Introduction to the Combinatorial Structures Package

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

With combstruct, you may define a combinatorial class of objects, generate uniformly at random objects belonging to that class, and count the number of objects of a given size. For certain common structures which have been pre-defined in the system, it is also possible to create an iterator which will traverse all the objects in the given class, and to list all such objects at once.

There are two main portions to combstruct - the facility to define your own structures using a grammar, and using the pre-defined structures provided by the system.

Note: The pre-defined structures portion of combstruct provides an extension of the functions in the combinat package involving combinations, permutations, partitions, compositions and subsets. The combstruct functions do everything these functions do, and more (in some cases, much more), as well as providing a consistent interface in terms of function names and syntax.

This worksheet is a fairly detailed explanation of how to use combstruct. A companion worksheet describes some of the uses of the package. The help pages give a shorter overview. In particular, refer to the help for combstruct, combstruct[specification] and combstruct[structures].

First, define the combstruct package functions.

```
> with(combstruct);
```

$$[allstructs, count, draw, finished, gfeqns, gfseries, gfsolve, iterstructs, nextstruct, prog\_gfeqns, prog\_gfseries, prog\_gfsolve]$$

Grammar Specifications

The strength of this system is in the ability to define your own 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. All grammar specifications may be labelled or unlabelled.

Grammar specifications are written using Atom, Epsilon, and the constructors Union, Prod, Set (multi-set: repetition of elements is allowed), Sequence, Cycle and Subst (substitution).

Once you have a grammar specification, you may draw objects of a given size, uniformly at random, or count the number of objects of that size.

Union and Prod

For a very simple example, let's look at binary trees. We define a specification for a binary tree using the constructors for unions and product as follows:

```
> sys := {B = Union(Z, Prod(B,B))};
```

$$sys := \{B = \mathbf{Union}(Z, \mathbf{Prod}(B, B))\}$$

A binary tree is either a single leaf, Z, for the tree of size 1, or (the Union gives the 'or') it is made up of two (smaller) binary trees joined together (by Prod) at the root.

We can generate binary trees of size 5 uniformly at random using the function draw.

```
> draw([B, sys], size=5);
```

$$\mathbf{Prod}(\mathbf{Prod}(\mathbf{Prod}(Z, \mathbf{Prod}(Z, Z)), Z), Z)$$

```
> draw([B, sys], size=5);
```

$$\mathbf{Prod}(\mathbf{Prod}(Z, Z), \mathbf{Prod}(Z, \mathbf{Prod}(Z, Z)))$$

Labelled and Unlabelled

By default, structures are unlabelled, so atoms are not distinguishable.
Any specification can also be treated as labelled.

The same specification can be used to generate labelled binary trees.

```
> count([B, sys, labelled], size=5);
```

$$1680$$

```
> draw([B, sys, labelled], size=5);
```

$$\mathbf{Prod}(\mathbf{Prod}(\mathbf{Prod}(Z_4, Z_2), Z_5), \mathbf{Prod}(Z_3, Z_1))$$

```
> count([B, sys, unlabelled], size=5);
```

$$14$$

Atom and Epsilon

Z is an atom that is built into the system. Other atoms can be defined in a grammar specification. For example, we can define a necklace that has beads of three different colours.

```
> necklace :=
  {N=Cycle(Union(red,blue,green)),red=Atom,blue=Atom,green=Atom
  }:
> draw([N,necklace,unlabelled], size=10);
```

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

All atoms have weight one. The object Epsilon has weight 0. Our previous definition of a binary tree did not allow for the empty tree. This new definition does.

```
> tree := {T = Union(Epsilon, B), B=Union(Z, Prod(Z,Z))}:
> draw([T, tree, unlabelled], size=0);
```

$$\mathrm{E}$$

(That's not the letter "E", that's how Maple prints a capital Epsilon.) Like atoms, you can give different names to Epsilon. This is particularly useful to tag smaller components of an overall structure. For example, to model series and parallel circuits of resistors, a parallel circuit is made up of two or more resistors in series, and a series circuit is made up of two or more parallel circuits.

```
> circuit :={C=Union(P,S,R), P=Set(Union(S,R),card>=2),
  S=Set(Union(P,R),card>=2), R=Atom}:
> count([C,circuit,labelled], size=5);
```

$$472$$

```
> draw([C,circuit,labelled], size=5);
```

$$\mathrm{Set}(\mathrm{Set}(R_3, R_5), \mathrm{Set}(R_4, \mathrm{Set}(R_2, R_1)))$$

