COAL (COmplex Arithmetic Language)

May 14, 2009
COMS W4115
Eliot Scull (CVN)
e.scull@computer.org

1 Introduction

This project document describes the specification, design, and implementation of the COAL language (COmplex Arithmetic Language). COAL is a "helper" language used to express complex arithmetic or algorithms expressively while allowing efficient interoperation with C. COAL is compiled to C which can then be compiled to native with any ANSI C compiler and linked to a C program.

COAL's motivation stems from the fact that expressing complex math in plain C code can be hard to maintain, debug, or understand. While moving to C++ or using Matlab can bring expressiveness to code using complex math, either of these options have high overhead in complexity and/or monetary cost. COAL is intended to be a lightweight computational language suitable for typical 21st century embedded systems or larger.

In terms of features, COAL has attributes of a very streamlined functional language. COAL allows the definition of functions containing expressions that return the evaluation of expressions. Iteration is achieved through recursion and built-in map/reduce operators.

2 Tutorial

To write a COAL program, you will need a text editor, the COAL compiler, and a C compiler, preferably ANSI (C99).

2.1 Functions

COAL programs are made up of named or unnamed functions. Named functions take this form:

```
name(arg1, arg2, …) -> <expression>
```

Unnamed functions take the form `(arg1, arg2… -> <exression>)` and must always be defined and invoked in the scope of a named function:

```
named(x) -> (i -> x+i)(x) # returns x + x
```

Unnamed functions acquire the scope of their parent functions and can be nested. Unnamed functions must have at least one argument while named functions can have none.

See the Cookbook section below for examples of useful expressions.

2.2 Getting Your First COAL Program To Compile

COAL programs are defined in one or more .coal source files. The COAL compiler then takes these .coal files and translates them into a .h and .c file. This .h and .c are integrated into an application that can interoperate with C where the functions defined in the .coal files can be called.

The steps below show how to get an example .coal program up and running.

Step 1) Create .coal source file(s):

square.coal:

```
# a simple function that squares its argument
# and returns the result.
square(x) -> x^2
```

Step 2) Compile .coal file(s) into target .h and .c files (it is assumed that coal is set in $PATH):

```
$ coal -o square square.coal
$ ls
square.c  square.coal  square.h
```

Step 3) Define a calling program that will use the COAL output:

run square.c:

```
#include <stdio.h>
#include "square.h" /* output from coal */

int main()
{
    /* result */
    coal_num res;

    /* get square of 9i */
    res = square(cl_num(0.0, 9.0));

    /* check for success */
    if (!cl_valid_num(res))
        return -1;

    /* print real and imaginary parts of result */
    printf("%f %f", cl_dbl_re(res), cl_dbl_im(res));

    return 0;
}
```

Step 4) Compile .c files into executable and run:

```
$ gcc -c square.c run_square.c
$ gcc -o run_square run_square.o square.o
$ ./run_square
-81.000000 0.000000
```

2.3 Cookbook

COAL is a simple language that facilitates computations involving complex numbers and arrays of complex numbers. Below are some examples of COAL expressions and functions they're defined in that can be used as is or customized as needed.

```
#evaluate a complex math expression
eval(a, b) -> (a * (b + 2.5i) / (3 - 3i )) ^ (1 + 6.1i)


#find the maximum value in an array
max_in_arr(x) ->
  (max, e-> if (e>max) then e else max){x[0], x}


#generate an array containing this series:
# 1 + 1i, 2 + 2i, … 5 + 5i
gen() -> (e -> e + e*1i){1..5}


#break up intricate expressions in a sequence:
calc_it(a, b) ->
  q <- a ^ 2 - 9/sin(17.0*PI);
  r <- b * 12 + mag(q) + phase(a);
  q + r # return sum of q and r


#double the values of a mutable array in place:
two_times(x) ->
  (arr, n-> arr[n]<-arr[n]*2.0; arr){x, 0..last(x)}
```

Please see section 3 for more details about the functionality offered by COAL. Section 3.5 covers the interoperation with C.

3 Language Reference Manual

3.1 Syntax Notation

The context-free grammar notation used throughout this document is written in *italics* and is loosely based on the format used in the C Reference Manual in the appendix of the "C Programming Language", by Kernighan and Ritchie.

Throughout the sections in this document non-terminals of this grammar notation are suffixed with a number (i.e. *expresion1*, *expresion2*,…) This number has no significance for the grammar and is only used to describe the subordinate parts of a production.

3.2 Lexical Conventions

A program is defined across modules, where a single COAL file (.coal) corresponds to a module. There is only one pass of the compiler from a .coal file to a single pair of .c and .h files with a default target name, or a target name specified at the command line . For example `foo.coal` is compiled to `coalout.c` and `coalout.h`.

3.2.1 Tokens

User defined identifiers, keywords, numerical literals, comments, and operators comprise COAL's tokens. These are discussed below, except for operators which are described in section 3.4.

3.2.2 Comments

One line comments are supported by the pre-pending of # to characters that are to be ignored by the compiler:

```
This is not a comment # This is a comment
```

Characters to the left of the # are not ignored by the compiler. Characters to the right and including # are stripped off. If multi-line comments are desired, then the next line must contain a # .

3.2.3 Identifiers

Identifiers are a sequence of letters, digits, and underscores . An identifier must start with a letter (upper or lower case). Any number of underscores may be used after the first character. An identifier can be of arbitrary length and is case sensitive:

```
f (x) -> x^2
f_1(x) -> 2*x
```

Identifiers are denoted with *identifier* in the grammar.

3.2.4 Keywords

There are four keywords: `if`, `then`, `else`, and `i`. The use of these keywords is described below.

3.2.5 Numerical Literals

All numerical literals are considered to be in the complex plane, where a number has a real and imaginary part. The following exemplify numerical constants:

```
97
97.0
1.0e2 - 2i
7e7i
```

The keyword `i` is used to signify an imaginary part of a complex number. There may only be one `i` in the imaginary term, and it must come at the end of the term. With the difference of the `i`, real and imaginary terms are lexically identical. `i` may not appear by itself which avoids possible namespace identifier clashes.

Real or imaginary literals are described by the following regular expressions:

```
Real      : (integer | float)
Imaginary : (integer | float) 'i'
```

where

```
digit = ['0'-'9']
sign = ('+'|'-')*
expon = 'e' sign digit+
integer = digit+
float = digit+ expon
      | digit+ '.' digit* expon?
      | digit* '.' digit+ expon?
```

Numerical literals map to the `Number` type.

Real and imaginary literals are denoted by *real-literal* and *imaginary-literal* in the grammar.

3.3 Meaning of Identifiers

Identifiers refer to variables of numbers, arrays of numbers, or functions. The types of variables and parameters of functions used in a program do not need to be explicitly defined. Types are inferred from usage.

3.3.1 Variables of `Number`

Identifiers can be defined as variables of the `Number` type, which represent the real and imaginary parts of a complex number. The real and imaginary parts of `Number` each have the same numerical range as an IEEE-754 double floating point number.

Variables of `Number` are always passed by value. They have only automatic scope and so are either defined in a function or passed to or from a function.

```
K <- (3.4+9.1i) / 7.8i # K is defined and assigned
```

3.3.2 Arrays of `Number`

Variables may also represent arrays of `Number`. Arrays are referred to by handles and as such are passed around from or to functions by reference. Array references do not go out of scope when returned from a function even though the identifier referring to the array will. Arrays may be passed into a function or created inside the scope of a function.

Arrays passed in from external C code calling COAL functions are mutable. Arrays created directly by the range operator within COAL code are immutable. Arrays created with the map operator within COAL are mutable.

```
f(x) -> x[9] * 8       # x is mutable: passed in from C
x <- 1..100\2          # x is immutable 2,4,6…100
x <- (n->n){1..100\2} # x is mutable 2,4,6…100
```

See range operator below (Section 3.4.2.3).

3.3.3 Functions

There are two kinds of functions, named and lambda (unnamed). Functions must always have a return value.

Named functions are always defined at module scope and cannot be defined within another function. A function can contain a single expression, returned after evaluation, or a semicolon separated series of expressions the last of

which is returned after evaluation. Named functions are denoted in the grammar as such:

function-definition:
identifier (*argument-list-opt*) -> *expression*

A named function may call itself recursively. Mutual recursion of two or more functions is not supported as COAL depends on a simple forward declaration model.

Named functions are defined in a global namespace and must be unique across all compiled COAL modules.

Examples:

```
constant(x) -> 3

sum(x,y) -> x + y


stuff(x,n) ->
    a <- x[n]
  , b <- x[n-1]
  , a^b
```

Only named functions are accessible from C (see section 3.5 on binding to C).

Lambdas (or unnamed functions) can only be defined within other functions and acquire the scope of the function in which they are defined. Lambdas must have at least one argument and must be invoked where they are defined using the function invocation operator (see 3.4.6). Lambdas are denoted in the grammar as such:

lambda-definition:
(*argument-list* -> *expression*)

Examples:

```
named(j,k) -> j + (n->n*k)(j)  # j + j*k

something(x) ->  (n->n*(h->h*2)(n))(x)   #  x*x*2
```

Identifiers for variables of `Number` or array of `Number` may only defined within the scope of a named function or lambda and must not clash with each other, if different types, or with named function identifiers. Declarations for

`Number` or array of `Number` variable identifiers are made either in the parameter list of a function definition or in an assignment expression. The following illustrates this scoping:

```
                # x    j    k  defined?
g(x) ->         # yes  no   no
  j <- x*x;     # yes  no   no
  k <- j*j;     # yes  yes  no
  k             # yes  yes  yes
```

3.4 Expressions

There are no procedural statements in COAL but only expressions involving the types `Number` or array of `Number`. Every function must return a result which is the evaluation of an expression, all the way to the point at which a COAL function is invoked from C.

The type of an expression, that is whether it is `Number` or array of `Number`, is determined by type inference.

3.4.1 `Number` Expressions

Primary `Number` expressions consist of either a numerical literal (3.2.5), a variable of `Number` (3.3.1), or a grouping of a `Number` expression (3.4.3):

expression:
identifier
real-literal
imaginary-literal
`(` *expression* `)`

Composite `Number` expressions can be formed by use of the following operators which take as operands primary `Number` expressions or other composite `Number` expressions.

3.4.1.1 Math Operators

The following table lists the math operators that work on `Number`. Operators are grouped at the same level of precedence. The bottom of the table has the highest level of precedence. All operators take and return the `Number` type.

Operators	Description	Associativity
`+ -`	Add and subtract	Left
`* /`	Multiple and divide	Left
`-`	Unary minus	Right
`^`	Exponentiation	Left

Math operators are denoted as follows in the grammar:

expression:
expression `+` *expression*
expression `-` *expression*
expression `*` *expression*

expression / *expression*
expression ^ *expression*
– *expression*

3.4.1.2 Relational Operators

The following table lists the relational operators that work on `Number`. Operators are grouped at the same level of precedence. All relational operators have lower precedence than the math operators. The bottom of the table has the highest level of precedence. All operators take and return the `Number` type.

Operators	Description	Associativity
= <>	Equal and not equal	Left
< <= > >=	Less than (or equal) Greater than (or equal)	Left

Because there is no Boolean type, these operators return the value 1.0 for true and 0.0 for false. Because "greater than" and "less than" are not defined for complex numbers, the imaginary parts of operands for "less than (or equal)" and "greater than (or equal)" are ignored when these operators are used.

The "equal" operator is provided to be complete but it is not recommended to test the equality of two floating point complex numbers. In addition to the "equal" operator, a built in function, `distance`, will be provided to test the proximity of two complex numbers (See 0).

Relational operators are denoted as follows in the grammar:

expression:
expression < *expression*
expression <= *expression*
expression > *expression*
expression >= *expression*
expression = *expression*
expression <> *expression*

3.4.1.3 Logical Operators

No specific logical operators are defined in COAL. However, logical operators can be substituted with math operators to get the same effect:

COAL	Meaning
`(a>b) * (d<e)`	(a>b) AND (d<e)
`(x=0) + (y=0)`	(x=0) OR (y=0)

For logical "not" functionality, a built-in function, `not`, is provided to invert logic.

3.4.1.4 if then else

Conditional expressions are formed with this syntax:

expression:
`if` *expression1* `then` *expression2* `else` *expression3*

The imaginary part of *expression1* is ignored. If the absolute value of the real part of *expression1* is greater than or equal to .5, then *expression2* will be evaluated; if it's less than .5, then *expression3* will be evaluated.

The `else` keyword is not optional.

Conditional expressions may be nested:

```
if (a>b) then if (a>c) then 2i else -2i else (2 + 2i)
```

3.4.2 Array of `Number` Expressions

COAL supports arrays of `Number`. Like the type `Number`, expressions can be made of arrays or `Number` but over a different set of operators.

The primary array of `Number` expressions consist of reference to an array of `Number` (3.3.2), or a grouping of an array of `Number` expression (3.4.3):

expression:
    identifier
    `(` *expression* `)`

3.4.2.1 Map Operator

A built-in operator is provided to transform one array to another array. This is the primary mechanism by which new arrays can be created in COAL. It has the following syntactical form:

expression:
    identifier `{` *expression* `}`
    lambda `{` *expression* `}`

identifier refers to a named function definition and *lambda* an unnamed function definition. *expression* is an array of `Number` which is the array to be "mapped". The function defined for *identifier* or *lambda* must take exactly one argument which gets passed to it consecutively all of the elements of the array (*expression*) passed into the operator. A new array is returned that gets formed like this:

```
function(array[0]), function (array[2]), function (array [3]), …
```

Some examples:

```
(n->n){10…100\10} # 10, 20, 30…
(x->x*x){-10..-1} # 100, 81, …
```

Arrays created by the map operator remain allocated until a call is made to the COAL C API function cl_free(). All arrays previously allocated become invalid after a call to cl_free(). Use of an invalid array with the map operator will result in a runtime error (see 3.5).

