Yazoo Help File

Brian Ross

November 6, 2012

Contents

1 Introduction

This document describes "Yazoo": a simple scripting language for controlling C or C++ routines or for embedding into a larger application. Our exposition can be divided into three parts. In the first part, consisting of Chapters 2 through the beginning of 3, we introduce the use of Yazoo by way of example. The bulk of Chapters 3 and 4 constitutes a more topical discussion of the scripting component, which aims more at being pedagogical than systematic. A reference section comes at the end. The goal is to get the user up to speed with as little reading as possible.

Choosing an appropriate programming language for a given task involves a number of tradeoffs. High-level languages tend to be intuitive and powerful; low-level languages are efficient and can control basic machine functions. Interpreted languages can be made interactive and machine-independent, whereas a compiled language protects source code and executes much more quickly.

Yazoo tries to borrow the best from both worlds. It is a basic high-level, interpreted scripting language that can be fleshed out with the user's own C or C++ routines, which can then be run from script files or through an interactive command prompt. Since the external C code must be incorporated into Yazoo at compile-time, Yazoo is provided as a set of source files rather than an executable. To embed our own routines we simply reference them in one of Yazoo's own source files, re-compile Yazoo – and voilà: we have an environment for executing these routines and managing their data.

For example, suppose we wish to write a program to alphabetize a list of names. It's a large list and will take quite a while to process, so we opt to write the sorting routine in C.

```
int SortPhoneBook(int argc, char **argv)     // format explained later
{
    ...
}
```

Speed is not critical for the rest of the job, but the flexibility to call our routine from the command line in a manner of our choosing may be desirable. So we paste the above routine into Yazoo, specifically in the source file userfn.c. We then give our sorting routine a name within Yazoo, by modifying one of the pre-existing lines in userfn.c.

```
UserFunction UserFunctionSet[] = { { "DoSort", &SortPhoneBook } };
```

Finally, we re-compile Yazoo.

```
user-prompt% make yazoo
```

At this point we can make our list and sort it from Yazoo's own command prompt.

```
> ListUSA := load("AllUSA.txt")
> call("DoSort", ListUSA)
> save("/Documents/SortedList.txt", ListUSA)
```

And we are done. Of course this example begs some elaboration, which hopefully the following sections will adequately provide.

2 Embedding C/C++ into Yazoo: A Neural Network Tutorial

Yazoo was originally intended to be a neural network scripting language, even though it admittedly ended up rather wide of that mark. So it seems fitting to demonstrate Yazoo scripting with a neural network example. This section walks the reader through the process of creating and training a neural network, a sort of crude electronic brain that can nonetheless be taught to perform some impressive tasks. The time-consuming 'think' and 'learn' routines of our brain will be written in C, and embedded as commands within Yazoo. We will then go through the process of writing and debugging a script to use our network for a simple pattern-matching task, which is what neural networks are good at. Although this tutorial uses bread-and-butter C, we could just as easily have used C++ instead.

2.1 Background: neural networks

Biological neural networks are the webs of neurons that wire the bodies of animals and form their brains. Each neuron is a sophisticated device that acts as a cable, a switch and a memory cell. It receives electrical signals from built-in sensors or from other neurons, processes this input and responds with a signal of its own. The output signal travels down the neuron's long axon, which branches and makes contacts called synapses with other neurons, stimulating (or inhibiting) them in turn. The type of processing a neuron does to its input can generally be adjusted to help the network learn a task or encode a memory. Biological neural networks can contain billions of neurons, trillions of synapses and their computations can be unfathomably complex.

Artificial neural networks are algorithms based on physical models of real neurons and their networks, that hopefully capture some of their computational properties. In the forthcoming example we'll use a crude (but popular) model in which the activity of a neuron is determined by a weighted sum of the activities of the upstream neurons. The weights form the memory; the network 'learns' by adjusting these weights (synapses) so to encode a pattern or improve its performance at some task. Our weight-adjusting algorithm will be a variant on the famous Hebbian learning rule, in which activating synapses between two neurons are strengthened if the neurons' activities are correlated, and weakened otherwise. This learning rule, inspired by observations of real neurons, tends to drive the network towards an ordered state, and is thus useful for memorizing patterns.

To summarize, our network will operate in two modes: computation, and learning. In the computing mode, certain input neurons' activities are fixed by the user, while the activities of the other neurons x_i evolve by the following formula:

$$x_i = f\left(\sum_j W_{ij}x_j + b_i\right)$$

Here x_i is, say, the mean frequency of signals of neuron i, W_{ij} is the strength of the synapse from neuron j to neuron i, and b_i is called a bias, which is a free parameter can also be trained. The sum runs over all neurons j feeding into neuron i, including the input neurons. The function f is called an 'activation function', which we will take to be the same for all neurons. Specifically, we'll use the common activation function f shown in Figure (1). Note the nonlinearity of f: that is what gives the neural network its power.

In the training (learning) mode, the neurons' activities stay frozen in time while their weights and biases are updated:

$$W_{ij} = W_{ij} + \eta x_i x_j$$
$$b_i = b_i + \eta x_i$$

The parameter η determines the learning rate, which must be large enough to train quickly, but not so big that the algorithm overshoots and destabilizes the network.

We will actually use a variant[1] of the basic Hebbian rule which increments the weights by both positive and negative Hebbian steps. The positive step is taken as the network is being 'taught'; i.e. the output

[1] J. R. Movellan, Contrastive Hebbian learning in interactive networks, 1990

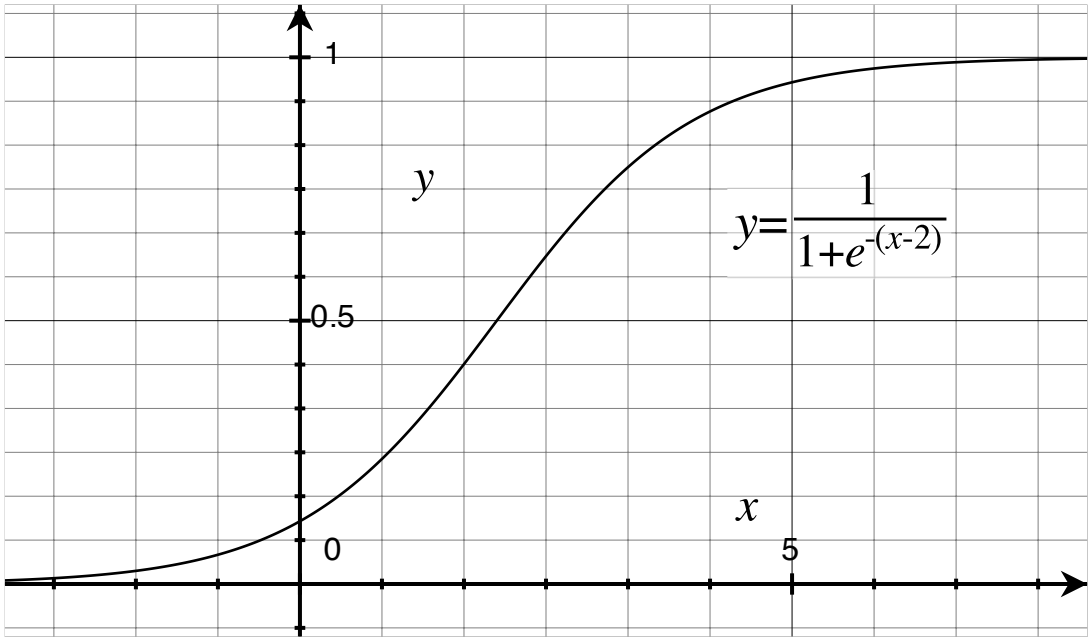

Figure 1: A 'squashed sigmoidal' neuron activation function

neurons are held at the values that we would like them to take. The anti-Hebbian update is made after the network does what it wants without the clamping. Heuristically, the network learns what we want it to do and unlearns its old bad habits in separate steps. It can be shown that for the symmetric networks ($w_{ij} = w_{ji}$) that we will exclusively work with, this combination of positive and negative reinforcement encodes memories more directly than the basic Hebbian rule would.

The two basic operations of a neural network—propagation of signals to perform computation, and adjustment of the weights—are pretty simple routines, but since they will be repeated many times they can be time-consuming. We should write that part of our program in low-level C code.

2.2 C implementation

NN.c

```
#include "NN.h"
#include <math.h>


int main(int argc, char **argv)
{
    neural_network myNN;
    double *inputs, step_size, *target_outputs, learning_rate;
    long i, inputs_num, outputs_num;


    /* ----- set up data types, etc. ----- */


    for (i = 0; i < inputs_num; i++)
        *(myNN.activity + i) = *(inputs + i);
    for (i = inputs_num; i < myNN.neurons_num; i++)
        *(myNN.activity + i) = 0;
```

```
    if ( argc == 6 )    {      // i.e. if we're in training mode then

        if (GetSteadyState(myNN, inputs_num, step_size) != 0)  return 1;
        Train(myNN, -learning_rate);

        for (i = 0; i < outputs_num; i++)
            *(myNN.activity + inputs_num + i) = *(target_outputs + i);

        if (GetSteadyState(myNN, inputs_num+outputs_num, step_size) != 0)
            return 1;
        Train(myNN, learning_rate);         }

    else if (GetSteadyState(myNN, inputs_num, step_size) != 0)  return 1;


    /* ----- save results ----- */


    return 0;
}


// Evolves a network to the self-consistent state x_i = f( W_ij x_j ).

int GetSteadyState(neural_network NN, long num_clamped, double StepSize)
{
    const double max_mean_sq_diff = 0.001;
    const long MaxIterations = 1000;

    double diff, sq_diff, input, new_output;
    unsigned long Iteration, i, j;

    if (num_clamped == NN.neurons_num)  return 0;

        // iterate until the network is at a steady state
    for (Iteration = 1; Iteration <= MaxIterations; Iteration++)    {
        sq_diff = 0;

        for (i = num_clamped; i < NN.neurons_num; i++) {
            input = 0;
            for (j = 0; j < NN.neurons_num; j++)   {
            if (i != j)   {
                input += (*(NN.activity + j)) *
                                (*(NN.weights + i*NN.neurons_num + j));
            }}
            new_output = 1./(1 + exp(-input));

            diff = (new_output - *(NN.activity + i));
            sq_diff += diff*diff;
            *(NN.activity + i) *= 1-StepSize;
            *(NN.activity + i) += StepSize * new_output;
        }

        if (sq_diff < max_mean_sq_diff * (NN.neurons_num - num_clamped))
            return 0;
    }

    return 1;
}
```

```
// Updates the weights and biases with the Hebbian rule.

void Train(neural_network NN, double LearningRate)
{
    int i, j;

    for (i = 0; i < NN.neurons_num; i++)    {
    for (j = 0; j < NN.neurons_num; j++)    {
    if (i != j)    {
        *(NN.weights + i*NN.neurons_num + j) += LearningRate *
                (*(NN.activity + i)) * (*(NN.activity + j));
    }}}
}
```

NN.h

```
typedef struct {
    unsigned long neurons_num;     // 'N'

    double *weights;               // N x N array of incoming synapses
    double *activity;              // length-N vector
} neural_network;

extern int GetSteadyState(neural_network, long, double);
extern void Train(neural_network, double);
```

2.3 Wiring the C into Yazoo

You might notice that our C program isn't yet complete – its `main()` function has a number of gaps, filled by comments indicating that we still have to add routines for allocating/deallocating memory, loading and preparing the data sets, and saving the results. Furthermore, it only performs a single calculation with the network, with an optional learning step. If we want to do something useful with our program, we might consider adding some interactive component like a command prompt – which will require significantly more overhead.

The sales pitch here is that Yazoo can perform all of these housekeeping functions with fairly minimal scripting overhead. So instead of chugging through the rest of the program in C, we'll stop and plug what we have so far into Yazoo's scripting language.

The first order of business is to obtain the C version of Yazoo. (The main difference between the C and C++ versions is just in the extensions of the source files.) None of the source files need concern us now except userfn.c. In the body of userfn.c, we include our new NN.c source file (given in the last section). This is probably poor programming style but it works. We can leave the `Debug()` function in place if we want, but we won't be using it.

```
...
// #include any user-defined header files here

#include "NN.c"
...
```

The second step is to adapt our `main()` function in NN.c to let it communicate with Yazoo. The declaration of a Yazoo-embedded function is the same as that of the `main()` function, so for the most part

we can keep what we wrote without any changes. We do, however, have to give our `main()` a new name, so as not to conflict with Yazoo's own `main()`. Let's call it `run_network()` instead.

```
int run_network(int argc, char **argv)
{  ...
```

We should now also add

```
extern int run_network(int, char **);
```

to NN.h.

The main channel through which Yazoo communicates with the C routine is the `argv` variable. `argv` points to a list of pointers to variables and arrays that are shared between the Yazoo script and the C code. (Structure-variables such as `myNN` are passed as several parameters, one for each element of the structure.) Importantly, since `argv` points to pointers, our C program can use it to both read data from, and send data back to, a Yazoo script.

To synchronize our C variables with our Yazoo variables, let's replace the "set up data types" comment line in NN.c with the following code. Note that `ArgInfo` is an extra parameter that Yazoo passes, which gives information about the other arguments.

```
arg_info *ArgInfo;

ArgInfo = (arg_info *) *(argv+argc);

myNN.neurons_num = (ArgInfo+1)->arg_indices;
inputs_num = (ArgInfo+2)->arg_indices;

myNN.weights = (double *) *argv;
myNN.activity = (double *) *(argv+1);
inputs = (double *) *(argv+2);
step_size = *(double *) *(argv+3);

if (argc == 6)  {
    outputs_num = (ArgInfo+4)->arg_indices;

    target_outputs = (double *) *(argv+4);
    learning_rate = *(double *) *(argv+5);     }
```

Notice that we assigned some of our variables, namely the `weights` and `activity` fields of `myNN`, by reference rather than by value. One reason for this is that they are arrays and copying them would be time-consuming. The other reason is that the job of our routine is to modify `activity` and, if we are training our network, the `weights` array as well, so we need to write into some space that is shared with the script. Note that if we accidentally write past this shared space, we in all likelihood blow a few of Yazoo's fuses and cause it to start misbehaving; this can be hard to catch so we have to be careful.

Since the results of our calculations are stored in Yazoo variables, there is no need to manually export data. We can simply delete the "save results" comment line in NN.c.

One small aside: when we declared the `ArgInfo` variable we used a data type that is defined by Yazoo in userfn.h; so we need to add

```
#include "userfn.h"
```

at the beginning of NN.c.

The final step is to make Yazoo aware of our new routine. For this we have to go back to Yazoo's 'userfn.c' source file. The only line we need to adapt is the definition of `UserFunctionSet`, which catalogs every C or C++ routine that Yazoo can access. To add a function of our own we need to give both the address of the function and a name (a string) that our scripts can know it by.

```
UserFunction UserFunctionSet[] = { { "Debug", &Debug }, { "Yazoo", &run_yazoo },
                                   { "RunNetwork", &run_network } };
```

And with that we are done. That is, the C code is fully integrated, and we should be able to recompile Yazoo and run our neural network from a script. Admittedly, there are things that we could have done better: in particular we should probably have included NN.h in userfn.c, and updated the makefile to include our source and header files. But what we have written should at least work.

To recompile: first make sure all source and header files, including NN.c and NN.h, are in the same directory as 'Makefile'; then go to that directory from the command prompt and type "`make yazoo`" (case sensitive). (The 'make' tool has to be installed for this to work.) With luck, we'll end up with an executable. To run it, type either '`yazoo`' or '`./yazoo`', depending on the system. We should see:

```
>
```

Once inside Yazoo, we will be able to execute our neural network routine by typing:

```
call("RunNetwork", param1, param2, ...)
```

Which begs the question of how to define and work with the parameters. The promise is that this will be quicker to do in Yazoo than in C. Our next order of business will be to write a neural-network class whose 'run' function will take care of all that overhead. Doing so will make the scripts more eloquent (no ugly `call()` statement), and will allow the user to leverage the flexibility of ordinary Yazoo function calls.

2.4 The Yazoo wrapper: writing and debugging

At this stage we're completely done with the C code. It's been written, integrated into Yazoo, compiled, and that the whole thing works, at least in the author's hands. Let's now try our hand at writing a Yazoo script to 'wrap' around our C routine.

NN.zoo

```
neural_network :: {
    neurons_num :: ulong
    inputs_num :: outputs_num

    weights[*][*] :: double
    activity[*] :: double

    init :: {

        hidden_neurons_num :: ulong
        args_copy :: { ins :: outs :: hiddens :: ulong }

        if trap( args_copy = args ) /= passed
            print("usage:  myNN.init(inputs, outputs, hidden neurons)\n")
            return 1
        endif
```

```
        { inputs_num, outputs_num } = { args_copy[1], args_copy[2] }
        neurons_num = inputs_num + outputs_num + args_copy.hiddens + 1

        activity[^neurons_num]
        weights[^0][^0]         | to speed up the resize
        weights[^neurons_num][^neurons_num]

        return 0
    }


    run :: {

        args_num :: rtrn :: ulong
        inputs[1] :: outputs[0] :: step_size :: learning_rate :: double
        inputs[1] = 1       // the 'bias' input

        code

        args_num = top(args)

        if trap(
            inputs[^inputs_num + 1]
            inputs[2, inputs_num+1] = args[1][*]
            if args_num == 4
                outputs[^outputs_num]
                outputs[1, outputs_num] = args[2][*]
                { step_size, learning_rate } = { args[3], args[4] }
            elseif args_num == 2
                step_size = args[2]
            else,  throw(1)
            endif
        ) /= passed
            print("usage:  myNN.run(input, step_size OR ",
                  "input, output, step_size, learning_rate)\n")
            return 1
        end if

        if args_num == 2
            rtrn = call("RunNetwork", weights, activity, inputs, step_size)
        else
            rtrn = call("RunNetwork", weights, activity, inputs, step_size, outputs, learning_rate)
        end if

        if rtrn == 1, print("run() did not converge; try lowering step size?\n"), endif

        return 0
    }

    init(0, 0, 0)
}
```

We should save NN.zoo in same directory that Yazoo, start.zoo and user.zoo reside in. (The start.zoo script provides the default command prompt; user.zoo pre-loads the command prompt with a number of useful routines which are listed in the reference chapter.) Once our own script is in working order we'll be able to apply our network to something useful.

To begin with, we need to make sure Yazoo is running in interactive mode – i.e. there should be a

command prompt. (The only reason it might not run interactively is that we may have run Yazoo with an extra argument. If you just type '`yazoo`' or '`./yazoo`' then by default start.zoo's interactive prompt will pop up.) In order to use our new `neural_network` class from the interactive prompt, we will need to execute the NN.zoo script via the `run()` function.

```
> run("NN.zoo")

run(): Syntax error in file "NN.zoo" on line 33: unknown command

        inputs[1] = 1       // the 'bias' input

                                ^
```

Hmm.. we're obviously not finished with NN.zoo yet. Fortunately this is a type-I error (not at all a technical term): Yazoo wasn't able to even read our script and it flagged the line where it got confused. It is the scripting equivalent of a compile-time error. In fact, Yazoo even calls its initial preprocessing 'compiling' even though it doesn't generate machine code. Type-I errors are usually the easiest problems to fix. We can see straight away what went wrong: we accidentally wrote a C-style comment '`//`' in place of a Yazoo comment '`|`'. Note that Yazoo somewhat missed the mark: the first thing it thought to complain about was the single-quote, which has no meaning it its language, rather than what looked to it like a double-division sign.

We make the straightforward fix to NN.zoo:

```
        inputs[1] = 1       | the 'bias' input
```

and try running the script again.

```
> run("NN.zoo")

Error in file "NN.zoo":  member 'outputs_num' not defined

3:      inputs_num :: outputs_num
```

We have made progress: NN.zoo succeeded in 'compiling'. Yazoo then tried running the script, made it part way (though it did print out a strange `usage:` message), but then impaled itself on line 3 and duly printed a complaint. Notice the different format of the error message compared with the first – in particular Yazoo was only able to flag the line at which the problem occurred, not any specific word or symbol.

Runtime errors, which is the second type of bug, are sometimes more difficult to debug than type-I errors since they more often result from problems upstream. Fortunately, that is not the case here. With a little knowledge of the scripting language we would see that we tried to define `inputs_num` to be a variable of type `outputs_num`, but the latter has not been defined yet. What we meant to do was to define both of type `ulong`, so we need to make the following correction to line 3:

```
    inputs_num :: outputs_num :: ulong
```

[When we run the script again on an *older* version of Yazoo, we get a different error, this time at an *earlier* point in the code than our last error.

```
> run("NN.zoo")

Error in file "NN.zoo":  type mismatch

1:  neural_network :: {
```

Why didn't we get this error last time? The reason is that a type mismatch error happens when we try to redefine an existing object to a new type. Technically the definition of `neural_network` did change since we corrected the last error, but unfortunately Yazoo would have thrown this error even if we hadn't since it doesn't bother to compare the contents of the old curly braces with the new. The current version has a smarter `run()` function that will automatically deletes old variables, but under older versions we must manually remove `neural_network` by adding the following *before* the definition of `neural_network` (i.e. before line 1):

```
trap(remove neural_network)
```

Note that we only have to remove the outer `neural_network` class, since all of its own members will then disappear. If writing `trap(remove ...)` gets tiresome.. upgrade!]

Let's re-run the script.

```
> run("NN.zoo")

usage:  myNN.init(inputs, outputs, hidden neurons)
```

Well, we managed not to get an official Yazoo error. Though it's puzzling why we keep getting this usage statement, which is supposed to indicate that the `init()` routine was invoked with bad arguments. The strange thing is that we never tried initializing a neural network. This is evidently a type-III bug, the most insidious kind: it doesn't cause a compile-time or runtime error, but it makes the program behave oddly. Type-III errors are best debugged from the command line.

Let's see if `init()` works when we try to invoke it normally. Try

```
> neural_network.init(3, 4, 5)


>
```

So far so good(?). There should now be 13 neurons in our network (including the 'bias' neuron).

```
> sprint(neural_network.activity)

{  }
```

So our `init()` call was a dud – nothing happened. Did it even attempt to run? We might try put a trace statement – say, `print("A")` – somewhere in the coding section of the `init()` function. Hmm.. what coding section? And so we discover that – aha! – we forgot the `code` marker, which explains all of our problems. Add it after the second line of `init()`; the function should now begin:

```
hidden_neurons_num :: ulong
args_copy :: { ins :: outs :: hiddens :: ulong }

code

if trap( args_copy = args ) /= passed
```

Having made this change, we can go back and try:

```
> run("NN.zoo"), neural_network.init(3, 4, 5)
> sprint(neural_network.activity)
```

```
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
```

Which is exactly what we were hoping to see: an array of neurons, initialized to a resting state and ready to start processing.

2.5 The Anagrambler

Of course, we would still have to run a few more tests to really convince ourselves that NN.zoo is working. In particular, we should try re-initializing the network, and still need to take the `run()` method out for a spin. As luck would have it the rest of the script turns out to work just fine. So at this point we can close NN.zoo, go back at the command prompt and put our networks to use.

The types of network and learning algorithm we are using are very well suited to the task of memorizing a set of output patterns in association with different inputs. In this example, we'll use our network to unscramble anagrams. There are a number of ways we could imagine doing this, the most obvious being to tell the network how many of each letter is found in the anagram, then read off the word directly from the output. That is, the value of each output neuron could encode a letter by its activity level – anything less than 0.05 might represent an 'a', 0.05 thru 0.1 would be a 'b', and so on. But the problem with this approach is that it will spit out garbage (with letters that weren't even in the anagram) unless the network can be trained to very high tolerances. So we'll use a more indirect approach that shows a more graceful performance curve.

Our strategy will be to associate letter order with letter count. The input will be the number of each letter in the anagram (so there will be 26 inputs). The output will, hopefully, be the ordering of the letters relative to an alphabetical ordering of the letters. For example, the output '32415' for the input 'ortob' would imply the ordering 3-2-4-1-5 of the characters 'b-o-o-r-t', which gives 'robot'. Different words require different output sizes, so we should set the number of output neurons to the maximum word length we expect to use. In our example 6 output neurons will be adequate.

To begin with, it's probably good practice to define a working variable that is separate from the template variable, as the latter was really intended to be a class. Ordinarily we might just want to do it from the command prompt, as in `Hob :: neural_network`. However, in this case we actually want to use the inheritance operator to give it a bunch of routines to read and re-spell words, which makes for a lengthy definition for the command prompt. So let's put it in a separate file instead: say, "Anagrambler.zoo", in the same directory as Yazoo.

```
foreach :: {

    code

    for (c1::ulong) in [1, top(args[1])]
        args(args[1][c1], c1)
    endf
}

Hob :: neural_network : {

    SetupNN :: {

        ltr :: string
        params :: { step_size :: learning_rate :: double }

        code

        params = { .5, .1 }
        if trap( (params<<args)() ) /= passed
            printl("Error: optional params are step_size, learning_rate")
```

```
            return
        endif

        the_word = args[1]
        foreach(NN_in ;
            ltr =! extract(alph, args[2], args[2])
            args[1] = find(the_word, ltr, 0)    )

        NN_out[^size(the_word)]
        NN_out[*].letter =! the_word
    }

    ask :: Hob.SetupNN : {

        out_str :: string

        code

        run(NN_in, params.step_size)

        sort(NN_out, 2)

        NN_out[^outputs_num]
        NN_out[*].order = activity[inputs_num+2, inputs_num+outputs_num+1]
        if size(the_word) < outputs_num, NN_out[^size(the_word)], endif

        sort(NN_out, 1)
        out_str =! NN_out[*].letter

        print(out_str)
    }

    teach :: Hob.SetupNN : {
        c1 :: ulong

        code

        foreach(NN_out ; args[1].order = args[2]/size(the_word))
        sort(NN_out, 2)

        NN_out[^outputs_num]

        for c1 in [1, args[2]]
            run(NN_in, NN_out[*].order, params.step_size, params.learning_rate)
        endf
    }
}


the_word :: string
NN_in[26] :: double
NN_out[Hob.outputs_num] :: { order :: double, letter :: ubyte }

alph := "abcdefghijklmnopqrstuvwxyz"
```

Just know that it would have been possible to do all this from the command line, just inconvenient. With direct command-prompt entry, lines must be separated by commas, with line-breaks only if we use the

line-continue marker '&': e.g.

```
> foreach :: { ; for (c1::ulong) in [1, top(args[1])], &
args(args[1][c1], c1), endf  }
...                  | etc.
```

At last, we're ready to put our newly-constructed brain to the task of unscrambling anagrams. We begin by booting up Yazoo, then running the two zoo source files.

```
> run("NN.zoo")

> run("Anagrambler.zoo")
```

Next we'll need to initialize the little brain. Let's decide to work with words of 6 or fewer characters—we can always change this—and we'll expect a 26-character alphabet. So we enter the following line at the command prompt.

```
> Hob.init(26, 6, 0)
```

Once everything is loaded into memory, we can try

```
> Hob.ask("lleoh")

ehllo
```

Hardly a surprise; we haven't taught it its first word yet.

```
> Hob.teach("hello", 10)    | 10 = # training cycles -- mandatory argument

> Hob.ask("lleoh")

hello
```

Cute. So this concludes our Yazoo demonstration. Of course we've barely investigated Hob's intelligence, which is pretty dim (though subtle in surprising ways). A great virtue of the interactive command prompt is that the cycles of investigation are very short. After `hello`, the author taught Hob `yazoo` and then `tomato` (10 cycles apiece), after which the network had gotten foggy on both `hello` and `yazoo`. Then a follow-up 5 cycles of '`yazoo`'-teaching, and Hob was for some strange reason able to recite all three words perfectly. It all took less than a minute.

Another thing to play around with is the learning rate and/or step size for evolving the neurons' activities. If runtime becomes an issue, will we be able to trade a higher learning rate for some training runs? It's simple to script.

```
> Hob.teach("hello", 5; learning_rate = that*2)
```

Finally, we could look into build a bigger brain. Increasing the number of output neurons would allow Hob to memorize longer words. Adding hidden neurons would increase both its memory capacity and the depth of its calculations. To explore these possibilities we would just re-initialize the network (`Hob.init(...)`) with different parameters and repeat the training.

That is the last that we shall say about neural networks – this is, after all, a Yazoo help file. The tutorial concluded, the remainder of this chapter will explain, rather than show, how in the general case one embeds C or C++ code into Yazoo.

2.6 Rules for embedding C/C++ code

The C version of Yazoo, whose source files end in `.c`, can be extended with user-defined C routines, and embedded into a larger C program. The C++ version of Yazoo, with source files ending in `.cpp`, is extensible with user-defined C++ routines and embeddable into C++ programs. The two versions are identical beyond the names of their source files.

The procedure for embedding C or C++ functions involves three steps. First, one declares the C/C++ routine in a particular way and writes the code accordingly; this is easy because the Yazoo format is almost identical to that for a stand-alone program. In the second step the author introduces his routine to Yazoo by modifying one of Yazoo's own source files. The final, optional step is to Yazoo-script a wrapper that allocates storage for the C/C++ function and provides it with a smooth interface.

The additional step required to embed Yazoo within a larger C or C++ program is to comment out the definition of `YazooMainProgram` in userfn.h. The embedding application can then run Yazoo by including `yzmain.h` and calling `run_yazoo(argc, argv)`. If there is a makefile then that will probably have to be merged with Yazoo's own makefile.

2.6.1 C function declaration

Yazoo runs embedded C or C++ functions via its a built-in `call()` routine. Those embedded functions must be of the following form (compare with C's `int main()`):

```
int MyFunction(int argc, char **argv)
{
    ...
}
```

Of course the names of the variables `MyFunction`, `argc` and `argv` are arbitrary. As in a `main()` function, when the function is executed `argc` stores the number of arguments to the function, and `argv` points to an array of pointers to the arguments. Thus `*argv` is the pointer to the first argument, `*(argv+1)` is the pointer to the second argument, and `*(argv+argc-1)` points to the last argument. Any given argument will contain either numbers (integers or floats), blocks of non-numeric byte data, or strings contained in linked lists.

Consider the following call from a Yazoo script:

```
names[2][5] :: double
...
call("MyFunction", 1, counter, names[*][*], 14)
```

This passes three variables and one array to a C function. It was a two-dimensional array, but all arrays get unravelled into one-dimensional lists before being handed off to C routines. The order of elements in the list is 1-1, 1-2, ..., 1-5, 2-1, ..., 2-5. Since the third `argv` argument is pointed at by `*(argv+2)`, the array element `names[2][3]` – eighth in the list – would be accessed in the C function by writing

```
double names_2_3;
double *names_array = (double *) *(argv+2);

names_2_3 = *(names_array+7);
```

In a sense each argument to a C function is treated as an array; simple variables are just arrays of size one. So the C code would pick up its first argument by writing `*(signed long *) *argv` (as `slong` is the default type for integers).

Strings are a bit more complicated. Internally, they are stored in linked lists that can change their sizes to accommodate strings of different lengths. In principle a string could be passed to C code as a byte array. That approach wasn't taken because it wouldn't allow the C routine to change the size of the string. So

instead, `call()` passes a pointer to Yazoo's own personal linked list that holds the string. This is quite risky on Yazoo's part, since it will suffer speedy or slow death if the user mismanages the linked list. The user is therefore encouraged to make use of Yazoo's tried and tested linked-list routines when dealing with string arguments; those routines are encoded in `lnklst.c` and described later in this document.

All numeric arguments are arrays (often of size 1), and all strings are linked lists. Putting two and two together, we can guess, correctly, that an argument that is a single string is an array, containing one element, of type `linkedlist` (defined in `lnklst.h`). Given the following Yazoo call

```
last_names[10] :: first_name :: string
...
call("Lookup", last_names, first_name)
```

an appropriate C interface might begin

```
#include "lnklst.h"

int Lookup(int argc, char **argv)
{
    linkedlist *last = (linkedlist *) *argv;         // must be a _pointer_
    linkedlist *first = (linkedlist *) *(argv+1);    // to the linked list
    linkedlist *first_last_name, *last_last_name;

    first_last_name = last+0;
    last_last_name = last+9;
    ...
}
```

(Note that we must include the linked list header file.) An *inappropriate* way of handling the linked lists would be to work with copies of the linked list rather than the pointers to the originals.

```
    linkedlist first_last_name, last_last_name;
    ...
    last_last_name = *(last+9);      // asking for trouble
```

The `linkedlist` variable itself stores important information that needs to be up-to-date; if the C routine were to resize the string, Yazoo would never know about it and would probably crash somewhere down the line.

One final service that `call()` provides is to furnish the `argv` list with an info table on the sizes and types of its arguments. This is useful for type-checking – since the user's C routine would probably crash if given parameters of the wrong size – and it also allows one to write C functions that accept variables and arrays whose types and dimensions cannot be known beforehand.

The info list is found after all the other arguments, at `*(argv+argc)`. It has one entry of type `arg_info` for each of the `argc` arguments that was passed. The `arg_info` type is defined in userfn.h: it contains the three unsigned longs `arg_type`, `arg_size`, and `arg_indices`. `arg_type` is the numeric type designation from Table 1. Composite types are listed in the table; however, the user will never see one passed to a function, since their primitive constituents are passed as separate `argv` entries. `arg_indices` is the number of array elements (remember, each argument is an array), which for multi-dimensional arrays is the product of ranges of each dimension. The `arg_size` field denotes the size, in bytes, of each element (not of the whole array), and is useful for `block` types which may be sized arbitrarily. Note that many of these quantities are machine-dependent, and that Yazoo rolls with the local C conventions on each computer. For example, machines with 32-bit addressing will have 4-byte long words, but 64-bit machines have 8-byte long words.

The info list is accessed in the same way as any of the other arguments.

type ID	type name	Yazoo name	C def name	byte size*	min value*	max value*
0	unsigned byte	`ubyte`	`Ubyte_type`	1	0	255
1	signed byte	`sbyte`	`Sbyte_type`	1	-128	127
2	unsigned short	`ushort`	`Ushort_type`	2	0	$2^{16}-1$
3	signed short	`sshort`	`Sshort_type`	2	-2^{15}	$2^{15}-1$
4	unsigned long	`ulong`	`Ulong_type`	4	0	$2^{32}-1$
5	signed long	`slong`	`Slong_type`	4	-2^{31}	$2^{31}-1$
6	single-precision float	`single`	`single_type`	2	-10^{38}	10^{38}
7	double-precision float	`double`	`double_type`	8	-10^{38}	10^{38}
8	block type	`block` N	`block_type`	N	N/A	N/A
9	string type	`string`	`string_type`	any	N/A	N/A
10	composite type	`{...}`	`composite_type`	any	N/A	N/A

Table 1: Variable types; machine-dependent quantities* are relevant to the author's 32-bit computer

```
#include "lnklst.h"

int MyFunction(int argc, char **argv)
{
    arg_info ArgsList;
    long counter;
    const char ExpectedTypes = { 0, 5, 7, 7 };
    ...

    if (argc != 4)  return 1;

    ArgsList = *(arg_info *) *(argv + argc);
    for (counter = 1; counter <= 4; counter++)
    {
        if ((ArgsList + counter - 1)->arg_type != *(ExpectedTypes + counter))  return 2;
        if ((ArgsList + counter - 1)->arg_indices != 1)  return 3;
    }
    ...
}
```

For readability, each Yazoo type has an associated name (e.g. `double_type`) which can be used in place of its number. This name appears in the third column of Table 1. The names themselves are defined in yazooh.h, so the user needs to include this header file in any source file that uses them.

2.6.2 Yazoo's 'userfn' source file

The most basic option controlling how Yazoo will be compiled is the following line in userfn.c or userfn.cpp:

```
#define YazooMainProgram
```

As is, this line will cause Yazoo's `int main()` function to compile, which is appropriate if Yazoo is to be a stand-alone application. Yazoo can also be embedded within a larger C/C++ application by simply commenting this section out:

```
// #define YazooMainProgram
```

In this latter case, to run Yazoo one would `#include "yzmain.h"` in the appropriate header, and make the following call in the executable code:

```
rtrn = run_yazoo(argc, argv)     | argc = 0 or 1
```

By setting `argc` to 0, Yazoo will try to run `start.zoo` in the default directory. If `argv` is 1 then the script to run should be stored in a file whose pathname is pointed at by `*argv`. Note that we omit the first `*(argv+0)` pathname that the command prompt traditionally passes to a C program.

All user-authored C or C++ functions make their formal acquaintance to Yazoo through the file `userfn.c` or `userfn.cpp`. (Which one of the two depends on whether it is the C or C++ version of Yazoo.) This file comes almost completely empty except for one short function named `Debug()`, which the author uses for setting breakpoints and printing trace statements, and a `run_yazoo()` function by which Yazoo can run itself recursively.

Following the example of `Debug()`, the first step in embedding a new function is to update the `UserFunctionSet[]` array by adding a new entry containing its Yazoo name (a string), and a pointer to the actual function. For example:

```
UserFunction UserFunctionSet[] = { { "Debug", &Debug },
                                   { "Integrate", &doIntegration } };
```

Note that the Yazoo name does *not* have to be the same as the name of the function, and it is not a variable name (so the user can still define an `Integrate` variable without causing any problems). This 'Yazoo name' is only the string by which the `call()` function will recognize that the user wants to run the `doIntegration()` function, as in

```
call("Integrate", initial_conditions, 5, temp_storage, result)     | run doIntegration()
```

As mentioned in the last section, `doIntegration()` must have been defined as a `(int)(int, char **)` or else there will an error in `UserFunctionSet[]` when Yazoo is recompiled.

The second step is to put the function in a place where the C or C++ compiler will be able to find it. The simplest way is to put the function directly inside `userfn.c/cpp` and prototype it somewhere above the `UserFunctionSet[]` declaration. Any header files should also be included at the top in the space provided. An alternative is to put the functions in separate source files, prototype the functions in separate header files, include these header files at the top of `userfn.c/cpp`, and add the new source and header files to Yazoo's makefile. The following changes would add (`Numerics.c` and `Numerics.h`) to a makefile:

```
...
SRC = yzmain.c cmpile.c ... userfn.c Numerics.c
OBJ = yzmain.o cmpile.o userfn.o Numerics.o
ALL_HEADERS = yzmain.h cmpile.h ... userfn.h Numerics.h
...
UserFunctions.o: $(ALL_HEADERS) userfn.c
Numerics.o: $(ALL_HEADERS) Numerics.c
```

The final step is to recompile Yazoo. The recommended method is to use the command-line tool `make`, which is available for free on many platforms. Of course it also requires that a C/C++ compiler like gcc be installed (some of these are free too). For simplicity all source and header files should be in the same directory as the makefile. To build Yazoo, navigate to that directory and type "`make yazoo`".

It is also possible to use a development environment for compiling Yazoo, in which case the makefile may not be necessary. The `#include` statements in `userfn.c/cpp` would probably still be needed.

2.6.3 Sample Wrapper

Compare the natural, compact syntax of a Yazoo function

```
y = f(x)
```

with the rigid formalism of an external function call:

```
error_code = call("f", x, y)
```

Wouldn't it be great if we could run our external functions using method number one. Well, for several reasons the author actually wants to keep the formal protocol of a `call()` command, but he almost always 'wraps' a Yazoo function around it to avoid having to deal directly with the `call()`. There are other advantages to doing so as well: some tasks, such as memory allocation, type checking and typecasting, are much more easily done on the Yazoo side than in C. Here we'll write a wrapper for our imaginary `f()` function.

```
f :: {
    x :: answer :: double
    error_code :: ulong

    code

    (answer =@ *) @:: double    | should always do this for return variable
                                | will explain later
    if top(args) /= 1 or trap(x = args[1]) /= passed
        print("Error in arguments to f\n")
        return *
    end if

    error_code = call("f", x, answer)
    if error_code == 0
        return answer
    else
        print("Error ", error_code, " in function f\n")
        return *
    end if
}
```

It may look like a lot of huffing and puffing but we have accomplished some important things. First, note that if the function is called with the wrong arguments or in a way that the C routine does not like, an error message will be printed and the function will exit gracefully (though it won't return a value, possibly causing whatever called it to crash as it rightly should). It often involves more overhead to do the type-checking in C.

