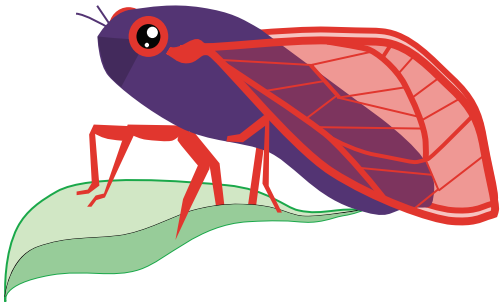
Cicada Scripting Language

Brian Ross

December 15, 2025



Contents

1 Summary

Cicada is a **small interpreted language** that plugs into C (or C++) code. That gives you both:

- the conveniences of a scripting language (memory safety, allocation/deallocation, interactivity);
- the speed and low-level functionality of C.

Embedding Cicada into a C program:

1. Include the `cicada.h` header file.
2. Run Cicada by making a C function call: `runCicada(f, s, r)`. Here `f` is an array of C functions to expose to Cicada; `s` is any script to run; and `r` tells Cicada whether to run an interactive terminal.
3. Cicada can callback other C functions in your code. *This last step is how the two languages communicate*: the callback passes Cicada variables and arrays into the C code for fast processing.
4. It's recommended to write a Cicada wrapper for each C function, to ensure that the arguments get passed into C correctly.
5. Pass an `-lcicada` option to the linker.

* * *

Here's an example:

```
#include <cicada.h>

ccInt countOddsF(argsType args)     // (for positive numbers)
{
    ccInt *n, *c, i;                // by default, ccInt = int
    getArgs(args, &n, &c);
    *c = 0;
    for (i = 0; i < args.indices[0]; i++)  *c += n[i] % 2;
    return 0;
}

const char *wrapperScript =
    "numOdd :: { n :: [] int, c :: int; n[] = args[1], $countOdds(n, c), return c }";

int main(int argc, char **argv)
{
    const Cfunction fs[] = { { "countOdds", &countOddsF } };
    return runCicada(fs, wrapperScript, true);
}
```

Running this program lets us interactively process any data we want. And since the counting is done in C, it'll run fast even on a huge dataset.

```
> numOdd({ 3, 5, 6, 7, 8 })

3

> if numOdd(bigList) > 1e6  then ...
```

2 Example: memory networks in Cicada

Cicada originally aimed to be a neural network engine, missed by a mile, and ended up as a scripting language. So let's use it to implement an *associative neural network*: a type of pattern-learning algorithm which will learn to associate inputs with desired outputs.

Algorithm: We'll use a training algorithm for symmetric networks[1] which trains in two phases: *clamped*, and *free*. In the *clamped phase*, set the input neuron activities to the network input, and set the output neuron activities to the pattern we are trying to learn. Then repeatedly adjust the other 'hidden' neurons' activities x_i using the formula:

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

Here w_{ij} is the connection weight from neuron j to neuron i, and b_i is a bias. Once the network settles into a steady state, apply the following training rule to the weights and biases:

$$w_{ij} \leftarrow w_{ij} + \eta \cdot x_i x_j$$
$$b_i \leftarrow b_i + \eta \cdot x_i.$$

Next, repeat the process for the *free phase* of operation, where that the activities of the output neurons update freely using the same rule as the hidden neurons. After reaching steady state, a *negative* weight/bias adjustment is applied. The idea is to reinforce correlated firing patterns on the difference between what the network should vs. does currently compute. As the saying goes, *neurons that fire together, wire together.*

To use the network to recover a memory, run it in its 'free' mode without applying any training.

2.1 C implementation

The code that trains a network will be written in C, for speed. Then we'll wrap Cicada functions around the C code so that we can easily call it from a script.

NN.c

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

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


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


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

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

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

    if ( args.num == 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;         // no error
}


// 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];
    }}}
}
```

NN.h

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

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

2.2 Putting the C in Cicada

Our C code focuses on the training loops (where speed is crucial) but memory allocation/deallocation, dataset loading, saving of results, etc. are all more easily scripted. In fact let's just have C immediately hand off control to the Cicada interpreter, and tell Cicada to callback `runNetwork()` as needed. To do this, include the Cicada header in `NN.c` and `NN.h`:

```
#include <cicada.h>
```

and add following to `NN.c` (say, just before `runNetwork`):

```
const Cfunction fs[] = { { "runNetwork", &runNetwork } };

int main(int argc, char **argv)
{
    return runCicada(fs, NULL, true);
}
```

We also need to adjust the definition of `runNetwork()` so that Cicada can call it properly. The new function prototype (in both `NN.c` and `NN.h`) is:

```
ccInt runNetwork(argsType args)
```

An `argsType` variable holds a list of Cicada variables passed into a C function call. The `ccInt` return type

is user-definable but defaults to just `int`.

Finally, we need to set up our variables at the beginning of `runNetwork()`, using Cicada's functions for a) checking and b) loading data/pointers from `args`. Since memory is shared between the two environments, any pointers we load can be used to send data back to Cicada. Replace the 'set up data types' block comment with the following code:

```
myNN.numNeurons = args.indices[1];
numInputs = args.indices[2];

if (args.num < 4)  return 1;
if (getArgs(args,
        arrayRef(double_type, &myNN.weights),
        arrayRef(double_type, &myNN.activity),
        arrayRef(double_type, &inputs),
        scalarValue(double_type, &step_size),
    endArgs) != 0)  return 2;

if (args.num == 6)  {
    numOutputs = args.indices[4];
    if (getArgs(args, fromArg(4),
            arrayRef(double_type, &target_outputs),
            scalarValue(double_type, &learning_rate)
        ) != 0)  return 3;
}
```

Our code loads pointers for `myNN.weights`, `myNN.activity` and `inputs`. There are two reasons to load these parameters by reference: 1) they are arrays not scalars, and 2) our C code will modify the values of `activity` and `weights` and we want those changes to persist outside of the function call. However, `step_size` and `learning_rate` are constant scalar parameters, so we load their data by value using the `scalarValue()` macro.

The C code doesn't need to save any results, since its data is shared with Cicada, and there are scripting functions for writing to files (e.g. `save()` or `saveTable()`). Just delete the "save results" comment line in `NN.c`.

It's time to build our program. To link against the Cicada library use the linker flag `-lcicada`. (For example, using `gcc` on a UNIX machine the command would be: `gcc -lcicada -o NN NN.c`). With luck, we'll end up with an executable which we can run from the command prompt (by typing '`NN`' or '`./NN`', depending on the system). We should see:

```
>
```

2.3 Writing and debugging a Cicada wrapper

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

```
> $runNetwork(...)
```

The main danger here is that we might pass the wrong datatypes, or arrays of the wrong sizes, into our C code, which will then promptly crash. To do things properly, let's write a Cicada class that stores the neural network data and provides methods for initializing, running and training the network.

This will be a learning exercise. It's our first time scripting so there will probably be a few bugs.

NN.cicada

```
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[^numNeurons][^numNeurons]
    }


    process :: {

        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.process(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("process() did not converge; try lowering step size?\n")
    }
```

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

Save `NN.cicada` in the same directory as our NN program. Then, from Cicada's command prompt, let's try out our new wrapper by typing:

```
> run("NN")

Error:  left-hand argument expected in file NN

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

What we see here is a 'compile-time' error (i.e. it failed to produce bytecode). Evidently we wrote a C-style comment '`//`' in place of a Cicada comment '`|`'. Make the straightforward fix to `NN.cicada`.

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

and try again.

```
> run("NN")

Error:  member 'numHiddens' not found in file NN

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

This is progress: at least `NN.cicada` is syntactically correct. Line 4 tried to define `numInputs` and `numOutputs` to be of type `numHiddens`, rather than defining all three variables as type `int`, so let's fix that:

```
        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 `init()` produced an odd usage message despite never having been run. But at least a `neural_network` object was constructed, so we can start looking around inside, using the command prompt as a sort of debugger. `init()` is suspicious so let's see if it works when we do 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. To take a better look around, let's 'go' inside our network.

```
> go(neural_network)


> weights

{  }


> numNeurons

0


> go()
```

The last line takes us back to our 'root' workspace.

So our `init()` call was a dud – nothing happened. Our next step might be to put a trace statement in the coding section of the `init()` function.. hmm, wherever that is.. Looks like we forgot a `code` marker separating the function variables from its executable code, which fully explains why it won't run. The `init()` method should begin:

```
    init :: {

        code

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

Making that final change, 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 begin training.

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. To demonstrate, let's build 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; mode = 0)    )

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


    ask :: setupNN : {

        outputString :: string


        code

        process(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]>  &
            process(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).

