EYUP: Yorkshire Computing

Introduction

EYUP is the programming language of choice for Yorkshire computing. It is simple, honest, straightforward and, when necessary, brutally forthright. EYUP is named after the famous Yorkshire greeting, but it is also an acronym for ENTIRE YORKSHIRE UNIVERSAL PROGRAM. Who said we're not ambitious? It's good to have ambition.

Interpreter

You will need an interpreter when you first encounter anything from Yorkshire. EYUP is an interpreted programming language, which means that you can make it up as you go along and see the results. Assuming you have EYUP installed on your computer, you simply type `eyup`[1] to launch the Yorkshire environment (hereafter you are *in Yorkshire*), in which anything you type at the prompt is assumed to be an EYUP instruction. You may never want to leave Yorkshire, but if you do, the right instruction is `sithee`[2], although you may find that `sithi` also works.

Entering EYUP

```
> eyup
Enterin' Yorkshire v1.0 (areyt tyke!)

Gaffer> …
```

Exiting EYUP

```
Gaffer> sithee
Leavin' Yorkshire v1.0 (flippin 'eck!)

> …
```

The Gaffer

When you first enter Yorkshire, the default bodger[3] is known as the `Gaffer`. You interact by typing anything the `Gaffer` understands, which includes basic input and output, simple arithmetic, and simple commands to define, or use things. The following examples show how you interact with the `Gaffer`. Whenever the `Gaffer` is ready to receive an instruction, the caret is `Gaffer>`. Other lines are output by the `Gaffer` in response to instructions typed at the caret. Other text read in as user input typed at the keyboard is shown in italic.

Simple output

```
Gaffer> write("Eyup, world!")
Eyup, world!
"Eyup, world!"
```

[1] Eyup is the universal greeting in Yorkshire.
[2] Sithee is the universal farewell in Yorkshire.
[3] Bodgers are different kinds of workers, described later.

Data definition

```
Gaffer> summat name: Script
Gaffer.name
```

Simple input

```
Gaffer> read(name)
John
"John"
```

Program definition

```
Gaffer> fettle wassname giz
  summat name: Script
  name := prompt("Wass yer name? ")
  write("Eyup " $ name $ ", areyt?")
oer
Gaffer.wassname
```

Program usage

```
Gaffer> wassname
Wass yer name? John
Eyup John, areyt?
nowt
```

More Bodgers

EYUP programs are organised around workers called bodgers. Every bodger knows how to do a particular job, and does it rather well; you need to know your bodgers to get anything done at all. You met the default bodger, the `Gaffer`, above. To call on the services of any bodger, you simply hail them using **`eyup`** followed by the bodger's name; and when finished, you bid the bodger farewell using **`sithee`**.

You can tell which bodger is listening to you by the name at the caret. When you hail a bodger, it will greet you and the name at the caret changes. When you release a bodger, it bids you farewell and you go back to your previous bodger. If this is the `Gaffer`, then you cannot go back any further (but you can leave Yorkshire).

Hailing a bodger

```
Gaffer> eyup TrigMath
TrigMath: eyup

TrigMath> pi
3.1415926535

TrigMath> sine(pi/2)
1.0
```

Releasing a bodger

```
TrigMath> sithee
TrigMath: sithee

Gaffer> …
```

Some useful bodgers

- `TrigMath` – defines the constant `pi` and standard trigonometric functions.
- `LogMath` – defines Euler's constant `e`, logarithm and exponential functions.
- `PolyMath` – defines polynomial powers and roots of different degrees.

Summat

Stuff is known as **`summat`**[4] in Yorkshire. This is the keyword used to define stuff used in EYUP programs. There are many different kinds of **`summat`**. Some commonly used types are:

- `Answer` – the two constants **`aye`** and **`nay`**;
- `Number` – any kind of integral or decimal number; e.g. `7, 3.14, -22, 0.5`
- `Letter` – any kind of character (Unicode compliant); e.g. `'a', 'c', 'A', '3'`
- `Bodger` –any kind of bodger; eg. `'Gaffer', 'TrigMath', 'PolyMath'`
- `List` – any kind of collection of other types; eg. `[1,2,3,4], ["John", "Sally", "Bill"]`
- `Script` – any kind of sequence of characters or text; e.g. `"Smith", "word"`

The Gaffer already knows about these types, so you can use them directly.

Declaring summat

You declare **`summat`** by telling your bodger about it. The bodger then has a definition of what you just declared, and can recognise it again. You need to tell your bodger the *name* by which you wish to refer to this thing, and say *what type* it is. Assuming you are talking to the Gaffer, you can declare a number like this:

```
Gaffer> summat weekday: Number
Gaffer.weekday
```

The Gaffer responds by telling you it has recognised the name `weekday`. The response indicates that `weekday` is known to, and is owned by, the `Gaffer`. If you forget to say what type it is, the Gaffer just assumes it is of the most general kind, called `Summat`.

