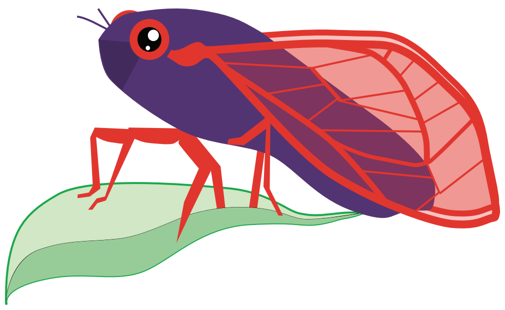
Cicada Scripting Language

Brian Ross

April 5, 2022



Contents

1 Introduction

Cicada script is a simple, interpreted language that interfaces naturally with C and C++. Its goal is to have the best from two worlds: an interactive and portable scripting language, embedded with our C routines that run at the top speed of the machine. We build a Cicada program by compiling the Cicada source files (which are written in C/C++) together with our own C/C++ files. Voilà, this gives us a command-line environment for executing our C routines and managing their data.

For example, suppose we want 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 it would be nice to be able to call our routine from the command line. So we paste our C routine into Cicada, specifically in the source file userfn.c. We then give our sorting routine a name within Cicada, by modifying one of the pre-existing lines in userfn.c.

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

Finally, we update our Makefile and re-compile Cicada.

```
user-prompt% make cicada CC=gcc
```

Finally we run Cicada, which brings up a command prompt allowing us to make our list and sort it.

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

And we are done. Of course there are a few details. The syntax should be easy to grasp at a glance – in fact you can change the default syntax if you want – but at its heart this language is a bit of an oddball. For example, many Cicada functions will call their own arguments.

The rest of this document explains the Cicada language in more detail three times over: through an extended example (Chapter 2), a textbook-style explanation (Chapters 3 and 4), and a reference section (Chapter 5).

2 Example: neural networks in Cicada

Cicada (formerly Yazoo) originally aimed to be a neural network simulator, missed by a mile, and ended up as a scripting language instead. As a nod to the original mission, our tutorial will be a program that lets a user create and train artificial neural networks, which are algorithms based on models of we think neurons in our brains work that are widely used for fuzzy tasks such as pattern-recognition. The time-consuming processes of simulating and training the networks will be written in C, for speed. (This code is also C++ compatible.) We will then write Cicada wrappers so that these networks can be controlled at the Cicada command prompt. Finally, we will write (and debug) a script to use these networks to 'learn' words that the user types at the command prompt.

2.1 C implementation

A neural network has two modes of operation: computation, and training. In computing mode, the user fixes the activity of certain input neurons, and the other neurons x_i will calculate:

$$x_i \leftarrow \frac{1}{1 + e^{-\sum_j w_{ij} x_j - b_i}}$$

Here x_i represents the activity level of neuron i, w_{ij} is the strength of the connection from neuron j to neuron i, and b_i is the neuron's sensitivity. In training mode, we will update the weights and biases of each neuron using the following formula:

$$\begin{aligned} w_{ij} &\leftarrow w_{ij} + \eta \cdot x_i x_j \\ b_i &\leftarrow b_i + \eta \cdot x_i. \end{aligned}$$

The parameter η determines the learning rate, which should be large enough to train quickly, but not so big that the algorithm overshoots and destabilizes the network. This training algorithm[1] works for symmetric networks ($w_{ij} = w_{ji}$) such as we will use. As the saying goes, *neurons that fire together, wire together.*

These two basic operations of a neural network are encoded by simple formulas, but since those formulas will be used many times in the course of using a network they are the time-consuming step. We will therefore write that part of our program is C. Notice that our outermost function has the same form as `main()`, but we will use a different name so as not to conflict with Cicada's own `main()` function.

NN.c

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


// runNetwork():  evolves a neural network to a steady state
// Takes the params:  1 - weights; 2 - neuron activities; 3 - input; 4 - step size
// (& additionally, in training mode):  5 - target output; 6 - learning rate

ccInt runNetwork(ccInt argc, char **argv)
{
    neural_network myNN;
    double *inputs, step_size, *target_outputs, learning_rate;
    int i, numInputs, numOutputs;


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

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

```
    for (i = 0; i < numInputs; i++)
        myNN.activity[i] = inputs[i];
    for (i = numInputs; i < myNN.numNeurons; i++)
        myNN.activity[i] = 0;

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

        if (getSteadyState(myNN, numInputs, step_size) != 0)  return 1;
        trainNetwork(myNN, -learning_rate);

        for (i = 0; i < numOutputs; i++)
            myNN.activity[numInputs + i] = target_outputs[i];

        if (getSteadyState(myNN, numInputs+numOutputs, step_size) != 0)  return 1;
        trainNetwork(myNN, learning_rate);         }

    else if (getSteadyState(myNN, numInputs, step_size) != 0)  return 1;


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


    return 0;
}


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

int getSteadyState(neural_network NN, int numClamped, double StepSize)
{
    const double max_mean_sq_diff = 0.001;
    const long maxIterations = 1000;

    double diff, sq_diff, input, newOutput;
    int iteration, i, j;

    if (numClamped == NN.numNeurons)  return 0;


        // keep updating the network until it reaches a steady state

    for (iteration = 1; iteration <= maxIterations; iteration++)    {
        sq_diff = 0;

        for (i = numClamped; i < NN.numNeurons; i++) {
            input = 0;
            for (j = 0; j < NN.numNeurons; j++)   {
            if (i != j)   {
                input += NN.activity[j] * NN.weights[i*NN.numNeurons + j];
            }}
            newOutput = 1./(1 + exp(-input));

            diff = newOutput - NN.activity[i];
            sq_diff += diff*diff;
            NN.activity[i] *= 1-StepSize;
            NN.activity[i] += StepSize * newOutput;
        }
        if (sq_diff < max_mean_sq_diff * (NN.numNeurons - numClamped))
            return 0;
    }

    return 1;
}


// trainNetwork() updates the weights and biases using the Hebbian rule.

void trainNetwork(neural_network NN, double learningRate)
{
    int i, j;

    for (i = 0; i < NN.numNeurons; i++)    {
    for (j = 0; j < NN.numNeurons; j++)    {
    if (i != j)    {
        NN.weights[i*NN.numNeurons + j] += learningRate * NN.activity[i] * NN.activity[j];
    }}}
}
```

<u>NN.h</u>

```
typedef struct {
    int numNeurons;     // 'N'
    double *weights;    // N x N array of incoming synapses
    double *activity;   // length-N vector
} neural_network;

extern int runNetwork(int, char **);
extern int getSteadyState(neural_network, int, double);
extern void trainNetwork(neural_network, double);
```

2.2 Putting the C in Cicada

Notice that our C code doesn't allocate/deallocate memory, load or prepare the data sets, save the results, or do any of the other miscellaneous operations that don't need to happen thousands of times per second. We will script the remainder of our program, not only because Cicada automates most of these housekeeping functions but also because scripted functions can be controlled from the command line.

We bring our C functions into Cicada by referencing them in a Cicada file called `userfn.c`. First, make sure the C compiler knows about our neural network routines. It's sloppy, but the easiest way to do this is to put `NN.c` and `NN.h` into the Cicada directory, and then add the following line to `userfn.c`.

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

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

Next, we need to tell Cicada about our C routine in `userfn.c`, by adding it to the `UserFunctions[]` array. We provide both a name and a function address.

```
userFunction UserFunctions[] = { { "pass2nums", &pass2nums }, { "cicada", &runCicada },
                                   { "RunNetwork", &runNetwork } };
```

Now we need to flesh out the 'set up data types' comment in `runNetwork()`. It turns out that Cicada provides a handy function for reading data from `argv` (an array of pointers to the variables and arrays passed to the C code, as you would expect). Since the memory is shared between the two environments, our C function can also send data back to a script by writing to these variables. Cicada also passes a list of array types and sizes at the end of `argv[]`. Putting all together, we add the following lines of code at the beginning of `runNetwork()` (i.e. in place of the first comment in `NN.c`).

```
arg_info *argInfo = (arg_info *) argv[argc];

myNN.numNeurons = argInfo[1].argIndices;
numInputs = argInfo[2].argIndices;

getArgs(argc, argv, &myNN.weights, &myNN.activity, &inputs, byValue(&step_size), endArgs);

if (argc == 6)  {
    numOutputs = argInfo[4].argIndices;
    getArgs(argc, argv, fromArg(4), &target_outputs, byValue(&learning_rate));      }
```

This code uses the `arg_info` data type, so we also need to add

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

at the beginning of NN.c.

Notice that we assigned `myNN.weights`, `myNN.activity` and `inputs` by reference rather than by value. One reason 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. On the other hand `step_size` is not a pointer variable, so we passed its value using the `byValue()` function.

We won't try to save the results of our calculations in the C code. We can simply delete the "save results" comment line in NN.c.

The final step is to recompile Cicada with our source files. 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 cicada CC=gcc`" (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 '`cicada`' or '`./cicada`', depending on the system. We should see:

```
>
```

Once inside Cicada, we can call our neural network function by typing

```
$RunNetwork(...)
```

with the function's arguments listed in place of the dots.

Our custom version of Cicada has fast, native neural network functionality, but it is hidden behind a clunky syntax. The next task is to write a Cicada class that bundles a neural network's data with user-friendly functions that initialize, run and train that network.

2.3 Writing and debugging a Cicada wrapper

Let's try our hand at writing a Cicada script to wrap around our C routine. It's our first time scripting so we will probably have a few bugs.

<u>NN.cicada</u>

```
neural_network :: {

    numNeurons :: int
    numInputs :: numOutputs :: numHiddens

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


    init :: {

        if trap( { numInputs, numOutputs, numHiddens } = args ) /= passed  then (
            print("usage:  myNN.init(inputs, outputs, hidden neurons)\n")
            return 1        )

        numNeurons = numInputs + numOutputs + numHiddens + 1

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

        return
    }


    run :: {

        numArgs :: rtrn :: int
        step_size :: learning_rate :: double
        inputs :: outputs :: [] double

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


        code

        numArgs = top(args)

        if trap(
            inputs[^numInputs + 1]
            inputs[<2, numInputs+1>] = args[1][*]
            if numArgs == 4  then (
                outputs[^numOutputs]
                outputs[<1, numOutputs>] = args[2][*]
                { step_size, learning_rate } = { args[3], args[4] }  )
            else if numArgs == 2  then &
                step_size = args[2]
            else  throw(1)
        ) /= passed  then (
            print("usage:  myNN.run(input, step_size OR ",
                  "input, target output, step_size, learning_rate)\n")
            return 1        )

        if numArgs == 2  then &
            rtrn = $RunNetwork(weights, activity, inputs, step_size)
        else &
            rtrn = $RunNetwork(weights, activity, inputs, step_size, outputs, learning_rate)

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


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

We should save NN.cicada in same directory that contains the `cicada` application, `start.cicada` and `user.cicada`. (start.cicada runs the command prompt, and user.cicada pre-loads a number of useful functions.)

Assuming we are at Cicada's command prompt, we can try out our new wrapper by typing:

```
> run("NN")

Error:  left-hand argument expected in file NN

32:          inputs[^1] = 1       // the 'bias' input
                                   ^
```

Hmm.. we're obviously not finished with NN.cicada yet. Fortunately compile-time errors like the one above are usually easy to sort out. In our case we accidentally wrote a C-style comment '`//`' in place of a Cicada comment '`|`', which Cicada interpreted as a pair of division signs. We make the straightforward fix to NN.cicada:

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

and try running the script again.

```
> run("NN")

Error:  member 'numHiddens' not found in file NN

4:      numInputs :: numOutputs :: numHiddens
                                   ^
```

We have made progress: NN.cicada successfully compiled and began running — before impaling itself on line 4 and duly filing a complaint. Runtime errors can be more difficult to debug than compile-time errors, but the interactive environment helps somewhat.

```
> neural_network.numHiddens

Error:  member 'numHiddens' not found
```

With a little knowledge of the scripting language we would see that we tried to define `numInputs` and `numOutputs` to be of type `numHiddens`, which would be allowed except that `numHiddens` has not been defined yet. What we meant to do was to define all three of these variables to be of type `int`. Make the following correction to line 4:

```
    numInputs :: numOutputs :: numHiddens :: int
```

and re-run our script.

```
> run("NN")

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

This time the script successfully compiled and ran, although it printed out a suspicious ‘usage’ message even though we never tried initializing a neural network. Let’s see if `init()` works when we’re actually trying to run it.

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


>
```

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

```
> neural_network.activity

{  }
```

So something is definitely wrong. At this stage we might want to look at a number of other variables that should have been set, and the easiest way to do that is to ‘go’ inside our network.

```
> go(neural_network)


> weights

{  }


> numNeurons

0


> go()
```

The last line takes us back to our normal workspace. So our `init()` call was a dud – nothing happened. Well, the next step is to put a trace statement in the coding section of the function.. wherever that is... So, ho ho, we find that we forgot the `code` marker in the `init()` method, which explains all of our problems. That should go at the beginning of `init()`; the method should now begin:

```
    init :: {

        code

        if trap( { numInputs, numOutputs, numHiddens } = args ) /= passed  then (
```

etc. For the last time, let’s go back and try

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

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

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

2.4 The Anagrambler

After running a few more tests we eventually convince ourselves that `NN.cicada` is working, so we open a new file in our Cicada directory and start thinking about how to put our networks to use.

The particular learning algorithm we are using is well suited to the task of pattern completion. We will demonstrate by building a network to unscramble anagrams. The inputs to this network will be the number of times each of the 26 letters appears in a word, encoded in the activity levels of 26 input neurons. The outputs will be the ordering of those letters relative to alphabetical, using n output neurons for a maximum word length n. (For example, a lowest-to-highest ranking of outputs of 3-2-4-1-5 for the input 'ortob' would imply the ordering 3-2-4-1-5 of the characters 'b-o-o-r-t', which spells 'robot'.)

anagrambler.cicada

```
forEach :: {

    counter :: int

    code

    for counter in <1, top(args[1])>  &
        args(args[1][counter], counter)
}


anagrambler :: neural_network : {

    setupNN :: {

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


        code

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

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

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


    ask :: setupNN : {

        outputString :: string


        code

        run(NN_in, params.step_size)

        sort(NN_out, 2)

        NN_out[^numOutputs]
        NN_out[*].order = activity[<numInputs+2, numInputs+numOutputs+1>]
        if size(the_word) < numOutputs  then NN_out[^size(the_word)]

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

        print(outputString)
    }


    teach :: setupNN : {

        c1 :: int


        code

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

        NN_out[^numOutputs]

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


the_word :: string
NN_in :: [26] double
NN_out :: [anagrambler.numOutputs] { order :: double, letter :: char }

alph := "abcdefghijklmnopqrstuvwxyz"
```

At last, we're ready to build a digital brain and put it to the task of unscrambling anagrams. We run Cicada, then load each of the two `.cicada` source files.

```
    > run("NN")


    > run("anagrambler")
```

Next we specify how big of a brain we need. Let's decide to work with words of 6 or fewer characters (so, 6 output neurons), and of course we expect a 26-character alphabet (lowercase only please). So we enter the following line at the command prompt.

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

With the custom brain built and ready, we can try

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

ehllo
```

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

```
> anagrambler.teach("hello", 10)     | 10 = # training cycles


> anagrambler.ask("lleoh")

hello
```

Thus concludes our Cicada demonstration. Of course we've barely probed our anagrambler's intelligence, but a great virtue of the interactive command prompt is that the experimental cycles are very short. For example, the author was taught it to perfectly recall three words (`hello`, `tomato` and `yazoo`, the former name of Cicada) within a minute of coaching. But maybe the network can learn even faster – is 10 rounds of training on each word too many? Will our network learn faster if we increase higher learning rate, or will it become unstable? It's simple to test.

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

We might also play around with the network architecture, for example by increasing the number of output neurons to allow the anagrambler to memorize longer words. Most intriguingly of all, we could add hidden neurons to increase the complexity of its calculations, using the third argument of `init()`.

That is the last that we shall say about neural networks – this is, after all, a Cicada help file. Hopefully this example shows the great advantage of marrying C code to an interpreted environment. The final section of this chapter explains the general procedure for bringing C/C++ into Cicada.

2.5 Rules for embedding C/C++ code

There are two ways to merge C or C++ with Cicada. 1) We can embed our own C functions inside Cicada, as shown in the tutorial. 2) Alternatively (or additionally), we can embed the Cicada inside of a larger C or C++ program.

Start with procedure (2) since it is very simple. In order for a C/C++ source file to run Cicada, it needs to include Cicada's `ccmain.h` header file. The actual function call takes the form:

```
errCode = runCicada(argc, argv);
```

With no arguments (`argc` = 0) Cicada will run the `start.cicada` script. If we set `argc` to 1 and pass the name of a script file in `argv[0]`, then Cicada will load and run that script file instead.

Procedure (1) for embedding external C or C++ functions into Cicada involves three steps. First, one declares the C/C++ routine in a particular way and writes the code accordingly; this is easy because Cicada-embedded functions look just like the `main()` function of a stand-alone program. In the second step the author introduces his routine to Cicada by changing a couple of lines in Cicada's `userfn.c` file. The final, optional step is to write a wrapper in Cicada's scripting language that allocates storage for the C/C++ function and provides it with a smooth interface.

2.5.1 C function declaration

An C/C++ function needs to be of type `(ccInt)(ccInt, char **)` in order to be called from within Cicada. Normally, `ccInt` is just C's basic `int` type. The two arguments to the C function are: 1) the number of variables/arrays passed to the function, and 2) an array of pointers to these variables; and its return value is usually interpreted as an error code. (This is the same function type as C's `main()` function, making it easy to adapt a stand-alone C program.) Each variable or array passed to the function is a list of one or more Booleans, characters, integers, floating-point numbers, or strings stored in linked lists.

Here is an example of a Cicada-compatible function that has one argument of each possible data type:

```
#include "userfn.h"

ccInt myFunction(ccInt argc, char **argv)
{
    ccBool firstArg = *argv[0];
    char secondArg = *argv[1];
    ccInt *thirdArg = argv[2];
    ccFloat *fourthArg = argv[3];
    linkedlist *fifthArg = argv[4];

    ...

    return 0;         // no error
}
```

We want to point out several features of this function call. First, we included Cicada's `userfn.h` header file. Second, notice that we passed the first two arguments by value but the other three arguments by reference. We must pass an argument by reference if it might be changed, *particularly* if it is a string (the `linkedlist` type). The reason it is essential to pass strings by reference is that any resize of a string using a copy of the original linked list will probably crash the program. Arrays are usually passed by reference because in C an array variable is the pointer to the first element.

Finally, we see that our C function used several Cicada-defined types: for example `ccInt` rather than `int`. Ordinarily these two types are interchangeable, according to definitions found in the `lnklst.h` header file. However, it is possible to change Cicada's default integer type to, say, a long integer by making changes to two of Cicada's files. The first change is the type definition in `lnklst.h`, which becomes

```
typedef long int ccInt;
#define ccIntMax LONG_MAX
#define ccIntMin LONG_MIN
```

The second change ensures that Cicada will print our new variables properly. At the top of `cmpile.c` we slightly change two lines to:

```
const char *printIntFormatString = "%li";
const char *readIntFormatString = "%li%n";
```

We can also change Cicada's default floating-point type `ccFloat` in `lnklst.h`, as well as `ccBool` which is the integer type used to store Booleans. For floating-point changes we will also want to modify the definitions at the top of `cmpile.c` to ensure proper printing (and possibly give the type a new name in `cicada.c`). On the other hand, character variables are always simply of type `char`, and string variables always use the `linkedlist` data type. Cicada's `lnklst` source files provide routines for accessing and resizing linked lists, as described in the reference section.

Cicada's `userfn` source files provide a handy `getArgs()` function, which helps us simplify our C routine.

```
#include "userfn.h"

ccInt myFunction(ccInt argc, char **argv)
{
    ccBool firstArg;
    char secondArg;
    ccInt *thirdArg;
    ccFloat *fourthArg;
    linkedlist *fifthArg;

    getArgs(argc, argv, byValue(&firstArg), byValue(&secondArg),
            &thirdArg, &fourthArg, &fifthArg);

    ...

    return 0;         // no error
}
```

The first two arguments of `getArgs()` are the two arguments to our C/C++ function: in this case `argc` and `argv`. After that we list the address of each variable to load, using a `getArgs()` macro for each C variable that is not a pointer variable. If we don't want to load all arguments, we can use the `endArgs` macro. For example our function could have loaded just the first three arguments with the command

```
    getArgs(argc, argv, byValue(&firstArg), byValue(&secondArg), &thirdArg, endArgs);
```

Likewise `getArgs()` can skip to a particular argument using the `fromArg()` macro, as in

```
    getArgs(argc, argv, fromArg(2), &thirdArg, &fourthArg, &fifthArg);
```

Notice that `fromArg()` uses *C-style indexing*, so the next argument read after `fromArg(2)` is `argv[2]` which is the *third* argument.

