Charm manual

Introduction

This is still a prototype, put out for comment. All the features described in this manual exist: a number of intended and desirable features don't yet. It has not been optimized at all, because I don't know what the final implementation will involve. (Benchmarking Charm at this point would be an act of aggression.)

Because the language is currently for discussion more than use, this is not a manual for complete beginners, but for people who've seen a programming language or two, who will be able to pick up the language mainly from examples, and can make useful comments.

Also, if this manual was written for general users I would have written a "standard prelude" for my users and discussed that *before* telling them anything about hand-rolled recursion. As it is, the sections on "recursion and how to avoid it" and the "prelude" come rather late in the manual.

The manual includes notes in pink for the language development community to explain what I'm trying to do and why I'm trying to do it. If you just want to see what Charm does and not why I did it that way, you can ignore them.

So, let's get started. Welcome to Charm!

And to the langdevs, welcome to the first pink note! What *am* I trying to do?

(1) I'm trying out a way of structuring code and dealing with state and purity and making functional programming more useful … based on the observation that people like databases and they like spreadsheets. If you read the manual bearing this in mind then you should see what I mean.

(2) Other ideas have been taken from the "data-oriented programming" paradigm, since that is obviously what I'm doing.

(3) I have reinvented the Functional Core / Imperative Shell paradigm and made it a fundamental part of the language.

(4) I don't know whether my ideas will scale. But in any case, what I'm doing with them right now is making a small, friendly, easy-to-learn dynamic scripting language with a *primary* use-case of letting people slap together small-to-medium CRUD apps easily and quickly. It's a

GPL, but that's what it'll do best, and my design choices have been based around this use-case. If it was a static language for people to write enterprise-scale software in, it would have interfaces and generics. As it is, it has ad hoc polymorphism. It isn't meant to do all things equally well, that way lies madness and C++. It'll be most helpful if you focus on the thing I'm trying to do rather than all the things I'm not trying to do.

Getting started

The source code is here. It's in Go, and you will need the latest (1.18) version of the Go compiler to use the nice new generics. It comes with demo scripts which I'll use to talk you through the language. Alternatively, here's some Mac OS object code for you to download, plus the demo scripts … assuming you have Mac OS and don't want to mess around with Go.

At present you will need to have Go installed to use some of Charm's libraries. I will shortly embed it to save you the trouble. In the meantime, you can do that or just stick to the core language for now.

Note for langdevs: why Go? I know some of y'all hate it and are already questioning my judgment, sanity, parentage, etc. Some reasons why I picked it: (1) if you write an interpreted language in Go, then it's automatically garbage-collected without you having to do anything; (2) the support for concurrency should make it really easy to exploit the purity of my functions; (3) general backend orientation suitable for my use-case; (4) Thorsten Ball's book *Writing An Interpreter In Go* is the business.

Let's assume you've done one of those things or the other and you now have an executable file and some accompanying folders

Because Charm is a REPL language, it will do quite a lot of things without any script at all (what I will call an "empty service"). If you just start it up with `./charm run` then the terminal will show you a logo and give you an arrow prompt. You're ready to start. (Note: in what follows I will use `Courier` for things done in the REPL and for filenames, and `Roboto Mono` to indicate a script, or generally to distinguish code from body text.)

```
Charm % ./charm run
```

→

Types, literals, and operators

`int`

There is an `int` type, of course, and its arithmetic operators work as you think they would:

```
→ 2 + 2
4
→ 1 + 2 * 3
7
→ -7 % 4
-3
→ 5 / 2
2
→ 5 / 0

[0] Error: division by zero at line 1 of REPL input.

→
```

I personally don't like that the % operator returns negatives but I will go for compatibility over my tastes. C did it, everyone else followed suit, and some battles aren't worth fighting.

`float`

Similarly there is a float type:

```
→ 3.14592 * 20.0
62.918400
→
```

`pair`

And a pair type, which has no built-in operators of its own except the one that creates it:

```
→ 1 + 2 :: "foo" + "bar"
3 :: foobar
→
```

We'll find a use for it in a minute.

Why the pair? As a form of syntactic and (so to speak) semantic sugar. It was originally my intention to use a single colon as syntax in slices and in giving key-value-pairs in map and struct literals. Because of the other things I'm doing with colons this would be slightly fiddly and would in very rare cases require the user to use parentheses to disambiguate the colon, but I was still going to do it. What turned me around was the realization that if I used a colon it could never be *more than* syntax: `foo : bar` can never refer to a first-class object. But this is very un-Charm-like. Hence the pair type: `foo :: bar` *does* refer to a first-class object. This is Charm-like and convenient and gives us some nice idioms some of which we will meet later.

`string`

Strings are in utf-8 and are indexed that way. Strings in double quotes have interpreted escape characters; strings in backtick quotes don't:

```
→ "♥Charm♥"[6]
♥
→ "Hello world!\nWhat a nice day!"
Hello world!
What a nice day!
→ `Hello world!\nWhat a nice day!`
Hello world!\nWhat a nice day!
→ "abcdefghijklm"[3::7]
defg
→ len "banana"
6
→
```

Note that here as elsewhere Charm is zero-indexed: ranges are from-including-to-not-including.

I don't like that. I think most languages should be one-indexed and from-including-to-including. But I also feel like it's too late to do anything about it now.

Slices give an out-of-range error when they're out of range, as in Go, rather than returning an empty string/list/whatever. Case for: noisy failure is better than silent failure. Case against: some recursive idioms would be easier if Charm did like Python. Rebuttal: they'd be easier still if the standard prelude included a few useful functions to do the same job.

`bool`

The `bool` type has constants `true` and `false`. The logical operators are `and`, `or` and `not`.

I've gone back and forwards on the operators but I feel like (a) Python has cleared the way (b) Charm gives a lot of facilities for making your code read as pseudocode if you really want to, and using && || and ! would obstruct that.

Evaluation of `and` and `or` is lazy.

```
→ false and 1 / 0
false
→
```

Besides this there is a conditional operator.

```
→ 2 + 2 == 3 : "Oops"; 2 + 2 == 4 : "Whew!"; else : "Who even knows?"
Whew!
→
```

Obviously conditionals are also lazily evaluated.

If you're wondering why I don't just use a ternary operator then (a) this will become obvious later (b) the ternary operator is dumb and I hate it.

Charm supports truthiness for the `int`, `string`, `set`, `list` and `map` types. It is also possible for users to make values truthy, as will be discussed later. Do this with caution.

I vaccilated on this one too, and could change my mind again. I am usually very much against implicit type conversion, but this one's useful. Making your own truthiness (see later) should be done with extreme caution. It should NOT be done just to give you a convenient boolean function on a type, but rather to single out as false that element or those elements of the type which naturally correspond to the termination of an iterative process. I have not for example implemented truthiness for floats, because hardly anyone iterates on floats, and when they do, they stop when they get within a certain margin of error, not when they reach 0.0.

`list`

Lists should look familiar …

```
→ [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
→ len [0, 1, 2]
3
→ ["marmalade", true, 42][2]
42
```

```
→ len [0, 1, 2, 3, 4, 5]
6
→ [0, 1, 2, 3, 4, 5][2::5]
[2, 3, 4]
→ 42 in ["marmalade", true, 42]
true
→
```

Comparison is done with the `==` operator:

```
→ 2 + 2 == 4
true
→
```

I've toyed with various ideas up to and including replacing `==` and `!=` with `is` and `isn't`, and yet here we are, conserving our strangeness budget.

Comparison of lists and of everything else is by value:

```
→ [3, [4, 5]] == [3, [2 * 2] + [5]]
true
→
```

Data-oriented programming, it's what the people want! according to my recent sample of 1.

`set`

```
→ {1, 2, 1 + 2} == {3, 2, 1}
true
→ len {1, true, "walrus"}
3
→ "walrus" in {1, true, "walrus"}
true
→
```

Yes, we get to use braces for sets like you always wanted to! In Charm, everything is an expression, and so the indents and outdents are syntactic sugar for ordinary parentheses.