More importantly, the Yazoo function `f()` will accept numeric arguments of any type, signed or unsigned integer or floating point, whereas the C `f()` is probably expecting a double. This is reflected in the way the wrapper was written — it explicitly defined the types all of the arguments to `call()`, including the return variable. The following example shows what happens when you don't.

```
call("f", 1, answer)
call("f", 1.5, answer)
```

In the first case the first argument to the C-encoded routine will be a signed long integer; in the second case it will be a double-precision floating point number. Due to this ambiguity the passing of constants to `call()` is to be discouraged. The user can, however, pass constants or variables of any type to the *wrapper* with abandon; Yazoo then performs the appropriate conversions before calling the C routine.

Depending on the function `f()`, the C program that performs the calculation may be rather involved. Oftentimes auxiliary memory is required, if for example `f()` is some heavy numerical thing. It is generally easier to do this memory allocation in Yazoo. If our function in the example above requires, say, two tables, one of whose entries needs indefinite storage, we might extend its wrapper with the following lines:

```
temp_1[3][5] :: ulong    | add these above the 'code' statement
temp_2[4] :: string
...
error_code = call("f", x, temp_1, temp_2, answer)
```

Using Yazoo's linked list routines the C program will be able to treat the `temp_2` variable as a dynamic array.

Finally, one can do some exotic things with a Yazoo function that are convenient for the top-level user. A favorite trick is used for functions that require many more parameters than the user will ever want to specify, although he would like to be able to modify them when he needs to. In Yazoo you can solve this problem by adding something like the following to the wrapper.

```
params :: {    | add these above the 'code' statement
    max_storage :: ulong
    convergence :: step_size :: init_size :: double
    file_to_write :: string      }
...
    | after the code statement, put:
params = { 10, 7.5, .001, 1e-15, "f_log" }
...
    | replace the first if statement with the following:
if top(args) /= 1 or trap(x = args[1]) /= passed &
        or trap((params<<args)()) /= 0
...
    | replace the old 'call' command with:
error_code = call("f", x, temp_1, temp_2, params, answer)
```

Now we call our function as before: `y = f(x)`, unless we want to, say, double the integration step size, in which case we do

```
y = f(x; step_size = that*2)
```

Although the wrapper resets the defaults with each function call, it subsequently invites the user to change these defaults. This sort of maneuver works because, in Yazoo, function arguments are themselves functions. Yazoo scripting is a topic that evidently requires somewhat more discussion than we have given so far, and that is the subject that will occupy most of the remainder of this document.

3 Yazoo scripting

3.1 Building and running Yazoo

The very first order of business is to make sure we have a working copy of Yazoo. Since Yazoo is intended to be embedded and/or extended with C or C++ routines at compile-time, it is distributed as a set of source files rather than as an executable. How to merge Yazoo with the external C/C++ code was explained in the last section. The simplest way to build the executable is to use the command-line tool `make` along with a C/C++ compiler such as `gcc`; free versions of these exist for many platforms. To build Yazoo with `make`, put *everything*—source files, header files, `Makefile` and '.zoo' files—into the same directory; navigate to that directory from a command prompt; and then type "`make yazoo`". This should, theoretically, create a `yazoo` executable in that same directory.

There seems to be no fail-safe way to package Yazoo so that it will build correctly on all machines. The makefile requires `gcc` (for C) or `gpp` (for C++), and obviously `make`, to be installed for Yazoo to be built. Different systems may have more idiosyncratic requirements. Most notably, a 'load-math' (`-lm`) option often has to be passed to the linker, necessitating the following change to the makefile:

```
yazoo: $(OBJ)
    $(CC) $(CFLAGS) -o yazoo $(OBJ) -lm
```

Inevitably, other problems will arise on some machines, but the author has not encountered them yet, and he can only trust in the resourcefulness of the user to deal with them.

To run Yazoo after it has been compiled, go to its directory and enter the command "`./yazoo`" (UNIX and Mac), or just "`yazoo`" (PC/Windows). Yazoo should present a command prompt. It will look like this:

```
>
```

3.2 Introduction to Yazoo scripting

To begin our talk on Yazoo scripting we'll first throw out, in no particular order, some of the simple and common programming operations that take up the lion's share of many source files. The following sections will elaborate on each of these more systematically.

The first thing to know when you are in Yazoo is how to get back out again in a dignified way. The command to escape Yazoo is "`exit`". Therefore, if the user is being prompted for a command (by the character "`> `"), then typing `exit` followed by ⟨enter⟩ or ⟨return⟩ will cause Yazoo to end gracefully. There is unfortunately no way to force the command prompt to appear, so if Yazoo gets trapped in an infinite loop, the user has to stop the program by some external means like Ctrl-C (UNIX/Mac) or Ctrl-Alt-Delete (Windows).

Simple Yazoo expressions involving numbers, variables and strings follow a pretty standard format. Numeric expressions are computed in the usual way with `+`, `-`, etc. Strings are surrounded by double-quotes, as in `"my_string"`. There is no use for the single-quotation mark in Yazoo. All variables *must* be defined before they are used; for this we use the define operator `:`, as in `var ::  type`. The most commonly used types are `slong` (signed long integers), `double` (floating point), and `string`. To make the variable an array, add the size of each dimension in square brackets following the array name. For example,

```
MyTable[5][7] :: double
```

defines a two-dimensional array. To access an array element, again use the square brackets.

```
print(MyTable[2][3])
```

It's usually convenient to define several variables or arrays of the same type together in the same 'sentence'.

We can do this by defining variable 1 to be of the same type as variable 2, which is defined as ... as variable N, which is defined as, say, an unsigned long.

```
var1 :: var2 :: var3 :: ulong
```

The following script shows some basic manipulations of variables and arrays. This one can actually be entered line-by-line from Yazoo's command prompt.

```
| program to find the length of the hypotenuse of a right triangle

sides[2] :: hyp :: double
ans :: string

print("Side 1:  ")
ans = input()
read_string(ans, sides[1])

print("Side 2:  ")
ans = input()
read_string(ans, sides[2])

hyp = (sides[1]^2 + sides[2]^2)^0.5
print("The hypotenuse has length ", hyp, ".\n")
```

The first line in this example is a comment, because of the vertical bar `|`. The functions `print()`, `input()` and `read_string()` (along with some 40 others) come hardwired into Yazoo. We can also define our own scripted functions, for which we again use the define operator, but with the function script inside braces in place of a variable type. Inside the braces, variable definitions come first, then a `code` command or a semicolon, and lastly the executable code of the function. To exit the function (as opposed to the bailing out of the whole program) use the canonical `return` statement. One calls the function in the same way as one would in C, with the arguments in parentheses immediately after the function name. As we will see below, the function accesses these arguments through something called the `args` variable, which has one member for every argument that was passed. All arguments are passed by reference in Yazoo: if we call `f(x)` then the function `f()` can change the value of '`x`' by overwriting its `args[1]` variable.

Let's make our earlier example a bit more sophisticated by using functions, and also by the way demonstrate `if` commands and `for` loops. Unfortunately we cannot enter this example line-by-line, because neither the function definition nor the `if`/`while` blocks fit on one line. Instead we would have to enter it into a file: say, "pythagoras.zoo".

```
| program to find the length of the hypotenuse of a right triangle

GetSide :: {
    ans :: string
    side_length :: double

    code

    side_length = 0
    print("Side ", args[1], ":  ")
    ans = input()
    read_string(ans, side_length)

    if side_length > 0
        return side_length
    else
```

```
        print("Side length must be positive\n")
        exit
    end if
}

sides[2] :: hyp :: double
counter :: ulong

for counter in [1, 2]
    sides[counter] = GetSide(counter)
end for
hyp = (sides[1]^2 + sides[2]^2)^0.5

print("The hypotenuse has length ", hyp, ".\n")
```

Now to run our script from Yazoo's command prompt, we would type

```
run("pythagoras.zoo")
```

where we are assuming that the script is in the same directory as Yazoo.

In addition to `if` and `for`, Yazoo has `while`...`end while` and `do`...`until` loops. There is no 'goto' statement. One quick note: the logical 'if-equal' symbol is the same double-equate '`==`' that you find in C, in order to distinguish it from the assignment operator; thus for example `if a == b` is syntactically correct. The 'if-not-equal' operator is '`/=`'.

3.3 Expressions

This section deals with the micro-structure of a Yazoo script: how the various words and symbols fit together, without bothering just yet about what they do or how they work. Let's ignore for now the grossest level of organization, which is the grouping of code by curly braces {}, since those are explored in later sections. The various flow-control commands, such as `if` and `while`, in some sense constitute a second organizational level, which we'll also describe separately. Within and between these blocks are chunks of code that execute straightforwardly from beginning to end, whose anatomy we are going to study here.

There is an obvious third layer of organization even within a simple code chunk: that code can be broken into lines, separated of course by line breaks (either carriage returns or end-of-line characters). These are the Yazoo equivalent of sentences: they represent complete thoughts (instructions). There are two 'sentences' in the following code fragment:

```
a := 2*7 + 9
print(a)
```

Sentences can also be marked off by commas, so there are also two sentences here:

```
a := 2*7 + 9, print(a)
```

This latter form is useful when several sentences should be entered together on one line. For example, to execute a for-loop from the command prompt we have to tap in the `for`, `end for` and everything in between all on the same line, which we can do using commas.

The other option is to break sentences over multiple lines using the line-continuation symbol '`&`', as in:

```
a := 2* &
    7 + 9
```

For one thing, this lets us enter multi-line commands at the command prompt. (The command prompt at present is very fussy however, requiring that the `&` be the *very* last character entered from the previous line – no trailing spaces.) It can also improve the readability of long sentences in scripts. Generally a sentence can be broken between any two names or symbols, but not within a name or symbol or string, and not between the name of an array and its opening bracket, or between a function and its opening parenthesis.

Most two sentences can be logically written in any order (even if they won't work in the wrong order). But within the sentence there is some internal structure – the various symbols inside can't just be moved around willy-nilly. The constituents of a sentence are variables and constants, and operators. Operators allow large phrases to be built up, since they glue variables and constants to one another and to other operators. To start with, let's dissect the following sentence:

```
a = b + 2
```

There are two operators: '`+`' which sums its two arguments (the variable `b` and the constant `2`), and '`=`' which sets the value of the left-hand argument `a` to the value of the right-hand argument. The right-hand argument to '`=`' cannot be `b` because the rules of syntax dictate that the two operators cannot share this same argument; therefore it is actually the entire sum `b+2`. The net effect is to store the sum `b+2` in the variable `a`.

There actually could have been two valid interpretations of our example, which we can make explicit with parentheses.

```
a = ( b + 2 )
( a = b ) + 2
```

The first line indicates that `b+2` is the right-hand argument to '`=`'; in the second line `a = b` is the *left*-hand argument to '`+`'. These are both valid Yazoo sentences; parentheses can be always placed around any group of operators in the same sentence in order to *force* a certain grouping of terms.

But as we indicated, sans parentheses Yazoo always interprets `a = b + 2` in the former sense. How do we know that the convention is for the equate operator to absorb the summation rather than the other way around? The answer is that there is a rule that says that, in the absence of parentheses, sums are always grouped within equates. Sums have *higher precedence* than equates: they are evaluated *first* so that their output can be handed off to equate. Likewise multiplications and divisions are always contained within additions and subtractions, so `3 + 5 * 6` groups as `3 + ( 5 * 6 )`. Multiplication has higher precedence than addition. The precedence of all Yazoo operators is given in Table 2, in order from highest (1st priority) to lowest (13th priority) (so high precedence equals low priority number).

The final column in the Table 2 specifies how operators of the same precedence level are grouped when they are found in series. Does the phrase `8 / 4 / 2` mean `(8 / 4) / 2`, which gives 1, or `8 / (4 / 2)`, which gives 4? Again, we could force either interpretation by writing the parentheses; but from the table we see that, in their absence, operators in the multiplication/division level are grouped from left to right, so `(8 / 4) / 2` is the correct interpretation. On the other hand, equate operators as in `a = b = c = 2` are grouped right-to-left, so the grouping of this expression would be `a = (b = (c = 2))`.

The two characteristics of an operator are the effect it has on its arguments, and the value it returns. An operator must return something in order to be used as an argument to an encompassing operator. A handful of commands, such as the `remove` and `delete` commands, are operators that indeed do something but do not return any value, so these cannot be embedded in larger expressions. By contrast numeric operators and comparison operators do not do anything to their arguments, so to be useful they must themselves be arguments of an encompassing operator to which they can deliver their result. The various define and equate operators are hybrids: they first alter their left-hand arguments, by rewriting or redefining them, then they return the altered variable if they are embedded in a larger phrase.

Each argument of a Yazoo operator expects a certain category of expression, and using the wrong category will cause an error. For example, because all of the arithmetic operators expect two numeric arguments, trying to add, say, a number to a string will cause an error. Conditional or logical expressions such as `1 > 2` return true or false values that, unlike in C, are non-numeric and can only be used in `if`, `while` and `do`

priority	commands	symbols	grouping

1	function calls
	step to member
	step to index/indices
		`()`
		`.`
		`[] +[] [+]`
			left to right

2	raise to power	`^`	left to right
3	negate	`-`	N/A
4	multiply, divide	`* /`	left to right
5	add, subtract	`+ -`	left to right
6	byte block	`block`	N/A
7	append code	`:`	left to right
8	substitute code	`<<`	left to right
9	define/equate forced equate	`::  ::@ = := :=@ @::` `=!`	right to left
10	conditionals	`== /= > >= < <=`	N/A
11	logical not	`not`	right to left
12	logical and, or, eor	`and or eor`	left to right
13	commands	`return remove delete`	N/A

Table 2: Order of operations

statements; `a = b and c` is completely illegal. The 'braces operator', as in { `a :: ulong; print(a)` }, returns code, which is a separate substance from numbers, strings and conditionals. The rule for a define or equate statement is that its left-hand argument must be a variable, and although that variable may be of any type – numeric, string, or 'code' (function) – both left and right arguments must have types that match or can be made to match (e.g. by integer-to-float conversion).

In any sentence there is always one top-level operator whose arguments encompass the entire sentence. Consider the two sentences

```
(a :: ulong) = B^2 + f(2)
(x^2 + y^2)^0.5 - x - y
```

After a bit of thought we can see that in sentence 1, the top-level operator is the equate '=', and in sentence 2 it is the *second* subtraction '-'. The top-level operator in sentence 1 is useful, in that when all else is said and done that equate still has some useful task to perform, which is to change the value of variable `a`. The top-level operator in sentence 2 seems less useful, since the subtraction operator only returns a value without affecting any variable, and at the top level there is nowhere to return the result to.

So, presumably whatever number sentence 2 computes will disappear into outer space? Well, actually no. When the Yazoo compiler hits some useless sentence like this (or at least one whose top-level operation is useless), it will fashion it into something called a 'token'. A token is a sort of placeholder for some value that was computed so that it can be recovered later if needed. We will come across tokens several times later on; they are especially useful in defining sets, and (as we have already seen, if you think about it) in passing arguments to functions.

3.4 Working with variables

The previous section described the anatomy of a Yazoo sentence. In these next few sections we'll do some microscopy on the various objects that comprise the sentence, what they do and how they work. Here we'll describe the most basic and ubiquitous object, which we hope needs no introduction: the variable.

3.4.1 Primitive variables

Most of the variables we have encountered so far are the 'primitive' variables that store data: as a number, a string, or a 'block' (sequence of bytes). Only numeric variables can be operated on by the arithmetic operators. Variables defined as byte blocks can be sized to arbitrary numbers of bytes, although they cannot subsequently be *re*-sized. Strings boast dynamic storage: each time data is copied into them they are automatically resized appropriately.

The numeric types are: `ubyte` and `sbyte` (for unsigned and signed bytes), `ushort` and `sshort` (short-integer words), `ulong` and `slong` (long words, machine-dependent but usually 4 or 8 bytes), and `single` and `double` for single- and double-precision floating-point numbers. These are also listed in Table 1 on page 18, which also gives the range of values each type can store. As mentioned before, variables are defined by the define '`::`' operator, and multiple variables of the same type can be defined on the same line.

```
height :: time :: g :: double
```

The variables are defined in the order left-to-right, so in this case `height` is the first floating-point double to be defined. All primitive numeric and block variables begin life initialized to zero, and all strings are initialized a no-string state (distinct from the zero-character string). To change the value stored by a variable, use the equate operator '='.

```
time = height = 0
g = -9.8
units = "m/s^2"
```

The type name for defining byte blocks is `block`, and the type for strings is, logically, `string`. The `block` type actually takes one parameter: the number of bytes to allocate for the variable. For example, the syntax for defining a 3-byte `triplet` variable is:

```
triplet :: block 3
```

Byte blocks are not useful very often and the author has toyed with the idea of taking them out altogether. The scenario that made him hesitate is the case of reading data with some fixed-width field via `read_string()`. In this case strings may rescale incorrectly, and `read_string()` might attempt an inappropriate string-to-number conversion if a numeric type is used instead.

3.4.2 Composite variables

All variables in Yazoo are either primitive or composite. The composite variable of Yazoo is quite a multifaceted object, but from our present perspective it is simply a collection of primitive variables. It is the analog of the `struct` in C. You can define composite types and create variables of those types in the following way:

```
StreetAddressType :: {
    number :: ulong
    street :: string
}

PetersPlace :: StreetAddressType
```

The first four lines define the composite `StreetAddressType`, using the same operator (`::`) that we use for defining data variables. Here the type name goes to the left of '`::`', and the type definition goes to the right in braces. Line breaks (or commas) separate the fields (members) in the type definition from each other

(and, optionally, from the closing brace); however, the *opening* brace must appear on the same line as the define symbol `::`. In the final line, `StreetAddressType` is used to coin the new variable `PetersPlace`.

As in C, we can use a period to access the various members of a composite variable. Thus after defining `PetersPlace` we might initialize it with something like

```
PetersPlace.number = 357
PetersPlace.street = "Gooseberry Drive"
```

Types may be nested within one another, either by using previously-defined types within the definition of a new type, or by inlining one type's definition inside another's. Both methods are shown below.

```
FullAddress :: {
    first_line :: StreetAddressType
    second_line :: {
        city :: state :: string
        zip :: ulong
    }
}
```

In this case, we would have to make double use of the '`.`' to access any of the primitive fields.

```
mailing_address :: FullAddress

mailing_address.first_line.number = 357
...
```

Yazoo even allows the user to use types that were defined inside other types.

```
general_whereabouts :: FullAddress.second_line

general_whereabouts.city = "Detroit"
...
```

One peculiarity of composite variables is that their internal structures can be rearranged after they have been defined. For example:

```
PaulineAddress :: FullAddress

remove PaulineAddress.first_line.street    | it's a PO box
PaulineAddress.country :: string           | in another country
```

This should be pretty self-explanatory: by the end `PaulineAddress` will have a third member (as if `country :: string` had appeared inside the definition of `FullAddress`); also the `street` field will be missing. Evidently `delete` removes members within variables as well as whole variables.

There are instances in which a composite variable will be created without an explicit type definition. Suppose that, out of the blue, we were to define

```
(James.first_line.street :: string) = "Halfway Ave."
```

without ever having defined `James :: FullAddress`. In Yazoo this is quite legal (though it wouldn't be if this were the only use for it)—we shall call it an implicit variable definition, and it basically accomplishes the following:

```
empty_type :: {}

James :: empty_type
James.first_line :: empty_type
James.first_line.street :: string
James.first_line.street = "Halfway Ave."
```

In other words, when Yazoo has to step through a variable which doesn't exist *in the process of defining a new member*, it will create an empty composite variable to which it will then immediately add a member. The left-hand argument of a define operator is the only circumstance in which members are created implicitly. The difference between a composite variable defined with one member and an initially empty variable with an added member is indeed significant, as we will see presently.

Most non-numeric operations that can be applied to primitive variables also work with whole composite variables. For example, the following code is quite acceptable, since both the equate '`=`' and comparison '`==`' operators can take non-numeric arguments provided that their types match.

```
Tom :: Bob :: StreetAddressType
...
Tom = Bob
if Tom == Bob      | yes, they will be equal
    print("They're sharing a room.\n")
endif
```

In this context, two variables have matching types only if their fields are identical, or can be straightforwardly converted (i.e. having corresponding fields of different numeric types). So had we defined `Bob` separately using a `BobsAddressType` that we wrote just for him, we would be able to copy and compare `Tom` and `Bob` if and only if `BobsAddressType` consisted of a (any) numeric type followed by a `string`.

* * *

Thus far we've been talking as if types and composite variables are actually two different things in Yazoo. In fact they're not. So, we can write things like:

```
Tom :: {
    number :: ulong
    street :: string    }

Tom.number = 63
remove Tom.street

Bob :: Tom
```

What just happened? First, when we defined `Tom` we were actually defining a composite variable with such-and-such structure, not a type per-se. Second: the define statement of the last line created the new variable `Bob`—read—"of the same type as variable `Tom`", not "of type `Tom`". More precisely, `Bob` is defined as `Tom` *was originally* defined, which is a `ulong` set to zero followed by an empty `string`.

So there is no distinction between, say, `Tom` and `StreetAddressType` from our earlier example: both are identically-structured composite variables. The fact that one happened to be defined directly and the other happened to have its definition copied from another variable is immaterial as far as Yazoo is concerned. One wonders whether we might also be able to define primitive variables off of each other too:

```
num1 :: ulong
num2 :: num1
```

Indeed we can. The conclusion: variable 'type' is some intrinsic quality of variables – no hard object that is only 'type' exists in Yazoo. (Though pretending otherwise, as we were doing earlier, can be a useful fiction if it improves the readability of our code.)

3.4.3 Other equate operators

As we saw in the last section, the define operator '`::`' is very similar to the equate operator '`=`', except that it copies 'type', not data. It copies *only* type. So, following an example from the last section, if we execute

```
Tom :: { ... }

Tom.number = 63

Bob :: Tom
```

then `Bob`'s street number will be 0 even though `Tom`'s street number is set to 63. `Bob` was defined as a variable of of `Tom`'s type, and like `Tom` it began life with all numeric fields initialized to zero.

There is a way to define `Bob` so that it inherits `Tom`'s data as well as its type, which is to use the define-equate operator '`:=`'. So if we had instead defined `Bob` by writing

```
Bob := Tom
```

then both `Bob` and `Tom` would have their `number` field initialized to 63. This is equivalent to

```
Bob :: Tom
Bob = Tom
```

The define-equate operator has problems when the type definition does not match the actual fields in the variable. The issue is that equate copies data from the fields that are actually present in a variable, whereas define creates fields based on the original type definition. Had we modified our template variable `Tom` so that its field structure no longer matched its original and immutable type definition, we would land ourselves a type-mismatch error.

```
Tom :: { ... }
Tom.country := "USA"
Bob := Tom          | will cause an error
```

Here we've defined `Bob` according to `Tom`'s intrinsic type, which is the structure in braces; then the equate half of the '`:=`' operator fails because `Tom`'s actual contents differ from that intrinsic type by the extra `country` field.

The define-equate operator often also fails when the template variable was defined implicitly. Again, it is for the same reason: in an implicit definition the intrinsic type is set to a blank {}, but the contents of the variable were promptly increased to one member (see last section's discussion). As we see later, arrays are generally defined implicitly, so they usually cannot be used as a template for define-equate—this is probably the major practical limitation of this operator.

There are actually over a hundred members of the define-equate operator superfamily, of which bread-and-butter define ('`::`') and equate ('`=`') are the two most prominent. Only a handful of the others, such as '`:=`', have symbols that make them accessible via ordinary scripting. The next two sections will dwell in part on most of the remaining symboled operators. A further few operators which are used discreetly by the compiler will be described in a later chapter.

One operator superficially resembles equate, although it is technically an unrelated branch on the family tree. It is called 'forced-equate' and is given the symbol '`=!`'. Like simple equate, forced-equate copies data between two variables, but they copy data in different ways and have different restrictions on their two arguments. A simple equate copies data from field to field, so the data structures contained within its two arguments must have the same number of primitive elements in the same hierarchy, and each corresponding primitive field must be of the same or a compatible type (i.e. integer to floating-point). By contrast, a forced equate simply takes the data contained in its right-hand argument and stuffs those N bytes in the same order into the left-hand variable. In a forced equate the byte sizes of the two variables must be the same, but there are no restrictions on how the storage space is parceled out within the destination variable.

When forcing an equate from an inlined numeric constant, remember that Yazoo interprets constants as either as signed long integers or floating-point doubles, according to the convention described in the earlier section on expressions. So on the author's old machine where long integers are 4 bytes and doubles are 8 bytes, `a =! -4` and `a =! 2e5` will copy 4 bytes while `a =! 4.` and `a =! 2e10` will each copy 8 bytes.

An example where equate may be used but forced-equate fails is the following.

```
pixel_coord :: {  x :: y :: z :: ushort  }
real_coord :: { x :: y :: z :: double  }

...
pixel_coord = real_coord    | will pass
pixel_coord =! real_coord   | will fail
```

The reason for this is that double-precision floating points generally take up 8 bytes whereas short integers usually take up only 2.

In the next two cases a forced equate can be used but not a simple equate.

```
var1 :: {
    part1 :: {
        a :: double
        b :: ulong    }
    c :: sshort
}

var2 :: {
    a :: double
    b :: ulong
    c :: sshort
}

var1 =! var2     | OK so far
var1 = var2      | no, this will not work
```

Here, although both `var1` and `var2` contain the same number of primitive variables, equate gets tripped up by their different groupings whereas the forced-equate is insensitive to this. Also:

```
var1 :: {    a :: b :: c :: d :: sshort    }
(var2 :: double) = 6.28319

var1 =! var2     | works on the author's old machine
var1 = var2      | won't work on any machine
```

Forced equate happily distributes the bit-representation of `var2` among the four short integers of `var1` (though the subsequent contents of `var1` will be hard to interpret).

One nice thing about a forced equate is that it is especially likely to work (as in, not give a type-mismatch error) when a string is involved on the left-hand side. The reason is that, because a string's storage space is flexible, forced-equate will try to size the string to accommodate whatever data is not soaked up by any fixed-storage fields. (If there are multiple strings then the first one absorbs all the extra data.) So if we write, for example,

```
chars :: {
    one :: ubyte
    two :: string
    three :: ubyte   }

my_string := "Yazoo"
chars =! my_string
```

then 'Y' and the final 'o' are encoded in the primitive members `one` and `three` respectively, while `two` will contain 'azo'. Of course we still have to copy at least 2 bytes of data to fill the fixed-storage field: replacing the last line with `chars =! "Y"` will not work.

3.4.4 Aliases

The original plan was for Yazoo to be a 'neural-network language'. Clearly the program evolved into quite another beast, but most of the organs of the ancient creature are still around in one form or another, having been co-opted for new roles rather than replaced altogether. The neuron became a primitive variable; a grouping of neurons we now call a composite variable; and the synapse that wired those original circuits together turned into something called an alias. After all the layers were added aliases came to have a pivotal backstage role in the language: they are routinely behind the scenes when the user does something like create sets or pass function arguments.

An alias is a variable that shares its data with some other variable. In C the closest equivalent of an alias would be a pointer to another variable. This analogy is not quite exact, however: Yazoo treats a variable and its alias in *exactly* the same way. The alias does not have to be dereferenced with a '`*`' or a '`->`', and does not have a different type specification from its target as in C. In fact, Yazoo itself does not know which was the original variable and which is the alias.

We can define an alias in the following way:

```
a := 2    | will default to a signed long
b :=@ a
c :=@ b
```

(So not only can we make an alias to '`a`', but we can also alias its alias, which accomplishes precisely the same thing.) If we were to now `print(a)`, `print(b)` or `print(c)`, we would uniformly get back the number 2. If we were to set any of the three variables to a different value:

```
c = 3
```

then printing any of `a`, `b` or `c` would then cause '3' to be printed.

An alias is not permanent; it can be reassigned to point at another variable. This is done using the equate-at operator '`=@`'.

```
a :: b :: ulong
a = 2
b = 3

c :=@ a
```

```
print(c)
c =@ b
print(c)
```

Importantly, both `a` and `b` have exactly the same type. Had their types been different, retargeting `c` on the second-to-last line would have caused a type-mismatch error, even if both `a` and `b` were numeric. The difference between the `:=@` and `=@` operators is that the former defines as well as targets aliases; the latter only retargets. (In this case we could actually have used `:=@` both times—if the variable it wants to define already exists then it will not complain as long as the types match—but there are certain cases in which we will definitely want to use `=@`.)

We mentioned that an alias is identical in all respects to the variable it points to. Let's follow this point to its logical conclusion. We just saw that our alias can be reassigned, but Yazoo does not remember which one was the alias. So, we could have reassigned the target of the original variable instead and gotten away with it.

```
a :: b :: ulong
c :=@ a
a =@ b
```

Yazoo is perfectly happy for us to reassign variables as aliases: all variables are potential aliases. And all aliases are variables. We can get away with the following acrobatics:

```
a :: b :: c :: d :: ulong
a = 2, b = 3, c = d = 4
a =@ b
b =@ c
c =@ d =@ b =@ a
```

So in the end, what numbers do `a`, `b`, `c` and `d` contain? Clearly they all store the same value—the last line guarantees that. One sensible guess is that the value would be 4, but the correct answer turns out to be 3. The reason: when we reassigned `b =@ c`, the alias '`a`' was still left pointing to `b`'s *original* data, which is the number 3. This follows, again, from Yazoo's ambivalence between the original (`b`) and the alias (`a`): if on that line we had instead written via `a =@ c` then the original '`b`' would have been unaffected, so why shouldn't reassigning `b` leave `a` alone?

To clarify the situation, we need to make the important distinction between members and variables. Until this point we have been rather sloppy on this point, and we shall have to be more careful from now on. A member is a named object in Yazoo: the name of a variable, or a field in a composite variable. The variable itself is the data that the member refers to. In Yazoo these two things are entirely separate, and there is no reason to require a one-to-one correspondence between them. After `a =@ b` in our last example we have two members, `a` and `b`, pointing to one variable: the original variable of `b` which holds the number 3. The subsequent `b =@ c` simply retargets member `b`, but that has nothing whatsoever to do with member `a`, which remains firmly pointed at that same number-3 variable.

The equate operator '`=`' copies data between two variables, and it has a natural counterpart in the comparison operator '`==`' which tests for equality of the data between two variables. The equate-at operator '`=@`' copies the reference between two members. To complete the symmetry, Yazoo also has a reference-comparison operator '`==@`' which tests to see whether two aliases point to the same thing. A reference-comparison will return true if and only if the two members have to point to the same variable, and if that variable is part of an array (see next section) then the members must point to the same array elements. Finally, there is a reference-not-equal operator '`/=@`' which is the logical negation of '`==@`'.

One syntactic point: we can give our scripts a dash more elegance by putting space (as much as we want) before the '@' of any of the aliasing operators. So a couple of lines from one of our earlier examples could have been written

```
c := @a
a = @b
```

The extra space makes it clearer that the relationship between '`=@`' and '`:=@`' is analogous to that between '`=`' and '`:=`'. It also highlights the similarity between '`=@`' and '`=`': in both cases the operation will cause the left-hand member to point to the value contained in the right-hand argument, only by different means. For similar reasons we may prefer to write our reference comparisons as `if a == @b` or `if a /= @b`.

The extra space before the alias symbol may tempt suspicion that just the '`@`' is by itself some new alias operator, something maybe similar to the address-of operator '`&`' in C – but now we have pushed the analogy too far. The `@` symbol by itself is meaningless; it is only the last character of the four aliasing operators.

3.4.5 The void

We saw in the last section that the equate-at operator a) unhooks a member from its variable, then b) wires it up to a different variable. As we shall see now, there is a way to skip (b) and just leave a member pointing out into the ether.

```
temp1 =@ nothing
```

An alternative syntax is:

```
temp1 =@ *
```

The asterisk '`*`' and '`nothing`' are synonyms for the void operator. After running either of the lines above, we will not be able to do anything with `temp1` until we re-alias it to someplace else. If we are unsure whether an object points into the void, we can test it using the reference-comparison operator:

```
if temp1 == @nothing
    print("out of order..\n")
endif
```

One obvious benefit of having a void is that it's good for storing members that we temporarily don't need. The void is a place of naught, no data and no code, and members stashed there don't use memory, jam arrays or otherwise bother our program. Yazoo won't allow us to accidentally use a void member for any data transactions: for example, `temp2 = temp1` now gives us a void-member error. If we decide we need `temp1` again, we can always reassign it new storage space (e.g. `temp1 :: string` if it was a string) or hook it on to an existing variable (`temp1 =@ str_7`) and it will work again as new.

There are actually two uses of the word 'void' in this document that are important to keep logically separate. A void member in the sense we were just using is one that has no storage space. Alternatively, a void-typed member is one with essentially no type restriction on what it can point to. An absence of type is considered by Yazoo to be the most general type, and this gives void-typed members some special uses. By making a definition like `a :: *` we are doing *two* things: we are defining a member '`a`' with no type, and we are neglecting to give it storage space. Yazoo accomplishes both of these by use of the void operator.

Members that are defined with a void type can later be *specialized* to any other type by a second use of the define operator '`::`'. Circumstances actually arise where one may want to define a member whilst keeping options open as to the type of variable it will target: for example in the case of optional function arguments that the user defines. In that situation we postpone the type definition by initially defining the member void, and then later specialize by redefining it as another type.

```
any_var :: *
...
any_var := 2.5    | turn any_var into a double
```

Specializing a type—restricting to a subtype of its original type—is always allowed in Yazoo. In fact, that is the *only* type change that is ever allowed. One can never take a `slong` member and turn it back into `nothing`; it must be removed and reallocated.

Members that have no type are actually most useful in and of themselves. They can alias any variable.

```
any_var :: *
x :: slong
y := "Hello"
z := { z1 :: double, z2 :: single }

any_var = @x
any_var = @y
any_var = @z
```

Normally, when the define operator creates a new member, it both assigns it a type and gives it a new variable. But, as a child of the void, member `any_var` has a nil type and hence can target any variable in the program.

The culmination of the previous section was in making a distinction between members and variables. We can see that this distinction muddies the notion of 'type' that we have been bandying around. After `any_var` is aliased to `x`, does `any_var` have a null type or is it a signed long? Well, that depends on whether you are asking about the member's type, or the type of the variable it points to. Both members and variables have types, and in general (as in the example above) they may be different.

Our example foreshadows a point that we will flesh out later on: a member's type specification determines which variables it is allowed to point to. The rule is that a member can only target variables with the same type, or having a sub-type derived from its own. (Sole exception: all members can target '`*`', but the void hardly qualifies as a variable.) Since any type specification can be thought of as nothing plus whatever it is ($x = 0 + x$), all types are, in a sense, subtypes of the void. A member with a void type is therefore compatible with any target variable.

Now that we have seen how a member's type may differ from its target's, we are at last in a position to introduce two further, comparatively obscure members of the define-equate operator family.

The variable-define operator '`@::`' is identical to ordinary define (`::`), except that it only acts on the variable while leaving the member's type unchanged. In the example below, it is used twice to allocate storage for an untyped member.

```
any_var :: *
any_var @:: slong
...
any_var =@ *
any_var @:: string
```

Two points are worth making here. 1) We could almost have gotten away with using the ordinary define operator on the second line (`any_var :: slong`). But doing so would have specialized `any_var` to a `slong`, resulting in a type mismatch on the last line. 2) Unlinking `any_var` (2nd-to-last line) was necessary because define operators will never make a new variable unless they are forced to, either because they have just created the member or because the existing member was pointing into the void. Had we left out the void-aliasing step, Yazoo would have tried to redefine the existing `slong` *variable* as a `string`, causing a type-mismatch error.

By the way, the first line in the last example above was actually unnecessary. The variable-define operator will create a new member if none exists, but that member's type will be void regardless of the type of the variable that was created. Note that unlike '`=@`', we cannot insert a space anywhere in the operator: '`@ ::`' will cause a syntax error.

The member-define operator '`*::`' is the counterpart to variable-define: it operates only upon the member, without affecting any variable that member may target. It gets its symbol from the fact that, if one

uses member-define to define a new member, it will indeed create a member of the desired type but it will not bother to create a variable for it, so the member will initially have no target.

```
ul1 *:: ulong
print(ul1)      | will cause an error
```

The above code causes a void-member error since `ul1` has no associated variable. (The 'void' in void-member always refers to the lack of a member's *target variable* – its *type* is still `ulong`, not void.)

On the other hand, if we were to use member-define on a pre-existing member, then that member would just stay linked to whatever target variable it was pointing at (the void if none). As with the standard define operator, member-define can specialize a void member's type; but a specialized type such as `ulong` can never be generalized back to the null void type.

3.4.6 This and that

Yazoo takes the libertarian view that a function should be allowed to do whatever it wants to itself, although it can only alter a variable outside of by naming that variable and its lineage. In general these rules are automatic with a C-style syntax for naming variables. Suppose we have a multilayered set of definitions like

```
functions :: {
    f1 :: { str :: string ; ... }
    g1 :: { ... }
    ...
}

...

f2 :: functions.f1
```

Both `g1` and `g2` will be able access their respective members `str` without interfering with one another, though they cannot access members outside of itself with confidence because their code does not know its path. (The statement `x = g1()` may work inside of `f1()` but it will cause `f2()` to crash.) This is all consistent with the libertarian rule. But there is one significant way in which the naming convention does not live up to the rule: functions have no way to reliably name *themselves*. `f1()` can print itself—as it should be allowed to do—by writing `print(f1)`, but that won't work inside `f2()` because its own name was changed. What a function needs is a way of naming one's own variable without having to find out the name of some member that points to it.

Enter '`this`'. `this` (with a lower-case 't') usually refers to whatever object encloses the code that is currently running. Be warned, there are two major cases where `this` may not point where we expect, because Yazoo is very literal about the enclosing-object rule. The first problem case arises when we try to use `this` inside any other curly braces, as in

```
a :: { b :: { f :: {... } } }
f_search_path :: { b, a, this }
```

The problem here is that code within braces always belongs to the definition of some other object, and inside those braces `this` refers to that object, not the current function. In this case `this` refers to the set `f_search_path`, not the enclosing function.

Caveat no. 2 is that `this` will not work inside of the arguments to another function call. The reason is a bit more complicated and will be explained later, but it precludes things like

```
print(top(this))    | won't work
```

We can get around the problem by printing from an alias, which obeys all the normal rules.

```
self := @this     | workspace or function space
print(top(self))
```

Within a function, `this` refers to the function itself. At the outermost level or at the command-line, `this` refers to the 'workspace' variable that encompasses all of the user's activities. (Technically, when Yazoo is run interactively at the command line, start.zoo runs all of the user's commands inside of an illusory workspace that is really one of its own variables.) The cosmos of Yazoo is arranged with the true workspace living side-by-side with a register variable (whose members include `R_slong`, `R_composite`, etc.) all within the universal variable called Zero. The members of Zero have no names, so there is no way to access anything outside the workspace except by using the built-in register commands.

A second miscellaneous name floating around Yazoo is '`that`'. `that` only exists to the right of an equate statement, where it refers to whatever was on the left. It can be used to abbreviate cumbersome expressions such as

```
facts.num.N = facts.num.N * log(facts.num.N) + facts.num.N
```

with

```
facts.num.N = that * log(that) + that
```

Outside of equations, or to the left of the equals sign, `that` is void.

Both `this` and `that` seem to be funny sorts of members, since their targets depend on the contexts in which they are used. Of course they are not really members at all. They are basic operators, just like '`+`' and '`=`' except that they happen not to take any arguments. In total, four operators masquerading as members inhabit Yazoo: `this`, `that`; `args` whom we will meet presently; and the void.