To give our C function some context, let's look an example script that calls it. A typical script first sets up variables and arrays to pass, and then runs the C code using either Cicada's `call()` function or its shorthand syntax where the function-name string follows a dollar sign (without extra spaces on either side). Notice that the function name-string is defined separately from the actual function name, in the file `userfn.c`.

```
nums :: [2][5] int
call("myFunctionInC", true, 'q', nums, pi, "a sample string")    | syntax 1
$myFunctionInC(true, 'q', nums, pi, "a sample string")           | syntax 2
```

This is pretty straightforward: three of the arguments are constants, `pi` was pre-defined in `user.cicada`, and `nums` is a two-dimensional array. It should be noted that `nums` is effectively a one-dimensional list as far as the C routine is concerned, and that the order of elements in the list is [1][1], [1][2], ..., [1][5], [2][1], ..., [2][5] (Cicada indices begin at 1). So, for example, the array variable written in Cicada as `nums[2][3]` is in memory slot `thirdArg[7]` within our C code.

Finally, the `call()` function actually passes one *extra* parameter, located at `argv[argc]`, which is a list containing `argc` elements of type `arg_info` (defined in `userfn.h`) giving the types and numbers of elements of each parameter passed from the script. This is useful for argument-checking, and it also tells the C code

type # (macro)	Cicada type name	C type name	default data type
0 (bool_type)	bool	ccBool	char
1 (char_type)	char	char	char
2 (int_type)	int	ccInt	int
3 (double_type)	double	ccFloat	double
4 (string_type)	string	linkedlist	{ int; void *; int; int; }
5 (composite_type)	{ }	N/A	N/A
6 (array_type)	[ ]	N/A	N/A

Table 1: Cicada data types. Types 0 - 4 are primitive

the total number of elements in each array that was passed (though not the sizes of the individual dimensions of 2+ dimensional Cicada arrays). The type of a parameter is given by the `argType` field of its respective `arg_info` variable, as defined in Table 1. The total number of array elements is given by a `argIndices` field.

Continuing our last example, we might want to take advantage of the `arg_info` array by adding these few extra lines to `myFunction()`.

```
#include "lnklst.h"

ccInt myFunction(ccInt argc, char **argv)
{
    arg_info myArgsInfo = argv[argc];

    ccInt numThirdArgElements = myArgsInfo[2].argIndices;

    ...

    if (myArgsInfo[2].argType != int_type)  return 1;      // do some type-checking

    ...
}
```

2.5.2 Cicada's `userfn` source files

Cicada interfaces with a user's C/C++ code through the file `userfn.c`. Usually this is the only Cicada source file we need to deal with. The cleanest way to incorporate C/C++ functions is to include their header files at the top of `userfn.c`, reference them in the `UserFunction[]` array, and add the source files to Cicada's `Makefile`.

To give an explicit example, let's assume our C function `myFunction()` is stored in a source file called `myFunctionFile.c`, and prototyped in `myFunctionFile.h`. First, we open up `userfn.c` (since we will be using the C version of Cicada). At the top we include the header file.

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

#include "myFunctionFile.h"
```

Next, we add a new entry to the `UserFunctions` array defined in `userfn.c`. The name of the C function is `myFunction()`, but earlier in our examples we invoked it by writing either `call("myFunctionInC", ...)` or `$myFunctionInC(...)`, so its Cicada name is `myFunctionInC`. The altered `UserFunctions` array (which already contained two entries) now reads:

```
userFunction UserFunctions[] = { { "pass2nums", &pass2nums }, { "cicada", &runCicada },
                                        { "myFunctionInC", &myFunction } };
```

We are done changing `userfn.c`, but we still need to modify the `Makefile` so that our C code will get compiled into Cicada. First we add a new object file to the end of the `OBJ` definition in `Makefile`:

```
OBJ = lnklst.o ... ccmain.o myFunctionFile.o
```

Second, we define our new object file by adding the following line to the very end of `Makefile`:

```
myFunctionFile.o: userfn.h myFunctionFile.h myFunctionFile.c
```

All of the Cicada source files and `.cicada` files should be in the same directory together with `Makefile` and our `myFunctionFile.c` and `myFunctionFile.h` files. Finally, from the command prompt we can navigate to that directory and type `make cicada CC=gcc`. Assuming that both the `make` tool and a C/C++ compiler like `gcc` are installed, we will then have an application named `cicada` with `myFunction` as embedded as a command.

As mentioned earlier, we can run Cicada from within another application by including `ccmain.h` and making the following function call:

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

where `argc` is either 0 if Cicada is to run the script `start.cicada` in the default directory, or 1 if the script to run is a file whose name and path are stored in a C string at `argv[0]`. (Note that `argv[0]` is not the pathname that the command prompt traditionally passes to a C program.)

2.5.3 A minimal wrapper

To finish off, we'll show how to a write a pretty generic Cicada wrapper for our C function. This will let us run our C code by typing something like

```
z = f(x, y)
```

rather than

```
$myFunctionInC(x, y, z)
```

The wrapper will also allocate data storage for our function, and prevent us from crashing the C code by passing in bad arguments.

Don't ask me what our C function does, but here is a pretty generic wrapper for it.

myFunctionWrapper.cicada

```
f :: {
    x :: char
    y :: double
    answer :: string
    calc_table :: [] int
    error_code :: int

    params :: {
        calcSize :: int
        doRecalc :: bool    }


    code

    params = { 100, true }
    if trap(
        { x, y } = args            | mandatory arguments
        (params << args)()         | optional arguments
    ) /= passed  then (
        print("usage:  [string] = f(char, double [; calcSize/doRecalc = ...])\n")
        return      )

    calc_table[^params.calcSize]

    error_code = $myFunctionInC(params.doRecalc, x, calc_table, y, answer)

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

We load our wrapper by going to Cicada's command prompt and typing:

```
    > run("myFunctionWrapper.cicada")
```

Here are some examples of function calls we can make once we've loaded our wrapper:

```
    result := f('a', 5)
    print( f('z', 5.78; doRecalc = false) )
    result := f('a', 5; calcSize = that*2, doRecalc = false)
```

We'll quickly go through the different parts of our wrapper function. First, everything before the `code` command just defines the variables used by our function. These include the input and output arguments, optional parameters and even a calculation table used internally by our C code. Importantly, by explicitly defining variables that we pass to the C code we ensure that our function call will communicate with C properly. For example, when we write `f('a', 5)` the integer argument 5 will be converted to a floating-point number before handing it off to C, because the input variable `x` is defined to be of type `double`.

The executable part of `f()` begins *after* the `code` marker. The first bit of code reads the arguments of the function. (We enclose the actual reading commands in a `trap()` statement so that we can fail gracefully with an error message if the function wasn't called properly.) Notice the two sorts of argument that the user passes. The mandatory arguments `x` and `y` are copied straightforwardly from the predefined `args` variable. The optional arguments, stored in `params` with the default values of 100 and `true`, can be changed using a very peculiar Cicada trick: `f()` *runs its own arguments* , as a function, inside of its own `params` variable. How this works is explained in the next chapter. For now, we'll just point out that in calling `f()` we separate the mandatory and optional parameters using a semicolon (which is effectively a `code` command), and use ordinary scripting commands separated by commas to change the optional parameters.

3 Cicada scripting

3.1 Building and running Cicada

Cicada is distributed as either a set of C or C++ source files. The C version of Cicada is designed to embed (or be embedded in) the user's C code, whereas the C++ version merges with a user's C++ code. See the previous section for details on how to do do this.

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 Cicada using `make`, put *everything*—source files, header files, `Makefile` and '.cicada' files—into the same directory; navigate to that directory from a command prompt; and then type "`make cicada CC=gcc`" for a C program, or just "`make cicada`" for a C++ program. This should produce a `cicada` executable in that same directory.

Once Cicada has successfully been compiled, we can run it by opening up a command terminal, navigating into Cicada's directory and typing "`./cicada`" (UNIX and Mac), or just "`cicada`" (PC/Windows). Cicada should present a command prompt, looking like this:

```
>
```

3.2 Basic Cicada syntax

Numbers, characters and string constants have pretty much the same syntax in Cicada as they do in C: characters are flanked by single quotes (`'C'`), strings by double-quotes (`"my_string"`), and numbers are read as either integer or floating-point depending on how they are written. Likewise, numeric expressions look the same between C and Cicada, with the exception that Cicada provides an exponentiation operator '`^`': for example `2^3` gives 8. In Cicada the standard six trig functions along with `log()`, `floor()` and `ceil()` can be used without including a math library.

The usual way to define variables in Cicada is to use the '`::`' operator, as in:

```
x :: int
```

Here `x` was the variable and `int` was the type. The other possible allowed types are `double`, `char`, `string` and `bool`. All Cicada variables *must* be defined before they can be used.

In order to define an array, add the size of each dimension in square brackets just before the type definition. For example,

```
myTable :: [5] [7] double
```

defines a two-dimensional (5x7) array of floating-point numbers. The syntax for accessing an array element again uses square brackets, in the same way as in C:

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

The only difference is that in Cicada array indexing begins at 1.

It's usually convenient to define several variables or arrays of the same type together in one long command. For example:

```
x :: y :: z :: int
table1 :: table2 :: [5] double
```

This example script shows basic usage of variables and arrays. It's easiest to type this into a file: say, "pythagoras.cicada"..

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

sides :: [2] double
hypotenuse :: double
response :: string

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

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

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

(The first line in this example is a comment, because of the vertical bar `|`. `print()`, `input()` and `read_string()` are three of Cicada's 30-odd built-in functions.) To run our script from Cicada's command prompt we type

```
run("pythagoras")
```

If we want to define our own functions we again use the `::` operator, but with curly braces containing the function code in place of a variable type. A standard Cicada function consists of three parts: 1) variable definitions; 2) a `code` command; 3) the executable code of the function. Unlike C functions, a Cicada function doesn't predefine either its arguments or its return value. Its arguments are accessed through an `args` variable, and they can be overwritten by the function. The `return` command works the same way as in C. Finally, the basic syntax for *calling* a function is the same as in C, although in Cicada fancier kinds of function calls are also possible.

Here is a more elaborate version of our previous example that uses functions, as well as `if` statements and `for` loops whose syntax is a bit different from C.

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

GetSide :: {
    response :: string
    side_length :: double

    code

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

    if side_length > 0  then (
        return side_length    )
    else  (
        print("Side length must be positive\n")
        exit    )
}

sides :: [2] double
hypotenuse :: double
counter :: int
```

```
for counter in <1, 2>  (
    sides[counter] = GetSide(counter)    )
hypotenuse = (sides[1]^2 + sides[2]^2)^0.5

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

Cicada also has `while` and `loop ... until` loops. Notice that although the equality test '`==`' is the same as in C, the not-equal test uses the symbol '`/=`' which is different from C. There is no 'goto' statement.

To exit Cicada type "`exit`" at the command prompt. If there is no command prompt then Cicada is probably stuck (e.g. in an infinite loop), in which case we need to hit Ctrl-C (UNIX/Mac) or Ctrl-Alt-Delete (Windows).

3.3 Expressions

At its most basic level, a Cicada script consists of *commands* which are usually written on separate lines:

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

although they may also be separated using commas:

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

(Commas are useful when entering a multi-line command, like a `for` loop, from the command prompt.)

Unlike in C, we don't need a semicolon at the end of each command – the line break automatically does that for us. If we don't want the line break to mark the end of the command, then we need to use the line-continuation symbol '`&`':

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

This even works from the command prompt (just make sure that the `&` is the *very* last character on the line – no trailing spaces). A `&` can appear between any two operators in a command, but not within an operator, name, symbol or string.

Each command consists of variables and constants, glued together by operators. For example, the command `a := 2*7 + 9` contains the variable `a`, three integer constants 2, 7, and 9, and three other operators: `+`, `*` and `:=` which both defines a variable and sets its value. The operators are grouped in the following way:

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

That is to say, the multiplication has the highest *precedence* (i.e. it is done first), followed by the addition, and lastly by the define-equate operation which sets `a` to the final value. If we want to change the default grouping of operators, we use parentheses just as in C: for example `a := 2*(7 + 9)` would do the addition first.

Table 2 gives the precedence levels of all Cicada operators. The final column determines how operators of the same precedence level are grouped. For example, the division operator falls within precedence level 11 which has left-to-right grouping, meaning that `8 / 4 / 2` is equivalent to `(8 / 4) / 2`. On the other hand, the define and equate operators all have right-to-left grouping, so `a := b = c = 2` is equivalent to `a := (b = (c = 2))`.

precedence	commands	symbols	grouping
1	command breaks	`\n , ;`	right to left
2	commands	`return remove` `if for while loop`	N/A
3	logical and, or, xor	`and or xor`	left to right
4	logical not	`not`	right to left

5	define/equate
	forced equate
		`::  ::@ := :=@ @::  *::` `= <- =@`
		`=!  <-!`
			right to left

6	comparisons	`== /= > >= < <=`	N/A
7	substitute code	`<<`	left to right
8	array type	`[]`	right to left
9	inheritance	`:`	left to right
10	add, subtract	`+ -`	left to right
11	multiply, divide, mod	`* / mod`	left to right
12	negate	`-`	right to left
13	raise to power	`^`	left to right

14	function calls
	step to member
	step to index/indices
	code number
		`()`
		`.`
		`[] [<>] [+] [-] [^]`
		`#`
			left to right

15	backstep	`\ parent`	left to right

Table 2: Order of operations

All of the information in Table 2 comes from the `cicada.c` source file. That means that anyone can change the Cicada language by just editing that file (see Chapter 4). In particular the `cicadaLanguageAssociativity[]` array controls the left/right grouping of operators within a precedence level.

3.4 Variables

Cicada has three basic variable types: primitive variables which store data, composite variables which contain other variables, and array variables which contain many identical copies of another variable. All variables must be be created (using one of the define operators) before they can be used. Once created, any variable can have its data overwritten (there are no constant types in Cicada) and can even have its storage reassigned so that it 'points' to another variable. In Cicada there are also several predefined variable names, such as `args` which stores a function's arguments.

3.4.1 Primitive variables and constants

Cicada has five primitive variable types: `bool`, `char`, `int`, `double` and `string`. The first four of these types occupy a fixed amount storage which may depend on the machine (for example, the `int` type generally is the same standard integer as C's `int` type). However, `string` variables can store strings (character arrays) of any length.

In order to define a variable, use the define '`::`' operator. Several variables of the same type can be defined on the same line. As an example, here we define three floating point variables and one string.

```
height :: time :: g :: double
units :: string
```

All primitive variables get initialized to some starting value when first created. That value is: 0 for numeric variables (`int`s and `double`s); the null character for `char` variables; `false` for Boolean variables;

and an empty (0-character) string for `string` variables.

To change the value stored by a variable, use the assignment operator, which can be written either '`=`' as in C or '`<-`'. Here we'll use the C convention.

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

In this case we just assigned a constant value to each variable. Numeric constants are obviously just numbers. String constants are inside double quotes, whereas character constants just use single quotes. The only two Boolean constants are `true` and `false`. Of course, all primitive types can also be assigned using more complex expressions: for example `isBigger = a > b` sets the Boolean variable `isBigger` to either `true` or `false` depending on the values of numeric variables `a` and `b`.

For the sake of brevity, we can define and set variables in a single step using the `:=` operator. So our last two examples could have been condensed a bit:

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

Cicada has to somewhat guess about numeric types when we do this: is '`g`' an `int` or a `double`? Because of the decimal point Cicada guesses `double`. Just know that if we had rounded to `g := 10`, then '`g`' would have been defined as an integer variable.

By default, Cicada's `bool` is stored internally as a byte, the `int` is a basic integer and the `double` type is a double-precision floating point. (See Table 1 on page 16.) However, these can be reassigned to different C types: for example some applications might prefer Cicada's `double` type to be C's `long double` data type. To change a data type, open up the `lnklst.h` source file and change the appropriate `#typedef` (see the 'C type name' column in Table 1).

3.4.2 Composite variables

The second type of variable in Cicada is a composite variable, equivalent to a structure variable in C. Composite variables are collections of *members* defined inside of curly braces, just like a struct variable in C. Member definitions can go on their own lines, but make sure the opening brace is on the first line.

```
StreetAddress :: {
    number :: int
    street :: string
}
```

If we don't care about the member names we can leave them out.

```
StreetAddress :: { int, string }
```

`StreetAddress` is a proper variable with its own storage, but for convenience Cicada allows us to also treat it as a new data type.

```
PetersPlace :: StreetAddress
```

In general any Cicada variable `A` can be used to define other variables `B` and `C`, which just copy `A`'s type definition (although not `A`'s data).

We access members of a composite variable the same way in Cicada as we do in C: using a period. For example (using the original definition with member names):

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

Composite variables may be nested inside one another, either using previously-defined types or by nesting curly braces. Here is an example showing both methods.

```
FullAddress :: {
    first_line :: StreetAddress
    second_line :: {
        city :: state :: string
        zip :: int
    }
}
```

In this case, we need to use a '`.`' twice to access any of the primitive fields.

```
FullAddress.first_line.number = 357
```

Notice that members of composite variables can also serve as data types.

```
GeneralWhereabouts :: FullAddress.second_line
GeneralWhereabouts.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
```

By the end `PaulineAddress` will have a three members including one named `country`, and the `street` member of `first_line` will be missing. One thing to be aware of is that if we ever use a modified variable like `PaulineAddress` to define another variable:

```
PaulineOldAddress :: PaulineAddress
```

the new variable is always defined using the *original* definition. So `PaulineOldAddress` will not have a member named `country` and it will have `street` inside of `first_line`. This can lead to problems with the define-equate operator:

```
> PaulinesDogAddress := PaulineAddress

Error:  type mismatch
```

The problem was that the define half of `:=` constructed `PaulinesDogAddress` based on the original type definition of `PaulineAddress`, while the equate half of `:=` copies data from the fields that are actually present between the two variables. Oops.

Assignment and equality-testing work with whole composite variables just like they do with primitive variables:

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

We were allowed to assign and compare `Tom` and `Bob` because they have the same { number, string } structure.

Numeric operations and comparisons do not work with composite variables, even if all their members are numeric. Expressions like `Tom+Bob` and `Tom > Bob` will always cause an error.

3.4.3 Array variables

The third type of Cicada variable is an array variable, which is simply N copies of some other variable type. To define an array we write N inside square brackets *before* the type we want to copy. Multi-dimensional arrays are just arrays of arrays. Here are some examples.

```
numDays :: [12] int
checkerboard :: [8][8] bool
coordinates :: [] { x :: y :: double }
```

Notice that `coordinates` was defined without a specific array size, so it will start off as a size-0 array until it is manually resized. Arrays defined using empty brackets (or equivalently with the `[*]` symbol) are given extra storage so that elements can be added easily. Use empty brackets for arrays that will be resized often.

We access array elements using square brackets *after* the array name, just as in C. The last element can be accessed using the keyword `top`. To access a range of elements a through b, write `[<a, b>]`. To access all elements we can simply write `[*]` or `[]`.

```
numDays[1] = 31                                  | assign a single element
numDays[<2, 4>] = { 31, 31, 30 }                 | assign 3 elements
checkerboard[1][1] = checkerboard[1][3] = true   | assign 2 elements
checkerboard[3] = checkerboard[1]                | assign a whole row of elements
checkerboard[top] = checkerboard[top-2]          | ditto for the 6th and 8th rows
coordinates[*].x = coordinates[*].y              | assign all 'x's
coordinates[*].y = -1                            | assign all 'y's (scalar-to-array copying)
```

Strings are effectively character arrays, and we can use array operators to access either single characters or ranges of characters. The single-element operator `[n]` returns a character, whereas the multi-index operator `[<a, b>]` returns a character array (equivalent to a string). Given two strings `s1` and `s2`, the following commands are legal: `s1 = s2[<3, 4>]`, `s1 = s2[<4, 4>]`, `s1 =! s2[4]`; however `s1 = s2[4]` is illegal since Cicada does not allow `char`-to-`string` assignment. Note that array operators only work on string variables, not string literals, so `"abc"[2]` will cause an error.