`map`

Maps can have any basic noncompound type as a key: int, float, bool, string, etc, but not other maps, lists, tuples, sets or structs.

```
→ (map "foo"::1, 42::"albatross")["foo"]
1
→ (map "foo"::1, 42::"albatross")[42]
albatross
→ (map "foo"::1, 42::"albatross")["walrus"]

[0] Error: key is not in map at line 1 of REPL input.

→
```

`tuple`

Tuples are what one might call "flat tuples" or "mathematicians' tuples" or "autosplats":

```
→ 1, ((2, 3), 4)
1, 2, 3, 4
→ 3 in 1, 2, 3, 4
true
→
```

We had a whole thread on flat tuples on r/programminglanguages. To summarize my thinking:

(a) Lists exist for those that want nesting.

(b) I have been guided in my design choices by the idea that mathematicians are probably right in how they communicate, because they have no limits on how they do it and are driven only by questions of readability by humans. They don't "pass" flat tuples to their functions because unfortunately 'cos of a design flaw in the whiteboard that's the only way their equations will compile — they do it because that's how they make things clear to other people. (Note that MatLab has flat tuples for just this reason, it would be infuriating to mathematicians if it didn't.)

I'm with them. If I have a function `swap(x, y) : y, x` then I want `swap swap x, y` to be the identity function and not a syntax error.

(c) If not, *what happens to referential transparency*? In Python we have a situation where `swap(1, 2)` is `2, 1` but if `x = 1, 2` then `swap(x)` is an error. Well, Python has no referential transparency anyway but I do and will protect it. I don't want functions that are guaranteed 99% referentially transparent any more than I want them guaranteed 99% pure. Those guarantees are not something on which one can base an automated process.

(d) If you had to splat tuples you'd have to splat them a lot. Look at some of the example code later on and tell me that it would be better if you had to remember to manually splat your tuples each time.

Unlike `string`, `list`, `set`, etc, `tuple` does not have a `len` function. Instead, it has an `arity` function. To see why, consider the following:

```
→ len "banana"
6
→ arity "banana"
1
→
```

and also ...

Besides these there are various other useful types, including `struct`, `nil`, `label`, `error`, `func`, `type`, and `enum`. But for one reason or another it seems sensible to defer discussion of these until later in the manual.

Variables

So far this has looked a lot like, for example, the Python REPL, but you will find that you can't create variables:

```
→ x = 1

[0] Error: attempt to access the value of a private or non-existent
variable or constant 'x' at line 1:0-1 of REPL input.

→
```

Variables are created in the `var` section of the script. To see an example, look in the `src` subfolder and open up `src/variables.ch` in your favorite text editor/IDE, or just read the source code in full, here:

```
var

h = "Hello world!"
x = 2 + 2
y = x, h, 42
```

You can tell Charm to initialize it like this:

```
→ hub run src/variables.ch
Starting script 'src/variables.ch' as service '#0'.
#0 →
```

Note about the hub: I should briefly explain what just happened there. So far you have been talking to the "empty service" through an almost invisible intermediary called "the hub". `hub run src/variables.ch` *tells the hub to start up a new service initialized by the script* `src/variables.ch`*. And now you've started up an actual service with something in it, the hub wants it to have a name, and since you didn't give it one, it's provided it with a serial number* `#0` *instead. There's more about the hub at the end of this document, but for now it's the thing that runs scripts for you. Or you can use* `hub help` *if you want to read ahead.*

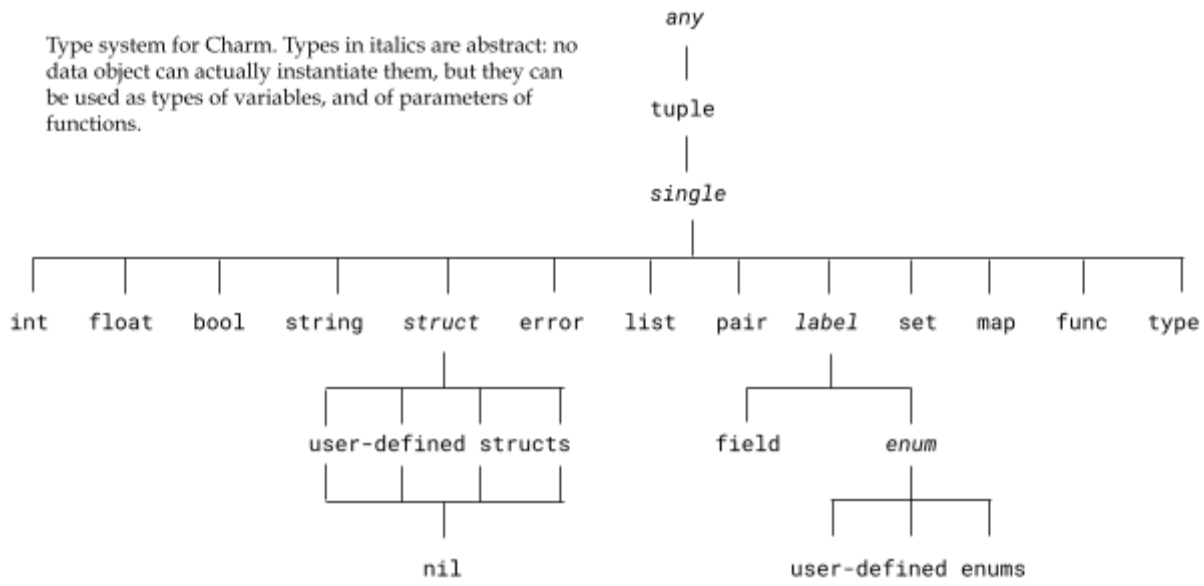
If you want to get your hands dirty by actually coding, the `hub why` *and* `hub trace` *commands will be useful for understanding errors.*

And now you've run the script, you have variables. You can now assign values to them and get values from them just as you'd expect:

```
#0 → h
Hello world!
#0 → x
4
#0 → y
4, Hello world!, 42
#0 → h[0]
H
#0 → x + y[2]
46
#0 → h = "Goodbye world!"
ok
#0 → h
Goodbye world!
#0 →
```

Type inference and how to avoid it

The variables are given types when they are assigned. By default, they are given the most specific type that can be inferred for them. Here is the type system as it stands at present, arranged from most general at the top to most specific at the bottom..



Hence in our example, `h` is of type `string`, `x` is of type `int`, and `y` is of type `tuple`.

However, we can explicitly declare a variable to be of a more general type than would be inferred. An example is given in the file `src/typing.ch`:

```
var

foo = "bananas"

zort single = "bananas"

troz any = "bananas"
```

If you run the script you will find that `foo` can only be assigned string values, that `zort` can be assigned anything but a tuple, and that `troz` can be assigned anything at all.

Most dynamic languages don't do that. Why did I? There are three things we can do if we want variables to be dynamic: have them all dynamic, have them dynamic by default, or have them dynamic by request. I've gone with the third option.

My thinking is that we rarely use the dynamism of dynamic variables *as such*. To truly exploit this feature we would be changing the type of a variable *at runtime*. And we do this sometimes, but what we mostly enjoy about dynamic languages is how easy it is to change the types of

things *between versions of our code*. And that is a result of type inference on the variables and of having untyped function parameters, not of the runtime dynamism.

For most variables, during any given runtime, we know what type they're intended to be, we intend them to be the same type throughout the runtime, and would like to be alerted if this goes wrong. And this will especially be the case in Charm, as you'll see. As this will usually be the case, this should be the default, so that's what I've done.

Comments and continuations

These are illustrated by the file `src/com&con.ch`. Comments are very much what you're used to from other languages. Continuations must be marked by a `..` at the end of the continued line and a corresponding `..` at the beginning. The allowed exception is that the continued line may end in a comma *where this is syntactic*, in which case the continuation must begin with `..` just the same. The continuations can be placed wherever is most readable: they are exempt from whitespace rules.

```
var

// This is a comment.

// And below, the two types of continuation.

x = "hello " + ..
 .. "world"

y = 1, 2, 3,
 .. 4, 5, 6
```

I haven't made many innovations in syntax. But this is a friendly one. Explicit is better than implicit.