```
> 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
```

Voilà—it successfully reconstructed a pattern.

A little bit of playing around shows that our little network can actually learn a few words at a time. Next we might rebuild it to learn longer words, or see if adding more hidden neurons increases its vocabulary – the experimental cycles are very short with a command prompt. There are also the training parameters – is 10 rounds of training on each word too many? Will our network learn faster if we increase the learning rate, or will it become unstable? It's simple to test.

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

One final detail: it's annoying to have to type "`run("NN"), run("anagrambler")`" every time we start our program. Going back to `NN.c`, let's write a script that runs those commands and have `runCicada()` run that script before presenting the command prompt. `NN.c`'s `main()` function now reads:

```
int main(int argc, char **argv)
{
    const char *startupScript = "run(\"NN\"), run(\"anagrambler\")";
    return runCicada(fs, startupScript, true);
}
```

Now our compiled application is basically a version of Cicada that has a C-coded Anagrambler engine built into the language.

3 Calling Cicada from a C program

Before we can use Cicada, we have to install it as a C library. Go to the download directory and run:

```
./configure && make && make install
```

Once installed, Cicada can be called from any C program. Here's how:

- Load the Cicada header file into any source file that needs it:

```
#include <cicada.h>
```

- Add the following linker flag:

```
-lcicada
```

- The actual function call to run Cicada is:

```
runCicada(fs, myScript, runTerminal);
```

The three arguments to `runCicada()` are:

1. `fs` is an array of `const Cfunction` variables, each corresponding to a C function in our code that Cicada will need to call. Each `Cfunction` is 1) a string which is the function's name in Cicada, followed by 2) a pointer to a function in our C code of type `ccInt f(argsType)`. For example:

```
ccInt f1_in_C(argsType args)  { ... }
ccInt f2(argsType args)  { ... }
const Cfunction fs[] = { { "f1_in_Cicada", &f1_in_C }, { "f2", &f2 } };
```

   The C/Cicada names may be the same or different. `Cfunction` and `argsType` are defined in `cicada.h`.

2. `myScript` is a C string that gives a script to run when Cicada starts up. For example:

```
char *myScript = "run(\"ThingsToDo\")";     // i.e. ThingsToDo.cicada
```

   If we don't want to run a predefined script, we can just pass an empty string or `NULL`.

3. `runTerminal` is a `bool` value and determines whether to run an interactive terminal. If it's `true`, Cicada will open an interactive prompt *after* running `myScript`. If `false`, Cicada will exit after running `myScript`.

3.1 C function declaration

A C/C++ function needs to be of type `(ccInt)(argsType)` in order to be called from within Cicada. `ccInt` will just be C's basic `int` type unless you go in and change it inside of `cicada.h`. The `argsType` variable contains all the variables and arrays passed from Cicada into the C function, along with information about their datatypes and array sizes. Each variable or array passed to the C function is a list of one or more Booleans (one per byte), characters, integers, floating-point numbers, or strings. The number of elements in each argument is fixed; the only dynamic memory allocation intended for C is through resizable strings, as each string is stored in a linked list (see the reference section).

This is the structure of an `argsType` variable named '`a`':

type # (macro)	Cicada type name	C type name	default data type
0 (bool_type)	bool	bool	bool
1 (char_type)	char	char	char
2 (int_type)	int	ccInt	int
3 (double_type)	double	ccFloat	double
5 (composite_type)	{ }	N/A	N/A
6 (array_type)	[ ]	N/A	N/A
7 (list_type)	[[]]	N/A	N/A

Table 1: Cicada data types. Types 0 - 3 are primitive

```
a.num           // the number of arguments
a.p[i]          // a pointer to the i'th argument
a.type[i]       // an integer giving the primitive type of the i'th argument
a.indices[i]    // the number of array elements in the i'th argument (=1 if it is a scalar)
```

Table 1 lists all of the Cicada types, although our C function will only ever see the primitive types (0-4). Composite (aka structure) types are broken into their constituent primitive arrays before being passed into a C function, and multidimensional arrays are unrolled into 1D lists.

Here is a Cicada-compatible C function that has one argument of each possible data type. It also checks a) that the correct argument types are being passed, and b) that the first two arguments are scalars.

```
ccInt myFunction(argsType args)
{
    bool firstArg = *(bool *) args.p[0];
    char secondArg[2];
    ccInt *thirdArg = (ccInt *) args.p[2];
    ccFloat *fourthArg = (ccFloat *) args.p[3];

    secondArg[0] = ((char *) args.p[1])[0];        // copy array data manually here
    secondArg[1] = ((char *) args.p[1])[1];

    if (args.num != 4) return wrong_argument_count_err;
    if ((args.type[0][0] != 0) || (args.type[1][0] != 1)
                || (args.type[2][0] != 2) || (args.type[3][0] != 3))  return type_mismatch_err;
    if ((args.indices[0] != 1) || (args.indices[2] != 1))  return type_mismatch_err;

    ...

    return 0;         // no error
}
```

This function copies its first two arguments by value and the latter two arguments by reference. We must pass an argument by reference if it might be changed in the C code. Arrays are usually passed by reference for efficiency.

Our C function used two Cicada-defined types: `ccInt` and `ccFloat`. By default, these two types correspond to `int` and `double`; however these definitions can be changed in `cicada.h`. If we change `ccInt` we should also make changes to `ccIntMin`, `ccIntMax`, `printIntFormatString`, and `readIntFormatString`; and if we change `ccFloat` then we should also make corresponding changes to `maxPrintableDigits`, `printFloatFormatString`, `print_stringFloatFormatString`, and `readFloatFormatString`.

There is a handy `getArgs()` function which simplifies the loading of C variables.

```
ccInt myFunction(argsType args)
{
    ...
    if (args.num != 4) return wrong_argument_count_err;

    if ((args.type[0][0] != 0) || (args.type[1][0] != 1)
               || (args.type[2][0] != 2) || (args.type[3][0] != 3))  return type_mismatch_err;
    if ((args.indices[0] != 1) || (args.indices[2] != 1))  return type_mismatch_err;

    getArgs(args, byValue(&firstArg), byValue(secondArg), &thirdArg, &fourthArg);

    ...
}
```

The first argument of `getArgs()` must be the argument variable; following that is the address of each variable to load, using the `byValue()` macro for each C variable that is not a pointer variable.

Using a different set of macros, we can both load the arguments and perform type-checking:.

```
ccInt myFunction(argsType args)
{
    ccInt errCode;

    ...

    if (args.num != 4) return wrong_argument_count_err;

    errCode = getArgs(args, scalarValue(bool_type, &firstArg), arrayValue(char_type, secondArg),
                scalarRef(int_type, &thirdArg), arrayRef(double_type, &fourthArg));
    if (errCode != 0)  return errCode;

    ...
}
```

This example uses all four macros that check-while-passing arguments. The `scalarValue()` and `arrayValue()` macros copy data into their arguments; the other two copy references (pointers). The `scalarValue()` and `scalarRef()` macros enforce that the number of indices should be one; the other two do not.

If we only want to load certain selected arguments, we can use `getArgs()` to skip to an argument, and `endArgs` to stop loading arguments. Here are some possibilities:

```
getArgs(args, byValue(&firstArg), byValue(secondArg), endArgs);
getArgs(args, fromArg(1), byValue(secondArg), &thirdArg, endArgs);
getArgs(args, fromArg(1), byValue(secondArg), fromArg(3), &fourthArg);
```

Notice that `fromArg()` uses *C-style indexing*, so the next argument read after `fromArg(1)` is `args.p[1]` which is the *second* argument.

Let's look an example script that calls '`myFunction()`'. A typical script first sets up variables and arrays to pass, and then calls the C routine using its Cicada name (the `Cfunction` string passed into `runCicada()`) prefaced by a dollar sign. We'll assume the Cicada name is the same as the C function name.