Evaluating summat

To find out what **`summat`** currently refers to, you simply enter its name:

```
Gaffer> weekday
nowt
```

This may come as a surprise. The constant **`nowt`** means nothing. When you just declared `weekday`, it had no value whatsoever. Everything you declare initially has the value **`nowt`**; this is the undefined value for all kinds of stuff.

Assigning a value to summat

However, once you have declared `weekday`, you can assign a value to it. The assignment operator is the symbol `:=` and it associates the name to its left with the value to its right:

```
Gaffer> weekday := 7
7
```

[4] Summat means something in Yorkshire.

The Gaffer has now associated the value 7 with the name weekday. The response from the Gaffer is the new value just assigned to weekday.

Defining summat

In order to speed up this process, you can associate an initial value with **summat** at the same time as you declare it. We call this *defining* **summat** (as opposed to just *declaring* **summat**). In this case, you just assign an initial value to the name and the Gaffer works out what kind of stuff this is, from the kind of value that you assign:

```
Gaffer> summat weekday := 7
Gaffer.weekday
```

The response shows that the Gaffer now understands the name weekday. This name is associated with the value 7. You can change this again later, by assignment, if you wish. The Gaffer also knows that the name weekday is of the kind Number, because the assigned value 7 is of this kind (if not told otherwise, the Gaffer assumes they have the same kind).

Unknown summat

You can of course confuse[5] a bodger by entering a name that it knows nothing about:

```
Gaffer> month
Flummoxed: weerz month?
```

This indicates that the Gaffer knows nothing about any name called month. However, once a value has been associated with a name, you can enter its name to see its current value:

```
Gaffer> weekday
7
```

Wrong kind of summat

The kinds of **summat** that you declare prevent you from mixing stuff of the wrong kind. For example, if you declare weekday is a Number, then assign some Script:

```
Gaffer> weekday := "Wednesday"
Vexed: weekday wi' bad'un "Wednesday":Script
```

The Gaffer is annoyed[6], since it cannot make a Script fit[7] where a Number was expected.

Forgetting summat

Once you declare or define **summat**, it exists permanently within the scope of your bodger until you tell it to forget the definition. This can be useful if wish to change the declaration or definition. Let's assume that you want to change weekday to be of the type Script, rather than Number:

```
Gaffer> forget weekday
weekday

Gaffer> summat weekday := "Monday"
Gaffer.weekday
```

[5] Flummoxed means confused; weerz is asking for where something is, since the Gaffer has no idea.

[6] Vexed means annoyed, bad-tempered and generally miffed about something.

[7] Bad'un means a no-good value; the Gaffer could not make "Wednesday" fit in a Number.

If you did not forget the old definition first, you would confuse the `Gaffer` if you tried to associate a new definition that contradicted the old one.

Fettling Programs

Fettling[8] is what bodgers do all the time: fettling stuff means making stuff, or making stuff work. The keyword **`fettle`** defines a program in EYUP. Each program belongs to a particular bodger. By fettling a new program, you extend the capabilities of your bodger.

Trivial program

Here is the simplest, most trivial program you can create. It is called `doNowt` because it does nothing at all. You define the program by the keyword **`fettle`** followed by the program name `doNowt` and then the **`gioer`** keyword[9] (which is a contraction of **`giz`** and **`oer`**)[10]:

```
Gaffer> fettle doNowt gioer
Gaffer.doNowt
```

The `Gaffer` responds, saying that it has recognised, and now owns, the name `doNowt`. You can see what the result of this program is, if you enter its name again at the `Gaffer` caret:

```
Gaffer> doNowt
nowt
```

Because this program did nothing, the response was **`nowt`**. This stands for the empty or undefined result. In general, every program has some kind of result, even if this is **`nowt`**.

Non-trivial program

You can specify that a program accepts input arguments, or generates an output result, by declaring these after the **`fettle`** keyword in the signature for the program. You declare arguments with a *name* and *type*, similar to the style used with **`summat`**. You can declare multiple arguments of the same type together:

```
Gaffer> fettle addUp(n1, n2: Number): Number giz
  addUp := n1 + n2
oer
Gaffer.addUp
```

The declarations between parentheses `()` are the *formal arguments*, declared with names and types. You declare the result type of the program after the parentheses. Following this, you define the body of the program in a *block*, delimited by the keywords **`giz`** and **`oer`**. This block contains a single statement, which says that the result of `addUp` is computed by adding together the two arguments. You can see what the result of this program is, for different inputs:

```
Gaffer> addUp(2, 3)
5

Gaffer> addUp(4, 7)
11
```

[8] Fettle means to make, fix, or tidy up in Yorkshire.

[9] Gioer means stop immediately.

[10] Giz means give us something; and oer means over, done or finished.

The different sets of inputs `(2, 3)` and `(4, 7)` are the *actual arguments* supplied to different executions of the program `addUp`. The program assigns these values to the *formal arguments* `(n1, n2)` before computing the result.

