Combinatorial Structures Package

The combstruct package is used to define, count and generate combinatorial structures.

A combinatorial class is defined by writing a grammar specification that describes it. In this way, an extensive collection of different classes may be defined. For example, the system applies to all regular and context-free grammars, grammars to define binary trees, plane general trees, necklaces, functional graphs, expression trees etc.

Given a grammar specification, the system creates a dedicated counting procedure for that structure. The number of objects of size n is counted in worst-case O(n^2) arithmetic operations. This bound is independent of the combinatorial class.

In the same way, an object of size n is generated uniformly at random in worst-case O(n log(n)) arithmetic operations. Again, this bound applies to all structures. O(n log(n)) is the best known bound for many combinatorial classes.

This worksheet explores through examples some of the uses of the combstruct package. A companion worksheet explains in detail how to write a grammar specification, the pre-defined structures available in combstruct, and the use of the various functions.

```
> with(combstruct):
```

General Trees

Combstruct can be used to analyze certain aspects of a combinatorial structure. By repeatedly generating random examples of our structure, we can get some idea of its average behaviour. For example, say we are interested in the average height of a general tree. We can use combstruct to model such trees and analyze the results.

First, we write a grammar specification that describes a tree. A tree consists of a root node, and zero or more children that are also trees.

```
> Tree := {T=Prod(Z, Set(T))}:
```

We can count and generate general trees.

```
> seq(count([T,Tree],size=i),i=1..20);
```

1, 1, 2, 4, 9, 20, 48, 115, 286, 719, 1842, 4766, 12486, 32973, 87811, 235381, 634847, 1721159, 4688676, 12826228

```
> draw([T,Tree],size=5);
```

$\mathbf{Prod}(Z, \mathbf{Set}(\mathbf{Prod}(Z, \mathbf{Set}(\mathbf{Prod}(Z, \mathrm{E}), \mathbf{Prod}(Z, \mathbf{Set}(\mathbf{Prod}(Z, \mathrm{E})))))))$

Now we write a small procedure that takes a tree and returns its height.

```
> height := proc(t)
  local h;
    if t=Prod(Z,Epsilon) then h := 1
    else h := 1+ max(op(map(procname, op(2,t))))
```

```
    fi;
    if length(t) < 100 then  # store small values in
  the remember table
      height(t) := h;
    fi;
    h;
  end:
> t := draw([T,Tree],size=5);
```

$$t := \mathbf{Prod}(\mathbf{Z}, \mathbf{Set}(\mathbf{Prod}(\mathbf{Z}, \mathrm{E}), \mathbf{Prod}(\mathbf{Z}, \mathrm{E}), \mathbf{Prod}(\mathbf{Z}, \mathrm{E}), \mathbf{Prod}(\mathbf{Z}, \mathrm{E})))$$

```
> height(t);
```

$$\mathbf{2}$$

Next we generate many trees of size 100, and for each tree we calculate its height.

```
> treesize := 100:
> numtrees := 200:
> heights := array(sparse, 1..treesize):
> to numtrees do
    h := height(draw([T,Tree],size=treesize)):
    heights[h] := heights[h] + 1:
  od:
```

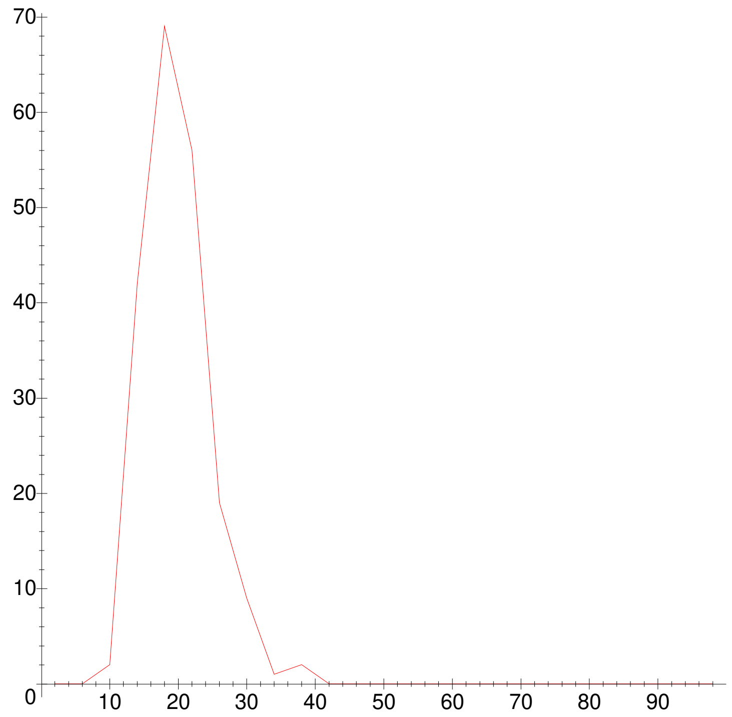
In order to get a reasonably smooth curve, we will group the heights into intervals of 4.

```
> groupedheights := array(sparse,1..treesize/4):
> for i to treesize do
     groupedheights[iquo(i-1,4)+1] :=
  groupedheights[iquo(i-1,4)+1]+heights[i]od:
```

