CCL
The **C**omputation and **C**ontrol **L**anguage

User's Manual

Eric Klavins

Electrical Engineering Department
University of Washington
Seattle, WA 98195
klavins at washington.edu

1 Introduction

This manual describes the CCL language and associated tools. To get started with CCL, make sure that you have CCL built and installed and that the `CCL_ROOT` and `LD_LIBRARY_PATH` environment variables are set correctly (see the CCL web page for details).

Here is how to say "hello world" in CCL. First make your own directory for your examples:

```
mkdir code
cd code
```

In your new directory, create a file called `hello.ccl` with the following text in it:

```
include standard.ccl

program main() := {
  str := "Hello world!";
  true : {
    print ( str, "\n" ),
    exit()
  };
};
```

Run this program by executing

```
ccli hello.ccl
```

on the command line. Note that `ccli` is the name of the CCL **i**nterpreter. The result of executing the above should be that "Hello world!" is printed and then the program exits. If you get this to work, you have CCL properly installed on your system.

Here's how `hello.ccl` works. The first line includes some standard function definitions. In particular, we need the `print` function and the `exit` function. The rest of the file declares a program, called `main`. The CCL interpreter expects that a program called `main` will be defined. If it finds one, it will execute it. If it does not, it will just quit. Within this particular `main` program, there is one variable initializer (for the variable `str`) and one *guarded command*. The initializer just sets the variable `str` to the string "Hello world!". The guarded command has a *guard* (`true` in this case) and a

set of commands. The commands print the value of `str` and then exit. The CCL interpreter starts executes programs by first initializing variables and then executing the guarded commands over and over again. This guarded command exits the first time it is executed. Otherwise, it would print "Hello world!" repeatedly until the interpreter is killed (with `Cntl-C`) (try removing the `exit` line to see this behavior).

Note: Another way to evaulate expressions with `ccli` is to put them on a line by themselves outside of a program definition. Thus, the above could have been written:

```
include standard.ccl

str := "Hello world!";
print ( str, "\n" ),
```

and essentially the same output would be produced.

2 Values

CCL supports the following types

unit	boolean	integer
real (floating point)	string	list
record	abstract functions (lambda)	external functions

Lists are homogeneous: All elements of a given list must have the same type. Here is an example program that demonstrates the different types:

```
// booleans
true;
false;

// integers
1;
2;

// reals
1.0;
3.1415927E-2;
```

```
// strings
"abcdefg\n";

// lists
{ 1, 2, 3, 4 };
{ "a", "b", "c" };
{};

// records
[ x := 5, y := {}, z := "ccl is cool!" ];

// the identity function
lambda x . x;

// you can also write the above as
\ x . x;
```

More interesting expressions are covered in the next section. External functions are covered later as well (`print` and `exit` are examples). Remember that expressions appearing outside a program, terminated by a semicolon, are evaluated as they are encountered by `ccli` and `ccli` prints each expression and its valuation. Thus, the output of the above program will look a bit repetitive. Also note that lines beginning with "//" are treated as comments. Comments may also appear between "/*" and "*/" as in C.

The "unit" type is reserved for external functions that do not return values. Don't worry about it for now.

3 Expressions

Values are *atomic* expressions: not made up of other shorter expressions. In this section we describe all the kinds of expressions you can build up from shorter expressions in CCL.

3.1 Boolean Operations

Here is how you write and, not and or:

```
// simple boolean expressions
```

```
true & false;
! true;
true | false;
```

You can also compare integers and reals with <, >, <= and >= as in

```
( 1.0 < 2 ) | ( 3.0 >= -7 & 3.0 > 5.0 ) | false;
```

liberally mixing integers and reals and putting in parentheses.

Finally, you can test for equality and inequality of booleans, integers, reals, strings and lists as in

```
1 = 2 | "a" != "b" | {1,2,3} = {4,5};
```

Test the equality of other types in not supported.

Warning: In `ccli` "=" means "equals" and testes for equality of its arguments. It does not perform variable assignment, as in some other languages. Assignments are done with the ":=" operator.

3.2 Type Errors

The `ccli` interpreter will not let you enter expressions that use types incorrectly. For example, you can not compare integers and strings as in the following:

```
1 = "one";
```

produces the error

```
Type error in 'example.ccl' on line 1:
  could not compare (= or !=) arguments
  1 has type integer, while
  "one" has type string
```

with information about what caused `ccli` to complain.

3.3 Arithmetic Operations

Integers and reals can be added, subtracted, multiplied, divided, mod-ed, and raised to each other. If the arguments to the operation are both integers, the result will be too. Otherwise, `ccli` with give you a real. Standard precedence rules apply. You may use parentheses wherever you need to. Here are some examples

```
// integer artihmetic
( 1 + 1 ) ^ 5 + 7 % 3;

// mixed
( 2.0 ^ 3 ) * ( 5.0 - 1.23E-3 ) - 1;
```

that evaluate to 33 and 38.990160 respectively.