3.4.2.2 Reduce Operator

A built-in operator is provided to transform an array into an expression, which could be a `Number` or another array of `Number`. It has the following form:

expression:
identifier { *expression1, expression2* }
lambda { *expression1, expression2* }

identifier refers to a named function definition and *lambda* an unnamed function definition. *expression1* is the initial value used for the reduce operation. *expression2* is an array of `Number` which is the array to be "reduced". The function defined for *identifier* or *lambda* must take exactly two arguments which get passed to them consecutively all of the elements of the array (*expression2*) and the accumulated result that started with the initial value (*expression1)*. A new array is returned that gets formed like this:

```
function(function(init_value, array[0]), array[1]) …
```

Some examples:

```
(sum, a-> sum + a){0,1..10} # sum numbers 1 to 10

# triple elements in array, in place
(arr, n-> arr[n]<-arr[n]*3; arr)(x, 0..last(x))
```

Use of an invalid array (i.e. deallocated with cl_free()) with the reduce operator will result in a runtime error (see 3.5).

3.4.2.3 Range Operator

The range operator generates immutable arrays containing real integer numbers. It has the syntactic form:

expression:
expression1 `..` *expression2*
expression1 `..` *expression2* `\` *expression3*

expression1, *expression2*, and *expression3* (abbreviated as *e1, e2,* and *e3,* respectively) are of `Number` type. The real parts of the operands automatically have their fractional parts truncated (i.e. 1.9 becomes 1) and imaginary parts set to 0. If literals are used for these operands, the compiler enforces that only real integers (see 3.2.5) are used. *e1* is the starting value

in the array and *e2* the upper limit value of the array. *e3* may optionally specify a step increment. The default step value is 1.

The value for *e3* must not be zero and must be positive if *e2* is greater than *e1* or negative if *e1* is greater than *e2*. If these constraints are not met for *e3,* a run time error will be raised (see 3.5). If *e1* equals *e2,* step must still be non-zero but its sign becomes irrelevant (see below).

The number of values, N, generated from this operator is:

$$N = \text{floor}((e2 - e1) / e3) + 1 \quad ,$$

where floor is an operator that truncates the fractional part of the above quotient.

The returned values from this operator when indexed from 0 to N-1 are:

$$e1 + 0*e3,\ \ e1 + 1*e3,\ \ldots\ e1 + (N-1)*e3$$

The following table exemplifies valid values for *e1*, *e2*, and *e3* and the resulting sequences:

e1	e2	e3	Sequence
0	9	1	0,1,2,3,4,5,6,7,8,9
-10	-1	2	-10,-8,-6,-4,-2
10.9	101.7	10	10,20,30,40,50,60,70,80,90,100
3	-6	-3	3,0,-3,-6
9	10	100	9
-9	-10	-100	-9
9	9	-123	9

The range operator acts as a seed into the map and reduce operators to create and index arrays, respectively.

3.4.2.4 Array Indexer

expression:
expression1 [*expression2*]

The array index operator is used to access specific elements of an array. *expression1* is of type array of `Number`. *expression2* is of type `Number`. Similarly to the range operator, the real part of *expression2* has its fractional

part truncated and the imaginary part is set to 0. Also, only integer literals may be used for *expression2* (see 3.2.5).

This operator returns a `Number` type. It is possible to assign a value to an array element using this operator if the array is mutable (see 3.4.4).

3.4.3 Grouping

The grouping operator is usedto force precedence for a sub-expression where the precedence would otherwise cause an expression to be evaluated differently.

It has the form:

expression:
    (*expression*)

expression inside the parentheses can be of type `Number` or array of `Number`.

3.4.4 Assignment Operator

The assignment operator is used to bind a `Number` or array of `Number` expression to an identifier:

It has the form:

expression:
    identifier <- *expression*

expression is either a *`Number`* or an array of `Number`. Within the resulting expression, *identifier* has no scope but rather must be used in conjunction with the sequence operator and referenced from a proceeding expression (3.4.5).

The result of this operation is *expression*, with a side-effect of defining *identifier* to be bound to *expression*. This operator is right associative.

3.4.5 Sequence Operator

The sequence operator is used to allow intermediate expressions, specifically assignments (see 3.4.4), to be evaluated before the last expression in the sequence which can incorporate the results of these intermediate expressions.

This operator has the form:

expression:
expression1 `;` *expression2*

expression1 would typically be an assignment and *expression2* is the resulting expression. Because of this behavior, any other sort of expression besides assignment doesn't usually make sense for *expression1*. However other sorts of expressions are allowed for the sake of debugging (see 3.7).

expression1 and *expression2* can be of different types.

Examples:

```
a<-3; b<-a; b^2      # 9
      |← a's scope is live from here

x<-19; x<-x*2; x+1   # 35
       |← x first live from here
                |← x w/ new value live from here

101; 97; 17.2        # 17.2
```

Variable identifiers can be re-assigned multiple times within a sequence.. However, variable identifiers may not be redefined to a different type. For example, a `Number` cannot then be overridden to be an array of `Number` using the sequence operator:

```
x<-1..10; x<-x[3]; x # ERROR! 2nd x is different type.
```

3.4.6 Function Invocation

Invoked named functions result in an expression defined as follows:

expression:
identifier `(` *invoke-argument-list-opt* `)`

Invoked lambda's result in an expression defined as follows:

expression:
lambda-definition（*invoke-argument-list*）

Examples:

```
f(x) + g(x+y)

stuff(x, x^2, x^3) + 23

(a, b-> if (a>b) then a*b else a/b)(4,5)

beg()..end()\step()

#recursion
fact(n) -> if (n=1) then 1 else (n * fact(n-1))
```

3.5 Binding to C

As described above, the COAL compiler will produce a pair of output files consisting of a .c and .h file, which can then be integrated into a target C application. In the generated .h file will be C bound declarations of COAL user defined functions as well as the declaration of a small C API used to facilitate the calling of and error handling of COAL functions. Below is table summarizing this API:

COAL C API Element	Returns	Description
`coal_num`		Structure that represents a complex number
`coal_arr`		Structure that represents an array of `coal_num`
`cl_num(double, double)`	`coal_num`	Real part set to `a`, imaginary to `b`
`cl_dbl_re(coal_num)`	`C double`	Real part of `coal_num`
`cl_dbl_im(coal_num)`	`C double`	Imaginary part of `coal_num`
`cl_real_arr(double* c_arr, int N)`	`coal_arr`	`c_arr` points to a C `double` array containing only real elements. N specifies the physical number of doubles in array. Length of returned array is N.
`cl_cplx_arr(double* c_arr, int N)`	`coal_arr`	`c_arr` points to a C `double` array containing only real and imaginary elements, interleaved (even offsets are real). N specifies the physical number of doubles in array. Length of returned array is N/2, half the physical number of doubles.
`cl_get_elem(coal_arr a, int i)`	`coal_num`	Get element of `i` of `a`.
`cl_put_elem(coal_arr a, int i, coal_num v)`	`coal_num`	Put `v` at index `i` of `a`.
`cl_len(coal_arr)`	`int`	Number of elements in `coal_arr`.
`cl_last(coal_arr)`	`int`	Last index of `coal_arr`.
`cl_free()`		Free all arrays dynamically created with map operator. Use `cl_valid_arr()` to check if an array has been free'd or not.
`cl_valid_num(coal_num)`	`int`	Non-zero if `coal_num` is valid. See error handling below.
`cl_valid_arr(coal_arr)`	`int`	Non-zero if `coal_arr` is valid. See error handling below.
`cl_last_err()`	`int`	Returns error code of last encountered error, 0 if no error. See error handling below.

3.5.1 Error Handling

It is possible to generate run time error conditions in COAL functions. As a result, the author of the calling C code is responsible for checking the validity of the returned a COAL value (`coal_num` or `coal_arr`), and the for any error codes (cl_last_err()). The following code exemplifies this:

```
coal_arr arr;
coal_num res;

arr = foo(); /* arr generated with map operator */
if (!cl_valid_arr(arr))
    printf ("COAL Error %d\n", cl_last_err());

res = bar();
if (!cl_valid_num(res))
    printf ("COAL Error %d\n", cl_last_err());

cl_free(); /* arr no longer valid */
assert(!cl_valid_arr(arr));
```

The following table enumerates possible COAL run time error codes defined in the generated .h file:

Error	Description
`CL_ERR_ILLEGAL_INDEX`	Attempt to index into an array beyond its bounds.
`CL_ERR_ARRAY_IMMUTABLE`	Attempt to write to an array created by a read-only range expression.
`CL_ERR_OUT_OF_MEMORY`	Attempt to create an array with map operator with insufficient storage – need to call cl_free() or make storage larger (see 3.7).
`CL_ERR_STEP_IS_ZERO`	Attempt to set step argument of range operator to 0 (see 3.4.2.3).
`CL_ERR_STEP_WRONG_SIGN`	Attempt to set step argument of range operator to sign inconsistent with bound arguments (see 3.4.2.3).
`CL_ERR_BAD_ARRAY`	Attempt to use map or reduce operator on an array that has been freed or is otherwise internally inconsistent.

3.5.2 External Arrays

In order to facilitate convenient interoperation with C, COAL allows a way for externally defined arrays to be used in COAL functions. This flexibility requires that the end user take care in setting up arrays to avoid illegal memory accesses. The following example illustrates the use of external arrays with coal:

```
double arr_re[5];   /* array of reals */
double arr_cp[10];  /* array of complex */

coal_arr carr_re;
coal_arr carr_cp;

coal_num res;

/* initialize arr_re and arr_cp ...  */

/* embed into COAL array types */
carr_re = cl_real_arr(arr_re, 5);
carr_cp = cl_cplx_arr(arr_cp, 10);

assert(5==cl_len(carr_re));
assert(5==cl_len(carr_cp));

/* operate on external arrays */
res = foo(carr_re, carr_cp);
if (!cl_valid_num(res))
    printf ("COAL Error %d\n", cl_last_err());
```

3.5.3 Required C Libraries

COAL depends on the following C library headers in order to function properly:

`math.h` – for builtins and ^ operator.
`setjmp.h` – for run time error handling.
`stdio.h` – for dumping trace data to standard out when -trace specified.

3.6 Built-in Functions and Constants

Below is a list of built in functions to improve the usability of COAL:

Function	Return Type	Description
`len(array of Number)`	`Number`	Number of elements in array
`last(array of Number)`	`Number`	Last index of array. Length - 1.
`re(Number)`	`Number`	Zero out imaginary part of argument.
`im(Number)`	`Number`	Zero out real part of argument.
`conj(Number)`	`Number`	Complex conjugate
`mag(Number)`	`Number`	Magnitude of complex, zero out imaginary part.
`phase(Number)`	`Number`	Phase of complex (radians), zero out imaginary part.
`distance(Number, Number)`	`Number`	Distance, as a real number, between two complex numbers
`not(Number)`	`Number`	If absolute value of real part of argument less than .5, returns 1; if greater than or equal to .5 returns 0.
`sin(Number)`	`Number`	Sine – argument in radians – imaginary part of argument ignored.
`cos(Number)`	`Number`	Cosine – argument in radians – imaginary part of argument ignored.
`tan(Number)`	`Number`	Tangent – argument in radians – imaginary part of argument ignored.
`exp(Number)`	`Number`	Euler number to the power given by argument. Argument and result can be complex.
`sqrt(Number)`	`Number`	Square root – argument and result can be complex. Equivalent to x ^ .5.

The following constants are defined for convenience:

Constant	Value
PI	3.1415927
PIi	3.1415927i
2PI	6.2831853
2PIi	6.2831853i

3.7 Compiler Options

COAL is invoked on the command line, where the .coal source files are specified and options can be specified. Usage is as follows:

```
coal [options] <file1>.coal <file2>.coal
```

The order in which coal source files appear on the command line determines the order in which functions are defined. Independent modules should appear first, then modules dependent on those modules.

The below table summarizes the compiler options available on COAL:

Option	Description
`-sast`	Dump semantically checked abstract syntax tree for each specified .coal to standard out. Inhibits generation of .c output.
`-o`	Specify root name of c output files. If not specified, default is coalout, resulting in coalout.h/c being generated. Ex. `coal -o minime minime.coal` produces `minime.h` and `minime.c`.
`-storesize`	Specify the total number of elements in the array store used for dynamic array creation. Default is 128, minimum is 1, and maximum is 65535.
`-trace`	Generate C output with debug tracing. Traces dumped to standard out via printf. This option ignored if -sast specified after. Tracing includes values returned from functions and raised errors. Stack depth of named functions shown by indentation of output. This option is used as a means of generating output for regression testing.
`-help/--help`	Display help for options

3.8 Grammar

The following is the grammar for COAL. Terminals *identifier, real-literal,* and *imaginary-literal* are described above.

The *opt* suffix on a non-terminal means zero or more of the non-terminal.

program: zero or more
    program function-definition

expression:
    identifier
    real-literal
    imaginary-literal
    expression + *expression*
    expression – *expression*
    expression * *expression*
    expression / *expression*
    expression ^ *expression*
    – *expression*
    expression < *expression*
    expression <= *expression*
    expression > *expression*
    expression >= *expression*
    expression = *expression*
    expression <> *expression*
    identifier <- *expression*
    expression [*expression*]
    identifier (*invoke-argument-list-opt*)
    lambda-definition (*invoke-argument-list*)
    identifier { *expression* }
    identifier { *expression, expression* }
    lambda { *expression* }
    lambda { *expression, expression* }
    (*expression*)
    expression . . *expression*
    expression . . *expression* \ *expression*
    expression ; *expression*
    if *expression* then *expression* else *expression*

function-definition:
    identifier (*argument-list-opt*) -> *expression*

lambda-definition:
    (*argument-list* -> *expression*)

argument-list:
    identifier
    argument-list, *identifier*

invoke-argument-list:
    expression
    argument-list, *expression*

4 Project Plan

4.1 Process

The language reference manual was used to scope the amount of work required as well as the complexity and risk associated with COAL's development.