Reflexive programs

As well as passing back computed results, a program can pass back the bodger who executed the program. This uses the keyword **`missen`**[11] to denote the executing bodger. The following defines summat called `total` for the `Gaffer` and then fettles `add` to accumulate values in the total, but the result is **`missen`**, the current bodger:

```
Gaffer> summat total := 0
```

```
Gaffer> fettle add(n: Number) : Gaffer giz
  total := total + n
  add := missen
oer
```

This allows the same bodger to execute a sequence of requests. The result of this program is in fact the `Gaffer` 'issen![12] This allows the programmer to stack up requests to the same `Gaffer`:

```
Gaffer> add(2).add(3).add(7)
gaffer
```

```
Gaffer> total
12
```

When executing the first add request, this adds `2` to the total and returns the `Gaffer`. The second add request asks the same `Gaffer` to `add(3)`; and the third add request asks the same `Gaffer` to `add(7)`. At the end of the sequence, the `total` contains `12`. The result `gaffer` is the value returned by the program[13].

Program misuse

You can confuse your bodger by using a program with too few arguments[14]:

```
Gaffer> addUp(3)
Flummoxed: addUp bah't n2:Number
```

You can confuse your bodger by using a program with too many arguments[15]:

```
Gaffer> addUp(3, 4, 5)
Flummoxed: addUp wi' traipsin' 5:Number
```

Flummoxing is just accidental misuse; but you can annoy your bodger, by trying to do something illegal. For example, you can *vex* your bodger by supplying a value of the wrong type[16]:

```
Gaffer> addUp("Wednesday")
Vexed: addUp wi' bad'un "Wednesday":Script
```

---

11 Missen means myself in Yorkshire.

12 Hissen or 'issen means himself in Yorkshire.

13 The lowercase form always denotes the default instance of the capitalised bodger-type.

14 Bah't means without (the) next thing: the Gaffer expected another argument n2:Number.

15 Traipsin' means wandering about aimlessly: the Gaffer did not expect an extra 5:Number.

16 Bad'un means a no-good value; the Gaffer could not make "Wednesday" fit n1:Number.

In general, you should seek to avoid either flummoxing or vexing your bodger!

Program with side-effects

Programs may, or may not, choose to take arguments or give a result. Programs can also have side-effects by changing the values associated with different names, through assignment.

```
Gaffer> fettle midweek giz
  weekday := "Wednesday"
oer
Gaffer.midweek
```

The program `midweek` assumes that the `Gaffer` already knows about a name called `weekday`. (If this were not the case, running the program would flummox the `Gaffer`). The body of the program assigns a new value `"Wednesday"` to the name `weekday`. However, this program does not give back any result:

```
Gaffer> midweek
nowt
```

Nonetheless, we can see that the value of `weekday` has changed, by entering this name to the `Gaffer` caret:

```
Gaffer> weekday
"Wednesday"
```

Program with local effects

Programs may change the values associated with names known to their bodger, or they may declare new names locally within the program. This is useful if you want to keep any changes local to the program, rather than make global changes.

```
Gaffer> fettle weekend: Script giz
  summat weekday := "Saturday"
  weekend := weekday
oer
Gaffer.weekend
```

The program `weekend` declares a new local name `weekday`, which happens to be the same as a name already known to the `Gaffer`. This local name is initially bound to the value `"Saturday"`. Finally, the result of the program `weekend` is bound to the value of `weekday`:

```
Gaffer> weekend
"Saturday"
```

```
Gaffer> weekday
"Wednesday"
```

This shows how the `weekend` program made changes to the locally declared `weekday`; but the version of `weekday` known to the `Gaffer` was not changed. Programs always use the most locally defined versions of names.

Program with input and output

Programs can read from the keyboard or write to the console. The `Gaffer` defines three programs to do this: `read`, `write` and `prompt`. A Yorkshire program[17] to greet someone looks like this:

```
Gaffer> fettle areytPal giz
  summat name: Script
  prompt("Giz yer name: ")
  read(name)
  write("Eyup " $ name $ ", areyt pal?")
  write("Aye, areyt thissen?")
  write("Aye, chuz pal!")
oer
Gaffer.areytPal
```

This is what you will see when executing the above program:

```
Gaffer> areytPal
Giz yer name: John
Eyup John, areyt pal?
Aye, areyt thissen?
Aye, chuz pal!
nowt
```

The program `write` writes out whatever you include in the parentheses as output text, ending with a newline. The program `prompt` writes out whatever you include in the parentheses, but does not generate a new line. The program `read` reads whatever you type at the console until you hit enter, and stores what you typed in **`summat`** you declared earlier. We will revisit these programs later.

The output is a proper Yorkshire conversation, apart from the last line **`nowt`**, which indicates that the program has no result to give back. The local `name` was used to store the script `"John"` that was entered at the keyboard. This name was used later in the greeting.