```
(dbls :: [2][5] double) = { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9} }
$myFunction(true, "hi", 5, dbls)
```

This is pretty straightforward: two of the arguments are constants, ``hi" is a string of two characters, and

`dbls` is a two-dimensional array. It should be noted that `dbls` 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] (in Cicada notation where indices begin at 1). So, for example, the array variable written in Cicada as `dbls[2][3]` is in memory slot `fourthArg[7]` within our C code. `dbls` has 10 array elements in total, so `args.indices[3]` will be 10. The individual dimensions (2, 5) were *not* passed into C, so if our function needs to know them, then we need to pass them as separate parameters.

3.1.1 Passing strings and other lists

A Cicada string is a 'list' of characters, the list being a special datatype roughly analogous to an array. Therefore we can read a string into C the same way we read in a character array, e.g. using the `arrayRef()` macro, as in the examples above. This method lets us both read and modify the characters in the string, since the characters were passed into C by reference. However, it does not allow us to resize the string.

In order to pass a string (or other list) that our C function can resize, we have have to instruct Cicada to pass more information about the string than just the pointer to its first character. This is done in the C function declaration passed to Cicada; for example:

```
const Cfunction fs[] = { { "myFunction:dadd", &myFunction } };
```

The argument styles are specified after the Cicada function name: '`d`' to pass argument data, and '`a`' to pass a resizable argument. '`d`' is the default when argument styles are not specified.

Inside the C code, a resizable argument is represented by an `arg` pointer, and requires special functions to access. Here is how we might modify our example to convert the string to a C-style string with a null terminator.

```
...
arg *secondArg;
char *secondArgChars;
ccInt numChars;
...

getArgs(args, byValue(&firstArg), &secondArg, &thirdArg, &fourthArg);

numChars = 1;
stepArg(secondArg, 1, &numChars);
setStringSize(secondArg, 1, numChars+1, &secondArgChars);
secondArgChars[numChars] = 0;
```

We could have also written `setMemberTop()` instead of `setStringSize()` – same arguments, identical functions, only a little bit clearer to write one or the other depending on context.

The other two functions that act on `args`-type variables are `getArgTop()` for obtaining the number of indices spanned by an argument, and `argData()` for getting the data pointer. For example, we could also obtain `secondArgChars` by writing

```
secondArgChars = argData(secondArg);
```

as long as we call this function *after* resizing `secondArg` (because the pointer will change).

When we pass a whole argument we obtain an object that may have a more complex type than one or more `bool`, `char`, `int`, or `double` variables. In the example above the argument was a list of characters, i.e. a list variable pointing to a character variable. We can check these more elaborate datatypes in several ways:

```
bool passedCheck = false;
```

```
if (args.type[1][0] == list_type)  passedCheck = (args.type[1][1] == char_type);
if (!passedCheck)  return type_mismatch_err;
```

or

```
errCode = getArgs(args, fromArg(1), scalarRef(listOf(char_type), &secondArg), endArgs);
```

or

```
errCode = getArgs(args, fromArg(1), scalarRef(string_type, &secondArg), endArgs);
```

since `string_type` is shorthand for a list of characters.

There is also a corresponding `array_type` and an `arrayOf()` macro for type-checking. Although multi-dimensional arrays become effectively one-dimensional arrays when passing arguments as data, that is not true when a resizable argument is passed, so if we were to pass the fourth argument as an `arg` pointer its desired type would be `arrayOf(arrayOf(double_type))`.

3.2 A generic wrapper

To finish off, we'll write a pretty generic Cicada wrapper for `myFunction()` that can be adapted to wrap most other C functions. One advantage of a wrapper is that it will let us run `myFunction` by typing something like

```
yf = f(x, y0)
```

rather than

```
$myFunction(param, x, y, calcData)
```

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 this function does.

myFunctionWrapper.cicada

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

    params :: {
        dim1 :: dim2 :: int
        doRecalc :: bool    }


    code

    params = { 2, 5, true }
    if trap(
        { x, y } = args           | mandatory arguments
        (params << args)()        | optional arguments
    ) /= passed  then (
```

```
        print("usage:  yf = f(xStr, y0 [; calcSize/doRecalc = ...])\n")
        return      )

    calc_table[^params.dim1][^params.dim2]

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

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

Our wrapper function has two parts. The first part is everything before the `code` command, which defines the variables used by function `f()`. These include the input and output arguments, optional parameters, and even a calculation table used internally by the C routine (because it's easier to pass storage into C than malloc/free it in C). The most important thing is that the wrapper explicitly defines the type of each variable passed into C, thus ensuring proper communication between the two languages. For example, when we write `f("a", 5)` the integer argument 5 will be converted to a `double` before handing it off to C, as required by `myFunction()`.

The executable part of `f()` begins *after* the `code` marker. First `f()` reads its arguments (within a `trap()` statement so that we can fail gracefully with an error message if the function wasn't called properly). Notice that there are two sorts of function argument: mandatory arguments (`x` and `y`) which are copied straightforwardly from a predefined `args` variable, and optional arguments stored in `params` with default values. The optional arguments be changed using a very peculiar Cicada trick: `f()` *runs its own arguments*, as a function, inside of its own `params` variable. Then it resizes `calc_table`, calls `myFunction()` and returns a result.

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

```
> run("myFunctionWrapper")
```

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; dim1 = that*2, doRecalc = false)
```

Make sure to separate the mandatory and optional parameters using a semicolon, and separate all other arguments or commands using commas.

Most prepackaged Cicada functions are actually Cicada wrappers around C functions, and their source files `ciclib.c` and `defs.c` are a rich source of further examples.

4 Cicada scripting

4.1 Basic Cicada syntax

Simple expressions—variable names, numbers, character/string constants—look pretty much the same in Cicada as they do in C. Character constants 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, the assignment and arithmetic operators are the same between C and Cicada, except that Cicada also provides an exponentiation operator '`^`' (e.g. `2^3` gives 8).

When we get to variable definitions, Cicada syntax starts to look significantly different from C. In C, variables definitions are noncoding statements that tell the compiler how to manage the stack. In Cicada, variable definitions are *commands* that allocate memory on the heap. The basic way to define a Cicada variable is to use the '`::`' operator:

```
x :: int
```

Here `x` is the variable and `int` is the type. The allowed primitive types are `int`, `double`, `char`, `string` and `bool` (see Table 1 on page 15). All Cicada variables *must* be defined before they can be used, and this type cannot change unless the variable is deleted and redefined.

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

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

defines a two-dimensional (5x7) array of floating-point numbers. Square brackets are also used to access array elements, in the same way as in C:

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

However: *in Cicada, array indexing begins at 1.*

It's often 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 file from Cicada's command prompt, 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 the arguments themselves can be modified 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 ... do` and `loop ... until` loops. There is no 'goto' statement. The equality test '`==`' is the same as in C, but the not-equal test uses the symbol '`/=`' which is different from C.

To exit Cicada type "`exit`" at the command prompt. If there is no command prompt then Cicada is probably stuck and we need to force quit.

4.2 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)
```

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 usually 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 the define-equate operator `:=`. 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))`.

All of the information in Table 2 comes from the `cclang.c` source file. That means that anyone can change the Cicada language by just editing that file (see Section 4).

4.3 Variables

Cicada variables are either: primitive variables which store data, composite (aka structure) variables which contain other variables, array variables which contain many identical copies of another variable, or lists which are like ragged arrays. All variables must be be defined before they can be used. Once created, any variable can have its data overwritten (since there are no constant types in Cicada) or its reference reassigned so that it 'points' to another piece of data. Several variables such as `args` are predefined.

4.3.1 Primitive variables and constants

Cicada has four primitive variable types: `bool`, `char`, `int`, and `double`. Strings are lists of `char`s. Types `bool` and `char` each occupy one byte per datum, whereas the sizes of `int` and `double` depend on the definitions `ccInt` and `ccFloat` but default to C ints and doubles respectively.

precedence	commands	symbols	grouping
1	command breaks	`\n , ;`	right to left
2	commands	`return remove` `if for while loop`	N/A

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

4	logical and, or, xor	`and or xor`	left to right
5	logical not	`not`	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

In order to define a variable, use the define '`::`' operator. Several variables of the same type can be defined on the same line.

```
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 string for `string` variables.

The assignment operator can be written either '`=`' as in C or '`<-`'. We'll use the C convention.

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

String constants are inside double quotes, character constants use single quotes, and Boolean constants are `true` and `false`. Unlike in C, Boolean values are not interchangeable with integers: `(i::int) = true` is illegal. However, characters can be treated as ints, so `((c::char) = 'c') = 27` is allowed.

For convenience, we can define and set variables in a single step using the `:=` operator. So we could have written:

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

Numeric types are assumed to be integers unless there's a decimal point, so if we had rounded to `g := 10` and omitted the decimal point, then '`g`' would have had an integer type.

4.3.2 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 as 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 we can also treat it as a data type.

```
PetersPlace :: StreetAddress
```

In general any Cicada variable `A` can be used to define other variables `B` and `C`. In a sense, the define operator copies 'type', but not 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.

```
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"
```

The type specification of a composite variable determines how it looks initially, but its internal structure can be rearranged later on.

```
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 potential pitfall is that these modifications did *not* change the type specification, so if we ever try to copy its type and data in one operation, then we'll have problems.