3.4 String Operations

Besides comparing strings with `=` and `!=` you can concatenate them with the `<>` operator. The following yields the value `true`, for example

```
( "abc" <> "def" ) = "abcdef";
```

3.5 List Operations

Besides comparing lists with `=` and `!=`, you can add elements to lists and concatenate them. Adding elements is done with the `@` operator, and concatenating elements is done with the # operator.

For example, to insert the value `1` into the beginning of the list {`2,3`} you do

```
1 @ { 2, 3 };
```

which yields the list {`1,2,3`}. To concatenate two lists, you write

```
{ 0, 1 } # { 2, 3 };
```

which yields {`0,1,2,3`}.

Note that you might find `ccli` printing out strange things like `0@(1@(2@{ }))`. No worries, this is just how `ccli` stores the expression {`0,1,2`}. The two expressions evaluate to the same thing.

3.6 Record Operations

If you want to get the value of certain field in a record, you use the "." operator. For example,

```
[ x:= 0, y := 1 ].x;
```

evaluates to `0`.

You may construct new records from old using the `<<` operator. For example,

```
[ x:= 0, y := 1 ] << [ x := -1 ]
```

evaluates to `[ x := -1, y := 1 ]`. More generally, the expression `r << s` has all fields appearing in either `r` or `s`. The value associated with the a particular field `x` in `r << s` is `s.x` if `x` exists in `s`, and otherwise it is `r.x`. If `r` and `s` have a field name in common but do not assign that field a value of the same type, a type error is produced.

3.7 Lambda Expressions, Applications and Currying

A lambda expression is a way to write a function without really giving it a name. For example, here is a function that takes a number and negates it:

```
lambda x . -x;
```

and here is another that takes two elements are compares them:

```
lambda x . lambda y . x = y;
```

You can apply these functions to arguments by juxtaposing them as in:

```
( lambda x . -x ) 10;
```

which results in the value -10.

Note that `lambda x . lambda y . x = y` is a function that takes an argument `x` and returns another lambda expression. Thus

```
( lambda x . lambda y . x = y ) 10;
```

returns another lambda expression as a value, namely

```
lambda y . 10 = y;
```

You can apply the whole expression to two numbers as in:

```
( lambda x . lambda y . x = y ) 10 11;
```

which evaluates to `false`.

The function `lambda x . lambda y . x = y` is *polymorphic*, meaning that it can be applied to any two arguments of the same type. In fact, `ccli` considers its type to be

```
'a -> 'a -> boolean
```

where `'a` is a variable type (i.e. it stands for any type). If you apply the function to a string as in

```
( lambda x . lambda y . x = y ) "ccl";
```

you get a function of type

```
string -> bool
```

That is, putting a string as the first argument adds the constraint to the `ccli` type checker that the second argument should also be a string. Thus,

```
( lambda x . lambda y . x = y ) "ccl" 5;
```

produces the type error

```
Type error in 'hello.ccl' on line 23:
  could not apply function to argument
  ( ( lambda x . ( lambda y . (x=y) ) ) "ccl" )
    has type string -> boolean, while
  5 has type integer
```

noting the problem. You also have to be careful about parentheses. For example, the following produces an error

```
( lambda x . lambda y . x + y ) 10 - 11;
```

because it parses as

```
( ( lambda x . lambda y . x + y ) 10 ) - 11;
```

which makes no sense (you can't subtract an integer from a function). Thus, use

```
( lambda x . lambda y . x + y ) ( 10 - 11 ); // or
( lambda x . lambda y . x + y ) 10  ( - 11 );
```

depending on your intentions.

3.8 Conditionals

You can write if-then-else statements that evaluate to values as in

```
if 1 > 0 then 1 else 0 end;
```

which evaluates to 1. The "then" and "else" parts of the statement must have the same types and the "if" part must be a boolean expression. Thus, the following produce type errors:

```
if 1 then 1 else 0 end;
if 1 < 0 then 1 else "oops" end;
```

It is important to remember that, `if`-statements evaluate to values (kind of like the `a ? b : c` syntax in C). So for example, the following is perfectly legal:

```
"ccl is " <> ( if true
                 then "strange but beautiful"
                 else "just strange"
               end );
```

which you can probably evaluate in your head.

3.9 Let Expressions

With a let-expression you can declare a local variable. For example, the expression

```
let x := 10, y := x/2 in
  x + y
end;
```

evaluates to 15. Once again, all expressions evaluate to a value, in this case an integer. Thus you can write, for example,

```
1 + ( let x := 10, y := x/2 in
        x + y
      end );
```

which evaluates to 16.