Since type inference is the key COAL's functionality, emphasis was put on understanding how to implement it first. After type inference was achieved, focus was put on the most basic features first, leaving more sophisticated features for last in the event that time ran out.

Testing was approached in a similar fashion where the language reference manual was used as a guide to dictate which features got tested and in which order.

4.2 Programming Style

The following OCaml coding guidelines were used:

- Indent function bodies under function declarations
- When functions are used less then once, inline the function within a parent function or inline its implementation
- In long match statements, put a space between cases
- Use comments strategically to describe "what", not "how"
- Keep names relatively short, but prefer more descriptive names if the abbreviated version of the name is vague

4.3 Timeline

The below table contains the major milestones of the COAL project.

Milestone	Date Achieved
Language Proposal	2009Feb09
LRM	2009Mar10
Setup git Repository	2009Mar14
Parser/Scanner Compiles	2009Apr05
Type Inference/Semantic Checking Works	2009Apr20
Setup Test Automation	2009May03
Code Generation Works	2009May07
Tests Complete	2009May10

4.4 Development Environment

The below table describes the elements of the development environment used to write COAL.

OS	Cygwin, Unix compatibility layer on Windows
OCaml	Standard Windows distribution for OCaml 3.10.2
Source Control	git v1.6.1.2
C Compiler	gcc v3.4.4
Build Automation	GNU make v3.81
Scripting	GNU bash v3.2.48
Editor	notepad++

Because COAL was developed on Cygwin, it should be able to build with little effort on any Unix variant where OCaml is available.

4.5 Log

4.5.1 Written Log

A written log was kept from March 1 to April 19, 2009.

2009Mar01

Decide on source control tool. Going with git to see
if it holds up to the hype. Already familiar with CVS
and Subversion. git is available in the cygwin environment.
Going to upgrade my cygwin to get git.

Decided to avoid the ability to pass/return functions, especially
lambda functions which acquire names from scope outside
the lambda definition. This is difficult because you then
have unique lambda functions for each different acquired scope.
Assuming this were valid COAL syntax:

```
f(x) -> (n -> n*x!)!  # return lambda
```

For the invocations f(1) and f(5), you would have to
manage two instances of lambda function; one where
x=1 and x=5, respectively. Anyhow, going to skip
this complexity unless I can figure out how to do
it easily in the coming weeks.

2009Mar03

Did initial setup of sources; took a stab at scanner and
parser. Ran into some confusion about how to do type inference.
In examples I've seen so far, it's easy because the type is
explicitly spelled out in the grammar. Trying to iron out
parser so I can generate the LRM.

2009Mar04

Perhaps type checking should happen after parsing as a part
of semantic analysis(?) In examples I've seen so
far (calculator/microc) there didn't seem to be a
need for a semantic analysis step to check the AST.

2009Mar05

Looking in the Golden Dragon book confirms that type inference happens after parsing. Also heard this mentioned this evening in the Mar 5th lecture.

2009Mar09

A little bit stuck on how to implement output statements in scanner and parser:

("The number is $(some*expression-1).")

Parser says:

out_expr_list:
  out_expr

2009Mar10

While finalizing first draft of LRM decided to drop support for output statements to keep things a bit simpler. Will instead implement a debug mode which will emit .c code that sends intermediate results to the C stdout - of course this assumes a system with stdout.

2009Mar14

Set up git for source control. Really easy to setup.

Got initial AST setup today. Got parser.ml to compile with it.

2009Mar29

Type inference requires the type unification algorithm, specified in Dragon book. Dragon book shows unification alg and a high level example of how to do type inference on a function body. I would maybe apply this to COAL like so:

```
f(x) ->  x*5 ! # unify what f(x) returns and expression (x*5),
                # but first unify x and 5, then come back up.
```

In the process of unifying I would then check for type inconsistencies and raise an error.

2009Apr09

Trying to follow the Sast/Ast suggestion in the lecture notes. Type inference adds a step where temporary types are generated and then overwritten with an actual type when it's discovered. This requires a special place holder type that's mutable.

2009Apr12

Got most of the semantic checker written and compiled. Will start testing it next.

Also have a first cut at the built in functions. Their implementation will be in C, and their symbols added to the global symbol table, as in the "print" function built-in example of micro-c.

2009Apr19

Found and resolved several bugs with sematic checker. Seems to be working as expected now. Need to generate type inference test cases. Will test by spitting out the sast with the inferred types printed explicitly before variables:

f(x) -> x[0] + 1

becomes

Num f(NumArr x) -> x[0] + 1

Once type inference tests are in place, code generation will be next.

4.5.2 Source Control Log

Below is a log for commits to the repository. Entries are listed latest first. The entries span from March 14 to May 12, 2009.

```
commit 14caf89491be4a60734a3687cf1561aa235a244b
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Tue May 12 22:39:19 2009 -0400

    Cleanup.


commit 711ccd5ca7cea92dfa4c590f9f9a67c60ab487e1
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun May 10 17:42:36 2009 -0400

    Remove simple.coal

commit b5b7c413623f8c73950987de4ad3b1491c1ce0ea
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun May 10 17:40:49 2009 -0400

    Add modules test; improve test for 'array immutable' error.

commit 32843191343d5bbebc38181103933ba5769544a2
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun May 10 17:39:44 2009 -0400

    Update make file for src and test directories.

commit 6d2a941ee4f33f2a01020d4fe84532efc749e6d1
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun May 10 17:38:53 2009 -0400

    Add a couple of mli's and source code clean up.

commit d8d23dc72963c7d0f1a3da7f826ed1b52268e32f
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun May 10 01:05:18 2009 -0400

    Add cbindings_test_assert to suite of tests.

commit c2ff8d5a5e7af80424b9e4d4f89ad6560ac5b861
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun May 10 01:04:40 2009 -0400

    Add better cleanup to Makefile.

commit 241f46e6dfafb36f4d9983f49096468866625f26
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun May 10 01:02:39 2009 -0400

    Fix typo in cl_dbl_im.

commit c4013da0fdd5160a27b5cddd6cbd73ffb99f7f3d
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat May 9 22:41:38 2009 -0400

    Convert some builtins implemented as macros to proper functions.
    This is so that they can be passed to the map and reduce operators.

commit a54a0e0d55923f2ac9255fc021a24fb441167699
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat May 9 22:36:42 2009 -0400

    Gold file update after fix in sast printer.

commit 22787b821b2753e80796eb89cf0aa028a75f0987
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
```

```
Date:   Sat May 9 22:36:01 2009 -0400

    Update makefile with filter test; minor changes to dft example.

commit 6e909508d5cc1f25a3257e88392b439102448fcb
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat May 9 22:33:44 2009 -0400

    Add filter example.

commit 91d4e1023d66bcb1707e96b5a89c1161f7d302a1
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat May 9 21:03:31 2009 -0400

    Imaginary numbers not getting printed with 'i' in sast dump.

commit 42faef1b01b4addd7af30b02e27305114312a612
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat May 9 17:06:45 2009 -0400

    Update makefile to provide automation for building and running tests.

commit 1b88115400ba7f58f981c39053a9b1dd5b81d727
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat May 9 17:06:13 2009 -0400

    Remove unneeded files.

commit 75843906008d7d028eb00e1ec2472a5db5ad1311
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat May 9 16:58:53 2009 -0400

    Add tests and automation for tests.

commit d80ec4f993c8d30dfc03716948a35c75ac93cb2c
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat May 9 16:54:41 2009 -0400

    Add higher precedence to [] and {} operators.

commit 81c93add80e5d38605c868a3dad783831644ce78
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat May 9 16:51:46 2009 -0400

    Fix cycle issue with variant structures used to rep types;
    Increase precedence of [] & {} operators;
    Add output of errors to trace mode.

commit 88d223f23bb3a176ccd30331fee0233a7dac1b33
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Thu May 7 18:10:48 2009 -0400

    Add build automation for lang_test.

commit 040b2d10192f90e8d90fe998305027083e8c5ffb
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Thu May 7 18:09:44 2009 -0400

    Correct formatting string in dump function printf that caused seg fault.

commit 44fa04328d1d8c70b3b6cd5c656976b50632d9ce
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Thu May 7 18:09:06 2009 -0400

    Put constant definition before id definition so that constants
    are recognized first.

commit 18afb5242f6fd7e1414b9cedcc8329a6890a0e03
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Thu May 7 13:31:13 2009 -0400
```

```
    Disallow ' character from COAL names to avoid naming issues in generated C.

commit 680550daf98a4a1574966e5a3bada59e70f39ec7
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Thu May 7 13:30:29 2009 -0400

    Refactor compile options.  Add debug trace option.

commit 02bc3b3b4202b2f2fe8a67752ccbe72ef726f3ed
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Thu May 7 13:28:41 2009 -0400

    Add coalopts.mli to more easily share compiler options between modules.

commit ff52a69aa66f6a26ab3ee9fce74f47e815d9d0fc
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Thu May 7 01:09:10 2009 -0400

    Overlooked negate operation; added def to ast/sast and tree walking code.
    Add more user access functions for creating arrays.

commit 8e1a98a90743da56e5486b50bc5d8b988ca861e4
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Wed May 6 16:47:09 2009 -0400

    First cut at error handling using setjmp.  Get map/reduce functional.

commit d4c57f99d6becff6230ded2e46fdb40dc07e139d
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Mon May 4 00:37:44 2009 -0400

    First cut at code generation for map, array index, and range operators.

commit b7b01a39cad279c5edddf70ebb0edc1cc46b103d
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun May 3 21:21:17 2009 -0400

    Get lambda functions functional with ability to acquire parent scope
    using static links.  Nested lambda functions implemented.

commit 5a18a72bc229e0ac8922f26956fddc0d7f33d939
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat Apr 25 00:05:42 2009 -0400

    Reverse order of arguments on reduce operator invocation
    to make consistent with signature of function used for
    reduce.

commit 88c94872f5d1e7334b2a64ffa0fd7520ac5a0746
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Fri Apr 24 23:57:18 2009 -0400

    Make one big SAST for all input modules then emit all
    together.  This simplifies code emission and built-in
    function strategy.

commit f22810ec02e252eee28bbe17b37a918740fd28db
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Fri Apr 24 23:11:14 2009 -0400

    Add in constants.  Change N() to len().

commit 5fd57e61cc4042938a43a46c9317a8659d953f46
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Tue Apr 21 22:36:07 2009 -0400

    Add coverage for unnamed functions and built-in functions to type inference test.

commit e15b6865f797a29157e3fb1f53f70f47b2d37eb9
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Tue Apr 21 22:35:18 2009 -0400
```

```
    Add preliminary built-in function support.

commit 7a788a61b0836319480ca9de418697947a840880
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Tue Apr 21 16:19:22 2009 -0400

    Add more type inference test cases.

commit af75731ebf6ef254944fa663028bab0428dd8f8d
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Tue Apr 21 16:07:43 2009 -0400

    Add special case to scanner to disambiguate floating point number
    and integers right before the range operator (Ex. 1..9).

commit ce862a2e19c771ce04729860dc63e4be08164a57
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Mon Apr 20 00:10:56 2009 -0400

    Get semantic checker working.

commit 4bcd1f7f55dac23d8e99031fb8c58ceb3472fa80
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Mon Apr 20 00:04:21 2009 -0400

    Fix parsing error for '<-' op.  Fix function parse order error.  Add first wave of
tests.

commit ba7d41a690a25f548213e7aaea3004de55bbfbc6
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun Apr 19 01:20:54 2009 -0400

    Fix bugs in semantic checker.  Add sast debug print.

commit cca63df65095cf273463cfe41c3907e3c3acede1
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun Apr 12 18:08:07 2009 -0400

    First cut at semantics module

commit f8ddcae11e5945a30970b813033a509e85408cf8
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sun Apr 5 13:03:37 2009 -0400

    Add cmd line processing and entry point

commit 45dbc724ae2260a699885db9c834005768d974b8
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat Mar 14 19:38:29 2009 -0400

    Add ast.mli and Makefile.  Define parser actions.

commit 510d9d715b8265f2baefd640c2df62bd1da8c6a2
Author: U-HARU\Eliot Scull <Eliot Scull@HARU.(none)>
Date:   Sat Mar 14 15:17:49 2009 -0400

    Initialize repository.
```