```
> PaulinesDogAddress := PaulineAddress

Error:  type mismatch
```

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.

4.3.3 Array variables

The third type of Cicada variable is an array variable, which we define by writing the dimension inside square brackets *before* the element type. Multi-dimensional arrays are allowed. Examples:

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

If we omit the dimension, as we did in defining `coordinates`, then a size-0 array is produced. However, it's not quite the same as writing the `[0]` dimension explicitly, because empty brackets tell Cicada that the array is likely to be resized often. This array will then be given extra internal storage so that new elements can be added efficiently.

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 `[]`.

```
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

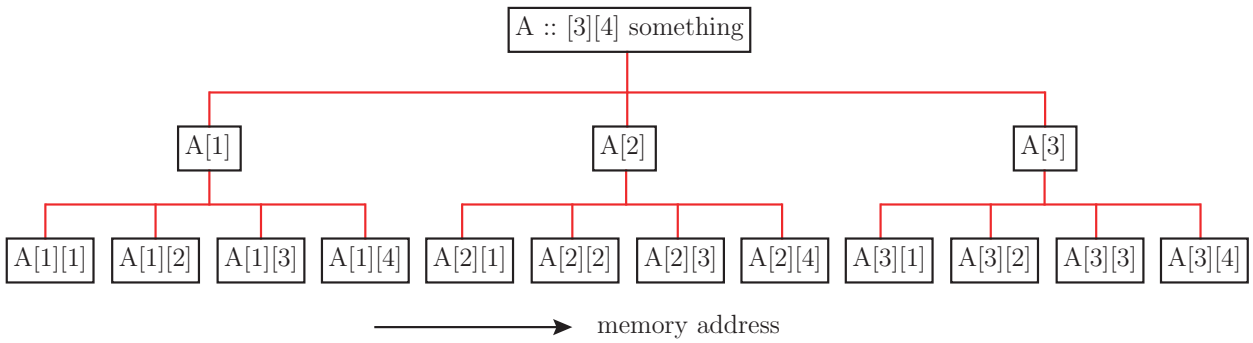


Figure 1: Order of elements in a 2-dimensional array '`A`'

`[<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. 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
```

When addressing part of an array using two or more `[]` or `[<...>]` operators, Cicada loses track of how many elements were in each dimension and thinks 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 with 25 elements.

4.3.4 Lists

Lists are very similar to one-dimensional arrays. Define a list using double brackets:

```
intList :: [[ ]] int
```

There is no option to set an initial list size – initially `top(intList)` is zero. However, if we try to copy data (`intList = { 2, 3, 8 }`), then the list will automatically resize as needed. At this point we can access the list elements using the normal array operators: `intList[2]`, `intList[<1, 2>]`, `intList[]`, etc.

The most important difference between lists and arrays is that an array of arrays is a multidimensional array, whereas in an array of lists each list may have a different size. Therefore the following is legal:

```
twoIntLists :: [2] intList
```

```
twoIntLists[1] = { 3, 4 }
twoIntLists[2] = { 0, 1, 2 }
```

In Cicada a string is defined as a list of characters. Therefore `s :: string` is equivalent to `s :: [[]] char`.

4.3.5 Resizing arrays/lists (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 `[]` 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 defining `B :: A, B[^22]` then

```
A[] = B[]
```

should work by forcing a resize of `A`, whereas

```
A[<1, 5>] = B[]
```

will not work.

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 `[-<..., ...>]` operator 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[-<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]`. On the other hand, if we add a new member by name rather than by index (e.g. `threeNums.d :: int`), that new member always goes at the end of the structure.

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

4.3.6 This and that

The keyword '`this`' acts as a synonym for the current variable or function where code is being executed. From the command prompt, `this` typically refers to the entire workspace. But when a function is running, '`this`' refers to that function. For example:

```
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 fact, `this` refers to the inside of *any* curly braces it appears in. For example, here:

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

`c` will be set to 2.

The really tricky situation is inside the arguments of a function call, where `this` refers to the function arguments themselves. For reasons explained elsewhere, if we write

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

then we are effectively calling `f(2, 5, 5)`. This means that, for example, we cannot print the number of workspace variables by typing

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

because that command will print '`1`' which is the number of arguments passed to `top()`. What we really want to do is back out 2 levels, from the arguments of `top()` 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 should appear only to the right of an assignment operator (`=` or `<-`), is called '`that`', and it 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 something like

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

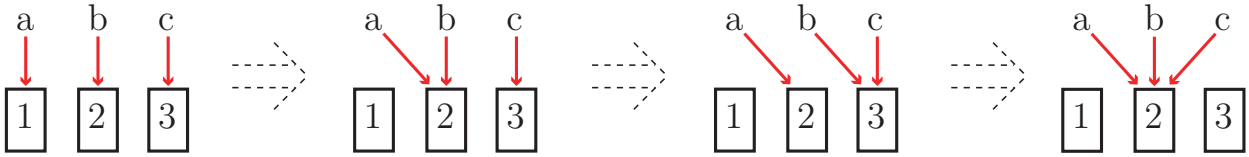


Figure 2: Aliasing of members (letters) to variables (boxes)

The final keyword, called `args`, refers to a function's arguments, which is usually a composite variable. Here's a quick example:

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

If we call `I(1, 2, 3)`, our function `I` sends back `{ { 1, 2, 3 }, 2 }`.

4.3.7 Aliases

Aliases in Cicada are a sort of memory-safe alternative to pointers. An alias is a member that shares its data with some other member: in other words those two members share a variable. In Cicada a member and an alias made to it are *exactly* on equal footing: the interpreter doesn't know or care which came first.

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.

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 permanently inaccessible and will eventually be cleared from memory.

Cicada clearly makes an important distinction between members and variables. 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, as Figure 2 shows.

Just as the data-copying operator '`=`' or '`<-`' has a data-comparison counterpart '`==`', so the 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 '`==@`'. Whitespace just before the '@' is allowed:

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

and can help make 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.

Jamming Arrays can also be involved in aliasing. For example, scalar members can alias single array elements, and array members can alias other array members having compatible types.

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

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

It gets a little more complicated when aliasing array *elements*. This is different from aliasing arrays: `a1[] = @a2[]` is different from `a1 = @a2`. Cicada doesn't let us re-alias only part of the source array, since doing so would fragment its storage, but we can alias all source array elements to a part of a target array. For example,

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

is legal, 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>]
```

There's an additional complication: what happens if we try to resize one of those two arrays? That could force a resize of another alias, which is not allowed. The solution is that parts of an array having multiple members pointing to them can become 'jammed', meaning that they cannot be resized until all but one of the aliases is removed. For 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, 'tokens' or 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.

4.3.8 The void

There is one last variable 'type', which is no type at all. There are two ways of writing this:

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

`nothing` and `*` are synonyms for the void. 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 now that it has a non-void type, `var1` cannot be redefined again except to the same type or a derived type (see inheritance).

What is the use of having void-typed variables, if we can't use them while void? The main reason is that members without no type are *universal aliases* (because all other data types are daughters of the void). Just make sure to alias using the `=@` operator, not the `:=@` operator as the latter will redefine its type.

```
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 to any other type. Cicada always permits an existing member's type to be specialized—restricted to a subtype of its original type—but never changed back to a parent or sibling type. Similar rules apply to aliasing—aliases can be made to variables having inherited types, but not parent types—except that any member is allowed to become unlinked by aliasing to the void.

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, and if the two types are different then the member type must be a parent of the variable type.

4.4 Other define and equate operators

The distinction between member type and variable type 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 currently aliased by this member. 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 an integer variable for the member.

There are actually 256 possible define/equate/alias operators, most of which cannot be scripted directly unless we modify the language. See Section 5.

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 or other list 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 data having numeric constants, remember that Cicada interprets each written number as either as `int` or `double`. So for example, if long integers are 4 bytes and doubles are 8 bytes, `a =! -4` and `a =! 2e5` each copy 4 bytes while `a =! 4.` and `a =! 2e10` 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`.

4.5 Loops and `if` blocks

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

4.5.1 `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 new member definitions and use of the keyword `this` won't do what you expect.)

```
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!")
```

4.5.2 `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"
```

4.5.3 `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(" ")
```

```
)
```

There is no `step` argument to control the increment of the counter variable, due to limitations of the compiler. If we need a non-1 step we have to use a `while` loop instead:

```
counter := 1
while counter <= 10-counter  do (
    print(counter, " ")
    counter = that + 2
)
```

4.5.4 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`.

