Programming Fundamentals

Dafny - week 7

A. Meijster

University of Groningen

university of groningen

Method contracts

```
method Triple(x: int) returns (r: int)
requires x%2 == 0
ensures r == 3*x
{
   var y := x/2;
   r := 6*y;
}

method Main()
{
    var y := Triple(14);
    assert y == 42;
    print y, "\n";
}
```

```
$ dafny triple.dfy
Dafny program verifier finished with 3 verified, 0 errors
```

university of groningen

Method contracts: specifications

- A method *contract* has two fundamental parts: a *precondition* and a *postcondition*.
- Together, they form the *specification* of the method.
- The precondition says when it is legal for a caller to invoke the method.
- This precondition is a proof obligation at every call site!
- The precondition of a method is defined via the `requires` keyword.
- The postcondition can be assumed to hold upon return from the invocation at the call site.
- The postcondition is a proof obligation for the implementer of the method.
- The postcondition of a method is defined via the `ensures` keyword.

- We can omit a precondition. This is equivalent with `requires true`.
- We can omit a postcondition. This is equivalent with `ensures true`.

university of groningen

(Under) specification

The specification of a method is a contract. It ensures that the postcondition holds after execution of the body starting from a state in which the precondition holds.

But, that says nothing about the implementation (of the body). This is *opaque*.
For example, consider the specification:

```
method Index(n: int) returns (i: int)
requires 1 <= n
ensures 0 <= i < n
```

Note that it is allowed (in contrast with C) to write `0 <= i < n`, meaning `0 <= i && i < n`.
The postcondition allows for several different correct implementations.

```
method Index(n: int) returns (i: int)
requires 1 <= n
ensures 0 <= i < n
{
  i := n/2;
}
```

```
method Index(n: int) returns (i: int)
requires 1 <= n
ensures 0 <= i < n
{
  i := 0;      // or i := n - 1
}
```

university of groningen

(Under) specification

Because the implementation of `Index` is opaque to the caller, the following `Main` method is not accepted by the verifier.

```
method Main()
{
  var x := Index(100);
  var y := Index(100);
  assert x == y;          // error
}
```

The verifier reports: `Error:  assertion might not hold`.

The problem is that `0 <= x < 100 && 0 <= y < 100` is not sufficient to conclude that `x == y`. The postcondition allows for several different correct implementations.

university of groningen

specification: multiple pre/post conditions

We want to specify a method that returns the minimum of two `int` arguments.
A first attempt may be:

```
method Min(x: int, y: int) returns (m: int)
ensures m <= x && m <= y
```

However, this is not sufficient. We also need to specify that the returned value is one of the inputs. We can use a second `ensures` line to specify this.

```
method Min(x: int, y: int) returns (m: int)
ensures m <= x && m <= y
ensures m == x || m == y
```

This is equivalent with:

```
method Min(x: int, y: int) returns (m: int)
ensures m <= x && m <= y && (m == x || m == y)
```

Similarly, you can have multiple preconditions.

university of groningen

Functions

A function maps its arguments to some value.

The key property of a function is that it is *deterministic*:
any invocation of the function with the same arguments result in the same value.

Here is an example function declaration in Dafny:

```
function Average(x: int, y: int): int
{
  (x + y)/2
}
```

Some important properties of funtions:

- A function has a result type, but no out-parameters.
- The body of a function is an expression, and contains no statements.
- Functions are *transparant*, while methods are *opaque*.
- Just like methods, functions can have preconditions.
- Functions can be used in methods and (other) functions.

university of groningen

Functions

Here is an example of a verified program that uses a function:

```
function Average(x: int, y: int): int { (x + y)/2 }

method Triple(x: int) returns (r: int)
requires x >= 0
ensures r == 3*x
{
    r := Average(2*x, 4*x);
}

method Main()
{
    var y := Triple(Average(14,7));
    y := y*Average(Average(11,13),1234);
    print y, "\n";
}
```

The example shows that we can call functions from methods. Moreover, functions are allowed to be part of expressions, because they are deterministic (no *side-effects*).

university of groningen

Predicates: functions returning a Boolean

Since pre/post-conditions are Boolean expressions, we will often use Boolean functions.

Boolean functions are called *predicates*.

Dafny has the keyword `predicate` which is reserved for defining Boolean functions.