Cicada has a technical restriction on the elements we can access simultaneously in multi-dimensional arrays: their memory has to be *contiguous*. Internally, the *last* index counts consecutive elements in memory; then the next-to-last index increments; etc. So the elements of some 2-dimensional array `A` elements are `[1][1]`, `[1][2]`, ..., `[2][1]`, `[2][2]`, ..., etc as shown in Figure 1. Here are some legal and illegal expressions involving a matrix `A :: [3][4] { x :: ... }`:

```
A[<2, 3>]              | legal
A[<2, 3>][<1, 4>]      | legal
A[<2, 3>][2]           | will cause error
A[2][<2, 3>]           | legal
A[1][2]                | legal of course
A[<2, 3>][*].x         | legal -- x is stored separately from other members
```

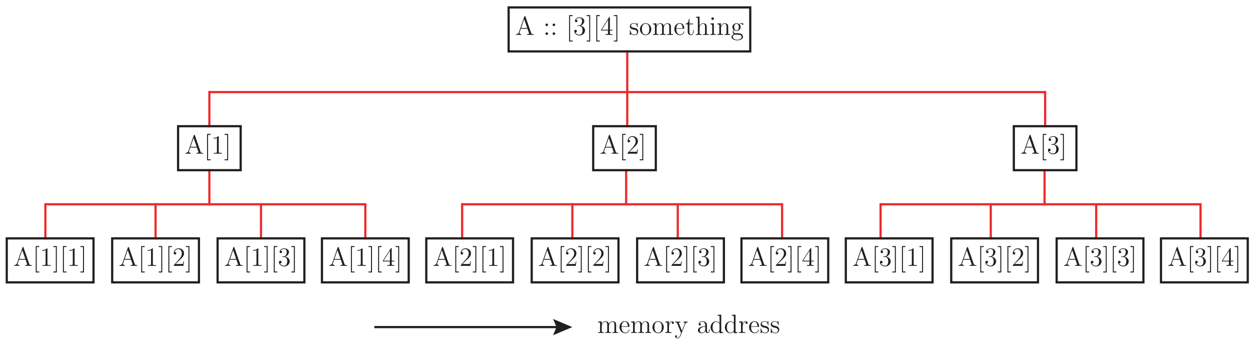


Figure 1: Order of elements in a 2-dimensional array '`A`'

When we address part of an array using two or more `[*]` operators, Cicada loses track of how many elements were in each dimension and pretends that it is looking at a one-dimensional list. For example, if we define:

```
    grid :: [5][5] int
```

then both `grid` and `grid[*]` refer to a 2-dimensional array, but `grid[*][*]` is effectively a single 1-dimensional list of 25 elements. This is another technical limitation of Cicada, and it may change in the future.

3.4.4 Resizing arrays (and composite variables)

All arrays in Cicada 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 size of a single dimension of the array, as in

```
    A[^9]
```

The effect can be either to increase or reduce the size of the array.

Another way to resize an array is to use the all-indices operator `[]` or `[*]` for the *last dimension* of an array that is being copied to. The last dimension of the array will be resized only if necessary to prevent a mismatched-indices error. So

```
    A[*] = B[*].x
```

should always work, whereas

```
    A[<1, 5>] = B[*].x
```

will only work if `B` has 5 elements.

Finally, we can insert an array index somewhere in the middle of an array using the `[+...]` operator, or the `[+<..., ...>]` operator for multiple indices. We can also delete array indices using either the `[-...]`/`[-<..., ...>]` operators or the `remove` command. Here are some examples:

```
    myArray[+5]                        | insert a new element before index 5
    myArray[+<3, 6>]                   | insert 4 new elements before index 3
    myArray[+<top+1, top+9>] = 17      | add 9 new elements at the end, and set them all to 17
    myArray[-13]                    | delete array element #13
    myArray[-<2, 4>]                | delete array elements 2, 3 and 4
    remove myArray[<2, 4>]          | same -- nix elements 2, 3 and 4
```

New array elements are always initialized in the same way as new arrays are: for example if it is a numeric array the new elements are set to zero.

All of the operators for accessing, adding and deleting array elements also work on composite variables. The difference is that, with composite variables, these operators access, add or delete members instead of array elements. For example, if we define

```
    threeNums :: {
        a := 2
        b := 5
        c := pi
    }
```

then `threeNums.b` is the same as `threeNums[2]`. If we want to delete the third member we can type either `remove threeNums.c` or `remove threeNums[3]`. And we can add two members *between* `a` and `b` by typing `threeNums[+<2, 3>]`. Those two new members will have no name, and initially will also have no data (see the forthcoming section on the void). If we enter '`threeNums`' after all these operations Cicada will print `{ 2, *, *, 5 }`, showing that `threeNums.b` is now `threeNums[4]`. In contrast, the old way of adding an extra member, say by typing `threeNums.d :: int`, sticks the new member `d` at the end of `threeNums`.

We can never reference multiple elements of a composite variable except when removing them. So `threeNums[<2, 3>]` is not allowed.

3.4.5 This and that

At any point in a script there are several objects (variables or functions) that can be referenced using a built-in keyword. The first of these special keywords is '`this`'. In most examples from this chapter `this` refers to the workspace variable – the top-level space where the user's variables live. But when a function is running, '`this`' refers to the function. For example, if we define the function

```
    f :: {  int; this = args, return this  }
```

and type f(5), our function will set its internal integer variable to 5 and return itself: `{ 5 }`. In general '`this`' is whatever object contains the code that is currently running.

Be warned that there are two situations where '`this`' may not point where we expect. 1) If we use the `this` variable inside *any* curly braces, as in

```
    myVar :: {
        a := 2
        b := 5
        c := this[1]
    }
```

then it refers to what is inside the curly braces, not the workspace (so in this case `c` will be set to 2). 2) If '`this`' appears inside the arguments of a function call, then it refers to any function arguments we have already passed. The reason is a bit more complicated and will be explained later, but for example if we write

```
    f(2, 5, this[2])
```

then we are effectively calling `f(2, 5, 5)`. In some cases this can be annoying: for example if we want to print the number of workspace variables by typing

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

we will instead print '1' which is the number of arguments of the top() function. What we really want to do is back out 2 levels, first to the arguments of print() and then to the workspace, by typing

```
print(top(parent.parent))
```

parent is the second predefined keyword, and refers to the object in the search path just before this. A shorthand for parent is a backslash character, so the last example could also have been written

```
print(top(\.\))
```

A third keyword which appears only to the right of an assignment operator (= or <-) is called 'that'. In its proper context, 'that' refers to whatever was on the left side of the assignment operator. 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
```

The final keyword, called args, refers to a function's arguments, which is almost always a composite variable. Functions are explained in more detail later, but here's a quick example:

```
I :: {
    code
    return { args, args[2] }
}
```

If we type the command I(1, 2, 3), our function I will return { { 1, 2, 3 }, 2 }.

3.4.6 Aliases

C has pointers; Cicada has *aliases*. An alias is a member that shares its data with some other member: they lead to the same variable. In Cicada a member and its alias are *exactly* on he same terms: Cicada doesn't even know which was the original member.

Here is how aliases are defined:

```
a := 2       | will default to an int
b :=@ a      | alias #1
c :=@ b      | alias #2
```

'b' is now an alias to 'a', and 'c' is now an alias to 'b' and therefore also to 'a'. If we were to now print(a), print(b) or print(c), we would 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 the new number 3 to be printed.

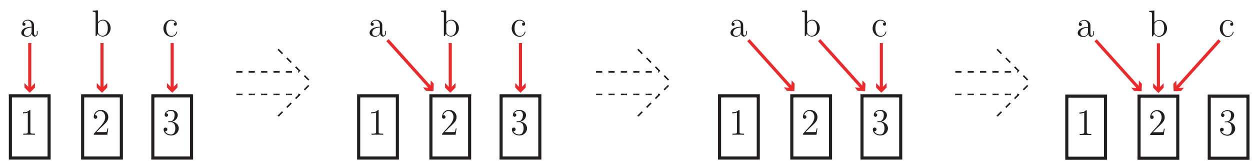


Figure 2: Aliasing of members (letters) to variables (boxes)

Any variable can become an alias for any other variable *of the same type* (or a derived/inherited type), by means of the equate-at operator '`=@`'.

```
a :: b :: c :: int
{ a, b, c } = { 1, 2, 3 }
a =@ b                  | 'a' now points to the variable storing '2'
b =@ c                  | 'a' STILL points to the variable storing '2'
c =@ b =@ a
```

The tricky bit is the fourth line: since member '`a`' has been aliased to member '`b`', does the command `b =@ c` now drag member `a` along with it? The answer is no: aliasing binds one member to another member's variable, not to the other member itself. If we follow through all the acrobatics, we find that members `a`, `b`, and `c` all end up referring to same the variable storing the number 2, as shown in Figure 2. (The other two variables are now inaccessible and will eventually be cleared from memory.)

Cicada clearly makes an important distinction between members and variables. Until now we have been rather sloppy on this point, so we shall be more careful from now on. A member is a named object in Cicada: the name of a variable, or a field in a composite variable. The variable itself is the data that the member refers to. In Cicada these two objects are entirely separate, and there is no reason to require a one-to-one correspondence between them.

Just as the data-copying operator '`=`' or '`<-`' has a data-comparison counterpart '`==`', so the member-reference-copying operator '`=@`' is mirrored in a reference-comparison operator '`==@`' which tests to see whether two members point to the same object. (If the left side spans a range of array elements then the right side must also span exactly those same elements in order for the test to return true). Finally, the are-references-not-equal operator '`/=@`' is just the logical negation of '`==@`'.

If we so prefer, we can write whitespace just before the '@' of any of the aliasing operators.

```
if a == @b  then c := @b
if a /= @b  then a = @b
```

The extra space makes clearer the analogy with the data-copying operators: both `a = @b` and `a = b` cause member '`a`' to equal member '`b`', though by different means. By the way an `@` symbol by itself is meaningless; it is only the last character of these aliasing operators.

Jamming Arrays can also be involved in aliasing. There is a potential complication, but on the surface it is straightforward. For example, we ordinary (non-array) members can alias single array elements.

```
array_1 :: [5] double
array_2 :: [10] double
oneEl :: double

oneEl = @array_1[4]
oneEl = @array_2[top]
```

Likewise *complete* arrays can alias other arrays, or parts of larger arrays. (Cicada doesn't let us re-alias just *some* indices of an array since that would fragment its memory.). So we can write

```
array_1[*] = @array_2[<4, 8>]
```

as long as `array_1` had 5 elements to begin with (aliasing can't resize an array); however we *cannot* write

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

The potential complication is that arrays having multiple members pointing to them can become 'jammed', meaning that they cannot be resized. The reason is that resizing one of its members would also force a resize of the other member which may be in an unrelated part of code, and this is not allowed. Here is an example.

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

array_1[*][^12]        | legal - indices 11-12 aren't aliased by array_2
array_2[^12]           | no, this would cause problems
remove array_1[*][5]   | not legal -- would remove 2nd column of array_2
remove 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, unnamed references to objects (such as elements of sets and function arguments) never jam arrays.

```
al := @my_array[<4, 6>]    | jams elements 4-6
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. An unjammed token becomes unusable until it is redefined, usually when the command defining the token is rerun.

3.4.7 The void

There is one last way to define a variable, and that is not to define it as anything at all. There are two ways of writing this:

```
var1 :: nothing
var2 :: *
```

`nothing` and `*` are synonyms for the void operator. If `var1` is defined as void then we will get a void-member error if we try to access its data. In fact, one of the few things we can do with a void member is to test whether it is void or not, using the reference-comparison operator `==@`.

```
if var1 == @nothing  then &
    print("out of order..\n")
```

One way to make use of a void member is to redefine it.

```
var1 :: nothing
var1 :: [1000] string
```

We could have redefined `var1` as anything: a primitive or composite variable, function, etc. But once it's been redefined once `var1` cannot be redefined again except to the same type or a derived type (see inheritance).

What is the use of defining void-typed variables (aside from the fact that they don't take up space until we redefine them)? The main reason is that 1) members defined this way have no type until there are redefined, and 2) members without a type can alias any other member, because in a sense all data types are derived from the void. This works as long as we don't redefine the alias (i.e. we have to just use the `=@` operator, not `:=@`).

```
any_var :: *
x :: int
y := "some string"
z :: { double, int }

any_var = @x
any_var = @y
any_var = @nothing
any_var = @z
```

The fact that the void is the universal parent type also explains why void members can be redefined. Cicada always permits an existing member's type to be specialized—restricted to a subtype of its original type—but never changed to an unrelated type. And we cannot un-specialize: an `int` member cannot be turned it back into `nothing`, except by removing and reallocating the member.

There are actually two uses of the word 'void' in this document that are important to keep logically separate. A void member is one that has no storage space. But a void-*typed* member is one with essentially no type restriction on what it can point to. A definition like `a :: *` does *two* things: it defines a member '`a`' having no type, and it therefore neglects to give '`a`' any storage space.

Cicada also distinguishes between member types and variable types. For example, after `any_var` was aliased to `x`, did `any_var` have a null type or was it an `int`? Well, that depends on whether you are asking about the member's type (which was void), or the type of the variable it points to (an integer). Both members and variables have types, and in general they may be different. A member's type specification determines which variables it is allowed to point to. The rule is that a member can only target variables of the same type, or having a sub-type derived from its the member's type. (Well strictly speaking there is an exception: any member can be aliased to the void to free up its storage space, regardless of its type. I guess the void plays by its own rules.)

3.5 Other define and equate operators

The difference between a member's type and the type of its target variable motivates two further define operators.

The variable-define operator '`@::`' is identical to ordinary define (`::`), except that it only sets the target variable's type, while leaving the member's type unchanged. That means that if it is used to define a brand new member, that member's type will be void. For example, here we use this operator to define and then redefine a member with different types of storage, and then alias it to another variable of a different type.

```
theVar @:: int
...
theVar = @nothing    | need to unlink before creating a new variable
theVar @:: string
...
theBool :: bool
```

```
    theVar = @theBool
```

Notice that the second `@::` operator was forced to make a new variable because `theVar` had been unlinked (aliased to the void) on the previous line. Had we left out the unlinking command, Cicada would have tried to redefine the existing `int` *variable* as a `string`, causing a type-mismatch error.

The member-define operator '`*::`' is the counterpart to variable-define: it only acts upon the member, without affecting any variable that member may target. Its symbol reflects 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: the member will point into the void. So the following code will cause an error:

```
    myNum *:: int
    myNum = 5      | will cause an error
```

which could have been avoided had we written `myNum :: int` or `myNum @:: int` between the two lines, in order to construct the integer variable behind the member. It has to be an integer variable in order to be compatible with member `myNum`'s type.

There are actually 256 operators in the define family, most of which cannot be scripted directly unless we modify the language. See Chapter 4.

Finally, there is another assignment operator called 'forced-equate', and given the symbol '`=!`' or '`<-!`'. While equate copies data between variables whose data structures match or are compatible (e.g. `{ int, string }` to `{ double, string }`), forced equate copies between variables having the same memory storage size. (Having a string in the destination variable makes the storage requirement somewhat elastic.) 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, with 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 Cicada 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.

Here are some examples showing the difference between these two operators:

```
    var1 :: { int, bool }
    var2 :: { double, bool }
    var3 :: { int, string }

    var1 = var2    | OK
    var1 = var3    | type-mismatch error

    var1 =! var2   | unequal data size error
    var1 =! var3   | unequal data size error unless string has 1 character
    var3 =! var1   | OK, surprisingly!
```

The last line works because forced-equate resizes `var3`'s string as needed to soak up any extra bytes from `var1`.

3.6 Loops and `if` blocks

Cicada has five commands for controlling the program flow: `if` statements, `for` and `backfor` loops, and `while` and the related `do` loops.

3.6.1 `if`

There are several differences in syntax between an `if` statement in Cicada and one in C. Firstly, the logical test is between the `if` and a `then` keyword, rather than being inside parentheses.

```
if a == b  then print("They're equal.\n")
```

Secondly, some of the logical operators are different. '`and`', '`or`', '`xor`' and '`not`' are all written out in words, and the is-not-equal operator is `/=`, not `!=`. Cicada also has the reference-comparison operators '`== @`' and '`/= @`' (see *aliases*).

```
if a == @nothing and b /= @nothing  then print("a, but not b, is void.")
```

Third, since the entire `if` statement is one single command and (unlike in C) a line break *ends* commands, if we want to break it over several lines we need to either use a '`&`' which continues lines, or else parentheses *beginning on the line* of the `if` statement.

```
if a == b  then &
    print("They're equal.\n")

else if a /= b  then (
    print("They're different.\n")
)
```

Finally, the latter method shows how to write multi-line `if` statements: using parentheses instead of curly braces. (Technical note: while curly braces actually work for most `if` statements, they also produce a new variable space, so defining new members or using the keyword `this` won't do what you expect. Just use parentheses.)

```
if a /= @nothing and b /= @nothing  then (       | avoid a crash
    if a == @b  then print("They're the same variable!\n")
    else if a == b  then print("They're equal.\n")
    else  print("They're different.\n")      )

else  print("At least one of these members is VOID!")
```

3.6.2 `while do` and `loop until`

Cicada's `while...do` loop is pretty similar to C's `while` loop, except that the logical condition goes between the `while` keyword and a `do` keyword instead of inside parentheses. Also, the logical operators are somewhat different and we use parentheses to group multi-line commands, in the same way as with the `if` statement.

```
while input() /= "pleeez"  do print("what's the magic word?  ")
while input() == "thank you"  do (
    print("you are welcome!\n")
    print("anything else?  ")
)
```

There are two differences between Cicada's `loop` loop and its `while` loop. First, the break condition of a `loop` loop is evaluated at the end of the loop, so it always runs at least once. Second, it is a `loop...until` loop, not a `loop...while` loop, so it keeps looping as long as the logical condition is *not* satisfied.

```
attempt := 0
loop  (
    print("Thank you!\n")
    attempt = attempt+1
)  until input() == "you're welcome"

loop print("> ") until input() == "exit"
```

3.6.3 `for`

Cicada's `for` loop is simpler and offers less control than C's. Here is an example showing the syntax.

```
counter :: int
for counter in <1, 10-counter>  print(counter, " ")
```

Notice that we had to define the loop variable before the loop, as either an `int` or a `double`. The loop itself will print the numbers 1 through 5; when `counter` reaches 6 it will be greater than `10-counter` and the loop will stop.

As always, we can break a loop over two lines using a '`&`', and we can group several lines of code using parentheses beginning on the line of the `for` command. So we could have rewritten our code above in several different ways.

```
counter :: int

for counter in <1, 10-counter>  &
    print(counter, " ")

for counter in <1, 10-counter>  (
    print(counter)
    print(" ")
)
```

Unlike Cicada's predecessor (Yazoo scripting language) there is no `step` argument to control the increment of the counter variable. This is due to limitations of the new compiler. Of course there are a number of ways around this, one of which is to change the loop to a `while` loop and do the step manually. The loop in our example is equivalent to:

```
counter := 1
while counter <= 10-counter  do (
    print(counter, " ")
    counter = that + 1
)
```

3.6.4 `backfor`

This is a `for` loop that counts backwards (since `for` does not have this capability). For example,

```
counter :: int
backfor counter in <1, 9>  print(counter, " ")
```

produces the output `9 8 7 6 5 4 3 2 1`.

3.6.5 `break`

There is no `break` statement in Cicada. But there is a way to 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"  then return    )
}
print("finished with counter = ", counter)
```

This works because the braces define a new composite variable, and the code inside the braces is actually running inside of this variable. This hack causes problems in some cases. If we define a member, that member will be inside of this variable, which was probably not intended. The `this` keyword refers to this newly-defined variable, while `parent` now refers to the place we were before hitting the braces. Also, if we had been running a function, we cannot escape this function using a `return` statement within the bracketed code. On the upside, we have a lot of latitude in choosing where to put the braces: it doesn't have to escape from a loop, and if we do use them inside loops it doesn't have to fall out to the next-outermost `for` or `while`.

3.7 Sets

One object in Cicada that is not supported by C is the *set*. A set is basically a magical bag of assorted items—variables, functions, classes, even other sets—that also exist somewhere else. To define a set we write curly braces around a list of objects separated by commas (or end-of-lines). Here is an example where each object is in several places.

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

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

We typically access members of a set using square brackets (as if they were arrays). So after running the code, `Bob`, `men[3]`, `neighbors[2]`, `cleaning_schedule[1]` and `cleaning_schedule[3]` are all the same thing. If we set `men[3].needsSleep = true` then `Bob.needsSleep` and `neighbors[2].needsSleep` will also be `true`. Notice how the same object can even appear in several places in a set. Sets come handy when the objects have convoluted path names.

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

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

Cicada sets are pretty flexible in what they can store. 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 }
```

Here `collections[3]` is an inlined subset containing two objects. The fourth and fifth items are inlined constants. The `collections[6]` *is* also the third item, the subset containing `Patty` and `Don`, and notice that it had to be listed after the third item because otherwise `this[3]` would not have existed yet. `collections[7]` is `collections` itself – so `collections[7][7][7][2]` is just `neighbors`. Trying to print `collections` from the command line won't work because of the self reference.

The reason we access set elements by their index (as opposed to name) is that those set elements *have no name*, at least the way we defined them in our examples. In other words, `collections[1]` is effectively an alias for `men` but that doesn't mean that that set element has the name `men` – `collection.men` will cause Cicada to draw a blank. But there is actually a way to name certain elements of a set, and that is to manually define aliases for their respective objects. To illustrate, here we give a different way of defining `collections` which assigns names to members 2, 3, 4, 7 and 8.

```
collections :: {
    men
    neighbors := @parent.neighbors      | use same name for convenience
    peeps :: { Patty, Don }
    Herby := "Herbert"
    3.3,  this[3]          | set elements 5 and 6
    myself := @this
    zilch := @nothing
}
```

With the verbose definition we can write `collections.Herby` in place of `collections[4]`, although `collections[4]` is still perfectly legal.

Just as with composite variables, items can be added to and removed from sets after they have been defined. So if we want to shuffle "`Herbert`" to the end of the set we might write:

```
collections[top+1] := @collections.Herby
remove collections.Herby
```

(and those 2 lines could go in any order).