4.5.5 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. Be wary of pitfalls: if we define a member, that member will be inside of this variable, which was probably not intended. '`this`' now refers to this newly-defined variable, while '`parent`' now refers to the location outside the braces. Also, if this is all inside of a function definition, we can't return from the function until we leave the braces block. On the upside, we have a lot of latitude in choosing where to break from and to: it doesn't have to escape from a loop, and we can jump out of several nested loops at once unlike in C where we always fall back to the enclosing loop.

4.6 Sets

Sets are like magical bags of objects that are also found somewhere else in the code: variables, functions, classes, other sets, whatever. 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 :: Joe :: friend_of_mine

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

We can 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 inlined subsets), 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. `collections[6]` is basically an alias for 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 infinite 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 it has the name `men`—typing `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
```

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 :: { ... }
```

Here is a tricky situation:

```
f :: ...
g :: { ; r :: {}, r[+1]:=args[1], return r }
rtrns :: { f(5), f(12), g(3) }
```

Each set element is a *copy* of a function's return value, as if it was defined using a `:=` operator. Normally this is fine, but here `g()`'s return value '`r`' *cannot be copied* using that operator, because its structure `{int}` doesn't match its original definition `{}`! In this case we can copy the return value 'by reference' instead, by making an anonymous alias:

```
rtrns :: { f(5), f(12), @g(3) }
```

Let's just hope `f()` doesn't have the same problem, because if we alias both `f()` calls then the second call will overwrite the first. This potential pitfall also lurks in function arguments, since an argument list is simply a set passed into a function.

4.7 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 the Cicada menagerie. 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 from their C counterparts. These changes together allow Cicada functions to be used in some unusual ways.

4.7.1 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 10*onesDigit + tensDigit
}
```

The function has two Cicada 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.

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 :: {

    nextFact :: *
    numToReturn :: int

    code

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

    return numToReturn
```

```
}
```

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

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, if we run `f#0()` we rerun `f`'s 'constructor' which will reset `f.num` to 2.

Short functions like `f()` are sometimes easier to code on one line. To do this, use the fact that 1) commas are equivalent to ends-of-lines, and 2) semicolons are equivalent to `code` markers.

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

4.7.2 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. But *after* the `code` marker, '`args`' is the parameter list passed by `doArgs`, 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 predates `g()`, 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 args[1]^2)        | calculate (3^2)^2

81
```

To summarize, Cicada's function arguments are themselves functions. To make this clearer, we can imagine a syntax that uses curly braces for function calls:

```
doTwice{ 3, code, return args[1]^2 }
```

In fact this syntax is legal! The only difference with a regular function call is that the argument space will linger as a new object after the function call, wherever that call happened. (Technically the argument space always persists but is normally hidden). This way of writing a function makes clear that the arguments are just some function object that's ordinarily invisible, but has an `args` alias inside of the running function.

The args variable is usually defined inline, using parentheses or braces, but it can also be aliased from an existing variable or function using the familiar `@` operator. For example:

```
f :: { ; return args + 1 }
g :: { ; return args*2 }
a := 5
print(g @ f @ a)
```

This produces a notation similar to the 'circle' syntax for function composition.

4.7.3 Code substitution

Why would a function ever want to run its arguments? One good reason is that running `args()` is an easy way for the caller to modify optional parameters. To do this, we put all of the function's optional parameters (or aliases to them) in one composite variable, set default values, then run `args()` *inside of that parameters variable* using 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` should be explicitly named. If we wrote:

```
    numSides :: int
    isLoaded :: bool

    params :: { numSides, isLoaded }
```

then `args` would find itself in a space with two nameless members. An alias to `numSides` or `isLoaded` variable is not the same as an alias named `numSides` or `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 old `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, the terminal script 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 wrote all of our commands after a `code` marker, and ran the substituted code as a function.

A code substitution is quite temporary, lasting only as long as the expression evaluation. 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
```

4.7.4 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. We'll show this with some examples:

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

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

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

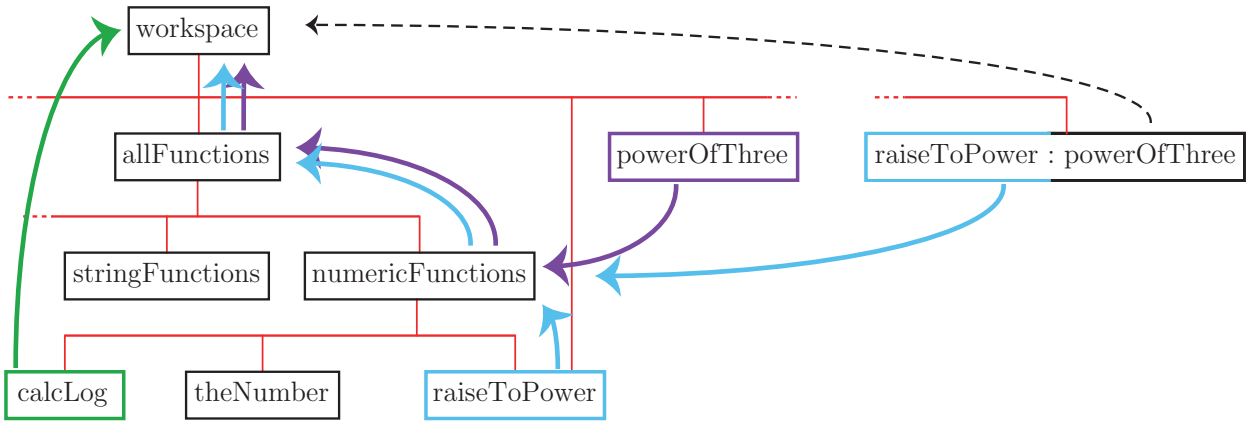


Figure 3: Search paths of various functions from an example in the text. For simplicity we've named each variable/function box by the member that defined it. Search paths are shown with heavy arrows: green arrows for `calcLog()`; light blue arrows for `raiseToPower()`; and purple arrows for `raiseToPower()` 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).

Compare the various functions. The `raiseToPower()` function has a long but straightforward search path, beginning at `raiseToPower`, then passing through `numericFunctions` and `allFunctions` and ending 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's not there it backs 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()` – it's a property of the function, not the member.

On the other hand, `calcLog()`'s search path 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 `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 log(allFunctions.numericFunctions.theNumber)   }
```

Everyone's door is unlocked, but you have to deliberately open the door to go into someone else's data space.

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 a more accurate picture 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.

4.8 Classes and Inheritance

A pair of curly braces can be thought of as a datatype, a composite variable, a function.. or even a class or class object.

4.8.1 Classes

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

```
myClass :: {
    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.

The `init()` function can be thought of as a constructor, since it initializes the class data. It might be more convenient to have that run automatically by taking it out of `init()`:

```
myClass :: {
    data :: int
    moreData :: string

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

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

Now `myClass` and any class instance `myObject :: myClass` begins initialized to `{ 1, "blank" }`. We can even bundle the constructor code together with the member definitions:

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

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

The constructors run when the object is created, but what if we need to rerun the constructor? Treating our object as a function, we just write:

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

Code number 1 of a function, which is 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. Theoretically, when no more members point to an object, that object will be deleted. In practice Cicada doesn't do this very well – we can help it out using its `springCleaning()` function.

4.8.2 Inheritance

Cicada also supports a sort of inheritance. The inheritance operator is a colon '`:`' separating the parent object from the code that specializes it. For example, to derive a class from `myClass` we could write:

```
myDerivedClass :: myClass : {
    newString := "Hello, I am new to this class."
}
```

`myDerivedClass` has all of the members of `myClass`, plus the new string member.

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 that it is possible for a derived object to have *fewer* members than its parent object.

Importantly, an object's type is sensitive to the order of inheritance. `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              | 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 be 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 applied to sets is best thought of as set concatenation.