Bodgers and programs

Above, we showed how you execute a program simply by typing the name of the program at the caret for the bodger owning the program. This means that if you want to run different programs, you have to know which bodger to ask. Earlier, we switched to another bodger:

```
Gaffer> eyup TrigMath
TrigMath: eyup

TrigMath> pi
3.1415926535

TrigMath> sine(pi/2)
1.0
```

This showed how we could ask the bodger `TrigMath` for the value of `pi` (**`summat`** declared as a named value); and then we asked for the `sine` of a number (where `sine` is a program created for `TrigMath` using the **`fettle`** declaration). Later, we will show how to use programs of other bodgers without having to switch from the main bodger.

[17] Thissen means yourself. Areyt means how are you? Chuz means cheers.

Standard Operations

You can do the obvious things with the different predefined kinds of **summat** that we introduced earlier. Each type comes with a number of standard operations. Each operation can only be used with values of the expected type. If an operation is applied to a value of the wrong type, or if it is applied to **nowt**, then this will typically confuse the bodger.

Answer

`Answer` is the type of any truth-value. There are two literal constants of this kind:

- **aye** – meaning yes, or true
- **nay** – meaning no, or false

You can declare names of the `Answer` type and associate them with the values **aye**, or **nay**. You can use three keyword operators with anything of the type `Answer`.

Where we have `a1, a2: Answer`, then:

- `a1` **and** `a2` – means the logical-and, or conjunction of `a1` and `a2`
- `a1` **or** `a2` – means the logical-or, or disjunction of `a1` and `a2`
- **not** `a1` – means the logical-not, or negation of `a1`

The meaning of these operators is as you would expect: the negation of **aye** is **nay**, and vice-versa. Both operands of logical-and must be **aye** for the result to be **aye**. Either operand of logical-or may be **aye** for the result to be **aye**.

Number

`Number` is the type of any kind of number, whether integral, or with some kind of decimal precision. You can declare names of the `Number` type and associate them with literal values, which are written as you would expect: `3`, `-12`, `43.978` and so on.

Where we have `n1, n2: Number`, then:

- `n1 + n2` – means the sum of `n1` plus `n2`
- `n1 - n2` – means the difference of `n1` minus `n2`
- `n1 * n2` – means the product of `n1` times `n2`
- `n1 / n2` – means the quotient of `n1` divided by `n2`
- `n1 % n2` – means the remainder of `n1` modulo `n2`

The last operation only makes sense with integral numbers. Prefixing a number by the minus sign reverses the sign of the number. Mixing integral and decimal numbers is allowed; operations that return whole numbers are treated as integral. The multiplicative operators have higher precedence than the arithmetic operators; otherwise operators are evaluated in left-to-right order. Numbers may also be compared using the `Order` comparison operations[18].

Letter

`Letter` is the type of any single character. `Letter` is mainly used in conjunction with `Script`. Letters may also be compared using the `Order` comparison operations.

[18] The type `Order` is described later.

Script

`Script` is the type of any sequence of letters, or text. Like other sequences of things, it may use the subscripting operator `[]` to access, or replace individual letters. The indexing of letters starts with zero, the first index, and the last index is one less than the length of the sequence.

Where we have `s1: Script` and `c1: Letter`, then:

- `c1 := s1[0]` – stores the first letter from `s1` in `c1`
- `s1[0] := c1` – replaces the first letter in `s1` by `c1`

`Script` uses the `$` sticky operator for concatenating sequences. So long as the first operand is a `Script`, the second operand can be any type with a printable representation. Where we have: `s1, s2: Script, n1: Number, c1: Letter` and `a1: Answer`, then:

- `s1 $ s2` – means the result of concatenating `s1` with `s2`
- `s1 $ n1` – means the result of appending `n1`, as text, to `s1`
- `s1 $ c1` – means the result of appending `c1`, as text, to `s1`
- `s1 $ a1` – means the result of appending `a1`, as text, to `s1`

The result of concatenation is a new `Script`, that is, no operand is modified. Concatenation is evaluated in left-to-right order. `Script` texts may be compared lexicographically using the `Order` comparison operations, to determine which comes first in alphabetical order.

Order

`Order` is the type of anything that can be compared with other objects of the same kind. All of the standard types `Answer`, `Number`, `Letter` and `Script` have an order in this sense. There are six different ordering relationships altogether.

Where we have `o1, o2: Order`, then:

- `o1 < o2` – is **aye** if `o1` is less than `o2`
- `o1 > o2` – is **aye** if `o1` is greater than `o2`
- `o1 <= o2` – is **aye** if `o1` is not greater than `o2`
- `o1 >= o2` – is **aye** if `o1` is not less than `o2`
- `o1 = o2` – is **aye** if `o1` is equal to `o2`
- `o1 != o2` – is **aye** if `o1` is not equal to `o2`

If you compare things of unrelated types, this will flummox your bodger. Numbers are compared along the negative to positive axis. Letters are compared in alphabetic Unicode order (digits precede alphabetic letters; uppercase precedes lowercase). Scripts are compared lexicographically by determining whether their individual letters precede corresponding letters. For the purposes of comparison, **nay** precedes **aye**.