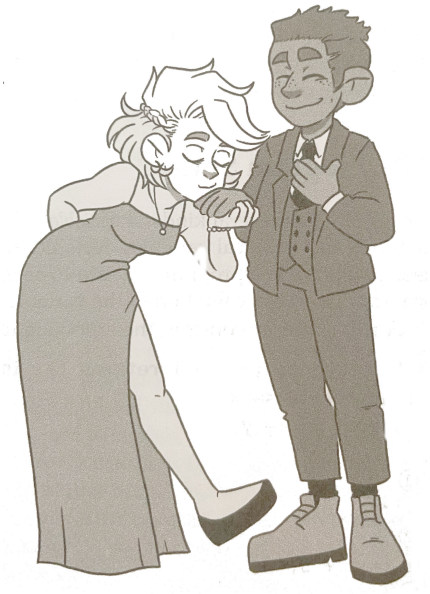
```
predicate IsEven(x: int) { x%2 == 0}
```

is identical to

```
function IsEven(x: int): bool { x%2 == 0}
```

university of groningen

Making it formal: Semantics



So far, we did not discuss how the verifier does its job.
To understand this, we have to define the *semantics* of program statements.

These semantics are mathematically described using *Floyd logic*. Floyd logic uses predicates to describe what is known before and after each statement.

It allows us to break down reasoning about programs into reasoning about individual statements of the program.

university of groningen

Program state

- An imperative program has several kinds of variables.
- These include the in- and out-parameters of methods, and the local variables declared in a method body.
- The variables that can be used at a point in a program are said to be *in scope*.
- The *state* at a program point is a *valuation* of the variables in scope at that program point.
- States can be described mathematically using predicates (e.g. `x==42 && y%2==0`).

```
method someMethod(x: int) returns (y: int)
requires x >= 10
ensures y >= 25
{
    // x >= 10
    var a := x + 3;
    //  x >= 10 && a == x + 3
    var b := 12;
    //  x >= 10 && a == x + 3 && b == 12
    y := a + b;
    // x >= 10 && a == x + 3 && b == 12 && y == a + b
    // equivalently: a == x + 3 >= 13 && b == 12 && y == a + 12
}
```

The implementation of the body is correct if and only if:
`a == x + 3 >= 13 && b == 12 && y == a + 12 ==> y >= 25`
This is clearly the case.

[Note: of course, a much simpler implementation of the body is `y := 25;`]

university of groningen

Program state

- The *annotation* of the program on the previous slide was performed top-down.
- After each assignment we wrote a predicate which holds **after** the assignment.
- To not lose information along the way, we wrote predicates that say all we know about the variables at each location. Such predicates are called the *strongest postconditions* of their preceding statements.
- However, strongest post conditions tend to grow quite a bit, and often carry more information than we actually need.
- In the example, we only needed in the end that `a >= 13 && y == a + 12` to conclude that `y >= 25`. For example, the conjunct that says that `b == 12` is not needed.

university of groningen

Program state

Often, we will reason about programs in the opposite direction (bottum-up).
To explain how this works, we introduce line numbers (not allowed in Dafny).

```
method someMethod(x: int) returns (y: int)
requires x >= 10
ensures y >= 25
{
1:     // x >= 10
2:
3:     var a := x + 3;
4:
5:     var b := 12;
6:
7:     y := a + b;
8:     // y >= 25
}
```

- If we want `y >= 25` at line 8, then we clearly need `a + b >= 25` at line 6.
- This means that we want `a + 12 >= 25` at line 4.
- On its turn, we want `x + 3 + 12 >= 25` at line 2.

What remains to show is that the predicate at line 1 implies the predicate at line 2.

In this case, that is trivial.

university of groningen

Program state

So, using a bottom-up approach, we found the following annotation.
It is significantly more compact then the one that was obtained using strongest postconditions.

```
method someMethod(x: int) returns (y: int)
requires x >= 10
ensures y >= 25
{
    // x >= 10
    // x + 3 + 12 >= 25
    var a := x + 3;
    // a + 12 >= 25
    var b := 12;
    // a + b >= 25
    y := a + b;
    // y >= 25
}
```

The bottom-up reasoning technique that we used for each assignment is to answer the question

" *which predicate P must hold such that predicate Q is true after the assignment?* "

In fact, to be more precise, we want predicate P to be as weak as possible. That predicate is called the *weakest precondition*. For example, `a + 12 >= 25 && b == 12` holds just before the assignment to `y`, but we only want the weakest predicate `a + b >= 25`.

university of groningen

Hoare triples

A program statement *S* is *correct* if it satisfies its *specification*.