```
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 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's another way of using the inheritance operator to accomplish the same thing:

```
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. This explains why objects can be specialized but not generalized: the specialization code tells Cicada how to convent a parent into a child object, but not the reverse.

5 Customizing the Cicada language

Cicada's language syntax is described in the `cclang.c` source file, specifically in two arrays that are passed to the compiler when Cicada loads. It follows that, by changing those arrays, we change the syntax of the language. Or by passing new arrays during runtime, our scripts can repurpose the compiler to run some entirely new language.

5.1 `cclang.c`

Inside the file `cclang.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. The `precedence` level determines how operators are grouped (see Table 2 on page 23). Next comes `rtrnTypeString` which explains to the Cicada compiler what type(s) of data 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. Most operators have a 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). The remaining operators lack the `inbytecode` marker, and their `translation` string is just an expression built from previously-defined operators.

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

`cmdString:` The command string defines the various operators that Cicada expects to see in the command. Typically a command string consists of a concatenation of constant operator strings, which serve as signposts for the compiler, placeholders for sub-expressions within the command, and tokens indicating miscellaneous objects like variable names and hardcoded constants. There must be some distinctive constant string in the first or second position of a command sequence, so that the compiler knows what it's looking at.

Many operators take left-hand and/or right-hand arguments, as indicated by keywords like `type3arg` to the left and/or right of the respective operator string. For example, the full command string of the define operator is

```
type3arg "::" type7arg
```

The define command 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 expect different objects and therefore require a different 'type' of expression. These types are unrelated to variable/member types. Looking at the comment before the `cicadaLanguage[]` array, we see that a type 3 argument represents a variable or function, and a type 7 argument represents a code-containing expression, as one would expect. The type specifications allow the compiler to throw type-mismatch errors when expressions don't make sense.

In some cases two different commands will use the same operator string. For example, compare '`*`' is used as a multiplication operator as well as the void operator, and '`-`' is 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 an unattached expression just to the left, and a negation otherwise.

More complex definitions like `while ... do` can involve several operators.

```
"while" type6arg "do" type1arg
```

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

```
"if" type6arg "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`. 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 `cclang.c` predefines a keyword for each precedence level and uses those keywords instead of numbers.

The `cicadaLanguageAssociativity[]` array in `cclang.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 only rewrite `A`.

The size of the `cicadaLanguageAssociativity[]` array determines the allowed precedence levels. So by adding an entry to that table we would bump up the maximum allowable operator precedence level to 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 `"567"`, so its return value can be construed as being of type 5, 6, or 7. The return types correspond to the argument types from the `cmdString` field (not precedence levels), so the expression `A = 2+5` is legal because the assignment operator expects a type-6 right-hand argument, but `2 + 5 = A` is *illegal* because the left-hand argument of the assignment operator should be of type 3.

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 `cclang.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 those operators last.

The translation strings of bytecode-written 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. `a1` and `a2` will each be replaced by potentially-long bytecode expressions (think `f().a = 5+cos(b)`). In `cclang.c` the macro `bcArg(x)` produces the `a1` and `a2` strings, so the `translation` string reads

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

where `bc_define(equFlags)` produces a define operator `"8"` with equate flags: `"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 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 `cclang.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`, 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 define hidden member IDs that won't conflict with the positive IDs of user-defined members.

Scripted operators – those without `inbytecode` keywords – work basically the same as bytecoded operators except that the arguments have to be encoded with the special keywords `arg1` through `arg9`. For example, the `remove` function is defined at the beginning of `cclang.c` in bytecode, and the `[-< ... >]` syntax for removal cites `remove` in its scripted translation:

```
"remove " arg1 "[<" arg2 "," arg3 ">]"
```

So the command `arr[-<2,3>]` is first translated into `remove arr[<2,3>]`, then 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 separates two sentence-level commands – so in terms of bytecode it works a lot like a comma or end-of-line.

5.2 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 abstruse numbers are the bytes of 3.14 split across two 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 four types of inlined constants—Booleans, characters, integers, and floating-point numbers—along with strings has its own 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 bytecode 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 loops considerably.

5.2.1 `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

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

(where `<<` denotes a bit-shift operator). Table 3 on page 63 summarizes the flags words for each Cicada operator. Clearly there are many possible operators that are not being used in Cicada.

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 either the source *variable* (by default), or if the source variable is void then to the type of 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 and allocate storage for 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 the 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 member or source variable, whichever is more restrictive. Since a member can only point to an as-or-more-restrictively typed variable, this means that the new type will be that of the source variable unless there is none (the source member points into the void), in which case it's the type specification of the source member.

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 left-hand argument 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* of the source member.

Flag 5 – run-constructor: causes the constructor of the destination variable to run after the variable has been created/had its type updated, but before any data has been copied from the source variable if the equate flag was set. 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.

5.2.2 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

```
[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 `cclang.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 5, 6, or 7 based on its return-types string in the `cicadaLanguage[]` array. But then Cicada notices an adapter that works on type-6 objects (named `type6arg_adapter` in `cclang.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 bc_define(deqxFlags) bc(search_member) anonymousmember bcArg(1)
```

The expression to adapt is considered the argument and goes in place of `bcArg(1)`. It's clear that this adapter turns our expression into something looking like

```
var1 := 2
```

except that the define-equate operator has slightly different define flags (`deqxFlags` instead of `deqFlags`).

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 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 hand-write bytecode that uses negative IDs to access anonymous members, but that's a heroic measure that's probably only useful for anonymous members that are also '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). A typical workspace has a sprinkle of hidden members, notably around function calls where they briefly shine as `args` while the function is running, then live almost invisibly until the function is rerun. (Unless the function was called using braces, in which case the `args` variable is not hidden). 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 an `args` variable *without running its constructor* so that `trap()` can do so in a controlled way.

The other define-operator flag used by (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>] }`, and 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 future array resizes like `myArray[+3]` are allowed because `anon1` will de-alias rather than cause 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 `cclang.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.

5.3 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()
```

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 `cclang.c` since doing so would break Cicada's built-in scripts. Instead we must use the `newCompiler()` 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 `defs.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 := newCompiler(newLanguage, newLanguageAssociativity)

myBytecode := compile(input("Give me math:  "); compilerID = newCompilerID)
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 `newCompiler()` are almost exactly the same as the two arrays that specify the language in `cclang.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 the scripting environment. Also, make sure to define both arguments as arrays – `newCompiler()` 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.

Functions produced by different compilers live in different namespaces, because each compiler 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, manually isolate the new bytecode's search path from the workspace using `transform()`.

6 Reference

6.1 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/list 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

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

`remove A` or `[-< A, B >]` : remove member 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 < A, B > ...` : 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

`[ N ]` or `[ ]` : array

`[[ ]]` : list

`{ ... }` : 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

6.2 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 numbers of bytecode words.

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 [C function], *function_ID*, *args_variable* : runs either a built-in or user-defined C function with the given ID and arguments, and returns an integer

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 [list], *type* : an empty list of *type*

48 [bool] : Boolean data type

49 [char] : character 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 [int] : integer data type

51 [double] : floating-point 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*, which occupies `sizeof(ccFloat) / sizeof(ccInt)` bytecode words

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

6.2.1 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.

6.3 Predefined functions and variables

Cicada comes prepackaged with a number of C-coded and scripted functions. The C-coded functions are come from the `ciclib.c` source file, and they are defined in exactly the same way as user-defined C functions/wrappers. The scripted functions come from a `defs.cicada` script (which is distributed in stringified form as `defs.c`), a number of which are just wrappers for the C-coded functions. `defs.cicada` also predefines some variables and constants. Finally, when Cicada is run interactively, the terminal itself defines a few variables in the user's workspace.

name	*f*	$	>	name	*f*	$	name	*f*	$
newCompiler	x	x		filePaths			add	x	x
compile	x	x		cd	x		subtract	x	x
transform	x	x+		pwd	x		multiply	x	x
disassemble	x			run	x		divide	x	x
trap	x	x		load/Load	x	x	pow	x	x
throw	x	x		save/Save	x	x	min	x	x
top	x	x		input	x	x	max	x	x
size	x	x		print	x	x	sum	x	x
type	x	x		printl	x		mean	x	
bytecode	x	x		sprint	x		abs	x	x
member_ID	x	x		mprint	x		floor	x	x
allNames			✓	read_string	x	x	ceil	x	x
do_in	x			print_string	x	x	round	x	x
compile_and_do_in	x			cat	x		exp	x	x
go	x			readTable	x		log	x	x
jump	x			readFile	x		cos	x	x
go_path			✓	readInput	x		sin	x	x
where				writeTable	x		tan	x	x
what	x			saveTable	x		acos	x	x
calculator	x		✓	find	x	x	asin	x	x
ans			✓	lowercase	x		atan	x	x
springCleaning	x	x		uppercase	x		random	x	x
				C_string	x		sort	x	x+
					x		binsearch	x	

Table 4: Built-in functions and variables. An 'x' in the '*f*' column indicates that it's an executable function. An 'x' in the '$' column indicates that there is a corresponding C function. A checkmark in the > column indicates that the definition only exists when running from the terminal (all of these are on the left side of the table)

Table 4 lists the predefined C functions, scripted functions, and other scripted variables. Their descriptions follow in alphabetical order.

Basic constants:

- `e`: the exponential constant. Calculate exponentials by writing `e^x`, or use the vectorized `exp()` function
- `pi`: the famous constant pi
- `inf`: infinity
- `nan`: not-a-number (used by floating-point arithmetic)
- `passed`: 0, denoting the error code of a function that did not cause any error
- `root`: an alias to the user's workspace

Predefined functions and variables

`abs()`, `$abs()`

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

C syntax: `$abs`((double) *x*, (double) *y*)

Returns the absolute value of its argument, which must be a numeric scalars or arrays.

`acos(), $acos()`

syntax: (numeric) *y* = `acos`((numeric) *x*)
C syntax: `$acos`((double) *x*, (double) *y*)

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]. The arguments may be numeric scalars or arrays.

`add(), $add()`

syntax: (numeric) *z* = `add`((numeric) *x*, (numeric) *y*)
C syntax: `$add`((doubles) *x*, (doubles) *y*, (doubles) *z*)

Computes $z = x + y$, for scalar or vector numeric data.

`allNames:` the list of member names that have been defined so far, including those defined in `terminal.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.

`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
```

