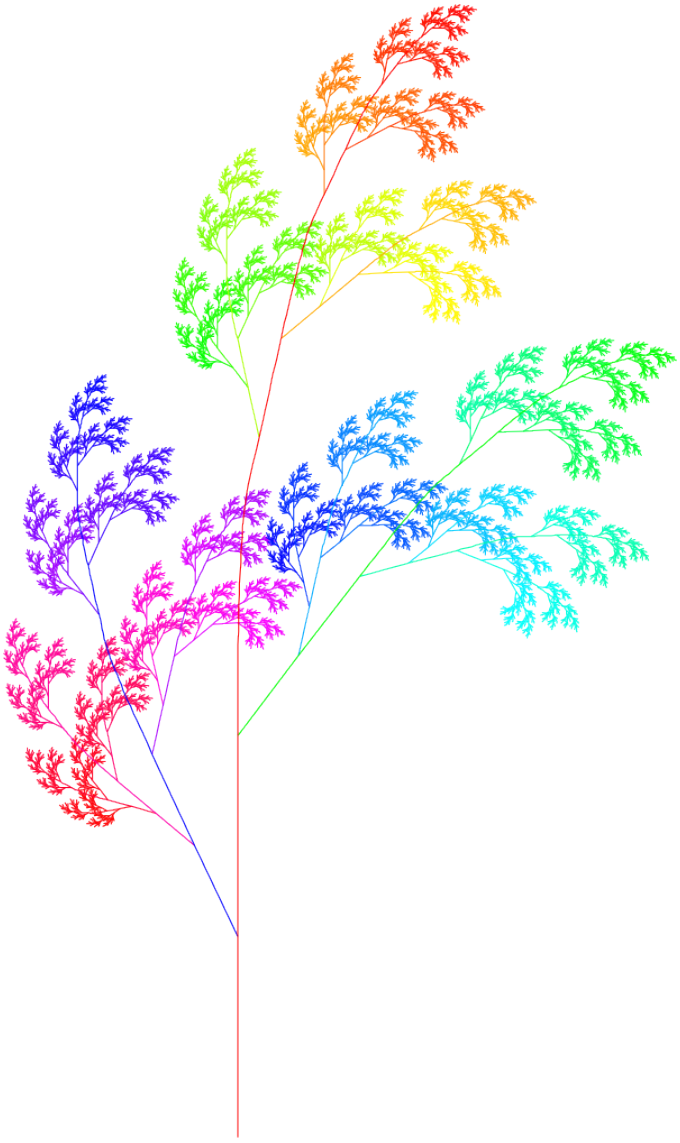
rainbow 1 1530 $ circ red 15111 1530

rainbow 1 33 $ circ red 13111 1530

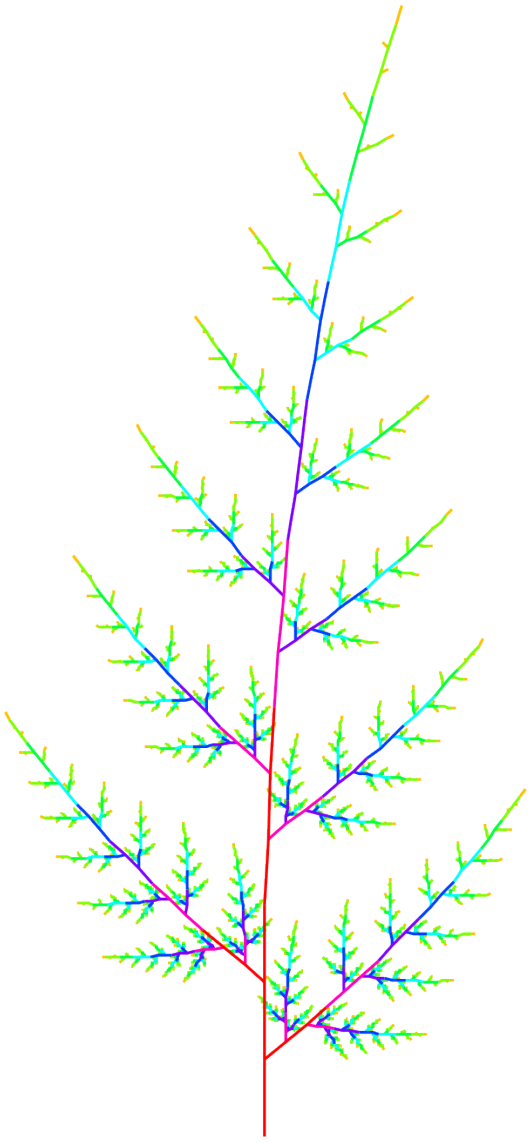




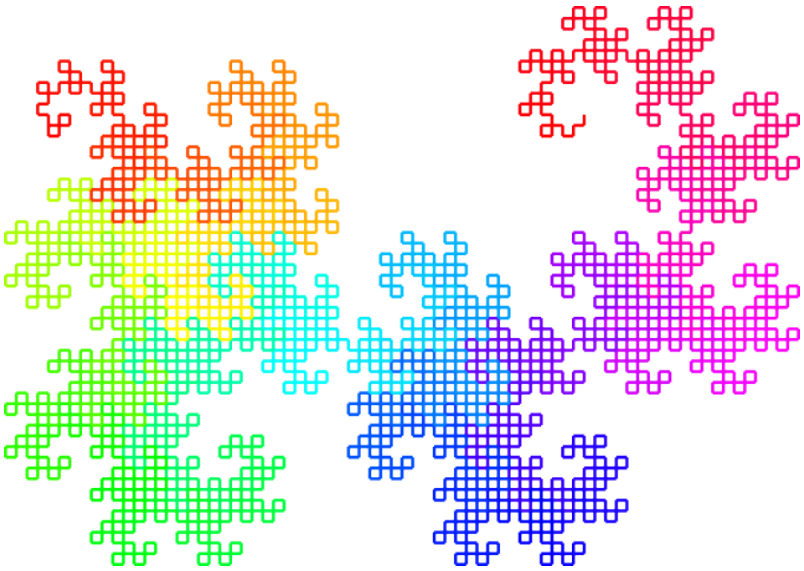
morphing 1 33 $ map (rainbow 1 33 . rect cyan 15211) [(100,50),(2,45),(100,10)]
morphing 4 11 $ map (rainbow 1 33 . rect yellow 15211) [(11,88),(88,11)]