4 Variables and Recursive Functions

In `ccli`, you can use variables in place of values. Assignments are done with the `:=` operator. For example, we could write

```
f := lambda x . lambda y . ( x + y ) / 2;
x := 10.0;
f x ( x + 1 );
```

which results in the value 10.5;

Special Variables Two variables are predefined by `ccli`. The first is `ARGC`, an integer defined to be the number of arguments on the command line to `ccli` (excluding options to `ccli`). The second, `ARGV`, is a list of the arguments on the command line, interpreted as strings. For example, if you call `ccli` like this:

```
ccli file.ccl 10 unix
```

then you get

```
ARGC = 4;
ARGV = { "ccli" "file.ccl" "10" "unix" };
```

Note: To convert the string `"10"` into the integer 10, use the `standard.ccl` function `atoi` as in

```
include standard.ccl
n := atoi ( ARGV[2] );
```

Recursion Having variables allows us to write recursive functions. `ccli` provides the keyword `fun` for declaring recursive functions. Here's the hoary chestnut

```
fun fact n .
  if n <= 0
    then 1
    else n * fact (n-1)
  end;

fact 5;
```

that you see in every programming language manual. The code above sets the variable `fact` to a representation of the factorial function and then applies it to the value 5 to give 120.

Note, you can't do

```
fact := lambda n .
  if n <= 0
    then 1
    else n * fact (n-1)
  end;
```

because `ccli` will complain that `fact` is an unbound variable (which it is in this case: the right hand side of an assignment is always evaluated before the left hand side unless you use the `fun` keyword).

5 External Functions and Parentheses

`ccli` provides some externally defined functions – and you can define your own (as described later). These functions are not lambda expressions and thus they use a different syntax. For example, the file `$CCL_ROOT/lib/math.ccl` defines some standard math functions, like cosine and sine. To use them, you write, for example,

```
include math.ccl
x := sin ( 0.123 ) + 1;
```

but not

```
include math.ccl
x := ( sin 0.123 ) + 1;
```

which will produce a type error saying something like "you can't apply the value `sin` as though it were a lambda expression". This is because `sin` is declared as an *external function type* which means that to use it, you have to put its arguments in parentheses. Another example is the `print` function, which takes any number of arguments of any types and prints them:

```
include standard.ccl

x := 5;
y := { 1, 2 };
print ( "x = ", x, " and y = ", y, "\n" );
```

which does about what you would expect. Note that every expression in CCL evaluates to a value of some type. Functions like `print` evaluate to the type `unit` which `ccli` prints as a period (as in "."). You aren't supposed to use the result of a `print` call, but it has to have a some sort of value (this is like `void` in C). The math library and other libraries are described in Section 7.

External functions are declared with return types are argument types. Types are denoted by type expressions (which you have seen in `ccli`'s type error reporting. Type expressions are defined by the following grammar:

typeexpr ::= variable | atomic | listexpr | rec | func
variable ::= `'a` | `'b` | ...
atomic ::= `unit` | `bool` | `int` | `real` | `string`
listexpr ::= typeexpr `list`
rec ::= `[` var `:=` typeexpr, ... `]`
func ::= typeexpr `->` typeexpr

The special symbol `...` may appear in record expressions and as the last expression in the list of arguments to an external function. It means there may be more arguments. For example,

```
external [ x := int, ... ] func ( 'a list, ... )
  "library.so" "func";
```

declares the external function `func`. It has a record return type that has the integer field `x` defined and may have other fields. It takes at least one argument, a list of any type, and may take more arguments (of any type).

Including Libraries We have already seen the `include` statement. It is a preprocessor directive (i.e. `include` is not a keyword, it is a directive to the lexer part of `ccli` that reads in `*.ccl` files). It tells `ccli` to read in the file noted in its argument before continuing with the preset file. `ccli` provides several libraries in `$CCL_ROOT/lib`, including standard input and output, math functions, list operations, and simple graphics tools. These are described in more detail in Section 7.

6 Programs

Programs are the heart of CCL. A program consists of two parts: A set of *initializers*, and a set of *guarded commands*. Programs are intended to be the guts of a *while statement*. As in

```
x := 1;
while true do
  if x > 0 then x := x + 1 end;
  print ( x );
end while
```

which **is not a ccli program**, by the way. In this psuedocode snippet, the line `x := 1` is an initializer and the if-statement is a guarded command. In CCL you write

```
include standard.ccl

program main() := {
  x := 1;
  x > 0 : {
    x := x + 1,
    print ( x, "\n" )
  };
};
```

which *is* a `ccli` program. If you make a file with the program above in it, and then run `ccli` on the file, you will get

```
2
3
4
5
6
...
```

ad infinitum (press cntl-C to halt `ccli`). It starts by setting `x` to 1 and then it continuously executes the guarded command over and over again.

The general syntax for a program declaration is

```
program p ( param_1, ..., param_n ) := {
  statement_1
  statement_2
  ...
  statement_m
}
```

where the statements are either assignments of values to variables (and therefore initializers) or they are guarded commands. Guarded commands have the form

```
boolexpr : {
  command_1,
  command_2,
  ...
  command_k
}
```

which means, to `ccli` anyway, that while executing the program `main`, if `boolexpr` is `true`, then execute all the commands in the body of the guarded command. In each iteration of the program, `ccli` considers the guarded commands in order and then, in those whose guards are true, executes the assignments in order as well. A command is either an assignment or an expression. In the latter case the expression is evaluated and its result is thrown out (which is useful when the expression has side effects, as with the external function `print`).