3.5 Arrays

If we were to try to program in Yazoo using only the methods described in the last section, one of the things that we would still find very difficult would be working with large amounts of memory. Simply setting aside, say, a megabyte of integer storage would require the order of a herculean megabyte of Yazoo code. The reason is that each variable, and each member of a composite variable, is considered to be a unique object, so the user has to give each one an individual name to distinguish it. But for cases when individuality is unimportant, we can make do with a blander structure called an array: a homogeneous set of identical elements that are distinguished only by their positions in the structure.

The specific element of an array is specified by one or more indices. An array whose elements are referenced by a single index is basically a list; an array with two indices (a two-dimensional array) can be thought of as a table; etc. One conjures up a particular element of an array by writing each of its indices in square brackets `[...]` immediately (no space) after the name of the array. To create the array in the first place, one just defines the maximum index for each dimension of the array using the normal define operator. Some basic manipulations on arrays are demonstrated below.

```
weeks := 50, years := 5
HoursWorked[weeks * years][5] :: ulong         | defines a 250 x 5 array
HoursWorked[1][1] = HoursWorked[1][2] = 8      | writes some elements
HoursWorked[1][2, 4] = HoursWorked[1][1, 3]    | copies a block of elements
HoursWorked[2] = HoursWorked[1]                | copies a whole row
```

There are a few tricks here: the fourth line shows how one can, sometimes, work with whole groups of indices (the conditions are given below), and the fifth line demonstrates that a partially-indexed array can

be thought of as an array in its own right of lower dimension. This example shows that a whole array or chunk of an array can be manipulated as if it were a single variable (which is not too surprising since the same is true for composite variables).

Arrays, like variables, can also be defined with more complex types.

```
TimeCard[52][5] :: { Hour[2] :: ulong, Minute[2] :: ulong }   | time in, out
```

Then we access the array elements by alternating named and indexed members.

```
TimeCard[10][3].Hour[2] = 17     |  leave at 5 PM
```

One unfortunate qualifier is that there are difficulties when the type-definitions of an array's elements has non-define operations in it. For example,

```
TimeCard[52][5] :: { Hour :: ulong, Hour  = 2 }
```

executes similarly to

```
TimeCard[52][5] :: { }
TimeCard[*][*].Hour :: ulong    | legal
TimeCard[*][*].Hour = 2         | illegal
```

where `[*]` represents all indices of a given dimension. This doesn't work because equate cannot (at present) copy data from a constant over to multiple array indices. Some of these restrictions may be loosened in a future version of Yazoo.

The user can access multiple indices at once by specifying the first and last elements of that range; e.g. `my_array[4, 7]` refers to indices 4, 5, 6 and 7. However, there are restrictions when working with multi-dimensional arrays, having to do with the way Yazoo stores these arrays in its (one-dimensional) memory. Internally, a multi-dimensional array is stacked so that the *last* index increments over consecutive elements in memory; adjacent blocks are determined by the next-to-last index; etc. So for the array `a[2][3] :: ulong`, the order of elements is `[1][1]`, `[1][2]`, `[1][3]`, `[2][1]`, `[2][2]`, `[2][3]`. The rule is that any block of an array the user accesses must be *contiguous in memory*. Some examples of statements that obey or violate this rule are given below.

```
grid[5][10] :: { score :: double, units[4] :: ulong }

print(
    grid[2, 3]                    | legal
    grid[4, 5][1, 10]             | legal
    grid[1, 5][4]                 | will cause error
    grid[1, 5][1, 10].units       | legal
    grid[1, 5][1, 10].units[1]    | will cause error
)
```

In this last example every case of multiple array indices was a token. We could have made the token explicit by an alias of the form:

```
grid_tok[*] := @grid[2, 3]
```

We can also copy data directly over ranges of indices. However, as mentioned above, we cannot copy inlined constants to multiple array indices. Thus

```
grid[2, 3].score = 0
```

is illegal; we have to set the score values one by one. The same rule usually prevents the define-equate operator from working in array definitions, as in:

```
array[10] :: { score := 0 }    | the equate part of := won't work
```

Thorny issues arise in abundance when one tries to run arrays of functions:

```
f[10] :: {
    counter :: ulong
    num :: double

    code

    for counter in [1, 5]
        print(num)
    endf
}

f[*]()        | here we have an error
```

Here the error happens in the `for` statement: Yazoo can't increment `counter` because `counter` is an array. To get around this problem, try defining `counter` outside of `f()`. If we do this then `f[*]()` will work. However, attempting `f[3]()` still won't work, for the (admittedly technical) reason that the argument inside the `print()` function cannot create a token for `num` in just one element of the `f[]` array, in the same way that one cannot form the alias `f[3].al := @num`.

(It must be noted that the `delete` command, which will be described presently, is slightly more relaxed about this rule. Since `delete` keeps track of the last index of its argument separately from the rest, it only requires that the *preceding* steps are contiguous in N-1-dimensional array space. So, referring again to this past example, the third argument to `print()` would have been legal had it instead appeared after a `delete` operator.)

One handy shortcut for accessing all the elements of a given array dimension is the 'wildcard' `[*]` operator. In the example above, the fourth line in the arguments of `print()` could have been written

```
grid[*][*].units
```

An array defined with `[*]` defaults to size 0. Be aware that this wildcard operator can automatically rescale the array when used to the left of an equate or forced-equate statement, as explained in the next section.

One caveat to bear in mind when passing subsets of arrays is that, depending on the notation used, the sub-array may get collapsed along one or more dimensions. That is because, as Yazoo steps across two or more consecutive array dimensions, it only counts the total number of elements being referenced while losing track of how they factor along the dimensions, and at the end of the day it just assumes they form a one-dimensional list. For example, pretend we have the following function arguments:

```
grid
grid[*]
grid[*][*]
```

The first line returns a two-dimensional array. The second line returns the one-dimensional list of all 'rows' of the table, each of which has a further set of indices over the columns, so it is also effectively a table. The third line returns the entire array rearranged into a one-dimensional list. If we are at the command prompt

we can `mprint()` each of these three lines, and that will confirm that the first two expressions print out as proper tables whereas the third writes out as a long list.

3.5.1 Resizing

All arrays in Yazoo that are not 'jammed' may be resized, and there are a number of ways to accomplish this. The most straightforward method is to use the modified index operator `[^...]` which sets the maximum index of a single dimension of the array. This can be used to either increase or reduce the size of the array, and several of these operators can act simultaneously on their respective indices in a given expression. The resize operator is similar to `[*]` in that it returns the full range of that index of the array; however, unlike `[*]` it does not return anything unless it is embedded in a larger expression (so in the case of `print(a[^5])` no token is created for `a[]` and so nothing is printed).

Indices may also be inserted into the middle of an array, using the `[+...]` and `+[...]` operators. (There is a subtle difference between the two which is only important in the rare case of an array with multiple members–a situation that is considered at the end of this section.) New indices are inserted into the array at the position specified in brackets before subsequent operations are performed. These operations return the *new* chunk of the array only, although as in the case of `[^...]` no token is created if an insertion operator begins a sentence.

The `delete` command nixes array indices in the same way that `[+...]` creates them. The syntax is slightly different: `delete my_array[2, 3]` is an example. Be careful not to confuse `delete` with `remove`! The former deletes the actual storage of variables whereas the latter removes *members*, which does not a-priori affect the content of variables (though Yazoo will eventually clean out variables that have no members to access them by). The `delete` command cannot be used within a larger expression, and it does not return a value under any circumstances, since after it has done its work there is hopefully nothing left to send back.

Let us suppose that we have an array which we have defined as '`a[2][3]` :: ...'. We can alter it in the following ways to illustrate use of the resize operators.

```
> sprint(a)

{ { 11, 12, 13 }, { 21, 22, 23 } }

> a[+2], sprint(a)

{ { 11, 12, 13 }, { 0, 0, 0 }, { 21, 22, 23 } }

> delete a[*][2], sprint(a)

{ { 11, 13 }, { 0, 0 }, { 21, 23 } }

> a[*][^5], sprint(a)

{ { 11, 13, 0, 0, 0 }, { 0, 0, 0, 0, 0 }, { 21, 23, 0, 0, 0 } }
```

Notice that newly-allocated memory is always initialized to zero.

One final way to resize an array is to do so implicitly through the wildcard `[*]` operator. If an equate or forced-equate statement is prevented from copying data because of mismatched array sizes, and if there is a wildcard index in the destination variable (i.e. to the *left* of the equate; Yazoo will never alter the source variable to the right), then Yazoo will see if it can resize the array to allow the data to be copied. Since the specifying of an index range unrolls the table into a list, Yazoo does not care whether the indices of the source and destination are individually matched: `a[1, 3][*] = b[1, 6]` is perfectly legal. For equate, the total number of indices must be equal, and in the previous example the second index will resize to 2. A forced-equate will attempt to make the total size in bytes of the two sides equal, if necessary.

3.5.2 Aliasing and jamming

Just as a member can stand in for another by means of an alias, so can it do for array elements; and likewise elements of one array can impersonate elements of another and even simple members, with some provisos. All moves in this masquerade ball are made by way of the old equate-at operator '`=@`' and its relatives. Simple members can always be aliased to arrays, no problem:

```
array_1[5] :: array_2[10] :: al :: double
al = @array_1[4]
```

We can also alias arrays to other arrays, provided we retarget all of the elements at once to some contiguous block of memory. So after the last two lines we can write

```
array_1[1, 5] = @array_2[4, 8]
```

but *not*

```
array_1[3, 5] = @array_2[4, 6]
```

except in the obscure case where indices 3-5 span one complete member (see the discussion in the next section).

The caveats on array-aliasing follow from the rule that elements of an array have to span contiguous memory. There is a way to get around this rule, namely by using proxies. A proxy array is defined by using a variant of the normal define operator: `#::`.

```
array_3[5] #:: double
```

For most practical purposes `array_1` and `array_3` are equivalent; however, the elements of `array_3` can each point to completely different places in memory (they are all initially void). We can now re-alias the members of the array one-by-one, though not all at once as with a normal array. After

```
array_3[4] = @al
```

we end up with an array consisting of `{*, *, *, 0, *}`.

Proxy arrays defined with the void type are most useful, as they can store any collection of aliases. But we do have to be careful about creating new elements of void proxy arrays, because we don't want to redefine the member type. Consider the following code:

```
u :: ulong

proxy_array[2] #:: *
proxy_array[1] = @u
proxy_array[2] :: slong      | error -- can't redefine the type of element 2
```

The problem here is that the last line tried to both create a variable and specialize the member type of the second element of the proxy array. As explained in the next section, all elements of `proxy_array` are contained within a second member, and because all elements of a member must be specialized at once this operation is illegal. To properly construct the `slong` variable without trying to specialize the array we should have written:

```
proxy_array[2] @:: slong
```

carefully using the variable-define operator.

Unlike normal array elements, elements of proxy arrays are always void when they are formed. Thus

```
my_array[2] #:: double
```

initially contains two void members even though it was assigned the primitive `double` type, and

```
my_array[+3]
```

will create a void third member regardless of whether elements 1 and 2 are still void.

An obnoxious and, unfortunately, unavoidable characteristic of Yazoo's arrays is that they can become 'jammed', meaning that they obstinately refuse to be resized. This can only happen if there exists some alias of the array floating around when the resize is attempted. Whenever part of an array is aliased, certain elements of that array can be accessed by two members, so adding or removing elements by resizing either one of its members would also force a resize of the other member. There are cases in which this can't be done without going into fractional indices — but mathematical limitations aside, resizing an unrelated member would violate the founding principle that no line of code may disrupt other regions of memory that it did not not explicitly name. So for the common good, Yazoo disallows any resizes that would disrupt other members, and it does this by jamming array indices that are double-referenced.

Here is an example of a partly-jammed array, and various legal and illegal attempts at resizing it.

```
array_1[3][10] :: ulong
array_2[4] := @array_1[2][4, 7]   | jams 4 indices of array_1

array_1[*][^12]        | legal
array_2[^12]           | no, this would cause problems
delete array_1[*][5]   | not legal
delete array_1[1]      | legal

remove array_1
array_2[+3]       | legal only because we removed the jamb
```

Explicit aliases always jam arrays. On the other hand, tokens—unnamed references to objects that are found in sets and function arguments—can never jam arrays. In other words:

```
al[1, 3] := @my_array[4, 6]    | jams these elements
my_array[4, 6]                 | does not jam
```

Tokens are 'unjammable', both in the sense that they cannot jam, and that they will become 'unjammed'—i.e. permanently deactivated—if their referent is resized through another member. Unjammability is on balance a good thing, since it keeps arrays that were passed as function arguments from getting gummed up. The token basically deactivates when the array is resized, and will only be renewed when the function is called again from that same point in the script.

But unjamming can also cause problems, most commonly in the case of a user-defined set.

```
a[10] :: ulong, b :: c :: ubyte
objects :: { a, b, c, a[9] }

a[^8]
```

Here the fourth element of the `objects` set will become unjammed when '`a`' gets resized, and we will get an error if we then try to use it. Had we instead defined the fourth member explicitly with `member\_4 := @a[9]`,

then the error would have occurred earlier, when we tried to resize the array, in order to protect the alias. Notably, the first element of `objects` will remain intact in either case since it points to '`a`' which is a composite variable, not to any of `a`'s indices—a point which we'll elaborate on in the next section.

3.5.3 Members as indices

It turns out that *all* members, even ordinary members of composite variables that have names, can be accessed with the array-index operators. For example, if we have

```
CountryInfo :: {
    name := "Angola"
    stats := @Angola_stats        | defined elsewhere
    rep := @Ambassadors.Angola    }
```

then `CountryInfo[1]` is "`Angola`", `CountryInfo[2]` is an alias for the `Angola_stats` variable, and `CountryInfo[3]` is an alias to the name of the Angolan ambassador.

It is possible, and probably inadvisable, to come up with confusing mixed-breeds between composite variables and arrays:

```
Compound :: {
    member1 := "1"
    this[2, 4] :: ulong     | indices 2 - 4
}
Compound.five := 5
Compound[8] :: double    | indices 6-8
```

The final line, it must be emphasized, does *not* define a single index, nor does it redefine all indices 1-8; instead since the previous highest index was 5 it defines a new width-3 member that brings the *total* number of indices to 8. As a general rule, an index operator in a define statement defines a new member if the index is above the current top index, but if the given index currently exists it tries to redefine only that index.

This past example illustrates an important distinction in Yazoo: the difference between a member and an index. The `Compound` variable above has 4 members spanning 8 indices. The first and third members have names and each has only one associated index; the second and fourth members are unnamed and have three indices apiece.

Let's extend our earlier rule on allowed and disallowed ranges of indices in the logical way: all indices in a range of elements must be part of the same member. So for example `Compound[3, 4]` is allowed but `Compound[4, 5]` is not. Remember that Yazoo only manipulates contiguous blocks of memory, and if two indices are not even part of the same member this requirement is thoroughly violated.

The small difference between the `[+...]` and `+[...]` operators is illustrated by the following example:

```
TwoArrays :: {
    this[5] :: ulong
    this[10] :: ulong        }

TwoArrays[+6]      | adds an index to the beginning of the second member
TwoArrays+[6]      | adds an index to the end of the first member
```

There is no operator that inserts a member in between two existing ones.

Let's take a step back and just take a last look at how the basic one-dimensional array 'works':

```
OneD[10] :: ulong
```

`OneD` itself is a member that points to a composite variable. That composite variable itself has in turn a single, unnamed member with ten indices, spanning an array of unsigned longs. So when the user accesses `OneD[3]`, he navigates first to the composite variable, then to index 3 of the sole member within that variable. Note that the composite variable was defined implicitly without any code (there is no '{}' anywhere). As the section on type-casting explains, code and type are two sides of the same coin for composite variables; as a result `OneD` (both the member and the variable) are very general objects, and can be specialized arbitrarily, or aliased later on to any composite variable or function.

3.6 Loops and `if` blocks

When Yazoo executes a script, a little bug called the Program Counter crawls along the script line-by-line and causes each command it encounters to do its thing. This section talks about various ways of making the PC hop from one place to the next, without using functions (which are described in a later section). This is a slight digression from the logical order of topics, but it's an important one since flow control is such a basic subject.

In Yazoo scripting there are four commands for controlling the PC: `if` statements, `for` loops, and `while` and the related `do` loops. All of these are of course familiar from C and other programming languages. Technically, none of these 'commands' are really fundamental operations, since the compiler expands them into longer expressions involving 'gotos'. For example, an "`if` *cond*" statement translates into "`if (not` *cond*`) goto` *endif*". There is no way to script a goto directly.

3.6.1 `if`

An `if` clause in Yazoo is written in the following style:

```
if villain.name == "Bob"
    print("Bob did it.\n")
end if        | or endif
```

An `if` executes the code between itself and its respective `end if` only if the conditional expression after the `if` evaluates to true. Notice that the `end if` can also be written as one word, `endif`.

In Yazoo, a conditional expression *always* involves one of the comparison operators '`==`' (if equal), '`/=`' (if not equal), '`==@`' (if aliased at), '`/=@`' (if not aliased at), '`<`', '`>`', '`<=`', or '`>=`'. In contrast to C, statements like `if true` or `if a = b` are not allowed (single '`=`' is an equate). The last four operators that we wrote take only numeric arguments. On the other hand, '`==`' and '`/=`' can compare the contents of any pair of variables or constants which have an identical type. Generally speaking, if `a = b` is legal, then `if a == b...` is also valid. So if we have

```
a :: { i :: j :: ulong }
b :: { m :: sshort, n :: double }
c[2] :: ulong
```

then `if a == b` is valid because different numeric types can still be compared to each other, but `if a == c` causes an error since '`a`' has two width-1 members while `c` contains a single member spanning two indices.

We can build up compound conditionals by conjoining these simple expressions with '`and`', '`or`', '`xor`' and augmenting them with '`not`'. The first three take both left- and right-hand conditional arguments (both of which are always evaluated), and return true only if either (`or`), both (`and`), or only one but not both (`xor`) expressions come out true. `not` only takes a right-hand conditional argument; if that argument is true it returns a false and vice versa. As an example we give:

```
if (not a /= b) and ((a < b or a > b) xor a == b)    | assume 'a' and 'b' numeric
```

which is really just a high-handed way of writing "`if a == b`".

3.6.2 `while` and `do until`

A close cousin to the `if` block is the `while` loop. The difference is that the code between `if` and `end if` runs only once if the condition is true, whereas the code between a `while` and an `end while` runs repeatedly until the condition is false, at which point execution picks up after the end of the loop. Consider the following:

```
counter := init_val
while counter <= 5
    print(counter, " ")
    counter = that+1
end while        | or endw

print("-- done!\n")
```

If `init_val` is 1, then our program will output "`1 2 3 4 5 -- done!`". Suppose `init_val` is 6. Then our condition is never satisfied and we just get "`-- done!`".

A second, very similar loop, is the `do-until` loop. This executes the code between the `do` and the `until` as long as the condition is *not* satisfied. Transcribing our above example into `do`-loop form, we have

```
counter = init_val
do
    print(counter, " ")
    counter = that+1
until counter > 5

print("-- done!\n")
```

One difference between `do` and `while` is that to achieve the same effect we have to use the opposite condition in a `do` loop from that of a `while` loop. A more substantive difference is that, because the condition in a `do` loop is evaluated only at the end of the loop, we are guaranteed of getting at least one run before getting booted out. So if `init_val` is 6, we get the output "`6 -- done!`". Whether or not we always want this initial run usually determines the choice of `while` or `do`.

For convenience, the end-of-loop marker `end while` can be abbreviated `endw`. There is no shorthand for a `do` loop.

3.6.3 `for`

A `for` loop is used to run code repeatedly for some defined number of iterations. We can adapt our last example to use a `for` loop instead.

```
for counter in [init_val, 5]
    print(counter, " ")
end for        | or endf

print("-- done!\n")
```

Notice how this saves us two steps: `counter` is automatically initialized at the beginning of the loop, and incremented by 1 (by default) after each iteration. (It is assumed that we already defined `counter` since the `for` loop will not do this for us). This example corresponds more closely to the one we gave for the `while` loop than it does to the `do` loop example, since if `init_val` is greater than 5 the loop will not run even once. Note that, for convenience, we can abbreviate '`end for`' as '`endf`'.

By default `counter` increases by 1 after each pass of the loop. The `for` loop comes with the option of changing this, by using the `step` keyword. For example, if we were to change the `for` line above to

```
for counter in [1, 5] step 2
```

then we would get `1 3 5 -- done!` as output. The `step` parameter is quite flexible; for example, we could change that same line to

```
for counter in [1, 10] step counter
```

to get the output `1 2 4 8 -- done!`. We can also have a fractional step, and a negative step is also legal – but regarding that second possibility what we *cannot* do is use a negative step variable. Thus

```
for counter in [10, 1] step -1
    print(counter, " ")
end for
```

works just fine, printing 10 through 1 in reverse order, but

```
loop_step := -1
for counter in [10, 1] step loop_step
    print(counter, " ")
end for
```

will not print anything. The reason is Yazoo decides whether to terminate the loop above or below the final loop index based on the sign of the counter variable, and if that's not a constant then it will assume a positive step. Here that assumption terminates the loop immediately because when `counter` is first initialized to 10 it is already above the cutoff threshold of 1, which is construed as an upper bound.

It may look suspicious that the same square brackets `[]` are used in both `for` loops and arrays to denote ranges. There is in truth no reason for this at all except that the keyboard manufacturers only put four pairs of bracket-like marks on their boards, and that the other three are already overworked as it is. The brackets, the comma, `in` and any `step` keyword along with `for` and `endf` are not even operators; they are simply signposts marking off the different parts of the loop. When the compiler comes across a `for` statement it simply lifts the expressions between these posts, compiles them and plops them into a precooked `for`-loop code involving gotos. The `if`, `while` and `do` blocks are all handled in the same way.

3.6.4 `break`

There is no `break` statement in Yazoo. We can however jerry-rig something that does pretty much the same. The trick is to put braces – with no `code` marker or semicolon – around the code we eventually want to escape from. Within those braces, a `return` statement will simply escape from the bracketed code and continue with what follows. For example:

```
{
    for counter in [1, 10]
        print(counter)
        if input() == 'q', return, endif
    endf
}
```

The disadvantage is that if we are running a function, we cannot do a direct return from the function within the bracketed code: we can *only* break. On the upside, we have a lot of latitude in choosing where to put the braces. We are not restricted to escaping from loops, and if we do use them inside loops it does not have to be to the next-outermost `for` or `while`.

3.7 Sets

One of the more robotic aspects of programming languages is the top-down, hierarchical and bureaucratic way in which they like to organize their memories. Classes contain variables containing fields pointing to more variables, etc.: every item has a sort of postal address which needs to be spelt out in full whenever the user wants to communicate with it. Contrast this to the web-like memory of our brains, where information about some given thing can usually be accessed via any number of lateral associations between it and other memories. Human brains have a lot to teach us: the relational web is more versatile and often faster to navigate than the hierarchical tree.

The set in Yazoo exists partly to give the language a more natural memory structure. A set is a bag of objects that were defined elsewhere. Those objects can be variables, functions, classes, even data types and other sets. Defining a set doesn't affect those objects at all—they will still be in the same place as before, except that they can also be found, magically, by reaching into the bag. The syntax for defining a set basically the same as that for defining a composite type: we give a name, a '`::`' and curly braces enclosing the objects inside, as shown below. The difference is that those objects are only listed between commas (or end-of-lines); no define operator needs appear within the braces.

```
Alice :: Bob :: Christine :: Daniel :: Elan :: friend_of_mine

men :: { Daniel, Elan, Bob }
neighbors :: { Christine, Bob }
cleaning_schedule :: { Bob, Alice, Bob, Alice, Elan, Elan, Daniel }
```

After we have run the above code, `Bob`, `men[3]`, `neighbors[2]`, `cleaning_schedule[1]` and `cleaning_schedule[3]` are all the same thing. Notice how the same object can appear in several places in a set. In this case `Bob` is the shortest name, but sets really come into their own when the alternative path name is long and inconvenient.

```
to_buy :: {
    food.produce.fruits.apples
    clothes.shoes.black
    clothes.socks.black
}
```

Here `to_buy[1]` is definitely quicker than `food.produce.fruits.apples`.

Sets in Yazoo are pretty flexible. Along with variables, we can throw constants, other sets (including ones that we define on the fly), and even the void into the bag.

```
collections :: { men, neighbors, { Patty, Don }, "Herbert", 3.3, this[3], this, nothing }
```

A few of these tricks require some explanation. The third item in `collections` is some nameless inlined set which can be thought of as a subset of `collections`. The fourth and fifth items are inlined constants. The sixth item *is* also the third item, that unnamed subset, and it obviously had to be listed after the third item because otherwise `this[3]` would not have existed yet. The seventh member of `collections` is the whole of `collections` itself.

To access the objects within any of these sets, we must use the square-bracket indexing operators (described in the section on arrays). The reason is that the set's members have no names, so if after our first example we were to request `collection.men` then Yazoo would draw a blank. As subsequent examples showed, it is not even clear how Yazoo could sensibly assign a unique name to each member. We actually can give names to the elements of a set ourselves, by manually aliasing the objects (for variables), or using define commands (for inlined composite objects) or define-equates (for inlined constants). Below is an alternative definition of `collections` which assigns names to members 2, 3, 4, 7 and 8.

```
collections :: {
```

```
    men
    neighbors := @neighbors      | use same name for convenience
    parents :: { Patty, Don }
    Herby := "Herbert"
    3.3,  this[3]
    self := @this
    zilch := @nothing
}
```

With this more verbose second definition we can write `collections.Herby` in place of the more abstract `collections[4]` (although `collections[4]` is still perfectly legal).

Sets are not immutable. New items can be added to them and existing members can be removed, in the same way we can add or remove variables or members of composite types and variables. The following code rearranges the members of the last example so that "`Herbert`" goes at the end.

```
collections[9] := @collections.Herby
remove collections.Herby
```

One last trick is also reminiscent of composite variables: we can use one set to define another. The following is legal:

```
men :: { Daniel, Elan, Bob }
males :: men
```

but, it can be simplified even further:

```
males :: men :: { Daniel, Elan, Bob }
```

3.8 Functions

Somewhere in the toolkit of just about any modern programming language is a handy mechanism that lets one program by the method of perturbation. The programmer writes some baseline code, *once*, and then adapts it to the local situations at various places in the program. The baseline code is called a 'function'. The perturbations are usually concentrated in a set of parameters called 'arguments' to that function. Programming languages are so improved by having functions that they almost universally have this feature, so we expect the user to be quite familiar with their use.

Yazoo also has functions. We'll show how they are defined momentarily (the syntax is similar to that for defining variables). Here is how you call one:

```
y = f(x)
```

It all looks familiar. But don't be fooled: what we just glimpsed is probably the strangest beast in the Yazoo jungle. Functions in Yazoo live in ordinary 'heap' memory, and are more akin to composite variables — indeed, even to their own arguments! — than they are to their stack-dwelling counterparts in, say, C.

3.8.1 Defining functions (properly)

In Yazoo, functions are *objects*. This is both a source of strength and one of Yazoo's greatest weaknesses. On the one hand, when something goes wrong we can go into a function as if it were a variable, peek at all its internal members, and fiddle around with it for the next run. The author, who has a weak sense of propriety, considers this a plus. The big drawback is that it is possible for the return value of a function

to be overwritten before it gets used. There are several ways to avoid this, but at the cost of some manual labor – about one line or so per function call.

Let's demonstrate some of these unusual properties of Yazoo functions by actually writing one. As we said, functions are objects, so, unsurprisingly, we define them using the same operators we have been using for all other objects. There are two new ingredients: an `args` variable, containing the function arguments; and the '`code`' marker (semicolon for shorthand), which marks the end the function's variable definitions and the beginning of its executable code. The function exits with the classic `return` statement, which may or may not be followed by some object or value to pass back to the caller. It is usually easiest to define functions from scripts, not from the command line, since that allows us to avoid trying to fit the whole definition onto one line.

```
SwapDigits :: {
    lh_dig :: rh_dig :: ubyte

    code

    lh_dig = round_down(args[1] / 10)
    rh_dig = args[1] mod 10

    return rh_dig*10 + lh_dig
}
```

Now that we have written the function, we can run it in the normal way.

```
print( SwapDigits(27) )
```

We get 72. So far so good. If for some reason we were dubious as to whether the function had worked properly, we might reassure ourselves by checking its two members as if the function was a composite variable.

```
print( SwapDigits.lh_dig, " ", SwapDigits.rh_dig, "\n" )
```

No surprises: the result is "`2 7`".

It may be reassuring to know that the two variables in the constructor (the part before the `code` marker) *will* definitely be defined inside the function. One might suspect that if there had already existed, say, a global member named `lh_dig`, then `SwapDigits()` would simply have tried to redefine that member. No – the define operators always create members inside the immediate function, so we don't have to worry about accidentally reusing common member names like counter variables. (Mind that this only applies to the *left-hand* arguments of the *define* operators; on the right-hand side Yazoo searches all the way back to the workspace. And other operators such as equate don't restrict either argument: `lh_dig = 5` does its job whether or not `lh_dig` is in `SwapDigits()` or not.)

As advertised, we can modify the members of a function rather arbitrarily after the function has been created. For example, we could remove one of its members, though `SwapDigits` would obviously stop working until we redefined that member. We can't change the function code itself, at least not with the tools we have so far, but we can introduce auxiliary codes within the function. So we can, for example, define a sub-function that prints the members of `SwapDigits`, as we did above.

```
SwapDigits.printout :: {
    code
    print(SwapDigits.lh_dig, " ", SwapDigits.rh_dig, "\n")
}
```

Two features of `printout()` are noteworthy in their own right. First, there is no `return` statement. This is fine; when Yazoo hits the end of a function it does as if it had encountered a `return` with no argument.

(Technically, in both cases it returns `nothing`, literally, as in: it returns the void.) The second point is that since `printout()` was defined from a path that did not pass through `SwapDigits()`, it does not have automatic access to `SwapDigits`'s variables and thus has to name `SwapDigits.lh_dig` and `SwapDigits.rh_dig` explicitly.

Because they are objects, functions can be used as templates for defining other functions. For example, we could write:

```
SD2 :: SwapDigits
```

Now `SD2` will contain the two variables `lh_dig` and `rh_dig`, and it will have all the code that followed the `code` marker in `SwapDigits`'s original definition. In hindsight we really should have defined `printout()` in the original definition of `SwapDigits()`. The reason is that `printout()` will *not* be present in `SD2`, since we added `printout()` separately and therefore it is not part of the definition of `SwapDigits`. For this reason (and only this reason) a command like

```
SD2 = SwapDigits
```

will give a type-mismatch error. Perhaps a slight misnomer, since their types are actually the same (those depend only on the original definition) – but their data structures do not match. Likewise, we could *not* have defined `SD2` using

```
SD2 := SwapDigits       | define-equate
```

(note the colon) since that would caused the same error for the same reason.

Usually, only one copy of a stand-alone function is ever needed. The exception is the case of recursive functions. Owing to the fact that a Yazoo function is an object, a new copy of that object needs to be defined if the function wishes to run itself in mid-execution. The memory requirement is then the same as for stack-based functions (N functions for a depth-N recursion). The basic procedure is to do something like the following:

```
factorial :: {
    fval :: ulong
    code

    if args[1] == 1
        fval = 1
    else
        new_fact :: this
        fval = args[1]*new_fact(args[1]-1)
    endif

    return fval
}
```

Importantly, the definition of `new_fact` could *not* have been put in the constructor: that would have caused Yazoo to try to construct an infinity of nested `factorial()` functions, and eventually throw in the towel with a recursion-depth error.

The same rules apply to indirect as to direct recursion. In other words, if `a()` calls `b()` calls `a()`, then function `b()` needs to make a new copy of `a()` in order to avoid overwriting the data of the outer call to `a()` that is still running, and `a()` needs to make a new `b()`.

Here's one unusual feature of Yazoo functions:

```
f :: {
```

```
    code
    return "a"
    code
    return "b"   }
```

Two coding blocks—so there should be a way to access them both. If we call `f()` we get back `"a"`, so the first block runs by default. The way to run the second coding block is to call `f#2()`, after which we get back `"b"`. We can infer that `f#1()` is simply longhand for `f()`, and `f#0()` runs the code *before* the first code marker (the constructor, when we talk about classes). We'll come back to this a bit later.

As we might suspect from earlier warnings, there was a rather serious defect in the functions we have been writing, which shows up when we try to do the following:

```
print( factorial(3), " ", factorial(5), "\n" )
```

What we get is

```
120 120
```

Well, we got back two copies of the second result – not at all what we wanted. The problem is due to an unfortunate coincidence of two facts: 1) unless instructed otherwise, functions return their arguments by reference, and 2) `print()` runs both arguments *before* either one is printed. Indeed, all functions evaluate all their arguments before they begin executing their own code. In our example above, by the time `print()` got around to printing, `factorial()` had already run twice, its return variable had been set to 6 and then reset to 120, and when all was said and done, both tokens in `print`'s argument pointed to this value.

We encounter the same problem when we define sets, since they also work with tokens.

```
swapped_dig :: {  factorial(3), " ", factorial(5)  }
```

Again, we end up with two tokens both pointing to the same return variable, which now stores the value 120.

One obvious way to get around the problem is to do away with the troublesome tokens, and straightaway copy the return value of each function call into a new variable (which we can do) before the function runs again.

```
print( arg1 := factorial(3), " ", arg2 := factorial(5), "\n" )
facts :: {  el1 := factorial(3),   el2 := factorial(5)  }
```

This is quite sloppy, and we're likely to forget to do it at some point. The better, fail-safe method is to have the function itself store each separate return value in a new variable, by unlinking and re-assigning the member with each function call. That way the first token will still cling to the old return variable after the new variable has been created for the second token.

```
factorial :: {
    rtrn :: ulong
    fval :: ulong

    code
    ...
    ((rtrn =@ *) :: ulong) = fval
    return rtrn
}
```

If Yazoo is being run interactively, there are a number of ways that we can use user.zoo's `new()` function to accomplish this with somewhat less trouble:

```
( rtrn = @new(ulong) ) = fval
return rtrn
```

or

```
rtrn = @new(fval)
return rtrn
```

or even just

```
return new(fval)
```

at the end. Reallocating the return variables is admittedly a nuisance, even with the `new()` function, but you take your chances otherwise.

3.8.2 Function arguments

It's hard to go any further without first confessing to a well-intended white lie that we made in the last section. It happened when we were explaining why `print()` only displayed the second return value of `SwapDigits()`. We wrote that `print()` evaluated both function calls before printing anything — all well and good. But, we were fibbing a little when we said that functions always evaluate *all* of their arguments before running their own codes. The true story is that an ordinary C-style argument list will, yes, evaluate entirely before the function begins to run; but if there is a `code` marker or semicolon present *inside the argument list*, any arguments that follow will not be evaluated until the function explicitly 'runs' them.

That last was probably a pretty mysterious statement; let's show what we mean with an example.

```
PrintArgs :: {
    code

    print("Before running args:  ", args)
    args()
    print("\nAfterwards:  ", args)
}
```

Then we call our function in the following way:

```
PrintArgs( 0.3, " 4", code, " word ", 10 )
```

The output is:

```
Before running args:  0.3 4
Afterwards:  0.3 4 word 10
```

Arguments before the `code` marker evaluate before the function executes, but arguments following the `code` marker evaluate only when the function calls `args()`.

The part of the argument list preceding the `code` marker looks a little bit like a constructor, because it dictates how the `args` set is first defined. But the rest of the arguments are almost analogous to the executable part of a function, because they are expressed by the peculiar `args()` call. There turns out to be

a closer analogy with functions than we might first suspect – because we can write in all sorts of executable commands besides just tokens, after (or before!) the argument-list's `code` marker.

```
> PrintArgs( 1, code, print("\nRunning args..."), " ", 2 )

Before running args:  1
Running args...
Afterwards: 1 2
```

In fact, just about the only code that may not work out quite as expected inside the arguments is a function or variable definition. The fact is that we can define variables and functions inside function arguments, and they will indeed appear in `args[]`, but they will not appear anywhere else in the program. The explanation for this anomaly comes out of the discussion of Chapter 4.

Let's try to push the args-function analogy to the breaking point. Who knows — if we can *run* the `args` list, maybe we can also pass it parameters? Suppose we change the `args()` line in `PrintArgs()` to:

```
PrintArgs :: {
    ...
    args(9)
    ...
}
```

We tweak our function call appropriately, and get the following output:

```
> PrintArgs( 1, code, print("\n'args' was passed ", args[1]), " ", args[1] )

Before running args:  1
'args' was passed 9
Afterwards:  1 9
```

It is important to understand that, in this last example, the '`args`' keyword referred to one of two different things depending on where it was used. Before the `code` marker of `PrintArgs()`'s arguments, `args` was whatever it had been earlier. Within the `PrintArgs()` function, '`args`' became that argument: containing `1, code, ..., 2`. *After* the `code` marker of `PrintArgs()`'s arguments, '`args`' was the parameter list that ran `PrintArgs`'s arguments; i.e. the one containing only the number 9. Each function call temporarily replaced the existing `args` variable with its own argument list; the old `args` came back when each function exited.

Arguments within arguments get confusing very quickly. We'll suffer one final example at level 3 to hammer in our last point about how `args` behaves to the left and right of a `code` marker. Suppose we defined the following two functions:

```
f :: {
    code

    g( print(args, " -- "), code, print(args) )
}

g :: {  code, args("B")  }
```

That is, `f()` runs `g()`, which in turn runs its own arguments. Now if we make the function call written below, we obtain the following output.

```
> f("A")
```

```
A -- B
```

Focus on the argument list in the function call to `g()` (in the line after `f()`'s `code` marker). The first half ran *before* `g()`'s code began, when `args` still had old value '`A`'. The second half of this argument list didn't run until `g()` called it with new arguments, at which point `args` was now '`B`'. What this example demonstrates is that `args` takes on a different meaning *after*, but not before, a `code` marker within a function call than it had in the code surrounding the function call.

Is there ever an earthly point to having functions running arguments that run their own arguments, etc.? Well, sometimes yes, and one good reason has to do with the 'permissions' that different functions have to access different variables. A piece of code has access to other variables and functions along its own search path. This applies to function argument lists as well as the functions themselves. Thus, whereas function `A` has access to its own members, plus the members of its enclosing class or function, etc., an argument list of `A(...)` has access to variables defined in whichever function `B()` made that call, along with the members of the class or function enclosing `B`, etc. By calling one's arguments with specific, on-demand requests, a function can exchange information with the calling function as the need arises.

As we show in the example below, one possible use of the `args` code is to set optional parameters that are otherwise assigned a default value. We will have a prettier way of accomplishing the same thing by the end of the next section.

```
lookup :: {
    word :: language :: string
    mode :: ubyte    | 1 = definition, 2 = thesaurus
    ...

    code

    word = args[1]    | get required params

    mode = 1    | set defaults
    language = "English"

    args(mode, language)

    ...
}
```

We can run this script with minimal arguments, if we are satisfied with the defaults:

```
lookup( "ulterior" )
```

or we can adjust one or several of the optional parameters after a `code` marker:

```
lookup_mode :: ubyte
in_str :: string
print("Type '1' for definition; '2' for synonyms: ")
in_str = input()
read_string(in_str, lookup_mode)

lookup( "munificent", code, args[1] = lookup_mode )
```

To make one final remark: we can replace any '`code`' marker flanked by commas or ends-of-lines with just a semicolon '`;`'. That latter shorthand was especially intended to be a less floral replacement for the

‘`, code, `’ sequence in function arguments. With that in mind, let’s dress up the `lookup()` call above using a semicolon instead:

```
lookup( "munificent"; args[1] = lookup_mode )
```

The moral of this section is that the coding block of a function’s argument list may sound abstruse but it is not. In particular, its ability to set optional parameters (elaborated on in the next section) has proven truly useful in the author’s hands. But the range of conversational possibilities between the function and the calling script is much broader, and greatly unexplored by the author.

