The handout on notation summarizes the (JML-inspired) syntax used in this course for formal annotation (appearing in *javadoc* comments). This note provides some guidelines for the writing of formal specifications.

Formal definitions should first aim at understandability Writing formal annotation resembles writing program code: every symbol matters and a small change can radically change the meaning (also see the example at the end). Because formal annotation and program code differ in purpose, also the trade-offs differ. Program code is written to be executable by a machine to solve a specific problem. Formal annotation is written to assist in the design and understanding of the program code.

For efficiency's sake, it can be necessary to sacrifice understandability of program code. However, formal annotation should always aim at ease of understanding; in particular, it should be easy to convince oneself that a formal specification correctly expresses the intention. There is no such thing as efficient execution of formulas.

Here are some examples of equivalent predicates involving integers b and c, and an integer array a of N elements. Which ones do you find easier to understand, which ones raise fewer doubts?

a is constant

$$(\exists\, C :: (\forall\, i : 0 \le i < N : a[i] = C)) \tag{1}$$
$$(\forall\, i : 0 \le i < N : a[i] = a[0]) \tag{2}$$
$$(\forall\, i, j : 0 \le i \wedge N > j \wedge i < N \wedge j > 0 : a[i] = a[j]) \tag{3}$$
$$(\forall\, i, j : 0 \le i < j < N : a[i] = a[j]) \tag{4}$$
$$(\forall\, i, j : 0 \le i, j < N : a[i] = a[j]) \tag{5}$$

a is sorted in ascending order

$$(\forall\, i : 0 < i < N : a[i-1] \le a[i]) \tag{6}$$
$$(\forall\, i : 0 \le i < N : a[i] = (\uparrow\ j : 0 \le j \le i : a[j])) \tag{7}$$
$$(\forall\, i : 0 \le i < j < N : a[i] \le a[j]) \tag{8}$$

a is a palindrome

$$(\forall\, i : 0 \le i < N/2 : a[i] = a[N-i-1]) \tag{9}$$
$$(\forall\, i : 0 \le i < N : a[i] = a[N-i-1]) \tag{10}$$
$$(\forall\, i, j : 0 \le i, j < N\ \wedge\ i + j = N - 1 : a[i] = a[j]) \tag{11}$$

b is prime ($d|b$ denotes that d is a divisor of b)

$$\neg(\exists\, d : 1 < d \le \sqrt{b} : d|b) \tag{12}$$
$$\neg(\exists\, d : 1 < d < \sqrt{b} + 1 : d|b) \tag{13}$$
$$(\forall\, d : 1 < d < b : \neg(d|b)) \tag{14}$$

Split longer definitions A formula consisting of many symbols is not easy to grasp. It is advisable to split this. In particular, nested quantifiers are a burden on the reader. The statement that 'b is the index of the leftmost element in array a having property P' can be expressed as the conjunction of three predicates:

$$Q_0 \;:\; 0 \leq b < N \tag{15}$$
$$Q_1 \;:\; P(a[b]) \tag{16}$$
$$Q_2 \;:\; (\forall\, i : 0 \leq i < b : \neg P(a[i])) \tag{17}$$

where

- (15) expresses that b is an index in array a
- (16) expresses that $a[b]$ has property P
- (17) expresses that no element to the left of b has property P

It is useful to name these conjuncts separately, rather than trying to combine them into a single predicate.

Define auxiliary concepts In general, it is a useful technique to define auxiliary predicates (*Divide & Conquer*). For instance, express that array a is a permutation of the numbers 0 (inclusive) to N (exclusive). One way of doing this could be:

$$(\forall\, v : 0 \leq v < N : (\exists\, i : 0 \leq i < N : a[i] = v)) \tag{18}$$

The nested quantifier can be avoided by defining predicate $C(a, v)$ stating that array a contains value v:

$$C(a, v) \;=\; (\exists\, i : 0 \leq i < N : a[i] = v) \tag{19}$$

Now, (18) can be shortened to

$$(\forall\, v : 0 \leq v < N : C(a, v)) \tag{20}$$

It can also be useful to define auxiliary concepts that are not predicates. For instance, define $a\#v$ as the number of occurrences of value v in array a:

$$a\#v \;=\; (\#\, i : 0 \leq i < N : a[i] = v) \tag{21}$$

Then we have $C(a, v) \;\Leftrightarrow\; a\#v > 0$. Hence, (18) can be rewritten as

$$(\forall\, v : 0 \leq v < N : a\#v = 1) \tag{22}$$

Notice the subtle difference between (20) and (22): the latter explicitly prohibits that values occur more than once in a permutation, whereas in the former it requires some work to infer that.

Parameterize definitions Even when defining a single predicate, it can be helpful to parameterize the definition. As an example, consider predicate $A(p, q)$ defined for $0 \leq p \leq q \leq N$ by

$$A(p,q) \quad = \quad (\forall\, i : p \leq i < j < q : a[i] \leq a[j]) \tag{23}$$

It expresses that array a is sorted in ascending order on the segment $[p..q)$. The statement that a is sorted in ascending order is then expressed by $A(0, N)$. In a program, the predicate $A(0, n)$ with $0 \leq n \leq N$ could play a role as invariant.

The alternative definition (7) of ascending order can be reformulated by first defining $M(k)$ as the maximum of $a[0..k)$ for $0 \leq k \leq N$:

$$M(k) \quad = \quad (\uparrow\ j : 0 \leq j < k : a[j]) \tag{24}$$

and then simplifying (7) as

$$(\forall\, i : 0 \leq i < N : a[i] = M(i+1)) \tag{25}$$

A parameterized definition makes it easier to express properties, such as:

$$M(0) \quad = \quad -\infty \tag{26}$$

$$M(k+1) \quad = \quad M(k)\ \uparrow\ a[k] \quad \text{for } 0 \leq k < N \tag{27}$$

Avoid the temptation to make everything a parameter, because this could clutter notation needlessly. For instance, generalizing (23) further to

$$A(a, R, p, q) \quad = \quad (\forall\, i : p \leq i < j < q : a[i]\ R\ a[j]) \tag{28}$$

is not helpful if the array a and order relation $<$ are fixed in the context.