Initializer statements can also have the form:

```
needs x, y, z;
```

which tells `ccli` that the enclosing programs needs the values of x, y and z to be initialized elsewhere. This will make more sense after you read Section 6.2, about program composition.

As we have noted, `ccli` looks for a program called `main()` to execute. So if you want something to happen, you need a `main` program defined somewhere.

The parameters can be used to create a whole family of programs, one for each set of values of the parameters. For example, you might write

```
include standard.ccl

program p ( x0, delta ) := {
  x := x0;
  x > 0 : {
    x := x + delta,
    print ( x, "\n" )
  };
};

program main() := p ( 5.35, 0.01 );
```

which produces the output

```
5.36
5.37
5.38
...
```

ad infinitum. The program `main` should not have parameters.

6.1 Variable Memory

After a variable is assigned, `ccli` can remember its previous value. For a variable is `x`, the expression `'x` (read *prev x*) refers to its previous value. Note that `''x` is the same as `'x`. That is, `ccli` only remembers one step back. This is useful in programs where you might need to know the difference between the current value of a variable and its value the last time through. For example, here is one way to write the Fibonacci sequence:

```
include standard.ccl

program main() := {
  x := 1;
  true : {
    print ( { 'x, x }, "\n" ),
    x := x + 'x
  };
};
```

Executing `fib(1)` produces

```
{ 1, 1 }
{ 1, 2 }
{ 2, 3 }
{ 3, 5 }
{ 5, 8 }
{ 8, 13 }
{ 13, 21 }
{ 21, 34 }
...
```

ad infinitum.

Warning: Assignments such as `a[i] := x` or `r.a = x` are designed to be as efficient as possible and for technical reasons do not preserve the variable memory structure. Thus,

```
a := {};
a := { 1, 2, 3 };
a[0] := 0;
print ( 'a, ", ", a } );
```

results in {}, { 0, 2, 3 }. Thus, don't use the *prev* operator on array or record variables that have their parts individually assigned elsewhere.

You can also *prev* entire expressions. Thus

```
'(x+y) = 'x + 'y;
'(lamda x . x + y) z = (lambda x . x + 'y) z;
```

are both `true` under normal circumstances.

6.2 Program Composition

You can compose programs together with the `+` operator on programs or the more general `compose` operator. This has the effect of unioning programs together. You can also specify that some of the variables used in the programs you compose are to be considered the same (or shared), while the rest of the variables are local to the programs. Here is an example.

```
include standard.ccl

program plant(a,b,x0,delta) := {
  needs u;
  x := x0;
  y := x;
  true : {
    x := x + delta * ( a * x + b * u ),
    y := x,
    print ( "  x = ", x, "\n" )
  };
};

program controller ( k ) := {
  needs y;
  u := 0.0;
  true : { u := - k * y };
};
```

```
program system ( x0, a, b ) :=
    plant ( a, b, x0, 0.05 ) +
    controller ( 2 * a / b ) sharing u, y;

program main() := system ( 3.141, 5.5, 1.1 );
```

Here, a simple one-dimensional linear system is declared in *plant*, which takes some parameters. Then the program *controller* is defined. By themselves, they don't do much (the value of `x` in the plant just explodes). But combined, they do. The program `system` is composed of instances of the `plant` and `controller` programs. It takes the initial value of `x` and the parameters `a` and `b` and passes them to `plant` and passes $2a/b$ to `controller` (the gain in the control law). Most importantly, it says the the variables `u` (the control input to the plant) and `y` are *shared* – that is, assignments in either the plant part or the controller part of `system` will be to the same memory location. Meanwhile, the variable `x` remains local to `plant` and invisible to other programs.

The `needs y` declaration in the `controller` program is used to specify that the variable y "needs" to be initialized by some other program, in this case, by `plant`. A similar situation holds for the `needs u` declaration in the `plant` program. Any variable declared as "needed" should be included in a "shared" part of a composition for it to be visible to the program that initializes it.

Programs may initialize the same shared variables if they do not use the `needs` delcarator. The order of composition is important! This is because `ccli` will execute the initializers in order of appearance. Thus, if program `p` initializes `x` to 1 and program `q` initializes `x` to 2, then program `p()+q() sharing x` initializes `x` to 2.

`ccli` infers the types of all variables appearing in a program and makes sure that if a variable is shared between two programs, it has the same type in each.

Warning: Be wary of accessing `'x` for any variable that is shared between multiple programs. The variable `x` may be assigned multiple times elsewhere before execution returns to the current program, with possibly unintended results. That is, `'x` **does not** refer to the value of `x` at the previous time step. It is only the variable's previous value. As a general rule, only use the prev operator on local variables.

