This lecture note supplements the treatment of predicates and quantifiers given in standard textbooks on Discrete Mathematics (e.g.: [1]) and introduces the notation used in this course. We will present central concepts that are important, when predicate logic is used for specification and verification of algorithms. The note is partly based on [2] and [3].

Notation for predicates

As you may recall, the predicate logic extends the propositional logic in two ways: Free variables are allowed, for instance:

\[ x < y \land y < z \]

is a predicate, which becomes a proposition (true or false) when \( x, y \) and \( z \) are assigned values. For instance, if \( x = 1, y = 2 \) and \( z = 3 \), then the predicate becomes a proposition:

\[ 1 < 2 \land 2 < 3 \]

which is obviously true.

And secondly, quantifiers may be used to state properties about sets of elements. For instance:

\[ (\exists i \mid 0 \leq i < 10 : b[i] = 0) \]

is a predicate that is true, if there exist a zero value in the array \( b[0..10) \) (the range from 0 inclusive to 10 exclusive).

We will use the above notation (1) for all predicates involving quantifiers. For instance, instead of writing the sum of all integers from 1 to \( n-1 \) (inclusive) as usual in math:

\[ \sum_{i=1}^{n} i = \frac{(n-1)n}{2} \]

we will write:

\[ (\Sigma i \mid 1 \leq i < n: i) = (n-1)n/2 \]
This one-line notation is more convenient for typing, and as we shall see below; it opens for generalisation.

The summation quantifier ($\Sigma$) is the generalised addition operator (+), and we may even avoid the special symbol “$\Sigma$” and just use the addition symbol “+” as symbol for the quantifier and then writing he predicate as:

$$(+ i \mid 1 \leq i < n: i) = 1 + 2 + ... + (n - 1)$$

But the summation symbol is well-known and widely used, so normally we will stick to:

$$(\Sigma i \mid 1 \leq i < n : i) = 1 + 2 + ... + (n - 1)$$

Another example:

$$(\Sigma i \mid 1 \leq i < n : i^2 + i + 1) = (1^2 + 1 + 1) + (2^2 + 2 + 1) + ... + ((n - 1)^2 + (n - 1) + 1)$$

In general, a quantified summation has the following form:

$$(\Sigma i \mid R(i) : T(i))$$

where the predicate $R(i)$ is the range and defines the set to be summed (often integers, but not necessarily an interval), and $T(i)$ is the term to be summed. Normally the term depends on $i$ and must be type compatible with the quantifier (summation in this case).

In (2) and (3) the range is the interval [0; $n$), in (2) the term is $i$ and in (3), $T(i) = i^2 + i + 1$.

In (1) the range is [0; 10) and the term is $b[i] = 0$. Note the type of the term in this case is boolean, since the return type of the existential quantifier (“$\exists$”) is boolean.

The term, $T(i)$ does not have to depend on $i$, but will mostly do so:

$$(\Sigma i \mid 0 \leq i < n : 2) = 2 + 2 + ... + 2 \text{ (n times)} = 2 \cdot n$$

The form given in (4) opens for ranges, which are not intervals, for instance (assume that even($x$) is a predicate returning true if $x$ an even number and false otherwise):

$$(\Sigma i \mid 1 \leq i \leq 2 \cdot n \land even(i) : i) = 2 + 4 + ... + 2 \cdot n$$

Here, the range is the sets of integers in [1; 2n] that also are even.

**Other quantifiers**

It is possible to define lots of other quantifiers. Some of the most used are $\Pi$, $\forall$ and $\exists$ (multiplication, universal (“for all”) and existential (“exists”)). These quantifiers correspond with the operators $\cdot$, $\land$ and $\lor$ (multiplication, logical “and” and logical “or”).
Another useful quantifier is count. We will use the symbol “#”, and define counting as follows:

\[(\# i \mid R(i) : T(i)) = (\Sigma i \mid R(i) \land T(i) : 1)\]

In (5), we count the number of elements in \(R(i)\) that have the property \(T(i)\). That is equivalent to summing ones in range \(R(i) \land T(i)\) (if you have experience in functional and/or parallel programming, then think Map-Reduce pattern).