3.8.3 Code substitution

Up until this point, running a function has involved some bit of code working hand-in-hand with the function’s own private variables, and perhaps also a few variables stored in the parent class or in the global space. What we will describe in this section is an operator that allows a function to invade some *other* object and do its dirty work there, reading and modifying that object’s members rather than its own. The operator that accomplishes this is called the code substitution operator, and it has the symbol ‘`<<`’.

First we’ll prove to ourselves that a substituted function truly has all the power of native code to read, write, create and destroy. Start off with a self-contained little function that writes its contents to the screen, and then quietly self-immolates by removing its only member.

```
goodbye :: {
    my_data := "Goodbye..."

    code

    myself := @this
    print(myself[1])
    remove myself[1]
}
```

One code substitution is enough to change `goodbye()` from suicidal to homicidal:

```
process_data :: {
    names[10] :: ...
    ...
}

(process_data << goodbye)()
```

As we can see, the code to run is the *right*-hand argument of the code-substitution operator, and the object that it will run inside is the *left* argument. Notice that, after substituting the code, we still had to put the usual parentheses to tell it to run. Nothing else changes: as in an ordinary function call only the three lines following the `code` statement of `goodbye` are executed inside `process_data`.

The substitution of code itself is quite temporary – the code stays substituted only until the end of that expression. On the other hand, whatever effect the substituted code has on its host when run is quite as permanent as anything native code can do. In our last example, the `process_data` function will be right back to its crusted old self next time we try to run it in the normal way, except with the major difference that it will be missing its `names` array.

Substituting *inlined* code can often save quite a bit of typing when some obscure place in memory needs prolonged use. For example, if we want to initialize a parameter list that has some awkward path, we could either write:

```
database.addresses.WriteAddress.params.include_country = true
database.addresses.WriteAddress.params.justification = "left"
...
```

or else, just:

```
(database.addresses.WriteAddress.params << {
    code

    include_country = true
    justification = "left"
    ...
})()
```

Very importantly, we had to write a `code` marker into the substituted code – when we run the substituted function *only* what follows the `code` marker will executed. The constructor of the substituted code (the part before the `code` marker) is not used at all for code substitution in the present version of Yazoo.

One thing that we will point out here, and delve into more thoroughly in the next section, is that a function substituted into some object has access to all of the global members that that function, *not* the object, normally sees. In this context, by 'global' we refer to any member outside of the immediate function; it could have been defined in the enclosing package, or all the way at the top. Code substitution only affects the first stop on the function's 'search path', which is now object being substituted into rather than the function's own storage space. The rest of the search path still runs through the variables enclosing the function, even though that function is not presently running in that region of memory. The upshot is that the substituted code in our last example could reference variables that we had defined just prior to the substitution by the same script, but it could not access, say, any of `WriteAddress`'s variables aside from `params`.

As advertised in the last section, code-substitution allows us to incorporate optional arguments into functions in an elegant way. To do this, we have to group all of the optional parameters into some composite variable. When the function runs we first read in the required parameters as usual, then initialize our optional parameters variable, and finally run the arguments, as a function, inside of that variable. That way, by writing commands after the `code` marker or semicolon of the function's arguments the default parameters can be changed in the same way that any other variable can be modified. Here is an example.

```
WriteAddress :: {
    params :: {
        include_country :: Boolean
        justification :: string    }

    name :: street_address :: string
    ...

    code

    name = args[1]
    ...

    params = { false, "left" }
    (params << args)()
    ...
}
```

A typical call to `WriteAddress()` might look like

```
jst := "center"
WriteAddress("Bob Jones", "65 Maple Ave", ... ; justification = jst)
```

Optional parameters are less obscure than the mandatory ones since their variables are named explicitly, and they can be named in any order and omitted when the defaults are suitable. Again, even though the `args` code was substituted into `params`, it retained its original search path, so that it could access, for example, the `jst` variable, though not the members of `WriteAddress()`.

One thing that will *not* work is to put the optional arguments into a set, unless the members of the set are explicitly named. For example, had we written

```
    include_country :: Boolean
    justification :: string

    params :: { include_country, justification }
    ...
}
```

the arguments to `WriteAddress()` would not access either of these parameters by name. The first element of the set points to `include_country`, but that `params[1]` has no name. The section on sets explains how to manually assign names to the set elements if we want to use this method.

Functions can have more than one coding block, and unsurprisingly function arguments can too. This can be useful if we want to run different function arguments in different spaces, or incorporate dependencies between different arguments. Here is an example that does both.

```
f :: {
    ...
    params_1 :: { save_log := false }
    params_2 :: { filename = "log.txt" }

    (params_1 << args)()
    if params_1.save_log == true
        (params_2 << args#2)()
    endif

    ...
}

f( ; save_log = true; filename = "tmp")
```

As we mentioned earlier, the so-called 'workspace' of Yazoo's interactive command prompt is not the real workspace where the start.zoo script runs. Instead, start.zoo creates this comfortable illusion inside of a fake workspace variable, furnished with the routines in user.zoo and within which the user's commands are run, all by substituting code into this variable. This gives is a third use for code-substitution: to create a little bubble-world where any number of functions can run and share data in the same playground. Actually, if we want true peace of mind that the substituted functions will not wreak havoc with the rest of the program, we sometimes also have to clip the functions' search paths, and we shall explain how to do that in the next section.

3.8.4 Search Paths and Fibrils

We can imagine Yazoo's memory as a graph in the mathematical sense: a swarm of dots representing the variables, connected by arrowed lines which represent the members. In this picture `a` $\longrightarrow$ `b` means that some member of `a` points to `b`. When we surf Yazoo's memory using the '`.`' and '`[]`' operators, we are exploring this graph in the 'forward' direction; i.e. along the direction of the arrows. We never explicitly navigate

backwards, since members are one-way channels; however, Yazoo does swim back upstream on its own when the user requests, say, a global variable from within a function. In doing so it follows a so-called 'search path'. The totality of search-paths forms a separate, parallel web of arrows atop our memory map.

Let's first look at how a search path works in general terms. Suppose that we look for

```
recipes[13].ingredients[3, 5].amounts
```

The `recipes` array may be in the active function, but it might also be in some enclosing package or in the global space. Yazoo checks the first possibility first: it looks for `recipes` in the active function. If it does not find any `recipes` here, then it goes up to the enclosing object and searches there. If `recipes` is not there either then Yazoo backs up one more layer; etc. on backwards until it either finds what it is looking for or reaches the end of the road.

After locating the starting point of the path (`recipes`) in reverse, Yazoo shifts into forward gear for the rest of the expression. Each successive step `[13]`, `ingredients`, `[3, 5]` and `amounts` must refer to a direct member of the previous step. (The exception is when the expression comes left of a define statement: if `recipes`, or any subsequent step, cannot be found, that and everything that follows will be created on the fly.)

This section will be concerned with an unresolved issue regarding the first, reverse step. The problem is that, due to aliasing, the current function might be targeted ('enclosed') by any number of members from different locations. So which one does it step back into, in reverse? Yazoo only takes one unique backwards path. The answer is that if Yazoo has to search backwards, then it backs up into *whichever object defined the function* and looks there – even if the member leading to the function was removed in the meantime. Then it tries the object that defined the object that defined the function, and so on.

As an example, suppose the following object was defined in some part of a script:

```
picture :: {
    bitmap[10][10] :: ubyte     | data
    vertices[4] :: ulong

    draw :: { ..., code, ... }     | methods
    crop :: { ..., code, ... }
}
picture.reset :: { ..., code, ... }   | externally defined method
picture.crop :: picture.crop : { ..., code, ... }     | specializes the old crop()
```

Now suppose that, at a different part of the script and from some very distant location in memory, the following alias has been made:

```
shortcut = @picture.draw
```

For all purposes but one, the two members `draw` and `shortcut` are *completely* equivalent. The only distinction between the two members, and the *only* justification for referring to that function by the name '`draw`', is that its search path will pass through the `picture` class where it was born, rather than through the object containing `shortcut`.

In general, the first step in a pathname can only find members that were defined immediately inside objects that, at some level, also defined that code. All other members have to be accessed by subsequent steps. In the example above, `draw()` is permitted access to all the members of the `picture` class because in some sense the definition of `draw()` is *part* of the definition of `picture`. But `draw()`'s members and `crop()`'s members lie on different and disjoint branches of code, so they must explicitly write `draw.whatever`, `crop.whatever` to access each others' members.

Finally, the `reset()` function and the second half of `crop()`'s code were *not* defined by `picture`, so their search paths skip directly from their respective variables to whatever object defined `picture`. Thus, whereas `draw()` has permission to call up `bitmap[3]` directly, its adopted sibling `reset()` must explicitly request

`picture.bitmap[3]`. The same of course holds true for foreign code that isn't even tied to a member of `picture`, such as the following.

```
(picture.draw << { code, print(bitmap)})()   | won't work
```

It's important to keep in mind that whenever an instance of the `picture` class is defined, instances of `draw()` and `crop()` will be defined and associated with the new variable, but `crop()` will only have the first half of its code and `reset()` will not even exist. The reason is that the code within `picture`'s braces, which can be thought of as `picture`'s constructor, makes no mention of these additional components.

Just as forward steps in memory are taken one member at a time, steps in a reverse search path go one 'fibril' at a time. A fibril is a little bit like a member in reverse: its stem (the thing it points to), is the next step in the search path, which is usually (but not always) the previous step from some forward path. Suppose we have

```
a.b :: { code, print(a) }
```

A member goes from '`a`' to '`b`', and a contrary fibril goes from '`b`' back to '`a`'. (The fibril is what allows '`b()`' to find and print '`a`'.) Each fibril is associated with the variable that is the given step on the search path.

Members and fibrils have significant differences beyond the directions of their arrows. Obviously, only members have names. Less trivially, whichever variable a member connects to may have many members of its own, so there can be many different forward paths from a given starting point. Each fibril, on the other hand, connects directly to another *fibril*, not that fibril's variable, so there is a unique fibril-to-fibril path backwards. One final difference is that, in the forward direction, we can go in loops if members target their own variables or parents (think `a.b.b.b` if `a :: { b := @this }`). In doing so we encounter the same member many times. But while we are going in circles, Yazoo keeps unrolling an ever-growing fibril trail that traces our circuitous path like a ball of yarn — and if we define a code at the end of all that, then that code will take the same redundant journey backwards in searching for variables (search '`a`' four times before making it to the enclosing variable). The same fibril is never encountered twice on a given path (which is a good thing, since otherwise Yazoo might go in circles forever); and the implication is that the fibril network forms a true tree, or set of trees – no loops.

Every piece of code in Yazoo has a fibril 'anchor', where the search for members begins while that code is running. The anchor is usually (except in the case of substitution) attached to that code's own variable, because a function looking for a member always looks inside its own belly first. If a function has two codes spliced together with the inheritance operator (à la our `crop()` function above), then each code will have a separate anchor and thus potentially different search paths. Whenever Yazoo needs to find a member, it simply grabs the anchor of the code that is currently running and follows backwards until it either finds what it is looking for, or else hits the end and throws an error.

Fibril trees are built up one branch at a time. A new branch (fibril) is added each time a pair of braces is encountered. The rule is that any new function or composite object obtains, firstly, a variable of its own; and secondly, an anchor in that variable whose stem is the anchor of the code that defined the new object.

Search paths may be invisible to the user, but that doesn't mean we can't operate on them. Yazoo comes equipped with two tools – the code-substitution operator '`<<`' and the built-in function `clip()` – which together can perform fairly arbitrary surgery on the fibrils. Code substitution temporarily grafts one fibril onto another fibril's stem. The `clip()` function permanently and irrevocably terminates a fibril path at a given point.

Code substitution is usually embedded within a function call, as in `(var << fn)()`. Earlier we described the object `var << fn` as a union of the code of `fn` with the variable `var`; now we can give a more sophisticated and formal explanation. The code substitution operator replaces the anchors of `fn`'s codes—temporarily—with new anchors inside of `var`. However, these new, temporary anchors in `var` inherit the original stems that `fn`'s normal anchors have. That means that the first stop on the search path is `var`, but the next stop is the *parent* of `f2`, followed by the grandparent, etc.

Code substitution is a temporary drug in the sense that code doesn't stay substituted outside of the immediate expression. Even other instances of `var` and `fn` in the same sentence won't be affected. One

permanent side effect, however, is that any functions that were defined during the substitution will be stuck with the crooked, 'temporary' spliced fibril path for the rest of their lives. So suppose that `fn` had defined some new function `daughter` when it ran inside of `var`. Then `daughter` would have a search path leading from itself back to `var`, and thence back to `fn`'s parent.

It is the balance between the need to secure data and Yazoo's desire to keep all its code 'open-source' that justifies the logic behind all this. On the one hand, any code is free to access any variable that it can draw a path to, so strict encapsulation of data is practically impossible (see next section). Requiring explicit paths to members of functions other than one's own is what is supposed to prevent accidental intrusions into one's neighbors. Everybody's door is unlocked, but you have to make a deliberate effort to burgle someone else's house. But, a function can access its parent's and grandparent's members without any effort, because that function is also under their roofs: its definition was *part* of its parent's and grandparent's codes. None of these rules change upon code substitution. A function can still draw on the resources of its biological parent when it is substituted. But not the parent of its temporary workplace – being hired to paint the child's room doesn't give anyone permission to whitewash the apartment.

Clipping a fibril is a little bit like disowning one's child. The `clip()` command simply unhooks a fibril from its stem, isolating any code whose search path used that fibril from everything beyond the clip point. For example, if we write

```
a :: double
b :: { ; print(a) }
```

then running `b()` works until we `clip(b)`, at which point `b()` can no longer see '`a`'. Of course it would still be able to see its own members, if it had any, since they would be above the clip point.

In contrast to the code-substitution operator, the effect of a `clip()` is both permanent and irrevocable. It might seem unbalanced that clipped functions can't reach their mother functions, while the mothers still have full access to their daughters, as `clip()` only affects fibrils, not members. But on this point the law is perfectly even-handed. Daughters can disown their mothers without leaving house by using the `remove` command, which deletes the member while leaving the corresponding fibril intact.

3.9 Classes and Inheritance

Most of the ingredients of a Yazoo script that we've encountered up to this point – variables, functions, data types – are standard fare, in one form or another, in bread-and-butter C. In this last section of the chapter we will demonstrate the measure of Yazoo's proficiency in the high cuisines of object-oriented programming. Yazoo was not explicitly intended to be object-oriented. But some of its capabilities look and smell quite a lot like features of the OOP languages. The rundown is: classes are straightforward; encapsulation is disappointing; polymorphism is obvious; and inheritance has some unusual generalizations to functions and the like.

Start with the bad news: encapsulation is one thing that Yazoo is really awful at. By use of the `clip()` tool we can blind a function to its environment, as we showed in the last section. But the environment can still see—and manipulate—the function and its contents. To hide the innards of a function we would need to `remove` its members, but by dismembering the function we would obviously ruin it. Any search path that leads *to* the function can go anywhere *within* the function; there is no way to selectively hide members only from certain paths.

A poor substitute for encapsulating code is to pseudo-compile a function into bytecode separately from the main script. Using tricks from the next chapter it would be possible to embed this bytecode underneath the skin of the main script. But that is not really encapsulation, since the function's members can still be accessed using the array-index operators '`[]`'. The best that can be said is that a separate compilation hides the function's source code and thus masks all the names of the members that were defined, so that the function cannot be explored from outside using the step-to-member operator '`.`', except by blundering about in a very awkward way.

3.9.1 Classes

Given that 'classes' in Yazoo are no more than glorified data types, the way we define them shouldn't come as any great surprise:

```
my_class :: {
    datum_1 :: ulong
    datum_2 :: string

    init_data :: {
        code
        { datum_1, datum_2 } = { 1, "blank" }
    }

    process_data :: { ...  }
}
```

Of course we haven't really introduced anything new here, except to include methods (functions) as valid members of a data type.

We don't yet have a constructor in quite the traditional sense of the word, but we can easily make one. As far as Yazoo is concerned a 'constructor' is just any code that does not follow a `code` marker, and its capabilities are completely general. So it is straightforward to adapt our existing class to automatically initialize its variables.

```
my_class :: {
    datum_1 :: ulong
    datum_2 :: string

    init_data :: { ...  }
    process_data :: { ...  }

    { datum_1, datum_2 } = { 1, "blank" }
}
```

Now each instance of `my_class` that is defined like so:

```
obj1 :: my_class
```

will begin initialized to `{ 1, "blank" }` . But Yazoo is flexible. We can write constructor code in anywhere we want, so a simpler alternative may be:

```
my_class :: {
    (datum_1 :: ulong) = 1
    datum_2 := "blank"

    process_data :: { ...  }
}
```

One apparent disadvantage of this constructor is that it seems not to be contained in any explicit function. Due to the oddities of Yazoo that is actually not true: we can re-initialize an instance of a class either by writing:

```
obj1#0()    | code #0 is the constructor
```

or, equivalently, by just redefining the object:

```
obj1 :: obj1    | reruns constructor
```

There is no destructor in Yazoo. If you think about it, there isn't even a direct way to delete an object such as an instance of a class, a variable or a function. What is possible is to irrevocably cut that thing off from the user: by removing all members that point to it, clipping any transiting fibrils, and if necessary returning the program counter from its code. Only after all these things have happened will Yazoo quietly delete the orphaned object so as to free its memory. The conditions sound complicated, but under the most basic circumstances (no aliases, etc.) simply removing an object's member will delete it. As it turns out, recognizing orphans is not always so straightforward, and as a result Yazoo is rather worse at automatically collecting the garbage than it maybe should be. Readers interested in the disposal of unused memory are referred to `SpringCleaning()` in the reference section.

3.9.2 Inheritance

The one operator that we have entirely ignored so far is the inheritance operator, which has the symbol ':'. In Yazoo, inheritance can be applied to most objects, including functions, but its most familiar use is probably in the context of deriving classes. The syntax is different from that in C++. The phrase '`a : b`' creates a type, derived from `a`, and refined by the additional code '`b`', as in:

```
parent_class :: {
    data_1 :: ulong
    func_1 :: { ... }
}

child :: parent_class : {
    func_2 :: {  ...  }
}
```

We defined `child` in the normal way, using the familiar define operator, but its type is a conjunction of the parent's type and some new component unique to the child. The members `data_1` and `func_1` are common to both `parent_class` and `child`, but `func_2` is specific to `child`.

Before going any further, we should make clear that inheritance applies *only* to composite objects. Well, 'composites' is a broad class; maybe a better statement is that the only entities that do not participate at all in any sort of inheritance are the primitive variables and data types (`double`, `string`, etc.). So both of the following are illegal.

```
x :: {  ...  } : ulong     | both completely illegal
y :: ubyte : string
```

In our earlier example, the left-hand argument of the inheritance operator was a predefined class, while the right-hand argument was inlined code within braces; this is the familiar case in which a pre-defined generic type `parent_class` is deepened by a type specific to the new object `child`. But one doesn't have to follow this pattern; in Yazoo, each type may be a predefined class or a new type in braces, independently of the other. Think of the ':' as a type-concatenation operator, conjoining the members from both left- and right-hand arguments, irrespective of whether they are inlined or not. We can, for example, 'derive' a different child class:

```
cousin : {
    func_2 :: {  ...  }
} : parent_class
```

Although `child` and `cousin` have been endowed with the same set of members, there are two important differences between them. 1) Types are concatenated left-to-right, so their members are in a different order. `child[1]` is `data_1`, while `cousin[1]` is `func_2`. 2) The left-to-right ordering of types creates an arrow of inheritance, so by Yazoo's reckoning `cousin` is not really a child of `parent_class`. Who is descended from whose tribe determines the ways that types may be specialized, as we explain below.

Suppose that we define some instance of a class

```
Bob :: parent_class
```

If we were sloppy we may have put this after the `code` marker of a function, or inside of a loop, in which case `Bob` might be redefined many times. This is fine – you can redefine anything as often as you like provided that you don't change its type.

```
Bob :: parent_class      | pointless, but legal
Bob :: parent_class
Bob :: parent_class
```

You can also specialize the type, by concatenating types *after*, but *not* before, the original type.

```
Bob :: parent_class
Bob :: parent_class : {  ...  }    | legal
```

Line 2 adds whatever additional members are contained in its braces to class instance `Bob`. But there is no turning back – you can never generalize a type. So the following would be illegal.

```
Bob :: parent_class
Bob :: parent_class : {  ...  }
Bob :: parent_class      |  illegal!  type mismatch
```

The rule is that, when redefining an object that already exists, each component of the existing type must be present, in the same order, in the new type. Only after all existing types have been recapitulated may we add (any number of) additional type specifications. The additions will be permanently added to the object's and/or member's type specifications (different define operators update different combinations of these).

```
a :: {  ...  }, b :: {  ...  }, c :: {  ...  }
d1 :: a,  d2 :: b:c
x :: d1 : d2
x :: a : b : c      | this is all perfectly legal
```

One potential point of confusion is that two separately-written *inlined* types are always considered different even if they are letter-for-letter completely identical. So, the following code will cause a type-mismatch error.

```
Joe :: { a :: ulong }
Joe :: { a :: ulong }    | t-m error
```

However, we have seen that the following is legal:

```
for counter in [1, 2]
    Joe :: { a :: ulong }
end for
```

Only separately-written (hence independently-compiled) pairs of braces are considered different codes. The compiler assigns a unique ID to each pair of braces it sees, and unfortunately it is not in the business of comparing what's *in* these braces to see whether they're compatible, so it assumes any two separate brace-pairs are totally unrelated. (For this reason earlier versions of the program would throw type-mismatch errors when the user executed a script file twice, unless each class or function definition `object :: ...` at root level was preceded by an explicit `trap(remove object)` in the script. The modern version of `run()` is intelligent enough to delete `object` before the script is rerun.)

Thus far we've been speaking in the language of classes to describe the effect of the inheritance operator. But, as we said at the beginning, inheritance works with most objects in Yazoo. It works with ordinary composite types and variables, obviously, since those are just a special case of classes (classes without functions). Inheritance also works with sets. In the case of sets the operator represents a true concatenation.

```
a :: {  Alice, Bob  }
b :: {  Charlie, David  }
c :: a : b
```

So `c` contains `Alice`, `Bob`, `Charlie`, `David`, in that order.

In all of our examples so far, each piece of code in the type specification has simply added members to the new variable or class. But these 'constructors' can just as easily destroy members as create them, so it is possible for the child to have fewer elements than the parent.

```
chocolate :: {
    cacao_solids :: protein
    cacao_butter :: fat    }

white_chocolate :: chocolate : {
    remove cacao_solids    }
```

We can perform equates, run loops, or do any other valid Yazoo operation when specializing a type.

```
tel_number :: {
    prefix :: 3_dig :: 4_dig :: ulong    }

DC_number :: tel_number : {
    prefix = 202    }
```

Since constructors are evaluated left-to-right, in neither of the last two examples could their order be reversed (the `remove` would fail in the former; `prefix` would not yet exist in the latter). These examples, by the way, demonstrate why Yazoo cannot un-specialize a variable, as many of these specializing operations are irreversible.

Finally, we can even apply the inheritance operator to functions. In practice this is often awkward and not terribly useful; inheritance is most useful for subroutines that do not return a value. The constructors (the part before the `code` markers) of two functions are concatenated as usual, and the codes (following the `code` markers) are run one after the other when the function is called. An example is given below.

```
absval :: {
    sign :: sbyte
    code

    sign = 1
    if args[1] < 0, sign = -1, endif
    args[1] = that*sign
}
```

```
sqrt :: {
    code
    args[1] = that^0.5
}

modulus_sqrt :: absval : sqrt
```

This would not work if the original function had a `return` statement, since Yazoo would never reach the second code. What we just did is cumbersome because we can't return the final value. A more promising approach might be to sequester any potentially heritable pieces of code into functions which can later be replaced.

```
sqrt :: {
    f :: { ; return args[1] }
    code

    return f(args[1])^0.5
}

modulus_sqrt :: sqrt : {
    remove f
    f :: { ; return abs(args[1])  }
}
```

One particular Yazoo stunt is to specialize an object that has *no* `code` marker, like a composite variable, with a type definition that *does* contain a `code` marker (i.e. a function). After this remarkable operation, the thing whose mother was a variable has acquired characteristics of a function, and you can run it and see for yourself that it will indeed run the newly-inherited code. Clearly, Yazoo blurs the distinctions between functions and variables. Clarifying this situation will be one of the first orders of business of the next chapter.

4 Yazoo bytecode

Life as a Yazoo script proceeds through three phases. A newborn script enters the world as a string of text stored in one of the user's variables. Metamorphosis to the second state, called *compilation*, involves the compiler taking that script, chewing it up and reforming the bits into a new, binary string – bytecode, in computer jargon. In some cases the programmer might want to tweak this bytecode a bit, but usually we are content to just let the butterfly be. In both stages 1 and 2 the script lives as an ordinary string in an ordinary variable of the user.

To attain the third and final stage of its existence, a compiled script transmigrates from a lowly string variable to a perch on the ivory bench of Yazoo's internal code registry. In Yazoo-speak, this process is called a transformation. A transformed script is no longer *contained in* a variable, so the user cannot modify it. Instead it has become the soul of some other variable, now more appropriately called a function because running it causes the script to do its thing within that function variable. The gatekeeper of the code registry, Yazoo's built-in `transform()` function, first scrutinizes each aspiring script for problems that could crash the interpreter, and is entrusted with a special set of error codes that it can throw when such a defect is encountered.

In this chapter we will discuss both the syntax of the bytecode string and what the transformed code does when it runs. Our purposes are twofold. One benefit is that the language of compiled bytecode is an arcane bit of lore that comes handy in certain (usually obscure) situations. The second justification is for a broader audience: Yazoo only executes bytecode, not scripts directly, so knowing how one's script translates into bytecode gives the ultimate insight into why it works in the way that it does. This chapter will show that scripting syntax is heavily constrained, and that the seeming plethora of entities in Yazoo is played by relatively few actors in the bytecode world. To begin with, we shall show that *all* objects in Yazoo are one of two things: primitive variables or functions.

4.1 Functions, revisited

One theme from the last chapter is that most everything in Yazoo conforms to a single syntax. For example, all objects use the same define operators. We can access the members of any composite object – function, classes, whatever – using the same index and member operators. Et cetera. We need to take these apparent similarities at face value. As a general rule: two things in Yazoo that look the same, smell the same and should be different, are probably really the same. By the end of this section, the reader should be persuaded that *all* Yazoo objects that are not primitive variables — this includes composite variables, sets, and classes — these animals are all of the same species. They are all (for lack of a better word) functions.

To begin with, let's dispel any notion that Yazoo has things called 'data types' or 'classes'; these are both really just composite variables. (We have already done all of this in various places in the last chapter, but it is worth repeating.) A data type is just a class with no member functions – Yazoo makes no distinction whatsoever between these two objects. It is a little bit less obvious why a class should be the same as a variable, but consider the following two equivalent ways of creating a variable:

```
my_class :: { ... }
var1 :: my_class
```

versus simply

```
var1 :: { ... }
```

The second method creates the variable `var1` directly, using precisely the same syntax that generated `my_class` in the first method. Therefore `my_class` must also be a variable. Indeed all supposed data types and classes are really just composite variables. The reason that Yazoo is able to get away without explicit type-objects is that any variable can be used as a template for defining other variables; the code

that defined the first is simply copied into the new variable. For example, after either definition of `var1` we could then spawn off another variable by writing:

```
var2 :: var1
```

Now let's wade into slightly deeper waters: a composite variable is the same as a function constructor. This implies that a variable can be thought of as a function without code. Let's create some function:

```
frac :: {
    num :: denom :: double

    code

    return num / denom
}
```

and strip away the coding section:

```
frac :: {
    num :: denom :: double
}
```

This is nothing more than a composite variable (or class – call it what you like). Indeed, when we created the `frac()` function it was this first chunk of code that generated the original function object. And in all respects this object works the same as any variable: we can read and write, add and delete members, etc. The *only* difference is that this function object has a coding section, which we can run by calling the function.

For that matter, it is also legal to 'run' composite variables and classes. It's just that nothing happens since their definitions lack a coding section.

In a similar vein, we can find two other kinds of Yazoo thing that maybe don't show much resemblance with functions or variables, but do look quite a bit like one another. These are sets and function arguments. Compare the contents within the braces of a set:

```
ToBuy :: { groceries, "bird", supplies("school") }
```

to what's inside the parentheses of a function call:

```
buy( groceries, "bird", supplies("school") )
```

They are identical. Not only that, but the syntax for accessing a function argument ('`args[1]`') is the same as the summons for an element of a set ('`ToBuy[1]`'). So it should be no surprise that these two things are one and the same: a function's arguments are really a set.

We now only have to merge our two piles: sets and function arguments are really just functions and variables in disguise. In a sense this shouldn't be surprising: we know from the last chapter that `args` may contain a `code` marker or semicolon and be run as a function in its own right. But it is not completely obvious how this can work, because we need to apply two syntactic rules to transform a set into something that looks like a variable (or function).

The first rule is that a comma breaks a Yazoo sentence just as well as the standard end-of-line does. Thus our demo set could have been written:

```
ToBuy :: {
    groceries
```

```
    "bird"
    supplies("school")
}
```

To understand the second rule, which is new, it is helpful to compare the set with an equivalent composite variable:

```
TB_var :: {
    mb1 := @groceries
    mb2 := "bird"
    mb3 := @supplies("school")
}
```

The three members of this variable contain or alias the same three objects that are in the set, and in the same order. Therefore '`ToBuy[1]`' is the same as '`TB_var[1]`'. There is a difference between the variable and the set: the aliases of the variable have names, while the members ('tokens') of the set do not. But this is a difference between aliases and tokens, not a fundamental difference between sets and variables. We can have objects that contain both tokens and aliases:

```
ToBuy :: {
    groceries
    "bird"
    for_Johnny := @supplies("school")
}
```

Recalling an earlier discussion on sets with named members, we could also have used the '`:=`' operator for either of the first two elements. Of course, since a set is really a composite variable we may construct it using any code we care to write, and since it is really a function we have the option of appending a coding section. (Whether or not that is good programming practice is entirely a separate question.)

The mystery of tokens has not yet been entirely resolved. We cannot see them in our scripts. We know that they are similar to aliases – but, as it turns out, they are usually not *quite* the same. They are inventions of the Yazoo compiler, so we will get our first good look at them when we start examining compiled code in the next section.

4.2 Compiled expressions

This paragraph marks a turning point, where we step away from Yazoo scripting and venture into the more abstract world of compiled Yazoo bytecode. Down here, syntax is a bit less fluid and intuitive, the logical structure is simpler and we are palpably closer to Yazoo's internal motor. It may sound like just an intellectual adventure, but we have a very practical reason for taking it: we are seeking a fuller understanding of how our scripts work. Learning about bytecode is a means to this end, because bytecode is what the fundamental physics of Yazoo operates with.

In general, 'bytecode' is a binary re-representation of a script – a string of signed longs in Yazoo's case – which encodes instructions to the machine more directly than a script does. Scripts are designed to be human-friendly, whereas Yazoo bytecode is Yazoo-friendly. As the two are somewhat at cross-purposes, Yazoo at some point has to translate (which is what the `compile()` function is for). The bytecode that Yazoo understands was nicknamed 'Hobbish' in memory of his parents' answering machine (which was in turn named after his sister), and that nickname still sticks around in various odd places.

It must be emphasized that Yazoo bytecode is *not* machine code. There are some similarities: the bytecode directly encodes Yazoo instructions, just as machine code laboriously specifies processor instructions. However, a Yazoo instruction is not at all the same as a fundamental instruction for the chip. A major syntactic difference is that Yazoo's bytecode is a recursive language. Whereas machine code is a simple list of commands – one instruction per line – Yazoo bytecode is a list of *sentences*, most of which correspond to

the sentences in the original script. Each sentence is composed of instructions called operators, and the operators themselves often take arguments which are other operators. Thus a given sentence can be composed of clauses and sub-clauses to a practically unlimited depth.

Because bytecode is a binary format, it makes for rather tedious reading in raw form. Back when Yazoo was still evolving, the author decided to write a 'disassembler' to give a textual representation to the bytecode where runtime errors were being flagged. Error messages have improved drastically since those days, and disassembly is no longer the way to go for most day-to-day scripting errors. But the disassembly tool is still around, and in the rest of this chapter we will make heavy use of it. It resides in start.zoo, partly for that script's own private use, but the routine is made accessible to the command-line user as well. To 'disassemble' a script, one makes the following call from the interactive command prompt:

```
print( disassemble( my_code_string ) )
```

where both `my_code` and the return value of the disassembler are strings. The output will look something like:

```
[print] ( def** ( sm -1 , scr { deq* ( sm -2 , cst "This is a sample script" ) } ) )
```

4.2.1 Anatomy of a compiled sentence

Yazoo bytecode is a free-standing format on its own, and one can write perfectly legitimate bytecode programs in it without ever compiling from a script. In fact, one can write many programs which *cannot* be compiled from any script. For now, though, we will always take some script as our starting point, and work out its bytecode translation. The teaser at the end of the last section came from the following script:

```
print("This is a sample script")
```

This one turns out to be a surprisingly tough nut to crack, and it will take us until the end of the chapter to get it open. But many other expressions have translations that can be inferred, in rough form, from a zeroth-order prescription which we'll give below.

The basic rule for translating a script into bytecode is to think of each operator as a function, and re-arrange its arguments so that they follow the operator. For example, in the following expression

```
b + 5
```

the addition operator is essentially a function with two arguments, and the expression could be heuristically rewritten as

```
Add(b, 5)
```

Notice how the word order gets scrambled – the operator comes *first*, followed by its left-hand argument, then its right-hand argument.

Just as functions sometimes modify their arguments directly instead of returning values (we often call these 'commands'), we can find Yazoo operators that do the same. For example, the equation

```
a = 12
```

can be thought of as the following command:

```
Equate(a, 12)
```

When expressions are built up from sub-expressions, the rule is that the outermost operator in a standard expression is also the outermost – i.e. first – operator in the functional representation. This ensures that outermost operator is evaluated last, since functions evaluate their arguments before they themselves do anything. Putting our last two examples together:

```
a = b + 5
```

we see that the equate operator goes last, *after* `b` and `5` have been added. The functional representation is thus:

```
Equate(a, Add(b, 5))
```

In such a way, we can build up arbitrarily complex sentences of functions that correspond to any reasonable sequence of operator expressions that we may encounter.

What we have been writing is fairly close to bytecode, but there is one further thing we would have had to change besides the word order. To make execution easier, the compiler replaces each variable name with an ID number. For example, '`a`' would be replaced by '5' if it was the fifth member name that the compiler had encountered when processing the script. This brings up a potential problem: when Yazoo encounters "`Add(..., 5)`" should it add 5, or the contents of variable '`a`'? To make matters worse, operators themselves are denoted by ID numbers; '5' corresponds to the invocation of a user-defined function which could also be legitimate to use here.

Yazoo discriminates numbers, variables, etc. by tagging each of these with an operator that specifies what kind of thing it is. For numbers we use either a 'slong' or 'double' operator; for variables the operator is really a 'member' operator (we have been speaking loosely when calling those things variables). Thus our expression turns into

```
Equate(member a, Add(member b, slong 5))
```

where we have to keep in mind that '`a`' and '`b`' will appear as numbers in the final reckoning.

Let's compare our guess to what the disassembler gives us. To disassemble the code officially, we can write:

```
compile("a = b + 5")
bc_str := R_string
print(disassemble(bc_str))
```

It's important to make sure that the compilation worked – `R_error_code` should be 0 after the `compile()` call. If we pass a bad `R_string` then the disassembler will probably crash, disassemble itself in confusion and bail out of Yazoo. In our case, we should get the following output:

```
equ ( sm 1 , add ( sm 2 , csl 5 ) )
```

If we look at it long enough, we can see that we hit the nail on the head. The disassembler does have its own abbreviations for commands, such as 'sm' for search-member and 'csl' for constant-signed-long. More confusing is that it writes member IDs rather than member names, since it doesn't know the mapping between the two. We can improve our output by compiling into the `AllNames` namespace and then passing this optional third argument to the disassembler telling it to read member names from `AllNames`.

```
compile("a = b + 5", AllNames)
bc_str := R_string
print(disassemble(bc_str), *, AllNames)
```

(Notice that, if we want to pass the third argument of `disassemble()`, we also need a placeholder for the second – just set it void.) The output is now:

```
equ ( sm $a , add ( sm $b , csl 5 ) )
```

which is very close to what we originally guessed.

We've been using the disassembler, so we are still a step away from what Yazoo actually looks at when it runs the program. The final step in reducing our script is to replace the operators with the operator ID numbers. Equate is special; it is one particular instance of a general-purpose define/equate operator (other instances being, e.g., '`::`' and '`=@`'). All define-equate operators have the ID number 11, but the variants are distinguished from each other by a second flags word, which in the case of equate is set to 1.

Let's go back to the original disassembly that made no mention of `AllNames`, where the member IDs show up as 1 and 2. Using the reference section to obtain the operator IDs (and replacing the member names with IDs), we obtain the following machine-level version of our bytecode:

```
11 1 ( 18 1 , 40 ( 18 2 , 50 5 ) )
```

The raw output of the compiler is a string; converting that string to an array of signed longs gives us something that we can compare with:

```
> words[*] :: ulong
> compile("a = b + 5")
> words[*] =! R_string
> sprint(words)

{ 11, 1, 18, 1, 40, 18, 2, 50, 5, 0 }
```

so we were correct. The only new item in the 'real' compiled array is the terminating zero, which signifies the end of the script. Every script ends with a null word, which tells the interpreter to fall back to the enclosing function – or, if that was the starting script, to exit the program.

Pathnames A Yazoo pathname looks superficially like a C pathname: it starts with the name of some member, which may be followed by some combinations of dots and brackets that refine the path. For example, the expression

```
my_struct.array[5].x
```

could be a legitimate path in either C or a Yazoo script. However, the roles of those dots and brackets are completely different. In C the compiler reduces the entire pathname to some fixed offset from `my_struct`. By contrast, Yazoo with its dynamic memory cannot know at compile-time where, or even if, each subsequent piece of the path can be found; it has to search for those at runtime. Thus the entire path is stored member-by-member in the compiled bytecode, with the dots and brackets retained as operators which move Yazoo's search-beam from one variable to the next.

A path typically begins with some member that Yazoo can find on its own, by searching backwards from the current function, followed (optionally) by members or indices which blaze a new trail forward one-step-at-a-time from that first member. The ordering in compiled code is, basically, backwards. Again thinking of the step-operators as functions, we reason that since the innermost function gets evaluated first, that must correspond to the first step. In heuristic language we might write our path as

```
step_to_member( step_to_index( search_member x, 5 ), my_struct.array )
```

where, as before, we have explicitly written the operators out longhand. We can compare with the disassembly.

```
dqa* ( sm $550 , sID ( sti ( sID ( sm $my_struct , $array ) , csl 5 ) , $x ) )
```

For now we will ignore the first few operators, up through `sm $550` (that number will change) – those will be explained at the end of the chapter. The step-to-member operator is abbreviated `sID` (since member names are replaced by ID numbers), and the step-to-index is abbreviated `sti`. Notice that step-to-member is a different operator from search-member (`sm`), since the latter begins a path, whereas the former *continues* a path and so requires an additional first argument specifying the path unto that point.

Some array operators require two arguments. The most basic example is the step-to-indices command (as opposed to step-to-index), which accepts two additional arguments after the initial path. The compiled forms of the array-element-insertion operators '`[+...]`' and '`+[...]`' also take index ranges, although the compiler accepts single indices and simply duplicates the entry to generate a range. For example, the code

```
array[+5]
```

translates as

```
iiu ( sm $array , csl 5 , csl 5 )
```

When confronted with a register (built-in variable), the disassembler will simply print out a register abbreviation, for example '`[R_sl]`' for R_slong. The actual bytecode contains two words: the first is the register operator, and the second is an ID number of the register to access. The ID numbers of the registers are given in the reference section.

The compiler does something funny to the pathname to the left of a define-operator: it replaces the 'search-member' operator with a 'step-from-this-to' sequence. For example, the sentence

```
my_var :: ulong
```

compiles into

```
def ( sID ( this , $my_var ) , ul )
```

This little oddity exists to prevent compiled define statements from accidentally re-defining members of enclosing functions. For example, if `my_var` had been defined in some class that also contains the active function, this command will define a *new* `my_var` member in the active function rather than modify the class's member; subsequent `my_var` invocations within the function will then access the new member. This allows functions to reuse common names, such as for-loop counters, without having to worry about accidentally overwriting global variables. Admittedly, that means we need a workaround when a script actually *does* want to redefine part of its parent. In the end this convention was adopted because it seemed the lesser of the two evils, and because it saves us from some rather uncouth programming practices.

Inlined constants Inlined constants – numbers and strings written directly into a script – survive compilation relatively intact and are copied in one piece into the bytecode. If we were to try reading bytecode in the raw, without the aid of an operator ID table or a disassembler, these occasional sprinkles would be the only recognizable parts of the original script. There are three types of inlined constants in bytecode, each associated with a unique operator. The raw data follows in subsequent words of bytecode.

Numeric constants are stored either as signed longs or double-precision floating point numbers. The compiler chooses the latter only if the number cannot be stored in a `slong` integer, either because it has a fractional component or it is too large to be stored as an integer. Only numbers typed directly into

the script are stored as constants; all numeric expressions (even trivial ones such as `5+2`) are evaluated at runtime. Signed longs are written with a constant-signed-long operator (1 word), followed by the signed long value (1 word). Doubles are written in bytecode with a constant-double operator (1 word) followed by the double-precision value (2 words).

String constants have the added complication of having indefinite length. Since Yazoo strings are often used for storing binary data, Yazoo opts for the 'Pascal' string convention rather than the C format: the byte-length of the string is stored explicitly, and there is no terminating character. The constant-string operator (1 word) is followed by the character length of the string (1 word, signed long, must be positive) followed by the bytes of the string (N/`size(slong)` words rounded up). Notice that the string length refers to the number of characters, *not* the number of bytecode words; and that the final byte of string data is followed by anywhere from 0 to one less than `\texttt{size(slong)` null bytes to fill in the last word of bytecode.

The disassembler doesn't show the string-length word – it just writes out the string in quotes. The phrase `"Hello"` disassembles to

```
cst "Hello"
```

which corresponds to the following four bytecode words:

```
{ 52, 5, 1214606444, 1862270976 }
```

Here, the string characters are contained in the upper five bytes of the last two words.

4.2.2 Flow control: conditionals and 'goto's

In the last chapter we described the four phony flow-control 'commands': `if`, `for`, `while` and `do`. Phony because they don't correspond to bytecode operators; the Yazoo compiler turns them into totally different expressions involving 'goto's. This section describes the formulae which the compiler uses to translate these commands.

Yazoo sports three goto operators: an unconditional jump, and jump-if-true and and jump-if-false operators. Each goto sequence begins with the relevant operator ID (1 word) followed by a jump offset (1 word). The jump offset should be read as a signed long (so it can be negative). It gives the relative position, measured in bytecode words from the location of the *jump-offset word*, of the sentence to jump to if the goto is to be performed. The jump must be to the start of some compiled sentence (or else the terminating null word) – otherwise `transform()` throws an error. In the case of the two conditional gotos, there is a final argument following the jump offset which is the condition on which to jump.

if An `if` statement executes its code if its conditional argument is true. That is, it skips the code – i.e. performs a goto – if the argument is false. Thus the `if` translates into a jump-if-false operator, where the jump offset is to the next `elseif`, `else` or `endif` marker in that if-block.

Any `elseif` translates the same way as an `if`, since it is basically just another `if` in series with the first. On the other hand, `else`s and `endif`s are just markers delineating different region of the script, and don't translate into anything at all. Since the various conditions are mutually exclusive, every conditional block (except for the last one) ends in an unconditional goto to the `endif`.

The following script gives an example of every type of if-related marker:

```
if a == 1
    c = 3
elseif a == 2
    c = 7
else
    c = 9
endif
```

```
print(c)
```

It has the following bytecode translation:

```
jf 14 ( eq ( sm $a , csl 1 ) )
equ ( sm $c , csl 3 )
j 22

jf 14 ( eq ( sm $a , csl 2 ) )
equ ( sm $c , csl 7 )
j 7

equ ( sm $c , csl 9 )

[print] ( def** ( sm $639 , scr { dqa* ( sm $640 , sm $c ) } ) )
```

Notice how the code blocks are demarcated by a chain of jump-if-falses (`jf`) at their beginnings, and unconditional jumps (`j`) at their ends. The `else` has no unconditional jump; that would be redundant since it would just jump to the next sentence.

for The compiler boils a for-loop down to 1) a counter initialization, 2) a conditional jump-if-true (`jt`) to the `endf` when the loop is finished, then 3) the code to be looped, followed by 4) a counter-increment and finally 5) an unconditional jump at the end back to the conditional goto. Schematically, the following code

```
for counter in [start, end] step stp
    ...
end for
```

gets rearranged into

```
counter = start

loop_start:
if counter > end goto loop_end
    ...
    counter = counter + stp
goto loop_start

loop_end:
```

The loop-fields `counter`, `start`, `end` and `stp` were written as above as variables, but each of these can be more general expressions as well. The entire compiled expression gets plunked down everywhere that field appears in the schematic above. For example, the following script

```
for (c1 :: ulong) in [0, top(array)]
    array[c1] = 0
end for

print(array)
```

disassembles into:

```
equ ( def ( sID ( this , $c1 ) , ul ) , csl 0 )
jt 50 ( gt ( def ( sID ( this , $c1 ) , ul ) ,
    [top] ( def** ( sm $650 , scr {
        dqa* ( sm $651 , sm $array ) } ) ) ) )

equ ( sti ( sm $array , sm $c1 ) , csl 0 )
equ ( def ( sID ( this , $c1 ) , ul ) , add ( def ( sID ( this , $c1 ) , ul ) , csl 1 ) )
j -50

[print] ( def** ( sm $652 , scr { dqa* ( sm $653 , sm $array ) } ) )
```

Notice how the `def ( sID ( this , $c1 ) , ul )` motif (corresponding to `c1 :: ulong`) occurs four times, so Yazoo will not only define `c1` in the initialization step, but also 'redefine' it three times per iteration of the loop. This is perfectly legal, but it can slow down short loops considerably. Likewise, the `top()` function is called anew (once) per loop iteration, which is also inefficient.

We can see now why negative steps don't work in Yazoo if the step is contained in a variable: the counter is always assumed to be increasing unless the step is a negative constant, and so the loop stops when the counter is *greater* than the second range field. A loop that tried to decrement its counter would probably satisfy this criterion without ever running the loop once; the other (worse) possibility is that the loop would run forever, continually decreasing the counter variable and falling ever further from its goal. Our example shows that if the optional step parameter is not specified, then it just defaults to 1: i.e. "`csl 1`" in bytecode-speak.

while A while-loop is essentially an if-block with an extra goto at the end. The translation consists of a conditional goto to the `endw` if the condition is false, followed by the loop-code and lastly an unconditional jump back to the first goto. For example, the following script

```
while random() < .9
    print(".")
end while

print("\n")
```

compiles into

```
jf 30 ( lt ( [random] ( def** ( sm $663 , scr { } ) ) , cf 0.9 ) )
[print] ( def** ( sm $664 , scr { deq* ( sm $665 , cst "." ) } ) )
j -30

[print] ( def** ( sm $666 , scr { deq* ( sm $667 , cst "#" ) } ) )
```

Since the condition is checked before the loop-code is encountered, the loop will not run even once if the condition starts out false.

do The simplest flow-control block is the `do-until` loop. The compiled loop consists of the code within the loop, followed by a single conditional if-false jump back to the beginning of this code. Nothing comes before the loop code, so it is guaranteed to run once. For example, the following script

```
do
    cmd = input()
until cmd == "quit"
```

corresponds to

```
equ ( sm $cmd , [input] ( def** ( sm $677 , scr { } ) ) )
jf -13 ( eq ( sm $cmd , cst "quit" ) )
```

4.2.3 `def-general` flags

This section is devoted to a single word of bytecode: the 'flags' immediately following the def-general operator. Given that such diverse scripting functions as define (`::`), equate (`=`) and equate-at (`=@`)—among others—are all def-generals, distinguished only by their flags, this flag-word must pack a punch. The unifying theme is that all these operators *copy* something, or some things, which are determined by these flags — data for an equate, or type/code in the case of define, or a reference with the equate-at. In all cases the copying is done from some source variable, constant or type on the right-hand of the operator in the original script, into some destination member and/or variable which is on the left.

The def-general flags consists of nine Boolean flags, stored in the lowest nine bits in the 32-bit flag word. For each flag that is 'on' the corresponding bit is set to 1. The flags are numbered from the lowest bit to the highest bit starting from zero. To find the value of the flags-word corresponding to given flags, shift binary 1 to the left N times for each flag N, and sum the resulting values. For example, define-equate-at (`:=@`) has flags 1, 2 and 4 set. Thus the value of the flags-word is

$$\begin{aligned} fw &= (1 << 1 = 2) + (1 << 2 = 4) + (1 << 4 = 16) \\ &= 22 \end{aligned}$$

The def-general flags are listed in the reference section, along with the flags corresponding to each scripted def-general operator. Here we will describe each flag in turn, starting from flag 0 in the lowest-order bit.

Flag **0** is the **equate** flag. When set, the data stored in the source variable is copied into the destination variable. This requires the structures of these two variables to be similar if they are composite. However, their types, as defined by their codes, may be different. For example, the following is allowed:

```
{ a :: ubyte, b :: string } = { c :: ulong, "" }
```

whereas the following

```
a[2] :: double
a = { 5.5, 3.9 }
```

is disallowed because the source contains two members, while the destination has only a single width-2 member. (The `copy()` routine handles the latter case in interactive mode.)

Flag **1** is the **update-members** flag. It causes the destination member to be updated to the type of the source *variable* (not the source member!). For example, suppose we write:

```
a :: *, b := 5
a = @b

c :: a
```

Member `a` has no type, but the variable it points to is a signed long. Thus the member named '`c`' will be defined as a signed long, because the define operator sets the update-member flag. Thus trying something later on like

```
d := "some string"
c = @d
```

would cause a type-mismatch error.

Flag **2** is called the **add-members** flag. If the destination member can't be found and this flag is set, the destination member is created inside the currently-running function. This works even for members without a name, notably members created implicitly while defining arrays. For multi-step paths where several members have to be created in a row, typically only the last member earns a type; the others will default to a void type which can be aliased to any variable later on. For example, suppose that `array` had not been defined until we wrote:

```
array[10].title :: string
```

Since define sets the add-members flag, Yazoo will create an `array` member (without a type) pointing to a composite variable; a width-10 unnamed member within (again lacking a type) pointing to a composite variable; and finally a member entitled '`title`' which will have a string type and point to some string variable.

Flag **3**, the **new-target** flag, performs two tasks. First, it creates a new destination variable if none existed already (i.e. if the destination member was void), but it will not overwrite an existing variable. Second, if necessary it updates (specializes) the type of the destination variable, regardless of whether or not it had just created that variable. The new type is the type of the source *variable* (not member), so it requires that the source member not be void. New-target is thus complementary to both the update-members and add-members flags.

The member-define operator (`*::`) sets the update-members flag, but not the new-target flag, so it operates only on members. On the other hand, the variable-define operator (`@::`) has its new-target flag set but the update-members flag clear, so it will specialize variable but not member types. It can however create new members since it sets the add-members flag. Plain old define (`::`) sets all three flags.

Flag **4** is the **relink-target** flag; it instructs Yazoo to make the destination member an alias of the source variable. This flag is set by the equate-at and define-equate-at operators. The destination (left-hand) must be an entire variable, since all indices of a given member must have the same target. An example of an illegal maneuver involving a relink is the following:

```
a[10] :: b[5] :: ulong
a[1, 5] = @b[1, 5]
```

In a sense, relink-target is the third of three pillars of the def-equate flags. Whereas post-equate copies data, and update-members and new-target copy code, the relink-target flag copies the target reference (which is something like a pointer) of the source member.

Flag **5** is named **run-constructor**; it causes the constructor of the destination variable to be run after it has been created but before any data has been copied. The constructor is the part of a script before the first `code` marker or semicolon. If the variable has several concatenated codes, the constructors of each code are run in order from first code to last. Primitive variables have no code, so they are unaffected by this flag.

The constructor-run is the 2nd-to-last operation performed by the def-equate operator, with the actual equate being the last. This is why we are able to copy composite variables in one step:

```
comp1 :: { ... }
comp2 := comp1
```

The new variable `comp2` is defined to have the same code as `comp1`, so when its constructor runs it should

grow the same set of members that `comp1` has. So the final equate should not have any problems, as long as we didn't modify `comp1` after defining it. A common case where the constructor does not recreate the variable is in implicit variable definitions, such as:

```
array[12] :: ulong
arr2 := array    | causes an error
```

where `array` and therefore `arr2` have no constructor, so the latter is created empty even though the run-constructor flag was set.

Flag **6** is called the **hidden-member** flag. We have not encountered hidden members yet; suffice it to say that members whose indices were defined with this flag set simply do not show up under the user's radar, so they have to be accessed by name. For example, suppose we have a variable, call it `three_members` because it has three members, and the members have 3, 4 and 5 indices respectively. Suppose that the second member is hidden. Then

```
three_members[1, 3]
```

spans the first member, and

```
three_members[4, 8]
```

spans the third member. The second member is there – but its index is hidden.

The equate (`=`) and compare (`==`) operators skip hidden members of both (left-hand and right-hand) arguments. In addition, `print()` does not print the contents of hidden members.

Flag **7** is the **unjammable** flag. A jamb is an alias that prevents a member from being resized: for example, if two members alias the indices of an array, ordinarily neither one can resize the array since doing so would also affect the other member. However, if one member is defined as unjammable, then it cannot jam the other member: the second member *can* be resized and the first member, which now has the wrong number of indices, becomes 'unjammed' – i.e., dead. An unjammed member has to be re-aliased before it can be used again without causing an error.

proxies Flag **8**, the final and left-most flag, is the **proxy** flag. Until recently the author considered proxies to be so obscure that he did not even bother writing an operator for defining them – they had to be byte-coded in by hand. That has changed: the operator '`#::`' now defines proxies, as in: `a[10] #:: double`.

Proxies distinguish themselves only when they are defined as arrays. Ordinarily, the indices of a member must point to a contiguous block of memory, so we can't do things like re-alias some of the indices of an array if they do not span one entire member. (The distinction between members and indices is explained in the discussion on arrays.) However, each index of a proxy-member may point to an entirely different variable, as long as its type matches. So if we follow up the definition of our array with the lines below, we get:

```
> b := 3.14

> a[4] =@ b, a[10] :: double

> sprint(a)

{ *, *, *, 3.14, *, *, *, *, *, 0 }
```

As we can see, when the proxy array is defined each element is initially void. We can create variables for them by subsequently defining the array elements one by one, using just the normal define operator '`::`'.

The advantage of proxies is that they bring the power of array manipulation to bear on aliases. There is also a disadvantage, which is that only one proxy can be accessed at a time. Any multiple-indices operation such as

```
array[2, 5]
```

is illegal when dealing with proxies, even if only one member is involved.

4.3 Inventions of the compiler

If Yazoo had a favorite philosophy it would surely be reductionism. In its eyes, everything that goes on inside its world can be explained by a very small set of physical laws. For example, functions and sets are translated and executed using exactly the same rules; Yazoo doesn't treat one as a function and the other as a variable because it does not know the difference between the two.

Historically speaking, this reductionist approach was forced from the top down: the author dreamt up a syntax that he liked, then searched for some minimal set of rules that would cause a script written in this ideal syntax to do the right thing when executed. As it turned out, this game would only work if Yazoo was allowed to add a few embellishments to the script at compilation. If we were able to see these little extra snippets of code, they would explicitly show us how sets could be construed as functions and vice versa.

Just to drive home the point that we haven't finished our fundamental theory of Yazoo yet, consider that we don't even have a good account for something as simple as

```
print("Hello")
```

The reason is that Yazoo's compiler adorns even this seemingly trivial script with two snips of code that we would only notice if we were to disassemble the script. They represent the two, really the only two, kinds of embellishment from the compiler, so we will be finished with our exposition of Yazoo once we've dealt with each of them in turn.

4.3.1 Tokens

A token is a bare reference to an object without an operation having being associated with it. In the examples below, there is a token associated with every instance of '`a`', '`b`' or '`5`'.

```
set :: { a, b, 5 }
print( a, b, 5 )
5
```

For the moment, we'll concentrate on that third 'sentence'.

A token is to Yazoo what an object is to an English-speaker. If we turned back the clock and tried to communicate with a cave man, we might get a lot of sentences like:

```
rock
```

which our brains would probably embellish with subject and verb to mean:

```
[That which I want to draw your attention to] [is] the rock.
```

Yazoo thinks along exactly the same lines. When it comes across some terse command like

```
5
```

it rolls its eyes and compiles something similar to:

```
var1 := 5
```

A couple of caveats have to be made. First, the member that '5' is assigned to isn't actually named '`var1`', as that would cause confusion if we had happened to be using a `var1` of our own in our script. Secondly, the operator that follows it is somewhat modified from a standard define-equate.

The way Yazoo avoids conflicts between token 'names' and user-defined names is by compiling tokens into members with negative ID numbers, while user-defined members are assigned positive IDs beginning from 1. (All bytecode words are read as signed long integers.) Token IDs start from -1 and count downwards, so the compiler needs to keep track of how many tokens it has assigned so far. Thus if N is the byte-size of a long word, then the first N bytes of a namespace (a string passed between the user and compiler that keeps track of member names) contains the number of tokens that have already been declared; the textual names of the user's members start at byte $N + 1$.

The most accurate portrait of a token is its disassembly, which for a "`5`" script is

```
deq* ( sm -1 , csl 5 )
```

or

```
deq* ( sm $1 , csl 5 )
```

if we use the compiled namespace in the disassembly. The (first and only) token variable has the ID -1 (the `$` signifies a negative ID). Note that the disassembler writes `deq*` to indicate that it differs from the user's `deq`: it has the unjammable flag set, and the update-members flag cleared. The unjammable flag prevents tokens from jamming arrays left and right. The update-members flag is cleared to prevent an error being thrown in case the same token is re-assigned to a variable of a different type (which can happen if, for example, a particular argument to a function changes its type between function calls).

Different objects translate into different kinds of tokens. A reference to a variable or a function, such as

```
my_var
```

translates into

```
dqa* ( sm $1 , sm $my_var )
```

Again, it is a modified `dqa` operator: the update-members flag has been cleared, and the unjammable flag set. If Yazoo senses that an array is being passed — for example, if the command was "`my_array[*]`" — then it modifies the expression somewhat:

```
dqa* ( s* ( sm $1 ) , s* ( sm $my_array ) )
```

which is analogous to `arr1[*] := @my_array[*]`. Finally, if the compiler encounters a code of any sort – function, set, whatever – sitting naked on its own line, it will wrap it in a token having a modified define operator. So

```
{2, 5}
```

turns into

```
def* ( sm $3 , scr { deq* ( sm $1 , csl 2 ) , deq* ( sm $2 , csl 5 ) } )
```

The flag modifications are the same: `def*` equals `def` minus update-members, plus unjammable.

This last example deserves a second look, because we can see not only that the set itself is represented by a token, but also that the set's members are tokens as well. Think of the set as a function, or a function constructor (since it lacks a `code` marker/semicolon). And remember that line breaks can be marked by commas as well as by end-of-lines. Essentially the constructor makes two tokens to hold the two numbers that the set represents. Had the numbers been variables instead, the tokens would have been aliases, which explains how sets can hold objects that are also somewhere else. Finally, subsets are simply composite variables within the set – functions within the function whose constructors in turn endow them with the appropriate members.

Tokens also help to explain function arguments. The code inside the parentheses of a function call is essentially a script; the fact that it is surrounded by parentheses rather than braces is only due to common syntactic convention that Yazoo adopted. This argument-code, or at least the constructor component of it, fills the `args` variable with a number of tokens (or, if the user wishes, explicitly-defined members) that the function will later see as `args[1]`, `args[2]`, etc.

One conspicuous mystery remains: what, and where, is the `args` variable? Identifying `args` and its cousins is the final project of the chapter.

4.3.2 Hidden members

We saw in the last section how Yazoo surreptitiously creates nameless members – tokens – that we can only access with the index operators. The user might be spooked to learn that his scripts are probably crawling with another kind of member that is even harder to detect: which not only have ineffable names, but can't even be reached by index because they have their 'hidden' flags set. These little elves may be invisible to the user, but they nevertheless perform important services for sets and function calls.

To begin with, consider a set, or more generally any sort of code block. An expression like

```
my_set :: { 5, 9 }
```

is straightforward, because a define statement looks for some type-defining right-hand argument, and a block of code fits that bill. On the other hand:

```
my_set = { 5, 9 }
```

is more problematic, because equate expects its right-hard argument to be some sort of data: either an inlined constant or a variable. The stuff in braces is one step removed from that: it is only code – constructor code. The constructor codes for tokens bearing the data that we want to copy, but we will only have that data in hand after the constructor has been run in some variable space.

In other words, that second command—which is of course legal—seems to imply that either equate does something complicated when it sees a set, or else that Yazoo's compiler cooks the expression in a way that makes it more digestible. Option 2 turns out to be correct. So how could Yazoo explicitly run the set's constructor – or at least, how would we do it? Probably by writing something like

```
my_set = ( temp :: { 5, 9 } )
```

That would make things simple because equate would just be copying data from (by the time it got around to it) a fully-formed variable. This is (almost) exactly what Yazoo's compiler will do with our original expression.

To see exactly how Yazoo fleshes out our set equation, we of course have to disassemble it. We get:

```
equ ( sm $my_set , def** ( sm $3 , scr { deq* ( sm $1 , csl 5 ) , deq* ( sm $2 , csl 9 ) } ) )
```

and we find one a new ingredient: the `def**` operator defining invisible variable number 3 (corresponding to `temp` above). Unlike tokens which pop up in a single-starred `def*`, `dqa*` or `deq*`, this new thing has two asterisks to symbolize that it is doubly invisible, because its hidden flag is set. The hidden flag is the only difference between singly- and doubly-starred operators.

What is true for the right-hand side of an equate is also true for the left. Our first two lines of example could have been replaced with

```
{ a, b } = { 5, 9 }
```

in which case Yazoo would insert two hidden members:

```
equ ( def** ( sm $3 , scr { dqa* ( sm $1 , sm $a ) , dqa* ( sm $2 , sm $b ) } ) ,
         def** ( sm $6 , scr { deq* ( sm $4 , csl 5 ) , deq* ( sm $5 , csl 9 ) } ) )
```

Furthermore, this trick works for other operators besides equate as well. Any time Yazoo needs to convert a script, or concatenation of scripts (`{} : {}`), into data, it will do so by slipping in a hidden variable. For example, our disassemblies would look much the same if we were to use forced equates instead.

Besides the `def**` operator, there is also a `deq**` whose sole habitat is the right side of a `return` statement. A function always returns some whole variable, not stand-alone data – we know this has to be true because expressions like

```
f().a :: double
```

are legal. (That adds or modifies member `a` within the return variable.) So for statements like

```
return 5
```

a variable has to be created to hold the return data. A scripted analog of Yazoo's fix would look like

```
return ( temp := 5 )
```

and the disassembly of the real thing is:

```
ret ( deq** ( sm $1 , csl 5 ) )
```

Finally, to return to a nagging question from earlier: what about the `args` variables? Nobody has ever seen one explicitly outside of its function, and not surprisingly, the reason is that they all lie behind hidden members. It's helpful to take a simple example, say,

```
sprint("Hello")
```

and look at its disassembly:

```
dqa* ( sm $3 , f ( sm $sprint , def** ( sm $1 , scr { deq* ( sm $2 , cst "Hello" ) } ) ) )
```

The entire expression is of course embedded inside a `dqa*` because Yazoo tokenizes all user-defined function calls. Concentrate on the bytecode function call which begins with `f (...`. It takes two arguments: the first

is the function variable, and the second is the argument *variable*. The argument list – `args` – needs to be a variable, not code. By now we know exactly how to turn the code in parentheses into a variable: we need to hidden-define a member with the argument code. And we see Yazoo doing exactly that with the `def**`.

It may seem trivial, but just to make sure we've touched home plate let's revisit the last taunting example from the beginning of this section. It was:

```
print("Hello")
```

where we are back to `print()` (i.e. not `sprint()`). (So it is now a built-in, rather than user-defined, function.) The disassembly looks like:

```
[print] ( def** ( sm $1 , scr { deq* ( sm $2 , cst "Hello" ) } ) )
```

The first thing to notice is that there is no preceding `dqa*`. Yazoo knows which of its own internal functions are worth making tokens out of, and `print()` is not one of those. Secondly, there is no function variable, obviously, since `print()` is hard-wired in. But the rest should be straightforward: the arguments to `print()` are constructed inside of a hidden variable before the `print()` function even begins. And if there is a problem in the arguments, then fine: Yazoo will halt execution and give the appropriate error before `print()` has a chance to run.

Which leads us to one last, microscopic little detail regarding the hidden arguments of the built-in `trap()` function. Unlike the other built-in functions, `trap()` needs tight control over its arguments when they run, carefully logging the type and location of any errors or warnings and allowing execution outside of `trap()` to proceed normally afterwards if an error does happen. In other words, `trap()` wants to run its arguments *by itself* and in its own way, rather than have them constructed automatically before it can control the situation. It follows that the hidden argument definition in a `trap()` call should *not* have the run-constructor flag set.

Thus we have one final species of the def-general clade to catalogue. Take the script

```
trap( exit )
```

(which, by the way, will *not* quit the program). Its disassembly looks like

```
[trap] ( def-c** ( sm $1 , scr { exit } ) )
```

It's the `def-c**` — in the ponderous taxonomy of the disassembler that means 'hidden-define-minus-constructor'. This last and rarest of birds, found exclusively in the function call of a `trap()` function, differs from `def**` only in that it lacks the constructor flag. The flags of this operator, and those of the other def-general operators that the compiler creates, are all tabulated in the reference section.

5 Reference

5.1 Operators and reserved words

`( ... )` : group terms in an expression

Numeric operators:

`x + y` : addition

`x - y` : subtraction

`x * y` : multiplication

`x / y` : division

`x ^ y` : raise to a power

`i mod j` : modulo (integer only)

Boolean operators:

`A == B` : if equal

`A ==@ B` : if same reference

`A /= B` : if not equal

`A /=@ B` : if different reference

`A >= B` : if greater than or equal to

`A > B` : if greater than

`A <= B` : if less than or equal to

`A < B` : if less than

`A and B` : if A and B (both arguments are always evaluated)

`A or B` : if A or B or both (both arguments are always evaluated)

`A xor B` : if A or B, but not both

`not A` : true only if A is false

Array/member operators:

`[ A ]` : step into array index A

`[ A, B]` : step into array indices A thru B

`[^ N ]` : resize array to given size N, step into all indices 1-N

`[+ A ]` or `[+ A, B ]` : insert indices (append to upper member)

`+[ A ]` or `+[ A, B ]` : insert indices (append to lower member)

`[*]` : step to all indices; sometimes can resize if necessary

`A.B` : step from A into member B

`remove A` : remove member A

`delete A[...]` : delete specified indices

Define/equate operators:

`A :: B` : define A (member with a variable) with the type of the variable that B points to

`A = B` : copy data from B to A

`A := B` : define A to type B, copy data from B to A

`A =@ B` : make A an alias of B

`A :=@ B` : define A and make it an alias of B

`A @:: B` : define A to be of type B (variable type only)

`A *:: B` : define A to be of type B (member type only)

`A =! B` : copy data from B to A; even between dissimilar data types

Predefined variables:

`this` : the function currently executing

`that` : on the right hand side of an equate, the variable on the left

`args` : the argument variable to the function currently executing

`nothing` or `*` : the void

Program flow:

`if A, ..., elseif B, ..., else, ..., endif/end if` : execute if A, B, etc. true

`while A, ..., endw/end while` : execute code as long as A is true

`do, ..., until A` : execute code until A is true

`for I in [ A, B ] { optionally:  step C }, ..., endf/end for` : iterate code

`A( ... )` : call A as a function with specified arguments

`code` or `;` : begin a new code block

`return A` or `return` : exit function with return variable A (if specified)

`exit` : exit Yazoo

Data types:

`ubyte`, `sbyte` : unsigned and signed byte

`ushort`, `sshort` : unsigned and signed short integer

`ulong`, `slong` : unsigned and signed long integer

`single` : single-precision floating point (real)

`double` : double-precision floating point (real)

`block N` : non-numeric block of N bytes

`string` : string of characters (bytes)

`"..."` : inlined string containing given characters

`{ ... }` : inlined function, set, class or data type

`A : B` : a composite type of A specialized by the code of B

`A << B` : the code/type of B substituted into the variable space of A

`A # B` : the Bth code block of variable A

5.2 Bytecode operators

The following list enumerates all of Yazoo's bytecode operators. Each entry consists of: a signed-long bytecode ID, a name in brackets, and then the arguments for that operator separated by commas. Arguments in plain text are 'fixed-width' – that is, takes up a set number of long words. Unless otherwise indicated, a fixed-width argument is one long word in size and should be read as an signed long. Arguments in italics are themselves bytecode expressions, which can span arbitrary word-lengths.

0 [null] : marks the end of a code block

1 [return], *return_variable* : exits the function and returns the specified variable

2 [register], ID : returns the register variable with the specified ID

3 [built-in function], ID, *args_variable* : runs the built-in Yazoo function with the given ID and arguments, and returns the appropriate register (if any)

5 [user function], code_skip, *function_variable*, *args_variable* : runs the given user function with the specified arguments, and returns the return variable (if any). Runs the coding block after *code_skip* `code` markers or semicolons – i.e. the constructor if this is 0, the normal function if it is 1, etc.

6 [args] : returns the `args` variable for the current function

7 [jump], offset : jumps the program counter to the position of the offset word plus *offset* (a signed long) long words

8 [jump-if-true], offset, *condition* : jumps the program counter to the position of the offset word plus *offset* long words (offset is a signed long), if the conditional expression evaluates to true

9 [jump-if-false], offset, *condition* : moves the program counter to the position of the offset word plus offset long words (offset is a signed long), if *condition* is false

10 [exit] : throws error 47 which, unless caught by `trap()`, causes Yazoo to exit with no error

11 [def-general], flags, *LH_var*, *RH_var* : applies the define/equate command specified by the flags from the source *RH_var* to the target *LH_var*

12 [forced_equate], *LH_var*, *RH_var* : copies the raw data from the source *RH_var* into the target *LH_var* if their byte-sizes match

13 [code number], *var*, *code number* : causes an enclosing function call to execute the given *code number* of *var*

14 [substitute code], *var*, *code* : returns *var* but with the given *code* instead of its native code

15 [append code], *f1*, *f2* : returns *f1* but with the concatenated code *f1* + *f2*

16 [remove], *member* : removes the specified member

17 [search member], ID : searches backwards from the current function for the member with the given ID (a signed long)

18 [step to member], *starting_variable*, ID : steps to the member with the given ID (signed long) of the given starting variable

19 [step to index], *starting_variable*, *index* : steps into the given index of the starting variable

20 [step to indices], *starting_variable*, *low_index*, *high_index* : steps into the given indices of the starting variable

21 [step to all indices], *starting_variable* : steps into all indices of the starting variable

22 [resize], *variable*, *top_index* : resizes the variable's member to have the given number of indices, and steps into these indices

23 [insert indices below], *variable*, *new_low_index*, *new_high_index* : adds the new range of indices to the member above the insertion point

24 [insert indices above], *variable*, *new_low_index*, *new_high_index* : adds the new range of indices to the member with old indices just below the insertion point

25 [delete], *variable* : deletes the part of *variable* that was stepped into

27 [this] : returns the function variable that is currently running

28 [that] : returns the variable on the left-hand side of the equate statement

29 [nothing] : returns no variable

30 [if equal], *expr1*, *expr2* : returns true if the two expressions' data are equal; false otherwise

31 [if not equal], *expr1*, *expr2* : returns false if the two expressions' data are equal and true otherwise

32 [if greater-than], *num1*, *num2* : returns true if and only if *num1* is greater than *num2*

33 [if greater-than-or-equal], *num1*, *num2* : returns true if and only if *num1* is greater than or equal to *num2*

34 [if less-than], *num1*, *num2* : returns true if and only if *num1* is less than *num2*

35 [if less-than-or-equal], *num1*, *num2* : returns true if and only if *num1* is less than or equal to *num2*

36 [and], *cond1*, *cond2* : returns true if and only if both conditions are true (both are always evaluated)

37 [or], *cond1*, *cond2* : returns true if and only if one or both of the conditions are true (both are always evaluated)

38 [xor], *cond1*, *cond2* : returns true if and only if one, but not both, conditions are true

39 [not], *cond1* : returns true if the condition is false and false if the condition is true

40 [if same reference], *expr1*, *expr2* : returns true if the two members point to the same data; false otherwise

41 [if different reference], *expr1*, *expr2* : returns false if the two members point to the same data and true otherwise

42 [add], *num1*, *num2* : returns the sum of its numeric arguments

43 [subtract], *num1*, *num2* : returns *num1* minus *num2*

44 [multiply], *num1*, *num2* : returns the product of its numeric arguments

45 [divide], *num1*, *num2* : returns *num1* divided by *num2*

46 [power], *num1*, *num2* : returns *num1* raised to the power *num2*

47 [modulo], *num1*, *num2* : returns the remainder of *num1* divided by *num2* after they have been truncated to integers

50 [constant slong], num : returns the given number, read as a signed long

51 [constant double], num : returns the given number, read as a double-precision floating point number

52 [constant string], characters_num, string_data : returns an inlined string having the given number of characters. The string_data field occupies one long word for every four characters.

53 [code block] : returns the inlined code beginning with the next bytecode sentence and ending with a null sentence

54 [ubyte] : signals an unsigned byte primitive type

55 [sbyte] : signals a signed byte primitive type

56 [ushort] : signals an unsigned short integer primitive type

57 [sshort] : signals a signed short integer primitive type

58 [ulong] : signals an unsigned long integer primitive type

59 [slong] : signals a signed long integer primitive type

60 [single] : signals a single-precision primitive type

61 [double] : signals a double-precision primitive type

62 [block], byte_length : signals a block type with the specified number of bytes

63 [string] : signals a string data type

65 [code] : delineates the boundary between two code blocks

5.3 Def-general operators and flags

Below are the common def-general operators and their associated flags. The 'flags' column is the resulting number in the ID field of the def-general operator – it is a decimal representation of the binary digits in the following columns, where the presence of a flag sets some bit to 1. The flags are: equate (0), update-members (1), add-members (2), new-target (3), relink-target (4), run-constructor (5), hidden (6), unjammable (7), and proxy (8). The last six operators that have asterisks in their names cannot be written into a script, but are generated automatically by the compiler in various situations.

name	abbr	symbol	flags	8	7	6	5	4	3	2	1	0
define	def	`::`	46				✓		✓	✓	✓	
member-define	mdf	`*::`	6							✓	✓	
variable-define	vdf	`@::`	44				✓		✓	✓		
equate	equ	`=`	1									✓
define-equate	deq	`:=`	47				✓		✓	✓	✓	✓
equate-at	eqa	`=@`	16					✓				
define-equate-at	dqa	`:=@`	22					✓		✓	✓	
proxy define	pxdef	`#::`	302	✓			✓		✓	✓	✓	
∼define	def*	N/A	172		✓		✓		✓	✓		
∼define-equate	deq*	N/A	173		✓		✓		✓	✓		✓
∼define-equate-at	dqa*	N/A	148		✓			✓		✓		
∼define	def**	N/A	236		✓	✓	✓		✓	✓		
∼define-equate	deq**	N/A	237		✓	✓	✓		✓	✓		✓
∼define	def-c**	N/A	204		✓	✓			✓	✓		

5.4 Yazoo registers

All objects within Yazoo are ultimately derived from an overarching variable that we will call Zero, although that word has no meaning within Yazoo. Zero encompasses two composite variables. The first variable contains the ten so-called registers that are described in this section. The second variable within Zero is the workspace, where the lowest-level script (e.g. `start.zoo`) begins running. It turns out to be impossible for any script to find a register in the normal way (the reason being that no member or search path leads out of the workspace). This is a deliberate roadblock: it prevents these important variables from being deleted or modified, which would cause problems for built-in functions, such as `log()` and `input()`. Instead, Yazoo provides a bytecode operator that gives access to the registers directly. From the perspective of a script, registers have names just like ordinary variables: `R_slong`, `R_composite`, etc.

Registers are used by the built-in Yazoo functions to return information to the user. In general, registers can be treated in the same way as any other variable, except that they may not be redefined or deleted. The following code is perfectly legitimate:

```
R_slong = that+1
R_double = R_slong*3.14
print(R_double)
```

However, using a register as a working variable is not recommended, since the register will be overwritten the next time some built-in function uses it for its return variable. It is generally a much better idea to copy any data one needs from the register immediately into a user-defined variable and work with it there.

ID	name	type	ID	name	type
0	R_slong	slong	4	R_error_code	ulong
1	R_double	double	5	R_error_index	ulong
2	R_string	string	6	R_error_script	composite
3	R_composite	composite	7	R_warning_code	ulong
			8	R_warning_index	ulong
			9	R_warning_script	composite

Table 3: Registers, by ID number

The ten registers are listed in Table 3, along with their types and ID numbers. The ID numbers appear only in compiled bytecode. The types are ordinary variable types, because Yazoo's 'registers' are ordinary variables — they are *not* the real registers on the chip that assembly-language programmers deal with!

5.4.1 Numeric and string registers

numeric: `R_slong`, `R_double`
string: `R_string`

These three registers are predefined variables of types corresponding to their namesakes. They are used for returning values from the built-in Yazoo functions, which are described in the next section.

The following built-in functions return their arguments to `R_slong`: `top()`, `size()`, `find()`, `variable_type()`, `variable_member_top()`, `variable_code_top()`,, `member_type()`, `member_code_top()`, `member_indices()`, `member_ID()`, `if_member_targeted()`, `if_hidden_member()`, `variable_code_ID()`, `variable_code_offset()`, `member_code_ID()` and `member_code_offset()`.

`R_double` is used by the following built-in functions: `abs()`, `round_down()`, `round_up()`, `log()`, `cos()`, `sin()`, `tan()`, `acos()`, `asin()`, `atan()`, `random()`, and `call()`.

The built-in functions that return values to `R_string` are: `load()`, `input()`, `compile()`, `extract()`, `YZ_get_variable_code()`, and `YZ_get_member_code()`. Note that `compile()` sets `R_string` but does not return `R_string`; the user must explicitly refer to `R_string` in order to access the compiled bytecode.

In practice (and with the exception of `compile()`), the user only rarely needs to deal explicitly with these primitive registers. For example, the code:

```
round_down(5.3)
result := R_double
```

is entirely equivalent to

```
result := round_down(5.3).
```

The second case is also better practice since it avoids storing the results in the registers any longer than is necessary. A primitive register is analogous to an explicit return variable within a user-defined function; it only stores the return value temporarily before handing it off to the calling function, and it can be completely invisible to the outside user.