To compose more than two programs, you could do, for example,

```
program main() := p(1)
                + p(2) sharing x
                + p(3) sharing x
                + p(4) sharing x;
```

but you could accomplish the same thing with

```
program main() := compose i in {1,2,3,4} : p(i) sharing x;
```

which is a bit more concise. The general form for the `compose` operator is

```
program q ( param1, param_2, ... )
  := compose v in L : p ( expr ) sharing var1, var2, ...
```

The variable i ranges over the values in the list `L` and may appear free in `expr`. It is very important to note that in the present version of `ccli`, the list `L` is evaluated in the global scope and can not refer to the parameters `param1`, `param2`, ... in the parameter list of q. So you can not say program q(n) := compose i in range n : p(i); because n is not defined in the global scope (in this example anyway). The `range` function is defined in `list.ccl`. It takes an integer `n` and returns { `0, ..., n-1` }. Here's a more complete example that you could extend to do a multi-agent simulation.

```
include list.ccl
include standard.ccl

program agent ( i ) := {
  print ( "Hello from agent ", i, "\n" );
  // etc.
};

program quit() := {
  true : { exit() }
};

n := atoi ( ARGV[2] );

program main() := compose i in range n : agent(i) + quit();
```

If this code is in a file is called `agent.ccl`, then executing

```
ccli agent.ccl 5
```

results in

```
Hello from agent 0
Hello from agent 1
Hello from agent 2
Hello from agent 3
Hello from agent 4
```

7 Standard Libraries

In this section we describe most of the functions defined in the core CCL libraries. These include standard I/O functions, mathematical funtions, list operations and graphics operations. The list of functions here may not be complete. Look at the actual library files (in $CCL_ROOT/lib/) for a complete list of functions available and brief descriptions of their use. To use the functions in a library, simply include it at the beginning of your code. CCL automatically looks in $CCL_ROOT/lib/ and then in the local directory for included files.

7.1 Standard Functions (standard.ccl)

This library defines basic I/O functions and other common functions. Almost every program you write will need at least one of the functions in this library.

`external unit print (...)`
This function allows you to print to the terminal (standard out). It returns `unit`, and so should not be used in assignments. It takes any number of arguments of any type, converts them to strings and prints them, one after the other. String arguments may contain standard escape sequences, like `\n` and `\t` (newline and tab). This is similar to the UNIX `printf` function, which `print` actually uses to print strings.

`external bool input_ready ()`
This functions takes no arguments and returns `true` if and only if a key has been pressed, but not read from the keyboard. More exactly, it returns `true` if and only if the standard input buffer is nonempty.

`external string get_char ()`
This function takes no arguments and waits for the next key to be pressed. It returns a length 1 string with the ASCII representation of the key pressed. To be used in a non-blocking sense, guard the function with `input_ready`.

`external unit exit ()`
This function kills `ccli`.

`external int atoi ( string )`
This function converts its argument, which is supposed to be a string representation of an integer, into an `int`.

`external real atof ( string )`
This function converts its argument, which is supposed to be a string representation of an `real`, into a `real`.

`external string tostring ( 'a )`
This function takes an argument of any type and converts into a string.

`external int uclock (), mclock (), dclock ()`
These functions return the number of microseconds, milliseconds and seconds, respectively, since `ccli` was started.

Here is an example program that uses all three time functions.

```
include standard.ccl
program main() := {
  print ( "Press any key to print the clock values\n" );
  input_ready() : {
    get_char(),
    print ( "\rms = ", mclock(),
            ", us = ", uclock(),
            ", s = ",  dclock(), "                \r" )
  }
};
```

`external unit usleep ( int )`
Sleeps for the number of microseconds specified by its argument. Note that this may not be especially accurate in a multitasking operating system like UNIX.

7.2 Math (math.ccl)

This library defines, as external functions, all the trigonometric functions, logarithms, square root, floor, and ceiling. It also defines the symbol `pi` to be an approximation of π.

The math libary also defines a few lambda functions, as follows.

`abs x`
Evaluates to the absolute value of (the integer or real value) of `x`.

`max x y`
Returns the maximum of `x` and `y`.

`min x y`
Returns the minimum of `x` and `y`.

`sign x`
Returns 1 if `x` is positive, -1 if it is negative and 0 if it is zero.

`dot x y`
Returns the dot product of the lists `x` and `y`, which should be integer or real lists of equal length.

`mmult A B`
Returns the matrix product of `A` and `B`, which should each be lists of lists of integers of reals.

To use `dot` or `mmult`, you would write, for example,

```
include math.ccl
include list.ccl

x := { 1, 2, 3 };
dot x x;

A := {
  { 1, 2, 3 },
  { 3, 4, 5 },
  { 5, 6, 7 }
};

y := mmult A (tocol x);
```

The function `tocol` converts the row vector `x` into a column vector and is defined list.ccl (discussed later).