5 Architectural Design

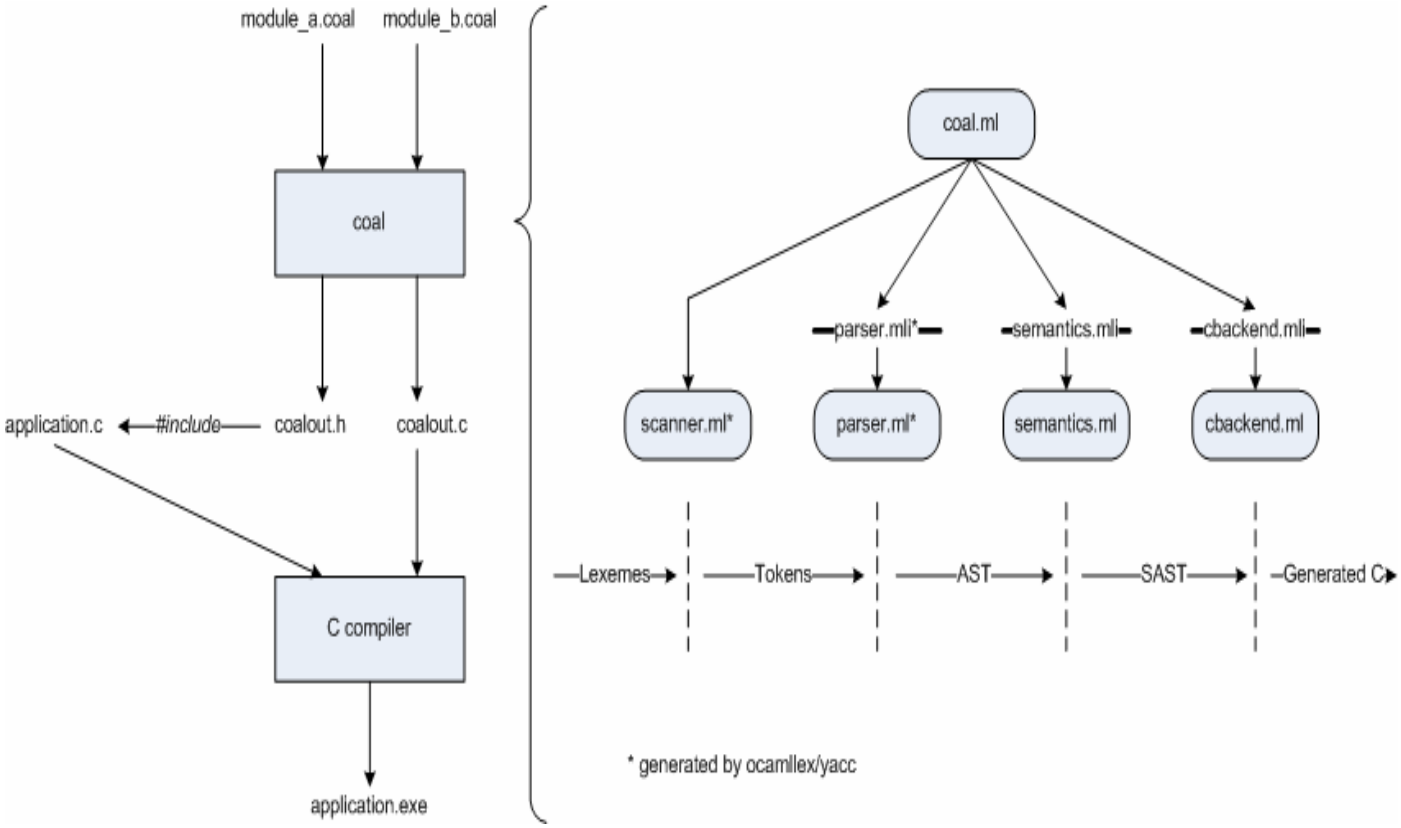


Figure 5-1 COAL Block Diagram

The overall architecture of COAL is depicted in Figure 5-1. The left side of the figure shows the data flow from input to output files. The right of the figure shows the main constituent OCaml modules of the implementation and how they are connected.

The `coal.ml` module acts as the entry point for the compiler and delegates responsibility to subordinate modules that support different compile phases. .mli interface files are employed to minimize coupling between modules and as such are the means by which `coal.ml` communicates to its subordinate modules, with the exception of the scanner. This strategy allows for the straight-forward insertion of new modules that might perform optimization or generate code to different targets at a later point.

The modules of the architecture and their role in compilation are described below.

5.1 Scanner/Parser

The OCamllex/yacc generated scanner and parser are responsible for ultimately generating an unverified Abstract Syntax Tree, or AST. The types used to represent the AST are contained in `ast.mli` (not shown in above figure).

The scanner and parser are de-coupled from each other as the parser obtains lexemes, not from the scanner module itself, but rather through an intermediate buffer defined the in the Lexbuf module library.

After an unverified AST is generated, it is passed on to the `semantics.ml` module so that it can be semantically checked before code generation.

5.2 Semantics

Types in COAL are not explicitly declared. This means that the types of variables and return values need to be inferred from usage. The `semantics.ml` module takes on this responsibility by walking the AST and unifying the types of expressions that it comes across. Inconsistent type usages result in an exception and a halt of the compiler. Upon success of walking the AST, a Semantically checked Abstract Syntax Tree, or SAST, is produced which is a guaranteed correct representation of the source program.

The types that represent the SAST as contained in `sast.mli`.

5.3 CBackend

The `cbackend.ml` module traverses the SAST and emits a C code representation which can be directly compiled by an ANSI C99 conformant compiler. All code generated by this module was tested under `gcc` (v3.4.4) with the `-pedantic` option.

COAL expressions somewhat overlap with the semantics of C expressions and as a result translating basic COAL expressions to C is straight-forward. However there are three areas that required special handling, described below.

5.3.1 Static Links

C99 does not support nested or lambda type functions. As a result, "static links" were devised to allow COAL lambda functions to acquire their parent's scope. These static links are implemented as static array of void pointers that get assigned with the address of local variables of parent scopes. When a lambda function needs to refer to its parent scope's variable, it refers to this table of links and copies the address of the target variable to a local pointer defined within the emitted lambda function; this copying of the pointer is crucial for avoiding collisions as the link table gets re-used between different function calls.

The assignments and lookups of the entries to parent scope variables in the link table are calculated in a separate pass on the SAST before code generation. By doing this, the maximum required size of the link table can be determined, if it is needed at all.

For examples of this implementation of static links, please see references to the array "`_links`" in the generated code of the COAL examples in sections 6.1.1 and 6.2.1.

5.3.2 Array Eras

The dynamic storage model in COAL is extremely simple where any arrays are incrementally allocated, as storage limits allow, and then freed all at once. This leaves the potential for client C code to inadvertently use an array that refers to freed storage. In order to prevent this scenario, an "era" scheme is used to check the validity of an array before an array operation, such as map or reduce.

An "era" is an integer that represents the current generation of arrays that get created before the next free. When free is called, the era is incremented. This way we know an array is invalid when its era, assigned at creation, does not match the current era.

The map and reduce operators will throw a run-time error if their designated array operands are invalid. The validity an array may be checked by calling `cl_valid_arr` from C. For efficiency, the array element accessor, "x[n]", does not check the era.

5.3.3 Run Time Error Handling

COAL provides a rudimentary run time error handling mechanism to respond to such events as running out of array storage or accessing arrays with invalid indexes. C does not support exceptions, so traditionally error conditions are discovered by checking return values of functions. Since this approach complicated code generation for COAL, a method by where `setjmp/longjmp` are used was developed.

The approach taken was to have `longjump` called in the event of a run-time error. This results in the stack unwinding and the flow of execution returning to where `setjmp` was called, preferable in the top-level function call made from C. To ensure that `setjmp` is only called in the top-level function call made from C, a call level counter is employed to keep track of where the call stack is (see references to `_cl_call_lev` in sections 6.1.1 and 6.2.1).

When the flow of control returns to setjmp in the case of an error, the top level COAL function returns an invalid `coal_num` or `coal_arr`. This indicates to the caller that a run time error occurred. COAL's run time errors are described in section 3.5.1.

6 Test Plan

This following describes two representative COAL program examples and the test suite used for regression testing COAL functionality.

6.1 DFT Example

The following example demonstrates the use of a single bin DFT taken on a periodic signal's fundamental and second harmonic which is then used to calculate harmonic distortion. A fictitious signal is used as in input to the DFT and is generated once within C and once within COAL to show COAL's flexibility with arrays.

6.1.1 DFT Example Listing

The files for this example, shown below, are summarized in this table:

File	Description
`dft.coal`	Where COAL source for example resides. This is where the DFT function is defined.
`dft.h`	Header file generated from COAL.
`dft.c`	C file generated from COAL – definitions for user functions are located towards the end of listing.
`dft_ex.c`	Application that calls COAL functions in dft.c.

`dft.coal:`

```
dft_one_bin(k, x) ->

  N <- len(x);
  (s, n-> s + x[n] * exp((-2PIi * k * n)/N)) {0, 0..last(x)}

distortion(s, f1, f2) ->

  fundamental <- dft_one_bin(f1, s);
  harmonic <- dft_one_bin(f2, s);
  mag(harmonic)/mag(fundamental)

make_two_tone(N, f1, a1, f2, a2) ->

 (n-> a1 * sin(2PI*f1*n/N) + a2 * cos(2PI*f2*n/N)) {0..N-1}
```

dft.h (generated):

```
#ifndef _dft_h
#define _dft_h

#define CL_ERR_ILLEGAL_INDEX 0x1
#define CL_ERR_ARRAY_IMMUTABLE 0x2
#define CL_ERR_OUT_OF_MEMORY 0x3
#define CL_ERR_STEP_IS_ZERO 0x4
#define CL_ERR_STEP_WRONG_SIGN 0x5
#define CL_ERR_BAD_ARRAY 0x6

typedef struct
{
   double re;
   double im;
} coal_num;

typedef union
{
   struct
   {
       unsigned int kind: 2;
       unsigned int era: 14;
       unsigned int num: 16;
       coal_num* parr;
   } cp;
   struct
   {
       unsigned int kind: 2;
       unsigned int reserved: 14;
       unsigned int num: 16;
       double* parr;
   } re;
   struct
   {
       unsigned int kind: 2;
       unsigned int reserved: 14;
       unsigned int num: 16;
       short start;
       short step;
   } rng;
} coal_arr;

extern void cl_free();
extern int cl_last_err();
#define cl_valid_num(x) (!isnan(x.re))
#define cl_dbl_re(x) (x.re)
#define cl_dbl_im(x) (x.im)
extern int cl_valid_arr(coal_arr);
extern coal_num cl_get_elem(coal_arr, int);
extern coal_num cl_put_elem(coal_arr, int, coal_num);
#define cl_len(x) (x.cp.num)
#define cl_last(x) (x.cp.num-1)
extern coal_num cl_num(double re, double im);
extern coal_arr cl_real_arr(double* re, int size);
```

```
extern coal_arr cl_cplx_arr(double* re, int size);
extern coal_num dft_one_bin(coal_num k, coal_arr x);
extern coal_num distortion(coal_arr s, coal_num f1, coal_num f2);
extern coal_arr make_two_tone(coal_num N, coal_num f1, coal_num a1,
coal_num f2, coal_num a2);
#endif
```

dft.c (generated):

```
#include <math.h>
#include <setjmp.h>
#include "dft.h"

#define CL_STORE_SIZE 1024
#define CL_MAX_ARR_SIZE 65535
#define CL_ARR_CPX_INT 0
#define CL_ARR_CPX_EXT 1
#define CL_ARR_RE_EXT 2
#define CL_ARR_RNG 3

static coal_num _cl_zero = {0.0, 0.0};
static coal_num _cl_one = {1.0, 0.0};

#define _cl_handle_err(ret)\
if (_cl_call_lev==0) \
{\
 if ((_cl_last_err=setjmp(_cl_top))!=0)\
 {\
   _cl_call_lev = -1;\
   return ret;\
 }\
}

static int _cl_last_err = 0;
static int _cl_call_lev = -1;
static jmp_buf _cl_top;

static coal_num _cl_arr_store[CL_STORE_SIZE];
static int _cl_used = 0;
static int _cl_era = 0;

void _cl_compile_asserts()
{
#define COAL_CT_ASSERT(e) ((void)sizeof(char[1 - 2*!(e)]))
COAL_CT_ASSERT(sizeof(double)==8u);
COAL_CT_ASSERT(sizeof(unsigned int)==4u);
}
```

dft.c (generated – con't):

```
coal_num _cl_invalid_num()
{
   unsigned int nan[4];
   nan[0]=0xFFFFFFFF;
   nan[1]=0xFFFFFFFF;
   nan[2]=0xFFFFFFFF;
   nan[3]=0xFFFFFFFF;
   return (*((coal_num*)(&nan)));
}

coal_arr _cl_invalid_arr()
{
   coal_arr bad;
   bad.cp.kind = 0;
   bad.cp.era = 0;
   bad.cp.num = 0;
   bad.cp.parr = 0;
   return bad;
}

#define _cl_check_arr(ca) { if
((ca.cp.parr==0)||((ca.cp.kind==CL_ARR_CPX_INT) &&
(ca.cp.era!=_cl_era))) longjmp(_cl_top, CL_ERR_BAD_ARRAY); }

#define _cl_check_idx(idx,ca) { if ((idx<0)||(idx>=ca.cp.num))
longjmp(_cl_top, CL_ERR_ILLEGAL_INDEX); }

int _cl_is_zero(coal_num x)
{
   return ((x.re==0.0) && (x.im==0.0));
}

int _cl_is_true(coal_num x)
{
   return ((x.re>=0.5)||(x.re<=-0.5));
}

coal_num _cl_rect(double re, double im)
{
   coal_num r;
   r.re = re; r.im = im;
   return r;
}

coal_num _cl_polar(double m, double theta)
{
   coal_num r;
   r.re = m * cos(theta);
   r.im = m * sin(theta);
   return r;
}
```

dft.c (generated – con't):

```
double _cl_mag_dbl(coal_num x)
{
   return sqrt(x.re*x.re + x.im*x.im);
}

double _cl_phase_dbl(coal_num x)
{
   return atan2(x.im, x.re);
}

coal_num _cl_negate(coal_num x)
{
   x.re = -x.re; x.im = -x.im;
   return x;
}

coal_num _cl_add(coal_num a, coal_num b)
{
   coal_num r;
   r.re = a.re + b.re;
   r.im = a.im + b.im;
   return r;
}

coal_num _cl_sub(coal_num a, coal_num b)
{
   coal_num r;
   r.re = a.re - b.re;
   r.im = a.im - b.im;
   return r;
}

coal_num _cl_mul(coal_num a, coal_num b)
{
   coal_num r;
   r.re = (a.re * b.re - a.im * b.im);
   r.im = (a.im * b.re + a.re * b.im);
   return r;
}
```