We specify a statement *S* with a *Hoare triple*:

`{P} S {Q}`

This notation means:

"If we execute `S` *in a state in which precondition* `P` *holds, then it will terminate in a state in which postcondition* `Q` *holds."*

A few examples:

- `{ x == 0} x := x + 1; { x == 1}` clearly holds.
- `{ x == 0} S { x == 1}` specifies that `S` ensures `x == 1` if initially `x == 0` holds.
- `{ x == 21} x := 2*x; { x == 42}` clearly holds.
- `{ x == 0} x := 2*x; { x == 42}` clearly does not hold.
- `{ true } S { x*x == 5}` is unsatisfiable if `x` is an `int`.
- `{ false } S { Q }` is always satisfied (but useless), whatever `Q` is.
- `{ P } S { false}` is unsatisfiable, whatever `P` is (apart from `P == false`).

[Note: the book uses the notation `{{` and `}}`. I prefer to use `{` and `}` instead.]

university of groningen

weakest precondition of an assignment

Consider the following specification: `{ ?? } x := E; { Q }`.
Here `Q` is some known predicate, and `E` an expression.
We want to determine a predicate that we can place at the question marks.

If we want to establish `Q` after the assignment `x := E` then the *most general predicate* that needs to hold before the assignment is `Q` in which you replace *all* occurences of `x` with `E`.

We call this predicate the *weakest precondition* of the assignment `x:=E`.
We use the notation (used in the book) `Q[x:=E]` for this.
[Note: some authors (have) use(d) the notation Q_E^x, `[E/x]Q`, or even $Q_{E \to x}$.]

Hence, if some predicate `P` implies `Q[x:=E]`, then we can place `P` at the question marks. Of course, most of the time we simply choose `P = Q[x:=E]`.

university of groningen

Proof rule

In general, the notation $\mathcal{WP}$`(S, Q)` denotes the weakest precondition of statement `S` given postcondition `Q`.

So, $\mathcal{WP}$`(x:=E, Q) = Q[x:=E]`.

Soon, we'll see what the precondition of an if-statement is or a while-loop.

We have the following general proof rule:

wp proof rule

A statement `S` satisfies `{P} S {Q}` if and only if `P ==>` $\mathcal{WP}$`(S, Q)`.

For example, to prove that `{x == y} x := x + 1; {y == x - 1}` we need to show that `x==y ==>` $\mathcal{WP}$`(x:=x+1, y==x-1)`.
This is trivial because $\mathcal{WP}$`(x:=x+1, y==x-1) = y==x+1-1 = y==x`.

university of groningen

Additional proof rules

Strengthening of the precondition

From `P ==> Q` and `{Q} S {R}` we may conclude `{P} S {R}`.

In terms of $\mathcal{WP}$s we can rewrite this rule as follows:

Strengthening of the precondition

From `P ==> Q` and `Q==`$\mathcal{WP}$`(S,R)` we may conclude `{P} S {R}`.

Actually, this is just a reformulation of the wp-rule.

university of groningen

Additional proof rules

Weakening of the postcondition

From `Q ==> R` and `{P} S {Q}` we may conclude `{P} S {R}`.

In terms of $\mathcal{WP}$s we can rewrite this rule as follows:

Weakening of the postcondition

From `Q ==> R` and `P==>`$\mathcal{WP}$`(S, Q)` we may conclude `P==>`$\mathcal{WP}$`(S, R)`.

Or even simpler (and equivalent):

Weakening of the postcondition

From `Q ==> R` we may conclude $\mathcal{WP}$`(S, Q)==>`$\mathcal{WP}$`(S, R)`.

university of groningen

Additional proof rules

Sequential composition

From `{P} S {Q}` and `{Q} T {R}` we may conclude `{P} S; T {R}`.

In terms of $\mathcal{WP}$s we can rewrite this rule as follows:

Sequential composition

$\mathcal{WP}$`(S;T, Q)=`$\mathcal{WP}$`(S, `$\mathcal{WP}$`(T, Q))`

university of groningen

How does all this work in practice?