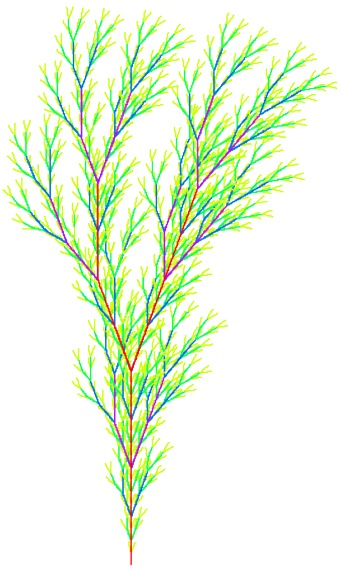
`hueCol 1 $ gras red 14111 8`



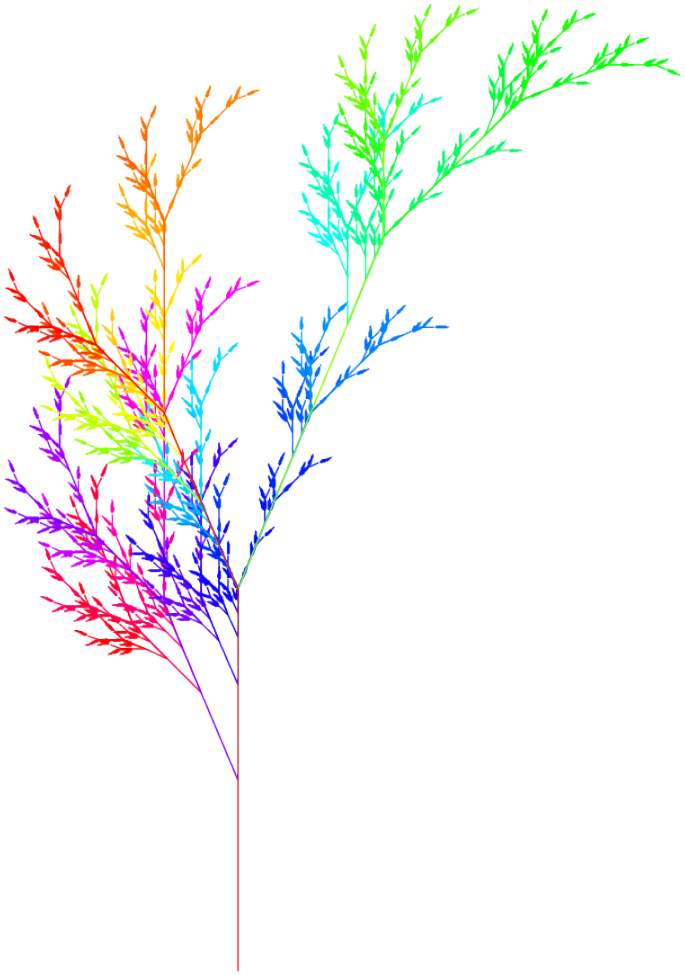
`fernL red 14111 8`



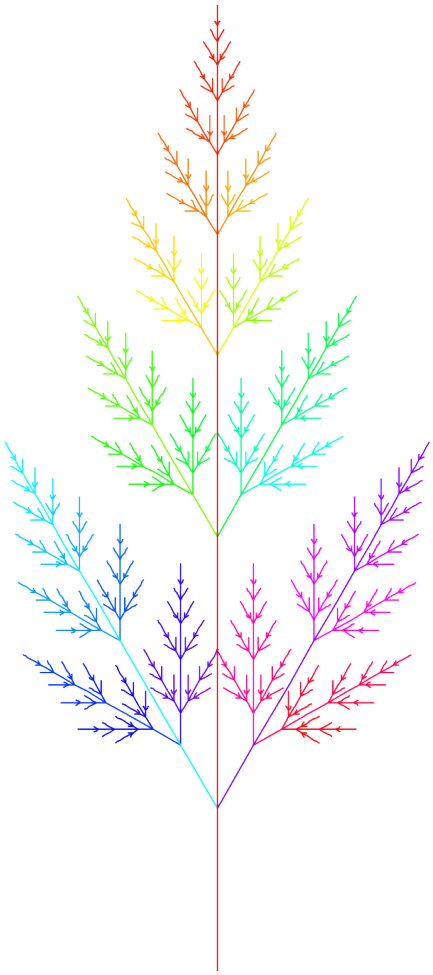
`dragonF red 12111 3333`



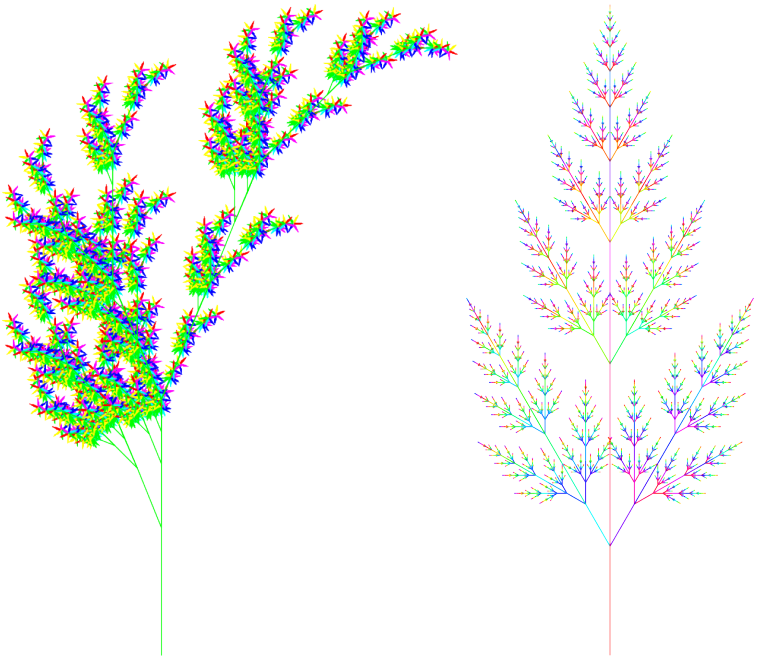
`bushL red 14111 5`



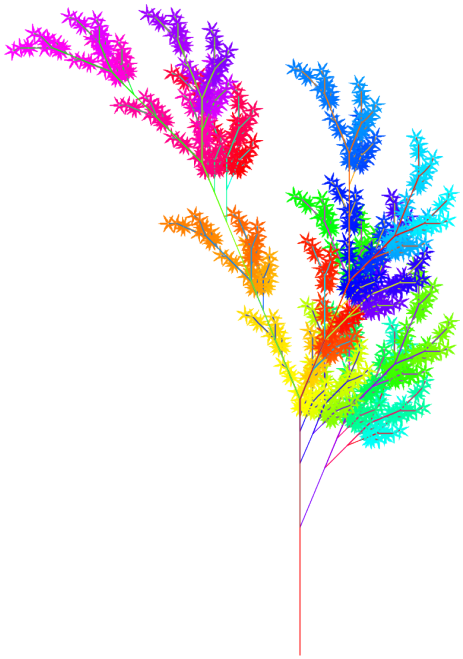
```
hueCol 1 $ gras red 14111 8
```



```
fernL red 14111 8
```



```
grasC green 14111 5 $ blosR red
```

```
hueCol 1 $ fernU red 12111 12
```

```
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)]]
```