`asin(), $asin()`

syntax: (numeric) *y* = `asin`((numeric) *x*)
C syntax: `$asin`((double) *x*, (double) *y*)

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 $[-\pi/2, \pi/2]$. The arguments may be numeric scalars or arrays.

`atan(), $atan()`

syntax: (numeric) *y* = `atan`((numeric) *x*)

C syntax: `$atan`((double) *x*, (double) *y*)

Returns the inverse tangent of the argument, which must be numeric scalars or arrays. The result is an angle in radians on the interval $[-\pi/2, \pi/2]$.

`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()`

syntax: (string) *codeString* = `bytecode`((function) *myFunction* [, (numeric) *memberIndex*])
C syntax: `$bytecode`((function) *myFunction*, (int) *memberNumber*, (string) *codeString*)

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 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`.

`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.

`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, the terminal 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 `defs.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
```

`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)`.

`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.

`ceil()`, `$ceil()`

syntax: (numeric) y = `ceil`((numeric) x)
C syntax: `$ceil`((double) x, (double) y)

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. Both arguments may also be arrays

`compile()`

syntax: (string) *script_bytecode* = `compile`((string) *script* [, (string) *char_positions* [, member_names]] [, `code`, `compilerID` = (int), `filename` = (string)])
C syntax: `$compile`((int) *compilerID*, (string) *script*, (string) *filename*, (string) *script_bytecode*, (string) *char_positions*, (int) *num_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()` returns a second string (`script_bytecode`) containing Cicada bytecode. 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 from a given script. Optionally, the compiler ID can be set to the return value from a `newCompiler()` call. 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. The scripted function calls `$getMemberNames()` to maintain the `allNames` list.

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 `filename` argument causes any error message to reference that file name.

```
> trap(; compile("x = "; filename = "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 a readable script into a string containing binary bytecode.

`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)
```

`cos()`, `$cos()`

syntax: (numeric) y = `cos`((numeric) x)
C syntax: `$cos`((double) x, (double) y)

Returns the cosine of its argument. The argument must be numeric scalars or arrays..

`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.

`divide()`, `$divide()`

syntax: (numeric) z = `divide`((numeric) x, (numeric) y)
C syntax: `$divide`((doubles) x, (doubles) y, (doubles) z)

Computes $z = x/y$, for scalar or vector numeric data.

`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
```

)

`exp()`, `$exp()`

syntax: (numeric) y = `exp`((numeric) x)
C syntax: `$exp`((double) x, (double) y)

Returns the exponential (base e) of its argument. The arguments must be numeric scalars or arrays.

`filePaths[]:` a string array of pathnames to folders. `Load()`, `Save()`, and `run()` will search each of these paths when looking for a file. The terminal preloads an empty path, which usually implies the Cicada directory. We can change the search paths just by manipulating this set: e.g. `filePaths[+2] = "/Desktop/"`.

`find()`

syntax: (numeric) *result* = `find`((strings) *search_in*, *search_for* [`, code, mode` = -1/0/1 [`, code, startPosition` = (numeric)]])
C syntax: `$find`((string) *search_in*, (string) *search_for*, (int) *mode*, (int) *starting_position*, (int) *result*)

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()`, `$floor()`

syntax: (numeric) y = `floor`((numeric) x)
C syntax: `$floor`((double) x, (double) y)

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. The arguments may be numeric scalars or arrays.

`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 the terminal. 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 the terminal will eventually 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.

`go_path[]:` a set of aliases to each composite object in the search path, beginning with `root` and ending with the current working variable. The terminal 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 the terminal will reset `go_path`

if it detects a problem. The user can add a small coding section to `go_path` which the terminal 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.

`input()`

syntax: (string) *str* = `input`(*args_to_print*)
C syntax: `$input`((string) *str*, *args_to_print*)

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.

`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.

`load()`

syntax: (string) *file_string* = `Load`((string or int) *file_name*)
syntax: (string) *file_string* = `load`((string or int) *file_name*)

C syntax: `$load`((string or int) *file_name*, (string) *file_string*)

Reads a file into a string. If there is an error in opening or reading the file (i.e. if the file was not found), then `load()` returns "I/O error", signifying that the error comes from the operating system, not Cicada. The counterpart to `load()` is `save()`.

Little-L `load()` only looks for files in the default directory. Big-L `Load()` extends this function by searching all paths specified in the `filePaths[]` array.

The filename may be an integer (1-3) rather than a string, in order to load one of the predefined scripts. The scripts are: 1) `defs.cicada`; 2) `terminal.cicada`; 3) the user's script passed to `runCicada()` (if given, otherwise an error is thrown).

`log()`, `$log()`

syntax: (numeric) y = `log`((numeric) x)
C syntax: `$log`((double) x, (double) y)

Returns the natural logarithm (base e) of its argument. The argument must be numeric scalars or arrays.

`lowercase()`

syntax: (string) *lowercase_string* = `lowercase`((string) *my_string*)

Converts a mixed-case string to lowercase.

`max()`, `$minmax()`

syntax: (numeric) *result* = `max`((numeric list) *the_list* [, `code`, `rtrn` = { `index` / `value` / `both`])
C syntax: `$minmax`((doubles) *the_list*, 1, (int) *index*, (double) *value*])

Returns the maximum element of a list: its index, value (the default), or both { index, value }.

`mean()`

syntax: (numeric) *result* = `mean`((numeric list) *the_list*)

C syntax: `$mean`((doubles) *the_list*, (double) *result*)

Returns the average (arithmetic mean) of the elements of a numeric list.

`member_ID()`

syntax: (numeric) *ID* = `member_ID`((composite variable) *var*, (numeric) *member_number*)

syntax: `$member_ID`((composite variable) *var*, (int) *member_number*, (int) *ID*)

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.

`min()`, `$minmax()`

syntax: (numeric) *result* = `min`((numeric list) *the_list* [, `code`, `rtrn` = { `index` / `value` / `both`])
C syntax: `$minmax`((doubles) *the_list*, -1, (int) *index*, (double) *value*])

Returns the minimum element of a list: its index, value (the default), or the combination { index, value}.

`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.

`multiply()`, `$multiply()`

syntax: (numeric) z = `multiply`((numeric) x, (numeric) y)
C syntax: `$multiply`((doubles) x, (doubles) y, (doubles) z)

Computes $z = x * y$, for scalar or vector numeric data.

`newCompiler()`

syntax: (numeric) *compilerID* = `newCompiler`((`compiledCommandType` array) operatorDefs, (int array) opLevelDirections)
C syntax: `$newCompiler`((`compiledCommandType` array) operatorDefs, (int array) opLevelDirections, (int)

compilerID)

Produces a new compiler from a language specification, and returns the new compiler's ID number. (The default Cicada compiler has ID number 1). The two arguments are: 1) an array of { `string, int, string, string` }, one element for each command, containing the command definitions; and 2) an array giving the direction of evaluation for each order-of-operations level. These mirror the `cicadaLanguage[]` and `cicadaLanguageAssociativity[]` arrays, respectively, which are defined in `cclang.c`.

The beginning of `defs.cicada` has lots of definitions to make a language specification simpler and more readable.