Each set contains (sub) circuits made up of one or more resistors. From this result, we cannot know which sub-circuits are joined in parallel and which are joined in series, because of the symmetry in their definitions. So we rewrite the grammar, adding an Epsilon tag to each portion of the specification.

```
> circuit2 :={C=Union(P,S,R),
  P=Prod(par,Set(Union(S,R),card>=2)),
  S=Prod(ser,Set(Union(P,R),card>=2)), R=Atom,
  par=Epsilon,ser=Epsilon}:
> count([C,circuit2,labelled],size=5);
```

$$472$$

```
> draw([C,circuit2,labelled],size=5);
```

$$\mathrm{Prod}(par, \mathrm{Set}(R_3, \mathrm{Prod}(ser, \mathrm{Set}(R_4, \mathrm{Prod}(par, \mathrm{Set}(R_2, \mathrm{Prod}(ser, \mathrm{Set}(R_5, R_1))))))))$$

Members of the sets that are associated with the tag "par" via a product are joined together in parallel, and those marked with the tag "ser" are joined together in series. Because Epsilon has weight 0, we have not changed the number of objects of each size.

Set, Cycle and Sequence

As shown in the above example, you may specifiy the cardinality of a set (here, we insisted that all circuits have at least two resistors). You may also specify the cardinality of cycles and sequences.

```
> count([M, {M=Set(Z, card > 8)}, labelled], size=7);
```

$$0$$

```
> draw([A, {A=Cycle(Z, card=4)},labelled],size=4);
```

$$\mathrm{Cycle}(Z_1, Z_4, Z_2, Z_3)$$

```
> count([A, {A=Cycle(Z, card=4)},labelled],size=3);
```

$$0$$

```
> draw([S, {S=Sequence(Set(Z, card>0), card <=10)}, labelled],
  size=6);
```

$$\mathrm{Sequence}(\mathrm{Set}(Z_2, Z_4), \mathrm{Set}(Z_1), \mathrm{Set}(Z_3), \mathrm{Set}(Z_6, Z_5))$$

```
> count([S, {S=Sequence(Z, card <=10)}, labelled], size=13);
```

$$0$$

```
> draw([S, {S=Sequence(Z, card <=10)}, labelled], size=13);
Error, (in combstruct/drawgrammar) there is no structure of this size
```

Subst

Subst is a mechanism that allows the substition of all the atoms of one object by another object. Subst(A,B) means take a B-object, and for every atom in that object, replace the atom with an A-object. We can use Subst to write a recursive definition of 2-3 trees. A 2-3 tree is a tree that has all its leaves at the same level, and every internal vertex has either two or three children. We define our tree by saying that it is either a single leaf, or it is a 2-3 tree with each of the leaves replaced by an internal vertex with 2 children or an internal vertex with 3 children.

```
> t23 := {T = Union(Z, Subst(Union(Prod(Z,Z), Prod(Z,Z,Z)), T))
  }:
> draw([T, t23], size=11);
```

$$\mathbf{Prod}(\mathbf{Prod}(\mathbf{Prod}(Z, Z), \mathbf{Prod}(Z, Z, Z)), \mathbf{Prod}(\mathbf{Prod}(Z, Z, Z), \mathbf{Prod}(Z, Z, Z)))$$

Manipulating Output

The object returned by draw is a maple object which can be manipulated with other commands. This is particularly useful to change the result from its very general form to one more suited to the particular problem.

For example, say we wish to generate words of the form C = (a C b) *

```
> sys := { C=Sequence( Prod(a, C, b)), a=Atom, b=Atom}:
> draw([C,sys], size=6);
```

$$\mathbf{Sequence}(\mathbf{Prod}(a, \mathbf{Sequence}(\mathbf{Prod}(a, \mathrm{E}, b), \mathbf{Prod}(a, \mathrm{E}, b)), b))$$

Here, Epsilon means that the empty sequence was chosen. All we really want is a string of a's and b's. The draw command gave more information about the derivation structure than is needed in this case. So we then do the following.

```
> eval(subs(Prod=( ()->args ), Sequence=( ()->args ),
  Epsilon=NULL, "));
```

$$a, a, b, a, b, b$$

Pre-Defined Structures

Certain combinatorial structures are common enough that they merit specialized algorithms rather than using the very general methods obtained with grammars. The structures that are pre-defined in the combstruct system are:

Combination (also called Subset) - combinations of elements
Permutation - permutations of elements
Partition - integer partitions (split into summands, order does not matter)
Composition - integer compositions (split into summands, order matters)