Constants

We can make constants in the `def` section, as demonstrated in the file `src/constants.ch`:

```
var

radius = 10.0
circumference = 2.0 * pi * radius
```

```
def

pi = 3.141592
```

If we try it out …

```
#0 → hub run src/constants.ch
Starting script 'src/constants.ch' as service '#1'.
#1 → pi
3.141592
#1 → pi = 4

[0] Error: attempt to update a constant 'pi' at line 1 of REPL input

#1 → circumference
62.831840
#1 →
```

Another note about the hub. It can have as many services as you like open at once. Service `#0` *is still there, and you can still talk to it by prefixing what you say with* `#0`*.*

```
#1 → #0 h
Goodbye world!
#1 → h

[0] Error: identifier not found: 'h' at line 1 of REPL input

#1 →
```

By default, though, you're talking to the service named in the prompt. To switch which service this is, just type `hub <service name>`*. To get a list of the services being run and the scripts they're running, type* `hub list`*.*

You will notice that the order of declaration is free: `src/constants.ch` declares `pi` after `circumference`, and the initializer has no difficulty.

Yes, you could use that to write bad code, but you could also use it to write good code, by putting your constants at the end of the script and naming them very clearly. This is better and cleaner than having the top of the script cluttered up with definitions of PI and MONTHS_IN_A_YEAR.

One thing you can't do is define constants in terms of the initial values of variables, that would be going too far.

Headwords

A quick note on `var` and `def`. Their meaning and use should be fairly obvious, but just to nail down the details:

They are "headwords": that is, once one occurs it has scope until another headword (or the end of the script) without needing any braces or parentheses or indentation to establish this. Other headwords are `cmd` and `import` , both of which will be discussed later. Headwords can be used in any order and as many times as you like, except for `import` which can occur only once, at the top. (So when I refer to "the `def` section" etc in what follows I am not necessarily talking about one continuous piece of the script, but rather about everything in it headed by `def`).

Variables, of course, are defined in the `var` section: in the `def` section we define constants, as you have seen, structs, which we will talk about later, and functions.

Functions

Functions are pure and stateless and so the only thing they can do is specify return values. The basics are demonstrated in `src/functions.ch`. Here's the code:

```
def

twice(x string) : x + x

twice(x int) : 2 * x

twice(b bool) :
     b : "That's as true as things get!"
     else : "That's as false as things get!"

twice(x any) :
     "Can't double " + string (type x) + "."

swap(x, y) : y, x

sign(n) :
```

```
     n > 0 : "positive"
     n == 0 : "zero"
     else : "negative"

(x) squared : x * x

(x) times (y) : x * y

(x) divides (y) :
     y % x == 0

(x) is even:
     2 divides x

say (x) nicely :
     "*~*" + x + "*~*"

gcd (x, y) :
     x < y : gcd(y, x)
     y divides x : y
     else : gcd(y, x % y)
```

And it all does what you'd think it would do:

```
#1 → hub run src/functions.ch as #2
Starting script 'src/functions.ch' as service '#2'.
#2 → twice 2
4
#2 → twice true
That's as true as things get!
#2 → say "hello" nicely
*~*hello*~*
#2 → 5 squared
25
#2 → 5 is even
false
#2 → sign 5
positive
#2 →
```

Note the functions with multiline bodies. As in many other syntactic whitespace languages, the newline is syntactic sugar for a semicolon (or vice versa). And indents and outdents are sugar for left and right parentheses. So

```
sign(n) :
      n > 0 : "positive"
      n == 0 : "zero"
      else : "negative"
```

… is just a nicer way of writing

```
sign(n) : (n > 0 : "positive"; n == 0 : "zero"; else : "negative")
```

We'll look more at conditionals and their syntax in the section after next.

Some things to note. (1) Charm functions have no return statement, since the whole body of the function is just one big (or small) expression saying what to return. (2) Yes, you can have whatever syntax you like, you're welcome (or you're absolutely furious, depending on your tastes). (3) The brackets are obligatory in the declaration but optional when calling (except as needed to indicate precedence). (4) Those are not type hints, they work. (5) Yes, I'm overloading the functions, they have multiple dispatch. (6) You are allowed to write a one-line expression after the `:` if you want to, rather than indenting it and putting it on a separate line.

Note on slightly flakey implementation: at present if you use the same word in too many syntactic roles so that it confuses the parser, the initializer won't throw an error, the parser will simply get confused. I'm aware of this. In future the parser will be less confused and the intializer will be more picky. The problem is already surprisingly small.

OK, there's a lot to unpack here. Welcome to my TED talk. About that flexible function syntax. Yes, anything that gives users any choice about anything can be used to write bad code. My current thinking is that I will ask them nicely not to, or to put it another way, I will write a style guide. I would very much appreciate your input on what it should say.

And also some people will just plain not like it. I saw someone complaining about Python the other day and saying "I don't *want* my code to look like pseudocode." Well, it would be a funny world if we were all the same. I *do* want my code to look like pseudocode.

That said, I probably wouldn't have gone to the trouble with the functions except (1) Charm is a REPL language, and so the nice syntax gives my users a way to put a linguistically friendly front-end on their scripts, they can make a DSL of their choosing. (2) I feel like functional

programming could do with some extra syntactic sugar to help it down. Stuff like this shouldn't be used willy-nilly but it's nice to be able to write `while condition do action to data`.

And the ad hoc polymorphism. I think it suits the use case. If this was a statically-typed language for enterprise-scale software, then it would have generics and interfaces instead. But instead it's meant to be easy to learn, easy to use, for making smaller things fast. And this supplies my users with pretty much all the abstraction they could ever need in one easy-to-grasp language feature.

The functional whitespace. Now, I can see that there are arguments on both sides. On the one hand proponents of braces point out that it's easy to lose your way in whitespaced code and braces avoid this problem. On the other hand proponents of whitespace point out that it's easy to lose your way in code with braces and whitespace avoids this problem. Personally I fall into the second camp, and if you're thinking "oh, that's because you were raised on Python like a baby", I was not so. I was raised on Pascal. Like a baby. (And I don't miss `begin` and `end` at all.)

However … back to the point, and I do have one … Charm is essentially a functional language. And they do tend to have syntactic whitespace, and the reason is that if you can by some perverse feat of ingenuity write a function in a functional PL which is both long enough and deeply-nested enough to get lost in, that would be a code smell. A function in Charm is not supposed to have enough nested structure that the people who like braces would actually need braces to be explicit about it.

For most people the use of commas to separate parameters might need no explanation but some FPLs drop them. I have been guided here as elsewhere by the practice of mathematicians: they could stop using them if they wanted to, their whiteboards wouldn't throw a syntax error. Guided only by the need for clarity of expression, they have kept the commas.

Finally, fans of functional programming might also wonder whether I've considered having lots of different kinds of conditionals. I have, I don't like it. TOOWTDI.

Return types

Function can optionally be given return types. A small example is given in `src/returntypes.ch`:

```
def

add(x, y) -> int : x + y
```

```
swap(x, y) -> string, single : y, x
```

Precedence

A note on precedence. Infix and suffix functions bind rather tightly to their parameters. E.g if you have an infix operator `moo` then `1, 2 moo 3, 4` parses as `1, (2 moo 3), 4`; if you have a suffix operator `foo` then `1, 2 foo` parses as `1, (2 foo)`.

You can give such functions more parameters than two and one respectively, but you would have to use parentheses to indicate this when calling, e.g. if you have a function `(x, y) foo : x + y` then you would call this on values `a, b` by saying `(a, b) foo`. If instead you wrote `a, b foo`, then the only thing this could mean that would be both syntactic and semantic would be "the tuple consisting of whatever `a` is, followed by whatever `foo` applied to `b` is, and if `b` isn't a 2-tuple there's going to be a runtime error."

However, regular functions with their parameters to the right try to slurp up pretty much everything to the right of them: this gives an error because all three numbers are passed as parameters:

```
#2 → gcd 26, 65, 3
error: 'gcd' doesn't have any suitable type signature to interpret
that at line 1 of REPL input
#2 →
```

If what you mean is `gcd(26, 65), 3`, then you can write that, or you can write `(gcd 26, 65), 3`.