External vs. Lambda If you look at `$CCL_ROOT/lib/math.ccl` you will notice that `mmult` is defined as

```
external real matrix_mult ( real list list, real list list )
  "libcclmath.so" "ccl_matrix_mult";
mmult := \ A . \ B . matrix_mult ( A, B );
```

That is, the lambda abstraction `mmult` is just a wrapper around the external function `matrix_mult`. You can use either one. The designers of `ccli` could put wrappers around all external functions that take a fixed number of arguments, and maybe they will in a future release. For now, the goal is to make things fast, so computational overhead involved in computing the wrapper, although small, is significant after repeated calls.

7.3 Common List Operations (list.ccl)

The list library contains a number of functions useful for manipulating lists and lists pretending to be arrays or vectors. The functions `map`, `length`, `table` and `range` are the workhorses of `ccli`.

`rev L`
Returns a list with the same elements as `L`, but in reverse order.

`map f L`
Returns a list obtained from `L` by applying `f` to each element of `L`.

`length L`
The length of the list `L`.

`zip A B`
Returns a list of pairs {`{A[0],B[0]}, {A[1],B[1 }, ... }`.

`makelist n default`
Returns the list that is `n` elements long all of whose elements are equal to `default`.

`sumlist L`
The sum of all the elements in `L` where `L` should be a list of numbers.

`table f n m`
Returns the list {`f n, f (n+1), ..., f m`}.

`range n`
Returns the list {`0,1,...,n-1`}.

`member x L`
Returns `true` if `x` appears in `L` and `false` otherwise.

`remove x L`
Returns a list identical to `L`, except with all occurences of elements equal to `x` removed.

`cross A B`
Returns the cross product of `A` and `B` taken as multisets.

`tocol v`
Good for changing a row vector into a column vector. It is equivalent to `map (\x.{x}) v`.

7.4 Interprocess Communication (iproc.ccl)

The `ccli iproc.ccl` library provides three functions and some behind the scenes functionality that allow programs to send messages to each other (or to themselves) via "mailboxes". The library keeps a list of such mailboxes internally — in reality they are queues of messages — and associates an integer id to each one. Messages have type

```
[ to := int, from := int, ... ]
```

where the `to` field defines what mailbox the message is to be sent to or what mailbox the message was received from. The `from` field is used to indicate what "agent" the message was sent from. Typically, each program in a composition of programs is associated with an id that it uses in the `from` field when it sends messages and that it expects in the `to` field when it receives them. Additional fields, defining the message content, may be included in messages. The `iproc.ccl` functions are:

`external unit send ( [ to := int, from := int, ... ] )`
This function is used to add (enqueue) a message to the end of the mailbox (queue) associated with the integer identifier `to`.

`external [ to := int, from := int, ... ] recv ( int )`
This function is used to remove (dequeue) a message from the mailbox (queue) associated with the integer identifier `to`. If that mailbox is empty, a message with the `from` field equal to `-1` is returned. The call is non-blocking.

`external bool inbox ( int )`
This function is used to check whether the mailbox associated with its integer

argument has any messages in it. If it does, `true` is returned, otherwise `false` is returned.

Two considerations need to be taken into account when using the `iproc.ccl` library. First, be careful that your programs do not send significantly more messages to a mailbox than will be received from that mailbox. If a program keeps sending messages without receiving them, it will use up all the available memory on the system. The `iproc.ccl` helps you be careful about this by issuing annoying warnings every time a message is sent to a mailbox with more than some maximum number of messages in it (100 in the last distribution). Second, `ccli` can not type check messages sent with the library. That is, one program might send a message with a string field named `msg`. Another program may receive the message and then try to access a field named `msg` but use it with type int. Or it may try to access a field that was not defined in the message when it was sent. Both of these errors will not be caught by `ccli` at compile time and will issue runtime errors.

Here is an example to study. It defines n agents numbered 0 to $n-1$. When agent i receives a message from agent $i-1$, it sends a message to agent $i+1$. Agents 0 and $n-1$ behave a little differently. Note that the program prints out the messages in increasing order of agent id, regardless of how the commands in the `main` composition are scheduled (try running it with and without the `-r` option).

```
include iproc.ccl
include standard.ccl
include list.ccl

program agent ( i, n ) := {

  got_mesg := if i = 0 then true else false end;
  sent_mesg := false;

  got_mesg & ! sent_mesg & i < n-1 : {
    send ( [ to := i+1, from := i,
             msg := "hi from " <> tostring ( i ) ] ),
    sent_mesg := true
  }

  inbox ( i ) & i != n-1 : {
    print ( recv ( i ), "\n" ),
```

```
    got_mesg := true
  }

  inbox ( i ) & i = n-1 : {
    print ( recv ( i ), "\n" ),
    exit()
  }

};

program main() := compose i in range 10 : agent ( i, n );
```

7.5 UDP Datagrams (udp.ccl)

The `udp.ccl` library allows CCL programs to communicate with other CCL programs running as truely separate processes (started with a different call to `ccli`), possibly on other machines. The library is similar to the `iproc.ccl` library, except that explicit servers and clients are used. The library consists of five functions. The first three are used by UDP servers.

`external int udp_new_server ( int )`
The argument to this function is the UDP port the server should listen to. Port numbers in the 7000s are good for experimental programs. The return value is an opaque integer server id (sid) that is used in `udp_is_ready` and `udp_get_data`.

`external bool udp_is_ready ( int )`
The argument to this function is the sid of a server started with `udp_new_server`. It returns `true` if and only if there is data ready to receive in the server's buffer. It does not block.

`external [ timestamp := int, from := string, data := 'a ] udp_get_data ( int )`
The argument to this function is a valid sid. The return value is a record that has three fields: an integer `timestamp` field that can be used to order messages from the same client in terms of the time they were sent; a string valued `from` field containing the host name of the client that sent the data; and a `data` field whose type depends on what the client sent.

An example server that simply prints to the terminal the messages it receives is as follows.

```
include standard.ccl
include udp.ccl

program server ( port ) := {

  sid := udp_new_server ( port );

  udp_is_ready ( sid ) : {
    print ( "got: ", udp_get_data ( sid ),
            " on port ", port, "\r" )
  }

};

program main() := server ( 8000 );
```

The other two `udp.ccl` functions are for clients.

`external int udp_new_client ( string, int )`
The first argument to this function is a string containing the hostname to which to send data and the second argument is the UDP port to address. The return value is the client id (cid) that should be used in calls to `udp_send_data`.

`external unit udp_send_data ( int, 'a )`
The first argument to this function is a valid cid. The second argument is the data of the message and may of any type.

Once again, care should be used with the `data` field as `ccli` can not type check whether the client's idea of the type of the data is consistent with the server's idea of what it is. An example client that can communicate with the above server is as follows.

```
include standard.ccl
include udp.ccl

program client ( host, port, period ) := {

  cid := udp_new_client ( host, port );
  count := 0;
  time := dclock();
```

```
  dclock() >= time + period  : {
    time := dclock(),
    udp_send_data ( cid, [ str   := "this some data",
                           count := count ] ),
    count := count + 1
  }

};

hostname := if ARGC > 2 then ARGV[2] else "localhost" end;
program main() := client ( hostname, 8000, 0.25 );
```

7.6 Graphics (windows.ccl)

The library `lib/windows.ccl` provides a simple interface to the GDK/GTK X Windows toolkit (see http://www.gtk.org/). To create a window, use one of the following

```
int new_window ( string )
int new_window_custom ( string, real, real, real )
```

The return value is an opaque window identifer, needed for future references to the window – so assign a variable to it. The string argument to both functions will be the window title. In the second function, the last three arguments are the scale factor, and the width and height of the window (in pixels). Thus

```
w := new_window_custom ( "data", 10.0, 100, 100 );
```

creates a new window 100 × 100 pixels big, named "data". The scale factor of 10.0 means that if you draw a line of length 1, with will be approximately 10 pixels long. The origin of the window is the center, x coordinates go from left to right, y coordinates fo from bottom to top. To draw something in a window, use one of the following functions

```
//          w    x1    y1    x2    y2
unit line ( int, real, real, real, real )

//          w    points          filled
unit poly ( int, real list list, bool )

//            w    x     y     r     filled
```

```
unit circle ( int, real, real, real, bool )

//         w    x     y     wid   hi    a1    a2    filled
unit arc ( int, real, real, real, real, real, real, bool )

//          w    x     y     text
unit text ( int, real, real, string )
```

The first argument to each of these functions is a window identifier created with one of the new window functions. The rest of the arguments are unique to the functions and should be apparent from the annotations above.

After drawing lines or polygons, you need to refresh the window before you will see anything. Do this using the function

```
unit refresh ( int )
```

which takes a window identifier as an argument.

If you want to draw in some color other than black (the default), use

```
unit setcolor ( int, string )
```

which takes a window identifier and a string that represents the color (currently, "black", "white", "red", "blue", "green" or "yellow"). To erase a window, use

```
unit erase ( int )
```

and then `refresh()`. To process events in a window (like exposures, mouse clicks, etc), use

```
unit update_windows ()
```

You usually don't use this function, but instead compose your windows progams with the following program, which is in `lib/windows.ccl`:

```
program window_manager() := {
  true : {
    update_windows()
  };
};
```

For example, here is a program that makes a spinning red square with a black border:

```
include windows.ccl
include math.ccl

fun make_square r theta .
  let c := r * cos ( theta ),
      s := r * sin ( theta ) in
    { { c, -s }, { s, c }, { -c, s }, { -s, -c } }
  end;

program spin ( r, delta ) := {

  theta := 0.0;
  w := new_window_custom ( "spin", 10.0, 30 * r, 30 * r );

  true : {
    erase ( w ),
    setcolor ( w, "red" ),
    poly ( w, make_square r theta, true ),
    setcolor ( w, "black" ),
    poly ( w, make_square r theta, false ),
    refresh ( w ),
    theta := theta + delta
  };

};

program p ( r, delta ) := spin ( r, delta ) + window_manager();
program main := p ( 10, 0.05 );
```

Another function that might be useful, if you want to save a graphic or make an animation for example, is the function

```
//                      w    filename
external unit tofile ( int, string )
  "libcclwindow.so"
  "ccl_tofile";
```

which saves the image in the window identified by `w` into a file named `filename`. The file name should end with the graphics format you want: `.jpg`, `.png`, `.ps` or `.ppm`.