Some examples:

\[
\begin{array}{ll}
(\Pi i \mid 0 \leq i < 3 : i + (i+1)) & = (\cdot i \mid 0 \leq i < 3 : i + (i+1)) \\
& = (0 + 1) \cdot (1 + 2) \cdot (2 + 3)
\end{array}
\]

\[
\begin{array}{ll}
(\Sigma i \mid 0 \leq i < 10 : even(i)) & = (\Sigma i \mid 0 \leq i < 10 \land even(i) : 1) \\
& = 5
\end{array}
\]

\[
\begin{array}{ll}
(\forall i \mid 1 \leq i \leq 2 : i \cdot d \neq 0) & = (\land i \mid 1 \leq i \leq 2 : i \cdot d \neq 0) \\
& = ((1 \cdot d \neq 0) \land (2 \cdot d \neq 0))
\end{array}
\]

\[
\begin{array}{ll}
(\exists i \mid 0 \leq i < 10 : b[i] = 0) & = (\lor i \mid 0 \leq i < 10 : b[i] = 0) \\
& = ((b[0] = 0) \lor (b[1] = 0) \lor ... \lor (b[9] = 0))
\end{array}
\]

Note that in the last three examples, the term is a boolean expression – otherwise it wouldn’t make sense to use the quantifiers #, \(\land\) and \(\lor\).

**Identity – or neutral value for a quantifier**

Take a look at this quantified expression that states that we want the sum of the primes in some interval (\(prime(i)\) is a predicate giving true if \(i\) is a prime and false, if not).

\[(\Sigma i \mid a \leq i \leq b \land prime(i) : i)\]

If for instance, \(a = 5\) and \(b = 10\), we have:

\[(\Sigma i \mid 5 \leq i \leq 10 \land prime(i) : i) = 5 + 7 = 13\]

(5 and 7 being the only primes in the interval).

But what if \(a = 8\) and \(b = 10\)? In that case, there are no primes in the interval, so what should the sum of nothing be? Intuitively, the sum of nothing could be defined to be 0 (zero) that will make sense. Fortunately, math and common sense agrees (at least in this case).

Generally, the result of a quantifier over an empty range is defined as the neutral value (also called the identity) for the corresponding operator. If an operator is applied to a value and the identity, then the result will be the value itself.
For instance:

\[ 0 + x = x, \quad 1 \cdot x = x; \quad true \land p \equiv p \quad \text{and} \quad false \lor p \equiv p \]

So the neutral value for addition is 0, for multiplication 1, for logical and \( true \) and for logical or \( false \).

In general, given a binary operator \( * \) the identity \( u \) for \( * \) has the property:

\[ u \ast x = x \]

for all \( x \) in the set of values where \( * \) is defined.

This means that the quantifiers used on an empty range will yield the following:

\[
\begin{align*}
(\Sigma i \mid false : T(i)) &= 0 \quad \text{-- summing nothing: 0} \\
(\Pi i \mid false : T(i)) &= 1 \quad \text{-- multiplying nothing: 1} \\
(#) i \mid false : T(i) &= 0 \quad \text{-- counting nothing: 0} \\
(\forall i \mid false : T(i)) &\equiv true \quad \text{-- if no elements, everything is true for all of them} \\
(\exists i \mid false : T(i)) &\equiv false \quad \text{-- if no elements, does there exists an element with a given property \( T(i) \)?: false}
\end{align*}
\]

In programming neutral values are used when initialising variables in connection with loops and defining base cases (termination rules) for recursive definitions and functions.

**Defining quantifiers**

It is possible to define new quantifiers. Every associative and symmetric operator with a neutral element can be generalised into a quantifier.

**A side:**

Let \( * \) note any binary operator. Recall these properties operators:

- * is associative, iff \( (a \ast b) \ast c = a \ast (b \ast c), \) for all \( a, b \) and \( c \)
- * is symmetric, iff \( a \ast b = b \ast a, \) for all \( a \) and \( b \)

“iff” is short for “if and only if”. Symmetric is also called **commutative**.
Given an array \( b[0; n) \) and any associative and symmetric operator with neutral value \( u \) that is defined for the element type of \( b \). Then the operator may be generalised to a quantifier:

\[
(* \ i \ | \ 0 \leq \ i < \ n \ : \ b[i] \) = (b[0] \ * \ b[1] \ * \ ... \ * b[n-1])
\]

and if \( n = 0 \) (empty range):

\[
(* \ i \ | \ 0 \leq \ i < 0 \ : \ b[i] \) = (* \ i \ | false : b[i]) = u
\]

For instance, the minimum operator \( \downarrow \) on integers (e.g. \( 7 \downarrow 17 = 7 \)) is associative: \( a \downarrow (b \downarrow c) = (a \downarrow b) \downarrow c \) and symmetric: \( a \downarrow b = b \downarrow a \).

