So we want to create an algebra...

An *algebra* is a *vector space* equipped with a *multiplication*.

Suppose we want to create an algebra with our own special type of elements, that are multiplied in their own special way. We then have to be able to do two things:

1. create vector spaces consisting of our special elements,
2. multiply our special elements.

As example we take group algebras of finite groups. (We have forgotten that they are already implemented in GAP.)

Group algebra elements

Let G be a group and F be a field. Then an element of the group algebra FG has the form

$$c_1 g_1 + \cdots + c_t g_t$$

where $c_i \in F$ and $g_i \in G$.

So in GAP we represent a group algebra element as a list of the form

$$[g_1, c_1, \ldots, g_t, c_t].$$

A natural way of doing this is by making group algebra elements positional objects.

Some declarations

```
DeclareCategory( "IsGroupAlgElt", IsVector );
DeclareCategoryCollections( "IsGroupAlgElt" );
DeclareRepresentation( "IsGroupAlgEltRep", Is-
PositionalObjectRep, [ 1 ] );
DeclareGlobalFunction( "GroupAlgElement" );
```

`GroupAlgElement` is a function for creating group algebra elements. An implementation of it looks like:

```
InstallGlobalFunction( GroupAlgElement,
  function( fam, list )
  return Objectify( NewType( fam,
                             IsGroupAlgElt and
                             IsGroupAlgEltRep ),
                              [ list ] );
end );
```

We also install a `Print` method for our group algebra elements, and also methods for \+, * (scalar multiplication), `AINV` (additive inverse). These methods "describe" the vector space structure of the group algebra. Other handy methods are `Zero`, \<, \=.

Then the following is already possible:

```
gap> G:= SymmetricGroup(3);;
gap> e:= Elements( G );;
gap> fam:= NewFamily( "GroupAlgEltFam", IsGroupAl-
gElt );;
gap> u:= GroupAlgElement( fam, [ e[2], -1, e[4],
1/2 ] );
-1*(2,3)+1/2*(1,2,3)
gap> v:= GroupAlgElement( fam, [ e[4], 2, e[6],
1/3 ] );
2*(1,2,3)+1/3*(1,3)
gap> u+v;
-1*(2,3)+5/2*(1,2,3)+1/3*(1,3)
gap> u-v;
-1*(2,3)-3/2*(1,2,3)-1/3*(1,3)
gap> 2*u;
-2*(2,3)+1*(1,2,3)
```

We can even form the vector space spanned by `u,v`:

```
gap> V:= VectorSpace( Rationals, [ u, v ] );
<vector space over Rationals, with 2 generators>
```

However,

```
gap> Basis( V );
Error no method found for operation Basis at
Error( "no method found for operation ", name
);
<function>( <arguments> ) called from read-eval-
loop
Entering break read-eval-print loop, you can 'quit;'
to quit to outer loop,
or you can return to continue
brk>
```

GAP cannot compute a basis of `V` because we haven't told the system yet how to perform linear algebra for group algebra elements. (Checking for linear independence, for instance, must somehow boil down to solving a system of linear equations, and we must tell the system how to construct those linear equations.)

Suppose we want to check whether a given set of group algebra elements is linearly independent. The intuitive way of doing this is to map group algebra elements onto elements of a row-space, and do the checking there.

GAP has a standard mechanism for doing this. To get this going methods for the following operations have to be installed:

- `PrepareNiceFreeLeftModule( V )`, which computes any necessary information for the calls to the next two functions.
- `NiceVector( V, u )`, which computes a rowvector correponding to the vector `u` in the space `V`.
- `UglyVector( V, rowvec )`, which calculates the vector in `V` corresponding to `rowvec`.

Let G be a finite group consisting of the elements $g_1, \ldots, g_6$.

$$
\begin{array}{lccc}
\text{Ugly} & & c_3 g_3 + c_5 g_5 + c_6 g_6 & \\
& \texttt{NiceVector} \Big\downarrow & & \Big\uparrow \texttt{UglyVector} \\
\text{Nice} & & [0, 0, c_3, 0, c_5, c_6] &
\end{array}
$$

```
InstallMethod( PrepareNiceFreeLeftModule,
  "for free module of group alg elms",
  true,
  [ IsFreeLeftModule and IsHandledByNiceBasis
  and IsGroupAlgEltCollection], 0,

    Ignore );
```

```
InstallMethod( NiceVector,
"for group alg space, and group alg.  element",
IsCollsElms,
[ IsFreeLeftModule and IsHandledByNiceBasis,
IsGroupAlgElt and IsGroupAlgEltRep ], 0,
function( M, v )

  local k,z,v1,e,pos,fam;

  fam:= FamilyObj( v );
  e:= AsList( fam!.underlyingGroup );
  z:= ListWithIdenticalEntries( Length( e ),
                     fam!.zeroCoefficient );
  v1:= v![1];
  for k in [1,3..Length(v1)-1] do
     pos:= Position( e, v1[k] );
     z[pos]:= v1[k+1];
  od;
  return z;
  end );
```

```
InstallMethod( UglyVector,
"for group alg space, and vector",
true,
[ IsFreeLeftModule and IsHandledByNiceBasis
  and IsGroupAlgEltCollection, IsVector ], 0,
function( M, vec )

  local fam,e,lu,k;

  fam:= FamilyObj( Zero( M ) );
  e:= AsList( fam!.underlyingGroup );
  lu:= [ ];
  for k in [1..Length(vec)] do
     if vec[k] <> 0*vec[k] then
       Add( lu, e[k] );
       Add( lu, vec[k] );
     fi;
  od;
  return GroupAlgElement( fam, lu );
  end );
```