As with variables, we can use one set to define another. The following is legal:

```
newCollections :: collections
```

Importantly, `newCollections` is defined using the old set `collections`'s *original* definition. So even if we had rearranged the elements of `collections`, in the new set `Herbert` will be both at `newCollections[4]` and `newCollections.Herby`. We might as well have defined both sets on the same line:

```
newCollections :: collections :: { ... }
```

3.8 Functions

A basic Cicada function call looks exactly the same as a C function call.

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

But behind the familiar syntax roams what is probably the strangest beast in all the Cicada language. For one thing, Cicada functions are *objects* that live in ordinary heap memory, unlike their stack-dwelling counterparts in C. Cicada functions have no input type and no output type, and they can have several coding sections. Function arguments are hopelessly different between the two languages. These changes together allow Cicada functions to be used in some unusual ways.

3.8.1 Defining functions (properly)

Cicada functions are *objects*, like composite variables and sets (the analogy here is closer than you would think). So we define functions using the same operators we use for defining other objects. The function code goes inside curly braces beginning on the line of the definition. Here is an example.

```
SwapDigits :: {

    tensDigit :: onesDigit :: int

    code

    tensDigit = floor(args[1] / 10)
    onesDigit = args[1] mod 10

    return new(10*onesDigit + tensDigit)
}
```

The function shows us several new keywords. 1) A '`code`' marker (or semicolon) separates the function's variable definitions from its executable code. Before the `code` marker, the function's internal variables are defined in exactly the same way as global variables in the workspace are defined. 2) Function arguments are contained inside of an `args` variable, and we access them using the index operators `[]`. The function exits with the classic `return` statement, just as in C except 3) we enclosed the return value inside of a `new()` function call.

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

```
> print(SwapDigits(27))

72
```

From the command line we can also just type the function name and read off the output:

```
> SwapDigits(27)

72
```

Sometimes the automatic output can be annoying: the function may be performing an important task but we don't care about the return value. To throw away the return value, append a '∼' operator to the function call: `SwapDigits(27)∼`.

In Cicada we can treat functions as if they were variables, peek at all their internal members, and even fiddle around inside them. This is useful for debugging.

```
> SwapDigits.tensDigit            | did it do what we wanted?

2

> remove SwapDigits.onesDigit     |  let's see if it still works..
```

Finally, functions can define other functions:

```
> SD2 :: SwapDigits
```

We broke `SwapDigits()` when we deleted its `oneDigit` member, but `SD2` is defined using the *original definition* of `SwapDigits` so it will work just fine.

One situation that requires several copies of a function is recursion. This is because each nested recursive instance of the function requires its own storage. This is true whether a function `f()` calls itself directly or indirectly (`f()` calls `g()` calls `f()`). Here is an example that handles recursion properly.

```
factorial :: {

    numToReturn :: int

    code

    if args[1] == 1  then numToReturn = 1
    else  numToReturn = args[1] * (new_fact :: this)(args[1]-1)

    return numToReturn
}
```

Importantly, the definition `new_fact :: this` had to be in the *coding section* of `factorial()`. If we had put the definition `new_fact()` *before* the `code` marker, `factorial.new_fact` would have defined another member `factorial.new_fact.new_fact`, etc. until Cicada threw in the towel with a recursion-depth error.

Our last example might have handled recursion correctly, but it did something else quite wrong, as we find out by typing

```
> { factorial(3), factorial(4) }

{ 24, 24 }
```

The second call to `factorial()` overwrote the first! The reason is that our set consists of two aliases to `numToReturn`, whose value changes with each function call. The solution is to have `factorial()` redefine `numToReturn` each time it is run.

```
    return new(numToReturn)
```

The `new()` function is defined in `user.cicada` which pre-loads when Cicada is run interactively. If `user.cicada` wasn't loaded we have to redefine our return variable manually.

```
factorial :: {

    numToReturn :: int

    code

    (numToReturn =@ *) :: int        | redefining an unlinked member creates a new variable

    if args[1] == 1  then numToReturn = 1
    else  numToReturn = args[1] * (new_fact :: this)(args[1]-1)

    return numToReturn
}
```

Here is one unusual feature of Cicada functions:

```
f :: {
    num := 2

    code
    num = that + 1

    code
    return num   }
```

Two coding blocks—so there should be a way to access them both.

```
> f()


> f#2()

3
```

The code-number operator '`#`' lets us run the code beginning at the Nth `code` marker. By running `f()`, which defaults to `f#1()`, we incremented `f.num`. But execution stopped at the next `code` marker. There was no `return` statement, which is fine: the function returns `nothing`, just as it would had we written `return` or `return *`. In order to get a value back we had to run the *second* coding block which returned `f.num`. Incidentally, had we run `f#0()` we would rerun `f`'s 'constructor' and find `f.num` reset to 2.

Short functions like `f()` are sometimes easier to code on one line. To do this we use the fact that commas are equivalent to ends-of-lines, and semicolons are equivalent to `code` markers.

```
> f :: {  num := 2; num = that + 1; return num  }
```

3.8.2 Function arguments

Notice that the definition of a Cicada function doesn't specify any argument list. That means functions in Cicada are automatically able to handle different argument types, variable numbers of arguments, etc. Here is a simple function that can take any arguments whatsoever, as long as there are at least 2 of them.

```
> arg2 :: {  code, return args[2]  }


> arg2(5, 3+9, 'C')                             | return a number

12

> arg2(pi, { a := 3, b := 4; return a*b })      | return a function!

{ 3, 4 }
```

Cicada functions also don't specify return types, and as our last example shows a single function can return different kinds of values even from the same `return` statement. And of course different `return` statements within the same function can return different types of objects.

What happens if we stick a `code` marker (or semicolon) *inside the function's argument list*? When we try this experiment, we'll find out is that anything after the function arguments' `code` marker isn't created, or doesn't run, until the function *runs its arguments*.

```
PrintArgs :: {
    code

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

If we call

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

then the function prints

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

We can even put coding statements in the function arguments.

```
PrintArgs( " *** announcement *** ", code, print("I am an argument list.\n") )
```

which causes the output

```
Before running args:  *** announcement ***
I am an argument list.
Afterwards: *** announcement ***
```

In fact an argument list can run any code whatsoever. A few tricky points: if we define variables, etc. inside of an argument list then they will only exist inside that function's `args` variable; `this` inside the argument list refers to `args`, not the variable space where the function was called (that will be `parent`); and `return` inside of `args` only stops execution of `args`'s code.

If we can *run* the `args` list, surely we can also pass it parameters? Write a new function to test this:

```
> doArgs :: {  code, args(9, "lives"), return args  }


> doArgs( args, code, print(top(args), " arguments were passed\n") )

2 arguments were passed
{ {  } }
```

This example is complicated, particularly the function call (2nd command entered at the prompt). On this line, within the parentheses, '`args`' refers to two different things depending on which side of the `code` marker it falls on. Before the `code` marker, `args` holds the same value it had when the function was called, except wrapped in a second layer of curly braces. But *after* the `code` marker, '`args`' was the parameter list that ran `doArgs` passed to its arguments, containing the number 9 and the string `"lives"`. Each function call temporarily replaces the existing `args` variable with its own argument list; the old `args` comes back when each function exits.

Here's another example showing more explicitly how `args` changes across the `code` marker.

```
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 run `f()`.

```
> f("A")

A --> B
```

Focus on the line in which function `f()` calls function `g()`. The part of `args` *before* the `code` marker is built before `g()` is run, so here `args` still has its old value '`A`'. The part of `args` *after* the `code` marker is called by `g()`, so here `args` is '`B`'.

Finally, function arguments can contain `return` statements just like normal functions.

```
> doTwice :: {  num :: double, code, num = args[1], return args(args(num))  }


> doTwice(3, code, return new(args[1]^2))        | calculate (3^2)^2

81
```

To summarize, Cicada's function arguments are themselves functions. If the syntax for function calls were not so standardized the author would have chosen curly braces, so imagine:

```
***  doTwice{ 3, code, return new(args[1]^2) }  ***
```

This way of writing a function makes it clear how the arguments are just some function object that's ordinarily invisible, but appears as `args` when we run `doTwice()`. This syntax is not legal by the way — but if you want to allow it, just duplicate the line defining function calls in `cicada.c` and change the parentheses to curly braces.

3.8.3 Code substitution

Why would a function ever want to run its arguments? One good reason is that an `args()` call is the easiest way for a function to accept optional parameters. To do this, we must write the function so that all of its optional parameters (or aliases to them) are grouped into one composite variable. When the function is called, it will first set those parameters to their default values, then run `args()` *inside of the parameters variable* with the help of the code substitution operator '`<<`'. Here is an example.

```
RollDice :: {

    numDice :: total :: loopDie :: int

    params :: {
        numSides :: int
        isLoaded :: bool    }

    code

    { numDice } = args

    params = { 6, false }
```

```
        (params << args)()
        ...
    }
```

If we are happy with the default values of `numSides` and `isLoaded`, so we can leave them out.

```
    RollDice(5)
```

The function call only needs a coding section when we want to cheat or roll exotic dice.

```
    RollDice(5; isLoaded = true)
```

All members of `params` must be explicitly named. This would be an *incorrect* definition:

```
        numSides :: int
        isLoaded :: bool

        params :: { numSides, isLoaded }
```

The problem is that neither member of `params` has a name, so a function call can't easily change it. A member aliasing the `isLoaded` variable is not the same as a member named `isLoaded`.

In general, a code substitution `D << F`, involving composite variables or functions `D` and `F`, returns the data space of `D` along with the code of `F`. On the other hand, plain `F` returns both the data space and the code of `F`. The two basic properties of composite variables and functions are data and code, and the code substitution operator is the tool for separating and recombining these properties.

There are many uses for code substitution beyond optional function arguments. For example, `start.cicada` uses it to run the user's commands inside of the workspace variable. Code substitution can save a lot of typing when working with awkward pathnames. For example, instead of

```
    games.backgammon.RollDice.params.numSides = 20
    games.backgammon.RollDice.params.isLoaded = true
    games.backgammon.RollDice.params.dieColor := "green"
    ...
```

we can just write

```
    (games.backgammon.RollDice.params << {

        code

        numSides = 20
        isLoaded = true
        dieColor := "green"
        ...
    })()
```

Notice that we had to write all of our commands after a `code` marker, and run the substituted code as a function. That way the assignment commands will run inside of the `params` variable.

A code substitution is quite temporary: it only lasts as long as the immediate expression is being evaluated. The next time we run `RollDice()` it will be its normal self, except that its `params` variable will have a new member `dieColor`.

The code-number operator can be used to control which code is passed to the code-substitution operator. Here is a function that uses either one or two sets of optional parameters, depending on how it is called.

```
f :: {
    ...

    code

    params_1 :: { doMoreStuff := false, ... }
    params_2 :: { otherNum :: int, ... }

    (params_1 << args)()
    if params_1.doMoreStuff  then (params_2 << args#2)()

    ...
}
```

When we call this function, our arguments will contain 1, 2 or 3 coding blocks.

```
f(2, 5)                                             | don't doMoreStuff
f(2, 5; doMoreStuff = true)                         | doMoreStuff w/ defaults
f(2, 5; doMoreStuff = true; otherNum = 12, ...)     | change defaults
```

3.8.4 Search Paths

A Cicada function has access not only to its own members, but also to members defined at the global level (the workspace). More generally, the function can access any members along its *search path*, which goes through any object that encloses the function. To give a concrete example, we'll define a few functions scattered inside and outside of other objects.

```
allFunctions :: {
    ...
    stringFunctions :: { ... }
    numericFunctions :: {
        ...
        theNumber :: double
        raiseToPower :: { code, theNumber := args[1], return new(theNumber^power) }
}   }
raiseToPowerAlias := @allFunctions.numericFunctions.raiseToPower

allFunctions.numericFunctions.calcLog :: { ; theNumber = args[1], return new(log(theNumber)) }

powerOfThree :: { power := 3 }
```

The simplest case is the search path for the `raiseToPower()` function, which begins at `raiseToPower`, then passes through `numericFunctions` and `allFunctions` and ends at the workspace variable (blue path, Figure 3). This means that when its code references a member, it looks first in its own space for that member, then if it wasn't found it will back out to `numericFunctions` and look there; etc. all the way to the workspace if necessary. If it doesn't find the member by the end of its search path it will crash with a member-not-found error. This search path is used regardless of whether we called `raiseToPower()` or `raiseToPowerAlias()`.

On the other hand, the search path of the `calcLog()` function only touches `calcLog` itself and the workspace variable (the green path in Figure 3). The reason is that `calcLog` was defined straight from the workspace, *not* from the code that defined `allFunctions` or `numericFunctions`. So `calcLog()` cannot find

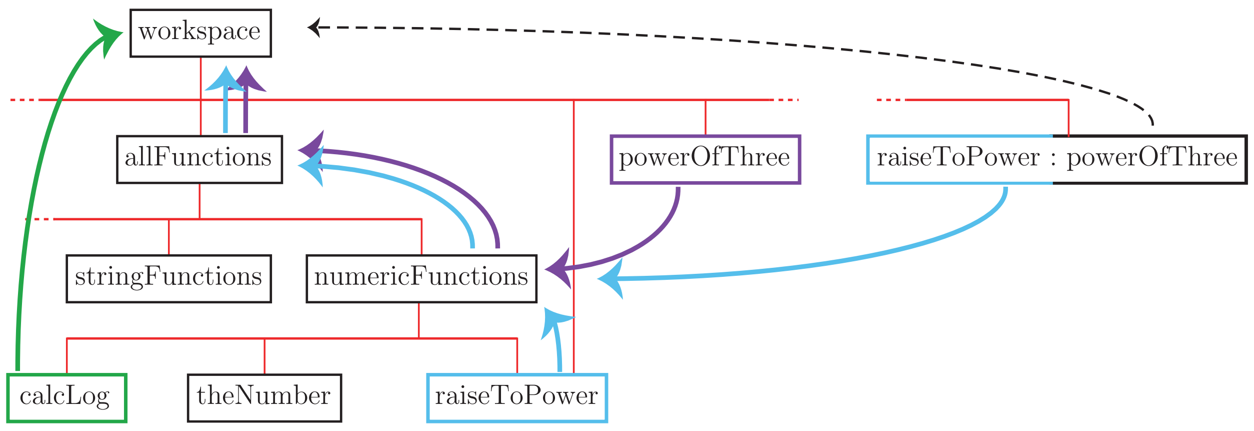


Figure 3: Search paths of various functions from an example in the text. For simplicity we've named each variable/function box by the member that defined it. Search paths are shown with heavy arrows: green arrows for `calcLog()`; light blue arrows for `raiseToPower()`; and purple arrows for `raiseToPower()` function *substituted into* `powerOfThree()`. A hypothetical function derived from `raiseToPower()` and specialized by `powerOfThree()` (using the inheritance operator) would have each half its code follow its respective original search path (blue and dotted black arrows from the box at right).

`theNumber`: it is a broken function. It's not that the data is inaccessible, it's that we just need to walk the function to where the data lives. We should have written:

```
allFunctions.numericFunctions.calcLog :: {
    code
    theNumber = args[1]
    return new(log(allFunctions.numericFunctions.theNumber))   }
```

Everyone's door is unlocked, but you have to make a deliberate effort to burgle someone's house other than your own parents'.

The define operator is special in that it always defines the member right in the first variable of the search path. So `theNumber` will be defined inside `raiseToPower` even if there is another member called `theNumber` further up the search path.

The notion of a search path gives us a more accurate explanation of code substitution: that operation just changes the *first step* on the search path of the substituted code. Suppose we write

```
(powerOfThree << allFunctions.numericFunctions.raiseToPower)(2)
```

Then the search path for the substituted function begins at `powerOfThree` but then passes on to `numericFunctions`, `allFunctions`, and the workspace variable just as before. This is the purple path in Figure 3. Of course the substitution is temporary, but when the substituted function is run it has a permanent effect on `powerOfThree`: it creates a member inside of it called `theNumber`. That member will inherit the spliced search path of the substituted code.

3.9 Classes and Inheritance

A pair of Cicada's curly braces is a versatile object: it can define not only data types, composite variables, sets and functions but also objects that look somewhat like C++ classes.

3.9.1 Classes

A Cicada class, or class instance, is nothing more than a composite variable having both members to variables and members to methods (functions).

```
myClassObject :: {
    data :: int
    moreData :: string

    init :: {
        code
        { data, moreData } = { 1, "blank" }
    }

    otherMethod :: { ... }
}
```

In a sense, all we have done is point out that functions such as `init()` can be defined inside of composite variables.

To make our class object more C-like we can give it a proper constructor. All we need to do is take the constructor code from the `init()` method and put it somewhere after the class member definitions.

```
myClassObject :: {
    data :: int
    moreData :: string

    { data, moreData } = { 1, "blank" }

    otherMethod :: { ... }
}
```

Now both `myClassObject` and each class instance such as `obj1 :: myClassObject` begins initialized to `{ 1, "blank" }`. We can even bundle the constructor code together with the member definitions:

```
myClassObject :: {
    data := 1
    moreData := "blank"

    otherMethod :: { ... }
}
```

We call these bits of code constructors because they run when the object is created. C++ constructors also run when an object is created, but they can also be rerun by calling a special function. How can a Cicada constructor be rerun? One way is simply to redefine the object:

```
myClassObject :: myClassObject
```

but there is a better way. Just as in C++, we can also run the constructor as a function.

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

It turns out that this same trick can also reinitialize members of functions, by rerunning everything up to the `code` marker. Remember that code number 1, the default function code, begins *after* the first `code` marker or semicolon; therefore the constructor code, which is code number 0, comprises everything *leading up to* that first `code` marker. In the case of a class object, there is no `code` marker, so the constructor is everything inside of the curly braces.

There is no destructor in Cicada. In fact there isn't even a direct way to delete an object – we can only `remove` members leading to an object. Only when Cicada discovers an object to be completely cut off from the user will dispose of the object and free its memory. (It's not particularly good at finding these objects though – to help it out use the `springCleaning()` function.)

3.9.2 Inheritance

Cicada also supports a sort of inheritance. The syntax is different from that in C++: we use a colon ':' to separate the parent object from the code that specializes it. For example, to derive an object from `myClassObject` we could write:

```
myDerivedClassObject :: myClassObject : {
    newString := "hello, I am a new string"
}
```

`myDerivedClassObject` has all of the members of the original `myClassObject` such as `data` and `otherMethod()` (see previous section), as well as the new string member `newString`.

In this last example, we specialized a predefined parent object with inlined code within braces, using a single inheritance operator. But one doesn't have to follow this pattern; we can combine class objects and inlined codes in any number and any combination.

```
A :: { var1 :: int }
B :: { } : { var2 :: string }
C :: A : B
D :: B : A
E :: { var3 :: int, var4 :: string } : A : { var5 :: char }
F :: C : { remove var1 }
```

So for example the type of `C` is effectively `{ var1 :: int } : { } : { var2 :: string }`, whereas `D` is `{ } : { var2 :: string } : { var1 :: int }`. The definition of `F` shows us that it is possible for a derived object to have *fewer* members than its parent object.

Importantly, the order of inheritance affects an object's type. `C` has a different type than `D`, so `C :: D` or `C = @D` will cause a type-mismatch error.

Cicada allows *existing* objects to be redefined as a different type only if the new type is derived from its original type. We can always specialize a member's type by adding more inheritance operators *at the end* of the type specification. All of the following commands will work in order except the last one.

```
C :: A : B              | sure -- was already defined this way
C :: A : B              | we already did this, but fine
C :: C                  | fine -- C equals A : B
C :: C : B              | OK -- now C will be A : B : B
C :: A : B : B : {}     | OK
C :: A : B : B : {}     | error!
```

The last line failed only because any two inlined types are presumed different – Cicada is not in the business of comparing what's inside those braces to see if they match up.

Although types can be specialized, they can never generalized. So typing `F :: C` will cause a type-mismatch error.

The same type-matching rules apply to aliases.

```
D ::  B : A              | was already defined this way
G :: B : A : { }
D = @G                   | legal
(D =@ *) :: B : A        | legal
G = @D                   | NOT legal!
```

Aliasing doesn't change a member's type, which explains why we could reassign `D` to a new variable of its original type.

It turns out that the inheritance operator can derive new types for any composite Cicada object: variables, sets, even functions. Inheritance of sets is best thought of as a concatenation of these sets.

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

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

Inheritance of *functions* basically tacks new code at the end of the old (parent) function. Each sub-code keeps its own original search path (the example of Figure 3 on page 44 shows a situation where these may be different). Function-inheritance makes the most sense when the function does not return a value (i.e. it's a subroutine), because any `return` statement will prevent the new code from running. The resulting function contains members from both parent functions.

```
absval :: {
    sign :: int
    code

    sign = 1
    if args[1] < 0  then sign = -1
    args[1] = that*sign
}

sqrt :: {
    code
    return args^0.5
}

modulus_sqrt :: absval : sqrt
```

Here is a better version of this example, where we use the inheritance operator to specialize a function using a composite object that has no coding section (no `code` marker).

