and have no impact on the emitted JavaScript (and therefore no run-time cost). The type and the enclosing < and > are simply removed from the generated code.

In a type assertion expression of the form < *T* > *e*, *e* is contextually typed (section 4.19) by *T* and the resulting type of *e* is required to be assignable to or from *T*, or otherwise a compile-time error occurs. The type of the result is *T*.

Type assertions check for assignment compatibility in both directions. Thus, type assertions allow type conversions that *might* be correct, but aren't *known* to be correct. In the example

```
class Shape { ... }

class Circle extends Shape { ... }

function createShape(kind: string): Shape {
    if (kind === "circle") return new Circle();
    ...
}

var circle = <Circle> createShape("circle");
```

the type annotations indicate that the 'createShape' function *might* return a 'Circle' (because 'Circle' is a subtype of 'Shape'), but isn't *known* to do so (because its return type is 'Shape'). Therefore, a type assertion is needed to treat the result as a 'Circle'.

As mentioned above, type assertions are not checked at run-time and it is up to the programmer to guard against errors, for example using the `instanceof` operator:

```
var shape = createShape(shapeKind);
if (shape instanceof Circle) {
    var circle = <Circle> shape;
    ...
}
```

## 4.14 Unary Operators

The subsections that follow specify the compile-time processing rules of the unary operators. In general, if the operand of a unary operator does not meet the stated requirements, a compile-time error occurs and the result of the operation defaults to type Any in further processing.

### 4.14.1 The ++ and -- operators

These operators, in prefix or postfix form, require their operand to be of type Any, the Number primitive type, or an enum type, and classified as a reference (section 4.1). They produce a result of the Number primitive type.