Now we plot the distribution of heights of our randomly generated trees.

```
> plot([seq([i*4-2, groupedheights[i]],
  i=1..treesize/4)],
   title=`distribution of heights of trees of size
  100`);
```

distribution of heights of trees of size 100



Connected Functional Graphs

Every function that takes distinct elements {1,...,n} into itself has a corresponding graph. Points on the graph are either cyclic, meaning that some iterate of the function takes that element back onto itself, or non-cyclic. The set of points that go to the same cyclic point is a general tree. A connected functional graph is a functional graph which has one component. Such a graph corresponds to a function which, for any two elements, iterating the function on the first element a certain number of times will reach the same result as iterating the function on the second element a (possibly different) number of times. In other "words",
a function f such that, for every x and y in {1,...n}, there exists a pair of non-negative integers (i,j) such that f^(i)(x) = f^(j)(y).

We can count the number of all such functions on n elements by counting the number of connected functional graphs.

```
> fungraph := {F=Cycle(tree), tree=Prod(Z,Set(tree))}:
> count([F, fungraph, labelled], size=40);
```

22997133066773554208839185040242998852045747354953842688000000
000

```
> seq(count([F, fungraph, labelled], size=i),
  i=1..10);
```

1, 3, 17, 142, 1569, 21576, 355081, 6805296, 148869153, 3660215680

We can also generate examples of functions with this property.

```
> draw([F, fungraph, labelled], size=11);
```

Cycle(Prod(Z_8, Set(Prod(Z_4, Set(

$\mathrm{Prod}(Z_5, \mathrm{Set}(\mathrm{Prod}(Z_{10}, \mathrm{Set}(\mathrm{Prod}(Z_1, \mathrm{E}), \mathrm{Prod}(Z_7, \mathrm{Set}(\mathrm{Prod}(Z_3, \mathrm{E})))))))$

$, \mathrm{Prod}(Z_6, \mathrm{Set}(\mathrm{Prod}(Z_9, \mathrm{Set}(\mathrm{Prod}(Z_2, \mathrm{Set}(\mathrm{Prod}(Z_{11}, \mathrm{E}))))))))))$

Alcohols

We can model alcohol molecules, C_n H_{2n+1} OH, using combstruct. We write a grammar for an alkyl, C_n H_{2n+1}, which is a ternary rooted tree that moves freely in space.

```
> molecule := {alkyl = Union(H, Prod(C, Set(alkyl,
  card=3))), H=Atom, C=Atom}:
> count([alkyl, molecule], size=6+2*6+1);
```

$$17$$

Thus we see that there are 17 different alcohol molecules with 6 carbon atoms. Here is one of them.

```
> draw([alkyl, molecule], size=6+6*2+1);
```

$\mathrm{Prod}(C, \mathrm{Set}(H, H, \mathrm{Prod}(C, \mathrm{Set}($
$\mathrm{Prod}(C, \mathrm{Set}(H, H, \mathrm{Prod}(C, \mathrm{Set}(H, H, H)))), \mathrm{Prod}(C, \mathrm{Set}(H, H, H)),$
$\mathrm{Prod}(C, \mathrm{Set}(H, H, H))))))$

Necklaces

A necklace is a classical structure from combinatorics. It is a cycle containing beads of different colours. Necklaces are particularly difficult to count because of the symmetries involved, since beads of the same colour are indistinguishable, and because knowing how many necklaces there are of size n does not help us determine how many necklaces there are of size n+1, since the symmetries change for each size depending on how that integer factors.