It would seem natural to generalise the minimum operator to a quantifier, so for instance \( (b[0..n]) \) is an array of integers):

\[
(\downarrow \ i \ | \ 0 \leq \ i < n \ : \ b[i]) = \text{‘the smallest number in the array } b[0..n] \text{’}
\]

But what about identity for \( \downarrow \)? We want a value \( u \), so \( x \downarrow u = x \) for all \( x \). The largest possible value in the set of possible \( x \)’s (the integers) will do. In math, that value is infinity: ‘\( \infty \)’. So

\[
(\downarrow \ i \ | \ 0 \leq \ i < 0 \ : \ b[i]) = \infty
\]

In programming, we will need some representation of infinity (most programming languages provide some constant \( \text{maxInt} \) representing the largest possible integer).

Correspondingly, we can define quantifiers generalising the maximum operator on integers \( (\uparrow) \), the union and intersection operator on sets \( (\cup \text{ and } \cap) \).

**Exercise:**

What are the identities for \( \uparrow \), \( \cup \) and \( \cap \) respectively?

**Solution:**

\( \uparrow: -\infty \uparrow x = x \) for all integers \( x \)

\( \cup: \emptyset \cup x = x \) for any set \( x \) (\( \emptyset \) being the empty set)

\( \cap: U \cap x = x \) for any set \( x \) (\( U \) being the universe for the sets in question)

**Free and bounded variables**

So far we have been using \( i \) as a helper variable in the quantified expressions. Such a variable is called a **bounded variable**, since it is bounded to the quantifier and only defined inside the quantifier’s scope. This means that we can use any name for the variable and change the name as long as the name not is used in the expression already. E.g.:
(6) \[ (+ i \mid 1 \leq i < n : i) = (+ j \mid 1 \leq j < n : j) \]

The scope of the bounded variable is limited by the parenthesis that surrounds the quantified expression. This corresponds perfectly with local variables of a method (or even better: with a local variable defined inside a block of code, for instance a for-loop).

Variables that are not bound (e.g. \( n \) in (6)) are called \textit{free variables} and must be defined somewhere outside the quantified expression. In order to evaluate the quantified expression all free variables must be assigned values. Often, it is convenient to state (some of) the free variables explicitly:

(7) \[ \text{prime}(n) \equiv (\forall i \mid 1 < i < n : n \% i \neq 0) \quad \text{‘\%’ is the modulus operator} \]

This states that \( n \) is a prime, if \( n \) is not divisible by any other number than 1 and \( n \) itself. \( n \) is a free variable. If we evaluate \( \text{prime}(6) \), we will get the result \textit{false}: \( \text{prime}(6) \equiv \text{false} \).

Correspondingly:

(8) \[ \text{sum}(m, n) = (\Sigma i \mid m \leq i < n : i) \]

(8) sums the integers from \( m \) (inclusive) to \( n \) (exclusive). \( n \) and \( m \) are free variables, and

\[ \text{sum}(3, 8) = 3 + 4 + 5 + 6 + 7 = 25 \]

We can define

(9) \[ \text{sortUp}(n, b) \equiv (\forall i \mid 0 < i < n : b[i-1] \leq b[i]) \]

This predicate determines if the array \( b[0; n) \) is sorted in ascending order. \( n \) and \( b \) are free variables, and \( \text{sortUp}(5, [1, 3, 4, 6, 8]) \equiv \text{true} \).

Free variables of a quantified expression correspond perfectly with field variables (attributes) of a class and/or formal parameters of a method. For instance, in (7) and (8) the free variables could be implemented as parameters for methods, while in (9), \( n \) might be a parameter and the array \( b \) a field variable.

