For *formal annotation* in program source code (typically appearing in comments), we will adhere to the Java syntax of expressions, extended with notation (adapted) from the *Java Modeling Language*. See [1] for an overview.

Notation in source code is (unfortunately) restricted to the ASCII alphabet. We will have to learn to live with this:

Operator	In source	NOT USED	Elsewhere
Equality	`==`	`=`	$=$
Inequality	`!=`	`/=`	$\neq$
Negation	`!`	`not`	$\neg$
Conjunction	`&&`	`/\`	$\wedge$
Disjunction	`||`	`\/`	$\vee$
Implication	`==>`	`=>`	$\Rightarrow$
Follows from	`<==`	`=>`	$\Leftarrow$
Equivalence	`<==>`	`==`	$\equiv$ or $\Leftrightarrow$

Quantified expressions take the form

(*quantifier quantified-var-decls* **;** *predicate* **;** *spec-expression*)

where

- *quantifier* is the operator (see table below);
- *quantified-var-decls* introduces the dummy or dummies; officially, it is required to include a type, but when clear from the context, we omit this;
- *predicate* expresses the range for the dummy; it is optional; leaving it out is the same as writing **`true`**;
- *spec-expression* is the quantified term of appropriate type.

Quantifier	In source code	Elsewhere	Type of term	Type of result
Universal	`\forall`	$\forall$	**`boolean`**	**`boolean`**
Existential	`\exists`	$\exists$	**`boolean`**	**`boolean`**
Minimum	`\min`	min or $\downarrow$	numeric	numeric
Maximum	`\max`	max or $\uparrow$	numeric	numeric
Sum	`\sum`	$\sum$	numeric	numeric
Product	`\product`	$\prod$	numeric	numeric
Number of	`\num_of`	$\#$	**`boolean`**	numeric

N.B. Outside program source code, we prefer standard mathematical notation.

In this course, we will not use the JML keywords. We will use

Clause	In source code	Elsewhere
Precondition	`@pre`	Assumes, Requires
Modifies	`@modifies`	Modifies
Postcondition	`@post`	Ensures, Effects
Return	`@return`	

The precondition and postcondition are *predicates*, possibly involving (instance) variables and parameters. The modifies clause provides a *list* of all variables and parameters that the method can modify. The return clause is an *expression* of the method's return type, possibly involving variables and parameters.

There are some special JML symbols that can be used in various clauses. We will encounter

- `\result` in postconditions, for the return value of the method that follows;
- `\old(`*expression*`)` in postconditions and returns clauses for referring to the value of *expression* at the time of entry into a method; it is (obviously) not needed in preconditions and only relevant when involving parameters or variables that appear in the modifies clause.

Thus, the following two clauses express the same condition:

- **Returns:** `F`
- **Postcondition:** `\result == F`

Tools that operate on formal annotation exist in rudimentary forms, and are currently developed further. These tools can check syntactic validity of such annotation, verify (to some extent) logical consistency (also with the program code), and assist in constructing valid annotation.

On being formal

We will strive to be as formal as possible in our annotations, within reason. For this, you need to develop a sense of *balance*.

Annotation is often informal, but must aim to be precise. We prefer a formula, where reasonably possible.

New formalism and notation will be introduced when needed.

Some examples

For all integers a, if a is divisible by 6, then it is also divisible by both 2 and 3:

```
(\forall int a; a % 6 == 0; a % 2 == 0 && a % 3 == 0)
```

If the range predicate is true, it can be left out. All integer squares are at least zero:

```
(\forall int a; ; 0 <= a * a)
```

is equivalent to

```
(\forall int a; true; 0 <= a * a)
```

If the type is clear from the context, it can be left out. In the context `int C, n`, for all C, there exists an n with $2^n > C$:

```
(\forall C; ; (\exists n; ; 2^n > C))
```

The greatest common divisor of integers a and b is given by:

```
(\max d; ; a % d == 0 && b % d == 0)
```

The number n is called perfect when its true divisors sum to n, i.e., when

```
(\sum d; 0 < d < n && n % d == 0; d) == n
```

Note that here $0 < d < n$ is shorthand for `0 < d && d < n`.
The number of strictly negative elements in array a:

```
(\num_of i; 0 <= i < a.length; a[i] < 0)
```

For arrays, we use the abbreviation `a.has(i)` for "i is a valid index of `a`":

```
a.has(i)  ==  0 <= i < a.length
```

For instance, the number of (strictly) negative numbers in array a is

```
(\num_of i; a.has(i); a[i] < 0)
```

Multiple quantifiers of the same kind can be combined. Number n is composite when it can be written as a product of two numbers larger than one:

```
(\exists a, b; 1 < a && 1 < b; a * b == n)
```

is equivalent to

```
(\exists a; 1 < a; (\exists b; 1 < b; a * b == n))
```

References

[1] *Java Modeling Language*. Wikipedia article.
en.wikipedia.org/wiki/Java_Modeling_Language