dft.c (generated – con't):

```
coal_num _cl_div(coal_num a, coal_num b)
{
   coal_num r;
   double denom;

   if (_cl_is_zero(a))
   {
       r.re = 0.0;
       r.im = 0.0;
       return r;
   }

   if (_cl_is_zero(b))
   {
       r.re = HUGE_VAL;
       r.im = HUGE_VAL;
       return r;
   }

   denom = (b.re * b.re + b.im * b.im);

   r.re = (a.re * b.re + a.im * b.im) / denom;
   r.im = (b.re * a.im - a.re * b.im) / denom;
   return r;
}


coal_num _cl_expon(coal_num x, coal_num p)
{
    coal_num r;
    double ln_m;
    double theta;
    coal_num q;
    double m_prime;
    double theta_prime;

    if (_cl_is_zero(x)) return x;

    if (_cl_is_zero(p)) return _cl_rect(1.0, 0.0);

    ln_m = log(_cl_mag_dbl(x));
    theta = _cl_phase_dbl(x);
    q = _cl_mul(_cl_rect(ln_m, theta), p);

    m_prime = exp(q.re);
    theta_prime = q.im;

    r = _cl_polar(m_prime, theta_prime);

    return r;
}
```

dft.c (generated – con't):

```
coal_num _cl_eq(coal_num a, coal_num b)
{
   return (a.re==b.re && a.im==b.im)?_cl_one:_cl_zero;
}

coal_num _cl_neq(coal_num a, coal_num b)
{
   return (a.re!=b.re || a.im!=b.im)?_cl_one:_cl_zero;
}

coal_num _cl_lt(coal_num a, coal_num b)
{
   return (a.re<b.re)?_cl_one:_cl_zero;
}

coal_num _cl_lte(coal_num a, coal_num b)
{
   return (a.re<=b.re)?_cl_one:_cl_zero;
}

coal_num _cl_gt(coal_num a, coal_num b)
{
   return (a.re>b.re)?_cl_one:_cl_zero;
}

coal_num _cl_gte(coal_num a, coal_num b)
{
   return (a.re>=b.re)?_cl_one:_cl_zero;
}

#define _cl_ge2(ca, cnidx) cl_get_elem(ca, (int)cnidx.re)

#define _cl_pe2(ca, cnidx, val) cl_put_elem(ca,(int)cnidx.re,val)

coal_arr _cl_new_arr(int N)
{
   coal_arr nca;
   if (N > (CL_STORE_SIZE - _cl_used)) longjmp(_cl_top,
CL_ERR_OUT_OF_MEMORY);
   nca.cp.kind = CL_ARR_CPX_INT;
   nca.cp.era = _cl_era;
   nca.cp.num = N;
   nca.cp.parr = _cl_arr_store + _cl_used;
   _cl_used += N;
   return nca;
}
```

dft.c (generated – con't):

```
coal_arr _cl_map(coal_num (*f)(coal_num), coal_arr ca)
{
   int idx;
   coal_arr nca;
   _cl_check_arr(ca);
   nca = _cl_new_arr(cl_len(ca));
   for (idx=0; idx<cl_len(ca); ++idx)
   {
       nca.cp.parr[idx] = f(cl_get_elem(ca, idx));
   }
   return nca;
}

coal_arr _cl_rng1(int start, int stop, int step)
{
   coal_arr nca;
   int diff;
   if (step == 0) longjmp(_cl_top, CL_ERR_STEP_IS_ZERO);
   if (  ((start > stop) && (step > 0))
       ||((start < stop) && (step < 0)) ) longjmp(_cl_top,
CL_ERR_STEP_WRONG_SIGN);
   nca.rng.kind = CL_ARR_RNG;
   nca.rng.num = ((stop - start)/step + 1);
   nca.rng.start = (short) start;
   nca.rng.step = (short) step;
   return nca;
}

#define _cl_rng2(start, stop, step) _cl_rng1((int)start.re,
(int)stop.re, (int)step.re)

coal_num _cl_reduce_num(coal_num (*f)(coal_num, coal_num), coal_num acc,
coal_arr ca)
{
    int idx;
    _cl_check_arr(ca);
    for (idx=0; idx<cl_len(ca); ++idx)
    {
        acc = f(acc, cl_get_elem(ca, idx));
    }
    return acc;
}

coal_arr _cl_reduce_arr(coal_arr (*f)(coal_arr, coal_num), coal_arr acc,
coal_arr ca)
{
    int idx;
    _cl_check_arr(ca);
    _cl_check_arr(acc);
    for (idx=0; idx<cl_len(ca); ++idx)
    {
        acc = f(acc, cl_get_elem(ca, idx));
    }
    return acc;
}
```

dft.c (generated – con't):

```
coal_num _cl_re(coal_num x)
{
   x.im = 0.0;
   return x;
}

coal_num _cl_im(coal_num x)
{
   x.re = x.im;
   x.im = 0.0;
   return x;
}

coal_num _cl_conj(coal_num x)
{
   x.im = -x.im;
   return x;
}

coal_num _cl_mag(coal_num x)
{
   x.re = _cl_mag_dbl(x);
   x.im = 0.0;
   return x;
}

coal_num _cl_phase(coal_num x)
{
   x.re = _cl_phase_dbl(x);
   x.im = 0.0;
   return x;
}

coal_num _cl_sin(coal_num x)
{
   x.re = sin(x.re);
   x.im = 0.0;
   return x;
}


coal_num _cl_cos(coal_num x)
{
   x.re = cos(x.re);
   x.im = 0.0;
   return x;
}

coal_num _cl_tan(coal_num x)
{
   x.re = tan(x.re);
   x.im = 0.0;
   return x;
}
```

dft.c (generated – con't):

```
coal_num _cl_sqrt(coal_num x)
{
   double m;
   double theta;
   coal_num r;
   m = sqrt(_cl_mag_dbl(x));
   theta = _cl_phase_dbl(x) / 2.0;
   r = _cl_polar(m, theta);
   return r;
}

coal_num _cl_exp(coal_num x)
{
   double exp_a;
   double b;
   coal_num r;
   if (_cl_is_zero(x)) return _cl_one;
   exp_a = exp(x.re);
   b = x.im;
   r = _cl_rect(exp_a*cos(b), exp_a*sin(b));
   return r;
}

coal_num _cl_distance(coal_num a, coal_num b)
{
   return _cl_mag(_cl_sub(a,b));
}

#define _cl_len(x) _cl_rect(cl_len(x),0.0)

#define _cl_last(x) _cl_rect(cl_last(x),0.0)

coal_num _cl_not(coal_num x)
{
   return (fabs(x.re)<.5?_cl_one:_cl_zero);
}

void cl_free()
{
   _cl_used = 0;
   _cl_era = 0x3FFF&(_cl_era+1);
}

int cl_last_err()
{
   return _cl_last_err;
}
```

dft.c (generated – con't):

```
int cl_valid_arr(coal_arr ca)
{
   if (ca.cp.parr == 0)
   {
       return 0;
   }
   else if ((ca.cp.kind==CL_ARR_CPX_INT) && (ca.cp.era!=_cl_era))
   {
       return 0;
   }
   else
   {
       return 1;
   }
}

coal_num cl_get_elem(coal_arr ca, int idx)
{
   coal_num _ret;
   ++_cl_call_lev;
   _cl_handle_err(_cl_invalid_num());
   _cl_check_idx(idx,ca);
   switch (ca.cp.kind)
   {
       case(CL_ARR_CPX_INT):
       case(CL_ARR_CPX_EXT):
         _ret = ca.cp.parr[idx];
         break;
       case(CL_ARR_RE_EXT):
         _ret = _cl_rect(ca.re.parr[idx], 0.0);
         break;
       case(CL_ARR_RNG):
         _ret = _cl_rect((ca.rng.start + idx * ca.rng.step), 0.0);
         break;
       default:
         longjmp(_cl_top, CL_ERR_BAD_ARRAY);
   }
   --_cl_call_lev;
   return _ret;
}
```

dft.c (generated – con't):

```
coal_num cl_put_elem(coal_arr ca, int idx, coal_num val)
{
   ++_cl_call_lev;
   _cl_handle_err(_cl_invalid_num());
   _cl_check_idx(idx,ca);
   switch (ca.cp.kind)
   {
       case(CL_ARR_CPX_INT):
       case(CL_ARR_CPX_EXT):
         ca.cp.parr[idx] = val;
         break;
       case(CL_ARR_RE_EXT):
         ca.re.parr[idx] = val.re;
         break;
       case(CL_ARR_RNG):
         longjmp(_cl_top, CL_ERR_ARRAY_IMMUTABLE);
         break;
       default:
         longjmp(_cl_top, CL_ERR_BAD_ARRAY);
   }
   --_cl_call_lev;
   return val;
}

coal_num cl_num(double re, double im)
{
    return _cl_rect(re, im);
}

coal_arr cl_real_arr(double* re, int size)
{
    coal_arr r;
    if ((re == 0)||(size<1)||(size>CL_MAX_ARR_SIZE))
    {
        _cl_last_err = CL_ERR_BAD_ARRAY;
        r.re.parr = 0;
    }
    else
    {
        r.re.kind = CL_ARR_RE_EXT;
        r.re.num = size;
        r.re.parr = re;
    }
    return r;
}
```

dft.c (generated – con't):

```
coal_arr cl_cplx_arr(double* re, int size)
{
    coal_arr r;
    if ((re == 0)||(size<2)||(size>(2*CL_MAX_ARR_SIZE)))
    {
        _cl_last_err = CL_ERR_BAD_ARRAY;
        r.cp.parr = 0;
    }
    else
    {
        r.cp.kind = CL_ARR_CPX_EXT;
        r.cp.era = 0;
        r.cp.num = size/2;
        r.cp.parr = (coal_num*)re;
    }
    return r;
}

static void* _links[5];

static coal_num _lambda2(coal_num n);
static coal_num _lambda1(coal_num s, coal_num n);

coal_num dft_one_bin(coal_num k, coal_arr x)
{
coal_num N;
coal_num _ret;
_links[0]=(void*)&x;
_links[1]=(void*)&k;
_links[2]=(void*)&N;
++_cl_call_lev;
_cl_handle_err(_cl_invalid_num());
_ret =
(N=_cl_len(x)
,
 _cl_reduce_num(_lambda1,
_cl_rect(0, 0.0),
_cl_rng2(_cl_rect(0, 0.0),
_cl_last(x)
,
_cl_rect(1, 0.0))
)
);
--_cl_call_lev;
return _ret;
}
```

dft.c (generated – con't):

```
coal_num distortion(coal_arr s, coal_num f1, coal_num f2)
{
coal_num harmonic;
coal_num fundamental;
coal_num _ret;
++_cl_call_lev;
_cl_handle_err(_cl_invalid_num());
_ret =
(fundamental=dft_one_bin(f1, s)
,
 harmonic=dft_one_bin(f2, s)
,
 _cl_div(_cl_mag(harmonic)
,
_cl_mag(fundamental)
)
);
--_cl_call_lev;
return _ret;
}

coal_arr make_two_tone(coal_num N, coal_num f1, coal_num a1, coal_num
f2, coal_num a2)
{
coal_arr _ret;
_links[0]=(void*)&a1;
_links[1]=(void*)&f1;
_links[2]=(void*)&N;
_links[3]=(void*)&a2;
_links[4]=(void*)&f2;
++_cl_call_lev;
_cl_handle_err(_cl_invalid_arr());
_ret =
_cl_map(_lambda2,
_cl_rng2(_cl_rect(0, 0.0),
_cl_sub(N,
_cl_rect(1, 0.0))
,
_cl_rect(1, 0.0))
)
;
--_cl_call_lev;
return _ret;
}

coal_num _lambda2(coal_num n)
{
coal_num _ret;
coal_num* pa1=(coal_num*)_links[0];
coal_num* pf1=(coal_num*)_links[1];
coal_num* pN=(coal_num*)_links[2];
coal_num* pa2=(coal_num*)_links[3];
coal_num* pf2=(coal_num*)_links[4];
_ret =
_cl_add(_cl_mul(*pa1,
```

dft.c (generated – con't):

```
_cl_sin(_cl_div(_cl_mul(_cl_mul(_cl_rect(6.2831853, 0.0),
*pf1)
,
n)
,
*pN)
)
)
,
_cl_mul(*pa2,
_cl_cos(_cl_div(_cl_mul(_cl_mul(_cl_rect(6.2831853, 0.0),
*pf2)
,
n)
,
*pN)
)
)
)
;
return _ret;
}

coal_num _lambda1(coal_num s, coal_num n)
{
coal_num _ret;
coal_arr* px=(coal_arr*)_links[0];
coal_num* pk=(coal_num*)_links[1];
coal_num* pN=(coal_num*)_links[2];
_ret =
_cl_add(s,
_cl_mul(_cl_ge2(*px,
n)
,
_cl_exp(_cl_div(_cl_mul(_cl_mul(_cl_negate(
_cl_rect(0.0, 6.283185))
,
*pk)
,
n)
,
*pN)
)
)
)
;
return _ret;
}
```

dft ex.c (C application that calls COAL functions):

```
#include <stdio.h>
#include <math.h>
#include "dft.h"

#define N 1000

/* This program demonstrates the complex arithmetic and array
features of COAL.  A signal is generated (once in C and once in COAL)
and its 2nd harmonic distortion calculated. */
int main()
{
    double signal[N];
    double f1;
    double f2;
    double w1;
    double w2;
    int i;
    coal_arr s1;
    coal_arr s2;
    coal_num res;

    /* create signal with 2nd harmonic distortion */
    f1 = 100.0;
    f2 = 200.0;
    for(i=0; i<N; ++i)
    {
        signal[i] = sin(2.0*M_PI*f1*i/N) + .2 * cos(2.0*M_PI*f2*i/N);
    }

    /* setup coal array to send as argument to COAL function */
    s1 = cl_real_arr(signal, N);

    /* do analysis */
    res = distortion(s1, cl_num(f1,0.0), cl_num(f2,0.0));
    if (!cl_valid_num(res))
    {
        printf("Error %d at distortion.\n", cl_last_err());
        return -1;
    }

    /* report */
    printf("2nd harmonic distortion is %f.\n", cl_dbl_re(res));


    /* create signal with 2nd harmonic distortion in COAL */
    s2 = make_two_tone(cl_num(N,0.0)
                    , cl_num(f1,0.0)
                    , cl_num(1.0,0.0)
                    , cl_num(f2,0.0)
                    , cl_num(.2,0.0)
                    );
```

```
    if (!cl_valid_arr(s2))
    {
        printf("Error %d at make_two_tone.\n", cl_last_err());
        return -1;
    }

    /* do analysis */
    res = distortion(s2, cl_num(f1,0.0), cl_num(f2,0.0));
    if (!cl_valid_num(res))
    {
        printf("Error %d at distortion.\n", cl_last_err());
        return -1;
    }

    /* report */
    printf("2nd harmonic distortion is %f.\n", cl_dbl_re(res));

    return 0;
}
```