I favor a TOOWTDI approach to things, but because of the way Charm works it will always be syntactic to write both those things. Also because of the way Charm works, the second of those things must always mean what you think it means. The only thing I could do to enforce TOOWTDI in this case would be to make it so that the first option *doesn't* mean what you think it means, but means `gcd(26, 65, 3)`. This would make people angry and upset.

The logical operators `or` and `and`, however, have lower precedence than functions/operators, i.e. `foo zort and bar` parses as `(foo zort) and bar`

Because there are ten thousand functions that return a boolean value for every one to which we actually wish to pass a boolean expression as its first parameter. I'm not sure I've written anything of the form `foo (zort and bar)` in my entire life.

A fuller explanation of the system of precedence will be given in an appendix.

Conditionals

The file `src/conditionals.ch` shows some more things you can do with conditionals.

```
def

// Conditionals nest more comprehesibly with whitespace.

classify (i int) :
     i > 0 :
          i < 10 : "small number"
          else : "big number"
     i == 0 : "zero"
     else :
          i > -10 : "small negative number"
          else: "big negative number"

// An incomplete conditional.

incomplete(i) :
     i > 0 :
          i < 10 : "small number"

// And a use for one.

objectToZero(x) :
     x == 0 : "Zeros are bad! Bad!"

(x) divides (y) :
     objectToZero(x)
     else : y % x == 0
```

The `classify` function does what you think it will.

```
#2 → hub run src/conditionals.ch
Starting script 'src/conditionals.ch' as service '#3'.
#3 → classify 100
```

```
big number
#3 → classify -5
small negative number
#3 →
```

The `incomplete` function will throw an error for numbers not captured by its conditions, and so is bad. Normally you should `else` all your conditionals to stop this from happening:

```
#3 → incomplete 5
small number
#3 → incomplete 100

[0] Runtime error: unsatisfied conditional at line 1 of REPL input.

#3 →
```

But note the `objectToZero` function and its use. Charm always needs a conditional to evaluate to something in the end. But this doesn't need this to happen all in one function. The last two functions demonstrate what I mean more quickly than I could explain it. If `objectToZero` finds its conditional unsatisfied, it would return an error to the end user, but it merely reports its unsatisfied state and not an unsatisfied conditional error to `divides`, or generally to any calling function. It's only when such an unsatisfied state works its way up to the REPL or to assignment to a variable that it's treated as an error:

```
#3 → 2 divides 4
true
#3 → 0 divides 4
Zeros are bad! Bad!
#3 → objectToZero 0
Zeros are bad! Bad!
#3 → objectToZero 2

[0] Runtime error: unsatisfied conditional at line 1 of REPL input.

#3 →
```

Structs

These are demonstrated in the file `src/structs.ch`:

`def`

```
person = struct(name string, age int)

cat = struct(name string, nobelPrizes int, pink bool)

catDefaults = nobelPrizes :: 0, pink :: false

var

doug = person "Douglas", 42

joe = person with name :: "Joseph", age :: 22

tom = person with age :: 49, name :: "Thomas"

myCat = cat with name :: "Felix", catDefaults

me = catOwner("Tim", myCat)

myField = name
```

Let's give this a spin in the REPL:

```
#3 → hub run src/struct.ch
Starting script 'src/struct.ch' as service '#4'.
#4 → doug
(name :: Douglas, age :: 42)
#4 → joe[name]
Joseph
#4 → tom[age]
49
#4 → doug == person with name :: "Douglas", age :: 42
true
#4 → myCat
name :: Felix, nobelPrizes :: 0, pink :: false
#4 → myCat = cat with name :: "Tibbles", catDefaults
ok
#4 → myCat
(name :: Tibbles, nobelPrizes :: 0, pink :: false)
#4 → myCat[myField]
Tibbles
#4 → doug[myField]
Douglas
```

`#4 →`

Some things to note:

(1) Structs are indexed like maps, with square brackets, because a struct is very like a little opinionated map.

I really don't know why anyone thought the syntax for indexing stucts should look like methods.

(2) Struct types are declared by `nameOfStruct = struct(<type signature of construtor>)`.

(3) There is also a long-form constructor using the word `with` and a tuple of label-value pairs.

Yes, that's not TOOWTDI. That's literally two ways of doing it. But there is an argument for both.

(4) The labels of the fields are first-class objects, of type `label`.

(5) Label-value pairs are also first-class objects.

See, this is why pairs are a good idea. `myCat = cat with name :: "Tibbles", catDefaults` is a beautifully non-toxic way of doing defaults, way better than overloading the constructor (which is possible, 'cos Charm lets you overload whatever you like, but that would be a bad idiom and this is a good one. Explicit is better than implicit.)

(6) Comparison is as always by value.

(7) For the purposes of passing structs to functions, all structs are considered less specific than the `struct` type and more specific than `nil`, which is a type of struct having no fields and no constructor, instantiated by the constant `NIL`. Bear in mind that `NIL` is the value and `nil` is its type.

Eventually I guess I'll have sum types but until then this allows you to do recursive data types more neatly than otherwise would be the case.

The `with` operator

We've just seen one use of `with`, as a constructor, where it takes a subtype of `struct` as its first argument. The other is as a modifier, returning an altered copy of the structure, or a map, or a list.

For example, if you still have `struct.ch` open, then you can do this:

```
#4 → doug with age :: 43
(name :: Douglas, age :: 43)
#4 →
```

This does not change the value of `doug`.

```
#4 → doug
(name :: Douglas, age :: 42)
#4 →
```

The desired changes can be concatenated with commas:

```
#4 → myCat with name :: "Harold", pink :: true
(name :: Harold, nobelPrizes :: 0, pink :: true)
#4 →
```

However, the same effect can be achieved by using several with operators, e.g. `myCat with name::"Harold" with pink::true`, and sometimes this may be preferable for clarity.

We can change the field of a field, etc:

```
#4 → me
(name :: Tim, pet :: (name :: Tibbles, nobelPrizes :: 0, pink ::
false))
     #4 → me with [pet, nobelPrizes] :: 5

(name :: Tim, pet :: (name :: Tibbles, nobelPrizes :: 5, pink ::
false))
#4 →
```

The same thing works for lists and structs:

```
#4 → ["apple", "banana", "cherry"] with 1 :: "zebra"
[apple, zebra, cherry]
#4 → map(1::"one", 2::"two") with 1 :: "ONE"
map (1 :: ONE, 2 :: two)
#4 → map(1::"one", 2::"two") with 3 :: "three"
```

```
map (1 :: one, 2 :: two, 3 :: three)
#4 → [doug, joe, tom] with [2, name] :: "Zebedee"
[(name :: Douglas, age :: 42), (name :: Joseph, age :: 22), (name ::
Zebedee, age :: 49)]
#4 →
```

I haven't implemented this for other indexable things such as tuples, user-defined indexing, and strings. Not for tuples, because I want people only to use tuples for passing parameters to functions and getting results back; not for user-defined indexing because it's impossible; and not for strings because we will have plenty of string-handling functions and because a string can't contain containers so it doesn't require the same machinery.

Overloading

As we have seen, there is support for overloading functions. This includes built-in functions and operators (except the logical operators and the "protected punctuation" `{ } [ ] ( ) : ; , " ` and `` ` ``).

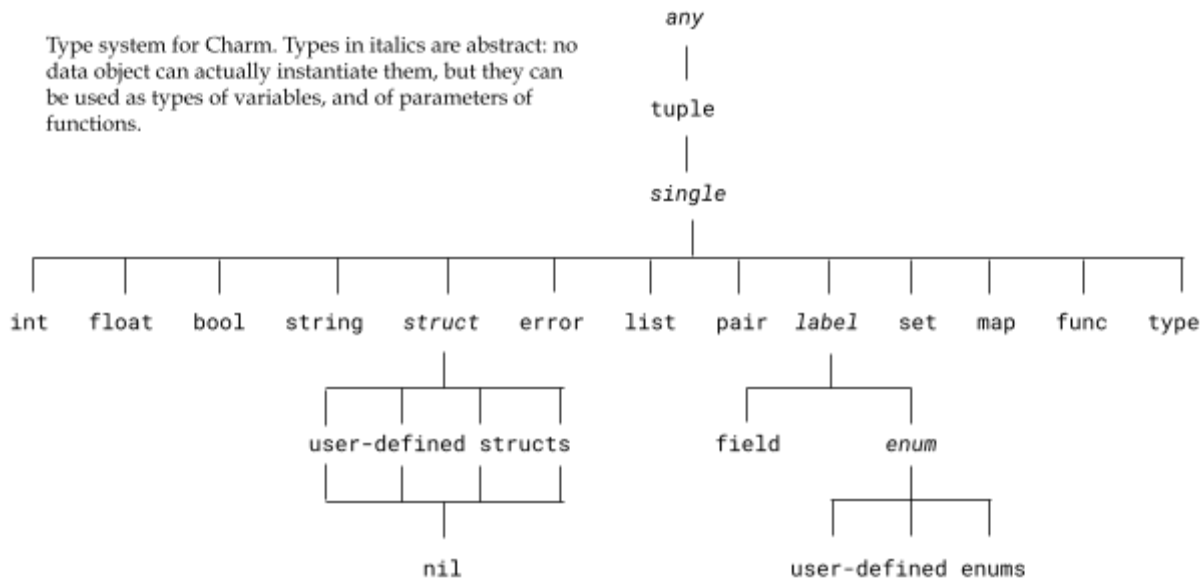
There are probably some things I've allowed you to overload that you really really shouldn't overload. When I find out what they are I will put a stop to this.

Also, by overloading the `bool` function, you can make a type truthy; and by overloading the `index` function, you can make it indexable by whatever type you like (except `tuple`, because of the current restrictions on what you can do with variadics). So if you define `index(i int, t yourType)` then you can index it by integers, if you define `index(p pair, t yourType)`, you can index it by pairs, etc.

Even if you disapprove of overloading you've got to give me some style points for that one.

When you overload a function, the interpreter's choice of which version of the function to use depends of course on the types of the passed parameters. If there are two such choices, for example if you have a function foo defined for `foo(x int)` and `foo(x any)` and you pass it an integer, then it will always use the more specific type signature, in this case `foo(x int)`. A file `src/overloading.ch` has been supplied to demonstrate this behavior.

As a reminder, here is the type system, ordered from most general at the top to most specific at the bottom.



You should not use overloading gratuitously. Cases where you should use it:

(1) In the `cmd` section, which we haven't met yet, you're in effect writing a tiny DSL as a front-end, so you can do what you like to gratify your end-users.
(2) For making a math library where it's idiomatic to reuse operators.
(3) When you genuinely need the sort of abstraction they supply.

You should not, for example, use it to supply you with default settings for functions (which I left out of the language for a reason). Instead, if you have for example a function `foo(x int, y string, b bool)` and you want the defaults for the last two parameters to be `"walrus", true`, then you should define a tuple `fooDefaults = "walrus", true`, and then call `foo(number, fooDefaults)` to get your defaults.

Explicit is better than implicit.

Enums

You can create your own enumerated types. Each such type is a subtype of `enum`, which is a subtype of `label`.

The script `src/enums.ch` supplies an example of usage:

```
def
```

```
Suit = enum CLUBS, HEARTS, SPADES, DIAMONDS

Value = enum ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
          .. EIGHT, NINE, TEN, JACK, QUEEN, KING

Card = struct(value Value, suit Suit)

isBlack(suit Suit) : suit in {CLUBS, SPADES}

isBlack(card Card) : isBlack(card[suit])
```

The len function can be applied to enum types, e.g. `len Suit` is `4`; and similarly enum types can be indexed by integers, e.g. `Suit[2]` is `SPADES`.

Lambdas and closures

We can make anonymous functions using the func keyword:

```
var

add = func(x, y) :
     x + y

g = addN(5)

def

addN (n) :            //  Look, a closure!
     func(x) : x + n

apply (f) to (x tuple) :
     f x

apply (f) twice to (x tuple) :
     f f x

while (p) do (f) to (x tuple) :
     p x : while p do f to f x
```

```
      else : x

condition = func (x, y) : x > 0
action = func (x, y) : x - 1, y * 2
```

Let's give it a spin in the REPL.

```
#4 → hub run src/lambdas.ch
Starting script 'src/lambdas.ch' as service '#5'.
#5 → add 7, 3
10
#5 → g 3
8
#5 → apply g twice to 3
13
#5 → while condition do action to 7, 1
0, 128
#5 →
```

Because of referential transparency I could do a thing that made the closures by actually substituting the values of the variables into a copy of the AST. It would take longer to create the closure but then it would work faster when you applied it, what do you think?

Fans of functional programming idioms will realize that what that last one needs to make it work properly are local constants. Let's have some.

Local constants and inner functions

These are demonstrated in `src/given.ch`:

```
def

g (x) : a * b * x
given :
      a = x + 1
      b = x + 2

// Now let's use that 'while' function properly.

while (p) do (f) to (x tuple) :
```

```
      p x : while p do f to f x
      else : x

power (x) :
      (while condition do action to x, 1)[1]
given :
      condition(x, y) : x > 0
      action(x, y) : x - 1, y * 2

// Inner functions can have their own 'given' blocks,
// as can lambdas.

var

f = func (x) : a * b(x)
given :
      a = x + 1
      b(x) : x + c(x)
      given :
            c = func(x): m * x
            given m = 5
```

Note that an inner function is syntactic sugar for a lambda. So it must have a vanilla `<function name> (<parameters>)` syntax, it cannot be overloaded, and it *can* be passed.

This

Functions can refer to themselves as `this`. This isn't just syntactic sugar: there isn't any other way for a lambda to refer to itself, since they have no names. The file `src/this.ch` has an example.

```
def

max (L list):
      len(L) == 0 : error "taking the max of an empty list"
      else : maxer(1 , L[0])
given :
      maxer = func(i int, m int) :
            i < len(L) : this i + 1, max(m, L[i])
```

```
          else : m

max (x, y int) :
     x > y : x
     else : y
```

Import

The `import` section has to come at the top of the script, if there is an `import` section at all. It consists of filepaths expressed as string constants and separated by newlines:

```
import

"lib/foo.ch"
"lib/bar.ch"
```

An example is given in src/import.ch, which imports lib/complex.ch, which is a tiny library to overload the + and * operators to let you treat pairs as complex numbers. Here's the main script:

```
import

"lib/complex.ch"

def

mandelbrot(c pair) :
     mandeler(0, (0.0 :: 0.0))
given :
     mandeler = func(i int, z pair) :
     i > 50 : true
     z[0] * z[0] + z[1] * z[1] > 4.0 : false
     else : this(i + 1, z squared + c)
```

And the library:

```
def

(c1 pair) + (c2 pair) : c1[0] + c2[0] :: c1[1] + c2[1]
```

```
(c1 pair) * (c2 pair) : c1[0] * c1[0] - c2[0] * c2[0] ..
                .. :: c1[0] * c2[1] + c1[1] + c2[0]

(c pair) squared : c * c
```

Most of the other libraries supplied wrap around Go functions rather than native Charm: such functions will be discussed in a later section.

These libraries are `strings`, `math`, and `fmt`. They contain a subset of the libraries of the same name in Go, having functions with the same names (except in camel case and not Pascal case) and with the same behavior.

Recursion and how to avoid it

Functional programming has a reputation of requiring recursive functions (as in the previous example), which sometimes alarms people. In fact, except in cases where you'd reach for recursion in any other language, you really only need one recursive function, it's very short, and I've already written it for you. As you'll have guessed, it's in `src/recursion.ch`:

```
def

// You only really need this one:

while (p) do (f) to (x tuple) :
     p x : while p do f to f x
     else : x

// But we can make other general-purpose recursive functions ...

for (n) do (f) to (x tuple) :
     (while unfinished do loop to 0, x)[1::count(x) + 1]
given :
     unfinished = func(i int, x tuple) : i < n
     loop = func(i int, x tuple) : i + 1, f x

// And all the nasty recursion disappears ...

power2 (n int) : for n do action to 1
given : action = func(x) : 2 * x
```

```
stars (n int) : for n do action to ""
given : action = func(x) : x + "*"

// Let's make that max function shorter

max(L list) : (while condition do action to 1, L[0])[1]
given :
     condition = func(i, m) : i < len(L)
     action = func(i, m) : i + 1, max(m, L[i])

max (x, y int) :
     x > y : x
     else : y
```

A quick look at the REPL to see this working:

```
#5 → hub run src/recursion.ch
Starting script 'src/recursion.ch' as service '#6'.
#6 → stars 5
*****
#6 → power2 8
256
#6 → max [72, 35, 84, 16]
84
#6 →
```

And so on.

Obviously such standard functions for doing generalized recursion should be contained in a standardlibrary, the *prelude* as it's called in Haskell.

The prelude

To demonstrate the principle, a small prelude has been supplied in `lib/prelude.ch`, including `while`, `for`, and `mergesort`.

More functions like this will be provided when I've decided what they should be, and what they should look like.

Ok. What should they be, and what should they look like? This flexible syntax gives me free rein and writing a standard prelude gives me first dibs on useful constructions. Of course we can make them all so they're tail-call-optimizable.

Type conversion and reflection

Type conversion has been supplied for some types and can be user-implemented for others. Built-in convertors include `string`, which can operate on int and float types, `float` and `int`, which will convert strings or one another to the respective type, and `bool`, which is defined for integers (false if `0`), and strings, maps, lists, and sets (false if length is `0`).

`tuple` will have no effect on a tuple, and will convert anything else *to a tuple of arity 1*. If instead you wish to spread the elements of a set or list into a tuple, use the `spread` operator.

```
#6 → "2" + string 2
22
#6 → 2.0 * float 2
4.000000
#6 → 2 * int 2.0
4
#6 → 2 * int "2"
4
#6 → spread [1, 2, 3]
1, 2, 3
#6 →
```

We can do reflection by using `type` to cast a thing to its type:

```
#6 → type "hello"
string
#6 → type 1
int
#6 → type true
bool
#6 → type bool
type
#6 → type type
type
#6 →
```

Eval and serialization

`eval` will evaluate a string as a Charm expression.

```
#6 → eval "2 + 2"
4
#6 →
```

Conversely, the `charm` function applied to a value will return a string which represents that value as a Charm literal:

```
#6 → [true, 3, "true", "3"]
[true, 3, true, 3]
#6 → charm [true, 3, "true", "3"]
[true, 3, "true", "3"]
#6 →
```

Error handling

A runtime error is a first-class object. When a function/operation returns an error, it literally returns an error:

```
#6 → 1 / 0

[0] Error: division by zero at line 1 of REPL input.

#6 →
```

With a few exceptions (pun unavoidable) when a function/operator is passed an error, it automatically returns that error:

```
#6 → 1 / 0

[0] Error: division by zero at line 1 of REPL input.

#6 → 1 + 1 / 0

[0] Error: division by zero at line 1 of REPL input.

#6 → string 1 + 1 / 0

[0] Error: division by zero at line 1 of REPL input.

#6 →
```

An uncaught error will therefore propagate upwards through the calling functions until either it is returned to the REPL, or an attempt is made to assign it to a variable, in which case the assignment will return the error instead just like any other operator.

(In Charm, everything is an expression, and everything returns a value, and assignment is therefore an operator. What assignment to a variable usually returns is a Successful Object, which is that green `ok` you keep seeing in the REPL. Unlike most things in Charm, Successful Objects are not meant to be first-class, because that would be terribly confusing. If you find a way to assign one to something, let me know and I'll put a stop to it.)

You can create your own errors by type conversion from string to error, and they work the same way:

```
#6 → error "this is my error"

[0] Error: this is my error at line 1 of REPL input.

#6 → 1 + error "this is my error"

[0] Error: this is my error at line 1 of REPL input.

#6 →
```

But there are things you can do to stop errors from propagating.

(1) You can use the `type` keyword to find its type. Instead of returning an error, this returns `error`.
(2) You can assign it to a local constant. (Of course this means you'd be assigning the valid data to the same constant if there was any, so you still have to do something with it.)
(3) (Not implemented yet.) You can index it by its fields, e.g. message and line number.

Let's have a look at `src/error.ch`. This shows four possible implementations of a `mod` function to supplant that nasty `%` operator we inherited from C.

```
def

// We could propagate the error thrown by %.

(x int) modA (y int) :
      remainder >= 0 : remainder
      else : remainder + y
```

```
given :
     remainder = x % y

// We could anticipate the error and throw our own.

(x int) modB (y int) :
     y == 0 : error "taking the modulus of a number by zero"
     remainder >= 0 : remainder
     else : remainder + y
given :
     remainder = x % y

// We could catch the error and throw our own.

(x int) modC (y int) :
     type remainder == error : error "taking the modulus of a number
by zero"
     remainder >= 0 : remainder
     else : remainder + y
given :
     remainder = x % y

// We could catch the error and return something other than an error.

(x int) modD (y int) :
     type remainder == error : "this isn't an error, just a friendly
warning"
     remainder >= 0 : remainder
     else : remainder + y
given :
     remainder = x % y

// ... etc, etc.
```

The way errors work has one consequence you might not guess, which I should point out as a potential hazard for the unwary. The rule that a function or operation applied to an error yields that same error applies also to the `,` operator, the comma.

This means that even in a function which normally returns multiple return values, you can't return an error and another piece of information. (You could pass information *inside* the error and I will provide facilities to do so later. It's on my list.)

It also means that you can't have a function that *accepts* an error and another piece of information. You might think that the following function would be a useful idiom, and so it would be if it worked, but it can't:

```
def

default (x, y) :
      type x == error : y
      else : x
```

If you try to do multiple assignment to local constants, and you assign an error to them, this is legal, and the same error will be assigned to all of them `x, y = error "just the one"` is legal and meaningful in the `given` block of a function.

I guess people are going to complain about the error-handling because: (a) There's no way of handling errors that makes everyone happy. (b) It does have limitations, even one small-calibre footgun, as pointed out above. I'm not quite happy with it myself for that reason: for once I found myself in a lesser-of-two-evils situation rather than just doing what the heck I want (but in this case what *do* I want? how else could it possibly work?) , and this is distinctly the lesser, but still slightly evil. If anyone has any thoughts … ? But as it is it works almost exactly like procedural throw and catch, the difference being that Charm can't distinguish between throwing an error and passing one as a value, and procedural languages can. One could maybe have a wrappedError type one is allowed to cast it to for the purposes of passing them? It's a bit kludgy but it would work.

Embedded Go

A function may be given a body in Go by following the `:` introducing the function body by the keyword `golang`, and then enclosing the body of the function in braces.

```
multiply(a, b int) : golang {
    return a * b
}
```

It is not necessary to give return types. Multiple return values are allowed.

While strictly speaking it isn't necessary to give types to the input of the function either, if you don't then they will be of type `interface{}` from the perspective of the Go code, and so in order to do anything interesting with them they would need to be downcast. This, for example, will fail during intitialization, since Go cannot multiply two things of type `interface{}`.

```
multiply(a, b) : golang {
    return a * b
}
```

The Charm interpreter automatically does type conversion between Charm types and Go types. If you wish the function to be passed a Charm object, use the suffix `raw`, e.g:

```
nthFromLast(L list raw, n int) : golang {
     return L.Elements[len(L.Elements) - 1 - n]
}
```

In order to use this properly it is necessary to learn the public methods of Charm's Object interface and the classes that implement it.

When the initialization of Go code fails, a file called `golang<n>.go` will be left in Charm's root file: it consists of Go code generated from your code. The wrapping around the code in your function(s) is simple and the resulting file no harder to debug than ordinary Go.

You can import from Go by prefacing the name of the thing to import with `golang`, e.g:

```
import

golang "strings"
```

The `strings`, `math`, and `fmt` libraries are implemented by wrapping Charm functions around the Go standard libraries.

The cmd section

So, we have functions with and without typing, with variadics if you please, prefix, infix, suffix, or whatever, named or anonymous, with or without given clauses, with single or multiple return values, wrapped around Go … but they are all remorselessly pure. So far, a Charm script can't change the state of anything. In fact, the only way the variables can vary at all so far is if the end-user tells them to at the REPL! What are we going to do about this?