The functions that understand these structures are:
draw

count
allstructs
iterstructs

Like structures defined by a grammar, you may draw and count them.

```
> draw(Combination({a,b,c,d,e,f,g}), size=5);
```

$$\{a, f, d, e, g\}$$

```
> count(Partition(95), size=40);
```

$$450768$$

All the functions that manipulate these structures have the same syntax. As in manipulating structures defined by grammars, the first argument to the function is a definition of the structure, and the second is the size. Here, the structure is defined by Structure(structure arguments), where the structure arguments depend on the structure. Thus Partition and Composition require integers, whereas Combination and Permutation will accept lists, sets or integers (in which case the integer n is treated to mean {1,...,n} for Combination and [1,...,n] for Permutation).

Each structure has a default size associated with it, so when no size is specified, the function will make the most natural assumption for the structure. Also, instead of giving an integer as a size, the string 'allsizes' may be specified, saying that the structure should be chosen from all possible sizes of the object defined. ('allsizes' is the default for all structures except Permutation, where the default is the number of elements in the list to be permuted.)

```
> draw(Combination({a,b,c,d,e,f,g}));
```

$$\{f\}$$

```
> draw(Permutation(42));
```

$$[7, 35, 25, 18, 16, 39, 31, 38, 19, 33, 5, 28, 34, 26, 4, 29, 12, 6, 23, 32, 2, 17, 13, 9, 24, 15, 20, 22, 40, 3, 10, 1, 41, 8, 37, 30, 36, 42, 14, 11, 21, 27]$$

```
> draw(Permutation(16),size='allsizes');
```

$$[2, 7, 12, 16, 15]$$

```
> draw(Composition(32));
```

$$[6, 3, 2, 4, 2, 4, 1, 3, 1, 3, 3]$$

The function allstructs is used to generate all the structures of a given size.

```
> allstructs(Combination(4));
```

$$\{\{1\}, \{\ \}, \{1, 2, 3, 4\}, \{2, 3, 4\}, \{2\}, \{1, 2\}, \{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}, \{1, 2, 4\}, \{4\}, \{1, 4\}, \{2, 4\}, \{3, 4\}, \{1, 3, 4\}\}$$

```
> allstructs(Permutation([a,a,b,c]),size=3);
```

$$[[a, a, b], [a, a, c], [a, b, a], [a, b, c], [a, c, a], [a, c, b], [b, a, a], [b, a, c], [b, c, a], [c, a, a], [c, a, b], [c, b, a]]$$

```
> allstructs(Partition(4));
```

$$[[1, 1, 1, 1], [1, 1, 2], [2, 2], [1, 3], [4]]$$

```
> allstructs(Composition(6));
```

[[1, 5], [3, 3], [2, 4], [4, 2], [5, 1], [6], [2, 1, 3], [1, 2, 3], [3, 1, 2], [2, 2, 2], [1, 3, 2], [2, 3, 1], [1, 4, 1], [4, 1, 1], [3, 2, 1], [1, 3, 1, 1], [2, 2, 1, 1], [3, 1, 1, 1], [2, 1, 2, 1], [1, 2, 2, 1], [1, 1, 3, 1], [1, 1, 4], [1, 1, 1, 1, 1, 1], [2, 1, 1, 1, 1], [1, 1, 1, 1, 2], [1, 1, 2, 1, 1], [1, 2, 1, 1, 1], [1, 1, 1, 2, 1], [1, 1, 1, 3], [1, 1, 2, 2], [1, 2, 1, 2], [2, 1, 1, 2]]

The function iterstructs creates a mechanism to generate all the structures of a given size, one at a time.   The functions nextstruct and finished are used to manipulate the table returned by iterstructs.

```
> it := iterstructs(Combination([apple, orange, banana]),
  size=2):
> while not finished(it) do nextstruct(it) od;
```

[*apple*, *banana*]

[*apple*, *orange*]

[*banana*, *orange*]

```
> it := iterstructs(Permutation(3), size='allsizes'):
> while not finished(it) do nextstruct(it) od;
```

[ ]

[1]

[2]

[3]

[1, 2]

[2, 1]

[1, 3]

[3, 1]

[2, 3]

[3, 2]

[1, 2, 3]

[2, 1, 3]

[3, 1, 2]

[1, 3, 2]

[2, 3, 1]

[3, 2, 1]

You may add your own specialized structures into the combstruct system. Details on how to do this are in the help page for combstruct[structures].

```
>
```