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

    return f(args[1])^0.5
}

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

So the inheritance operator can conjoin different types of objects: functions, sets, classes, basically any composite object. To see how this works, think of any non-function as the constructor part of a function (for example, imagine putting a `code` marker at the very end). Inheritance in Cicada is really a concatenation of code, whether that be constructor code (the commands before the first `code` marker), the first coding block, or any subsequent code blocks. In Cicada type is code. Incidentally, this explains why objects can be specialized but not generalized: we can always do new things to an existing object, but there's no general way to reverse what's already been done to it.

4 Customizing the Cicada language

The biggest improvement in Cicada over its predecessor (Yazoo script) is the completely rewritten bytecode compiler. Along with being simpler, and much faster, Cicada's new compiler makes the language *customizable*. The entire language description is now contained in two arrays in `cicada.c` instead of being baked into the compiler. By changing those arrays, we change the syntax of Cicada script.

4.1 `cicada.c`

Inside the file `cicada.c` lives an array called `cicadaLanguage[]` which defines basically every symbol we find inside of a script. Each array element is a `commandTokenType` structure variable that defines one operator in the Cicada language:

```
typedef struct {
    const char *cmdString;
    ccInt precedence;
    const char *rtrnTypeString;
    const char *translation;
} commandTokenType;
```

The first string, `cmdString`, is the operator symbol or name as written in Cicada. Then we give the precedence level of the operator (see Table 2 on page 22). Next comes a string that explains to the Cicada compiler what type(s) of object this operator 'returns' to the surrounding expression. The final string, `translation`, either encodes the operator directly as bytecode (Cicada's native language), or else 'expands' the operator in terms of other Cicada commands. The first batch of operators have direct bytecode translation, indicated by the fact that their `translation` begins with a `inbytecode` marker (which concatenates to the following string as an unprintable character). But for the second group of operators, towards the end and lacking a `inbytecode` marker, the `translation` is just a fragment of a script built from the previous operators.

To add a *new operator* into the language, simply add a new entry to the end of the `cicadaLanguage[]` array, in the indicated space, and fill in the four fields.

`cmdString:` The command string defines what sort of object in the script will be recognized as the operator. For most operators, operators are recognized by some string of letters, symbols, etc. that marks the operator in the script. (There is really no restriction on this string other than it consist of printable characters.) For example, looking through the array we find simple operators like \ (written with two backslashes in the C string) and `exit`.

Many of the operators take left-hand and/or right-hand arguments, as indicated by keywords like `type3arg` to the left and/or right of the operator string. These keywords are part of the command string – they are just unprintable characters that get concatenated to the operator strings (because they are only separated by a space). For example, the full command string of the define operator is

```
type2arg "::" type6arg
```

The define operator requires both a left-hand expression or argument (the member to define) and a right-hand argument, which can be a member name but also a type like `bool`. The two arguments therefore have a different 'type'. Looking at the comment before the `cicadaLanguage[]` array, we see that a type 2 argument represents a variable or function, and a type 6 argument represents a code-containing expression, as expected. Having different types allows the compiler to throw type-mismatch errors when expressions don't make sense.

In some cases two different operators will have the same operator string. For example, compare '`*`' as a multiplication operator versus the void, or '`-`' as either subtraction or negation. This is *only allowed* if one of the operators expects a left-hand argument and the other doesn't, so that the compiler will immediately know which of the two operators it is looking at when it sees the operator string in a script. For example, when it stumbles upon a '`-`', that symbol will be interpreted as a subtraction if there is a dangling expression just to the left, and a negation otherwise.

More complex definitions like `while ... do` can involve several operator strings.

```
    "while" type5arg "do" type1arg
```

The pattern is always: operator strings like '`do`' alternating with arguments. Sometimes, these complex definitions involve a `optionalargs` keyword: everything before the keyword is required, but everything afterwards is optional. For example, the `if` command

```
    "if" type5arg "then" type1arg optionalargs "else" type1arg
```

requires an `if` and a `then`, but the `else` is optional.

There are 10 allowed argument types: `type0arg` through `type9arg`. (Cicada only uses 9 of these types). There are also a few special types. A `typeXarg` accepts any type of argument, and is used by the `(...)` operator with *no* left-hand argument (i.e. the grouping operator, not a function call) to allow the user to group any sort of expression, even entire commands. The `commentarg` keyword denotes an block of text to be entirely ignored until the next operator string is encountered (i.e. everything from a comment bar '`|`' to an end-of-line is skipped). `chararg` and `stringarg` treat the argument as text containing one or several characters respectively.

Finally, there are several special operators that don't have any operator string at all. If the operator string is simply `int_constant`, then the operator is read when the compiler encounters a number that it deems to be an integer; and the operator whose operator string is `double_constant` corresponds to a floating-point number. The `variable_name` operator is assumed to apply whenever the compiler encounters a novel word beginning with a letter (which may be followed by underscores and numbers). In these three cases the number or word should be thought of as an argument, insofar as the bytecode is concerned.

The final class of special operator strings is the adapters, an important element of Cicada scripting that is explained in the next section. Suffice to say that there is an adapter for each of the 10 argument types, `type0arg_adapter` through `type9arg_adapter`, along with a `noarg_adapter`.

`precedence:` The precedence level determines how operators are bound into expressions. The high-precedence operators are grouped most tightly to their neighbors, and evaluated before the low-precedence operators. Thus `A = 2 * B - 2` is grouped: `A = ( (2*B) - 2)` because of the three gluing operators '`= * -`', multiplication '`*`' has the highest precedence and assignment '`=`' has the lowest precedence. The precedence level is just an integer, although notice that `cicada.c` predefines a keyword for each precedence level and uses that names instead in the operator definitions.

The `cicadaLanguageAssociativity[]` array in `cicada.c` explains how to group operators of the same precedence level, when there are no parentheses to break the tie. This can be important. For example, multiplication and division operators have precedence level 11, and the eleventh entry of the associativity array (i.e. `cicadaLanguageAssociativity[10]`) is `l_to_r` signifying left-to-right grouping. Therefore the expression `8/2/4` groups as `(8/2)/4` which equals 1, as opposed to `8/(2/4)` which equals 16. On the other hand, assignment works at precedence level 5, which has `r_to_l` or right-to-left grouping. Therefore `A = B = C = 2` groups as `A = (B = (C = 2))`, so that 2 copies to `C`, then to `B`, then to `A`. If the grouping were the other way, then each assignment would rewrite `A`.

The size of the `cicadaLanguageAssociativity[]` array determines the allowed precedence levels. So by adding an entry the maximum allowable operator precedence level will be 16. Anything outside the interval [1, *max_precedence*] will cause an out-of-range compiler error.

`rtrnTypeString:` Many operators 'return' a value to the enclosing expression, and which type(s) of value they are allowed to return is encoded in the return-types string. For example, the addition operator has the return-types string `"456"`, so its return value can be construed as being of type 4, 5 or 6. The return types correspond to the argument types from the `cmdString` field, so the expression `A = 2+5` is legal (because the assignment operator expects a type-5 right-hand argument) but `2 + 5 = A` is *illegal* (because the left-hand argument of the assignment operator should be type 2). Notice how each 'type' is really an operator *argument type*: arguments have one type and entire operators have many, which is maybe backwards to the way we usually think.

There is a special `argXtype` return type which is paired with a `typeXarg` argument type. This is used by the grouping operator `(...)`, causing the type inside the parentheses (its 'argument') to be the type returned back to the enclosing expression. The parentheses only force a grouping, without affecting the type of the enclosed expression.

An entire script must be of type 0 – Cicada enforces this using adapters (see below).

`translation:` The last field of an operator definition explains how it will be translated into bytecode. If the bytecode string begins with a `inbytecode` keyword, then the string contains a list of integers which are the bytecode representation of the operator. For ease of reading, the bytecode translations in `cicada.c` are built from string macros defined in `cicada.h`. If there is no `inbytecode` keyword, then the string is interpreted as a fragment of Cicada code that will be translated into bytecode using *previously-defined* operators – so it is best to define these operators last.

The translation strings of bytecode-coded operators have strings of numbers separated by spaces in their translation strings, but also some funny letters: '`a`', '`j`' and '`p`'. The '`a`' letter stands for an argument that is to be substituted into the bytecode at the given location, and is followed immediately (no space) by a number from 1 to 9 indicating *which* argument. (Cicada only supports up to 9 arguments in an operator.) For example, the assignment operator has a bytecode string

```
inbytecode "8 1 a1 a2"
```

meaning that the operation consists of two integers (8 followed by 1), then the first (left-hand) argument, and last the second (right-hand) argument. Each of these arguments can themselves be expressions (think `f().a = 5+cos(b)`), in which case the entire expression translated into bytecode is substituted for the argument. In `cicada.c` the macro `bcArg(x)` produces the "`ax`" string, so the assignment operator reads

```
inbytecode bc_define(equFlags) bcArg(1) bcArg(2)
```

where `bc_define(equFlags)` produces a define operator with equate flags: `"8 1"`.

The '`j`' and '`p`' bytecode symbols are used to specify jump offsets ('`j`' – effectively gotos) and jump positions ('`p`') in the bytecode. Offsets are the number of code words to jump ahead *from the offset word* (negative offsets jump backwards), and the cicada compiler calculates these as the difference between a jump (`j`) marker and a target position (`p`) marker. Each of position/jump marker is followed immediately by a number 1-9 indicating which position to define/jump to. For example, the bytecode string of the `if-then-else` command which potentially takes 3 arguments is:

```
inbytecode "3 j1 a1 a2 1 j2 p1 a3 p2"
```

In `cicada.c` the position markers are produced using `bcPosition()` macros, and the jump operators have dedicated macros taking the jump offsets as arguments, so this same operator definition reads

```
inbytecode bc_jump_if_false(1) bcArg(1) bcArg(2)
                  bc_jump_always(2) bcPosition(1) bcArg(3) bcPosition(2)
```

In this case the first bytecode command – bytecode operator 3 which is the jump-if-false command – jumps to the position of the first position marker, so the compiler calculates this offset by taking the difference in the code position between the `j1` command and `p1` (which is effectively the beginning of argument 3 because `p1` is not a code word) and puts that value in place of the `j1` word. Likewise, there is an unconditional jump later on to the end (`j2`). The third argument `a3` may or may not exist because the third argument is in the optional `else` block: if there is no `else` then `a3` is basically ignored, but the second position marker is still defined.

Many of the adapter operators (explained in the next section) have `anonymousmember` keywords in their bytecode. These are replaced by unique (and negative) member IDs that are found nowhere else in the script: the first use of a `anonymousmember` in the bytecode becomes the number -1, the second use represents a -2 in the bytecode, etc. These are used to construct hidden members that won't bother anyone by conflicting with user-defined members.

Scripted operators – those without `inbytecode` keywords – work basically the same as bytecoded operators except that unfortunately the arguments have to be encoded with special keywords (`arg1` through `arg9`) rather than directly in the string. For example, a `cicadaLibraryFunction()` function is defined at the beginning of `cicada.c` in bytecode, and the `call()` function uses its definition in its script translation:

```
    "cicadaLibraryFunction#0(" arg1 ")"
```

So the command `call("myF", 12)` is first translated into `cicadaLibraryFunction#0("myF", 12)` before being converted into bytecode.

Some operators (usually comments) have no effect on the bytecode whatsoever, and for those we give neither bytecode nor a script translation but instead write `removedexpression` for their translation string. The `|* ... *|` comment block uses this keyword, as does the line-continuation operator `&` which ignores everything to the end of the line. Oddly enough the single-line comment `| ...` doesn't use this keyword, and the reason is that it always breaks two separate commands – so in terms of bytecode it works a lot like a comma or end-of-line.

4.2 Cicada bytecode

Occasionally, we might want to extend the Cicada language in a way that can't be scripted, in which case we have to define our new command directly in Cicada's native 'bytecode', using `cicada.h` as a dictionary to the command words. Scripts run much faster from bytecode than they ever could by reading their original text, but it should be emphasized that Cicada's bytecode is *different from* (and much slower than) raw machine code. One significant difference is that Cicada's bytecode has a recursive structure, composed of expressions and sub-expressions, just like Cicada script itself. In fact, there is pretty much a one-to-one correspondence between the symbols (operators) we write in Cicada and bytecode commands, except that the bytecode commands are in a different order.

For example, when we type the following command into the command prompt:

```
    > area = 3.14 * R^2
```

Cicada's compiler produces bytecode output that looks roughly like:

```
    equate [ <area> , product_of ( 3.14, raise_to_power ( <R> , 2 ) ) ]
```

where we've bracketed the arguments of each bytecode operator. These are just the arguments of each operator in the script, but the compiler has made two changes. 1) Each operator's arguments *follow* the actual operator command: for example the equate operator is followed by two immediate arguments which correspond to the expressions to the left and right of the equate symbol in the script. 2) The operators are reordered, in ascending precedence when parentheses don't force otherwise. The equate is done *last*, so it becomes the *outermost* function in the bytecode. The trick is to think of every operator as a function in bytecode, and write the function command first followed by its arguments.

There's actually a way we can see bytecode from the command prompt: by using the slightly anachronistic `disassemble()` function (dating from before error messages).

```
> bytecodeStr := compile("area = 3.14 * R^2")


> disassemble(bytecodeStr)

equ ( sm $area , mul ( 3.14 , pow ( sm $R , 2 ) ) )
```

Using this tool we find out that there's a 'search-for-member' (`sm`) operator before each member identifier. The member identifier is simply an integer ID number: positive ID numbers for user-defined members (counting upwards from 1), and negative ID numbers (counting downwards from -1) for so-called hidden members which the compiler adds to the bytecode. The 'disassembly' doesn't show it but there's also a 'constant-floating-point' operator just before the 3.14 constant. The raw bytecode will look something like:

```
> bytecd :: [] int

> bytecd[*] =! compile("area = 3.14 * R^2")

> bytecd

{ 8, 1, 10, 309, 29, 55, 1374389535, 1074339512, 31, 10, 310, 54, 2, 0 }
```

where each bytecode command is now just a number which we can look up in `cicada.h`. The two complicated numbers are the bytes of 3.14 broken into integers. The actual output varies based on machine and also on what has been run beforehand (which determines which member ID numbers are assigned). Every script ends with a null word, telling the interpreter to either fall back to the enclosing function or else exit the program.

Pathnames Cicada pathnames consist of a sequence of steps starting from some variable. For example the path

```
myVar.array[5].x
```

takes 3 steps: to `array`, to the fifth element, and finally to `x`. In bytecode the final step is the *outermost* operator, so the entire path looks like

```
step_to_member( step_to_index( step_to_member( "array", search_member "myVar" ), 5 ), "x" )
```

(For speed reasons the 'step-to-member' operator takes the member-to-step-to as its first argument, which is backwards from the other step operators.) Notice that step-to-member continues a path, whereas search-member *begins* a path and so takes one fewer arguments.

Inlined constants Each of the five types of inlined constants—Booleans, characters, integers, floating-point numbers and strings—has a unique bytecode operator. The raw data of the constant follows in subsequent bytecode words (integers). The data for large constants – floating-point numbers and many strings – takes up several bytecode words.

String constants in bytecode use the 'Pascal' string convention rather than the C format: the constant-string operator is the first bytecode word, followed by the *character-length* of the string (also 1 bytecode word), followed by the raw string data (N/`size(int)` words rounded up). There is no terminating character.

Flow control commands The four flow-control commands in Cicada—`if`, `for`, `while` and `do`—are all higher-order commands that the compiler expands into expressions involving 'goto's. Cicada sports three 'goto's: an unconditional jump, and jump-if-true and and jump-if-false operators. Each goto sequence begins with its bytecode command word followed by a jump offset (1 word). The jump offset is the number of bytecode words to jump ahead *from the jump offset*, which is negative if we want to jump backwards. The jump must be take us to the *start of a command* – otherwise `transform()` throws an error. In the case of the two conditional gotos, there is a final bytecode expression following the jump offset which is the condition on which to jump.

The most complicated flow-control command is the `for` statement, which basically consists of a `while` along with an assignment (to initialize the counter) and a counter-increment command at the end of the loop. Notice that if we define a variable inside the `for` loop, as in `for (j::int) in <1, 5>`), then Cicada will plunk the whole expression `j::int` into both the initialization and the increment command, which can slow down short loops considerably.

4.2.1 `define` flags

Member definition (`::`), assignment (`=`), and aliasing (`=@`) are all done by different flavors of the *define operator*. What makes them different is their 'flags': a set of binary properties such as: does this operator define new members? does it copy data? etc. In bytecode, all 8 flags are stored in a single word immediately following the define-operator bytecode word, just before the left- and right-hand arguments to the operator. This section is devoted to that single flags bytecode word.

To calculate a flags word for a set of flags, we treat each flag as a binary digit and read out the number in decimal. For example, the define operator has flags 1, 2, 3 and 5 set, so its flags bytecode word is

$$\begin{aligned} flag &= (1 << 1 = 2) + (1 << 2 = 4) + (1 << 3 = 8) + (1 << 5 = 32) \\ &= 46 \end{aligned}$$

(`<<` is the bit-shift operator). Table 3 on page 63 summarizes the flags words for each Cicada operator. Clearly most of the 256 possible operators are not predefined in `cicada.c`.

Flag 0 – equate: copies data from the source variable into the destination variable. This is used by both the assignment `=` and define-equate `:=` operators, but *not* the aliasing operators.

Flag 1 – update-members: 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 an integer. Thus the member named '`c`' will be defined as an integer, because the define operator sets the update-member flag.

Flag 2 – add-members: creates the looked-for member if it doesn't already exist inside the current-running function. Set by all define operators, even `:: @` which doesn't assign a type to a new member but will create it.

Flag 3 – new-target: does two things. 1) 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. 2) This flag also 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.

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 – relink-target: instructs Cicada 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, not certain indices of an array (i.e. `a[*].b = @c` is legal but `a[<2, 3>] = @c` is not).

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 – run-constructor: causes the constructor of the destination variable to be run after it has been created/having its type updated, but before any data has been copied from the source variable. 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 will grow the same set of members that `comp1` has. Then the final equate should not have any problems, as long as we didn't modify `comp1` after defining it.

Flag 6 – hidden-member: sets the 'hidden' flag of any newly-defined member. Hidden members, which are created by bytecode adapters, can only be accessed by name (which is usually unwriteable). Array-index operators, assignment operations (`=`), comparisons (`==`), and built-in functions like `print()` all skip over hidden members.

Flag 7 – unjammable: makes a member unjammable, i.e. unable to prevent another member from being resized. Ordinarily, if two members alias overlapping indices of an array, 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., inoperable. An unjammed member has to be re-aliased before it can be used again without causing a void-member error.

4.2.2 Adapters

There's something missing in our scripts, and the easiest place to see this is in an ordinary function call:

```
f(2, pi, { 1, int })
```

Realizing that commas are just ends-of-lines, we could write this

```
f(
    2
    pi
    { 1, int }
)
```

which shows that 2, `pi`, and `{ 1, int }` are somehow all valid commands. How can this be?

Let's draw an analogy. Back in the cave man days, there were probably a lot of sentences like

```
rock
```

which in modern English would be written

```
[That which I want to draw your attention to] [is] the rock.
```

In other words, if a sentence only contains an object, the cave man's brain fills it out by adding some stock subject and verb. Cicada works exactly the same way: the stock subjects and verbs to use in different situations are the so-called adapters defined in `cicada.c`. Each adapter allows the compiler to convert some bare expression (e.g. the object of a sentence) to another type (a full sentence) by throwing in a few extra bytecode words.

When the user enters the command '2', the compiler rolls its eyes and reaches for the type-mismatch error button, because it a complete command is a type 1 expression whereas an `int_constant` can only be construed as types 4, 5 or 6 based on its return-types string in the `cicadaLanguage[]` array. But then Cicada notices an adapter that works on type-5 objects (named `type5arg_adapter` in `cicada.c`), and moreover that adapter's return-types string includes a "1" which is what we want. So the adapter adds its code and the error never happens. Here is the adapter's bytecode:

```
inbytecode "8 173 10" anonymousmember "a1"
```

The expression to adapt is considered the argument and goes in place of `a1`. Looking at `cicada.h` or the reference section, we could figure out that this adapter turns our expression into something looking like

```
var1 := 2
```

except that the define-equate operator has slightly different define flags (173 instead of 47).

Other types of objects use different adapters when they appear by themselves. Variables use an adapter that creates an alias: for example the expression '`pi`' becomes something like

```
var2 := @pi
```

though again using slightly different flags from a normal aliasing operator. Finally, type-objects are assigned turned into full commands using a third adapter that adds a define-like operator. Thus '`{ 1, int }`' turns into a modified version of

```
var3 :: { 1, int }
```

or more precisely:

```
var3 :: { var3a := 1, var3b :: int }
```

The `anonymousmember` keyword produces a unique member 'name' that is inexpressible by the user. (Names become ID numbers in the bytecode: user-typed names become positive ID numbers, whereas anonymous members get assigned sequential negative ID numbers as they are encountered. The namespace consists of both the name-ID list and the negative ID counter.) Thus `var1`, `var2`, etc. in the last paragraph don't really don't have those names or any other, so function `f()` will access those members using the bracket operators (`args[1]`, `args[2]`, etc.). It's technically possible to write bytecode having hardcoded negative IDs to access anonymous members, but that's a last resort usually used for anonymous members that are also, in the lingo, hidden.