Nothing. This is an extremely desirable state of affairs. The end-user's data shouldn't change without the end-user saying so. If I wanted something that changed state spontaneously, I'd buy a banana and leave it to rot.

No, the only remaining problem is not that the script can't change the data but that the way the end-user does it is presently unsafe, unfriendly, and inconvenient. We will remedy this with the `cmd` section and the `private` access modifier.

The other day I saw Simon Peyton Jones on YouTube talking about how regrettable it is that Haskell is "useless" (his word, I know he's exaggerating) because of its difficulties of doing stuff to state; whereas the "useful" languages are unlike Haskell. Because they can. He concedes that there are some languages that hit what he considers to be the sweet spot (like SQL) but then, he says, they're all embedded in something.

This suggests a general way to make functional languages more useful. You could take your FPL and embed it in an all-purpose thing for embedding your FPL in, but which can also do stuff. Putting mutability of state on the outside of the service, always closer to the end-user than the business logic is. Again, like SQL.

"But wait", you ask, "wouldn't any such "all-purpose thing" for embedding your FPL in and affecting state have to be another programming language itself, in order to be all-purpose?"

No. It has to be embedded in a language, but not in a *programming language*. The FPL is already Turing-complete, so it has no need to be embedded in a language as complex as itself. We can embed it in something which is not Turing-complete, which has no possibility of recursion and no flow of control much more sophisticated than "do the next thing on the list". But which has the same basic syntax and type system as the FPL. So that from the perspective of a programmer who hasn't read the last few paragraphs, Charm is *one* language, but a very opinionated one.

The inner, pure, functional, language defined in the def section is easy to understand because it is pure and referentially transparent. The outer, imperative, state-affecting language defined in the cmd section is easy to understand because it's as dumb as a brick. Divide and conquer! As a side-effect (or perhaps I should say: "as an absence of side-effects") the smart bit that does the thinking can do so concurrently with itself.

The cmd section is demonstrated in `src/cmd.ch`:

```
var

x = 42

cmd
```

```
times (n) :
     x = n * x

add (n) :
     x = x + n
     return x

step2A :
     x = x + 1
     return x
     x = x + 1
     return x

step2B :
     add (n)
     add (n)
given n = 1
```

In the REPL :

```
#6 → hub run src/cmd.ch
Starting script 'src/cmd.ch' as service '#7'.
#7 → x
42
#7 → times 5
ok
#7 → x
210
#7 → add 6
216
#7 → step2A
217, 218
#7 → step2B
219, 220
#7 →
```

Things to note: commands don't need to take parameters. Nor do functions in fact, but they'd be useless if they didn't — commands have access to the data, whereas functions have to be passed the variables as parameters in order to see them.

Commands don't have to have a `return` statement. If they do, it doesn't halt execution of the command: rather, all the return values accumulate to be returned in one tuple.

Commands can have `given` blocks, and for purposes of modularity and code reuse they can call other commands, though the initializer will object to any circular dependencies.

Note: when I say the intitalizer "will" do this, I mean I haven't written that bit yet.

Conditionals currently don't work in the `cmd` section: when they do they will be frowned on except for simple gatekeeping purposes.

Transactions

Every command is a transaction: if it returns a runtime error, no matter at what point in its execution, none of the variables will be changed.

In fact, any single line you type into the REPL is treated as a transaction, so if you had `src/variables.ch` running and you entered `h = "bar"; x = 1 / 0`, then you would find that the division by zero error prevents `h` from updating as well as `x`.

Encapsulation

Now that the `cmd` section is supplying us with getters and setters for our data, it remains only to encapsulate it. Things following any headword are public by default. The `private` modifier makes everything after it private until the next headword or the end of the script. There is no corresponding `public` modifier: things are public first by default, and private after you say so. Using `private` twice without a change of headwords is a syntax error.

(Because it's an indication that you don't know what you're doing, destroys readability, etc. And there's no public modifier because people would expect things to be public first by default, private later by declaration: so if you allowed a public modifier it would be usually used about as much as the 9 on a microwave and would only surprise people if it ever was used. So.)

Here is a tiny demonstration in `src/private.ch`:

```
cmd

get :
      return x
```

```
set (y) :
     x = y

var

private

x = "heffalump"
```

In the REPL:

```
#7 → hub run src/private.ch
Starting script 'src/private.ch' as service '#8'.
#8 → x = "piglet"
[0] Error: reference to private or non-existent variable 'x' in REPL.
input
#8 → x
[0] Error: reference to private or non-existent variable 'x' in REPL.
input
#8 → get
heffalump
#8 → set "piglet"
ok
#8 → get
piglet
#8 →
```

Commands and functions can also be made private in the same way, by use of the `private` modifier. `private` does not apply to user-defined types (enums and structs) because I don't know what it should mean for such things to be private; and not to constants because I've only just noticed I didn't do that and have decided to leave it 'til the next sprint.

That's pretty much everything you need to know about imperative programming in Charm.

Communication between services: `exec`

The `exec` keyword allows one service to use the public functions and commands of another named service. The calling service is the inner scope. For example, if we have the script src/foo.ch:

… and the script src/bar.ch:

… then we can do this:

```
→ hub run src/foo.ch as FOO
Starting script 'src/foo.ch' as service 'FOO'.
FOO → a times b
6
FOO → hub run src/bar.ch as BAR
Starting script 'src/bar.ch' as service 'BAR'.
BAR → multiply a by b
35
BAR →
```

File access

You can issue the instructions `save as <filename>`, `save`, and `open <filename>` to the service, which will then save all its data to / load all its data from the given file.

The language itself has been given a little file-handling capacity with the `file` type. `file <filename>` creates a `file` object, which has a field `contents` which contains the file as a list of strings.

This is pretty much a stopgap to let me write test programs, I haven't put much thought into it.

Service variables

As the name suggests, these are settable variables that change the state of the service. For example `$view`, which can be set to `"plain"` or `"charm"`, (and which is the only one I've actually implemented) determines how things are output: if `"charm"` is selected output will be in the form of Charm literals.

They are loaded and saved with the rest of the data of a service.

They have default values, but if the script of the service initializes them in the `var` section, they will be initialized like that, and if they are initialized as `private` then the end-users won't be able to change them.

The hub

The hub has a number of present and intended purposes.

Housekeeping

As you've seen, the hub can start services for you with `hub run <script name>`. It lets you talk to any of the services, and to switch which one is the default to be talked to. `hub run` without any parameters will give you an "empty service", a mere REPL.

So far in this manual/demonstration we haven't given our services names and so the hub has just assigned them serial numbers `#0`, `#1`, etc. You can start a named service using `hub run <script name> as <service name>`.

When you close down Charm, it will remember the named services you had running and which of those, if any, was your current service (the one named in the prompt, the one you're talking to by default).

When you start up Charm again with `./charm`, it will start up all the named services, and if the current service was a named service when you shut it down, then Charm will make it the current service when you start it up again.

If you start Charm up with `./charm <parameters>` then Charm will behave as described in the last paragraph and then behave as though you had typed `hub <parameters>` into the prompt.

`hub services` will list all the services the hub is running, whether named or numbered.

`hub halt <service name>` will halt the named service.

`hub quit` closes all the services and Charm itself.

`hub reset` will start the service up again with its variables re-initialized. `hub rerun` will re-execute the last `hub run` command, and so unlike hub reset it doesn't need the last `hub run` command to have been successful: if it failed to start up a service because of a syntax error, then `hub reset` has no service to reset, but `hub rerun` has a script to rerun.