We define a necklace that has three different colours of beads.

```
> necklace := {N=Cycle(bead),
  bead=Union(red,blue,green),red=Atom,blue=Atom,green=
  Atom}:
> seq(count([N, necklace], size=i), i=1..15);
```

3, 6, 11, 24, 51, 130, 315, 834, 2195, 5934, 16107, 44368, 122643, 341802, 956635

```
> draw([N, necklace], size=10);
```

$$\mathrm{Cycle}(red, red, green, green, blue, red, green, blue, green, green)$$

We can also create necklaces of necklaces.

```
> neckneck := {NN=Cycle(Cycle(bead)),
  bead=Union(red,blue,green), red=Atom,
  blue=Atom,green=Atom}:
> count([NN,neckneck], size=80);
```

$$1018458770206928194077562044012744905194163840242675245$$

```
> draw([NN,neckneck], size=15);
```

$$\mathrm{Cycle}(\mathrm{Cycle}(\mathit{green}, \mathit{green}), \mathrm{Cycle}(\mathit{blue}, \mathit{blue}, \mathit{green}), \mathrm{Cycle}(\mathit{green}), \mathrm{Cycle}(\mathit{blue}, \mathit{blue}, \mathit{red}), \mathrm{Cycle}(\mathit{blue}, \mathit{blue}, \mathit{blue}, \mathit{red}), \mathrm{Cycle}(\mathit{blue}), \mathrm{Cycle}(\mathit{blue}))$$

Involutions

An involution is a permutation of elements that is its own inverse. Such a permutation must decompose into cycles of length one or two. We can count the involutions on n distinct elements.

```
> involution := {inv = Set(Cycle(Z, card <= 2))}:
> count([inv, involution,labelled], size=12);
```

$$140152$$

We can compare this with the total number of permutations on 12 elements. (Here, we use one of the pre-defined structures of combstruct.)

```
> count(Permutation(12));
```

$$479001600$$

Here is one of the involutions on 12 elements.

```
> draw([inv, involution,labelled], size=12);
```

$$\mathrm{Set}(\mathrm{Cycle}(Z_5, Z_9), \mathrm{Cycle}(Z_{10}), \mathrm{Cycle}(Z_4), \mathrm{Cycle}(Z_3, Z_{11}), \mathrm{Cycle}(Z_1, Z_7), \mathrm{Cycle}(Z_6, Z_8), \mathrm{Cycle}(Z_2, Z_{12}))$$

Generating Test Data

Another use of combstruct is to generate tests for another program (not limited to other Maple programs).

The project CROAP at INRIA, France, is developing a programming environment generator called Centaur. Given the grammar and "pretty-printer" of a computer language, Centaur produces an editor that knows about the syntax and appearance of that language. The researchers of CROAP are using combstruct in order to produce test cases for the resulting editor. A specification for the computer language is written in combstruct, and the "programs" that combstruct produces are fed to the editor. Here is one of the specifications (which they generate automatically from the syntax of the language) that is used to create tests for Centaur. (Thanks to Laurence Rideau, <Laurence.Rideau@sophia.inria.fr>, for explaining her application of combstruct.)

```
> Exp_spec := {EXP_S =
  Union(Prod(`Exp$exp_s`,Sequence(EXP,
                        card >=1))),
     EXP = Union(Prod(`Exp$assign`,VAR, EXP),
                       `Exp$integer`,
```

```
                Prod(`Exp$minus`,EXP, EXP),
                Prod(`Exp$plus`,EXP, EXP),
                Prod(`Exp$prod`,EXP, EXP),
                Prod(`Exp$uminus`,EXP),
                `Exp$variable`),
    VAR = Union(`Exp$variable`),
    INTEGER = Union(`Exp$integer`),
    ENV = Union(Prod(`Exp$env`,Sequence(PAIR))),
    PAIR = Union(Prod(`Exp$pair`,VAR, INTEGER)),
    COMMENT = Union(`Exp$comment`),
    COMMENT_S = Union(Prod(`Exp$comment_s`,Sequence(
                      COMMENT))),
    `Exp$assign`=Atom, `Exp$comment`=Atom,
    `Exp$comment_s`=Atom, `Exp$env`=Atom,
    `Exp$exp_s`=Atom, `Exp$integer`=Atom,
    `Exp$meta`=Atom, `Exp$minus`=Atom,
    `Exp$pair`=Atom, `Exp$plus`=Atom,
    `Exp$prod`=Atom, `Exp$uminus`=Atom,
    `Exp$variable`=Atom}:
```

The test programs are then generated using the draw function.

```
> draw([EXP_S, Exp_spec], size=17);
```

Prod(*Exp$exp_s*, Sequence(Prod(*Exp$uminus*, Prod(*Exp$uminus*, Prod(*Exp$minus*, Prod(*Exp$minus*, *Exp$variable*, Prod(*Exp$prod*, Prod(*Exp$uminus*, *Exp$integer*), *Exp$variable*)), *Exp$integer*))), *Exp$integer*, *Exp$integer*, *Exp$variable*, Prod(*Exp$minus*, *Exp$integer*, *Exp$variable*)))

```
>
```