Hidden members are invisible to the array-index operators. In bytecode-speak, these are produced using define operators whose hidden-member flags are set (flag 6 in Table 3 on page 63). The compiler sprinkles anonymous hidden members discreetly around the code for a variety of reasons, where most of them operate almost undetectably. The one exception: a hidden-define operator embedded in each function call creates a hidden member in the calling space to store the arguments, which becomes directly visible within the function itself through the pseudo-member `args`. (Maybe a better notation would show this member explicitly using braces for function calls, as in `f{x, y}`). The rarest bird of all is the hidden-define-minus-constructor operator (`def-c**` in the table), living exclusively in `trap()` function calls, where it defines the `args` variable *without running its constructor* so that `trap()` can do so in a controlled way.

The other define-operator flag used by (in this case all) adapters is the unjammable flag (flag 7; see Table 3 on page 63), which prevents members from jamming arrays. Consider the function call `f(myArray[<2, 4>])`, which produces a hidden `args` variable consisting of `{ myArray[<2, 4>] }`, which compiles to something like `{ anon1 := @myArray[<2, 4>] }`. Ordinarily `anon1[]` would jam `myArray[]`, and the actual array could not be resized from either member since doing so would also force a resize of the other. But in this case `anon1` was defined as unjammable, as in unjam-able (can be unjammed), so any array resize (e.g. `myArray[+3]`) is allowed because the now-out-of-date `anon1` gets de-aliased rather than causing a jammed-member error. That's OK because the alias will be restored the next time that same function call happens, since the argument constructor will be rerun. (However there can be a problem in other contexts where the constructor is not rerun each time it is used, for example in sets containing aliases to array subsets. Use hard-coded aliases for these cases.) As an aside, these adapters also clear their update-member-type flags (flag 1), so that their anonymous members can be re-assigned to variables of different type (in case, for example, between two iterations of the command `f(a)` member '`a`' gets removed and redefined).

One last type of adapter called `noarg_adapter` replaces missing expressions altogether. These adapters are necessary to allow blank scripts, or situations like two consecutive end-of-lines (command-conjoining operators) which lack a command between them to conjoin. Another no-argument adapter allows a script to contain a `return` command without a variable. The final set of no-argument adapters in `cicada.c` is used to convert a sentences-type expression (type 1) to a script expression (type 0), by adding a null bytecode word at the end.

4.3 Custom-compiling within a script

For a variety of reasons, we might occasionally want to run a script manually without using the `run()` function. This involves two steps. The first step is to produce bytecode, easily done using the `compile()` function. Second, the `transform()` function gives bytecode a perch on a function's internal code registry. Here is a simple example:

```
    myBytecode := compile("myMessage := \"Hello, world!\"; print(myMessage)")

    newFunction :: transform(myBytecode)

    newFunction()
```

Ordinarily `run()` does these things for you. If `user.cicada` wasn't loaded by `start.cicada` (probably because Cicada was set to run another script from the command line), then `run()` isn't defined, so we need to manually load, compile and transform any script we want to run.

Another `compile()-transform()` situation arises when we run a script that uses a *different syntax* from the default Cicada language. For example, our script might want to process commands typed by the user, that are in a completely different format from the Cicada language. In that case we won't want to change `cicada.c` since doing so would break our other scripts like `start.cicada`. Instead we must use the `setCompiler()` function to process the user's input using a different syntax from that of our own scripts. Here is an example, relying on language-constant definitions in `user.cicada`.

```
newLanguage :: [] compiledCommandType
newLanguage[*] = {
    { cat("add  ", type1arg, "to", type1arg), 1, "0",
            cat(inbytecode, "8 173 10", anonymousmember, "27 a1 a2 0") },
    { cat("negative", type1arg), 2, "1", cat(inbytecode, "29 54 -1 a1") },
    { int_constant, 0, "1", cat(inbytecode, "54 a1") },
    { double_constant, 0, "1", cat(inbytecode, "55 a1") }
}
newLanguageAssociativity :: [] int
newLanguageAssociativity[*] = { l_to_r, r_to_l }
newCompilerID := setCompiler(newLanguage, newLanguageAssociativity)

myBytecode := compile(input("Give me math:  "))
doMath :: transform(myBytecode)

doMath()
printl("The answer is:  ", doMath[1])
```

Running this script:

```
Give me math:  add negative 1 to 3.14
The answer is:  2.14
```

Notice how the two array arguments to `setCompiler()` are almost exactly the same as the two arrays that specify the language in `cicada.c`, the main difference just being the use of `cat()` to concatenate strings. For consistency, all of the constants used in the C file (such as `int_constant` and `inbytecode`) are also defined in `user.cicada`. Also, make sure to define both arguments as arrays – `setCompiler()` will not understand anything inside of curly braces. As always, any script must have an overall type of 0. In this primitive example the only possible valid script is an `add` command.

Cicada always opens with a single compiler – compiler ID 1 – so to switch back just type `setCompiler(1)`. Calling `setCompiler()` with no arguments returns the ID of the current compiler, without changing the active compiler.

Functions produced by different compilers live in different namespaces, because each compiler which keeps its own running tally of all member names and anonymous members it has encountered. But the use of separate compilers does *not* prevent collisions of member names between these functions: if anything switching compilers makes collisions more likely. (Any new compiler member will assign member IDs starting from 1 and counting upwards, and that ID is the only thing Cicada sees when the function runs.) To avoid problems, set the search path manually with `transform()` so that the new bytecode can't see the workspace.

5 Reference

5.1 Operators and reserved words

`,` or line break : demarcate commands

`( ... )` : group terms in an expression

`X`∼ : ignore output of expression `X`

`|` : single-line comment

`|* ... *|` : multiple-line comment

`&` : continue command on next line

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

Member/array operators:

`A.B` : step from A into member B

`[ A ]` : step into array index A

`[< A, B >]` : step into array indices A through B

`[^ N ]` : resize array to given size N, step into all indices 1-N

`[+ A ]` : insert index

`[+< A, B >]` : insert indices

`[*]` or `[]` : step to all indices; can resize before `=` or `=!`

`remove A` or `[- A ]` or `[-< A, B >]` : remove member, array index A or indices A through B

Define/equate operators:

`A :: B` : define A (member with a variable) with the type of the variable that B points to

`A = B` or `A <- B` : copy data from B to A

`A := B` : define A to type B, copy data from B to A

`A =@ B` or `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` or `A <- ! B` : copy data from B to A; even between dissimilar data types

Predefined variables:

`this` : the function currently executing

`parent` or `\` : the next function up the search path

`that` : variable on the left side of an equate

`args` : the argument variable to the function currently executing

`top` : within array brackets, the number of indices

`nothing` or `*` : the void

Program flow:

`if A then ..., else if B then ..., else ...` : do if A, B, etc. are true

`while A do ...,` : do as long as A is true

`loop ... until A` : do until A is true

`for I in < A, B >  ...` : do for a defined number of times

`backfor I in < B, A >  ...` : same as `for`, except starts at B and counts backwards to A

`A( ... )` : call A as a function with given arguments

`code` or `;` : begin a new code block

`return A` or `return` : exit function with return variable A (if specified)

`exit` : exit Cicada

Data types:

`bool` : Boolean

`char` : character

`int` : integer

`double` : floating point

`string` : string of characters

`'...'` : inlined character constant (one character only)

`"..."` : inlined string containing given characters

`{ ... }` : inlined function, set, class or data type

`A : B` : an inherited type specialized by B from parent type A

`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

Each entry in the following list consists of: a bytecode command word (which is a number), a name in brackets, and then the arguments for that operator separated by commas. Arguments in plain text take up a defined number of bytecode words. Unless otherwise indicated, a fixed-width argument is one word long and should be read as a signed integer. Arguments in italics are themselves bytecode expressions, which can span arbitrary word-lengths.

0 [null] : marks the end of a code block

1 [jump], offset : jumps the program counter to the position of the offset word plus *offset* bytecode words

2 [jump-if-true], offset, *condition* : jumps the program counter to the position of the offset word plus *offset* bytecode words, if the conditional expression evaluates to true

3 [jump-if-false], offset, *condition* : moves the program counter to the position of the offset word plus offset byetcode words, if *condition* is false

4 [code] : delineates the boundary between two code blocks

5 [return], *return_variable* : exits the function and returns the specified variable

6 [user function], *function_variable*, *args_variable* : runs the given user function with the specified arguments, and returns the return variable (if any).

7 [built-in function], *function_ID*, *args_variable* : runs the built-in Cicada function with the given ID and arguments, and sends back any return value

8 [define], flags, *LH_var*, *RH_var* : applies the define/equate/equate-at command specified by the flags from the source *RH_var* to the target *LH_var*

9 [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

10 [search member], ID : searches backwards from the current function for the member having the given ID

11 [step to member], ID, *starting_variable* : steps to the member with the given ID from the given starting variable

12 [step to index], *starting_variable*, *index* : steps into the given index of the starting variable

13 [step to indices], *starting_variable*, *low_index*, *high_index* : steps simultaneously into the given range of indices from the starting variable

14 [step to all indices], *starting_variable* : steps into all indices of the starting variable

15 [resize], *variable*, *top_index* : resizes the variable's member to have the given number of indices, and steps into these indices

16 [insert index], *variable*, *new_index* : adds a new index to *variable* at position *new_index*

17 [insert indices], *variable*, *new_low_index*, *new_high_index* : adds the new range of indices to *variable* beginning at *low_index*

18 [remove], *member* : deletes the member or part of the member that was stepped into

19 [if equal], *expr1*, *expr2* : returns true if the two expressions' data are equal; false otherwise

20 [if not equal], *expr1*, *expr2* : returns false if the two expressions' data are equal and true otherwise

21 [if greater-than], *num1*, *num2* : returns true if and only if *num1* is greater than *num2*

22 [if greater-than-or-equal], *num1*, *num2* : returns true if and only if *num1* is greater than or equal to *num2*

23 [if less-than], *num1*, *num2* : returns true if and only if *num1* is less than *num2*

24 [if less-than-or-equal], *num1*, *num2* : returns true if and only if *num1* is less than or equal to *num2*

25 [if same reference], *expr1*, *expr2* : returns true if the two members point to the same data; false otherwise

26 [if different reference], *expr1*, *expr2* : returns false if the two members point to the same data and true otherwise

27 [add], *num1*, *num2* : returns the sum of its numeric arguments

28 [subtract], *num1*, *num2* : returns *num1* minus *num2*

29 [multiply], *num1*, *num2* : returns the product of its numeric arguments

30 [divide], *num1*, *num2* : returns *num1* divided by *num2*

31 [power], *num1*, *num2* : returns *num1* raised to the power *num2*

32 [modulo], *num1*, *num2* : returns the remainder of *num1* divided by *num2* after they have been truncated to integers

33 [not], *condition* : returns true if the condition is false and false if the condition is true

34 [and], *condition1*, *condition2* : returns true if and only if both conditions are true (both are always evaluated)

35 [or], *condition1*, *condition2* : returns true if and only if one or both of the conditions are true (both are always evaluated)

36 [xor], *condition1*, *condition2* : returns true if and only if one, but not both, conditions are true

37 [code number], *code number*, *var* : causes an enclosing function call to execute the given *code number* of *var*

38 [substitute code], *code*, *var* : returns *var* but paired with the given *code* instead of its native code

39 [append code], *f1*, *f2* : returns *f1* but with the concatenated code *f1* + *f2* instead of its native code

40 [args] : returns the `args` variable for the current function

41 [this] : returns the function variable that is currently running

42 [that] : returns the variable on the left-hand side of the equate statement

43 [parent] : returns the parent of the currently-running function

44 [top] : returns the highest index of the enclosing array brackets

45 [nothing] : returns no variable

46 [array], *num_indices*, *type* : an array of *num_indices* elements of *type*

47 [bool] : Boolean data type

48 [char] : character data type

49 [int] : integer data type

name	abbr	symbol	flags	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				✓		✓	✓	
∼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	✓	✓			✓	✓		

Table 3: The flags of all various define, assignment and aliasing operators

50 [double] : floating-point data type

51 [string] : string data type

52 [constant bool], num : returns the Boolean stored in *num* (one bytecode word)

53 [constant char], num : returns the character stored in *num* (one bytecode word)

54 [constant int], num : returns the integer stored in *num* (one bytecode word)

55 [constant double], num : returns the floating-point number stored in *num* (*num* is `sizeof(double)/sizeof(int)` bytecode words long)

56 [constant string], characters_num, string_data : returns an inlined string having the given number of characters. The string_data field occupies one bytecode word for every four characters.

57 [code block] : returns the inlined code beginning with the next bytecode sentence and ending with a 0 bytecode word

5.2.1 Define operator flags

Table 3 lists the common define/equate/equate-at operators and their associated bytecode flag words (the bytecode word following the define operator). Each flag word encodes the various binary properties of a define operator: it is the decimal representation of the columns in the table treated as binary digits (where a checkmark equals 1). The flags are: equate (0), update-members (1), add-members (2), new-target (3), relink-target (4), run-constructor (5), hidden (6), and unjammable (7). 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.

5.3 Cicada built-in functions

Cicada provides a number of built-in functions that work just like user-defined functions. These are listed in the Table 4. A function's ID number is just its 'name' in bytecode – we don't usually have to deal with this. Functions that leave behind a value when used as a command (i.e. print a value from the command prompt) have checkmarks in the '@' column. This section explains each built-in function in alphabetical order.

ID	name	@	ID	name	@	ID	name	@
0	call		10	trap		20	tan	✓
1	setCompiler	✓	11	throw		21	acos	✓
2	compile	✓	12	top	✓	22	asin	✓
3	transform		13	size	✓	23	atan	✓
4	load	✓	14	abs	✓	24	random	✓
5	save		15	floor	✓	25	find	✓
6	input	✓	16	ceil	✓	26	type	✓
7	print		17	log	✓	27	member_ID	✓
8	read_string		18	cos	✓	28	bytecode	✓
9	print_string		19	sin	✓	29	springCleaning	

Table 4: Built-in functions, by bytecode ID number. Functions which return values even as commands have a checkmark in the '@' columns

`abs()`

syntax: (numeric) y = `abs`((numeric) x)

Returns the absolute value of its argument (which must be a number).

`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].

`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 [$-\pi/2$, $\pi/2$].

`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 [$-\pi/2$, $\pi/2$].

`bytecode()`

syntax: (string) *codeString* = `bytecode`((variable) *myFunction* [, (numeric) *memberIndex*])

Returns the bytecode of a given variable or member. If there is one argument it returns the bytecode of that variable; if there are two members then it returns the bytecode of member `myFunction[memberIndex]`. Member code is never run directly, but it determines the sort of variable a member can point to (because *code* and *type* are equivalent in Cicada).

To read the bytecode we need to move the bytecode data from the string into an array of integers using the `=!` operator. The last integer is always 0, signifying the end of bytecode. If there are multiple codes (due to the inheritance operator) then the codes are concatenated in parent-to-child order in the same string, and each separate code ends in a null integer. `bytecode()` is the inverse operation to `transform()`.

The `bytecode()` function return the code for functions, but also many other objects that we don't normally think of as having code. In fact the only restriction is that `myFunction` must be some composite object (defined using curly braces). So if we define

```
pow :: {
    params :: { x :: y :: double  }

    code

    params = args
    return new(params.x^params.y)
}
```

then `bytecode(pow)` returns the bytecode for everything inside `pow()`'s definition (including the definition of `params` and the `code` marker), whereas `bytecode(pow.params)` is also legal and returns the bytecode corresponding to `x :: y :: double`.

`call()`

syntax: (numeric) *return_code* = `call`((string/numeric) *C_routine*, [*arguments*])

Runs a user-defined C or C++ routine referenced in `UserFunctions[]` (in `userfn.c`). The first argument specifies which function to run, either as a string containing the Cicada function name (the string in `UserFunctions[]`), or else as the array index (beginning at 1) of the function in `UserFunctions[]`. The subsequent arguments form the `argv` array that the C routine receives. Returns the return value of C/C++ function (an integer).

For example, if we write some C function

```
ccInt myFunction(ccInt argc, char **argv)  {  ...  }
```

just as if it were a complete program. In order to use it from inside Cicada we would add an entry to the `UserFunctions[]` array in `userfn.c`.

```
userFunction UserFunctions[] = { { "pass2nums", &pass2nums }, { "cicada", &runCicada },
                 { "runFunctionInC", &myFunction } };
```

`callFunctionInC` is Cicada's name of the function, even though `myFunction` is its C name. Then, after recompiling Cicada, we could run the function with the command

```
result := call("runFunctionInC", arg1, 14, "another argument")
```

or

```
result := call(3, arg1, 14, "another argument")
```

(because `runFunctionInC` is the 3rd C function in `UserFunctions[]`). If we are using a string to specify the C function, then we can also use a shorthand syntax where that string follows a dollar-sign:

```
result := $runFunctionInC(arg1, 14, "another argument")
```

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[0]` is the pointer to the first argument, and `argv[0][0]` is the first byte of the first argument; `argv[1][0]` 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. In the example above the number of arguments is at least 3, depending on the type of `arg1`. Suppose `arg1` was an array of `{ bool, double }` – then the number of arguments is 4. (The first argument is an array of `bool`s, the second is an array of `double`s, the third contains the integer 14, and the fourth is a string). Strings are passed as linked lists (see the reference section). Void arguments or members are skipped.

`call()` also adds one final argument at the end, in position `argv[argc]`, that can be used for type-checking. This argument-info array is a list of elements of type `arg_info`, defined in `userfn.h`:

```
typedef struct {
    ccInt argType;
    ccInt argIndices;
} arg_info;
```

There is one entry in this array for each argument passed by the user. The `argType` codes are defined in Table 1 on page 16. `argIndices` 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 Cicada's own data storage, so data can be exported as well as imported. It is easy to crash Cicada by overwriting the wrong regions of memory: a `call()` preceding a crash is frequently at fault, even if the crash happens far downstream.

`ceil()`

syntax: (numeric) y = `ceil`((numeric) x)

Returns the nearest integer that is as high as or higher than the argument, which must be numeric. For example, `ceil(5.6)` returns 6, `ceil(-5.6)` returns -5, and `ceil(2)` returns 2.

`compile()`

syntax: (string) *script_bytecode* = `compile`((string) *script* [, (string) *file_name* [, (string) *char_positions* [, member_names]]])

Before Cicada can execute a script, that script must be compiled into a binary form called bytecode that is much easier to execute than the raw text. The built-in `compile()` function does this job. Given a string containing a Cicada script (`script`), `compile()` then returns a second string (`script_bytecode`) containing Cicada bytecode. Importantly, the bytecode is *not* machine code – it is only used by Cicada.

A basic `compile()` call looks like:

```
myBytecode := compile("x = 3")
```

This command produces bytecode using the currently-active compiler. Each compiler keeps a record of all variable names, so if `x` had been defined with bytecode produced by the current compiler then this command will run just fine. To change compilers use the `setCompiler()` command.

A compilation error will actually crash the script running the `compile()` command. To prevent this we can enclose the `compile()` call inside of the `trap()` function. If we want to print out the error message, we can write a semicolon or `code` marker at the beginning of `trap()`'s arguments.

```
> trap(; compile("x = "))

Error:  right-hand argument expected

x =
   ^
```

The optional second `file_name` argument causes any error message to reference that file name.

```
> trap(; compile("x = ", "myFile.txt"))

Error:  right-hand argument expected in file myFile.txt
```

Often a script will compile but cause an error when it runs. In order to properly flag runtime error messages we must collect another piece of information: the character position in the original script of each bytecode word. This lets the error message flag the offending line in the original script. The character positions are stored inside of any string that is passed as an optional third argument to `compile()`. Both that string and the original Cicada script will be passed to `transform()`, the function that actually allows compiled bytecode to be run.

In some cases we may want to avoid using `compile()`, but instead hand-code the bytecode and load it in using `transform()`. After all, `compile()` is 'only' a string operation: it converts an ASCII script into a string containing binary bytecode.

`cos()`

syntax: (numeric) y = `cos`((numeric) x)

Returns the cosine of its argument. The argument must be numeric.

`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 optional 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.

`floor()`

syntax: (numeric) y = `floor`((numeric) x)

Returns the nearest integer that is as low as or lower than the (numeric) argument. For example, `floor(2.3)` returns 2, `floor(-2.3)` returns -3, and `floor(-4)` returns -4.

`input()`

syntax: (string) *str* = `input`()

Reads in a single line from the C standard input (which is usually the keyboard). `input()` causes Cicada's execution to halt until an end-of-line character is read (i.e. the user hits return or enter), at which point execution resumes. The return string contains all characters before, but not including, the end-of-line. Reading in 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. Both ASCII-encoded and binary files can be read this way. 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 Cicada. The counterpart to `load()` is `save()`.