Errors: 'why', 'where', and 'trace'

When Charm produces syntax or runtime errors, you can ask it for an explanation. A demonstration is given in `src/errordemo.ch`. If you try to start this up, you will find that it's riddled with syntax errors, to each of which Charm gives a number in square brackets when it reports them. You can get further information on these errors with `hub why <error number>`. E.g:

```
→ hub run src/errordemo.ch

Starting script 'src/errordemo.ch' as service '#0'.

[0] Error: attempted assignment in 'import' section at line 3:2 of
'src/errordemo.ch'.
[1] Error: declaration of function in 'var' section at line 7:2 of
'src/errordemo.ch'.
[2] Error: redeclaration of 'private' at line 15:0-7 of
'src/errordemo.ch'.
[3] Error: '(' unclosed by outdent at line 24:0 of
'src/errordemo.ch'.
[4] Error: inexplicable occurrence of ':' at line 21:8 of
'src/errordemo.ch'.
[5] Error: a line ending in ',' must be followed by a line beginning
with '..' at line 26:0-3 of 'src/errordemo.ch'.
[6] Error: inexplicable occurrence of 'else' at line 26:4-8 of
'src/errordemo.ch'.
[7] Error: if it occurs, 'import' must be the first headword at line
28:0-6 of 'src/errordemo.ch'.

→  hub why 2

Error: redeclaration of 'private'.

In blocks of the script where things can be declared private (at
present only 'var' blocks), the 'private' modifier can only be used
once after each headword: things before the 'private' modifier are
private, things after it are public.

You're seeing this error because you used the 'private' modifier
twice after the same headword.

                                    Error has reference 'init/private'.
```

If Charm produces a runtime error, then you can see the trace of the error with `hub trace`. There is no need to give the number of the error, since you will only ever get one runtime error, which will therefore be number 0. You can demonstrate the trace feature right now just by putting `1 / 0` into the REPL, but as that doesn't produce a very interesting trace we have supplied a file `src/rtedemo.ch`.

The `hub where <error number>` command has the same syntax as `hub why` but will show you the line with the relevant token in red and underlined, for when it's hard to find it in a long and complicated line.

```
→ 1 + int("1") + len("42") + int("42") + len("1.0") + 1 + 1.0 +
int(4.0)


[0] Runtime error: can't apply '+' in context <int> '+' <float> at
line 1:54-55 of REPL input.

→ hub where 0

Found at line 1:54-55 of REPL input:

1 + int("1") + len("42") + int("42") + len("1.0") + 1 + 1.0 +
int(4.0)
```

As a server

If you tell the hub `hub listen <path> <port>` then it will become a server: e.g. if you do `hub listen /foo 3333` and then open up a new terminal and do `curl -X POST -d '2 + 2' 'http://localhost:3333/foo'`, you should get the answer `4`.

The hub also has capabilities for registering users and allowing admins to do role-based access management. This is not documented here: there are some rough edges I want to trim.

Talking to other programs

`os <parameters>` will behave as though you had typed `<parameters>` into the command line, allowing you to create and delete files, change directory, etc.

`hub edit <filename>` will open the file in vim.

Testing

In a REPL language it's almost inevitable users will test their scripts via the REPL. The `hub snap` command allows you to do that better. The syntax is either `hub snap <filename>` or `hub snap <filename> with <data filename>` or `hub snap <filename> with <data filename> as <test filename>`. If no test file name is given Charm will supply a suitable one, and if no data file is supplied Charm will just initialize the service as usual.

Charm will then turn serialization on, so that you can tell the difference between `"true"` and `true`; and between four spaces and a tab, etc.

And then what you type and the service's responses will be recorded. (There will be a `#snap →` prompt to remind you that this is what you're doing.) To finish recording, you tell it what to do with the snap:

- `hub snap good`: this is the desired behavior and I want to make it into a test that will ensure it keeps happening
- `hub snap bad` : this is undesirable behavior and I want to make it into a test that checks that it doesn't happen
- `hub snap record` : I want to be able to replay this and see how the output changes as I change the script and/or data
- `hub snap discard` : I don't need this

All the tests classed `good` or `bad` associated with a script will be run by `hub test <script filename>`. As an example, try running `hub test src/testall.ch`: Charm will run the associated test in the `tst` folder and hopefully confirm that the interpreter hasn't regressed since I wrote this bit of the manual.

To run a test classed as `record`, use `hub replay <test filename>`.

The testing system could be improved: I basically wrote as much of it as I needed to use it as a regression test on Charm itself.

The duck

All programming languages are represented by animal mascots: it is so written. Charm will be represented by an adorable Aylesbury duckling named Daisy. Daisy the Data Duck.

My decision on the duck is final.

Appendix A: precedence

Technically, Charm has a horrifying number of levels of precedence, but this is because Everything Is An Expression, and so the parser needs to know things like the precedence of `given` and `func` and `struct` and newline, all of which are operators. This is not something you need to think about.

(Yes, newline is an operator, under the hood Charm has *semantic whitespace*, deal with it.)

If we leave that aside and just talk about the stuff you need to think about when you think about precedence, we can use the following simplified scheme.

This is subject to change. In particular, the precedence of `not` is clearly too high.

`or`
`and`
`== !=`
`> < <= >=`
functions, `with`
`,`
`::` and user- defined infixes
`+ -`
`* / %`
suffixes
`not` and `-` as a prefix
`[` used as an indexing operator

Appendix B: example code

These are not particularly good examples because they were written while some language features such as inner functions and enums hadn't been written yet – and indeed while I was still learning the idioms of my own language. Still, they show that one can get stuff done in Charm

A CRUD app

Let's write a tiny CRUD app! This can be found in `src/crud.ch`.

For brevity I will omit the listing here, and just give the API. You can add a student and their score with `add <"student name">, <score>`; you can show a table of the entries with show; you can sort the students in decreasing order of scores with sort; and use <function> will

give you an "adjusted" score, e.g. `use func(x) : 3 * x;` or `use func(x) : string(x) + "/20"` or `use func(x) : string(int(100.0 * float(x) / 20.0)) + "%"`.

A Forth

As a more substantial exhibition of Charm's capacities as an FPL, the file src/forch.ch implements a Forth in 300 lines of code.

I've followed the spec for Forth I found on this webpage, except for keyboard input.

`ex` `` `<forth code>` `` will execute the given code.

You should use backtick quotes as shown so you don't have to escape " for Forth.

`show` followed by either `code`, `stack`, `mem`, `vars`, `consts`, `defs`, `output`, `error`, or `loopVar` will show the relevant aspect of the state of the Forth machine: `showall` will show all the fields.

Similarly `clear` followed by any of those things will set it to its original state, and `clearall` will reset the Forth machine entirely.

Note 1 : `show code` will only return a non-empty list if execution has failed with an error, in which case it will contain the unexecuted code.

Note 3 : similarly `show loopVar` will always return `NIL`: it has an integer value only during the execution of a loop.

Note 3 : `clear defs` will still leave the built in definitions of `?` and `+!`

A toy Z80 interpreter

"Toy" because it only supports the most popular subset of the language, and "interpreter" because the assembly code isn't converted to bytecode and put in the "memory" of the emoulated machine, but just executed against the machine state.

See the file `src/z80.ch` for the code.

Specification of Z80:

Operations : `ld`, `push`, `pop`, `add`, `adc`, `sub`, `sbc`, `cp`, `neg`, `nop`, `inc`, `dec`, `jp`.

Registers : `af`, `bc`, `de`, `hl`

Flags : zero, negative, carry

Numbers are given as two- or four-digit lower-case hexadecimal preceeded by a `#`, e.g. `#e6`, `#50f4`.

As usual in Z80 assembler, parentheses around a number indicate that it is an address.

Labels are any string preceded by `@`.

Example code to compute as much of the Fibonacci sequence as will fit in eight bits:

```
ld a, #01
ld (#0001), a
ld (#0002), a
ld c, a
ld e, a
ld l, #02
@loop
ld a, c
add a, e
ld c, e
ld e, a
inc hl
ld (hl), a
ld a, l
cp #0d
jp nz, @loop
```

Specification of Charm frontend:

`ex <string>` : applies the given line to the machine state, eg `ex "ld a, #05"`

`load <filename>` : loads a z80 file

`reset` : resets the memory, stack, registers, etc

`run` : runs the loaded file

`step` : takes one step through the instructions in the loaded file

`show` : shows the machine state

A Lisp

To be found in `src/lisp.ch`. Basically a Scheme. I have no time to document it at present. This, however, was written after I introduced enums.

Dimensioned types

The file `src/dim.ch` shows how dimensioned types can (in effect) be implemented in Charm.