Well, let's have a look again at this annotated program fragment:

```
// x >= 10
// x + 3 + 12 >= 25 is wp(a:=x+3,25<=a+12)=wp(a:=x+3;b:=12;y:=a+b,y>=25)
var a := x + 3;
// a + 12 >= 25 is wp(b:=12, 25<=a+b)=wp(b:=12;y:=a+b, y>=25)
var b := 12;
// a + b >= 25 is wp(y:=a+b, y>=25)
y := a + b;
// y >= 25
```

Note that we could have simplified a predicate like `a + 12 >= 25` to `a >= 13`, but then it is harder to see that we applied the wp-rule in a completely mechanical fashion.

university of groningen

An other example: swapping variables

```
var tmp := x;
x := y;
y := tmp;
```

To specify what this program fragment does, we need a way in the postcondition to refer to the initial values of `x` and `y`. We do this by introducing *specification constants* (called *logical variables* in the book). As a convention, we will use upper-case letters for specification constants.
[I prefer the term specification constants, because they are constant (not variable!).]

Specification constants are not allowed to appear in the program, but they do appear in the proof.

Using specification constants `X` and `Y` we write the specification:

```
{x == X && y == Y}
var tmp := x;
x := y;
y := tmp;
{x == Y && y == X}
```

Here, we implicitly quantify over all possible values for `X` and `Y`. So, we should understand this as:

`∀X,Y:: {x == X && y == Y} tmp:=x; x:=y; y:=tmp; {x == Y && y == X}`

university of groningen

An other example: swapping variables

So, we need to show that

```
x == X && y == Y ==> WP(tmp:=x; x:=y; y:=tmp, x == Y && y == X).
```

The proof goes as follows (read from bottom to top to recognize the wp-rule!):

```
{x == X && y == Y}
var tmp := x;
{y == Y && tmp == X}
x := y;
{x == Y && tmp == X}
y := tmp;
{x == Y && y == X}
```

The top-down (strongest post condition) proof is more elaborate:

```
{x == X && y == Y}
var tmp := x;
{tmp == x && x == X && y == Y}
{tmp == X && x == X && y == Y}
x := y;
{tmp == X && x == y && y == Y}
{tmp == X && x == Y && y == Y}
y := tmp;
{tmp == X && x == Y && y == tmp}
{x == Y && y == X}
```

university of groningen

The tricky swap again

As you know, we can do without an extra variable (read again from bottom to top!):

```
{x == X && y == Y}
// arithmetic
{(x + y) - ((x + y) - y) == Y && (x + y) - y == X}
x := x + y;
{x - (x - y) == Y && x - y == X}
y := x - y;
{x - y == Y && y == X}
x := x - y;
{x == Y && y == X}
```

A version of this proof that is a combination of a top-down and bottom-up proof goes as follows:

```
{x == X && y == Y}
{x + y == X + Y && y == Y}
x := x + y;
{x == X + Y &&  y == Y}
{x == X + Y &&  x - y == X}
y := x - y;
{x == X + Y && y == X}
{x - y == Y && y == X}
x := x - y;
{x == Y && y == X}
```

university of groningen

Simultaneous assignment

Dafny allows several variables to be assigned in one statement.
This is called a *simultaneous assignment* (a.k.a. *concurrent assignment*).

As an example, `x,y := x + y, x - y` computes the sum of `x` and `y`, and the difference between `x` and `y`, and then updates `x` and `y` to these, respectively.

Hence, this statement is equivalent with

```
var x' := x + y; var y' := x - y; x := x'; y := y';
```

university of groningen

Simultaneous assignment

Let us investigate `{??} x,y := y,x; {x==X && y==Y}`.

The semantics of this simultaneous assignment are:
`{??} x':=y; y':=x; x := x'; y:= y'; {x==X && y==Y}`.

So, we compute $\mathcal{WP}$`(x':=y;y':=x;x:=x';y:=y';, x==X && y==Y)`.

```
   wp(x':=y;y':=x;x:=x';y:=y';, x==X && y==Y)
=
   wp(x':=y, wp(y':=x, wp(x:=x', wp(y:=y', x==X && y==Y))))
=
   wp(x':=y, wp(y':=x, wp(x:=x', x==X && y'==Y)))
=
   wp(x':=y, wp(y':=x, x'==X && y'==Y))
=
   wp(x':=y, x'==X && x==Y)
=
   y==X && x==Y
```

So, the simultaneous assignment `x,y := y,x;` swaps the variables `x` and `y`. We conclude:

```
{y==X && x == Y}
x,y := y,x;
{x==X && y==Y}
```

university of groningen

Simultaneous assignment

Indeed, in proving the correctness of programs that make use of the simultaneous assignment we could rewrite code in this style, and use the wp-rule.

But, it is better to introduce a separate rule for calculating the $\mathcal{WP}$ of simultaneous assignments.

$\mathcal{WP}$ of simultaneous assignments

$\mathcal{WP}$`(x,y,..:=E,F,..,  Q) = Q[x,y,..:=E,F,..]`

`Q[x,y,..:=E,F,..]` means `Q` in which we simultanaeously subtitute `E` for `x`, `F` for `y`, and so on.

university of groningen

Simultaneous assignment

Using this rule, we again investigate `{??} x,y := y,x; {x==X && y==Y}`.

```
  wp(x,y:=y,x;, x==X && y==Y)
=
  (x==X && y==Y)[x,y:=y,x]
=
  y==X && x==Y
```

So, again we conclude that `x,y := y,x;` swaps the variables `x` and `y`.

We can now write the following (silly program) to swap two variables:

```
method Swap(x: int, y:int) returns (x': int, y': int)
ensures x'==y && y'==x
{
  x',y' := y,x;
}

method Main()
{
  var x,y := 42,21;
  print "before swap: x=", x, " y=", y, "\n";
  x,y := Swap(x,y);
  print "after swap: x=", x, " y=", y, "\n";
}
```

university of groningen

Difference between `x,y:=y,x` and `x:=y;y:=x`

On the other hand, `x:=y; y:=x;` has a completely different effect.

```
   wp(x:=y;y:=x;, x==X && y==Y)
=
   wp(x:=y, wp(y:=x, x==X && y==Y))
=
   wp(x:=y, x==X && x==Y)
=
   wp(x:=y, x==X==Y)
=
   y==X==Y
```

This proves the correctness of the Hoare triple
`{y == Y}  x := y; y := x; {x == y == Y}.`
Clearly, this is equivalent with
`{y == Y}  x := y; {x == y == Y}.`

university of groningen

Deriving a statement

Given some assignments and a pre- and post-condition you now know how to verify correctness.

But, even nicer.....This technique can help you to derive the statements of a program fragment.

As an example, we will try to find a statement `S` that satisfies
`{x == y*y + X && y + 1 == Y} S {x == y*y + X && y == Y}.`
Here, `X` and `Y` are specification constants.

Comparing the pre- and post-condition we expect `S=(x := ??; y := y + 1).`

So, we compute $\mathcal{WP}$`(y := y + 1, x == y*y + X && y == Y).`

```
    wp(y := y + 1, x == y*y + X && y == Y)
=
    x == (y+1)*(y+1) + X && y + 1 == Y
=
    x == y*y + 2*y + 1 + X && y + 1 == Y
=
    x == y*y + X + 2*y + 1 && y + 1 == Y
```

Hence, we found the assignment `x := x + 2*y + 1` !

university of groningen

Deriving a statement

The formal annotation of the program fragment is (read bottom up!):

```
{x == y*y + X && y + 1 == Y}
// arithmetic
{x + 2*y + 1 == y*y + X + 2*y + 1 + X && y + 1 == Y}
// arithmetic: (y+1)*(y+1) == y*y + 2*y + 1
{x + 2*y + 1 == (y+1)*(y+1) + X && y + 1 == Y}
x := x + 2*y + 1;
{x == (y+1)*(y+1) + X && y + 1 == Y}
y := y + 1;
{x == y*y + X && y == Y}
```

university of groningen

Conditional Control Flow: if-then-else

Clearly, we need some **if-then-else** statement.

The syntax in Dafny looks like: `if B { S } else { T }`

Here `B` is a Boolean expression called the guard (or test), and `S` and `T` are statements.

Note that (unlike in C), no parentheses are required that enclose `B` (but it is allowed).

A simple example that uses the conditional statement is:

```
method Maximum(x: int, y: int) returns (max: int)
ensures (max==x || max == y) && max>=x && max>=y
{
  if x >= y
  {
    max := x;
  } else {
    max := y;
  }
}
```

university of groningen

Conditional Control Flow: if-then-else

We zoom in on the semantics of the **if-then-else** statement.

```
// here U holds
if B {  // note the layout
  // here V holds
  S
  // here X holds
} else {
  // here W holds
  T
  // here Y holds
}
// here Z holds
```

Floyd logic tell us that this annotation is correct if and only if

- `U && B ==> V`
- `U && !B ==> W`
- `{V} S {X}`, so `V ==> wp(S,X)`
- `{W} T {Y}`, so `W ==> wp(T,Y)`
- `X ==> Z`
- `Y ==> Z`

In practice, we will often use:

- `V = B && U`
- `W = !B && U`
- `Z = X || Y`

university of groningen

Conditional Control Flow: if-then-else

So, in practice we will often use:

```
// here P holds
if B {  // note the layout
  // here B && P holds
  S
  // here Q holds
} else {
  // here !B && P holds
  T
  // here Q holds
}
// here Q holds
```

Floyd logic tell us that this annotation is correct if and only if

- `{B && P} S {Q}`, so `B && P ==> wp(S,Q)`
- `{!B && P} T {Q}`, so `!B && P ==> wp(T,Q)`

So, the weakest precondition of a conditional statement is:
$\mathcal{WP}$`(if B then S else T, Q) = [(B ==>` $\mathcal{WP}$`(S, Q)) && (!B ==>` $\mathcal{WP}$`(T, Q))]`

In terms of Hoare triples: `{P} if B then S else T {Q}` if and only if
`P ==>` $\mathcal{WP}$`(if B then S else T, Q).`

university of groningen

Example: if-then-else

This seems very abstract, but in practice it is not difficult.

In my view, the use of ASCII notation obscures the readability of annotations a lot. So, for the sake readability I will switch to a more common mathematical notation in the slides, while we keep the ASCII notation in actual Dafny programs.

From now on, in the slides I will use the regular mathematical symbols $\wedge$ for `&&`, $\vee$ for `||`, $\neg$ for `!`, and $\Rightarrow$ for `==>`.

Moreover, in slides I will use the keywords **begin** and **end** for blocks that are enclosed by the curly braces `{` and `}`. This makes that we can use the curly braces for comment lines denoting the validity of some predicate.

So, using this notation the proof rules for a conditional statement now looks like this:

From $\{P \wedge B\}\ S\ \{Q\}$ and $\{P \wedge \neg B\}\ T\ \{Q\}$ we may conclude

$\{P\}$ **if** B **then** S **else** T **end** $\{Q\}$

From $\{P \wedge B\}\ S\ \{Q\}$ and $\{P \wedge \neg B\}\ T\ \{R\}$ we may conclude

$\{P\}$ **if** B **then** S **else** T **end** $\{Q \vee R\}$

university of groningen

Example: if-then-else

We aim for a statement S satisfying
$\{P :\ x$ **max** $y = X \wedge x$ **min** $y = Y\}\ S\ \{Q : x = X \wedge y = Y\}$.

$\{P :\ x$ **max** $y = X \wedge x$ **min** $y = Y\}$
if $x < y$ **then**
  $\{P \wedge x < y\}$
    $(*\ P$ and calculus $*)$
  $\{y = X \wedge x = Y\}$
  $z := x;$
  $\{y = X \wedge z = Y\}$
  $x := y;$
  $\{x = X \wedge z = Y\}$
  $y := z;$
  $\{Q : x = X \wedge y = Y\}$
else
  $\{P \wedge \neg(x < y)\}$
    $(*\ \neg(x < y) \equiv y \leq x$; use P and calculus $*)$
  $\{Q : x = X \wedge y = Y\}$
  skip; $(*$ do nothing $*)$
  $\{Q : x = X \wedge y = Y\}$
end $(*$ collect branches $*)$
$\{Q : x = X \wedge y = Y\}$

university of groningen

Match statement

In C we used the `switch` statement to avoid deeply nested `if-else if-else if ...` constructs.

Dafny has a more powerful construct, called `match`.
For the time being, we will use it as a `switch` like statement, but later in the course you will see that `match` is more powerful.

```
match grade
{
  case 10 => print "Outstanding\n";
  case 9  => print "Excellent\n";
  case 8  => print "Very good\n";
  case 7  => print "Good\n";
  case 6  => print "Satisfactory\n";
  case _  => print "Unsatisfactory\n";
}
```

Note that, in constrast with C, there is no fall-through (and therefore no `break`). The default case is written using an underscore (_).

university of groningen