5.4.2 `R_composite`

Not surprisingly, the `R_composite` register is a composite variable; i.e. it contains members. Thus

```
R_composite.q[10] :: double
```

is quite legal.

`R_composite` is used by the Yazoo function `transform()`. When `transform()` processes a string of bytecode, that bytecode becomes `R_composite`'s 'type' (if `R_composite` is thought of a variable), or function code (if `R_composite` is viewed as a function). Note that although `transform()` alters `R_composite`, this register is not its return variable.

In interactive mode, `start.zoo` alters the code – though not the data – of `R_composite` in the course of its normal activity; however, it saves and restores the type/code of this register between each command the user types. Thus `R_composite` is stable from the user's perspective, even in interactive mode.

5.4.3 Error and warning registers

numeric: `R_error_code`, `R_error_index`, `R_warning_code`, `R_warning_index`
composite: `R_error_script`, `R_warning_script`

The error and warning registers are modified by three built-in Yazoo functions. Compile-time errors are returned by `compile()`. Certain pathologies in bytecode are returned by `transform()`. Runtime errors and warnings may be caught using `trap()`. All three of these functions use `R_error_code` as the return variable.

Every error is assigned a numeric code which is stored in `R_error_code`. The names and descriptions of the error codes are described later in this document. An error code of zero always means no error.

The value of `R_error_index` defines the location of an error. `compile()`-generated errors return the character position at which the error was flagged in the source code, beginning at '1'. On the other hand, errors generated by `transform()` and `trap()` give the position of the error as an index, again beginning at '1', of an array of long-integer words containing the bytecode. Thus an error index of '14' represents an error flag either at the 14th character of uncompiled source code (in the case of `compile()`), or else at the 14th word of bytecode (bytes 53-56 if the computer uses 4-byte long words).

When `trap()` catches a runtime error, the offending bytecode is transformed into the type/code of the composite error register `R_error_script`. This allows the user to back out the problem by using `variable_code(R_error_script)` to return this bytecode as a string. (This strategy is used by the `disassemble()` routine in `start.zoo` to flag bytecode errors directly when the text of the original script is not available.) The two-step process of trapping the error and then copying the script to a string may seem inconvenient; it is done this way because it is internally faster to do this than to copy every error-script into a string, when in many cases only a minority of errors are actually examined by the user.

A warning differs from an error in that warnings do not stop execution. Warnings are treated exactly analogously to errors, and `R_warning_code`, `R_warning_index` and `R_warning_script` work in almost exactly the same way as their `R_error_` counterparts. The main difference is that warnings can only be detected using `trap()`, as neither `compile()` nor `transform()` will alter the warning registers. (`transform()` actually *can* generate warnings, but in order to catch them `transform()` must be enclosed within a `trap()` statement). `R_warning_code` is not used as the return variable for any Yazoo function.

5.5 Yazoo functions

Yazoo provides a number of built-in C-coded functions, which can be invoked directly by name (in contrast to user-defined C functions which require a `call()` invocation). Some of these built-in routines perform various common and handy chores, like calculating logarithms. Others allow a script to read from and even alter some of Yazoo's private records, enabling scripts to do things like detect hidden members or obtain the compiled bytecodes of any function.

The built-in functions work hand-in-hand with the registers (described in the previous section). Any built-in function that returns a value uses some register for storing that value. For example, `abs()` returns a type-`double` variable. It is a built-in C-coded routine, but it behaves as if it had been written:

```
abs :: {
    code

    R_double = ...
```

ID	name	@	ID	name	@	ID	name	@
0	call		14	abs	✓	27	variable_type	✓
1	compile		15	round_down	✓	28	variable_member_top	✓
2	transform		16	round_up	✓	29	variable_code_top	✓
3	load		17	log	✓	30	member_type	✓
4	save		18	cos	✓	31	member_code_top	✓
5	input		19	sin	✓	32	member_indices	✓
6	print		20	tan	✓	33	member_ID	✓
7	read_string		21	acos	✓	34	if_member_targeted	✓
8	print_string		22	asin	✓	35	if_hidden_member	✓
9	clip		23	atan	✓	36	variable_code	✓
10	trap		24	random	✓	37	member_code	✓
11	throw		25	extract	✓	38	variable_code_ID	✓
12	top	✓	26	find	✓	39	member_code_ID	✓
13	size	✓				40	variable_code_offset	✓
						41	member_code_offset	✓
						42	GetParentScriptInfo	
						43	SpringCleaning	

Table 4: Built-in functions, by number. Functions which generate tokens have a checkmark in the '@' columns

```
        return R_double
    }
```

That is, if we write

```
y = abs(x)
```

then the result is copied to `y`, from the register `R_double` where `abs()` first stored it. Other routines use different registers, and not all registers are so-called return registers: `call()` writes to six different registers, only one of which can be the return variable. It is generally best to copy data out of the registers as quickly as possible, since they are global variables that can be quickly overwritten with another function call.

The compiler translates each built-in function call into: the built-in-function operator, followed by the ID number of the given function, and at the end the argument variable. The ID numbers of the built-in functions are listed in Table 4. Some of the functions will create a token if they are not explicitly assigned to a variable; these are denoted by checkmarks in the '@' column. Each function is discussed in turn below.

`abs()`

syntax: (numeric) y = `abs`((numeric) x)

Returns the absolute value of its argument (which must be numeric). `R_double` is the return variable for this function.

`acos()`

syntax: (numeric) y = `acos`((numeric) x)

Returns the inverse cosine of its argument. The argument must be a number on the interval [-1, 1] (a number outside this range will generate the 'not a number' value on many machines). The result is on

the interval [0, π]. `R_double` is the return variable for this function.

`asin()`

syntax: (numeric) y = `asin`((numeric) x)

Returns the inverse sine of its argument. The argument must be a number on the interval [-1, 1] (a number outside this range will generate the 'not a number' value on many platforms). The result is on the interval [$-\pi/2$, $\pi/2$]. `R_double` is the return variable for this function.

`atan()`

syntax: (numeric) y = `atan`((numeric) x)

Returns the inverse tangent of the argument, which must be numeric. The result is an angle in radians on the interval [$-\pi/2$, $\pi/2$]. `R_double` is the return variable for this function.

`call()`

syntax: (numeric) *return_code* = `call`((string/numeric) *C_routine*, [*arguments*])

Runs a C or C++ routine from within a Yazoo script. The routine is specified by name or by number in the first argument. The subsequent arguments form the `argv` array that the C routine receives. The return value of the routine is stored in `R_double` which is the return register of the `call()` function.

User-written C or C++ functions make their introductions to Yazoo through the `UserFunctionSet[]` array, which is in the userfn.c or userfn.cpp source file. This array consists of a list of Yazoo names (strings) paired with the addresses of C/C++ functions that they call. (The only functions that a script will be able to call are those that had an entry in the `UserFunctionSet[]` array when Yazoo was compiled, so adding new C/C++ routines always necessitates a rebuild.) To call a C/C++ function from a script, the user needs to specify either its Yazoo name (the string in the `UserFunctionSet[]` entry) *or* its index within the `UserFunctionSet[]` array. The index method follows the Yazoo convention that the first element begins at 1, even though it is an element of a C array. The only advantage of calling by number rather than name is that it can be slightly faster, since it saves Yazoo the trouble of searching its names list.

As described in Chapter 2, an embedded C or C++ function must be written in the following style:

```
int my_function(int argc, char **argv)
```

just as if it were a complete program. The number of arguments is passed through `argc`, and the address of the array of pointers to the actual arguments is located at `argv`. So `*argv` is the pointer to the first argument, and `**argv` is the first byte of the first argument; `*(argv+1)` is the pointer to the second argument; etc. Only primitive variables can compose an argument to the C routine; a composite argument to `call()` generally contains, and is passed as, multiple primitive arguments, one for each primitive component. Strings are passed as linked lists (see the reference section), and `call()` defragments these lists before running the embedded C code. Void arguments or members are skipped.

`call()` also creates one argument of its own that can be used for type-checking: an 'argument-info' array, whose pointer it tucks into `*(argv+argc)`. This is a list of elements of type `arg_info`, which is defined in userfn.h. There is one entry for each argument (not including itself), and each entry looks like:

```
typedef struct {
    Ulong arg_type;
```

```
    Ulong arg_size;
    Ulong arg_indices;
} arg_info;
```

The `arg_type` codes are defined in Table 1 on page 18. The `arg_size` field gives the size in bytes of each element (in case the type is a block), and `arg_indices` gives the number of indices that were passed (1 if it was just a variable, N if an array).

All arguments are passed by reference: they are pointers to Yazoo's own storage for that data, so data can be exported as well as imported. It is easy to crash Yazoo by overwriting the wrong regions of memory. One of the very first things to check if there is a crash is whether a `call()` routine was run, since that is very often the culprit. Beware that this crash often happens far downstream, long after the `call()` had finished.

The return value of the C/C++ function is stored in `R_double` (even though that is an integer), which is the return variable of `call()`. So if we write:

```
y = call("Factorial", x)
```

then whatever value the C-coded `Factorial` function, or whatever it is called, returns will be copied into the script variable `y`.

`clip()`

syntax: `clip`((functions) *f1*, *f2*, ...)

When a variable is requested *by name*, Yazoo first looks for a member of that name in the current function. Failing that, it works backwards up the current search path and looks for the member in each parent function. The `clip()` routine cuts the search paths that cross the arguments to `clip` (which are usually functions or function-containing classes), preventing any code inside these functions from accessing members that lie outside. (A void argument causes the currently-running function to be clipped). This is the only way to encapsulate code in Yazoo. `clip()` does not return any value.

The following example illustrates the use of `clip`.

```
var1 := 1

clip_demo :: {
    var2 := 2
    code

    print("Before clipping: ", var2, ", ", var1, "\n")
    |  clip(this)   (doesn't work)
    clip(clip_demo)       | this is one way to do it
    clip(*)         | this is another
    print("After clipping: ", var2, ", ")    | so far so good...
    print(var1, "\n")     | this line will cause an error
}

clip_demo()
```

When `clip_demo()` is run, it crashes on the last line since, after clipping, it is no longer able to see the global variable `var1`. As demonstrated, `clip(*)` simply clips the current function. Surprisingly, `clip(this)`, which might be expected to do the same, does not work, because `this` is the argument space of `clip()`, not the function space of `clip_demo()`.

Note that `clip(f1)` clips *all* fibrils (search paths) that pass from the function `f1` through the program counter's fibril, *at* the location of `f1`. On the other hand, `clip(*)` clips only the program counter's fibril.

A clipped fibril is one that has no stem; see the section on search paths and fibrils.

`compile()`

syntax: (numeric) *error_code* = `compile`((string) *source* [, (string) *var_names* [, (string) *line_positions*]])

Before Yazoo can interpret a script, the script must be pseudo-compiled into a binary form that is much easier to execute than the raw text. The built-in `compile()` function accepts a Yazoo text script as a string and returns its bytecode into the `R_string` register. `compile()` emphatically does *not* generate executable machine code; the bytecode is something only Yazoo will understand. (Likewise the "disassemble()" routine in start.zoo parses bytecode, not real assembly code.)

To generate a stand-alone script, it is sufficient to invoke `compile()` with a single argument, as in the following example.

```
source_code := "a := 1, b := 2, print(a+b)"

err := compile(source_code)

if err == 0
    translated_code := R_string
    save("add.hob", translated_code)
else
    print("Error ", err, " at character ", R_error_index)
endif
```

Notice that `compile()` does not return the bytecode, but rather the error code returned by the compiler, which is zero upon a successful compilation. The user must fetch the bytecode manually from the `R_string` register, and it is usually a good idea to do this immediately after the compiler finishes since other subsequent commands may overwrite `R_string`.

The built-in compiler only returns one error at a time. The error code is stored in `R_error_code`, and as we see this register is also the return variable. The register `R_error_index` stores the character number (not the line number), starting from 1, at which the compiler flagged the error.

Two scripts that are compiled independently as above will generally not be compatible with each other, because the names of all variables and functions get replaced with numbers during compilation; since the namespaces are separate there isn't likely to be much correspondence between the member IDs of the two scripts. If the two scripts are to be run in the same space they must be compiled using the same namespace. The user can control the namespace by providing it as a second string (argument number 2) to `compile()`; the compiler will then update the namespace string in addition to writing bytecode in `R_string`.

The first N bytes of the namespace string (where N equals `size(ulong)`), when viewed as an unsigned long integer, give the number of hidden members that Yazoo has defined so far. These hidden members are for tokens and do not correspond to names that the user generates; they have negative ID numbers. Starting from byte $N + 1$ in the names-table string, the user-defined variables are stored as null-terminated strings in the order that they were discovered by the compiler. So if the variables `x`, `y`, `result` were encountered (in that order) along with two tokens, the names string will read

```
\00\00\00\02x\00y\00result\00
```

The following procedure properly initializes the names table:

```
(InitialNullNamesNum::ulong) = 0
(NamesTable :: string) =! InitialNullNamesNum
...
```

```
err := compile(source_code, NamesTable)
```

Any scripts subsequently compiled with this name table will be compatible with each other. They will most likely *not* be compatible with the script that compiled them, unless that script has access to its own name table. In fact, the command-prompt user does have access to his own name table, in the variable `AllNames`. Be warned that if the `AllNames` variable gets improperly overwritten the user should expect immediate and irreversible dysfunction for the rest of his Yazoo session.

The optional third argument to `compile()` gives the location in the compiled bytecode of each line break in the original text script. (The user must then provide a names table as well; otherwise the line-breaks argument will be mistaken for the names table.) Each line in Yazoo script translates into roughly one line of bytecode, so by using the line-break table a runtime error, whose location is necessarily given as an index in the bytecode, can be traced back to a specific line in the original script. By storing both the original human-readable text and the table of line breaks that `compile()` returns, it is possible to flag runtime errors next to the uncompiled text where the error occurred.

The line-breaks string is composed of a number of place-markers, and each place-marker consists of two unsigned long integers. The first integer in each place-marker is the position of some character in the uncompiled text, beginning at 1 for the first character in the script. The second integer is the corresponding index in the compiled bytecode, where index 1 is the first word in the bytecode. Whereas indices in the text enumerate single-byte characters, each word in the bytecode is the length of a long integer: traditionally 4 or 8 bytes.

While `compile()` is certainly the easiest way to generate bytecode, it is not the only way, and in fact some bytecode cannot be generated at all using `compile()`. In this case, certainly the user can do the compilation himself, but this is laborious. Unless you are doing something wild, the easiest way to generate non-scriptable code is to compile the next-closest thing that can be scripted using `compile()` and then tweak the resulting output string as needed (see `do_in()`).

In a sense `compile()` is 'just' a string operation: it takes in one string and generates a second string that is based in some complicated way upon the first. As far as Yazoo is concerned the second string is treated just like any other string: it is not in any way executable. In order to actually run the compiled script one must subsequently incorporate the output string into Yazoo's internal code bank, using the built-in routine `transform()`.

`cos()`

syntax: (numeric) y = `cos`((numeric) x)

Returns the cosine of its argument. The argument must be numeric. The return variable is `R_double`.

`extract()`

syntax: (string) *sub_str* = `extract`((string) *str*, (numeric) *first_char*, *last_char*)

Returns a piece of the string given in the first argument, starting from the character position specified by the second argument and ending with the character position specified by the third argument. The following two scripts are equivalent:

```
part_str := extract(full_str, 5, 10)
```

and

```
all_chars[*] :: ubyte
```

```
part_string :: string

all_chars[*] =! full_str
part_str =! all_chars[5, 10]
```

Both the second and the third arguments, which give character positions (starting from 1), must be positive and less than or equal to the number of characters in the original string, or else an error will be generated. The range includes the two endpoints, so if the third argument equals the second then one character will be returned, etc. The return variable is `R_string`.

`find()`

syntax: (numeric) *result* = `find`((strings) *search_in*, *search_for* [, (numeric) *mode* [, (numeric) *starting_position*]])

Finds an instance of, or counts the number of instances of, a substring (argument 2) within another string (argument 1). If `find()` is used in search mode, it returns the character position (where 1 denotes the first character) where the substring was first found, and 0 if it was not found anywhere. If `find()` is run in count mode, it returns the number of instances of the substring found within the larger string. The return register is `R_slong`.

The third argument controls the mode that `find()` is run in: it needs to be -1, 0 or 1. If a mode is not specified then it defaults to mode 1, which denotes a forward search; i.e. it will return the first instance of the substring that it finds. Mode -1 corresponds to a reverse search, which will find the last instance of the substring. Mode 0 is the count mode.

By default, a forward search begins from the first character, and a reverse search begins with the last character. A count proceeds forward from the first character. The starting character can be changed by specifying a starting position in the fourth argument. A mode has to be given in order for a starting position to be specified.

If `find()` is used to count the number of instances of the substring, it counts only non-overlapping instances. Thus `find("ooo", "oo")` returns 1 even though the substring occurs starting at both positions 1 and 2. In this case the routine finds the substring at position 1, then jumps to the end of that instance—i.e. to position 3—before it resumes searching.

`GetParentScriptInfo()`

syntax: `GetParentScriptInfo`((function) *f*, (numeric) *code_index*, (numeric) *code_ID* [, (numeric) *code_offset*])

Yazoo maintains a global internal registry of all active scripts that have been loaded (see `transform()`). For example, `start.zoo`, `user.zoo`, the scripts that have been run using the `run()` function, and entries from the command prompt are all stored as separate scripts. For some purposes (notably error flagging) it is important to know the internal ID number of a script, which is what `GetParentScriptInfo()` provides. The first two arguments specify a code of some function to query (the code index is usually just 1 unless the function was defined with the inheritance operator). `GetParentScriptInfo()` stores the global ID of that code's enclosing script in the third argument, and the location of the function's bytecode relative to the start of the compiled script (after compilation, measured in long words) into the optional fourth argument.

The following illustrates the distinction between a function, or a function's code, and the script that encapsulates it.

```
f1 :: {
    code
    a = b+c    | f1 has only one code index
}
```

```
f2 :: { ... } : { ... }    | f2 has two code indices
```

If the above text was loaded from a single file using `run()`, it will have been loaded into memory with a single invocation of `transform()` and hence the codes of both `f1` and `f2` will be contained within the same script. However, the two codes of `f2` will have a larger offset from the beginning of the script than the single code of `f1`, since `f1` was defined before `f2`.

`if_hidden_member()`

syntax: (numeric) *if_hidden* = `if_hidden_member`((composite variable) *var*, (numeric) *member_number*)

Returns 'true' if the given member is hidden, and 'false' if it not. Unless the user is writing his own compiled bytecode, the members he defines are not hidden. Hidden members correspond to compiler-generated tokens. The member is specified by its enclosing variable in argument 1, and its member *number* in argument 2 (*not* an index!). The return variable is `R_slong`.

`if_member_targeted()`

syntax: (numeric) *if_not_void* = `if_member_targeted`((composite variable) *var*, (numeric) *member_number*)

Returns 'true' if the given member has a target, and 'false' if it has no target (i.e. it points at the void). The member is specified by its enclosing variable (argument 1), and its member *number* (argument 2) – which is *not* the same as an index. The return variable is `R_slong`.

`input()`

syntax: (string) *str* = `input`()

Reads in a single line from the C standard input (which is usually the keyboard). `input()` causes Yazoo's execution to halt until an end-of-line character is read (i.e. the user hits return or enter). The string of characters before, but not including, the end-of-line, is loaded into `R_string` and returned to the enclosing expression, and script execution resumes. A null character causes the error "I/O error" to be thrown.

`load()`

syntax: (string) *file_string* = `load`((string) *file_name*)

Reads a file into a string. `R_string` is used as the return variable because the storage space is a-priori unknown; however, there is no requirement that the data be ASCII-encoded. The file name must include a path if the file is not in the default directory, as in "`/Users/bob/Desktop/MyFile.txt`". If there is an error in opening or reading the file (i.e. if the file was not found or there was a permissions problem), then `load()` returns "I/O error", signifying that the error comes from the operating system, not Yazoo. The counterpart to `load()` is `save()`.

`load()` searches only in the default directory. The user.zoo routine `Load()` extends the built-in `load()` by searching all paths specified in the `DirectoryPaths[]` array. (The `run()` function in user.zoo also searches all `DirectoryPaths[]`.)

`log()`

syntax: (numeric) y = `log`((numeric) x)

Returns the natural logarithm (base e) of its argument. The argument must be numeric. The logarithm is only defined for arguments greater than zero. The return variable is `R_double`.

`member_code()`

syntax: (string) *code_str* = `member_code`((variable) *var*, (numeric) *member_number*, *code_index*)

Returns the compiled bytecode of a given member into `R_string`. This, along with `variable_code()`, is an inverse operation to `transform()`.

In a sense, `member_code()` simply reconstructs the original output of the `compile()` function that generated its script. The only potential difference is that `member_code()` only returns the part of the script that was used to define the member in question. For example, we may have compiled and transformed a script that, among other things, defined a function.

```
compile( "var1 :: ulong, f1 :: { code, ...  }" )
transform(R_string)

big_function :: R_composite
sub_function :: big_function.f1
```

(Realize that although `transform()` suppresses the constructor when it is first introduced into `R_composite`, the constructor *does* run when variables are defined in the ordinary way using `R_composite` as a template.) Now, we can inspect our two members' codes:

```
big_code := member_code(big_function, 1, 1)
little_code := member_code(little_function, 1, 1)
```

The code defining the `big_code` member (and also the variable) is the full script: everything between `compile()`'s quotation marks. The code defining both the `little_code` member and its variable is just the code within (and excluding) the braces defining `f1`.

`variable_code()` is the complement to `member_code`; it returns the codes of the functions themselves, as opposed to the member's code. Since a member's code is never actually executed, it is better understood as the type restriction on the variables that member is allowed to target.

`member_code_ID()`

syntax: (string) *the_code_ID* = `member_code_ID`((variable) *var*, (numeric) *member_number*, *code_index*)

Returns the global code ID of a given code of a given member, via the return variable `R_slong`. This ID was assigned by `transform()` when the code was transformed. Note that code ID numbers can be recycled if the original code is no longer used and was cleaned out of memory.

For reasons that were explained in the last subsection on `member_code()`, the code ID of a compiled script is identical to that of the codes of all members that the script defined (as their codes are subsumed within the larger script). To differentiate between the script and all members, one has to look at both the code ID and the code offset (see `member_code_offset()`).

`member_code_offset()`

syntax: (string) *the_code_offset* = `member_code_offset`((variable) *var*, (numeric) *member_number*, *code_index*)

Returns the code offset of a given code of a given member, using the return variable `R_slong`. The code offset is measured relative to the start of the script in which it the code was loaded. See the explanation for `member_code()` for an example of this. The offset is the number of long-integer words the member's bytecode is inset from the beginning of the full script that it was defined in; if the code *is* this entire script then the offset is zero.

`member_code_top()`

syntax: (numeric) *member_codes_num* = `member_code_top`((composite variable) *var*, (numeric) *member_number*)

Returns the number of codes in a member definition. The member is given by the member number (2nd argument) of the given variable (1st argument). Note that the member *number* is different from an index. Primitive members or untyped members have no codes, so they return a zero. The member may be void. The return variable is `R_slong`.

`member_ID()`

syntax: (numeric) *mbr_ID* = `member_ID`((composite variable) *var*, (numeric) *member_number*)

Returns the ID number of a given member of a composite variable. The ID is essentially the member's name; when a script is compiled all names get converted to numbers. Under normal conditions user-defined names are assigned positive ID numbers, whereas hidden members are given unique negative ID numbers. The variable enclosing the member is the first argument, and the member number (not index!) is the second argument. The return variable is `R_slong`.

`member_indices()`

syntax: (numeric) *num_indices* = `member_indices`((composite variable) *var*, (numeric) *member_number*)

Returns the number of indices spanned by a given member. The member is specified by its host composite variable (the first argument) and a member number (the second argument); hidden members are allowed. The return variable is `R_slong`.

`member_type()`

syntax: (numeric) *mbr_type* = `member_type`((composite variable) *var*, (numeric) *member_number*)

Returns the type restriction of a given member of a composite variable. The variable is the first argument, and the member number is the second argument. The member number is not an index; for example an array may have one member with many indices. Hidden members are included. The types IDs are listed in Table 1 on page 18. A composite-typed member only returns a '10' (composite) even though its full type is properly determined by its code list. The return variable is `R_slong`.

`print()`

syntax: `print`((vars) v1, v2, ...)

Writes data to the standard output (which is normally the command prompt window). The arguments are printed sequentially and without spaces in between. Numeric arguments are converted to ASCII and printed as legible integers or floating-point numbers. Both string- and block-typed arguments are written verbatim (byte-for-byte) to the screen, except that for *string* types only, unprintable characters are replaced by their hexadecimal equivalents "`\AA`" (which is also the format in which these characters may be written into a string). Also, carriage returns in strings are written as end-of-line characters, so a PC-style line ending marked by "`\0D\n`" outputs as a double line-break.

`print_string()`

syntax: `print_string`([(numeric) precision,] (string) to_write, (vars) v1, v2, ...)

Writes data to a text string. `print_string()` is the counterpart to `read_string()`. Roughly speaking, `print_string()` is to `print()` as C's more elaborate `sprintf()` is to `printf()`. The string to write is followed by any number of variables whose data Yazoo writes to the string (with no spaces in between). Importantly, numeric variables are written as text, so `print_string` is different from a forced equate. For example:

```
print_string(str, 5, 2.7)
```

sets `str` to "52.7", whereas

```
str =! { 5, 2.7 }
```

gives something illegible (the raw bytes encoding the two numbers in binary format).

Strings from the source variables get copied into the destination string verbatim. Block-typed variables (and only block variables) are copied in binary form, as in a forced equate.

If the first argument is numeric, then it is taken as the output precision for floating-point `single` and `double` variables; then the output string must follow as the second argument. Otherwise the output precision is determined by the C constants `FLT_DIG` and `DBL_DIG` for `single` and `double` variables, respectively. Thus, when no precision is specified, `print_string` prints considerably more digits than does `print()`.

`random()`

syntax: (numeric) $y =$ `random`()

Returns a pseudo-random number uniformly drawn on the interval [0, 1]. To obtain the random number to double-precision, Yazoo uses C's `rand()` function in the following formula:

$$\texttt{random()} = \texttt{rand()}/\texttt{RAND\_MAX} + \texttt{rand()}/(\texttt{RAND\_MAX})^2$$

The random number generator is initialized by Yazoo to the current clock time each time the program is run, so the generated sequence should not be repeatable. The return variable is `R_double`.

`read_string()`

syntax: `read_string`((string) to_write, (vars) v1, v2, ...)

Reads data from an ASCII string into variables. The first argument is the string to read from; following arguments give the variables that will store the data. `read_string()` is the humble cousin to C's `sscanf()` routine (it does not take a format string). The various fields within the string must be separated by white space or end-of-line characters.

`read_string()` converts ASCII data in the source string into the binary format of Yazoo's memory. Thus numeric fields in the source string need to be written out as text, as in "3.14" rather than its unintelligible floating-point representation. String fields must be one English (as opposed to integer) word long, so "the quick brown" will be read into three string variables, not one. `block` variables are simply copied byte-for-byte from the string. Composite variables are decomposed into their primitive components, which are read sequentially from the source string. Void members are skipped.

Here is an example of the use of `read_string()`

```
date :: { month :: string, day :: year :: ushort }
activity :: string
read_string("Jan 5 2007   meeting", date, activity)
```

If the string cannot be read into the given variables (i.e. there are too many or too few variables to read), Yazoo throws a type-mismatch warning. Warnings can also be thrown if Yazoo cannot read a field that should be numeric, or if there is an overflow in a numeric field.

`read_string()` is a counterpart to `print_string()`. However, `print_string()` does not write spaces in between the fields, so unless spaces are put in explicitly its output cannot be read directly by `read_string()`.

`round_down()`

syntax: (numeric) y = `round_down`((numeric) x)

Returns the nearest integer that is as low as or lower than the (numeric) argument. This is equivalent to the `floor()` function in C. For example, `round_down(2.3)` returns 2, `round_down(-2.3)` returns -3, and `round_down(-4)` returns -4. The return variable is `R_double`.

`round_up()`

syntax: (numeric) y = `round_up`((numeric) x)

Returns the nearest integer that is as high as or higher than the argument, which must be numeric. `round_up()` is equivalent to the `ceil()` function in C. For example, `round_up(5.6)` returns 6, `round_up(-5.6)` returns -5, and `round_up(2)` returns 2. The return variable is `R_double`.

`save()`

syntax: `save`((strings) *file_name*, *data_to_write*)

Saves the data from the second argument into the file specified in the first argument. There is no return value, although the error "I/O error" will be thrown if the save is unsuccessful. (An error would likely indicate a bad pathname, disk full, or that we don't have write permissions for that file or directory). If the

directory is not explicitly written before the file name, as in "`/Library/my_file`", then the file is saved in the default directory, which is probably the directory where Yazoo resides.

There is no need for the data to be encoded in ASCII format, even though it gets passed to `save()` as a string. Online conversion to the proper string type can be done in the following way:

```
save("my_data", (temp_str :: string) =! the_data)
```

where `the_data` may be a variable or array or any other object. `save()` writes the data verbatim; if the data is ASCII text, then a text file will be produced; otherwise the output should be considered a binary file. The saved data can be read back into a string by the `load()` function.

`sin()`

syntax: (numeric) y = `sin`((numeric) x)

Returns the sine of its argument. The argument must be numeric. The return variable is `R_double`.

`size()`

syntax: (numeric) *var_size* = `size`((var) *my_var*)

Returns the size, in bytes, of the argument variable. For composite variables, this is the sum of the sizes of all its members. If two members of a composite variable point to the same data (i.e. one is an alias of the other), then that data will indeed be double-counted. Therefore the number that is returned gives the number of bytes that will participate in, for example, a forced-equate or `save()`, which may be more than the number of bytes of actual storage. If a member points back to the composite variable, as in

```
a :: {
    self := @this
    data :: ulong    }

size(a)   | will cause an error
```

then the size of `a`, including its members and its members' members, etc., is effectively infinite, and Yazoo throws a self-reference error.

`SpringCleaning()`

syntax: `SpringCleaning`()

This function removes all unused objects from Yazoo's memory. An object is termed 'unused' if it cannot be accessed by the program counter (PC) or any location on the PC stack (the list of currently running functions). Removing these frees up system memory that might otherwise slowly fill with disused debris.

Internally, Yazoo keeps track of which data structures are being used by assigning them 'references': each reference marks a use of that structure by somewhere else in memory. For example, a running function might be referenced twice: once by a member that points to it, and once by the PC acknowledging that the function's code is currently being used; in turn that function references the internally-stored code that is being run. When an object's references reach zero, there is no way to ever access it again—no members or fibrils point to it, since otherwise the object would be referenced—so Yazoo frees its memory.

Although much memory is freed up automatically, unfortunately there are some things that the referencing apparatus will miss. The reason is that two objects can simultaneously become unhooked from the main memory tree that still reference each other: both are now garbage, but they both have at least one reference. For example, variable `a` might have a member pointing to variable `b`, while a member of `b` points back to `a`. This particular case might sound contrived, but for various reasons all sorts of self-referencing, self-sustaining symbioses form during normal execution.

There is no way to identify and exterminate once and for all these living dead without combing through the whole memory tree, which is what `SpringCleaning()` does. It first explores every item in memory that could ever be accessed by the program counter; then it scours the entire memory tree and removes anything that was not reached in the first pass. This not only frees up memory but probably speeds up program execution for processes like memory allocation where Yazoo has to search for free slots within its reference lists.

When Yazoo is run in interactive mode, start.zoo disinfects with a `SpringCleaning()` after every command from the user. Lengthy, memory-intensive scripts may also benefit from a periodic scrubbing, especially if they allocate and subsequently remove, or un-alias, large variables frequently. For example, the following script:

```
a :: b :: { this[1000][1000] :: double }
for c1 in [1, 10000]
    (b = @nothing) :: a
endf
```

would benefit greatly from a `SpringCleaning()` within the loop.

`tan()`

syntax: (numeric) y = `tan`((numeric) x)

Returns the tangent of its (numeric) argument. The return variable is `R_double`.

`throw()`

syntax: `throw`((numeric) error_code [, (composite) error_script [, (numeric) error_index [, (Boolean) if_warning]]])

Causes an error to occur. This of course stops execution and throws Yazoo back to the last enclosing `trap()` function; if there is none (even in start.zoo) then Yazoo bails out completely. The first argument gives the error code; the optional second and third arguments allow one to specify which script and where in that script the error appears to come from. If one sets the optional fourth argument to true, then the error will be thrown as a warning instead. All arguments may be skipped with a '`*`'.

Although all real errors have error codes in the range 1-46, `throw()` works perfectly well for larger error codes that Yazoo has never heard of. It can be hard to tell when `throw()` is working. For starters, if the error code is zero then it will appear that `throw()` is not doing its job, just because 0 is code for 'no error'. `throw()` does require that the error code be zero or positive, so it gives a number-out-of-range error if the argument is negative. However, the following also gives a range error:

```
throw(6)
```

In this case `throw()` actually worked: we got an out-of-range error because that is error #6. (That caused the author some confusion at first.)

`top()`

syntax: (numeric) *vartop* = `top`((composite variable) *my_var*)

Returns the number of indices of the argument variable. The argument must be a composite variable or equivalent (e.g. set, function, class, etc.). `top()` does *not* count hidden members. Therefore the value it returns corresponds to the highest index of the variable that can be accessed, so

```
my_var[top(my_var)]
```

is legal (unless the top member is void) whereas

```
my_var[top(my_var) + 1]
```

is always illegal (unless we are in the process of defining it).

`transform()`

syntax: (numeric) *error_code* = `transform`((string) *compiled_code* [, (numeric) *code_ID* [, (composite array) *search_path*]])

Copies compiled bytecode, stored as a string, into the internal code of register `R_composite` *without* running the constructor. The transformed code may be given the search path of the transforming code, or encapsulated in the sense of having no search path leading out of the register, or given a custom search path. After invoking `transform()`, `R_composite` can either be used to run the code directly, or else serve as a template for defining objects that will have the same code. The internal ID number that Yazoo assigns the new code is copied into the optional second argument, if that is provided (it may also be skipped with a '`*`' void placeholder).

Every composite object in Yazoo has a code or sequence of codes (if the inheritance operator is used) that defines that object's type, its constructor and, if it is a function, its action upon execution. Code is something manifestly different from data: whereas data can be altered during runtime, a bytecode definition lies in Yazoo's internal memory and is generally inaccessible to the user. `transform()` bridges the gap between the separate worlds of code and data, by generating a new internal code definition from a string (data). Its counterparts, `variable_code()` and `member_code()`, copy internal code definitions back into strings.

We can demonstrate a transformation by using it to define a variable. Ordinarily, we would do this by writing something like:

```
cmplx :: {  real :: imag :: double  }
```

However, we could also have done the same thing manually, by writing

```
code_def := "real :: imag :: double"
compile(code_def, AllNames)      | bytecode stored in R_string
transform(R_string)    | embed bytecode into R_composite
```

(Ignore the `AllNames` list — this just ensures that the command prompt will remember our members' names.) After these three lines it is almost as if we had written

```
R_composite :: {  real :: imag :: double  }    | won't work though
```

except that this latter definition won't actually work because there will be a type mismatch with `R_composite`'s existing code definition. `transform()` removes any existing codes before incorporating the new code definition, so it doesn't have that problem.

One significant difference with the usual way of defining things is that `transform()` doesn't run a constructor, so `R_composite` won't have members `real` and `imag`. We do construct these members in our own variable whenever we copy the code into our own variables.

```
cmplx :: R_composite      | the constructor will run inside cmplx
```

There is one minor difference between a `cmplx` that was defined directly, and the one whose code we manually transformed. In the former case, the code of `cmplx` will be part of a larger script, so if `cmplx` causes an error then start.zoo script will be able to flag it correctly. If we have transformed the code ourselves, any error will have to be flagged in bytecode by the `disassemble()` routine unless we add the code and relevant info to the `ScriptStrings` and `YH_Lines` variables that start.zoo provides.

We did not set it up this way, but there is also a potentially bigger difference between native and user-transformed code, having to do with the search paths inherited upon transformation. The default is to use the search path of the calling function: the transformer was privileged to explore back through a certain set of variables looking for members, so its transformed progeny can be entrusted with the same. But this can be changed: if the transformation had a null third argument as in

```
transform(R_string, *, *)    | * in the third argument
```

then `R_composite` and by extension any variables defined from it would not inherit any search path: their code would only see members directly inside their respective variables. The practical consequence is that the following:

```
my_data :: double
...
print_data :: {  code, print(my_data)  }
print_data()       | fine
```

works, but the manually-transformed version

```
my_data :: double
...
compile("code, print(my_data)", AllNames)
transform(R_string, *, *)
print_data :: R_composite

print_data()     | member-not-found
```

will not work, since it will not be able to see `my_data`.

In fact, `transform` gives us great flexibility over the search path beyond the default/null options. Yazoo reasons: the calling function is entitled to explore whatever it can name, so its transformed code should have the same privileges; the order of things it can name is irrelevant; so why not allow a transformer to make any search path where it can name each step along the path? The way to do this is to put all stops on the search path, from last to first, inside a set which we put in the third argument. Thus `transform(my_str, *, { root, a, r.s })` causes code that runs in `R_composite` to look for members in `R_composite` first, then `r.s`, then `a`, then finally `root`. Then it will explore the transformer's search path (including the space of the transforming function). To prevent the transformer's path from grafting onto the manually-specified path, we put a void '`*`' token in the first position of the path.

Here are some examples of `transform()`-generated search paths.

```
f :: {
    my_code :: string
    code
    ...
    transform(my_code, *, *)            | R_composite
    transform(my_code, *, { * })        | R_composite

    transform(my_code, *)               | R_composite --> f --> root
    transform(my_code, *, { })          | R_composite --> f --> root

    transform(my_code, { a, b })        | R_composite --> a --> b --> f --> root
    transform(my_code, { *, a, b })     | R_composite --> a --> b
}
```

Why ever use `transform()`, if it is obviously more roundabout than defining things directly? One good reason is that all scripts barring the startup script must be transformed into memory from some other script, and `transform()` is the only tool that can do this. `start.zoo` transforms `user.zoo` in order to run it, and `user.zoo`'s `run()` routine invokes a transformation each time it runs a script. When Yazoo is run interactively, each entry from the command line is essentially a new script that must also be transformed. Some bytecode actually cannot be scripted, because the compiler doesn't have symbols for certain operations ('goto', def-general operators with unorthodox flags, etc.), and in these cases the bytecode has to be written and transformed manually. Both `start.zoo` and `user.zoo` at times find it necessary to transform hand-written snippets of bytecode that the compiler cannot produce.

Yazoo bytecode generated by `compile()` should always be legitimate; if it is not then there is a bug in the compiler. However, it is rather easier to obtain bogus bytecode than the legitimate sort when you're writing out the binary yourself. Therefore `transform()` makes no assumptions about the quality of bytecode it's given, and it runs a series of checks on it before loading into `R_composite`. Any error will of course prevent the code from being loaded. Error information is stored in the error registers, notably `R_error_code` which stores the error code (0 means no error).

`trap()`

syntax: (numeric) *error_code* = `trap`([code to run])

Runs its argument as code in the calling function, and returns any error value. No `code` marker is needed within a `trap()` call. Upon error, the argument ceases execution and the error code, index and script are placed into the registers `R_error_code`, `R_error_index` and `R_error_script` respectively; if the argument runs to completion with no error, then `R_error_code` and `R_error_index` are both set to 0. `R_error_code` is the return variable.

Likewise, a warning will cause `R_warning_code`, `R_warning_index` and `R_warning_script` to be set; the former two are cleared if no warning occurred. Since warnings do not stop execution, it is possible for several warnings to have been generated, in which case only the last one will be stored in the registers.