**Central rules in predicate logic**

As the propositional logic (e.g.: [1], Theorem 1, p. 67), the predicate logic has a number of central rules, which we shall state in the following:
Let * be any associative and symmetric operator with identity u:

- **Empty range:** 
  \[(\ast i \mid false : T(i)) = u\]
- **Split off term:** 
  \[(\ast i \mid 0 \leq i < n+1 : T(i)) = (\ast i \mid 0 \leq i < n : T(i)) \ast T(n)\]
- **Distributivity:** 
  \[(\ast i \mid R(i) : T(i)) \ast (\ast i \mid S(i)) = (\ast i \mid R(i) : T(i) \ast S(i))\]
- **Range split:** 
  \[(\ast i \mid R(i) \lor Q(i) : T(i)) = (\ast i \mid R(i) : T(i)) \ast (\ast i \mid Q(i) : T(i))\]
- **De Morgan:** 
  \[-(\forall i \mid R(i) : T(i)) \equiv (\exists i \mid R(i) : \neg T(i))\]
  \[-(\exists i \mid R(i) : T(i)) \equiv (\forall i \mid R(i) : \neg T(i))\]

Split off term only applies when \(R(i) \land Q(i) = false\). Distributivity and range split only applies, if \(R(i)\) and \(Q(i)\) are finite sets, or if \(T(i)\) and \(S(i)\) are Boolean expressions.

We will examine the rules by looking at summation:

- **Empty range:**
  \[(\Sigma i \mid false : T(i)) = 0\]

As discussed earlier: summing nothing, yields 0.

- **Split off term:**
  \[(\Sigma i \mid 0 \leq i < n+1 : T(i)) = (\Sigma i \mid 0 \leq i < n : T(i)) + T(n)\]

Since addition is associative, one may sum the \(n\) first terms and then add the last term (e.g.: \(2+5+6 = (2+5)+6\)).

- **Distributivity:**
  \[(\Sigma i \mid R(i) : T(i)) + (\Sigma i \mid R(i) : S(i)) = (\Sigma i \mid R(i) : T(i) + S(i))\]

Again associativity gives that we can change the order of the operations (e.g.: \((2+4+7) + (1+5+9) = (2+4+7+1+5+9)\)).

- **Range split:**
  \[(\Sigma i \mid R(i) \lor Q(i) : T(i)) = (\Sigma i \mid R(i) : T(i)) + (\Sigma i \mid Q(i) : T(i))\]

If we want to sum all elements satisfying \(Q(i)\) or satisfying \(R(i)\), then we can sum all elements satisfying \(Q(i)\) and then sum all elements satisfying \(R(i)\) and finally adding the two sub results.

Finally, let’s take a look at De Morgan’s laws:

\[-(\forall i \mid R(i) : T(i)) \equiv (\exists i \mid R(i) : \neg T(i))\]
As an example, let the range be \([0; n]\) and the term even\(i\):

\[
\neg (\forall i \mid 0 \leq i < n : \text{even}(i)) \equiv (\exists i \mid 0 \leq i < n : \neg \text{even}(i))
\]

The left-hand side states that “it is not true that all numbers between 0 and \(n\) are even”. The right-hand side states that “there exist a number between 0 and \(n\) that is not even”. Obviously, the two statements are equal.

For the second law, we get:

\[
\neg (\exists i \mid 0 \leq i < n : \text{even}(i)) \equiv (\forall i \mid 0 \leq i < n : \neg \text{even}(i))
\]

The left-hand side states that “it is not true that there exists an even number between 0 and \(n\)”. The right-hand side states that “all numbers between 0 and \(n\) are not even”. Again, the two statements are obviously equal.

**Concluding remarks and examples from software development**

Predicate logic plays an important role in many areas of computer science. Later in this course we shall see how predicate logic is used in specification, construction and verification of algorithms. Also, specification and verification tools like Code Contracts (API for C#) and JML (Java Modelling Language) are based on predicate logic. Also, predicate logic is central in relational database theory (and practise); and SQL is partly based on it.

**An SQL example**

For instance, SQL support the existential quantifier (\(\exists\): EXISTS), but not the universal quantifier (\(\forall\)). A well-known work-around in SQL-programming is using “double NOT EXISTS” when one wants to express a “for-all” query (see [4]).