6.1.2 DFT Example Output

When run from the command line, this example outputs the following:

```
$ ./dft_ex
2nd harmonic distortion is 0.200000.
2nd harmonic distortion is 0.200000.
```

6.2 Bandpass Filter Example

The following example outputs the magnitude and phase response of a simple RLC filter described by this network function:

```
H(s) = (R/L)*s / ((s^2 + (R/L)*s + (1/(L*C))))
```

Values were chosen for R, L, and C that puts the pass band of this filter at around 100kHz.

6.2.1 Bandpass Filter Listing

The files for this example, shown below, are summarized in this table:

File	Description
`filter.coal`	Where COAL source for example resides. Work done here to calculate mag and phase responses.
`filter.h`	Header file generated from COAL.
`filter.c`	C file generated from COAL – definitions for user functions are located towards the end of listing. Note that for brevity common generated code is not shown (see previous DFT example for full listing of common generated code).
`filter_ex.c`	Application that calls COAL functions in filter.c and prints out responses to standard out.

filter.coal:

```
bp_filter(R, L, C, freqs) ->

  (f ->
        s<-2PIi*f;

        # network func for simple bandpass filter
        (R/L)*s / ((s^2 + (R/L)*s + (1/(L*C))))

            ) {freqs}

mag_resp(resp) -> mag{resp}

phase_resp(resp) -> (c -> phase(c)*180.0/PI) {resp}

get_freqs(start, N, step) ->

       (i-> start + i*step){0..N-1}
```

filter.h (generated):

```
#ifndef _filter_h
#define _filter_h

#define CL_ERR_ILLEGAL_INDEX 0x1
#define CL_ERR_ARRAY_IMMUTABLE 0x2
#define CL_ERR_OUT_OF_MEMORY 0x3
#define CL_ERR_STEP_IS_ZERO 0x4
#define CL_ERR_STEP_WRONG_SIGN 0x5
#define CL_ERR_BAD_ARRAY 0x6

typedef struct
{
   double re;
   double im;
} coal_num;

typedef union
{
   struct
   {
       unsigned int kind: 2;
       unsigned int era: 14;
       unsigned int num: 16;
       coal_num* parr;
   } cp;
   struct
   {
       unsigned int kind: 2;
       unsigned int reserved: 14;
       unsigned int num: 16;
       double* parr;
   } re;
   struct
   {
       unsigned int kind: 2;
       unsigned int reserved: 14;
       unsigned int num: 16;
       short start;
       short step;
   } rng;
} coal_arr;

extern void cl_free();
extern int cl_last_err();
#define cl_valid_num(x) (!isnan(x.re))
#define cl_dbl_re(x) (x.re)
#define cl_dbl_im(x) (x.im)
extern int cl_valid_arr(coal_arr);
extern coal_num cl_get_elem(coal_arr, int);
extern coal_num cl_put_elem(coal_arr, int, coal_num);
#define cl_len(x) (x.cp.num)
#define cl_last(x) (x.cp.num-1)
```

```
extern coal_num cl_num(double re, double im);
extern coal_arr cl_real_arr(double* re, int size);
extern coal_arr cl_cplx_arr(double* re, int size);
extern coal_arr bp_filter(coal_num R, coal_num L, coal_num C, coal_arr
freqs);
extern coal_arr mag_resp(coal_arr resp);
extern coal_arr phase_resp(coal_arr resp);
extern coal_arr get_freqs(coal_num start, coal_num N, coal_num step);
#endif
```

filter.c (generated):

```
#include <math.h>
#include <setjmp.h>
#include "filter.h"
```

... snip ...

NOTE: Common generated code not shown for brevity. For full listing of common generated code see DFT Example Listing in section 6.1.1.

... snip ...

```
static void* _links[3];

static coal_num _lambda3(coal_num i);
static coal_num _lambda2(coal_num c);
static coal_num _lambda1(coal_num f);

coal_arr bp_filter(coal_num R, coal_num L, coal_num C, coal_arr freqs)
{
coal_arr _ret;
_links[0]=(void*)&R;
_links[1]=(void*)&L;
_links[2]=(void*)&C;
++_cl_call_lev;
_cl_handle_err(_cl_invalid_arr());
_ret =
_cl_map(_lambda1,
freqs)
;
--_cl_call_lev;
return _ret;
}
```

filter.c (generated – con't):

```
coal_arr mag_resp(coal_arr resp)
{
coal_arr _ret;
++_cl_call_lev;
_cl_handle_err(_cl_invalid_arr());
_ret =
_cl_map(_cl_mag,
resp)
;
--_cl_call_lev;
return _ret;
}

coal_arr phase_resp(coal_arr resp)
{
coal_arr _ret;
++_cl_call_lev;
_cl_handle_err(_cl_invalid_arr());
_ret =
_cl_map(_lambda2,
resp)
;
--_cl_call_lev;
return _ret;
}

coal_arr get_freqs(coal_num start, coal_num N, coal_num step)
{
coal_arr _ret;
_links[0]=(void*)&start;
_links[1]=(void*)&step;
++_cl_call_lev;
_cl_handle_err(_cl_invalid_arr());
_ret =
_cl_map(_lambda3,
_cl_rng2(_cl_rect(0, 0.0),
_cl_sub(N,
_cl_rect(1, 0.0))
,
_cl_rect(1, 0.0))
)
;
--_cl_call_lev;
return _ret;
}
```

filter.c (generated – con't):

```
coal_num _lambda3(coal_num i)
{
coal_num _ret;
coal_num* pstart=(coal_num*)_links[0];
coal_num* pstep=(coal_num*)_links[1];
_ret =
_cl_add(*pstart,
_cl_mul(i,
*pstep)
)
;
return _ret;
}

coal_num _lambda2(coal_num c)
{
coal_num _ret;
_ret =
_cl_div(_cl_mul(_cl_phase(c)
,
_cl_rect(180.0, 0.0))
,
_cl_rect(3.1415927, 0.0))
;
return _ret;
}

coal_num _lambda1(coal_num f)
{
coal_num s;
coal_num _ret;
coal_num* pR=(coal_num*)_links[0];
coal_num* pL=(coal_num*)_links[1];
coal_num* pC=(coal_num*)_links[2];
_ret =
(s=_cl_mul(_cl_rect(0.0, 6.283185),
f)
,
 _cl_div(_cl_mul(_cl_div(*pR,
*pL)
,
s)
,
_cl_add(_cl_add(_cl_expon(s,
_cl_rect(2, 0.0))
,
_cl_mul(_cl_div(*pR,
*pL)
,
s)
)
,
_cl_div(_cl_rect(1, 0.0),
_cl_mul(*pL,
*pC)
```

```
)
)
)
);
return _ret;
}
```

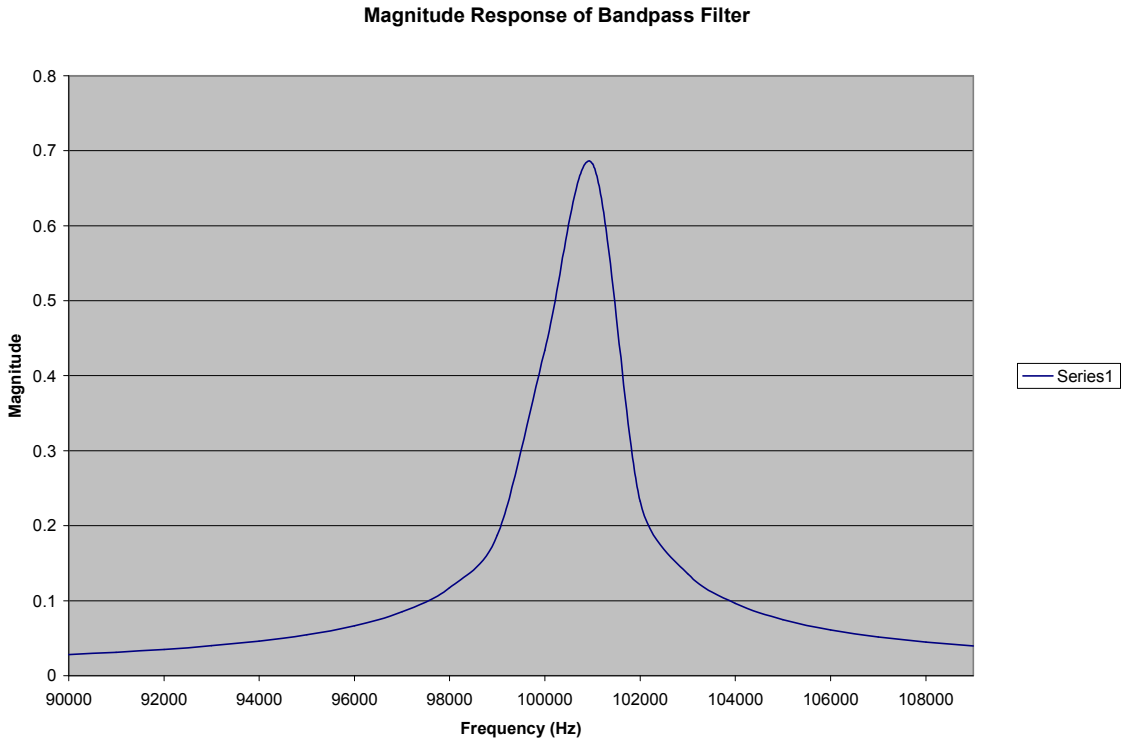
6.2.2 Bandpass Filter Output

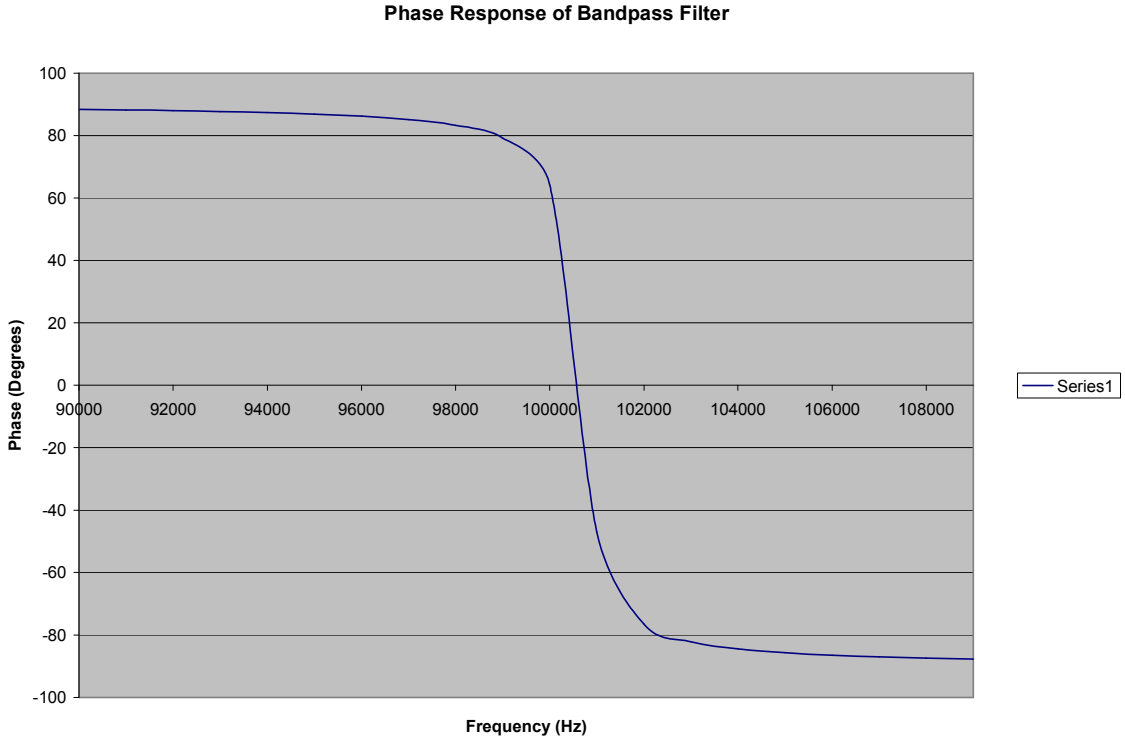
When run from the command line, this example outputs the following:

```
$ ./filter_ex
magnitude response
90000.000000 0.028184
91000.000000 0.031281
92000.000000 0.035089
93000.000000 0.039888
94000.000000 0.046121
95000.000000 0.054546
96000.000000 0.066563
97000.000000 0.085087
98000.000000 0.117318
99000.000000 0.186983
100000.000000 0.434087
101000.000000 0.682375
102000.000000 0.232303
103000.000000 0.136220
104000.000000 0.096362
105000.000000 0.074656
106000.000000 0.061017
107000.000000 0.051658
108000.000000 0.044838
109000.000000 0.039647

phase response
90000.000000 88.384967
91000.000000 88.207462
92000.000000 87.989124
93000.000000 87.713984
94000.000000 87.356511
95000.000000 86.873196
96000.000000 86.183379
97000.000000 85.118961
98000.000000 83.262657
99000.000000 79.223214
100000.000000 64.272812
101000.000000 -46.970489
102000.000000 -76.567276
103000.000000 -82.170834
104000.000000 -84.470262
105000.000000 -85.718565
106000.000000 -86.501784
107000.000000 -87.038898
108000.000000 -87.430134
109000.000000 -87.727826
```

The following figures show the magnitude and phase responses in graphical form.