We also have to tell the vector space that the nice-vector mechanism has to be used when doing linear algebra. For this there is the category `IsHandledByNiceBasis`. If a vector space `V` lies in this category, then the nice-vector mechanism is triggered.

Since `IsHandledByNiceBasis` is a category, we must set it when creating the vector space. Therefore we have to install a new method for `FreeLeftModuleByGenerators`.

We also need two OtherMethods for `FreeLeftModuleByGenerators`, namely

- `FreeLeftModuleByGenerators( R, [ ], zero )` for creating the trivial space.
- `FreeLeftModuleByGenerators( R, list, zero )` for creating a space with prescribed zero.

```
  InstallMethod( LeftModuleByGenerators,
  "for ring and list of group alg elements",
  true,
  [ IsRing,
    IsGroupAlgEltCollection and IsList ], 0,
  function( R, vecs )

    local dims, V;

    V:= Objectify( NewType( FamilyObj( vecs ),
                   IsFreeLeftModule
                   and IsAttributeStoringRep
                   and IsHandledByNiceBasis ),
             rec() );

    SetLeftActingDomain( V, R );
    SetGeneratorsOfLeftModule(V, AsList(vecs));
    return V;
end );
```

```
gap> G:= SymmetricGroup( 3 );;
gap> e:= Elements( G );;
gap> fam:= NewFamily( "GroupAlgEltFam", IsGroupAl-
gElt );;
gap> fam!.underlyingGroup:= G;;
gap> fam!.zeroCoefficient:= 0;;
gap> u:= GroupAlgElement( fam, [e[2],3,e[5],-
1]);
3*(2,3)-1*(1,3,2)
gap> v:= GroupAlgElement( fam, [e[3],3/2,e[4],-
1,e[6],7/6]);
3/2*(1,2)-1*(1,2,3)+7/6*(1,3)
gap> V:= VectorSpace( Rationals, [ u, v, 2*v ]
);
<vector space over Rationals, with 3 generators>
gap> Basis( V );
Basis( <vector space over Rationals, with 3 gen-
erators>, ...  )
gap> BasisVectors( last );
[ 1*(2,3)-1/3*(1,3,2), 1*(1,2)-2/3*(1,2,3)+7/9*(1,3)
]
```

Now we want to turn our vector spaces into algabras. So we first must implement a method `\*` for multiplying elements. For this to be applicable we must also change the declaration of `IsGroupAlgElt` into something like

```
DeclareCategory( "IsGroupAlgElt", IsVector and
IsRingElement and
IsMultiplicativeElementWithOne );
```

An implementation of * is for instance

```
InstallMethod(\*,
"for two group alg elements",
true,
[ IsGroupAlgElt and IsGroupAlgEltRep,
  IsGroupAlgElt and IsGroupAlgEltRep ], 0,

function( u, v )

  local lu,lv,z,prod;

  lu:= u![1];
  lv:= v![1];
  if lu = [ ] then return u; fi;
  if lv = [ ] then return v; fi;

  z:= 0*lu[2];
  prod:= ZippedProduct( lu, lv, z,
              [ \*, \<, \+, \* ] );

  return GroupAlgElement(FamilyObj(u), prod );

end );
```

Now we can do the following:

```
gap> G:= SymmetricGroup( 3 );;
gap> e:= Elements( G );;
gap> fam:= NewFamily( "GroupAlgEltFam", IsGroupAl-
gElt );;
gap> u:= GroupAlgElement( fam, [e[2],3,e[5],-
1]);
3*(2,3)-1*(1,3,2)
gap> v:= GroupAlgElement( fam, [e[3],3/2,e[4],-
1,e[6],7/6]);
3/2*(1,2)-1*(1,2,3)+7/6*(1,3)
gap> u*v;
1*()-7/6*(2,3)-3*(1,2)+9/2*(1,2,3)+7/2*(1,3,2)-
3/2*(1,3)
A:= Algebra( Rationals, [ u, v ] );
<algebra over Rationals, with 2 generators>
gap> Basis( A );
Error no method found for operation Basis at........
```

To be able to work with the algebra we have to tell it that it `IsHandledByNiceBasis`. To do that we have to install a new method for `FLMLORByGenerators`.

We also need two OtherMethods for `FLMLORByGenerators`:

- `FLMLORByGenerators( R, [ ], zero )` to create the trivial algebra.
- `FLMLORByGenerators( R, list, zero )` to create an algebra with a prescribed zero.

```
InstallMethod( FLMLORByGenerators,
  "for ring and list of group alg elements",
  true,
  [ IsRing,
    IsGroupAlgEltCollection and IsList ], 0,
  function( R, vecs )
    local A;

    A:= Objectify( NewType( FamilyObj( vecs ),
                            IsFLMLOR
                        and IsAttributeStoringRep
                        and IsHandledByNiceBasis ),
                   rec() );
    SetLeftActingDomain( A, R );
    SetGeneratorsOfLeftOperatorRing( A,
                                     AsList( vecs ) );
    return A;
    end );
```

So finally....

```
gap> u:= GroupAlgElement( fam, [e[2],3,e[5],-
1]);
3*(2,3)-1*(1,3,2)
gap> v:= GroupAlgElement( fam, [e[3],3/2,e[4],-
1,e[6],7/6]);
3/2*(1,2)-1*(1,2,3)+7/6*(1,3)
gap> A:= Algebra( Rationals, [ u, v ] );
<algebra over Rationals, with 2 generators>
gap> Basis( A );
Basis( <algebra of dimension 6 over Rationals>,
...  )
gap> Dimension( A );
6
```

Now we can implement a function for creating a group algebra:

```
DeclareGlobalFunction( "GroupAlgebra" );

InstallGlobalFunction( GroupAlgebra,

  function( F, G )

    local e,fam,bas;

    e:= AsList( G );
    fam:= NewFamily( "GroupAlgEltFam",
                     IsGroupAlgElt);
    fam!.underlyingGroup:= G;
    fam!.zeroCoefficient:= Zero( F );
    bas:= List( e, x ->
             GroupAlgElement( fam, [x,One(F)] ));
    A:= FLMLORByGenerators( F, bas );

    return A;

  end );
```

Subalgebras of `A` are automatically `IsHandledByNiceBasis`. However, ideals are not. (Subalgebras are created using `FLMLORByGenerators`, ideals are created in a different way because they need to know the parent algebra.) So if we want to use ideals we have to install a new method for `IdealByGenerators`.

Also if the algebra has a one (as in our case), a method for `One` is handy (essential for some things to work).

```
InstallMethod( IdealByGenerators,
  "for group alg.  and coll.  of group alg elts",
  IsIdenticalObj,
  [ IsFLMLOR,
    IsGroupAlgEltCollection and IsList ], 0,
  function( A, vecs )

    local I;

    I:= Objectify( NewType( FamilyObj( A ),
                             IsFLMLOR
                          and IsAttributeStoringRep
                          and IsHandledByNiceBasis ),
                         rec() );

    SetLeftActingDomain( I, LeftActingDomain(A));
    SetGeneratorsOfTwoSidedIdeal( I, vecs );
    SetLeftActingRingOfIdeal( I, A );
    SetRightActingRingOfIdeal( I, A );

    if IsEmpty( vecs ) then
    SetIsTrivial( I, true );
    fi;

    return I;
end );
```

```
InstallMethod(One,
  "for group alg element",
  true,
  [ IsGroupAlgElt and IsGroupAlgEltRep ], 0,

  function( u )
    local fam, G;

    fam:= FamilyObj( u );
    G:= fam!.underlyingGroup;
    return GroupAlgElement( fam,
      [ One(G), One( fam!.zeroCoefficient )] );

  end );
```

Succes...

```
gap> G:= SymmetricGroup( 4 );;
gap> A:= GroupAlgebra( Rationals, G );
<algebra over Rationals, with 24 generators>
gap> Basis( A );
Basis( <algebra of dimension 24 over Rationals>,
...  )
gap> RadicalOfAlgebra( A );
<algebra of dimension 0 over Rationals>
gap> e:= CentralIdempotentsOfAlgebra( A );;
gap> Length( e );
5
gap> ideals:= List( e, x -> Ideal( A, [ x ] )
);;
gap> List( ideals, Dimension );
[ 1, 9, 9, 1, 4 ]
```

Summary

To create a new type of algebra one has to

1. implement a function to create elements of the algebra,
2. implement some methods for these elements (like `Print`, `\+` `AINV` etc),
3. implement methods for `PrepareNiceFreeLeftModule`, `NiceVector`, `UglyVector`,
4. implement methods for `FreeLeftModuleByGenerators`,
5. implement methods for `FLMLORByGenerators`,
6. implement a method for `IdealByGenerators`,
7. implement other handy methods, like `One`, `IsFiniteDimensional`.