If we want to retrieve employees, who work on all projects (assuming the well-known Company database example from [4]), it may be done using a query like this:

```sql
SELECT E.LNAME, E.FNAME
FROM EMPLOYEE E
WHERE NOT EXISTS (SELECT *
                   FROM WORKS_ON B
                   WHERE NOT EXISTS (SELECT *
                                      FROM WORKS_ON C
                                      WHERE C.ESSN=E.SSN
                                      AND C.pno = B.pno)
                   )
```

This query may be translated into; “Select employees such that there does not exist a project that the employee does not work on”. This actually follows from De Morgan’s Law:

Let \(x\) be an arbitrary element in some set and \(p(x)\) a predicate stating some condition on \(x\):
De Morgan’s Law: \[ \neg(\exists x : p(x)) \equiv \forall x : \neg p(x) \]

Apply this to \( \neg p(x) \):
\[ \neg(\exists x : \neg p(x)) \equiv \forall x : \neg \neg p(x) \]

Reduce the right hand side, and we get:
\[ \forall x : p(x) \equiv \neg(\exists x : \neg p(x)) \]

So asking “is \( p(x) \) true for all \( x \)” is equivalent to asking “is it not true that there exists an \( x \) such that \( p(x) \) is not true”.

(So, be careful with double negations: “I don’t know nothing” actually means “I know everything”\(^\odot\)).

**Examples from the C# Collections library**

The C#/.NET Collections library actually supports many quantifiers. The quantifiers are implemented as higher order functions take the term as a lambda as parameter and assuming the range to be the collection.

For instance, if we want to sum the elements of an array \( b[0..n] \):

\[ \text{sum} = (\Sigma i \mid 0 \leq i < n : b[i]) \]

these lines of C# code will do the trick:

```csharp
int[] b = { 1, 2, 3, 4 };
Console.WriteLine("b.Sum(i => i): " + b.Sum( i => i ));
```

So we simply call the method `Sum` on the array with the term as a lambda as argument.

Similiarly, if we want the square sum:

\[ \text{squareSum} = (\Sigma i \mid 0 \leq i < n : b[i]^2) \]

we pass a lambda: \( x \Rightarrow x \times x \):

```csharp
Console.WriteLine("b.Sum(x => x * x): " + b.Sum(x => x * x));
```

Many other quantifiers are supported, for instance counting:

\[ (# i \mid 0 \leq i < n: \text{even}(b[i])) \]

In C#:

```csharp
Console.WriteLine("b.Count( i => i % 2 == 0): " + b.Count( i => i % 2 == 0));
```

The term \( \text{even}(b[i]) \) is implemented using the modulus operator ‘\%', if an even number is divided by 2, the remainder is 0; hence the lambda: \( i \Rightarrow i \% 2 == 0 \).
Also the existential and universal quantifiers are supported:

$$\exists i \mid 0 \leq i < n : \text{even}(b[i])$$

The existential quantifier is called Any in C#, so the predicate above becomes

```csharp
Console.WriteLine("Exists: b.Any( x => x%2 == 0): " + b.Any(x => x % 2 == 0));
```

and

$$\forall i \mid 0 \leq i < n : \text{even}(b[i])$$

becomes (the universal quantifier is called All)

```csharp
Console.WriteLine("ForAll: b.All( x => x%2 == 0): " + b.All(x => x % 2 == 0));
```

Also empty collections and neutral values are (partly) supported:

If we run this peace of code:

```csharp
int[] b = { };
Console.WriteLine("b.Sum(x => x * x): " + b.Sum(x => x * x));
Console.WriteLine("b.Count( i => i % 2 == 0): " + b.Count( i => i % 2 == 0));
Console.WriteLine("Exists: b.Any( x => x%2 == 0): " + b.Any(x => x % 2 == 0));
Console.WriteLine("ForAll: b.All( x => x%2 == 0): " + b.All(x => x % 2 == 0));
```

we get the expected output:

```
[0]
b.Sum(x => x * x): 0
b.Count( i => i % 2 == 0): 0
Exists: b.Any( x => x%2 == 0): False
ForAll: b.All( x => x%2 == 0): True
```

But be careful: not all quantifiers can handle empty range. For instance Min:

$$\downarrow i \mid 0 \leq i < n : b[i]$$

= ‘the smallest number in the array b[0..n]’

In C# this becomes:

```csharp
//int[] b = { };
int[] b = { 7, 6, 8, 5, 7, 7, 8, 9, 2, 3 };
Console.WriteLine("Min, blows up on empty collection: b.Min(): " + b.Min());
```

This works fine and yields the result 2. But if b is empty (commented out above), one will get an ‘System.InvalidOperationException’ thrown.

If you want to study the support of quantifiers in C# a bit more, then you could take a look at [5]. Experiment with the code. Use the documentation to investigate more quantifiers.
References:


