TypeLiteral:
  ObjectType
  ArrayType
  FunctionType
  ConstructorType

ArrayType:
  PredefinedType `[` `]`
  TypeReference `[` `]`
  ObjectType `[` `]`
  ArrayType `[` `]`

FunctionType:
  TypeParameters$_{opt}$ `(` *ParameterList*$_{opt}$ `)` `=>` *Type*

ConstructorType:
  `new` *TypeParameters*$_{opt}$ `(` *ParameterList*$_{opt}$ `)` `=>` *Type*

Object type literals are the primary form of type literals and are described in section 3.7. Array, function, and constructor type literals are simply shorthand notations for other types:

Type literal	Equivalent form
T `[ ]`	`Array <` *T* `>`
`<` *TParams* `> (` *Params* `) =>` *Result*	`{ <` *TParams* `> (` *Params* `) :` *Result* `}`
`new <` *TParams* `> (` *Params* `) =>` *Result*	`{ new <` *TParams* `> (` *Params* `) :` *Result* `}`

As the table above illustrates, an array type literal is shorthand for a reference to the generic interface type 'Array' in the global module, a function type literal is shorthand for an object type containing a single call signature, and a constructor type literal is shorthand for an object type containing a single construct signature. Note that function and constructor types with multiple call or construct signatures cannot be written as function or constructor type literals but must instead be written as object type literals.

In order to avoid grammar ambiguities, array type literals permit only a restricted set of notations for the element type. Specifically, an A*rrayType* cannot start with a *TypeQuery*, *FunctionType*, or *ConstructorType*. To use one of those forms for the element type, an array type must be written using the 'Array<T>' notation. For example, the type

```
() => string[]
```

denotes a function returning a string array, not an array of functions returning string. The latter can be expressed using 'Array<T>' notation

```
Array<() => string>
```