Modify the text in section 2.7

The nesting of braces must follow the number of dimensions, unlike in C. However, replicate operators can be nested. The inner pair of braces in a replication are removed. Each replication represents one dimension.

```
    int n[1:2][1:6] = {2{{3{4,5}}}}; // {{4,5,4,5,4,5},{4,5,4,5,4,5}}
```

If the type is not given by the context, it must be specified with a cast.

```
    typedef int ~~[1:3]~~ triple [1:3]; // Array of 3 integers ~~packed together~~
    ~~b = triple'{0,1,2};~~
    $mydisplay(triple'{0,1,2});
```

Array literals can also use their index or type as a key, and a default key value (see Section 7.13):

```
    b = {1:1, default:0}; // indexes 2 and 3 assigned 0
```

ADD to the end of section 2.8

Replicate operators can be used to represent their exact number of members. The inner pair of braces in a replication are removed.

```
    struct {int X,Y,Z;} XYZ = {3{1}};
    typedef struct {int a,b[4]} ab_t;
    int a,b,c;
    ab_t v1[1:0][2:0];
    v1 = {2{{3{a,{2{b,c}}}}}};
```

Move section 7.12 static prefixes before section 7.11 Concatenation.

DELETE the following text in 7.11 (will become section 7.12)

~~Note that braces are also used for initializers of structures or unpacked arrays. Unlike in C, the expressions must match element for element and the braces must match the structures and array dimensions. Each element must match the type being initialized, so the following do not give size warnings:~~

```
~~  bit unpackedbits [1:0] = {1,1}; // no size warning, bit can be set to 1~~
~~  int unpackedints [1:0] = {1'b1,1'b1}; //no size warning, int can be set to 1'b1~~
```

~~A concatenation of unsized values shall be illegal, as in Verilog. However, an array initializer can use unsized values within the braces, such as {1,1}.~~

~~The replication operator (also called a multiple concatenation) form of braces can also be used for initializers . For example, {3{1}} represents the initializer {1, 1, 1}.~~

Modify section 7.13

Braces are also used for expressions to assign to unpacked arrays. Unlike in C, the expressions must match element for element, and the braces must match the array dimensions. ~~The type of each element is matched against the type of the expression according to the same rules as for a scalar.~~ Each expression item shall be evaluated in the context of an assignment to the type of the corresponding element in the array. This means that the following examples do not give size warnings, unlike the similar assignments above:

The syntax of multiple concatenations can be used for unpacked array expressions as well. Each replication represents a single dimension.

```
    unpackedbits = {2 {y}} ; // same as {y, y}
    int n[1:2][1:3] = {2{{3{y}}}}; // same as {{y,y,y},{y,y,y}}
```

SystemVerilog determines the context of the braces ~~by looking at the left hand side of an~~ when used in the context of an assignment. If ~~the left hand side is an~~ used in the context of an assignment to an unpacked array, the braces represent an unpacked array literal or expression. Outside the context of an assignment ~~on the right hand side~~ to an aggregate type, an explicit cast shall be used with the braces to distinguish it from a concatenation.

An unpacked array expression can address specific indexes with the `{index:value}` syntax. The expression need not specify all elements in the array unless being used as an initialization in a declaration. For dynamic arrays and queues, the element must already exist. See section 4.1.4 for addressing associative arrays.

```
  int n[3:1] = {2:1,3:2,1:3};
  n = {2:x,3:y}; // only element 2 written, equivalent to
  n[2] = x;
  n[3] = y;
```

It can sometimes be useful to set array elements to a value without having to keep track of how many members there are. This can be done with the **default** keyword:

```
  initial unpackedints = {default:2}; // sets elements to 2
```

For more arrays of structures, it is useful to specify one or more matching types keys, as illustrated under structure
expressions, below.

```
  struct {int a; time b;} abkey[1:0];
  abkey = {{a:1, b:2ns}, {int:5, time:$time}};
```

When the braces include an index, type, or default key, the braces are syntactically distinguished from being a concatenation for both packed and unpacked array types.

The rules for unpacked array matching are as follows:
— An `index:value` specifies an explicit value for a keyed element index. The value is evaluated in the context of an assignment to the indexed element and must be castable to its type. It shall be an error to specify the same index more than once in a single array expression.
— For `type:value`, if the element or sub array type of the unpacked array ~~exactly matches~~ is equivalent to this type, then each element or sub array shall be set to the value that has not been matched by an index above. The value is evaluated in the context of an assignment to the matching type must be castable to the array element or sub array type. Otherwise, if the unpacked array is multidimensional, then there is a recursive descent into each sub array of the array using the rules in this section and the type and default ~~specifiers~~ keys. Otherwise, if the unpacked array is an array of structures, there is a recursive descent into each element of the array using the rules for structure expressions and the type and default ~~specifiers~~ keys. If more than one type matches the same element, the last value shall be used.
— For `default:value`, this key specifies the default value to use for each element of an unpacked array that has not been covered by the earlier rules in this section. The value is evaluated in the context of each assignment to an element covered by the default and must be castable to the array element type.

Every element shall be covered by one of these rules.

If the type key, default key, or replication operator is used on an expression with side effects, the number of times that expression evaluates is undefined.

Modify section 7.14

A structure expression (packed or unpacked) can be built from member expressions using braces and commas, with the members in declaration order. Each member expression shall be evaluated in the context of an assignment to the type of the corresponding member in the structure. It can also be built with the names of the members

…

SystemVerilog determines the context of the braces ~~by looking at the left hand side of an~~ when used in the context of an assignment. If ~~the left hand side is an~~ used in the context of an assignment to an unpacked structure, the braces represent an unpacked structure literal or expression. Outside the context of an assignment ~~on the right hand side~~ to an aggregate type, an explicit cast must be used with the braces to distinguish it from a concatenation. When the braces include a label, type, or default key, the braces are syntactically distinguished from being a concatenation for both packed and unpacked structure types.

…

The matching rules are as follows:
— A `member:value`: specifies an explicit value for a named member of the structure. The named member must be at the top level of the structure—a member with the same name in some level of substructure shall not be set. The value must be castable to the member type and is evaluated in the context of an assignment to the named member, otherwise an error is generated.
— The `type:value` specifies an explicit value for a field in the structure which ~~exactly matches~~ is equivalent to the type and has not been set by a field name ~~specifiers~~ key above. If the same type key is mentioned more than once, the last value is used. The value is evaluated in the context of an assignment to the matching type
— The `default:value` applies to members that are not matched by either member name or type key and are not either structures or unpacked arrays. The value is evaluated in the context of each assignment to a member covered by the default and must be castable to the member type, otherwise an error is generated. For unmatched structure members, the type and default specifiers are applied recursively according to the rules in this section to each member of the substructure. For unmatched unpacked array members, the type and default ~~specifiers~~ keys are applied to the array according to the rules for unpacked arrays.

Every ~~element~~ member shall be covered by one of these rules.

If the type key, default key, or replication operator is used on an expression with side effects, the number of times that expression evaluates is undefined.