`load()` searches only in the default directory. The user.cicada routine `Load()` extends the built-in `load()` by searching all paths specified in the `filePaths[]` array. (The `run()` function in user.cicada also searches all `filePaths[]`.)

`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.

`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 bytecode representation of the member's name. 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 is the second argument.

`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. String arguments are written verbatim (byte-for-byte) to the screen, except that 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.

When Cicada is run from the command prompt, `user.cicada` loads three further printing functions: `printl()` (print with line break), `sprint()` (for printing composite structures), and `mprint()` (printing arrays). `sprint()` is the default function for printing expressions typed by the user.

`print_string()`

syntax: `print_string`([(numeric) field_width, [(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 Cicada writes to the string (with no spaces in between). Strings from the source variables get copied into the destination string verbatim. Numeric variables are written as text, and here `print_string` differs 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).

If the first argument is numeric, then it is taken as the minimum field width for numeric and Boolean (but not string or character) variables to be printed; otherwise the default minimum field width is zero. If both the first and second arguments are numeric, then the second argument is the output precision for floating-point `single` and `double` variables; otherwise the output precision is determined by the C constant `DBL_DIG` for `double`-typed variables. When no precision is specified, `print_string` prints considerably more digits than does `print()`, whose precision is set by `printFloatFormatString` at the top of `cmpile.c`.

`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, Cicada calls C's `rand()` function twice:

$$\texttt{random()} = \texttt{rand()}/\texttt{RAND\_MAX} + \texttt{rand()}/(\texttt{RAND\_MAX})^2$$

The random number generator is initialized by Cicada to the current clock time each time the program is run, so the generated sequence should not be repeatable.

`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 Cicada's memory. Thus numeric fields in the source string need to be written out as text, as in "3.14". Each string field must be one written word long, so "the quick brown" will be read into three string variables, not one. 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 :: int }
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), then `read_string()` throws a type-mismatch warning. Warnings can also be thrown if `read_string()` 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()`.

`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 Cicada directory.

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 using the `load()` function.

`setCompiler()`

syntax: (numeric) *compilerID* = `setCompiler`([(numeric) *compilerIDtoUse*]) *or:* (numeric) *compilerID* = `setCompiler`(array of { (string) commandString, (int) precedence, (string) rtrnTypes, (string) translation }, (int array) opLevelDirections[])

Optionally sets the active compiler, and returns the ID number of the active compiler. To use a particular compiler, pass its ID number as the first argument. To simply find the ID number of the current compiler, we can just call `setCompiler()` with no arguments. To create a new compiler we pass `setCompiler()` having two arguments: 1) an array of `{ string, int, string, string }`, one element for each command, containing the definitions of each command, and 2) an array of the direction to evaluate each order-of-operations level. These mirror the `cicadaLanguage[]` and `cicadaLanguageAssociativity[]` arrays, respectively, which are defined in `cicada.c`.

`sin()`

syntax: (numeric) y = `sin`((numeric) x)

Returns the sine of its argument, which must be numeric.

`size()`

syntax: (numeric) *var_size* = `size`((var) *my_var* [, (bool) *physicalSize*])

Returns the size, in bytes, of the first argument. 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 *unless* the optional second argument is set to `true` (its default value is `false`). If the second argument if `false` then `size()` returns 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 which is returned by setting the second argument to `true`.

If a member points back to the composite variable, as in

```
a :: {
    self := @this
    data :: int    }

size(a)   | will cause an error
```

then the size of `a`, including its members and its members' members, etc., is effectively infinite, and Cicada throws a self-reference error unless the second argument was set to `true`.

`springCleaning()`

syntax: `springCleaning`()

This function removes all unused objects from Cicada's memory, in order to free up memory. An object is termed 'unused' if it cannot be accessed by the user in any way. For example, if we `remove` the only member to a function then that function's internal data can never be accessed unless it is currently running.

Cicada tries to free memory automatically, but unfortunately it is not always able to do so. (The reason is self-referencing loops between objects in memory.) The only way to eliminate these zombies is to comb the whole memory tree, which is what `springCleaning()` does. When Cicada is run from the command prompt, it disinfects itself with a `springCleaning()` after every command from the user. But we might want to scrub the memory more often if we are running a lengthy, memory-intensive script that allocates and removes memory frequently. `springCleaning()` can help unjam arrays, if there is no member leading to the jamb.

`tan()`

syntax: (numeric) y = `tan`((numeric) x)

Returns the tangent of its (numeric) argument.

`throw()`

syntax: `throw`((numeric) error_code [, (composite) error_script [, (numeric) concat_number [, error_index [, (Boolean) if_warning]]]])

Causes an error to occur. This of course stops execution and throws Cicada back to the last enclosing `trap()` function; if there is none then Cicada either prints an error (if run from the command line) or bails out completely. The first argument is the error code to throw – these are listed in Table 5 on page 92. The optional second, third and fourth arguments allow one to specify the function, the part of the function (should be 1 unless the inheritance operator was used) and the bytecode word in that function the error appears to come from. If one sets the optional fifth 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 1-50, `throw()` works perfectly well for larger (or smaller) error codes that Cicada 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(2)
```

In this case `throw()` actually worked: we got an out-of-range error because that is error #2. (That once caused the author some confusion..)

`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). Notice that in both of these cases we can replace the `top()` function by the `top` *keyword*, which is always defined inside of array brackets: e.g. `my_var[top+1]`.

`transform()`

syntax: (composite) *target_function* = `transform`((string) *bytecode* [, (composite) *target_function* [, (composite) *code_path* [, (string) *file_name* [, (string) *script* [, (string) *char_positions*]]]]])

Copies compiled bytecode stored as a string (1st argument) into the internal code of a target function variable (return value and/or 2nd argument), *without* running the code's constructor. The bytecode is typically generated using the `compile()` function:

```
    newFunction :: transform(compile("toAdd := 2; return args[1]+toAdd"))
```

but it is also possible to write the bytecode by hand. This probably won't work – the member IDs depend on your workspace history – but the code looks something like:

```
    newFunction :: {}
    (newBytecode :: string) =! { 8, 47, 10, 314, 54, 2, 4, &
                        5, 8, 237, 10, -999, 27, 12, 40, 54, 1, 10, 314, 0 }
    transform(newBytecode, newFunction)
```

At this point it is as if we had written

```
    newFunction :: { toAdd := 2; return args[1]+toAdd }
```

We can now execute the new code by running the target function.

```
    newFunction(3)        | will return 5
```

When we define a function as the return value of `transform()`, as in the previous example, the constructor runs automatically. If we don't want this to happen, we should pass in a target function as the second argument of `transform()`. If a function appears here, that is not void, then that function's existing codes are erased and replaced by the transformed code (assuming no error) without running the constructor.

The default search path for the transformed code is the same search path used the function that called `transform()`, but we can replace this default with a manually-constructed path by passing a set of variables as the optional 3rd argument. For example

```
    A :: B :: C :: { D :: {} }
    transform(newBytecode, newFunction, { A, C.D, B })
```

causes `newFunction()`'s search path to go from `newFunction` to `A` to `C.D` and finally end at `B`.

The optional fourth, fifth and sixth arguments help Cicada to give helpful error messages if the new code crashes when we try to run it. The fourth argument is just the name of the file containing the script, if applicable (otherwise set it to the void). The fifth argument is the original ASCII text of the script, and the sixth is the mapping between bytecode words and script characters that is an optional output of `compile()`. Here is how we pass all of this information between `compile()` and `transform()`:

```
    fileName := "scriptFile.cicada"
    myScript := load(fileName)
    opPositions :: string
```

```
scriptBytecode := compile(myScript, fileName, opPositions)

newFunction :: {}
transform(scriptBytecode, newFunction, { }, fileName, myScript, opPositions)
```

It is certainly possible to pass bogus bytecode to `transform()` (particularly if we're trying to write out the binary ourselves). `transform()` checks the bytecode's syntax, and if there is a problem then it crashes out with an error message.

`trap()`

syntax: (numeric) *error_code* = `trap`([code to run])

Runs the code inside the parentheses (i.e. its argument), and returns any error value. Error codes are listed in Table 5 on page 92. No `code` marker is needed within a `trap()` call. Upon error, the argument stops running and the error code is returned; if the argument finishes with no error then the return value is 0. `trap()` thus prevents a piece of dubious code from crashing a larger script. Note that the most egregious errors are compile-time errors and `trap()` will not be able to prevent those – this includes some type-mismatch errors like `trap(string = 4)`.

A `trap()` call can optionally print out an error message if needed. To do this we add a semicolon (or `code` marker) immediately at the beginning of its arguments. A second opening semicolon causes it to print any error message *without* clearing the error – so code execution will then fall back to the next enclosing `trap()` and possibly print another message. This can help to trace errors through multiple nested functions.

```
trap((a::*) = 2)                | prevents a crash
errCode := trap((a::*) = 2)     | returns the type-mismatch error code
trap( ; (a::*) = 2)             | prints a type-mismatch error but doesn't cause crash
trap( ; ; (a::*) = 2)           | prints a type-mismatch error, then crashes out
```

Notice that `trap()` will also print warning messages (minor errors that don't stop the program). Warning codes are the same as error codes except that they are negated: for example an out-of-range error will return error code 2, but an out of range warning will return -2. If several warnings have been produced, `trap()` will only print and return the error code for the last one.

The `trap()` function has the unique ability to run its arguments in whatever function called `trap()`, rather than in a private argument variable used by all other built-in and user-defined functions. So variables which are defined within the `trap()` argument list will be accessible to the rest of the function. Also `this` and `parent` have the same meaning inside a `trap()` command as outside of it.

`type()`

syntax: (numeric) *theType* = `type`((composite variable) *var* [, (numeric) *memberNumber*])

Returns a number representing the type of the given variable (one argument) or one of its members (if there is a second argument). The variable is the first argument, and the member number is the second argument. The types IDs are listed in Table 1 on page 16. A composite-typed variable or member only returns a '5' even though its full type is properly determined by its code list – use the `bytecode()` function to obtain the code list.

5.4 Functions define in Cicada scripts

When Cicada is run from the command line with argument, as in

```
user-prompt%./cicada myfile
```

it runs the user's script `myfile.cicada`. On the other hand, when Cicada is run without a script file:

```
user-prompt% ./cicada
```

then Cicada runs its own script `start.cicada` which brings up the interactive command prompt. The first thing `start.cicada` does is to run a second script file `user.cicada` which predefines a number of variables and functions for the user. Here we list a few relevant contents of the prepackaged start.cicada file, along with the definitions in `user.cicada`.

5.4.1 `start.cicada`

The `start.cicada` script provides an interactive command prompt, while keeping most of its internal functions hidden from the user. However, a few of its variables and functions have aliases inside the user's workspace, and those are the focus of this section.

`calculator:` is a function that prints the results of incomplete expressions at the command line. An incomplete expression is one that is not assigned to a variable, 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, thanks to the calculator.

To be completely technical, the calculator prints the data of all hidden members created by the user. Since these members will never be used again `start.cicada` removes them after each command from the user, but only after asking the calculator to print them first. The calculator is only a printing function, not a calculator per-se.

By default, the calculator is aliased to `user.cicada`'s `sprint()` function. We can change the output style by aliasing it to another printing function:

```
> calculator = @mprint
```

Or, if calculator is getting annoying (e.g. printing the output of functions we don't care about, or that return enormous blocks of data), we can turn it off altogether:

```
> calculator = @nothing
```

`ans:` : short for "answer". This is aliased to whatever the calculator last printed (void if the calculator hasn't printed anything yet). Use `ans` like any other variable:

```
> 2+5
```

```
7

> ans*2

14
```

`allNames`: the list of member names that have been defined so far, including those defined in `start.cicada`. `allNames` is updated with every new command-line prompt, or every time the user calls `compile()` with `allNames` as the fourth argument. This list is used by `user.cidada`'s `go()` and `what()` functions.

`go_path[]`: a set of aliases to each composite object in the search path, beginning with `root` and ending with the current working variable. `start.cicada` uses `go_path` to form the search path for each command entered by the user. Both `go()` and `jump()` work by modifying `go_path`, and `start.cicada` will reset `go_path` if it detects a problem. The user can add a small coding section to `go_path` which `start.cicada` will run if it needs to reset the path; in this coding section do not define any variables or run any functions or problems will start happening.

5.4.2 `user.cicada`

When Cicada is run interactively, `start.cicada` runs the script file `user.cicada` file which simply predefines a number of useful constants and functions in the workspace. As its name suggests, the user is absolutely encouraged to customize `user.cicada` in whatever way makes it most useful.

Constants and variables

`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, Cicada defines `e` so that the user can evaluate exponentials by writing `e^x`, `e^-2`, etc.

`pi`: the famous constant pi.

`inf`: infinity.

`nan`: not-a-number (used by floating-point arithmetic).

`root`: an alias to the user's workspace.

`where`: the current search path of the user, stored as a string.

`filePaths[]`: a string array of pathnames to folders. The `user.cicada` routines (but *not* the built-in Cicada functions!) will search each of these paths when looking for a file. `user.cicada` preloads the empty path, which is usually the Cicada directory. We can change the search paths just by manipulating this set: e.g. `filePaths[+2] = "/Desktop/"`.

We will skip the custom compiler definition here since they are explained in Chapter 4. Next is a function that fixes one of the more awkward aspects of Cicada's use of functions.

Function calling

`new()`

syntax: (same type) *var2* = @`new`((any type) *var1* [, (variable) *data_to_copy*] [; modifying code])

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(x + y)     }
```

`new()` will not work if the variable's type does not match its current contents, which can happen if the variable had been modified after being created:

```
comp1 :: {  a :: int  }
comp1.b :: char

new(comp1)     | cannot copy the data
```

In this case `new()` will create a new variable using `comp1`'s constructor, but it will not be able to copy the data over and will output a warning. There are two ways to fix this. 1) The optional second argument to `new()` decouples the variable that provides 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. 2) Any coding section in the arguments are run *inside* the new variable immediately after it is created, allowing us to modify the variable before the 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.

File input and output

`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.

`cd()`

syntax: `cd`((string) *filepath*)

The easiest way to change Cicada's file-search directory. `cd()` resizes the `filePaths[]` array to size 1 and sets that to the string given as its argument.

`pwd()`

syntax: `pwd`()

Prints all file directories (all entries in the `filePaths[]` array) to the screen.

String operations

`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.

`C_string()`

syntax: (string) *string bytes* = `C_string`((string) *my_string*)

Cicada strings are normally stored internally as linked lists. `C_string()` converts a length-N resizable

Cicada string to a $N + 1$-byte C-style string containing a terminating 0 character.

`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)`.

Printing routines

`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. It prints each member of an object separated by commas, and each composite object is enclosed in braces. Void members are represented by asterisks. The output is in exactly the format that Cicada uses for constructing sets.

```
> sprint({  a := 5, b :: { 4, 10, "Hi" }, nothing }, 'q')

{ 5, { 4, 10, Hi }, * }, q
```

`sprint()` is the default calculator (i.e. `calculator` aliases `sprint()`).

`mprint()`

syntax: `mprint`([data to print] [; (ints) `fieldWidth`, `maxDigits`, (string) `voidString` = values])

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:

```
> mprint({ 2, { 3, nothing, 5 }, { 5/2, "Hello" } })

2
3         *         5
```

```
   2.5       Hello
```

`mprint()` has three user-adjustable optional parameters that can be changed in the argument coding section. `mprint.fieldWidth` controls the number of spaces in each row; it defaults to 12. `mprint.maxDigits` controls the precision of numbers that are printed out; it defaults to 6. A `maxDigits` of zero means 'no limit'. `mprint.voidString` is the string used to represent void members.

Reading/writing tables and data structures

`writeTable()`

syntax: (string) *table_string* = `writeTable`((table) *data* [; (ints) `fieldWidth`, `maxDigits`, (string) `voidString` = values])

`writeTable()` exports table data as a string. This function takes the same three optional arguments as `mprint()`.

`saveTable()`

syntax: `saveTable`((string) *filename*, (table) *data* [; (ints) `fieldWidth`, `maxDigits`, (string) `voidString` = values])

The `saveTable()` routine exports data stored a set or array to a file. This routine attempts all file paths when saving, just like `user.cicada`'s general-purpose `Save()` function. The optional arguments are the same as those used by the function `mprint()`.

`readTable()`

syntax: `readTable`((table) *table_array*, (string) *table_text* [; (bools) `ifHeader`, `resizeColumns`, `resizeRows` = 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. 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 list of 1-dimensional arrays 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.errCode` and an error message printed to the screen.

`readInput()`

syntax: `readInput`((table) *table_array* [; (bools) `ifHeader`, `resizeColumns`, `resizeRows` = values])

Identical to `readTable()`, except reads the table string from the command line input.

`readFile()`

syntax: `readFile`((table) *table_array*, (string) *file_name* [; (bools) `ifHeader`, `resizeColumns`, `resizeRows` = values])

Identical to `readTable()`, except reads the table string from a file. Searches all directories in the `filePaths[]` array.

Running code

`run()`

syntax: (numeric) *script_return_value* = `run`((string) *filename* [, (composite) *target*])

The essential `run()` function runs a script stored in a file. `run()` compiles, transforms and finally runs the code in the current `go{}` location and search path. Any errors in the process are flagged along with the offending text. `run()` searches all directories in the `filePaths[]` 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. If this is a problem then set `run.CleanUp = false`. (This parameter is not set within the arguments.) To make sure it knows when a script was rerun, make sure that the Boolean `run.caseSensitive` is set properly for your file system (it defaults to `false` meaning that Cicada assumes the file system doesn't discriminate filename cases).

`do_in()`

syntax: `do_in`((composite) *target* [, *search path* [, *code_args* [, *bytecode_mod_args*]]] , code, base script [, `code`, code modifying `bytecodeWords[]`])

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 Cicada 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 `bytecodeWords` of integers, 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 `bytecodeWords[]` to an existing integer array that is already filled with bytecode.

Here we show how to use `do_in()` to create an unjammable alias to some variable `var1`, which cannot be done using ordinary Cicada scripting.

```
do_in(
    root

    code

    al := @var1

    code

    bytecodeWords[2] = 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. We write the arguments just as we did for `do_in()`, except with an extra pair of double-quotes around the code to compile (even though it's in the coding section of the arguments). The analog of the `do_in()` example would be:

```
compile_and_do_in(root; "al := @var1"; bytecodeWords[2] = that + 128)
```

Working variable

`go()`

syntax: `go`([code,] *path*)

Cicada's `go()` function 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)

> b    | we are now in 'a', so this is legal

2

> a    | search path extends back to root, so we can see 'a' as a member

{ 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.cicada. 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(...)`, Cicada 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 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 an 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 trying to reset the path by typing `go()` will not work at that point, so in this sticky situation `start.cicada` will eventually take pity and bail the user out. The upshot of all this is that `go()` does not work very well 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), where

root.a.b

> go(a), where   | starting from a.b

root.a

> go(b), where

root.a.b

> jump(a), where   | again, starting from a.b

root.a.b-->a
```

`jump()` takes advantage of the fact that search paths in Cicada 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.

`what()`

syntax: (string) *var_names* = `what`([(composite) *var_to_look_in*])

Returns the names of the variables in the current directory, which is usually `root` (see `go()` and `jump()`). If an argument is provided then `what()` returns the names of the variables inside that argument variable. Remember that `what()` *requires* the parentheses!

Numeric

`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.

`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.

`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 }
for (c1::int) in <1, 10> myTable[c1] = { random(), random() }

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 (although it will work with character fields).

`binsearch()`

syntax: `binsearch`((table) *table_to_search*, (numeric) *value_to_find*)

Searches a sorted list for a given value. The list must be numeric (`char`-typed lists are OK). If the list is not sorted then `binsearch()` will probably not find the element.

Bytecode

`disassemble()`

syntax: [(string) *disassembly* =] `disassemble`((string) *compiled_code* [, (string array) *name_space* [, (int) *start_position*]] [, code, (bool) `expandFunctions`, (int) `flagPosition` = values])

The `disassemble()` function returns a textual interpretation of compiled Cicada bytecode. The first argument is a string containing the bytecode. The optional second argument allows the user to pass a different namespace (a string array) other than `allNames[]`, or `*` to avoid printing member names. The function will return the 'disassembly' as a readable string. Used by the author to satisfy the odd craving for a rush of bytecode:

```
> disassemble(compile("x = that + 2", *, *, allNames))
```

By passing a third argument, the disassembler can be used to skip over a bytecode expression. 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 command in a compiled expression.

```
go_to_Nth_sentence :: {

    code

    code_string := args[1]
    N := args[2]

    code_index := 1
    for (n :: int) in <1, N-1>  &
        disassemble( code_string, *, code_index )

    return new(code_index)
}
```

When run in this 'skip' mode, `disassemble()` does *not* return any bytecode string. If you want the output string you should first find the end of the expression that *start_position* begins, then do a full disassembly on just that expression.

The `expandFunctions` option determines whether inlined code definitions (as in, objects defined within curly braces) are disassembled (`true` is the default), or skipped with an ellipsis if `false`. If `flagPosition` is set to an integer value then the disassembler will flag that bytecode word, which is useful for marking errors.