The `trap()` function possesses the special ability of running its arguments in the function that called `trap()`, rather than in a private argument variable, as is the case for all other built-in and user-defined functions. Thus variables which are defined within the `trap()` argument list will be accessible to the rest of the function.

Importantly, the `trap()` function clears the Yazoo error flag that stops execution. So if `trap()`'s argument causes an error, Yazoo immediately bails out to the original script, but then execution resumes after the `trap()` as if the error had never occurred. The only record of the problem is in the error registers, which are for the user's reference only and do not affect execution.

`variable_code()`

syntax: (string) *code_str* = `variable_code`((variable) *var*, (numeric) *code_index*)

Returns the specified (compiled) bytecode of a given variable. The way that multiple codes may be present is through the inheritance operator, which concatenates codes. `variable_code()` copies the bytecode from Yazoo's internal registry into a string (`R_string`), so it is an inverse operation to `transform()`.

In many respects the output of `variable_code()` is identical to the original output of the `compile()` function when that script was compiled, except that `variable_code()` also returns partial scripts. For example, we can compile and transform the following script:

```
code
var_a :: { x :: y :: double  }
var_b :: string
```

Now the above code is stored in `R_composite`, and by invoking

```
c_code := variable_code(R_composite, 1)
```

we can obtain the original bytecode (i.e. the direct output of the `compile()` function). But suppose we run `R_composite()` to generate the variables `var_a` and `var_b`. Then we can also write

```
c_code := variable_code(R_composite.var_a, 1)
```

and obtain the compiled version of "`x ::  y ::  double`". Of course, we cannot get any code back from `var_b` since that is a primitive variable.

Both functions/variables and members have code definitions. The member-side counterpart to `variable_code()` is `member_code`, which works in an analogous way.

`variable_code_ID()`

syntax: (string) *the_code_ID* = `variable_code_ID`((variable) *var*, (numeric) *code_index*)

Returns the global code ID of a given code of a given variable (there may be more than one per variable). `transform()` assigns a unique code ID to each chunk of bytecode that it loads, although disused codes that are cleaned from memory will generally have their ID numbers recycled. The return variable is `R_slong`.

As explained in above in the section on `variable_code()`, a given script may have a number of sub-codes, which are essentially the codes of the various members of whatever variable the script defined. The entire script gets a unique ID number, which `transform()` returns via its optional argument. All the sub-codes share the same global ID number. Thus one must use `variable_code_offset()` to discriminate between the codes of objects defined from the same script.

`variable_code_offset()`

syntax: (string) *the_code_offset* = `variable_code_offset`((variable) *var*, (numeric) *code_index*)

Returns the code offset of a given code of a given variable, relative to the start of the script in which it was loaded. This serves to distinguish the codes of different composite members that were defined in the same script and thus have the same code ID. The return variable is `R_slong`.

When a given script is transformed, the recipient variable of the transformation inherits its entire code, with an offset of zero. However, if that code defines some composite member, as in

```
...
f1 :: { ...  }
...
```

then that function's code is just the part of the larger script's code that is contained within the braces. It will thus have the same ID number. However, it will have a nonzero offset, since the code within braces necessarily begin some whiles into the script. (The offset begins with the first statement inside the braces, not the braces command itself.) The offset is given in units of bytecode words, starting from 1.

`variable_code_top()`

syntax: (numeric) *var_codes_num* = `variable_code_top`((variable) *var*)

Returns the number of codes contained within the argument variable. A variable may have multiple codes if it was defined using the inheritance operator, or if its template variable had multiple codes. Primitive variables do not have codes so they return a zero. The return variable is `R_slong`.

`variable_member_top()`

syntax: (numeric) *var_members_num* = `variable_member_top`((variable) *var*)

Returns the number of members of the argument variable. The number of members is emphatically *different* from the number of indices, since a given member may have zero, one or multiple indices, or its indices may be 'hidden'. Primitive variables return zero as they do not contain members. The return variable is `R_slong`.

`variable_type()`

syntax: (numeric) *var_type* = `variable_type`((variable) *var*)

Returns the type of the argument variable. The types IDs are listed in Table 1 on page 18. A composite variable simply registers as a '10' (composite) even though its type is properly determined by its code list. The return variable is `R_slong`.

5.6 Functions from Yazoo scripts

When Yazoo is run from the command line with argument, as in

```
user-prompt%./yazoo myfile.zoo
```

it executes the script "myfile.zoo". (If the filename ends in '.hob' then it will treat it as pre-compiled bytecode – anything else it treats as a text script.) On the other hand, if no argument is provided, i.e.

```
user-prompt% ./yazoo
```

then Yazoo looks for the default script file "start.zoo", or if that is not found, the pre-compiled bytecode file "start.hob". One of these two must be present. The user is of course free to course write his own start-up script. Below we discuss certain relevant contents of the prepackaged start.zoo file, which is what provides Yazoo's interactive command prompt.

Before opening the prompt, the pre-packaged start.zoo tries to run the file "user.zoo" in the user's workspace. Thus the interactive session begins with the user.zoo variables and functions pre-loaded into memory. The author has some ideas of his own about what is useful, which he has written into the provided user.zoo file (and which are explained in this section); but as the name of the file suggests, the user is absolutely encouraged to customize this script to suit his own tastes.

5.6.1 start.zoo

The start.zoo file provided with the Yazoo source files exists to provide an interactive command prompt for the user. It also performs some minor housekeeping chores, runs user.zoo and provides a few definitions of its own, etc. We won't say much about the bulk of this script, other than that it generally works, but there are a few variables and functions that might be of interest to the user. Most variables within start.zoo are invisible to the user living inside its workspace, but there are some, where noted, which can be accessed by the user, because start.zoo explicitly aliases them into the user's workspace.

The twelve variables and functions that follow are members of start.zoo that it makes available to the user. It is best not to go too rough on them, since it's pretty easy to crash start.zoo if they get disrupted.

`disasm:` if set to 'true' or 'on', causes errors to be flagged using the disassembler. When Yazoo flags a runtime error, it normally reprints the offending line from the pre-compiled source script alongside the error message. There is an alternative, which is to have it spit out the original bytecode instead. The bytecode is made somewhat less unreadable by the so-called 'disassembler' (which takes Yazoo bytecode, not real machine code), but is still more difficult than the original script. One advantage is that the error can be flagged more precisely in bytecode. Disassembly is normally the method of last resort, for cases when Yazoo cannot find the original code for some reason. Used by the author to satisfy the odd craving for a rush of bytecode.

`calculator:` causes the results of incomplete expressions to be printed from the command prompt. This only applies to expressions that are not assigned to variables, or used in any other way. For example, if the user types

```
a = 5 + 2
```

then nothing will be printed, regardless of whether `calculator` is on or off. However, if the user were to enter just

```
5 + 2
```

then the answer '7' will be printed if, and only if, `calculator` is set to `on`.

From the perspective of the start.zoo, 'incomplete expressions' are simply tokens within the workspace. Because these tokens are generally never used again, the prune package in start.zoo removes all the workspace tokens after every entry from the user. The calculator mode works by printing out tokens just before they are deleted.

The main reason for the calculator mode is that it saves the user from typing `print(...)` for every value he may want to calculate. The drawback is that sometimes things are printed that the user doesn't want. For example, functions may return error codes that are normally discarded, which will show up as incessant 0s, etc. that can get annoying – or worse, enormous blocks of data that expound voluminously onto the screen. Despite the potential disadvantages, the calculator defaults to `on`.

One standing problem is that the calculator doesn't necessarily know the order in which the arguments were entered (in the case of, say, '`5, 3, 2`'). For that reason it is best to group multiple results by braces.

`calculator_function():` : the function the calculator uses to print. user.zoo by default aliases this function to `sprint()`. To change the function one would type, e.g., `calculator_function = @printl`.

`ans:` : short for "answer". This is whatever the calculator last printed. Void if the calculator hasn't printed anything yet.

The following three user-accessible variables allow one to compile code that is compatible with the user's own variable names.

`AllNames:` the list of member names that have been defined so far. With each entry from the command prompt, this list is needed to sync the names in the current command with what has been defined so far. If this list goes then everything goes. By passing `AllNames` to the compiler the user can compile code that can access other externally-defined members by name.

`YH_Lines[]:` an array of strings, two per transformed script, containing information about where line breaks in the original text would fall in the compiled bytecodes. When an error occurs in a script, Yazoo tries to use this array to figure out which line of original text to display, rather than show a disassembly. The array is two-dimensional: `YH_Lines[5][1]` is the name of the file that code number 5 was compiled from, and `YH_Lines[5][2]` is a single string which gives the positions of all line-breaks for that entire script. This latter string is returned by `compile()` via the optional third argument. Upon an error, start.zoo writes the file name in the first string, and uses the second string to figure out which line in that file to display.

`ScriptStrings[]:` an array of strings, each containing the original text of a script that was compiled. By adding compiled scripts to this array, the user enables start.zoo to write out the offending source line at the site of any error. Each element of `ScriptStrings` corresponds to the respective code number; i.e. `ScriptStrings[6]` is the script whose code ID (as returned by `transform()`, or `variable_code_ID()`, etc.) is 6.

The next array is used by `go()` and `jump` in user.zoo.

`go_path[]:` a proxy array containing the search path, beginning with `root` and ending with the current working variable. This array is modified by `go()` and `jump()`, and is occasionally reset by start.zoo if it detects a problem. The user can add a small coding section to `go_path` which start.zoo will run if it needs to reset the path; the restriction is that the coding section should not declare any variables or run any functions or problems will start happening. user.zoo uses the coding section of `go_path` to reset the `pwd` variable.

The following two error lists and one function are made accessible to the user in the workspace.

`LLErrorStrings:` contains the names of each linked-list error according to their error codes; so for example `LLErrorStrings[1]` corresponds to error code number 1. Linked-list errors can be thrown to the user's C routines when they use linked list routines for string-processing.

`CompileErrorStrings:` the error strings that `compile()` outputs.

`RuntimeErrorString()`

syntax: (string) *error_string* = `RuntimeErrorString`((numeric) *error_code* [, (numeric) *code_word_at_error*])

This *function* returns the text of a runtime error (or transformation error – they use the same error list). The first argument is the error code; the optional second argument is the signed long word in the bytecode where the error was flagged (the location of an error that was given by `R_error_index`), in order to back out the name of the member that caused the problem. If the second argument is provided, and if the error was member-related error and Yazoo always throws the flag where the offending member ID was written in the bytecode, then `RuntimeErrorString()` can fill in a member name in the error message. For example:

```
> print(RuntimeErrorString(8))

member not defined

> print(RuntimeErrorString(8, 1))

member 'AllNames' not defined
```

In an interactive session member ID 1 corresponds to the `AllNames` variable.

Notice that `RuntimeErrorString()` lacks an 's' at the end of its name, whereas the two other error lists are both plural.

`disassemble()`

syntax: [(string) *disassembly* =] `disassemble`((string) *compiled_code* or { (string) *compiled_code*, (numeric) *start_pos* } [, (Boolean) *write_output* [, (string) namespace [, (Boolean) *expand_functions* [, (numeric) *flagpoint*]]]]

The `disassemble()` function returns a textual interpretation of compiled Yazoo bytecode. The required first argument is a string containing the bytecode. The function will return the 'disassembly' as a readable string unless the second argument is passed as `false`. If a namespace is provided in the third argument, then the disassembler will substitute member names for member numbers. The command-line namespace is `AllNames`. If given, the fourth argument determines whether inlined code definitions are disassembled (if true; this is the default), or skipped with an ellipsis if false. The fifth argument tells the disassembler to flag a certain location in the code, which is useful for marking errors in the code. The flag position is the index, starting from one, of the compiled long-integer word that caused the error; the default value of 0 causes no flag to be drawn. Arguments 2-5 are all optional and can be skipped or passed with a '`*`' to use the default values.

For example, suppose we wanted to flag the location of a warning (as these would not normally be disassembled). We could extract the bytecode and view some sensible representation of it by writing:

```
original_code :: disasm_code :: string

original_code = variable_code(R_warning_script, 1)
disasm_code = disassemble(original_code, *, *, *, R_warning_index)

print("Error ", R_warning_code, ":\n\n", disasm_code)
```

An alternative use for the disassembler is to skip over bytecode expressions, and one can do this by passing both the compiled code and a starting word index together within braces in the first argument. In

this case the disassembler will only disassemble up to the end of the expression, and if the starting word index was passed in a variable then that variable will be updated to the beginning of the next expression. For example, we can use this feature to write a function that finds the Nth sentence in a compiled expression.

```
go_to_Nth_sentence :: {

    code

    code_string := args[1]
    N := args[2]
    code_index := @args[3]

    code_index = 1
    for (n :: ulong) in [1, N-1]
        disassemble( { code_string, code_index }, false )
    endf
}
```

Skipping over code is much faster if we don't have to generate the bytecode text, which is why we passed a `false` second argument to `disassemble()`.

The following two classes in start.zoo are only used internally, but the user might like to know that they do what they do.

`R_Backup:` contains functions that save the registers after each command from the prompt has executed, and retrieves them again before the next command executes. From the user's point of view, this effectively isolates the registers from the influences of start.zoo.

`PrunePackage:` weeds out tokens and hidden members from the user's workspace. Tokens are created all the time in the normal course of operation, notably when the user's functions, or certain built-in Yazoo functions, are called but not assigned explicitly to a variable. Hidden members are also created with function calls. The following line generates both a token and a hidden member:

```
> do_something()
```

Both tokens and hidden members would ordinarily prevent the memory associated with these objects from being freed when the user no longer needed them. To prevent this, the `PrunePackage` removes all direct members of the workspace variable that have either a negative ID or have the hidden flag set. It then runs `SpringCleaning()` for good measure. The workspace is pruned after each entry from the user has finished executing.

Workspace tokens are almost completely decoupled from the rest of the code once they are created, so deleting them should be quite innocuous. The only way that the user could even detect the pruning would be to notice that, say, `this[1218]` or `root[1218]` mysteriously disappears between command-prompt entries. It would be a peculiar style, but if the user wants to use workspace tokens for storing valuables then he should consider commenting out the `PruneMembers()` invocation in start.zoo.

If the `calculator` option is set, then tokens get to print their contents as a final act before the execution. Hidden members get no such privileges.

5.6.2 user.zoo

Aside from a select few shared variables, start.zoo is careful to isolate the workspace from its own data structures, and it compiles the user's scripts in a namespace that is entirely separate from its own. Thus

it is natural for there to be two script files provided with Yazoo: start.zoo which defines the outer world, and user.zoo which inhabits the user's workspace and uses its namespace. user.zoo is just an ordinary script which is run before the command prompt appears – any sort of code can appear in it. The user.zoo file that comes with the program defines a number of constants and functions that the author has found useful (although as its name suggests the user should feel welcome to modify it to suit his own purposes). This section describes each of the pre-programmed members that are defined by the author's script.

The following are the pre-loaded variables and types defined in user.zoo.

`Boolean:` a data type, equivalent to an unsigned byte but intended for Boolean variables.

`true, false:` 1 and 0 respectively, intended to be the two values of Boolean variables.

`on, off:` 1 and 0 respectively; for the `disasm` and `calculator` variables (from start.zoo).

`passed:` 0, denoting the error code of a function that did not cause any error.

`e:` the exponential constant. Rather than provide an `exp()` function, Yazoo defines `e` so that the user can evaluate exponentials by writing `e^x`, `e^-2`, etc.

`pi:` the famous constant pi.

`root:` an alias to the user's workspace.

`DirectoryPaths[]:` a string array of pathnames to folders. The user.zoo routines (but *not* the built-in Yazoo functions!) will search each of these paths when looking for a file. user.zoo preloads the empty path, which is C's default directory (which in the author's experience has always just been the current shell directory when Yazoo was run). One adds elements to this array in the normal way, using the `[^...]`, `[+..]` operators, etc.

`pwd:` the current path to the workspace, stored as a string.

The first four routines in user.zoo assist Yazoo in some of the more awkward aspects of the way it handles variables.

`new()`

syntax: (same type) *var2* = @`new`((any type) *var1* [, (compatible type or *) *data_to_copy*])

The `new()` function returns a new instance of a variable, of the same type as the old and storing the same data. This is particularly useful within functions, where, as emphasized earlier, it is usually wise to create a new return variable with each function call. Instead of writing

```
f1 :: {
    ...
    ((rtrn =@ *) @:: double) = x + y
    return rtrn     }
```

we accomplish this with

```
f1 :: {
    ...
    return ( new(rtrn) = x + y )     }
```

or even just

```
f1 :: {
    ...
    return new(x + y)     }
```

`new()` is certainly helpful, but it is not necessarily the panacea that it may appear at first. The data contained in the arguments must be copied to the new variable, but in some cases the types may not match. For example, suppose we created a variable and later modified it:

```
comp1 :: {  a :: ulong  }
comp1.b :: sshort

new(comp1)     | cannot copy the data
```

In this case `new()` will create a new variable using `comp1`'s constructor, but since this does not match its current composition it will not be able to copy the data over, and will print a warning. The most common case where type does not match actual data is the case of an array:

```
arr1[4] :: string

new(arr1)     | cannot copy the data
```

As explained in the section on arrays, `arr1` is actually a composite variable with no explicit definition; the `string` data type defines that variable's single, 4-index member. Thus `arr1` itself has a null type, and it follows that `new()` will create a null variable with no members which can't be copied to. However, `new()` *can* copy arrays created using the `array()` function (also in user.zoo), so these two functions are synergistic.

By using the optional second argument to `new()`, one can decouple the variable that donates the type specification (first argument) from the variable that provides the data (second argument). If this second argument is void (as in `new(v, *)`) then no data is copied.

Note that `new({a, b})` doesn't make new instances of `a` and `b`, but rather just returns a new set of aliases to those variables.

`array()`

syntax: (numeric) *new_array* := @`array`((numeric) *size1*, *size2*, ..., (string) *type*)

Generates an N-dimensional array having the specified number of indices in each dimension, of the specified type (written as a string). The advantage of `array()`-generated arrays over those defined in the ordinary way is that these arrays have their full type definitions embedded in them, so they can be used as templates for defining copies of themselves.

```
a := @array(5, 2, "{ x :: ubyte, s :: string }")

a2 := a
```

The type definition explicitly encodes the given size in each dimension, so each copy of the array will be born with the original size even if the template array had been subsequently resized. If we had written

```
a := @array(5, ...)
a[^8]

a2 := a
```

then `a` would have had dimensions 8 x 2, `a2` would be the original 5 x 2, and there would be a type mismatch error when we tried to copy the data of one to the other.

The `new()` function does not work with standard arrays, but does work with arrays defined by the `array()` function.

`resize()`

syntax: (numeric) `resize`((composite) *array_var*, (numeric) *size* [(any type) *template_var*])

Some data types cannot be conveniently packaged into arrays. For example, a single function running as part of an array can run into difficulties that it would not have running solo:

```
f :: { storage[*] :: ubyte ; storage[^args[1]], .... }
f_array[5] :: f
f_array[2](4)   | won't work
```

Here, the third line will throw an incomplete-variable error, because it tries to perform an operation that would disrupt the uniform structure of the `f_array`. To get around this problem, we can bundle the five functions into a set, rather than a true array – and then create, add and delete elements from that set by use of the `resize()` function instead of the usual array-resize operator `[^...]`.

```
f_array :: {}
resize(f_array, 5, f)
...
resize(f_array, 2)
...
resize(f_array, 10)
...
```

The first argument of `resize()` is the set, and the second is the number of elements to rescale to. Like '`[^...]`', the `resize()` function either deletes elements from the end or adds new elements to the end, without affecting the other elements at the beginning of the set. The optional third argument gives the type of new array elements that are to be created. Unlike an ordinary array, this type can differ from that of existing elements in the set, but that will only affect new elements that are created. If no type is provided then the first element in the set is used as the template for any new elements that are created; if there is no existing first index then the new elements will be of the void type.

`copy()`

syntax: `copy`((variable) *var1*, { `"from"` or `"to"` }, (variable) *var2*)

This routine copies data member-by-member from one variable to another. This is useful for cases when otherwise equivalent variables cannot be copied directly using ordinary equate '=', because their indices are packaged in different groupings of members. For example, the following causes an error:

```
num3_a[3] :: double
num3_b :: {  a :: b :: c :: double  }

num3_a = num3_b     | error on this step
```

A type-mismatch error results because Yazoo will not copy three separate members' data into the three indices spanned by a single member. `copy()` circumvents that problem by manually copying each `num3_b[i]` to `num3_a[i]`.

The direction of data flow is determined by the second argument. If the second argument is `"from"`, data is copied from the third argument into the first; if it is `"to"` data is copied from the first argument into the third. There is no return value, but the error code is stored in the variable `copy.err_code`, with 0 indicating no error.

Next, `user.zoo` provides five routines that manipulate strings.

`cat()`

syntax: (string) *concatenated string* = `cat`((variables) *var1*, *var2*, ...)

Returns a string which is the concatenation of the arguments. This is just a convenient implementation of the `print_string()` function: `s = cat(v1, v2)` is equivalent to `print_string(s, v1, v2)`.

`char()`

syntax: (ubyte) *ASCII_code* = `char`((string) *single_character_string*)

Returns the ASCII code for a single-character string.

`C_string()`

syntax: (block) *string bytes* = `C_string`((string) *my_string*)

Strings in Yazoo are normally stored internally as linked lists. `C_string()` converts a length-N resizable Yazoo string to a $N + 1$-byte C-style string containing a terminating 0 character. The C-string is returned as a block type.

`lowercase()`

syntax: (string) *lowercase_string* = `lowercase`((string) *my_string*)

Converts a mixed-case string to lowercase.

`uppercase()`

syntax: (string) *uppercase_string* = `uppercase`((string) *my_string*)

Converts a mixed-case string to uppercase.

There are three routines in user.zoo for printing to the screen in different ways.

`printl()`

syntax: `printl`([data to print])

This function is the same as `print()` except that it adds an end-of-line character at the end.

`sprint()`

syntax: `sprint`([data to print])

`sprint()` is used for printing composite objects such as variables and functions; the 's' probably originally stood for 'spaced', 'set', or 'structure'. This is one of the most useful functions. Each member of the object is separated by a comma; the contents of composite members are enclosed by braces. Void members are represented by asterisks. The output is in exactly the format that Yazoo uses for constructing sets.

```
> my_object :: {  a := 5, b :: { 4, 10, "Hi" }, nothing }

> sprint("object: ", my_object)

object, { 5, { 4, 10, "Hi" }, * }
```

`sprint()` should not be used to print any of the error or data registers, since it calls functions that can overwrite those registers. A known bug is that if `sprint()` crashes due to a recursion limit error, then the next (but only the next) `sprint()` call will fail as well.

`mprint()`

syntax: `mprint`([data to print])

This 'matrix' print function prints tables of numbers. Each index of the argument is printed on a separate line; each index of a row prints separately with a number of spaces in between. For example:

```
> table :: { 2, { 3, nothing, 5 }, { 5/2, "Hello" } }

> mprint(table)

2
3         *         5
2.5       Hello
```

`mprint()` has two user-adjustable parameters. `mprint.field_width` controls the number of spaces in each row; it defaults to 11. `mprint.max_digits` controls the precision of numbers that are printed out; it defaults to 6. A `max_digits` of zero means 'no limit'. `mprint()` should not be used to print registers.

The following six routines are applied to the running of scripts or pre-compiled bytecode, or pertain to the execution of commands from the prompt.

`run()`

syntax: (numeric) *script_return_value* = `run`((string) *filename* [, (composite) *target*])

The essential `run()` function runs a script stored in a file. It is quite possibly the most useful routine in user.zoo, since running code is otherwise rather a formal and tedious procedure. `run()` performs a compilation incorporating the user's namespace (unless the source file ends in ".hob", in which case the script is presumed precompiled); it then transforms the bytecode and runs it. Any errors in the process are flagged along with the offending text. `run()` searches all directories in the `DirectoryPaths[]` array. If there is a direct `return` from the lowest level of a script (i.e. not within a function or type definition) then the return variable will be handed back to the calling script.

Normally the specified script is run in the user's workspace. Optionally, we can pass some other variable or function as a second argument to `run()`, in which case the script runs inside that object instead.

A given script is often run multiple times. By default, when executing a script `run()` first checks to see whether it has seen that script before, and if so removes any root-level objects that the script defined when it was last run. This is to avoid type-mismatch errors when the script tries redefining those objects. Occasionally this can be damaging. The user can tell `run()` *not* to pre-delete prior definitions by setting `run.CleanUp = false`.

`do_in()`

syntax: `do_in`((composite) *target* [, *search path* [, *code_args* [, *bytecode_mod_args*]]] , code, base script [, `code`, code modifying `bytecode[]`])

The `do_in()` tool allows one to run code in a specified location and with a specified search path, and gives the option of manually modifying the bytecode before it is run. The idea is that it is easier to write bytecode by perturbing a compiled script than to write everything from scratch.

The first argument to `do_in()` is the variable to run the code inside. The optional second argument gives a customizable search path, and it exactly mirrors the optional third argument to `transform()` (see the reference on `transform()` for how to specify a path). The third and fourth arguments, if given, are passed as `args[1]` for the script to be run and the bytecode-modifying script respectively.

Following the first code marker we give the text of the script that we want to run, or the closest that the Yazoo compiler can achieve. Often this is all we need. On occasion we may wish to modify the

compiled bytecode of the baseline script before it executes, perhaps to achieve something that is unscriptable. `do_in()` accommodates this need by running, in unusual fashion, the code following an optional *second* `code` marker/semicolon in its argument list (if that exists) after compilation but before execution. At that time the compiled baseline script will be stored in an array entitled `bytecode` of signed longs, and we may alter in any way whatsoever provided the bytecode comes out legitimate. In the extreme case we can give no baseline script and simply alias `bytecode[]` to an existing `slong`-typed array that was already filled with bytecode.

Here we show how to use `do_in()` to create an unjammable alias, which cannot be done using ordinary Yazoo scripting.

```
do_in(
    root

    code

    al := @var1

    code

    bytecode[3] = that + 128    | add an unjammable flag
)
```

`compile_and_do_in()`

syntax: `compile_and_do_in`((composite) *target* [, *search path* [, *code_args* [, *bytecode_mod_args*]]] , code, (string) *base script string* [, `code`, code modifying `bytecode[]`])

Compiles a script, optionally modifies it, and then executes the script in the provided directory. This is equivalent to `do_in()` except that the script is stored as an uncompiled string rather than compiled code. Even though the script string appears in the second coding block of the function arguments, it is passed in the same way as parameters in the first coding block (the constructor); for example:

```
compile_and_do_in(target_variable; "addendum :: string")
```

`go()`

syntax: `go`([code,] *path*)

Yazoo's `go()` function is similar to UNIX's `cd` command: it changes the working variable for commands entered from the prompt. A search path is dragged along behind that leads eventually back to `root` (the original workspace). To see how this works, type:

```
> a :: { b := 2 }

> go(a)

> sprint(b)    | we are in 'a', so this is legal

2
```

```
> sprint(a)    | search path extends back to root, so we can see 'a'

{ 2 }
```

The search path exactly backtracks the given path. If one types `go(a[b].c().d`, then the working variable is '`d`', and the search path goes backwards through (in order): the return variable of '`c`', then '`c`' itself, then the `b`'th element of '`a`', then '`a`' itself and finally `root`. Typing just `go()` sends one back to the root; typing `go(root)` is actually not quite as good because it puts `root` on the path list twice. To see the path, look at the global `pwd` variable.

`go()` works by updating the `go_paths[]` array defined by start.zoo. Each command entered from the prompt is transformed and run according to the current state of `go_paths`, so invoking `go()` does not take effect until the next entry from the prompt. Thus it was necessary in our example to separate the second and third lines: `go(a), sprint(b)` would have thrown a member-not-found error. For the same reason, while running a script (via `run()`), `go()` will do nothing until the script finishes – use `do_in()` instead.

When the user calls `go(...)`, Yazoo constructs the argument list before `go()` itself has a chance to run. Owing to this fact, certain sorts of go-paths will cause an error that `go()` can do nothing about. For example, `go(this[3])` will never work because '`this`' is construed as the argument variable, not the working variable. To get around this problem, `go()` gives us the option of writing the path after a `code` marker or semicolon, as in `go(code, this[3])`, as those paths are not automatically evaluated. A `code` marker is also useful if we need to step to a function's return variable but don't want the function to run more than once. `go(code, a.f().x)` will evaluate `f()` just a single time in the course of go-processing, whereas for technical reasons `f()` would have run twice had we not included the `code` marker.

`go()` at present has many limitations. Each path must begin with a member or register name or `this`, and all subsequent steps must consist of step-to-member (`a.b`) and step-to-index (`a[b]` and related) operations and function calls (`a()`). No `[+..]` or `+[..]` operators are allowed. The step-to-index operations are particularly dicey because of two nearly contradictory requirements: the path can only step through single indices, and for practical use the path must nearly always span complete members (i.e. *all* of the indices of most arrays). Although the latter is not a hard requirement, it is really hard to do anything meaningful within a single element of an array, because so many common operations involve creating tokens and hidden variables which can only be done for *all* elements of the array simultaneously. Even `go()` will not work at that point, so in this sticky situation start.zoo will eventually take pity and bail the user out. The upshot of all this is that `go()` is not very good inside of arrays.

`jump()` is a similar operation to `go()`, except that `go()` can shorten a path whereas successive `jump`s keep appending to the current search path.

`jump()`

syntax: `jump`([code,] *path*)

`jump()` is basically identical to `go()` except in the way that it handles the first step in a search path. For most details, see the explanation of `go()` above. The difference between the two functions can be seen by example.

```
> a :: { b :: { ... } }

> go(a.b), print(pwd)

root.a.b

> go(a), print(pwd)   | starting from a.b

root.a
```

```
> go(b), print(pwd)

root.a.b

> jump(a), print(pwd)   | again, starting from a.b

root.a.b-->a
```

`jump()` takes advantage of the fact that search paths in Yazoo can twine arbitrarily through memory space; we don't have to restrict ourselves to paths where each variable is 'contained in' the last. A more useful path would be something like `root.a.b-->c.d`: that would allow us to work inside of '`d`' while retaining access to '`a`' and '`b`', even if those latter lie along a different branch.

`ls()`

syntax: (string) *var_names* = `ls`([(variable) *var*])

Returns the names of the variables in the current directory, which is usually `root` (see `go()` and `jump()`). If an argument is provided then `ls()` returns the names of the variables inside that argument variable. Remember that `ls()` *requires* the parentheses! Just typing '`ls`' (no parentheses) at the command prompt will print out the internal variables of the `ls()` function.

The next four functions in user.zoo perform various file I/O and list- or table-related operations.

`Load()`

syntax: (string) *filedata* = `Load`((string) *filename*)

`Load()` (capital 'L') extends the built-in `load()` function by searching all paths in the `DirectoryNames[]` array.

`Save()`

syntax: `Save`((string) *filename*, (string) *filedata*)

`Save()` (capital 'S') extends the built-in `save()` function by searching all paths in the `DirectoryNames[]` array. This is important when a filename such as `archive/mail.txt` is provided, since the `archive/` folder may not be in the default (`./`) directory.

`SaveTable()`

syntax: `SaveTable`((string) *filename*, (table) *data* [, (string) *header*])

The `SaveTable()` routine exports data stored a set or array in table format to a file. In some ways it is similar to `mprint()`: successive indices of the table are written to successive lines, although fields within each index are separated by tabs (not spaces, as in `mprint()`). If the optional header is specified, that is printed verbatim at the top of the table, whether or not the header rows correspond to the rows of the table.

`ReadTable()`

syntax: `ReadTable`((table) *to_read*, (string) *raw_text* [, `code`, (Booleans) `IfHeader`, `ResizeFirstIndex`, `ResizeSecondIndex` = values])

The counterpart to `SaveTable()` is `ReadTable()`, which loads data into an array. It reads the data from a string, not a file, and tries to parse the data into the provided table (and if it fails it will print an error message to the screen). If the `IfHeader` variable is set to true, then the first line of text is skipped. Setting the `Resize...Index` arguments gives `ReadTable()` permission to adjust the size of the table to fit the data; in order for this to work the table must be a square array (i.e. not a set of members that can be resized independently). The default values of the optional arguments are `false` for `IfHeader`, and `true` for `ResizeFirstIndex` and `ResizeSecondIndex`. An error results in a non-zero value for `ReadTable.err_code`.

Finally, `user.zoo` provides six mathematical operations.

`round()`

syntax: (numeric) *rounded_integer* = `round`((numeric) *real_number*)

Rounds a real number to the nearest integer. For example, 1.499 rounds to 1, 1.5 rounds up to 2, and -1.5 rounds 'up' to -1.

`min()`

syntax: (numeric) *result* = `min`((numeric list) *the_list* [, `code`, `rtrn` = { `index` / `value` / `both`])

Returns the minimum element of a list: its index, value (the default), or the combination { index, value}.

`max()`

syntax: (numeric) *result* = `max`((numeric list) *the_list* [, `code`, `rtrn` = { `index` / `value` / `both`])

Returns the maximum element of a list: its index, value (the default), or both { index, value }.

`sum()`

syntax: (numeric) *result* = `sum`((numeric list) *the_list*)

Returns the sum of elements of a numeric list.

`mean()`

syntax: (numeric) *result* = `mean`((numeric list) *the_list*)

Returns the average (arithmetic mean) of the elements of a numeric list.

`sort()`

syntax: `sort`((table) *table_to_sort*, { (list) *sort_by_list* or (numeric) *sorting_index* } [, `code`, `direction` = { `increasing` / `decreasing`])

Sorts a list or table, which is passed as the first argument. If it is a table then a second argument is required: either the column number to sort by, or a separate list to sort against. So the following two sorts are equivalent:

```
MyTable[10] :: { a :: b :: double }

sort(MyTable, 1)     | sort by first column
sort(MyTable, MyTable[*].a)
```

The sort-by list will be unaffected.

Whether to sort in increasing or decreasing order can be specified after the semicolon/`code` marker; the default is 'increasing'. The column to sort by, whether it is in the same table or in a separate list, must be numeric; `sort()` will not alphabetize strings.

5.7 Linked list routines for handling Yazoo strings in C

Internally, nearly all of Yazoo's data memory is stored in resizable data structures called linked lists. This would be of little consequence to the user, except that `call()` passes strings as linked lists rather than, say, null-terminated byte arrays. The lazy reason for doing this is that each string is already stored in a linked list within Yazoo, so just passing the list saves Yazoo a conversion step. The better reason is that this method allows the user's own C code to share dynamic memory with Yazoo.

Each linked list consists of a header and a number of sublists. The header contains global information about the linked list but stores no data, so its size is fixed. Data is stored in the sublists. Each sublist begins with a header and is followed by an array of data storage, and in general each sublist holds a different number of elements. The C definitions of the relevant data types (taken from lnklst.h) are reproduced below.

```
typedef struct {
    unsigned long elementnum;
    sublistHeader *memory;
    unsigned short elementsize;
```

```
    unsigned char spare_room;
} linkedlist;

typedef struct sH_temp {
    unsigned long num_elements;
    struct sH_temp *next_sublist;
    unsigned long max_elements;
} sublistHeader;
```

The most important fields are `elementnum` and `elementsize` in the `linkedlist` data type, which are, respectively, the number of elements in the array (characters) and the size of each element (for strings, this must be 1). If the list is initialized then the `memory` field points to the first sublist; otherwise this pointer should be zero.

Each `linkedlist` variable contains important fields that are updated by the LL routines. Therefore it is critical that linked lists always be passed *by reference* to any C routine that could conceivably modify that list. Otherwise the original copy of the linked list variable will have out-of-date information on the size of and pointer to the list, which will probably cause a crash somewhere downstream.

To speed up the insertion of new elements, many of the linked lists in Yazoo allocate extra space in the sublists that they create. The `spare_room` field gives the amount of extra space to allocate, as a percent of what is requested. Thus if this field is set to 100 then each time a new sublist is created it will have double the space that is needed at the time. The maximum amount of spare room is 255.

The user really never needs to worry about the sublist data structure. Each sublist contains both the number of elements currently stored, and the maximum number that can be stored, in that sublist. Resizing an array often involves adding elements only to one sublist. The `next_sublist` pointer is to the header of the next sublist in the chain; if the current sublist is the last then this field is zero.

It is of course recommended that the user manipulate the linked lists using Yazoo's own LL routines. This is partly to avoid mangling Yazoo's strings, which would probably crash the program down the road; also, it should save the user a good deal of effort. Most linked list routines come in two flavors: a regular version and an 'F' version. 'F' stands for fast – the so-called fast routines just omit type-checking and some error-flagging. The error codes that many of these routines return will be explained in a later dedicated section on errors.

`NewLinkedList(), FNewLinkedList()`

syntax: (numeric) *err_code* = `NewLinkedList`((linkedlist *) *LL*, (numeric) *element_num*, *element_size*, *spare_room*, (Boolean) *if_cleared*)

Allocates the memory for a new linked list. The `linkedlist` variable itself is not created; rather its `memory` field is filled with a pointer to a newly-allocated sublist. The first three arguments are just three of the fields of the `linkedlist` data type described above: the initial number of elements, the byte size of each element, and the percentage of extra room to maintain in the list. The final argument, if set, zeros the memory of the linked list. Yazoo always sets `if_cleared` when allocating primitive data storage.

`DeleteLinkedList(), FDeleteLinkedList()`

syntax: [(numeric) *err_code* =] `DeleteLinkedList`((linkedlist *) *LL*)

De-allocates the storage of a new linked list. The actual variable of type `linkedlist` is not itself deleted, but its `memory` pointer is set to zero, indicating that the list is no longer defined. The fast version of this routine does not return an error value; the slow version at present always returns 'passed' (an error code of zero).

`InsertElements(), FInsertElements()`

syntax: (numeric) *err_code* = `InsertElements`((linkedlist *) *LL*, (numeric) *insertion_point*, *new_elements*, (Boolean) *if_cleared*)

Adds elements to the beginning, middle or end of a linked list. The elements are added *before* the specified existing index. So to add before the first element we must set the insertion-point argument to 1; to add after the final existing element we set that argument to the number of current elements plus one. The number of new elements is given by the third argument. The fourth argument, if set, zeros the new memory. Yazoo always sets the clear flag when new indices are added to primitive arrays, so they are always initialized to zero.

Each linked list has a field signifying the amount of spare room it should try to maintain in the list. This spare room takes up more memory, but it can improve the speed with which lists are resized, since adding new elements may not require allocating more sublists if the existing ones have the extra space. When there is not enough room, `InsertElements()` creates and inserts new sublists, again with the extra storage specified in the `spare_room` field of the linked list variable.

`AddElements(), FAddElements()`

syntax: (numeric) *err_code* = `AddElements`((linkedlist *) *LL*, (numeric) *new_elements*, (Boolean) *if_cleared*)

Same as `InsertElements()`, except that the elements are appended to the end of the existing list. (This is equivalent to calling `InsertElements` with an insertion point of `elementnum`+1.)

`DeleteElements(), FDeleteElements()`

syntax: (numeric) *err_code* = `DeleteElements`((linkedlist *) *LL*, (numeric) *first_index*, *last_index*)

Removes a given range of elements from the linked list. (This is not the same as zeroing the elements!) The range of elements to delete includes the first and last indices, so for example a range of {4, 6} removes three elements.

`DefragmentLinkedList(), FDefragmentLinkedList()`

syntax: (numeric) *err_code* = `DefragmentLinkedList`((linkedlist *) *LL*)

Rearranges the linked list's storage into one contiguous block of memory. A linked list is ordinarily broken up over a number of sublists, since that reduces the amount of memory shuffling that has to occur when elements are inserted or removed. If there will be no resizing of the list then it is faster to access in a de-fragmented form, and it saves memory too because zero extra storage is allocated in the defragmented sublist, regardless of the value of `spare_room` in the linked list variable.