All types in EYUP are related in a hierarchy. The most general type[19] is `Summat`; and all other types are like some kind of `Summat`. `Order` is like a kind of `Summat` that can be compared. `Number` is like a kind of `Order` that can also do arithmetic. `Script` is like a kind of `Order` that can also do concatenation, letter search and letter replacement. The type `Nowt` is the least type, a kind of everything else[20].

---

[19] In other words, `Summat` is the top type.

[20] In other words, `Nowt` is the bottom type.

Program Control

The EYUP programs shown above only carried out a short sequence of instructions between the delimiting keywords **giz** … **oer**. More complex programs may execute different paths, depending on certain branching conditions; or they may repeat certain program sections multiple times, depending on a termination condition.

Conditional Branching

EYUP programs can execute different paths, depending on certain branching conditions. The simplest conditional branching form is delimited by keywords, and is one of:

- **if** … **then** … **else** … **oer**
- **if** … **then** … **oer**

The keyword **if** introduces a test condition, an expression which yields **aye** or **nay**. The keyword **then** introduces the instructions to carry out in the case of **aye**; and the keyword **else** introduces the instructions to carry out in the case of **nay**. The **else** branch is optional. The keyword **oer** terminates the conditional branching form, either after **else** in the case of a two-branch conditional, or after **then** in the case of a single-branch conditional.

An example program to find the maximum of two numbers looks like this; in fact this is how the program is defined for the bodger `PolyMath`:

```
PolyMath> fettle maximum(n1, n2: Number): Number giz
  if n1 < n2
  then maximum := n2
  else maximum := n1
  oer
oer
Gaffer.maximum
```

The program `maximum` has two branches, which happen to contain just one statement each. Statements in the **then** branch are terminated by the **else** keyword; statements in the **else** branch are terminated by the **oer** keyword. The final **oer** keyword terminates the **giz** … **oer** surrounding the program body.

A few examples of usage are:

```
PolyMath> maximum(3, 5)
5
```

```
PolyMath> maximum(11, -4)
11
```

```
PolyMath> maximum(7, 7)
7
```

`PolyMath` also defines a program called `minimum` to find the least of two numbers.

Conditional multibranching

If your program decision needs to have more than two branches, then it is possible to use multiple if-conditional forms. However, this can start to stack up the nesting of syntax. Imagine a program to decide if a number is negative, zero or positive:

```
Gaffer> fettle sign(n: Number): Script giz
  if n < 0
  then sign := "negative"
  else
    if n = 0
    then sign := "zero"
    else sign := "positive"
    oer
  oer
oer
Gaffer.sign
```

This is perfectly sensible, but requires an increasing number of **oer** keywords to close each of the nested conditional forms (**oer** is always needed because a *sequence of statements* can follow **then** and **else**). For convenience, EYUP provides the alternative conditional multibranching form:

- **if** … **then** … **when** … **then** … **else** … **oer**

This allows you to test a series of related conditions within one form. Each new condition is introduced by the keyword **when**. The keyword **then** introduces the instructions to carry out in the case of **aye**. In the case of **nay**, the program will jump to the next **when** condition. This continues until the **else** keyword introduces the last set of instructions to carry out if none of the conditions answered with **aye**. We can write the above program more simply as:

```
Gaffer> fettle sign(n: Number): Script giz
  if n < 0
    then sign := "negative"
  when n = 0
    then sign := "zero"
  else sign := "positive"
  oer
oer
Gaffer.sign
```

The **when** … **then** parts are repeated as many times as needed.

Conditional repetition

EYUP programs can repeat certain program sections multiple times, depending on a termination condition. The conditional repetition form is delimited by keywords and is one of:

- **while** … **gowon** … **oer**
- **gowon** … **while** … **oer**

They keyword **while** introduces an exit-condition[21], an expression which yields **aye** or **nay**. If the exit condition is **aye**, then the repetitions stop. The keyword **gowon** introduces the block of repeated statements[22]. The keyword **oer** terminates the conditional repetition.

We use the **while**-first form, if an exit-condition needs to be tested before any of the repeated statements are executed; in this way, we can repeat the statements zero to many times. We use the

[21] Wait while 5 o'clock means wait *until* 5 o'clock in Yorkshire.

[22] Gowon means go on, or carry on, in Yorkshire.

gowon-first form, if the repeated statements have to execute at least once. In Yorkshire, **while** introduces an *exit-condition* (not a *continuation-condition*), which EYUP programmers should keep in mind; otherwise bad accidents have been known to happen[23]. If in doubt, re-read this paragraph while yer gaum[24] it!

An example program to find the sum of a series of positive numbers looks like this:

```
Gaffer> fettle sumSeries(n: Number): Number giz
  summat total := 0
  while n = 0
  gowon
    total := total + n
    n := n -1
  oer
  sumSeries := total
oer
Gaffer.sumSeries
```

The program `sumSeries` expects some positive `n` and adds this to a local `total`, then subtracts one from `n` on each repetition, *until* `n` reaches zero. The result of `sumSeries` is the accumulated `total`. Of course, if you give the program zero, it stops immediately; and if you give it negative `n`, then it runs away forever.

A few examples of use are:

```
Gaffer> sumSeries(3)
6

Gaffer> sumSeries(4)
10

Gaffer> sumSeries(0)
0

Gaffer> sumSeries(-1)
Flippin 'eck: sumSeries weerz tha bin?
```

The last example runs away until the EYUP interpreter decides that it has waited long enough for `sumSeries` to give a result[25]. We could also guard against this by putting a condition into `sumSeries` stating that if `n < 0` then `sumSeries` should give up immediately.

Signalling mistakes

Sometimes it is not possible to carry on executing a program with an incorrect value. In this case, you can signal the mistake at the start and cause the program to stop. The keyword **wang** specifies the error-condition that should halt the program[26]:

```
Gaffer> fettle sumSeries(n: Number): Number giz
  wang n < 0
  summat total := 0
```

---

[23] Faulty "Wait while the lights flash" signs cause many car/train collisions in Yorkshire.
[24] Gaum means to understand or comprehend in Yorkshire.
[25] Weerz tha bin means something like where have you been, in Yorkshire.
[26] Wang means to throw, as in chucking a stone, in Yorkshire.

```
  while n = 0
  gowon
    total := total + n
    n := n -1
  oer
  sumSeries := total
oer
Gaffer.sumSeries
```

In this case, calling the program `sumSeries` with a negative value immediately halts, signalling the error back to the programmer[27]:

```
Gaffer> sumSeries(-1)
Vexed: sumSeries wi' n:Number < 0
```

This is better than allowing the program to run away. Note how the bodger converted the *wanged* condition into a useful error message. A bodger can be *flummoxed* if it doesn't understand your intentions; it can be *vexed* if you give it something obviously wrong to do; and it can exclaim *flippin 'eck* if it gets into a mithering tangle over something.

Building a Bodger

EYUP programmers start by interacting with the `Gaffer` and other bodgers at the command line, but later they go on to build their own bodgers. You can create a bodger interactively at the command line, using the keyword **`bodger`** followed by the name of the new bodger. This has to be a name that is not in use by any other bodger. Bodger names start with an uppercase letter.

Creating a new bodger

The following interactions create an empty bodger called `Circle`. Afterwards, the programmer switches to this new bodger, ready for the next steps.

```
Gaffer> bodger Circle
Gaffer: Circle

Gaffer> eyup Circle
Circle: eyup

Circle>
```

Defining bodger features

We wish to add a `radius` to the circle; and then we wish to be able to set the value of this `radius` to any value we like:

```
Circle> summat radius: Number
Circle.radius

Circle> fettle setRadius(n: Number): Number giz
  setRadius := radius := n
oer
Circle.setRadius
```

[27] Vexed means something like annoyed, frustrated or worried.

The program setRadius assigns the value n to the radius; and the program returns the same value, by convention. Note that multiple assignments can appear on one line. Assignments evaluate from right to left; that is, firstly the radius is set to n and secondly the result of setRadius is set to the radius.

Defining a worker bodger

We now wish to calculate other properties of circles, which happen to make use of the mathematical constant pi. Earlier, we said that the bodger TrigMath defines this constant. Rather than define the constant pi again, the correct style is to use the pi defined in TrigMath. Therefore, we first need to get hold of a local copy of TrigMath and add this to our Circle bodger.

```
Circle> summat trig := eyup TrigMath
Circle.trig
```

This defines a name trig, which is bound to a fresh copy of the bodger TrigMath. The expression **eyup** TrigMath creates the copy, known as an instance of TrigMath, and assigns this to the local name trig. Circle knows that this name is of the kind TrigMath, by the rule of type inference described earlier. Now, trig refers to a *worker bodger*, that is, a bodger that is subordinate to our Circle *mester* bodger[28].

Using a worker bodger

The advantage of this is that we can now refer to the named properties of TrigMath through our local name trig. As an example of this, we define the programs perimeter and area for our Circle bodger:

```
Circle> fettle perimeter: Number giz
  perimeter := 2 * trig.pi * radius
oer
Circle.perimeter

Circle> fettle area: Number giz
  area := trig.pi * radius * radius
oer
Circle.area
```

Within the programs of Circle, we refer to the value of the constant *pi* by trig.pi, that is, the value of the name pi defined in TrigMath. The program perimeter computes the formula for the circumference $2\pi r$ and the program area computes the formula for the area πr^2.