6.3 Test Suites

The following test cases cover all of the features of COAL that are specified in the language reference manual (see 3). The tests are broken down by individual feature so as to isolate faults.

Automation of tests was achieved with Bourne shell scripting and make. make is responsible for building a .coal source files to c, and then from c to target test executables. The run_test.sh script executes the generated executable (or test script), compares output of the test to a gold file with diff, and reports a pass or fail.

Output for most of the tests using diff is generated by the –trace option on the COAL compiler. Trace mode dumps the value returned from functions as well as error conditions. For brevity one example of a trace output is given for illustration in 6.3.2.

In each of the sections below, the approach of the test is described along with a source code listing.

6.3.1 type_inference_test

This test exercises the scanner, parser, type inference, and semantic checking of COAL. type_inference.coal is passed to the coal executable with the –sast command line switch which dumps the semantically checked abstract syntax tree to standard out.

test/type inference.coal:

```
# Simple math
A(x) -> 1 + x

B(x,y) -> x - y

C(x,y) -> x * 1; 7i ^ y

# Infer over many assignments
D(x,y) ->
p <- x;
q <- p;
r <- q;
w <- r;
w / y

E(x,y) ->
p <- x;
q <- p;
r <- q;
```

```
w <- r;
y[w]; y

# Figure type of map argument
mapf1(a) -> a ^ 2.0
F(x) -> mapf1{x}

# Figure types of reduce arguments
reducef1(a,b) -> b + a
G(x,y,z) -> reducef1{y,z} / x

reducef2(a,b) -> a[b] * 2; a
H(x,y) -> reducef2{x,y}

# Range arguments
I(a,b,c) -> mapf1{a..b\c}

# Array put
J(p, q, v) -> p[q] <- v

# Conditionals
K(c, a, b) -> if c then a else b + 1
L(c, a, b) -> if c then 0..a else b

# Already defined method
M(p, q, r) -> L(p, q, r)

# Infer from built-ins
O(x) -> sin(x)
P(a) -> 0..len(a)
Q(r, s) -> distance(r,s)

# Infer within unnamed functions
R(x, y) ->
 (x, z -> z + len(x))(y, x)  # hole for x - x's have different types

S(w, u) ->
 z <- (q, p -> if q > p then w else u)(0,1);
 (w + z) / u

T(w, u) ->
 z <- (q, p -> if q > p then w else u)(0,1);
 len(z)

# Negation
U(a) -> -a
```

Below is an example of what the –sast command line switch will produce.

test/type inference test.sh.gold.txt:

```
Num A(Num x) ->
1 + x

Num B(Num x, Num y) ->
x - y

Num C(Num x, Num y) ->
x * 1;
7i ^ y

Num D(Num x, Num y) ->
Num p <- x;
Num q <- p;
Num r <- q;
Num w <- r;
w / y

NumArr E(Num x, NumArr y) ->
Num p <- x;
Num q <- p;
Num r <- q;
Num w <- r;
y[w];
y

Num mapf1(Num a) ->
a ^ 2.0

NumArr F(NumArr x) ->
mapf1{x}

Num reducef1(Num a, Num b) ->
b + a

Num G(Num x, Num y, NumArr z) ->
reducef1{y, z} / x

NumArr reducef2(NumArr a, Num b) ->
a[b] * 2;
a

NumArr H(NumArr x, NumArr y) ->
reducef2{x, y}

NumArr I(Num a, Num b, Num c) ->
mapf1{a..b\c}

Num J(NumArr p, Num q, Num v) ->
p[q]<-v

Num K(Num c, Num a, Num b) ->
```

```
if c then a else b + 1

NumArr L(Num c, Num a, NumArr b) ->
if c then 0..a\1 else b

NumArr M(Num p, Num q, NumArr r) ->
L(p, q, r)

Num O(Num x) ->
sin(x)

NumArr P(NumArr a) ->
0..len(a)\1

Num Q(Num r, Num s) ->
distance(r, s)

Num R(Num x, NumArr y) ->
(NumArr x, Num z -> z + len(x))(y, x)

Num S(Num w, Num u) ->
Num z <- (Num q, Num p -> if q > p then w else u)(0, 1);
w + z / u

Num T(NumArr w, NumArr u) ->
NumArr z <- (Num q, Num p -> if q > p then w else u)(0, 1);
len(z)

Num U(Num a) ->
-a
```

6.3.2 literals_test

Make sure that numbers and constants hold to spec.

test/literals.coal:

```
lit1() -> 15 - 27i
lit2() -> 15. - 27.i
lit3() -> 15.0 - 27.0i
lit4() -> 1.5e1 - 270.0e-1i
lit5() -> - 3 - 9i + 3i - 2
lit6() -> PI + 2PIi
lit7() -> 2PI + PIi

test() ->
    lit1();
    lit2();
    lit3();
    lit4();
    lit5();
    lit6();
    lit7();
    0
```

test/literals test.gold.txt:

(This trace output only shown once for illustration)

```
test begin
 lit1 begin
 lit1 end with
 (15.000000,-27.000000)
 lit2 begin
 lit2 end with
 (15.000000,-27.000000)
 lit3 begin
 lit3 end with
 (15.000000,-27.000000)
 lit4 begin
 lit4 end with
 (15.000000,-27.000000)
 lit5 begin
 lit5 end with
 (-5.000000,-6.000000)
 lit6 begin
 lit6 end with
 (3.141593,6.283185)
 lit7 begin
 lit7 end with
 (6.283185,3.141592)
test end with
(0.000000,0.000000)
```

6.3.3 math_test

Ensure basic math operators work and test basic precedence.

test/math.coal:

```
add(a,b) ->  a+b
sub(a,b) ->  a-b
mul(a,b) ->  a*b
div(a,b) ->  a/b
expon(a,b) ->  a^b
neg(a) -> -a
prec1() -> 1 + 2 * 3 ^ 2 - 4
prec2() -> 1i - 2i ^ 2 / 3i + 4i

test() ->
    add(5,9);
    add(-5i,-9i);
    add(-5,9i);
    sub(2,5);
    sub(-2i,-5i);
    sub(-2,5i);
    div(2,5);
    div(-2i,-5i);
    div(-2,5i);
    expon(2,10);
    expon(3,.5);
    expon(1i,2.0);
    neg(1+1i);
    neg(1-1i);
    neg(-1-1i);
    prec1();
    prec2();
    0
```

6.3.4 relop_test

Cycle through all variations of operands for all relational operators.

test/relop.coal:

```
lt(a, b)  -> a < b
lte(a, b) -> a <= b
gt(a, b)  -> a > b
gte(a, b) -> a >= b
eq(a, b)  -> a = b
neq(a, b) -> a <> b

test() ->
    lt(1,2);
    lt(2,1);
    lt(1,1);
    lte(1,2);
    lte(2,1);
    lte(1,1);
    gt(1,2);
    gt(2,1);
    gt(1,1);
    gte(1,2);
    gte(2,1);
    gte(1,1);
    eq(1,1);
    eq(1,2);
    neq(1,1);
    neq(1,2);
    0
```

6.3.5 assign_test

Try using results of assignments with in subsequent assignments.

test/assign.coal:

```
asn1() -> x<-5+9i; x

asn2() -> z<-5+9i; z<-z*2; z

asn3() ->
    a <- 1.23;
    (b <- a + 1; c <- b + 1);
    d <- c + 1;
    d

test() ->
  asn1();
  asn2();
  asn3();
  0
```

6.3.6 map_test

Test map with named function and lambda.

test/map.coal:

```
dbl(n)->2*n

map1() -> (p->2^p){0..8\2}
map2(in) -> (e->2*e){in}
map3(in) -> dbl{in}

test() ->
  r<-map1();
  q<-map2(r);
  map3(q);
  0
```

6.3.7 range_test

Try ranges going in both directions and try when step doesn't fit evenly into range. Also verify that imaginary numbers are ignored in range arguments.

test/range.coal:

```
range1() -> -9..10
range2() -> -9..10\2
range3() -> 10..-9\-1
range4() -> 10..-9\-2
range5(a,b) -> a..b
range6(a,b,c) -> a..b\c

test() ->
  range1();
  range2();
  range3();
  range4();
  range5(1.1-1i,3.5-1i);
  range6(100+1i,199+1i,10);
  range6(199+1i,100+1i,-10);
  0
```

6.3.8 reduce_test

Try reduce with a named function and a lambda function. Also try reduce with a number and with an array as the "accumulator".

test/reduce.coal:

```
sum(a,n)-> a+n*1i

red1() -> (a,n->a+n*1i){0,10..100\10}
red2() -> sum{0,10..100\10}

red3() ->
   arr<-(e->e){1..10};
   (arr,n->arr[n]<-arr[n]/2.0;arr){arr,0..last(arr)}

test() ->
  red1();
  red2();
  red3();
  0
```

6.3.9 array_test

Try getting element on both range array type (immutable) and dynamically generated array type. Try putting value in dynamically generated array type. Also, make sure that assignment works for arrays, too.

test/reduce.coal:

```
get(arr, n) -> arr[n]
set(arr, n, v) -> arr[n] <- v

test() ->
  get(0..4,3);
  get(z<-(e->e*1i){0..4},3);
  get(z,3);
  set(z,3,3);
  z
```

6.3.10 if_test

Ensure that imaginary values have no impact on condition. Make sure that if/then/else works with both number and array types in the then/else portion of the expression.

Even though they are the same, separate if1 and if2 definitions are needed because type inference does not allow re-assignment of types to arguments and return values, even if the functions are used in different contexts.

test/if.coal:

```
if1(c,a,b) ->
  if (c) then a else b

if2(c,a,b) ->
  if (c) then a else b

test() ->
  if1(.4-1i,10,-10);
  if1(.5-1i,10,-10);
  a<-1..3;
  b<--1..-3\-1;
  if2(-.4+20i,a,b);
  if2(-.5+20i,a,b);
  0
```

6.3.11 lambda_test

Try these cases:

- lambda only uses its own arguments
- lambda uses its arguments plus variables inherited from parent scope
- above two cases with nested lambdas

test/lambda.coal:

```
lam1() -> (a,b,c,d->a*(b+c)*d)(1,2,3,4) + 1

lam2(x) -> (z -> x + z)(x)

lam3(x,y,z) ->
(q -> x +
  (p -> y +
    (r -> z + r)(p)
                  )(q)
                    )(10)

test()->
  lam1();
  lam2(15);
  lam3(1,2,3); # 1 + (2 + (10 + 3)) = 16
  0
```

6.3.12 builtins_test

Test all built in functions against well known expected values. Don't let trace output actual values of functions. Instead compare results to expected values and fill array with 1's or 0's depending on result of comparison. This avoids floating point issues when running on different platforms and is easier to verify manually.

Make sure that `not()` function meets its threshold spec set forth in language reference manual.

test/builtins.coal:

```
eq(a,b) -> distance(a,b) < .0005

res() ->
  r <- (e->0){0..23};
  r[0] <- eq(re(3-4i),3);
  r[1] <- eq(im(3-4i),-4);
  r[2] <- eq(conj(-7+8i), -7-8i);
  r[3] <- eq(mag(.5+.5i), .707);
  r[4] <- eq(phase(.5+.5i)*180/PI, 45.0);
  r[5] <- eq(sin(PI/2 + 9i), 1.0); #imag ignored
  r[6] <- eq(cos(2PI - 9i), 1.0); #imag ignored
  r[7] <- eq(tan(PI/4 - 9i), 1.0); #imag ignored
  r[8] <- eq(sqrt(2), 1.414);
  r[9] <- eq(sqrt(2i), 1+1i);
  r[10] <- eq(exp(0), 1);
  r[11] <- eq(exp(1), 2.718);
  r[12] <- eq(exp(PIi/4), .707 + .707i);
  r[13] <- eq(distance(.5,.6), .1);
  r[14] <- eq(distance(.5+.5i,-.5-.5i), 1.414);
  r[15] <- eq(distance(.5+.5i,.5+.5i), 0.0);
  r[16] <- len(r);
  r[17] <- last(r);
  r[18] <- not(0);
  r[19] <- not(1);
  r[20] <- not(.4);
  r[21] <- not(.5);
  r[22] <- not(-.4);
  r[23] <- not(-.5);
  r

test() ->
  res();
  0
```

6.3.13 recursion_test

Try two examples of recursion to ensure basic functionality.

test/recursion.coal:

```
fact(n)-> if (n>0) then n * fact(n-1) else 1

gcd(a, b)->
if (distance(a, b)>.05)
   then if (a>b) then gcd(a - b, b)
                 else gcd(a, b - a)
   else a

test()->
  fact(0);
  fact(1);
  fact(3);
  gcd(5,7);
  gcd(27,18);
  0
```

6.3.14 cbindings_test_assert

This set of tests ensures the functionality of the C API used to access COAL functions. It also exercises error handling of all the COAL run-time errors.