`pow(), $pow()`

syntax: (numeric) z = `pow`((numeric) x, (numeric) y)
C syntax: `$pow`((doubles) x, (doubles) y, (doubles) z)

Computes the power function $z = x^y$, for scalar or vector numeric data.

`print()`

syntax: `print`((vars) v1, v2, ...)
C 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, `defs.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: (string) *result* = `print_string`((vars) v1, v2, ... [`, code, maxFloatingDigits` = (numeric)])
syntax: `$print_string`((string) result, (int) max_floating_digits, (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 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`.

`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.

`pwd()`

syntax: `pwd`()

Prints all file directories (all entries in the `filePaths[]` array) to the screen.

`random()`

syntax: (numeric) y = `random`()
C syntax: `random`((doubles) y)

Returns pseudo-random numbers 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. The scripted function returns a scalar; for vectorized random data run the C function.

`read_string()`

syntax: `read_string`((string) to_write, (vars) v1, v2, ...)
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()`.

`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.

`readInput()`

syntax: `readInput`((table) *table_array* [; (bools) `ifHeader`, `resizeColumns`, `resizeRows` = values])

Identical to `readTable()`, except reads the table string from the command line input.

`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.

`round()`

syntax: (numeric) *rounded_integer* = `round`((numeric) *real_number*)
C syntax: `$round`((double) *x*, (double) *y*)

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. Arguments may be scalars or arrays.

`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).

`save()`

syntax: `Save`((string) *filename*, (string) *filedata*)
syntax: `save`((string) *file_name*, (string) *filedata*)
C syntax: `$save`((string) *filename*, (string) *filedata*)

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.

`Save()` (capital 'S') extends the `save()` function by searching all paths in the `DirectoryNames[]` array. This is useful when `filename` involves a path that may only be found in another directory.

If our data isn't already in string format, it's easy to do an on-line conversion:

```
save("my_data", (temp_str :: string) =! the_data)
```

`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 the general-purpose `Save()` function. The optional arguments are the same as

those used by the function `mprint()`.

`sin()`, `$sin()`

syntax: (numeric) *y* = `sin`((numeric) *x*)
C syntax: `$sin`((double) *x*, (double) *y*)

Returns the sine of its argument, which must be a numeric scalar or array.

`size()`

syntax: (numeric) *var_size* = `size`((var) *my_var* [, code, `storageSize` = (bool)])
C syntax: `$size`((var) *my_var*, (bool) *storageSize*, (int) *var_size*)

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 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`.

`sort()`

syntax: `sort`((table) *table_to_sort*, { (list) *sort_by_list* or (numeric) *sorting_index* } [, `code`, `direction` = { `increasing` / `decreasing`])
C-1 syntax: `$makeLinkList`((doubles) *list_to_sort*, (ints) *link_list*, (int) direction, (int) *first_index*, (ints or doubles) *sorted_list*)
C-2 syntax: `$sort`((ints) *link_list*, (int) first_index, (ints or doubles) *lists_to_sort*, (ints or doubles) *sorted_lists*)

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).

The Cicada `sort()` function first calls `$makeLinkList()`, then attempts `$sort()` using the link list from the first step. The C-coded `$sort()` only works if each list to sort is numeric; if that's not true then `$sort()` uses a slower scripted sort function that works on more general data types. `$sort()` can sort multiple lists per function call: i.e. it accepts $2N + 2$ arguments.

`springCleaning()`

syntax: `springCleaning`()
C 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.

`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()`).

`subtract()`, `$subtract()`

syntax: (numeric) z = `subtract`((numeric) x, (numeric) y)
C syntax: `$subtract`((doubles) x, (doubles) y, (doubles) z)

Computes $z = x - y$, for scalar or vector numeric data.

`sum()`, `$sum()`

syntax: (numeric) *result* = `sum`((numeric list) *the_list*)
C syntax: (numeric) `$sum`((doubles) *the_list*, (double) *result*)

Returns the sum of elements of a numeric list.

`tan()`, `$tan()`

syntax: (numeric) y = `tan`((numeric) x)
C syntax: `$tan`((double) x, (double) y)

Returns the tangent of its numeric argument (scalar or array).

`throw()`

syntax: `throw`((numeric) error_code [, (composite) error_script, (numeric) code_number, error_index] [, code, if_warning = (bool)])
C syntax: `$throw`((int) error_code, (bool) if_warning, (composite) error_script, (int) code_number, (int) error_index)

Causes an error to occur. This 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 87. 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.

Although all real errors have error codes in the range 1-50, `throw()` is happy to cause an error with any integer error code. If the error code is zero then it will seem that `throw()` is not working, 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, `throw(2)` also gives an out-of-range-error.. because that's what error code 2 represents!

`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]`.

For technical reasons `top()` is defined inside of `cclang.c` rather than in `defs.cicada`. Don't try to use the C-coded `$top()`, it just confuses the compiler.

`transform()`

syntax: [(composite) *target_function* =] `transform`((string) *bytecode* [, (function) *target_function*] [`, code, codePath` = @(set of functions), `errInfo.filename/sourceCode/opCharPositions` = (string)])
C syntax: `$transform`((string) *bytecode*, (function) *target_function*, (set of functions) *codePath*, (string) *errInfoFilename*, (string) *errInfoSourceCode*, (string) *errInfoOpCharPositions*)

Copies compiled bytecode stored as a string (1st argument) into the internal code of a target function variable (return value 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 87. 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 some egregious errors are caught at compile-time 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. Two opening semicolons causes `trap()` to re-throw the error (without printing an error message), effectively redirecting the source of the error to the `trap()` command. Three opening semicolons causes it to both print any error message and re-throw the error as a thrown-to error – so code execution will then fall back to the next enclosing `trap()` and 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 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 is actually defined in `cclang.c`, and 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. The C-coded `$trap()` lacks this ability and has the usual run-constructor arguments flag, so just use the scripted function call.

`type()`

syntax: (numeric) *theType* = `type`((variable) *var* [, (numeric) *memberIndex*])
C syntax: `$type`((variable) *var*, (int) *memberNumber*, (int) *theType*)

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 index is the optional

second argument. The types IDs are listed in Table 1 on page 15. 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.

`uppercase()`

syntax: (string) *uppercase_string* = `uppercase`((string) *my_string*)

Converts a mixed-case string to uppercase.

`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!

`where:` the current search path of the user, stored as a string.

`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()`.

6.4 C functions for working with whole arguments

`getArgTop()`

syntax: (ccInt) *member_top* = `getArgTop`((arg *) *theArg*)

Returns the 'top' of the argument. This is either the array dimension (if it is an array variable), the number of list elements spanned (for a list variable), or the number of members (if it's a composite variable).

`setMemberTop()`

syntax: (ccInt) *err_code* = `setMemberTop`((arg *) *theArg*, (ccInt) memberNumber, (ccInt) newTop, (char **) argData)

Sets the 'top' of one *member of* the argument. An array variable has one member whose top is one array dimension; a list variable has one member for each list it contains; and a composite variable typically each member has a name. If the member points to primitive data, returns a pointer to that data in `argData`.

`setStringSize()`

syntax: (ccInt) *err_code* = `setStringSize`((arg *) *theArg*, (ccInt) stringNumber, (ccInt) newStringSize, (char **) stringChars)

`stepArg()`

syntax: (arg *) *sub_arg* = `stepArg`((arg *) *theArg*, (ccInt) memberNumber, (ccInt *) numIndices)

Steps into a member of an argument, returning another `arg`-type pointer. If `numIndices` is non-null, its referent is multiplied by the top of that member. Typically one would set `numIndices` to `args.indices[n]` and then update this variable with each `stepArg()` call.

`argData()`

syntax: (void *) *arg_data* = `argData`((arg *) *theArg*)

Returns a pointer to the data list of a primitive (`bool`/`char`/`int`/`double`) argument.

6.5 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.

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>]
```

is legal, whereas

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	thrown-to error
18	illegal command	44	nonexistent 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

```
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>]
```

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 '`[]`', 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 `[]` 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 C function (#44)

A C function was invoked that does not exist. Each C function has function ID: positive IDs for the user's C functions, and negative IDs for the in-built functions. The ID was not on the interval [-numInbuiltCfs, -1] or [1, numUserCfs].

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 `newCompiler()`: 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. 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. Currently unused.

Target was deleted (#39)

A member was removed while it was being aliased. One has to try hard to get this error.

Thrown-to error (#43)

This causes a message "Error was thrown in ...". It is an error produced by a `trap(;; ...)` call where the two (or three) semicolons re-throw some error in its arguments as a 'thrown-to' error. This enables tracing back of errors through nested functions.

[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