5.5 Linked list routines for handling Cicada strings in C

Cicada strings are all stored in linked lists, and when a Cicada script calls a user-defined C function any string arguments are passed as `linkedlist` structure variables. The `linkedlist` data type is defined in `lnklst.h`, so that header file needs to be included in any C source file that uses Cicada strings. `lnklst.h` also prototypes the functions that help the user read, write, and resize these strings. The `lnklst` source files were written for Cicada, but they are completely stand-alone and can be used in other C programs.

The `linkedlist` data type is defined as follows:

```
    typedef struct {
        ccInt elementNum;
        sublistHeader *memory;
        ccInt elementSize;
        ccFloat spareRoom;
    } linkedlist;
```

The only field relevant for Cicada strings is `elementNum`, which stores the number of characters (bytes) in the string. `memory` points to the first storage sublist, while `elementSize` should always be 1 (the byte-size of a character). `spareRoom` is amount of extra room to allocate in sublists relative to `elementNum`; this extra space uses memory but can speed up the insertion of new elements.

Here is an example of how to use a linked list in a C function.

```
    ccInt doubleString(ccInt argc, char **argv)
    {
        linkedlist *theString = (linkedlist *) argv[0];
        ccInt numInitChars = theString->elementNum;

        addElements(theString, numInitChars, ccFalse);
        copyElements(theString, 1, theString, numInitChars+1, numInitChars);

        return 0;
    }
```

Notice that we used a data type defined in `lnklst.h` called `ccInt`, which is (by default) just an `int` type. The reason for this is that we can change the default integer type to some other signed integer – for example a `short` or `long int` type – by changing the `typedef` at the beginning of `lnklst.h` along with the two integer limits defined below it. We can also change Cicada's default floating-point type here. If we do this then we should also change the format-string constants at the beginning of `cmpile.c`.

There are two rules to keep in mind when using linked lists, to avoid crashing Cicada. 1) The `linkedlist` variable should only be updated by Cicada's linked list routines. 2) Whenever a linked list is updated the *original* `linkedlist` variable (i.e. Cicada's own copy) must be updated as well. Therefore it is critical that linked lists always be passed *by reference* to any C routine that could conceivably modify that list.

`newLinkedList()`

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.

`deleteLinkedList()`

syntax: `deleteLinkedList`((linkedlist *) *LL*)

De-allocates the storage of a 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.

`insertElements()`

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 to add is given by the third argument. The fourth argument, if set, zeros the new memory.

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 `spareRoom` field of the linked list variable.

`addElements()`

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()`

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.

`deleteElement()`

syntax: (numeric) *err_code* = `deleteElement`((linkedlist *) *LL*, (numeric) *element_index*)

Same as `deleteElements()` except only deletes one element.

`resizeLinkedList()`

syntax: (numeric) *err_code* = `resizeLinkedList`((linkedlist *) *LL*, (numeric) *num_elements*, (Boolean) *if_cleared*)

Adds elements to the end, or deletes elements from the end, so that the linked list will have the required number of elements. If elements are being added then they will be zeroed if and only if *if_cleared* is set to true (`ccTrue`).

`defragmentLinkedList()`

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 work with when it is de-fragmented, and it can save memory too (even when `spareRoom` is 0).

The `call()` function defragments all of its string-arguments before executing a user-defined C or C++ routine.

`copyElements()`

syntax: [(numeric) *err_code* =] `copyElements`((linkedlist *) *source_LL*, (numeric) *first_source_index*, (linkedlist *) *dest_list*, (numeric) *first_dest_index*, *num_elements_to_copy*)

Copies data 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 is 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 even when the source and destination ranges overlap). The two lists' element sizes must match.

`compareElements()`

syntax: (Boolean) *err_code/result* = `CompareElements`((linkedlist *) *source_LL*, (numeric) *first_source_index*, (linkedlist *) *dest_list*, (numeric) *first_dest_index*, *num_elements_to_compare*)

Compares data between two linked lists, or between two parts of the same linked list. The return value is either an error code, or the Boolean result of the comparison.

`fillElements()`

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.

`clearElements()`

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.

`setElements()`

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 buffer address is a pointer to the beginning of the data to copy.

`setElement()`

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.

`getElements()`

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 write address points to the start of the buffer.

`getElement()`

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.

`elementExists()`

syntax: (numeric) *err_code/if_exists* = `elementExists`((linkedlist *) *LL*, (numeric) *index*)

Returns `ccTrue` or `ccFalse` depending on whether a given element of a linked list exists. An error code is thrown if the `memory` field of the linked list is `NULL`.

`findElement()`

syntax: (void *) *element_pointer* = `findElement`((linkedlist *) *LL*, (numeric) *element_index*)

Returns a pointer to the given element of a linked list, or `NULL` if the element doesn't exist.

`element()`

syntax: (void *) *element_pointer* = `element`((linkedlist *) *LL*, (numeric) *element_index*)

Returns a pointer to the given element of a linked list. Identical to `findElement()` except that `element()` doesn't do range checking: if the index is out of range then `element()` will simply crash the program.

`skipElements()`

syntax: (void *) *element_pointer* = `skipElements`((linkedlist *) LL, (sublistHeader *) sublist, (void *) *starting_pointer*, (ccInt *) *sublist_index*, (numeric) *indices_to_skip*)

Only for real pros. Starting from a pointer to an element in the linked list, 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 `skipElements()` than to call `element()`, which searches from the first sublist each time it is called. The first three arguments are the linked list, a pointer to our current sublist and a pointer to the local index of our starting element (beginning at 0 for the first element in the sublist). The new element must have a higher index than the old; Cicada's linked lists cannot be explored in reverse.

5.6 Errors

When an error happens in a Cicada script, one of two things happens. If the error happened inside of a `trap()` function, then Cicada falls out of the `trap()` call, printing an error message if `trap()`'s arguments began with a `code` marker or semicolon. If there was no `trap()`, then the script crashes with an error message. Table 5 lists the error message by error code (the number passed to `throw()` and returned by `trap()`). The rest of this section explains each error message, in alphabetical order.

Argument expected (#12)

There was an empty expression where there shouldn't be. For example, the expression `5+()` causes this error because there was nothing inside the parentheses.

`call()` **can't find C function (#44)**

A `call()` statement or `$myCFunction()` command 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). 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.

Cannot step to multiple members (#27)

Cicada tried to step into two or more members at once. This is not allowed; the user can step to multiple indices of an array, but never two members. So, for example,

```
a :: [2] double
a[<1, 2>]
```

ID	name	ID	name
0	passed (no error)	26	member is void
1	out of memory	27	cannot step to multiple members
2	out of range	28	incomplete member
3	initialization error	29	incomplete variable
4	mismatched indices	30	invalid index
5	error reading string	31	multiple indices not allowed
6	error reading number	32	invalid index
7	overflow	33	variable has no parent
8	underflow	34	not a variable
9	unknown command	35	not a function
10	unexpected token	36	not composite
11	[token] expected	37	string expected
12	argument expected	38	illegal target
13	left-hand argument expected	39	target was deleted
14	right-hand argument expected	40	unequal data sizes
15	no left-hand argument allowed	41	not a number
16	no right-hand argument allowed	42	overlapping alias
17	type mismatch	43	nonexistent Cicada function
18	illegal command	44	call() can't find C function
19	code overflow	45	wrong number of arguments
20	inaccessible code	46	error in argument
21	jump to middle of command	47	self reference
22	division by zero	48	recursion depth too high
23	member not found	49	I/O error
24	variable has no member	50	[return flag]
25	no member leads to variable	51	[exit signal]

Table 5: Error messages, by error code

is legal, whereas

```
b :: { one :: two :: double }
b[<1, 2>]
```

causes an error on the second line.

Code overflow (#19)

`transform()` was given bytecode that 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.

Division by zero (#22)

This warning is caused when the user tries to divide by zero at runtime. Cicada still performs the division, resulting in either infinity or a not-a-number value. And if on some machines a divide-by-zero crashes the computer, then the computer will crash.

Error in argument (#46)

There was a problem with a parameter passed to a built-in function. This is a catch-all error for miscellaneous problems with arguments. It can be caused by: `compile()` if the fourth (variable-names) argument is not a string array, or `transform()` if the bytecode string length is not a multiple of the size of an integer.

Error reading number (#6)

The compiler tried to read a number that didn't follow the expected format. This can be caused by the compiler or by the `read_string()` function.

Error reading string (#5)

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. Certain special characters can be encoded with escape sequences: `\r` (carriage return), `\t` (tab), `\n` (end-of-line) and `\\` (backslash). General characters can be encoded using hexadecimal codes beginning with a backslash (e.g. `\3D` is an equals character).

Exit signal (#51)

This error code, thrown by the `exit` command, is a bookkeeping device that causes Cicada 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 51. Typing `throw(51)` is equivalent to typing `exit`.

I/O error (#49)

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.

Illegal command (#18)

Cicada found a nonsensical command in bytecode that it was trying to transform into memory. For example, the string-constant bytecode command has a string length word – if the string length is less than zero then this error will be thrown. Or, Cicada may hit a command ID that simply doesn't exist in any of its tables, which will also cause this error.

Illegal target (#38)

Cicada tried to make an alias to something other than a variable or the void.

Inaccessible code (#20)

This warning is thrown by `transform()`. It indicates that a null end-of-script bytecode word was encountered before the end of the bytecode was reached. The code will still run, but the spurious code-terminating marker will prevent the last part of the bytecode from being used.

Incomplete member (#28)

There are two situations that cause this error. The first is that the user tried to redefine or re-alias a part of an array — these are operations that must be done to the entire array. For example:

```
myArray :: [5] int
myArray[3] = @nothing     | not legal
```

will cause this error. The second scenario is that we tried to step into only some indices of an array starting from a range of indices. This is due to a technical limitation – Cicada is only able to reference contiguous blocks of memory. So once we step into more than one index of an array, each subsequent step must be into the full range of indices. So this code

```
b :: [5][2] double
print( b[<1, 3>][1] )        | causes an error
```

also causes an incomplete-member error.

Incomplete variable (#29)

Only a partial variable was used for some operation that can only be done on an entire variable: redefining or re-aliasing, or adding/removing indices. For example, for a hypothetical array `a :: [3][2] int` the following causes an incomplete-variable error.

```
    a[1][^4]
```

If we want to resize the second dimension we have to do that for the entire array: `a[*][^4]`.

Initialization error (#3)

An uninitialized linked list was passed to a linked list routine (other than `newLinkedList()`). An uninitialized linked list is marked by a null pointer in its `memory` field. To initialize a list one must first clear the `memory` field, then make a successful call to `newLinkedList()`..

Invalid index (#30, #32)

The most common cause of this error is that the user requested an index of an array that does not exist: e.g. `array[5]` when it only has only four indices, 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. (However `array[<4, 3>]` is allowed and just returns zero indices.) This error also is thrown if we resize an array to a size less than zero, or try to add a huge number of indices (more than `INT_MAX`).

The fact that this error has two error codes is irrelevant to the user, having only to do with the way Cicada keeps its books.

Jump to middle of command (#21)

`transform()` found a goto statement pointing somewhere other than the beginning of a bytecode command. The specific bytecode commands that can cause this error are: jump-always, jump-if-true and jump-if-false.

Left-hand argument expected (#13)

An operator is missing its left-hand argument. For example,

```
    a = + 3
```

will cause this error because the `+` operator expects a number to the left, after the equals sign.

Member is void (#26)

Cicada attempted to step into a void member (one that has no target). Here is the most common sort of situation that will cause a void-member error:

```
    x :: *
    x = 2
```

Many built-in functions throw this error if any of their parameters are void.

A void-member error can also happen when we try to use an 'unjammed' member: a member defined as unjammable whose target variable was later resized. This typically only happens to members of sets. For example, this code will unjam the only member of `set` and cause a void-member error:

```
array :: [5] int
set :: { array[<1, 5>] }

remove array[3]       | unjams the alias in our set
set[1][1] = 2         | the second '[1]' causes the error
```

Had we defined the member of `set` explicitly:

```
set :: { alias := @array[<1, 5>] }
```

we would have gotten an overlapping-alias error when we tried deleting the array element.

Member not found (#23)

The user gave a member name that Cicada could not find. If the missing member is at the beginning of the path, for example if it flags member `list` in

```
print(list[5])
```

then Cicada is telling us that that member (`list`) was not to be found anywhere along the search path: the current function or any enclosing object up through the workspace. If the problematic member was some intermediate point in the path, for example if `list` is flagged in

```
data[5].list = anotherList
```

then the missing member (`list`) was not immediately inside the given path (`data[5]`).

If possible, Cicada gives the member name in single quotes along with the error message, as in: "`member 'header' not found`".

Mismatched indices (#4)

Usually, this error means that 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>]
```

Notice that this can happen when working inside of an array; for example:

```
q :: int
threeQs :: [3] { qAlias := @q }
```

which is basically equivalent to `threeQs[*].qAlias := @q`.

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] int
r :: [2][2] int
q[*] = @r[*][*]
```

If the last index of the array on the left-hand side of a compare or equate is a '`[*]`' or '`[]`', then Cicada 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] string
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.

A mismatched-indices error can also be thrown by the linked list routines (used for working with Cicada strings in C). The only scenario where this would happen is when the two linked lists have different element sizes. Strings always have an element size of 1 (the byte size of a character), so this should never happen unless the linked list routines are being used for something else.

Multiple indices not allowed (#31)

Several instances of a variable were given where only one was expected. Whenever Cicada expects either a number or a string, that quantity has to be a constant, a single variable or a single element of an array. For example, the following causes this error.

```
if 4 < a[<1, 2>]  then print("dunno how this worked")
```

Likewise, the following expression is also (at present) disallowed for the same reason.

```
a :: [2] { b :: int, if 4 < b  then ... }
```

Many built-in Cicada functions will throw a multiple-indices error when passed an array parameter when only a single number or string was expected.

No left-hand argument allowed (#15)

An operator had a left-hand argument that it was not allowed to have. For example, the following line

```
10 return
```

will cause this error because it interprets the `return` statement as having a left-hand argument `10`. `return` is only supposed to have an argument on the right.

No member leads to variable (#25)

Cicada attempted an operation that involving 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() :: double
```

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 either using `[^...]`, or via a `[*]` or `[]` operator. (That is because both the variable and the member have to be resized.) Finally, the insertion and removal operators operate on members and therefore will generate this error if none is provided.

No right-hand argument allowed (#16)

An operator had an argument to its right that it was not allowed to have. Depending on precedence, this problem can also cause a no-left-argument-allowed error.

Nonexistent Cicada function (#43)

The user invoked a built-in Cicada function with a function ID that does not exist. Cicada functions are those functions like `call()` and `trap()`, and these are distinguished in bytecode by a function ID immediately following built-in-function command. This number must fall in the range 0-29 because there are 30 built-in functions; any other number generates this error. This error should only happen if the user modified the language, or else wrote or modified bytecode by hand.

Not a function (#35)

The user tried to do something involving code with an object that doesn't have code. For example:

```
    transform(compile(""), 2)
```

causes this error because `transform()` cannot put the transformed code into the primitive variable storing '2'. This error message is technically redundant with the 'not-composite' error message, because every composite object (i.e. defined using curly braces) is a function and vice versa.

Not a number (#41)

Cicada 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 "type mismatch".

Not a variable (#34)

We tried to do some variable operation on a non-variable. For example, the define operator operates on both a member and a variable, so trying

```
nothing :: string
```

will cause this error. Likewise, both equate and forced equate require an existing variable to copy data into. Likewise, the comparison operator '`==`' requires two arguments that are either constants or variables. Code substitution and alias-comparisons both involve variables and can cause this error.

Not composite (#36)

Cicada expected a composite variable (i.e. one defined using curly braces), but was given a primitive variable instead. All of the 'step' operators – '`.`', '`[]`', '`[^]`', '`[+]`', etc. – must start from some composite variable. For example, the following generates a not-composite error:

```
a :: double
print(a.b)
```

This error can also be thrown by a number of built-in Cicada functions such as `top()` and `transform()`, which expect certain arguments to be composite.

Out of memory (#1)

Cicada 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. Cicada 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 Cicada'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.

Out of range (#2)

A number was used that was not within expected bounds. This error is often thrown as a warning. One common cause of this is that the user assigned a value too large or too negative for a given type, for example

```
(a::char) = 400
```

This warning is *not* caused by rounding-off errors: for example, we can assign the value 1.9 to an integer variable, and Cicada will quietly round it off to 1 without raising any warning flag.

`transform()` can throw at out-of-range error (an actual error, not a warning) if some element of the character-positions list (optional 6th argument) is not within [1, *num_chars*]. It can also be caused by certain defective numbers passed to `setCompiler()`: a negative number of commands or precedence levels (2nd/4th arguments); a precedence level outside of bounds; an operator direction other than `l_to_r` or `r_to_l`; or an argument number or jump-to number in the bytecode (e.g. '`a5`' or '`j4`') beyond the number of arguments/jump positions (or outside the range 1-9). Finally, this error can be thrown by the linked list routines if a nonexistent/nonsensical element number of a list is passed (although `InsertElements()` allows the insertion point to be one greater than the top element). Remember that linked list indices begin at 1, not 0.

Overflow (#7)

The compiler tried to read a numeric constant that was larger 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.

Overlapping alias (#42)

The user tried to do resize an array some of whose elements are also aliased elsewhere. For example:

```
a :: [5] int
b := @a[<2, 4>]

a[+3]
```

Resizing one member ('`a`') does not affect any aliases ('`b`'), so there would be a contradiction in the array if the resize were allowed. Notice that an alias to the array variable (as opposed to certain indices of the integer variable), as in

```
c := @a
```

does not have this problem.

Recursion depth too high (#48)

Too many nested functions are being run. In order to avoid blowing the program stack, Cicada sets a limit to the number of nested functions that can be run inside one another. This limit is set in the `glMaxRecursions` variable at the top of bytecd.c – Cicada 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 defining '`f`' requires an infinite level of recursion. (`f` creates `f2` which creates its own `f2`, etc.)

Return flag (#50)

This error code is used internally to cause Cicada 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. Writing `throw(50)` in a script is the same as writing `return`.

Right-hand argument expected (#14)

An operator requiring a right-hand argument does not have one. For example,

```
a = 5 +
```

causes an error because the `+` operator requires a number or expression to the left *and* the right.

Self reference (#47)

A variable with an alias to itself was given to an operation in which self-aliases are not allowed. It is fine for a variable to have aliases to itself, but we cannot sensibly, 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.

String expected (#37)

A built-in Cicada function requiring a string argument was passed something that was manifestly not a string. For example, `call()` expects either a string or a number as its first argument to specify the function to run, so writing

```
call(true)
```

will generate this error.

Target was deleted (#39)

A member was removed while it was being aliased. One has to try hard to get this error.

[Token] expected (#11)

A multi-token command (like '`while`' followed by '`do`') was missing one of its parts. So a typical error message would be

```
> while true

Error:  'do' expected
```

Notice that the error message contains the name of the token that is missing.

Type mismatch (#17)

A member or variable does not have the expected type. This error can occur either when data is being copied or compared, when a variable or member is having its type altered (e.g. via the define operator), or when a built-in function is run with the wrong argument types.

When two variables are copied or compared, Cicada requires that they have identical structures with the proviso that numeric types are interchangeable. So 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 array member must have the same number of indices. Here are some commands that don't work:

```
{ string, int } = { int, string }
{ [5] int } = { [4] int }
string = char
```

and nor does substituting specific `int`-, `char`- and `string`-typed variables work. Numeric members in one variable must correspond to numeric variables in the other, strings with strings, Booleans with Booleans.

Cicada is also fastidious about redefining members and variables: it doesn't care about the structure of the variables, but it does require that their old types exactly match or be compatible with the new types. A type can only change by being updated, using the inheritance operator. Redefining a variable as a different numeric type, as in

```
myNum :: int
myNum :: double
```

will cause this error, even though copying/comparing different numeric types (e.g. `{ int } = { double }`) is legal.

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`.

Underflow (#8)

A number was encountered which was so small that it was read as zero. `1e-400` will do the trick on most machines.

Unequal data sizes (#40)

Cicada was not able to perform a forced equate because the byte-sizes of the left- and right-hand arguments were different. For example, the following fails:

```
(a :: double) =! (b :: int)
```

even though a normal equate would have worked in this situation.

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 string 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, Cicada may not able to figure out how to resize the array correctly in which case the user must resize the array manually.

Unexpected token (#10)

The compiler encountered a token that usually follows another token, but didn't. For example, writing a `do` without a `while` will cause this error.

Unknown command (#9)

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 Cicada.

Variable has no member (#24)

A step was attempted into a variable without a member. For example `{}[*]` will cause this error. This error is also thrown if we use the `top` keyword (which is different from the `top()` function!) anywhere outside of array brackets.

Variable has no parent (#33)

A pathname tried to step to `parent` or `\` when it was already at the beginning of the search path. Just typing `parent` at the command prompt will cause this error.

Wrong number of arguments (#45)

A built-in Cicada 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