Using the mester bodger

We can use the *mester* Circle bodger in the same way as any other bodger, by interacting with it on the command line. Of course, we need to set the radius to something valid first; otherwise, this will vex the Circle:

```
Circle> perimeter
Vexed: perimeter wi' radius:Number = nowt

Circle> setRadius(5)
5
```

[28] Mester means master, typically in the sense of a master craftsman.

```
Circle> perimeter
31.4159265358

Circle>area
78.5398163397
```

Decimal numbers are accurate to about 15-16 significant figures, but display no more than ten decimal places by default, in the *Yorkshire* environment. We have been interacting in a piecemeal fashion with our `Circle` bodger, at the command line. We will bid farewell to `Circle` for now, ready to demonstrate the next step.

```
Circle> sithee
Circle: sithee
```

Creating multiple bodgers

We may of course create a local instance of `Circle`, or indeed multiple instances, in our `Gaffer`. In this case, we use the same syntax as when creating worker bodgers, since these local `Circle` instances are workers for the `Gaffer`:

```
Gaffer> summat c1 := eyup Circle
Gaffer.c1

Gaffer> c1.setRadius(5)
5

Gaffer> c1.perimeter
31.4159265358

Gaffer> summat c2 := eyup Circle
Gaffer.c2

Gaffer> c2.setRadius(10)
10

Gaffer> c2.perimeter
62.8318530717
```

These local copies of `Circle` are known by the names `c1` and `c2` in `Gaffer`. They each have a separate state, as demonstrated by setting their radii to different values, and by observing the different results for `perimeter` for each instance.

Bodger Lore

We have learned about bodgers, but inevitably, there's more to tell. We have collected various nuggets of Bodger wisdom under this section, as there's no single unifying theme.

Mester bodger lifetimes

How long do bodgers live? The default bodger `Gaffer` and the various bodgers that come with the Yorkshire 1.0 installation of EYUP are always available. Technically speaking, the *mester* instance of the `Gaffer` is created when you enter the *Yorkshire* environment. When you hail another bodger interactively, this creates the *mester* instance of that bodger. These bodgers persist in the current environment until you exit Yorkshire.

This means, for example, that if you define **`summat`**, or **`fettle`** new programs for these bodgers in the current runtime session, then the definitions and programs will persist, until you exit Yorkshire.

If you bid farewell to a bodger, and later hail it again, the definitions you added in the current session will still be there. However, once you leave *Yorkshire*, any interactive changes will be lost, and the next time you enter *Yorkshire*, you will load the original versions of the bodgers.

Worker bodger lifetimes

When you create a local worker bodger, adding it as **`summat`** to a program, or to another *mester* bodger, the lifetime of this worker bodger depends on the lifetime of its owner:

- If you create the worker as **`summat`** in a *mester* bodger, it will exist until either you tell the *mester* to **`forget`** the worker; or until the *mester* itself is forgotten;
- If you create the worker as **`summat`** in a program, then it will exist for the scope of the program execution, and will cease to exist when the program finishes running.

Therefore, when you leave *Yorkshire* and forget all the *mester* bodgers, these will in turn forget all of their worker bodgers. The workers will only be recreated, if the *mester* definition file also redefines the workers. Similarly, even though a program forgets all of its local workers, when it finishes executing, it will recreate them each time that the program runs.

Persistent bodger changes

If a bodger has a *mester* definition file, then this determines what is remembered about the bodger between sessions in Yorkshire. To make permanent changes to a bodger, you should make the changes to the *mester* definition file for that bodger. When interacting with a *mester* bodger, the command **`remember`** may cause the *mester* definition file to be updated interactively[29]. Otherwise, it should be possible to write *mester* definition files from scratch and put these in a place that your EYUP installation recognises at start-up.

Bodger encapsulation

When you define **`summat`** for a bodger, you may want to control who has rights to change the value associated with the name. EYUP deals with this matter by the "talk to yer face" *rule of encapsulation*, which is simple compared to other programming languages[30]. The rule is that if you are talking to a bodger directly, then you may assign values to its names (providing they be of the right kind). This also applies to programs belonging to the bodger, which by definition talk to it directly:

```
Circle> radius := 7
7
```

Otherwise, if you are not talking directly, you may only *access* a value in a bodger; and it is strictly forbidden to *reset* it by remote assignment. If you do attempt this, the bodger will be vexed, saying that it encountered a malicious remote assignment[31].

```
Gaffer> c1.radius
5

Gaffer> c1.radius := 7
Vexed: faffin' wi' c1.radius
```

However if the `Circle` bodger provides a program `setRadius` to reset the value of the `radius`, this can be used. In this way, a bodger controls whether its local names can be reset.

---

[29] This feature may be available in some versions of EYUP, but depends on the version.

[30] Talk to yer face is an expression of a desire for honest communication in Yorkshire.

[31] Faffin' about means messing around, and faffin' wi' means fiddling with something, in Yorkshire.