Rewrite later, depending on role After giving a formal definition, it can be useful to rewrite it into alternative forms. One reason for this is that it may help improve understanding. Another reason is that it can help in problem solving.

Note that a predicate can play different roles in reasoning. On the one hand, it can be a condition that needs to be established (as is the case with a method precondition when designing a call to that method, or with a method postcondition when designing an implementation for that method). On the other hand, it can be a condition that can be assumed to hold and which can be exploited (as is the case with a method postcondition when designing a call to that method, or with a method precondition when designing an implementation for that method).

When exploiting a condition, the predicate is most useful when one can easily derive consequences from it. For instance, (2) as predicate to express that array a is constant offers fewer possibilities to infer $a[i] = a[j]$ than (4), which in turn is (slightly) more restrictive than (5).

Conversely, when establishing a condition, one prefers a predicate that requires less intellectual effort to infer. In that case, one would prefer (2) over (4) and (5).

The initial definition needs to be easy to understand. As a next step one can look for equivalent predicates that are more suitable as precondition or postcondition.

Balance formality depending on purpose In this course, we will strive for precise specifications, preferably expressed in formal mathematical notation. However, not all problem domains facilitate formal definitions. For instance, in the area of user interface design, it will not always be easy to provide formal specifications. Still, one can do one's best to be as precise as possible. Even specifications written in natural language can aim at precision. In [1], the authors provide carefully worded specifications, using little formal notation.

One needs to develop a feeling for balancing the level of formality, weighing effort and benefits. We believe that it is better to learn to use formal notation as early as possible. Once one is familiar with it, one can learn to be precise in a less formal way. The other way round —first informal, then more formal— is less likely to be successful.

The effect of changing a single symbol This example is borrowed from [2]. Consider the following two predicates P_- and P_+, where $\mathbb{N}'$ denotes the set of positive natural numbers and $\mathbb{P}$ denotes the set of prime numbers.

$$\begin{array}{lcl} P_- & : & (\forall\, n : n \in \mathbb{N}' : (\exists\, k : k \in \mathbb{N}' : n + k \in \mathbb{P} \ \wedge\ n - k + 2 \in \mathbb{P})) \\ P_+ & : & (\forall\, n : n \in \mathbb{N}' : (\exists\, k : k \in \mathbb{N}' : n + k \in \mathbb{P} \ \wedge\ n + k + 2 \in \mathbb{P})) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \uparrow \end{array}$$

Visual inspection shows that these two predicates differ in one symbol only: at the arrow, P_- has a $-$ symbol where P_+ has a $+$ symbol.

What do these predicates 'mean'? Are they just arbitrary useless predicates that happen to resemble each other?

Convince yourself that these predicates can be rewritten into G and T respectively:

$$\begin{array}{lcl} G & : & (\forall\, n : n \in \mathbb{N}' : (\exists\, p_1, p_2 : p_1 \in \mathbb{P} \ \wedge\ p_2 \in \mathbb{P} : 2n + 2 = p_1 + p_2)) \\ T & : & (\forall\, n : n \in \mathbb{N}' : (\exists\, p_1, p_2 : p_1 \in \mathbb{P} \ \wedge\ p_2 \in \mathbb{P} \ \wedge\ p_1 > n : p_2 - p_1 = 2)) \end{array}$$

Here is the reasoning in condensed form:

- $P_- \Rightarrow G$: take $p_1, p_2 = n + k, n - k + 2$, then $p_1 + p_2 = 2n + 2$
- $P_+ \Rightarrow T$: take $p_1, p_2 = n + k, n + k + 2$, then $p_2 - p_1 = 2$
- $G \Rightarrow P_-$: without loss of generality (w.l.o.g.), assume $p_2 \leq p_1$; hence, $p_2 \leq n{+}1$; now take $k = n{-}p_2{+}2$; thus $k \in \mathbb{N}'$ and $p_1, p_2 = n{+}k, n{-}k{+}2$
- $T \Rightarrow P_+$: w.l.o.g., assume $p_1 \leq p_2$; now take $k = p_1 - n$; thus $k \in \mathbb{N}'$ on account of $p_1 > n$, and $p_1, p_2 = n + k, n + k + 2$

Predicate G is the famous *Goldbach Conjecture* that states that *every even number greater than two can be written as the sum of two primes*. Predicate T is the famous *Twin Prime Conjecture* that states that *there exist infinitely many twin primes*, that is, pairs of primes differing by two. Both conjectures remain open as of today.

> This example is philosophically interesting because the two predicates P_- and P_+ involve exactly the same concepts, but P_- could, in principle, be disproved by a single counterexample (an even number greater than two that is not the sum of two primes; there is only a finite number of candidate primes to test), whereas P_+ cannot be so disproved.

The lesson for us is that changing a single symbol can make a huge difference in meaning. Therefore, formal definitions must be written so that they convey the intended meaning as clearly as possible and that they can be easily validated. It is also advisable to accompany a formal definition with an informal description, and to state and prove some elementary properties of the defined concept to support the formal definition.

References

[1] B. Liskov, J. Guttag. *Program Development in Java*. Addison-Wesley, 2001.

[2] K.R. Popper. *Realism and the Aim of Science*. Hutchinson, 1983.