The `call()` function defragments all of its string-arguments before executing a user-defined C or C++ routine.

`CopyElements(), FCopyElements()`

syntax: [(numeric) *err_code* =] `CopyElements`((linkedlist *) *source_LL*, (numeric) *first_source_index*, (linkedlist *) *dest_list*, (numeric) *first_dest_index*, *elements_to_copy*)

Copies the elements between two linked lists, or between two parts of the same linked list if the source and destination lists are the same. If the copy in being done within a linked list, then it is performed in such a way that data never overwrites itself and then gets re-copied (in other words, the procedure works correctly and gives the expected result). For obvious reasons the two lists' element sizes must match. The fast `FCopyElements()` does not have a return value.

`CompareElements(), FCompareElements()`

syntax: (numeric) *err_code/result* = `CompareElements`((linkedlist *) *source_LL*, (numeric) *first_source_index*, (linkedlist *) *dest_list*, (numeric) *first_dest_index*, *elements_to_compare*)

Compares the elements between two linked lists, or between two parts of the same linked list. The return value is either an error code (1-5), or the result of the comparison. If the two lists are equal these routines return 100 ('`equal`', defined in `lnklst.h`); otherwise they return 101 ('`notequal`').

`FillElements(), FFillElements()`

syntax: [(numeric) *err_code* =] `FillElements`((linkedlist *) *LL*, (numeric) *first_index*, *last_index*, (char) *byte_pattern*)

Fills the given range of elements of a linked list with the byte pattern specified. That is, each byte of data storage used by those elements is set to the value of the byte given in the fourth argument. The fast `FFillElements()` does not return a value.

`ClearElements(), FClearElements()`

syntax: [(numeric) *err_code* =] `ClearElements`((linkedlist *) *LL*, (numeric) *first_index*, *last_index*)

Clears – i.e. sets to zero – the given range of elements of a linked list. This is equivalent to calling `FillElements()` with a byte pattern of 0. `FClearElements()` does not return a value.

`SetElements(), FSetElements()`

syntax: [(numeric) *err_code* =] `SetElements`((linkedlist *) *LL*, (numeric) *first_index*, *last_index*, (void *) *read_address*)

Copies data from a buffer (the final argument) into a given range of elements in a linked list. The range includes the first and last elements. Whereas the linked list has fragmented memory, the buffer's storage is expected to be contiguous, and the given address is a pointer to the first element of data to copy. `FSetElements()` does not return a value.

`SetElement(), FSetElement()`

syntax: [(numeric) *err_code* =] `SetElement`((linkedlist *) *LL*, (numeric) *index*, (void *) *read_address*)

Copies data from a buffer into a single element of a linked list. This is equivalent to calling `SetElements()` with the same first and last element. `FSetElement()` does not return a value.

`GetElements(), FGetElements()`

syntax: [(numeric) *err_code* =] `GetElements`((linkedlist *) *LL*, (numeric) *first_index*, *last_index*, (void *) *write_address*)

Copies data from a range of elements of a linked list (including the first and last elements) into a buffer. The given address points to the start of the buffer. `FGetElements()` does not return a value.

`GetElement(), FGetElement()`

syntax: [(numeric) *err_code* =] `GetElement`((linkedlist *) *LL*, (numeric) *index*, (void *) *write_address*)

Copies data from a single element of a linked list into a buffer. This is equivalent to calling `GetElements()` with the same first and last element. The fast `FGetElement()` does not return a value.

`ElementExists(), FElementExists()`

syntax: (numeric) *err_code* = `ElementExists`((linkedlist *) *LL*, (numeric) *index*)

Tests to see whether a given element of a linked list exists. If the element is zero or greater than the maximum number currently in the list, it returns `false` (0). Otherwise it returns `true`, which is numerically 1 and hence unfortunately overlaps with the memory-allocation error code. However, since this routine cannot throw this particular error, a return value of 1 from `ElementExists()` always means 'yes', as in, yes the element does exist.

`Element(), FElement()`

syntax: (void *) *element_pointer* = `Element`((linkedlist *) *LL*, (numeric) *element_index*)

Returns a pointer to the given element of a linked list. Both versions of this function return the same thing; however, if the element doesn't exist `Element()` will politely return a zero whereas `FElement()` will simply crash the program.

`FSkipElements()`

syntax: (void *) *element_pointer* = `FSkipElements`((void *) *starting_pointer*, (numeric) *starting_index*, *indices_to_skip*)

Starting from a pointer to an element in the linked list (whose index must be specified), this routine returns the pointer to another element a given number of indices farther down the list. When canvassing large, heavily-fragmented lists it may be slightly faster to go through the list once using `FSkipElements()` than to call `FElement()`, which searches from the first sublist each time it is called. The new element must have a higher index than the old; Yazoo's linked lists cannot be explored in reverse. There is no 'slow' version of this routine.

5.8 Errors

This Help File is sorry to end on an unpleasant note. Yazoo scripts can generate a number of errors, upon compilation and during runtime, and despite all the author's best efforts it can still be a challenge to decipher

the exact mechanism that failed. Hopefully, having this dedicated section on goofs and glitches will help remove some of the mystery surrounding error messages, as it gives us the opportunity to enumerate the various possible circumstances around each given error more systematically than we have done so far.

There are four types of error that Yazoo can generate. At the lowest level, errors can be tripped by the linked-list routines that lie at the foundations of Yazoo – for example, if the computer runs out of memory, or a C routine manhandles a string. The compiler, invoked by `compile()`, can generate a second set of errors. `transform()`, which checks the compiled bytecode, returns a third category of error. The fourth category is the runtime errors. Here the `transform()` errors will be discussed in the section on runtime errors, since their error codes are intermingled.

Yazoo is poorly designed to recover from runtime memory errors. Running out of memory is quite likely to, for example, cause some operation to remain half-completed, or lead to incomplete garbage disposal. The author has not had any problems with memory errors so far, but has not had much opportunity to test them either, so he recommends that if one occurs then Yazoo should probably be restarted.

5.8.1 Linked-list errors

ID	name	ID	name
0	passed (no error)	2	index out of range
1	out of memory	3	linked list not initialized
		4	element sizes do not match

Linked list errors, by number

Element sizes do not match (#4)

Two linked lists are being copied or compared whose elements have different byte sizes. This error is only thrown by the slow routines; since the fast routines do not check sizes they will probably crash the program if the sizes turn out to be unequal.

Index out of range (#2)

A non-existent index of a linked list was passed to a linked-list routine. An index of zero always causes this error, since Yazoo elements begin at 1. Alternatively, passing an index greater than the number of elements in the list causes an error. The exception to this latter rule is that `InsertElements()` allows the insertion point to be one greater than the top element, denoting that the new elements are to be added at the end of the list. None of the fast linked list routines check the ranges of their arguments, so they do not throw this error and will likely crash if a bad index is passed.

Linked list not initialized (#3)

A linked list was passed to a routine (other than `NewLinkedList()`) that was not initialized. An uninitialized list is marked by a null pointer in the `memory` field of the linkedlist variable. To initialize a list one must first clear the `memory` field, then make a successful call to `NewLinkedList()` or `FNewLinkedList`. The fast routines do not check initialization and crash when handed an uninitialized list..

Out of memory (#1)

Memory could not be allocated as requested. The linked list routines that throw this error are `NewLinkedList()` and `InsertElements()`, as well as their fast equivalents. It is, in theory, and probably only in theory, possible for `DeleteElements()` to also throw this error, because if all elements are deleted it tries to allocate a size-1 sublist – but in practice this would never be a problem unless the computer was already really on the brink.

5.8.2 Compile errors

ID	name	ID	name
0	passed (no error)	17	not a number
1	out of memory	18	not a string
2	unexpected end of script	19	missing arguments
3	end of line expected	20	must follow if
4	end of script expected	21	conditional expected
5	unknown command	22	“end if” expected
6	invalid command	23	“end for” expected
7	argument expected	24	“end while” expected
8	no left-hand argument allowed	25	“until” expected
9	no right-hand argument allowed	26	‘in” expected
10	variable or number expected	27	‘[’ expected
11	variable expected	28	‘,’ expected
12	‘]’ expected	29	illegal control command
13	‘}’ expected	30	unexpected control terminator
14	‘)’ expected	31	not a statement
15	overflow	32	unknown error
16	underflow		

Compiler errors, by number

‘)’ expected (#14)

A left parenthesis was not matched by a corresponding right parenthesis, or else a spurious symbol (such as a right bracket) appeared before the right parenthesis. Parentheses are used both to group parts of expressions, and to run functions; in both cases both left- and right-parentheses must be present.

‘,’ expected (#28)

A comma, denoting the second half of a range, was not found when it was expected. The only place that a full range (starting and ending point), rather than just a singe index, has to be given is in a `for` statement; if only one number is given there then this error is thrown.

‘[’ expected (#27)

There was no left bracket following the `in` keyword in a for-loop. Each for-loop needs to give a range for the loop counter, which goes within brackets immediately after the `in` keyword.

‘]’ expected (#12)

An expression did not end with a closing right-bracket as it was supposed to. Both array-index operators and the range of a `for` statement are enclosed in brackets.

‘}’ expected (#13)

An expression that was supposed to end with a closing right-brace did not. A right brace is expected to the end any code block or composite object, such as a function, set, or composite variable.

Argument expected (#7)

An operator was missing an argument, either to the left or to the right. For example, the following two lines cause this error:

```
a = + 2
b =
```

In the first line, the addition operator was missing its left argument; in the second case equate was lacking a right-hand argument.

Conditional expected (#21)

An `if`, `while` or `until` was found that had nothing after it. Each of these statements needs a condition, such as the '`2 > 1`' part of

```
if 2 > 1, ...
```

If this part is missing, then a conditional-expected error is generated.

'End for' expected (#23)

A `for` statement did not close with a `end for` or, equivalently, `endf`. That means that either the end of the script or a different ending-marker (like `end while`) was encountered when the compiler was expecting the end of a `for` block.

'End if' expected (#22)

An `if` statement did not have a closing `end if` or `endif`. Either the compiler reached the end of the script without finding one of these, or it found a different ending-marker (like `end while`).

'End while' expected (#24)

An `while` block did not end with `end while` or `endw`. If the compiler encounters either the end of the script, or else a different ending-marker (like `until`) first, then it will throw this error.

End of line expected (#3)

A line of script continued after Yazoo expected it to end. The end of a line is marked by either an end-of-line character or carriage return, or a comma.

Certain reserved words, such as `else` and `endw`, require that the line end immediately afterwards; otherwise this error is thrown. The following code will cause this error, but it could be fixed by adding a comma between the `else` and `print()`.

```
if ...
else  print("no")    | incorrect
```

The second situation that causes an eol-expected error is when a '`&`', which breaks a sentence of script over separate lines, is followed by code on the same line, as in:

```
result = a + b + c &  + d
       + e + f
```

The line-continuation operator needs to be the last operator on the old line.

End of script expected (#4)

For some reason, the Yazoo compiler finished without encountering a null character, indicating an end of script. The user does not need to write in the null character to the strings that are sent to the compiler; Yazoo does that automatically. This must be some 'safety' error code for an unknown contingency; it is quite possible that this error could never be thrown.

Illegal control command (#29)

A line began with a flow-control marker that cannot begin lines, such as `step` or `in`. These are reserved words which cannot be used for variable names.

'In' expected (#26)

There was no `in` operator in a for-loop, or else it was in the wrong place. The correct syntax is to write '`in`', then the range in brackets, following the loop counter.

Invalid command (#6)

An operator has an argument of the wrong type. For example, numeric operators expect numeric arguments, so something nonsensical like

```
a = 3 + code
```

will cause this error.

Missing arguments (#19)

An argument-list was expected and none was found. The user's functions are regular objects, so they don't have to have an argument list after the function name. However, Yazoo's own built-in functions are different – they always have to be executed, by writing the (possibly empty) argument list after the function name. For example,

```
a = abs
```

causes this error, because `abs` is a reserved function that can only be run, not an object that can be copied directly from without running.

Must follow if (#20)

The compiler encountered an `elseif` or `else` without first encountering an `if` statement. These two words can only appear within an `if` block: i.e. after the `if` and before the `endif`.

No left-hand argument allowed (#8)

An operator had an argument to the left that it was not allowed to have. For example, the following line

```
a = return
```

will cause this error. Because the `return` operator has low precedence, the compiler grouped the expression to the left and construed that expression as a left-hand argument to `return`, which is only supposed to have an argument on the right.

No right-hand argument allowed (#9)

An operator had an argument to the right that it was not allowed to have. A confused C programmer might throw this error by writing:

```
a := *b
```

The only way to interpret '`b`' is as a right-hand argument to the asterisk, which because of the context is taken to be the void. The void takes no arguments, so this causes an error.

Not a number (#17)

The compiler tried to read a number that didn't follow the expected format. The format for a decimal number is a series of numeric digits (0-9) that may contain a single decimal point somewhere and, optionally, a minus-sign prefix (but not a plus sign!), and may contain an exponent suffix. The optional suffix consists of an '`e`' or '`E`' followed by a number (with an optional minus sign) that gives the base-10 exponent. No spaces may appear anywhere in the number, even between the number and the suffix. A valid decimal constant is:

```
-9.17e-17
```

Hexadecimal constants can also be written, and they have three differences with decimal constants. 1) A hexadecimal constant begins with a '`$`'. 2) The numeric digits of a hexadecimal constant include the six letters '`A`' through '`F`' or their lowercase equivalents. 3) The base-16 exponent of a hexadecimal is denoted by a '`H`' or '`h`' rather than '`E`' or '`e`'. The exponent is, however, written *in* base-10. Thus:

```
$-AC000000000h-10
```

is equivalent to `$-A.C` since the decimal point was shifted 10 (not 16) hexadecimals to the left.

Not a statement (#31)

Some operator that is not allowed to begin did. Many things that are not officially allowed to begin lines in practice are, since Yazoo will insert a prefix (like 'something `:= ...`') to make it into a legal line. However, in cases where this prefix would not make sense, like

```
a /= b
```

then this error will be thrown.

Not a string (#18)

A string didn't follow the allowed format. Strings begin and end with a double-quotation-mark character: `"`. The string must all be written on one line; the line-continuation operator '`&`' does not work inside strings. A line break (other than a comma) within a string causes this error, as does the presence of a null character. These and other special characters can be encoded with escape sequences, which if not done properly is the other way to cause this error.

Each escape sequence begins with a backslash '\' character. The characters that have special escape sequences are the end-of-line sequence '`\n`' (which is not a carriage return!), the tab sequence '`\t`', a backslash '`\\`' and a double-quotation mark '`\"`'. More generally, any character can be written into a string by specifying its ASCII code with two hexadecimal (0 through F/f) digits after the initial backslash. There *must* be two digits – characters before ASCII-16 have a zero as their first digit. The sequence "`\4A`", for example, represents the character 'J'.

Out of memory (#1)

Yazoo was not able to allocate memory while compiling a script. This usually means that Yazoo ran out of memory, or that the program tried to allocate a block of memory that was larger than some maximum set by the operating system (i.e. on a DOS/Windows machine). Try compiling a smaller script.

Overflow (#15)

The compiler tried to read a numeric constant that was larger in magnitude than the maximum that will fit in a double-precision variable. For negative numbers this means that the number was less than the most negative allowed number. For example, writing the number `5e999` causes this error.

Unexpected control terminator (#30)

The Yazoo compiler ran into some end-of-block marker like `endif` or `until`, outside of *any* flow-control-block command like `if` or `do`. A different error will be generated if an ending marker follows the *wrong* flow-control command.

Unexpected end of script (#2)

A null character was encountered in mid-script. Usually, or perhaps always, the presence of a spurious null character will actually cause a different error, but this 'safety' error code is reserved for some unforeseen problem that would slip past the other error checks.

Underflow (#16)

A number was encountered which was so small that it was read as zero. This particular error is currently ignored by the compiler, so it should not be encountered when compiling. Its only relevance has to do with the fact that the built-in `read_string()` function uses the compiler's number-reading machinery, and during runtime *that* function will throw a number-read warning upon underflow.

Unknown command (#5)

The compiler encountered some mysterious symbol which it cannot recognize as an operator. For example, the following will cause this error

```
a = %
```

because a percent sign has no use in Yazoo.

Unknown error (#32)

Something strange and unclassifiable happened. This error should never occur, and if it does then it probably indicates a bug in Yazoo.

'Until' expected (#25)

A `do` block did not end with `until` as it was supposed to. Either the compiler reached the end of the script first, or else the `do` block ended with some different ending-marker (like `end for`).

Variable expected (#11)

A number or symbol was found where a variable, register or function (which may return a variable) was expected. There are two situations that Yazoo expects to find a variable: 1) immediately following a `for` statement, and 2) immediately following a step-to-member '`.`' operator.

Variable or number expected (#10)

Yazoo expected to find either a number, a register, or the name of a variable or function (which might return a variable), and it found something else instead (usually a symbol of some sort). The three places in which Yazoo expects to find either an inlined number or a variable are: 1) within the brackets of a `for` loop; 2) after the `step` marker of a `for` loop; and 3) within the brackets of an index operator. The index operators are step-to-index/indices '`[]`', resize '`[^]`', and add-indices '`[+]`' and '`+[]`'. When two indices are given, both are required to have either numbers or names of potential members.

5.8.3 Runtime errors

ID	name	ID	name
0	passed (no error)	24	invalid block size
1	out of memory	25	not numeric
2	illegal command	26	not a variable
3	code overflow	27	unequal data sizes
4	inaccessible code	28	error reading number
5	jump to middle of sentence		
6	number out of range	30	mismatched indices
7	division by zero	31	variable is jammed
8	member not defined	32	unjammed member
9	variable has no member	33	unjammed proxy
10	invalid index	34	not composite
11	member is void	35	undefined string
12	proxy is void	36	multiple indices not allowed
13	cannot step to multiple members	37	wrong number of arguments
14	cannot step to multiple proxies	38	string expected
15	underflow	39	invalid index
16	type mismatch	40	error in argument
17	argument out of range	41	nonexistent register
18	self reference	42	nonexistent Yazoo function
19	illegal target	43	can't find external function
20	no member associated with variable	44	recursion depth too high
21	target was deleted	45	I/O error
22	incomplete member	46	return flag
23	incomplete variable	47	finished signal

Runtime errors, by number

Argument out of range (#17)

A built-in function was passed a numeric argument that was outside of the allowed range. The functions that cause this error are: `extract()` or `find()` (if a string position is either zero or greater than the length of the string, or if the second range element is greater than the first); and any function that pulls up information about a member (e.g. `member_type()`, `GetParentScriptInfo()`) or code (such as `variable_code()`), as the member/code number must be on the interval 1 through the top of that list.

Cannot step to multiple members (#13)

Yazoo encountered a path which spans multiple members at some juncture. This is not allowed; the user can step to multiple indices of a given member, but never two members. So, for example,

```
a[2] ::  double
a[1, 2]
```

is legal, whereas

```
b :: { one :: two :: double }
b[1, 2]
```

causes an error on the second line. Both the `[*]` and `[^...]` operators require there to be only one member in the variable.

Cannot step to multiple proxies (#14)

Yazoo tried to step into multiple proxies at the same time. The restriction on proxies is that only one can be accessed at any given time, since they all generally point to different variables.

Can't find external function (#43)

A `call()` statement was instructed to run a C/C++ function that does not seem to exist. `call()` takes either a function name or a function ID. If it is a name, a string corresponding to that name must appear in the `UserFunctionSet` array (in userfn.c/cpp). If it is an ID, it must be on the interval $1 - N$ (if there are N user-defined C/C++ functions). Thus

```
call(0)
```

will produce this error.

Code overflow (#3)

A transforming script did not seem to end in the way it was supposed to. For example, the constant-string command gives a string length followed by the characters of the string; if the length of the string is greater than the remaining length of bytecode, this error will be thrown. All scripts must end with a 0 terminating code word (long integer).

Division by zero (#7)

This warning is caused when the user tries to divide by zero at runtime. Yazoo still tries to perform the division as normal, so it will likely generate an infinity or a not-a-number value. And if for some reason a divide-by-zero crashes the computer, then the computer will crash.

Error in argument (#40)

There was a problem with a parameter passed to a built-in function. This is a catch-all error for miscellaneous problems with arguments, but only two functions actually throw it: `compile()` and `find()`. In the case of `compile()`, it means that the user passed a nonsensical namespace: either it was smaller in size than a single long integer, or it didn't end in a null (zero) character. In the case of `find()` it means that the mode argument (whether to find or count, and which way to search) was not in the range -1 – 1. (It is supposed to be -1, 0 or 1, but if it is within the allowed range any fractional part is simply discarded).

Error reading number (#28)

The `read_string()` function tried to read a number into a numeric variable and failed (because the number was written incorrectly). This always occurs as a warning, not an error. It is the equivalent of the "not a number" compile-time error, except that it occurs at runtime since that is when `read_string()` parses its arguments.

Finished signal (#47)

This error code, thrown by the `exit` statement, is a bookkeeping device that causes Yazoo to fall out of the program. It does not mean that anything went wrong in the code. If an `exit` command is written inside of a `trap()` function, the program does not quit, but instead falls out of the `trap()` with error code 47. Typing `throw(47)` is equivalent to typing `exit`.

Illegal command (#2)

Yazoo found a nonsensical command in bytecode that it was trying to transform into memory. For example, the addition operator expects two numeric arguments: if one argument is the constant-string operator, this error will be thrown. Or, Yazoo may hit a command ID that simply doesn't exist in any of its tables, which will also cause this error. The final cases are that the user tried to run the Nth code of a user function where $N < 0$, or tried to define a N-length string where again $N < 0$.

Illegal target (#19)

Yazoo tried to make an alias to something other than a variable. For example,

```
a := @double
```

will cause this error.

Inaccessible code (#4)

This transformation 'error' is always thrown as a warning, so it does not stop the transformation. It occurs when a null end-of-script operator was encountered before `transform()` hit the end of the compiled-code string. The code will still run, but the spurious code-terminating marker will likely cause it to finish early.

Incomplete member (#22)

The user tried to do something with part of a member that was not allowed. The most basic situation is in referencing some (more than one), but not all, of a member's indices – legal, but that must then be the last step in the path. Each step before the final step in a path must go either into a single index or complete member (all indices of that member). So, the following causes an incomplete-member error:

```
b[5][2] :: double
print( b[1, 3][1] )      | causes an error
```

It is a purely technical limitation – this error is the only thing preventing the user from referring to several non-contiguous blocks of memory at once, which is a situation that Yazoo is not equipped to handle.

Other situations that require complete members are the redefinition and removal of members. In these cases the member must really be complete: a single index is not allowed if the member spans multiple indices. Continuing our last example, the following command

```
remove b[2, 3]
```

would also cause this error.

Incomplete variable (#23)

Only a partial variable was referenced where Yazoo expected to find an entire variable. Certain operations can only be performed sensibly on whole variables (all indices): resizing, re-aliasing and redefining variables, or adding or removing members. For example, for a hypothetical array `a[3][2] :: ubyte` the following causes an incomplete-variable error.

```
a[1][*] = @*
```

It is worth noting that Yazoo generally ignores incomplete-member and incomplete-variable references in define statements if the redefinition won't change the type. Thus it is legal to call `a[5] :: ulong` twice, even though the second time that would be construed as a redefinition of just the fifth element of '`a`'.

Invalid block size (#24)

The user tried to define a block-typed variable with either a negative number of bytes, or a byte-count greater than the maximum unsigned long, which is not allowed. This error will also be generated if the byte-count is $\pm\infty$ or NaN.

Invalid index (#10, #39)

The most common cause of this error is that the user requested an index of a variable that does not exist: e.g. `array[5]` when only four members exist, or `array[0]` under any circumstance. Remember that hidden members do not contribute to the total index count. A second possibility is that an index range was given where the second index was (more than one) lower than the first, which is not allowed: `array[4, 2]` for example. (`array[+4, 3]` is not allowed, but `array[4, 3]` is actually legal and just returns zero indices.) Finally, the index-addition operators `[+...]` and `+[...]` only enlarge existing members, so if the variable does not contain any members this error will be thrown.

Unfortunately, this error tends to crop up *any* time something goes wrong inside of an index expression, irrespective of the cause. An example is the case below, where '`b`' is not defined.

```
array[5] :: double
array[b]     | b not defined
```

This currently causes an invalid-index error, although a better error message would be: member 'b' not found. This may change in the future.

There are also a few wild possible ways to generate an invalid-index error: by resizing arrays beyond the maximum signed long value, or by adding infinite or NaN indices.

The fact that this error has two error codes is irrelevant to the user. It has to do with the way Yazoo keeps its books: error code 10 signifies that, if it happened in the left-hand argument of a define operator, the problem might be fixed by adding a new member, whereas 39 always stops execution.

I/O error (#45)

One of the following built-in functions couldn't perform the action instructed of it: `load()`, `save()`,

`input()` or `print()`. The usual cause of this is that the user tried to read a non-existent file, or read or write a file with a bad pathname or without the necessary read/write permissions. Specifically, this error is thrown if a file cannot be opened or closed, or if data could not be read or written properly to a file or the standard input or output.

Jump to middle of sentence (#5)

Yazoo found a goto statement leading to the middle of a compiled sentence. One thing that Yazoo checks before it transforms code is that each goto is a jump to the beginning of a compiled sentence. (This applies to both conditional and unconditional jumps.)

Member is void (#11)

Usually this means that Yazoo attempted to step into a void member (one that had no target). For example, if we unlink some variable via `x =@ *`, and then try to use `x` in any way before reassigning it, then we will probably get this error. Many of the built-in functions throw this error if any of their parameters are void.

Member not defined (#8)

The user gave a pathname with some member name that Yazoo could not find. If the offending member is at the beginning of the path, as in

```
print(list[5])
```

where '`list`' causes the error, then Yazoo is telling us that `list` was not to be found anywhere along the search path: the current function or any enclosing object (unless the path was clipped at some point). If the problematic member was some intermediate point in the path, as in

```
data[5].header = ""
```

assuming the error was due to the '`header`' member, then that only means that `header` was not immediately inside `data[5]`.

If possible, Yazoo gives the member name in single quotes along with the error message, as in: "`member 'header' not defined`".

Mismatched indices (#30)

The user either tried to copy or compare data between arrays of different sizes, or else alias one array to another of a different size. For example, the following will cause this error regardless of how '`a`' and '`b`' were defined.

```
a[1, 3] = b[1, 2]
```

One common cause of this error is for multiple indices of a constructor to be running at once:

```
b :: ulong
a[1, 3] :: { q := @b }
```

where the definition of `q` attempts something analogous to `a[*].q := @b`.

Note that arrays of different dimensions can be copied/compared if their indices are specified manually and the total number of indices is the same (and each is a contiguous block of memory – see the section on arrays). So, for example, the following is legal:

```
q[4] :: r[2][2] :: ulong
q[1,4] = @r[1, 2][1,2]
```

If the last index of the array on the left-hand side of a compare or equate is a '`[*]`', then Yazoo will automatically resize it if that will prevent a mismatched-indices error. Sometimes this does not work; for example, in the case below:

```
a[2][3] :: b[5] :: string
a[*][*] = b[*]
```

we will get a mismatched-indices error because only the *last* index of '`a`' can be resized, which is incompatible with '`b`' having an odd number of indices. We would also get this error if the first dimension of '`a`' was sized to zero, even if '`b`' was also of zero size.

Multiple indices not allowed (#36)

Several instances of a variable were given where only one was expected. Whenever Yazoo expects either a number or a string, only variables or single array indices are allowed. For example, the following causes this error.

```
if 4 < a[1,2], ...
```

Likewise, the following expression is also (at present) disallowed for the same reason.

```
a[2] :: { b :: ulong, if 4 < b, ... }
```

Most cases of a multiple-indices error involve the built-in Yazoo functions, since many of their parameters require single numbers or strings.

No member associated with variable (#20)

Yazoo attempted an operation that involves a member, not just a variable, and it didn't have one. For example, the define operator specializes the member type as well as the variable type. The following code

```
f :: { code, return 5 }
f() :: ulong
```

will generate this error since the `return` command returns only a variable, not the member of the function that points to it.

In addition to the define operator, both equate and forced-equate require a member in order to resize an array via the `[*]` operator. (That is because both the variable and the member have to be resized.) Finally, both of the member-insert operators '`[+...]`' and '`+[...]`', as well as `remove`, operate on members and therefore will generate this error if none is provided.

Nonexistent register (#41)

Yazoo tried to transform code that somewhere invokes to a register that does not exist. The step-to register operator in compiled bytecode is followed by a register ID, which as a signed long must be in the range 0-9 (because there are ten registers). Any number outside this range causes a non-existent-register error. Yazoo's compiler should never generate fake register IDs, so this error should only happen if the user wrote or modified the bytecode by hand.

Nonexistent Yazoo function (#42)

The user invoked a Yazoo function with a function ID that does not exist. The function ID appears immediately following the built-in-function operator in compiled bytecode. As a signed long it must fall in the range 0-44 because there are 45 built-in functions; another number generates this error. This error should only happen if the user wrote or modified the bytecode by hand, since the compiler theoretically should only produce valid ID numbers.

Not composite (#34)

Yazoo expected a composite variable but was given a primitive variable instead. All of the 'step' operators – '`.`', '`[]`', '`[^]`', '`[+]`', etc. – require that the starting variable be composite. For example, the following generates a not-composite error:

```
a :: double
print(a.b)
```

This error will also be generated if there is something other than a composite variable in the left-hand argument of a code-substitution operator ('`<<`'), or in the argument of a `top()` function.

Not a variable (#26)

Something that was not a variable was passed to an operation that needs a variable. Both equate and forced equate require an existing variable to copy data into, and either a constant expression or a variable to copy the data from. Likewise, the comparison operator '`==`' requires two arguments that are either constants or variables. In addition, all the step commands (step-to-member, step-to-index, resize, etc.) require that the user start from some variable. So if we define:

```
f :: { }     | 'returns' the void
```

then the following return a not-a-variable error:

```
f() = 5
f().b
f = *      | because * is not a variable
```

A common cause of this error is to attempt to copy a scalar into an array – e.g. `a[1, 3] = 2` – because when it sees multiple indices on the left it expects an array on the right as well. For the same reason, writing `tp :: { (a :: double) = 3 }, arr[5] :: tp` will also cause this error.

Not numeric (#25)

Yazoo was given a non-numeric expression where it expected a number. For example:

```
b := "7"
2 + b
```

generates this error. In different contexts one can also get compile-time or other runtime errors; the phrase "`2 + "7"` generates "illegal command".

Note that this is a different error from the "not a number" compile-time error, which occurs when a number was mistyped.

Number out of range (#6)

This error is usually thrown as a warning. Yazoo tried to assign a numeric variable a value beyond what it can store. For example, trying to assign 255 to a signed byte, or -100 to an unsigned byte, will cause this warning. This warning is *not* caused by rounding-off errors: for example, we can assign the value 1.9 to an integer variable, and Yazoo will quietly round it off to 1 without raising any warning flag.

There are two instances in which this message is generated as an actual error. The first is when `print_string()` is given a precision-argument outside the range 0 - 255. The second case is when `throw()` is given an error code outside the range 0 - `ULONG_MAX`.

Out of memory (#1)

Yazoo was not able to allocate memory while it was running a script. Any memory error will cause this message: for example, if the computer is out of memory, or if the memory manager for some other reason refuses to allocate a block of the requested size. Yazoo is not particularly well-designed to recover from run-time memory errors – or at the very least, it has not been well-tested in this regard – so it is recommended that the program be restarted if this error occurs.

The usual cause of a memory error is frequent creation and removal of variables within a loop. Due to Yazoo's incomplete garbage collection the deleted variables often do not get erased from memory until the loop is finished and the command prompt is brought up again. Calling `SpringCleaning()` periodically within the loop will force complete garbage collection.

Proxy is void (#12)

Yazoo tried to step into a proxy that was void. Before this proxy can be used, it must be assigned a target via the equate-at '`=@`' operator.

Recursion depth too high (#44)

Too many nested functions are being run. In order to avoid blowing the program stack, Yazoo sets a limit to the number of functions that can be run simultaneously within one another. This limit is set in the `GL_MaxRecursions` variable in hobbsh.c – Yazoo comes with it set to 100. So if we have `f1` call `f2` which then calls `f3`, then there is no problem because the total depth is only 4 (the three functions plus the calling script). On the other hand, if we try

```
f :: { f2 :: this }
```

then we will immediately get this error because this requires an infinite level of recursion. (`f` creates `f2` which creates its own `f2`, etc.)

Return flag (#46)

This error code is used internally to cause Yazoo to fall out of a function when it hits a `return` statement. Since returning from functions is a perfectly legitimate thing to do, the error code is always set to 0 (no error) after the function has been escaped.

Self reference (#18)

A variable with an alias to itself was given to an operation in which self-aliases are not allowed. It is fine to have aliases to oneself or to an enclosing variable, which is what the 'self-reference' refers to. However, one cannot, say, copy data to that variable since it has an infinite depth. For example, if we define:

```
me :: { self := @this }
```

then `me` contains `me.self`, which contains `me.self.self`, etc. Therefore if we try to use this variable, or any enclosing variable, in an equate, comparison, forced equate, or the built-in functions `print_string()`, `read_string`, `size()`, `load()` or `save()`, we will get this error. If we try to print this variable, then `print()` will simply skip any part it has already printed and generate a self-reference warning instead of an error.

String expected (#38)

A built-in Yazoo function requiring a string argument was passed something that was manifestly not a string. For example, `transform()` expects a string (since compiled code is stored as a string), so writing

```
transform( my_code :: block 50 )
```

will generate this error.

Target was deleted (#21)

A variable was deleted while it was being aliased. One has to try hard to get this error; in this newer version of Yazoo it may well not be possible. The author tried and was unsuccessful.

Type mismatch (#16)

Two variables/members do not have matching types. This error can occur either when data is being copied or compared, or when a variable or member is having its type altered (e.g. via the define operator).

When copying or comparing two variables, Yazoo is quite puritanical about ensuring that they have very similar (though not necessarily identical) structures. If (any part of) the first variable is composite, the (corresponding part of) the second variable must also be composite, have the same number of (non-hidden) members, and each member must have the same number of indices. Primitive variables and variable members must also match: numeric with numeric variables (though their types may be different), strings with strings, and blocks with blocks of the same size.

The `read_string()` function copies data from a string into a variable. It reads block data directly byte-for-byte from the string, so if there is not enough data in the string to fill the block then this error will be thrown. If it has to fill a recipient string variable with a null string, as in the case below:

```
read_string("5", { ulong, string })
```

then it throws a type-mismatch *warning*, on the assumption that there probably should have been something more substantive to read.

Yazoo is fastidious in a different way when it comes to redefining members and variables: it doesn't care about the structure of the variables, but it does require that their types exactly match or be compatible, with the new types. Two different numeric types are not compatible. Two block types of different sizes are also incompatible. Trying to redefine an existing non-proxy member with the proxy flag also causes this error.

One composite type can be changed (specialized) into another only if all of the existing N codes are also the first N codes in the new type, in the same order. Thus if we define `var1` to be of type `a:b`, then we can specialize it into `a:b:c` but not `c:a:b` or `b:a:c`.

Some variables are created implicitly, in particular when arrays are defined.

```
arr2[3][4] :: string
```

For example, if `arr2` had not been previously defined, then both `arr2` and the target of its only member (which has 3 indices) would have been defined implicitly; only the member of that first member would be typed to a string. On the other hand, if `arr2` or its member had already been defined to a primitive type, the above would have caused either a not-composite error if `arr2` had a target, or a type mismatch error if it was void.

Trying to apply the inheritance operator to two primitive types, even outside of a define clause, causes this error if the types are not the same. Concatenation of primitive types is totally meaningless, by the way, even when it works — it doesn't generate a compound type.

Undefined string (#35)

A string was used in a comparison or passed to a built-in function before having been assigned a value. For example, if we define

```
s :: string
```

then the following commands

```
if s == "", ...
load(s)
```

will give us this error until we have given '`s`' a value via `s = "..."`.

Unlike other primitive variables, strings have to be actually initialized to a value before they can be used. Note that when `s` was first defined it did *not* contain a null string (those are perfectly legitimate values to hold). It had no value. If we had instead initialized string '`s`' with the code

```
s := ""
```

then we would have avoided this error.

Underflow (#15)

The underflow warning alerts the user that `read_string()` attempted to read a number containing significant digits that are below the double-precision accuracy limit of the machine (`DBL_MIN` in C). Numbers as small as 5e-324 can be stored; however, reading "1e-310" gives an underflow warning since only the first 14 digits will be reliable.

Unequal data sizes (#27)

Yazoo was not able to perform a forced equate because the byte-sizes of the left- and right-hand arguments were different. Before throwing this error, the forced-equate operator explores two options for making the data fit. 1) If the final step into the left-hand variable involved a `[*]` operator it tries to resize that last member. 2) If the left-hand variable contains a string, that can soak up excess bytes from the right-hand argument. If after (1) and (2) the data on the right still cannot fit into the variable on the left, then this error is thrown.

If both (1) and (2) apply (i.e. the user is forcing data into a string-containing array), Yazoo may not able to figure out how to resize the array correctly. In such a case the user must resize the array manually to avoid an unequal-data-sizes error.

Unjammed member (#32)

Yazoo tried to step into a member that was 'unjammed' – i.e. a member that was defined as unjammable, and one whose target variable was later resized. When this happens the unjammable member immediately becomes obsolete: it has the wrong number of indices for its target. Therefore it cannot be used again until it has been redefined.

Unless the user does something sophisticated, unjammable members are only generated by the compiler, for example when it generates tokens. The following code will generate an unjammed-member error:

```
array[5] :: ubyte
set :: { array[1, 5] }

delete array[3]       | unjams the token in our 'set'
print( set[1][1] )    | the second '[1]' causes the error
```

Had we aliased `array` explicitly in our `set` variable, the alias would not have been defined as unjammable, and we would have gotten a jammed-variable error when we tried deleting the array element.

Unjammed proxy (#33)

Yazoo tried to step through a proxy that had been 'unjammed'. This is identical to the unjammed-member error, expect that the member was defined with the proxy flag set.

Variable is jammed (#31)

The user tried to resize a variable that has more than one member pointing to it. That is, indices are being added to or removed from an array, and there is an alias array that encompasses these same indices whose size would also change if this error did not prevent it. For example, consider the following setup:

```
a[5] :: double
```

```
b := @a
c[3] := @a[1, 3]
```

Member 'a' (and also 'b') points to a composite variable containing one (unnamed) member spanning five indices; that member in turn points to five instances of a type-double variable. Member 'c' is structured the same way; it points to a different variable, but that variable's width-3 member points to some of the same instances of the double-type variable. Now if we try

```
a[+2]
```

we will get a jammed-variable error. The culprit is 'c' – or more accurately, the width-3 member inside c's composite variable. 'b' does not cause any problems, since it shares the same width-5 member that 'a' uses; resizing 'a' implicitly does the same to 'b'.

Note that if we had instead tried

```
a[+5]
```

then there would have been no error, since that part of the array is not shared between two members.

The alias coming from 'c' is said to have jammed the variable we were trying to resize. Certain members (usually, members that Yazoo adds implicitly to the script during compilation) cannot jam variables, because they were defined with the unjammable flag set. No error jumps in to prevent the user from resizing an unjammable member's target variable by another member, so if this happens the unjammable member gets unjammed – i.e. deactivated.

Variable has no member (#9)

Either the '[*]' or the '[^...]' operator was used in a variable that does not have any members. Both these operators need at least one member to operate ('[*]' requires no more than one). The only exception is when the expression falls on the left-hand side of a define statement, which causes the members to be created if they do not already exist.

Wrong number of arguments (#37)

A built-in Yazoo function was called with the wrong number of arguments. For example, the following expression will cause this error

```
top(a, b)
```

because top() accepts only a single argument.

Index