Bodger constants

Sometimes **summat** should always have a constant value. For example, it would be bad if any program could alter the value of `pi` in `TrigMath` by re-assignment of a new value! It would also be bad if you could alter `pi` by talking directly to `TrigMath`. Fortunately, EYUP has a keyword **allus** meaning that something should always have a given value[32]:

```
TrigMath> summat pi := allus 3.1415926535
TrigMath.pi
```

This has the sense that `pi` is always set to the given value. This means that you cannot even alter the value of `pi` when talking directly to `TrigMath`:

```
TrigMath> pi := 2.7182818284
Vexed: faffin' wi' pi allus 3.1415926535
```

Inspecting a bodger

If you regularly add many new names to your bodger (noting that both **summat** and **fettle** add new names), and cannot remember which names you would rather keep and which you would rather forget, then you bodger may become cluttered and untidy. To find out what's in your bodger, you can always have a **gander** using this keyword[33]:

```
Circle> gander
Circle.radius
Circle.setRadius
Circle.trig
Circle.perimeter
Circle.area
```

This displays all the names known to the `Circle` bodger. If you want to know a bit more about any of these names, you can ask to **gander** the name directly:

```
Circle> gander radius
Circle.radius: Number

Circle> gander area
Circle.area: Number

Circle> gander setRadius
Circle.setRadius(n: Number): Number
```

This will remind you of the types attached to the names. Programs may display the types of values they expect, as well as the result type (programs with no result have the result type `Nowt`).

Tidying up a bodger

You could decide that you want to get rid of the radius-setting program, so that in future you can only create `Circle` workers with a `radius` supplied at the time of creation. To get rid of **summat**, you simply **forget** it:

```
Circle> forget setRadius
setRadius
```

[32] Allus means always in Yorkshire.
[33] Gander means to have a look, often in the sense of stickin' your neb in.

Now when you **gander** your bodger, you won't see `setRadius` any more:

```
Circle> gander
Circle.radius
Circle.trig
Circle.perimeter
Circle.area
```

If you forget useful programs, of course, you will have to start adding them to `Circle` all over again. As an exercise: how would you redefine the local worker `trig` so that no-one could change this worker again, even if talking to `Circle` directly?

Changes to the Initial concepts

Small changes

- Error messages might slightly differ and will produce a traceback.
- Prompt changed to no longer need the read keyword (as it is in python) – see above.
- If statements: when = elif in other languages => has to start with 'if' and branch that way due to errors I had with multiline
- Default types now Answer, Number, Letter, Script, List, Bodger, (Fettle -> must be defined as fettle...).

Omissions

- No snicket.
- No ginnels.
- No utensils – replaced by basic *list* type.
- No bodger initialisation.

Built-in functions and variables

In the Gaffer (and thus every bodger)

- nowt – nowt type in EYUP
- aye - EYUP's equivalent to True
- nay - EYUP's equivalent to False
- write() - function that prints to screen (args = [Script])
- prompt() - function that takes input from the user with (args = [*Script])
- isNumber() - function that returns aye if it's a number else nay (args = [summat])
- isScript() - function that returns aye if it's a script else nay (args = [summat])
- isList() - function that returns aye if it's a list else nay (args = [summat])
- isFettle() - function that returns aye if it's a fettle else nay (args = [summat])
- isLetter() - function that returns aye if it's a letter else nay (args = [summat])
- isBodger() - function that returns aye if it's a bodger else nay (args = [summat])
- isAnswer() - function that returns aye if it's a answer else nay (args = [summat])
- toNumber() - function that converts a valid script to a number (args = [Script])
- toScript() - function that converts default type into script (args = [summat])
- add() - function that adds an item to a list (args = [list, item])
- take() - function that takes an item out of a list (args = [list, index(number)])
- has() - function that returns aye if item is in the list and nay otherwise (args = [list, item])
- hasOwt() - function that returns aye if something is in the list and nay otherwise (args = [list])
- hasNowt() - function that returns aye if nothing is in the list and nay otherwise (args = [list])

- addAll() - function that adds all the items in one list into the other list (args = [list, list_of_items])
- takeAll() - function that takes all items from a list in another list (args = [list, list_of_indexes(numbers)])

In the TrigMath:

- pi – number 3.141592…
- sin() – get the sin of a number (args = [number])
- cos() – get the cos of a number (args = [number])
- tan() – get the cos of a number (args = [number])
- hypot() – get the sin of 2 numbers (args = [number1, number2])
- degrees() – convert radians number into degrees (args = [number])
- radians() – convert the degree number into radians (args = [number])
- asin() – get the arc sin of a number (args = [number])
- acod() – get the arc cos of a number (args = [number])
- atan() – get the arc tan of a number (args = [number])

In the LogMath:

- e – number 2.71828…
- log() – get the log (base e) of a number (args = [number])
- log2() – get log (base 2) of a number (args = [number])
- log10() – get log (base 10) of a number (args = [number])
- logBase() – get log (base passed in) of a number (args = [number, base])

In the PolyMath:

- sqrt() – square root of a number (args = [number])
- pow() – x^y of two numbers passed in (args = [number, number to raise to the power of])
- maxiumum() – get the maximum of two numbers (args = [number])
- minimum() – get the minimum of two numbers (args = [number])