8 Defining External Functions in C++

You can define you own external functions in ccli. This is very useful for interfacing ccli with other APIs, such as the GTK, or to low level hardware code. In this section we outline how to do this. The procedure is, roughly:

1. Define a C++ function
2. Compile it into a shared object library
3. Delcare it within a ccli file

It is easiest to explain with an example. The rest of this section shows how the `cos` function is defined.

First, in standard/ccl_math.cc, we define the C++ function

```
#include <math.h>
#include "SymbolTable.hh"

extern "C" Value * ccl_cos ( list<Value *> * args ) {
  return new Value ( cos ( (*args->begin())->num_value() ) );
}
```

All external function definitions must be declared this way. The `external "C"` part tells the C++ compiler to use C naming conventions, important for making shared object code usable by other modules. The return value must be `Value *`, a pointer to a `Value` object. The `Value` class is defined in "SymbolTable.hh". The argument is a list of values (to be computed from within ccli when the function is called). We use the STL list template to define lists.

The body of the function makes a new value using the `Value ( double )` constructor and the C++ `cos` function. The argument to the `cos` function is the numerical value of the first value in the argument list.

To compile, you do something like

```
g++ -shared -o ccl_math.so ccl_math.cc -I \$(CCL_ROOT)/base
```

where `ccl_math.cc` is the name of the file containing the aboce function. Look at standard/Makefile for details.

From within a ccl file you declare the `cos` function as in

```
external real cos ( real ) "libcclmath.so" "ccl_cos";
```

This declaration states that the symbol `cos` should be considered to be an external function that takes a single real value and returns a single real value. This type information is needed by ccli because, without access to its C++ definition, ccli is not able to infer the type of the function. Furthermore, the declaration tells ccli to find the object code for the function in the file `libcclmath.so`, which should be either in the current directory or in a directory listed in the `LD_LIBRARY_PATH` environment variable. The last argument gives the name of the C++ function, in this case `ccl_cos`, within the shared object library.

For more information, see the code in the `standard` and `graphics` directories. Also, learn about shared object libraries by reading the `dlopen` man page (i.e. `man dlopen`).

9 Invoking `ccli`

The `ccli` command has the following general form.

```
ccli file.ccl <args|options>*
```

That is, the first argument to `ccli` is a file containing CCL code. The rest of the arguments are either arguments to your CCL program or, if they begin with a leading "-" (a dash or minus sign), they are options to `ccli`. The arguments to your program are available via the `ARGV` and `ARGC` variables as discussed in Section 4. The options are stripped before being put in the in `ARGV` array (and do not count toward the size of `ARGC`). The options are put as strings into a global variable called `OPTIONS` just in the off chance that your program might need to know about them. At present, `ccli` takes the following options.

`-pstring` Defines the name of the main program to be executed to be `string`. If this option is not given, then the name `main` is used, as discussed above.

`-r` Tell `ccli` to executes the clauses in the main program in a random order. Normally the clauses are executed in order of appearance in the main program. With the `-r` option, each clause is executed once before any can be executed again. A sequence of steps wherein each clause is executed once is called an EPOCH. Each epoch has a different ordering. Using this option allows you to see the effect of arbitrary orderings on

your program — in case you are interested in modeling distributed interleaved systems.

`-d` Tells `ccli` to execute the main program using a simple, single-stepping debugger. The commands are executed in order. After all variables are initialized, a command prompt is printed along with a line number and file name of the next guard or command set to be executed. At this prompt you may run the following commands:

- `s` Execute the currently printed guard or command and move to the next.
- `p var` Print the value of the variable `var`.
- `t n` Print the symbols and their values in top `n` scopes.
- `T` Print the symbols and their values in all scopes.
- `q` Quit.
- `l` Clear the screen.
- `h` Print a very simple help message.

Note: the `-r` and `-d` options cannot be used together.

10 Tips and Tricks

10.1 Side Effects

Some external function calls have side effects and return type "." (such as `print` or many of the windows functions). You might want to write a function that draws a picture involving several windows calls. One way to do this is, for example,

```
fun draw w .
  { line ( w, -1, 0, 1, 0 ),
    line ( w, 0, -1, 0, 1 ) };
```

which draws a crosshairs in the window `w`. It's return type is `unit list`, which you can just throw out. What if you want the function to return a value or the functions you want to call don't all return `unit`? Then, put the commands in a let expression, as in

```
fun draw w .
  let dummy := { line ( w, -1, 0, 1, 0 ),
                 line ( w, 0, -1, 0, 1 ) } in
    true
  end;
```

which returns `true` (and could have any expression you want for the body of the let expression). You would have multiple definitions in the let statement for functions return other types.