This test does not use the same gold file methodology but instead uses asserts in the C harness code.

test/cbindings.coal:

```
rng_arr() -> 0..4
cplx_int_arr() -> (e->e*1i){0..4}

get(a,i) -> a[i]
set(a,i,v) -> a[i]<-v

illegal_index1(a) -> get(a,-1); 0

illegal_index2(a) -> get(a,len(a)); 0

illegal_index3(a) -> set(a,-1,0); 0

illegal_index4(a) -> set(a,len(a),0); 0

immutable(a) -> set(a,0,0); 0

get_rng(a,b,c) -> a..b\c

sum(a) -> (s,e->s+e){0,a}
```

test/cbindings test assert.c:

```
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include "cbindings.h"

/* This program demonstrates the C API to COAL */
int main()
{
    double ext_re[5];
    double ext_cp[10];
    int i;

    coal_num x;
    coal_arr rnga;
    coal_arr cia;
    coal_arr cia_out_of_mem;
    coal_arr cea;
    coal_arr rea;

    /* initialize external arrays */
    for (i=0; i<5; ++i)
    {
        ext_re[i] = (double)i;
        ext_cp[2*i] = (double)i;    /* real */
        ext_cp[2*i+1] = -(double)i; /* imag */
    }

    /* cl_num */
    x = cl_num(17.0, -93.0);

    /* cl_dbl_re, cl_dbl_im */
    assert(cl_dbl_re(x)==17.0);
    assert(cl_dbl_im(x)==-93.0);

    /* cl_real_arr */
    rea = cl_real_arr(ext_re, 5);  /* external real array */

    /* cl_cplx_arr */
    cea = cl_cplx_arr(ext_cp, 10); /* external complex array -
interleaved */

    rnga = rng_arr(); /* range array */
    cia = cplx_int_arr(); /* internally generated array */

    /* cl_len */
    assert(5==cl_len(rea));
    assert(5==cl_len(cea));
    assert(5==cl_len(rnga));
    assert(5==cl_len(cia));

    /* cl_last */
    assert(4==cl_last(rea));
    assert(4==cl_last(cea));
    assert(4==cl_last(rnga));
```

test/cbindings test assert.c (con't):

```
    assert(4==cl_last(cia));

    /* cl_put_elem & cl_get_elem */
    cl_put_elem(rea, 3, cl_num(89.0, -89.0));
    x = cl_get_elem(rea, 3);
    assert(89.0==cl_dbl_re(x));
    assert(0.0==cl_dbl_im(x));

    cl_put_elem(cea, 3, cl_num(27.0, -27.0));
    x = cl_get_elem(cea, 3);
    assert(27.0==cl_dbl_re(x));
    assert(-27.0==cl_dbl_im(x));

    x = cl_get_elem(rnga, 3);
    assert(3.0==cl_dbl_re(x));
    assert(0.0==cl_dbl_im(x));

    cl_put_elem(cia, 3, cl_num(21.0, -21.0));
    x = cl_get_elem(cia, 3);
    assert(21.0==cl_dbl_re(x));
    assert(-21.0==cl_dbl_im(x));

    /* CL_ERR_ILLEGAL_INDEX - range array */
    assert(!cl_valid_num(illegal_index1(rnga)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);
    assert(!cl_valid_num(illegal_index2(rnga)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);

    /* CL_ERR_ILLEGAL_INDEX - complex internal array */
    assert(!cl_valid_num(illegal_index1(cia)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);
    assert(!cl_valid_num(illegal_index2(cia)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);
    assert(!cl_valid_num(illegal_index3(cia)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);
    assert(!cl_valid_num(illegal_index4(cia)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);

    /* CL_ERR_ILLEGAL_INDEX - complex external array */
    assert(!cl_valid_num(illegal_index1(cea)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);
    assert(!cl_valid_num(illegal_index2(cea)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);
    assert(!cl_valid_num(illegal_index3(cea)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);
    assert(!cl_valid_num(illegal_index4(cea)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);

    /* CL_ERR_ILLEGAL_INDEX - real external array */
    assert(!cl_valid_num(illegal_index1(rea)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);
    assert(!cl_valid_num(illegal_index2(rea)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);
    assert(!cl_valid_num(illegal_index3(rea)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);
```

test/cbindings test assert.c (con't):

```
    assert(!cl_valid_num(illegal_index4(rea)));
    assert(cl_last_err()==CL_ERR_ILLEGAL_INDEX);

    /* CL_ERR_ARRAY_IMMUTABLE */
    assert(!cl_valid_num(immutable(rnga)));
    assert(cl_last_err()==CL_ERR_ARRAY_IMMUTABLE);

    /* CL_ERR_OUT_OF_MEMORY - storesize set to 9 */
    cia_out_of_mem = cplx_int_arr();
    assert(!cl_valid_arr(cia_out_of_mem));
    assert(cl_last_err()==CL_ERR_OUT_OF_MEMORY);

    /* CL_ERR_STEP_IS_ZERO */
    assert(!cl_valid_arr(
        get_rng(cl_num(1.0,0.0)
              , cl_num(5.0,0.0)
              , cl_num(0.0,0.0))
        ));
    assert(cl_last_err()==CL_ERR_STEP_IS_ZERO);

    /* CL_ERR_STEP_WRONG_SIGN */
    assert(!cl_valid_arr(
        get_rng(cl_num(5.0,0.0)
              , cl_num(1.0,0.0)
              , cl_num(1.0,0.0))
        ));
    assert(cl_last_err()==CL_ERR_STEP_WRONG_SIGN);

    assert(!cl_valid_arr(
        get_rng(cl_num(1.0,0.0)
              , cl_num(5.0,0.0)
              , cl_num(-1.0,0.0))
        ));
    assert(cl_last_err()==CL_ERR_STEP_WRONG_SIGN);


    /* CL_ERR_BAD_ARRAY */
    cl_free();

    /* since we've free'd, cia should be invalid now as it was created
during a
              different era */
    assert(!cl_valid_arr(cia));

    /* trying a reduce operation should cause an error */
    assert(!cl_valid_num(sum(cia)));
    assert(cl_last_err()==CL_ERR_BAD_ARRAY);

    /* reallocate */
    cia = cplx_int_arr();
    assert(cl_last_err()==0);
    assert(cl_valid_arr(cia));

    /* free memory again and repeat test... */
    cl_free();
```

test/cbindings test assert.c (con't):

```
    assert(!cl_valid_arr(cia));

    /* again, trying a reduce operation should cause an error */
    assert(!cl_valid_num(sum(cia)));
    assert(cl_last_err()==CL_ERR_BAD_ARRAY);

    /* reallocate */
    cia = cplx_int_arr();
    assert(cl_valid_arr(cia));

    cl_free();
    return 0;
}
```

6.3.15 modules_test_assert

This set of tests ensures that multiple modules can be fed into the COAL compiler.

This test does not use the same gold file methodology but instead uses asserts in the C harness code.

test/module a.coal:

```
func_a(v) -> v + 1
```

test/module b.coal:

```
func_b(q) -> func_a(q) + 1
```

test/module c.coal:

```
func_c(r) -> func_b(r) + 1
```

test/module test_assert.coal:

```
#include <stdio.h>
#include <assert.h>
#include "modules.h"

int main()
{
    coal_num res;

    res = func_a(cl_num(1.0, 0.0));
    assert(cl_valid_num(res));
    assert(((int)cl_dbl_re(res))==2);

    res = func_b(cl_num(1.0, 0.0));
    assert(cl_valid_num(res));
    assert(((int)cl_dbl_re(res))==3);

    res = func_c(cl_num(1.0, 0.0));
    assert(cl_valid_num(res));
    assert(((int)cl_dbl_re(res))==4);

    return 0;
}
```

7 Lessons Learned

In OCaml, there seems to be a higher correlation between a program compiling and its correctness than with imperative languages.

Developing even a remotely useful language requires discipline and thoughtfulness.

Seemingly small decisions made at the beginning of a language project can have significant impact later during development.

Writing the Language Reference Manual up front was critical for staying on a clear development path and keeping project scope consistent.

When planning code modules it's helpful to decide on a general debug strategy up front so that isolating problems is relatively painless. This can be faster than just depending on a step through debugger in the long run.

8 Appendix

8.1 ast.mli

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

type op = Add | Sub | Mul | Div | Pow | Eq | Neq | Lt | Lte | Gt | Gte

type expr =

  (* returns Num *)
    Real of string * bool
  | Imag of string
  | Id of string
  | Negate of expr
  | Binop of expr * op * expr
  | Assign of string * expr
  | GetElem of expr * expr
  | PutElem of expr * expr * expr

  (* returns NumArr *)
  | Map of callee * expr
  | Range of expr * expr * expr

  (* returns Num or NumArr *)
  | Invoke of callee * expr list
  | Reduce of callee * expr * expr
  | IfThenElse of expr * expr * expr
  | Sequence of expr * expr

and callee =
  Named of string
| Lambda of func_def

and func_def = {
    fname : string;
    fargs : string list;
    fbody : expr;
  }

type program = func_def list
```

8.2 sast.mli

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

type expr =

  (* returns Num *)
    Real of string * bool
  | Imag of string
  | Id of string
  | Negate of typed_expr
  | Binop of typed_expr * Ast.op * typed_expr
  | Assign of string * typed_expr
  | GetElem of typed_expr * typed_expr
  | PutElem of typed_expr * typed_expr * typed_expr

  (* returns NumArr *)
  | Map of callee * typed_expr
  | Range of typed_expr * typed_expr * typed_expr

  (* returns Num or NumArr *)
  | Invoke of callee * typed_expr list
  | Reduce of callee * typed_expr * typed_expr
  | IfThenElse of typed_expr * typed_expr * typed_expr
  | Sequence of typed_expr * typed_expr

and
typed_expr = expr * Types.typ

and callee =
  Named of string
| Lambda of func_def

and func_def =
{
    fname : string;
    fargs : string list;
    fbody : typed_expr;
    flocals : Sym.symbol_table
}

type program = func_def list
```

8.3 scanner.mll

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

{ open Parser
  open Lexing }

let digit = ['0'-'9']
let sign = ('+'|'-')*
let expon = 'e' sign digit+
let integer = digit+
let float = digit+ expon
          | digit+ '.' digit* expon?
              | digit* '.' digit+ expon?
let whitespace = [' ' '\t' '\r' '\n']

rule token = parse

whitespace { token lexbuf }

| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '^' { EXPON }

| '>'   { GT }
| '<'   { LT }
| ">="  { GTE }
| "<="  { LTE }
| '='   { EQL }
| "<>"  { NEQL }

| ','   { COMMA }
| ';'   { SEMI }
| "<-"  { LARR }
| "->"  { RARR }


| '('   { LPAREN }
| ')'   { RPAREN }

| '{'   { LCURLY }
| '}'   { RCURLY }

| '['   { LSQUARE }
| ']'   { RSQUARE }

| '\\'   { BACKSLASH }

| "if"      { IF }
| "then"    { THEN }
| "else"    { ELSE }

| integer as lit { INT(lit) }
| float as lit { REAL(lit) }
```

(scannel.mll con’t)

```
| (integer|float) 'i' as lit { IMAG(lit) }

| ".." { RANGE }

(* Special case to disambiguate a floating point number
   and an integer right before the range operator.  In
   latter case, extract integer lexeme and back up two
   characters so that range operator is scanned as usual. *)
| integer as lit ".." {
      lexbuf.lex_curr_pos <- lexbuf.lex_curr_pos - 2;
      INT(lit)
      }

(* Constants *)
| "PI" { REAL("3.1415927") }
| "PIi" { IMAG("3.1415927") }
| "2PI" { REAL("6.2831853") }
| "2PIi" { IMAG("6.2831853") }

(* ID's *)
| ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']* as name { ID(name) }

(* Comments *)
| '#'[^ '\n' '\r']* { token lexbuf }

| eof { EOF }
```

8.4 parser.mly

```
/* COMS W4115, COAL, Eliot Scull, CUID: C000056091 */

%{
open Ast

(* enumerate lambda functions internally in order of occurence *)
let lambda_num = ref 0;;
let next_lambda_num () = (lambda_num:=!lambda_num + 1); !lambda_num;;

%}

%token LPAREN RPAREN LCURLY RCURLY LSQUARE RSQUARE COMMA SEMI BACKSLASH
%token PLUS MINUS TIMES DIVIDE EXPON
%token EQL NEQL LT LTE GT GTE
%token LARR RARR
%token IF THEN ELSE
%token RANGE
%token <string> INT
%token <string> REAL
%token <string> IMAG
%token <string> ID
%token COMMENT EOF

%left COMMA SEMI
%right LARR
%nonassoc ELSE
%nonassoc RANGE
%nonassoc BACKSLASH
%nonassoc RARR
%left EQL NEQL
%left LT LTE GT GTE
%left PLUS MINUS
%left TIMES DIVIDE
%left EXPON
%right negate_op
%nonassoc LCURLY RCURLY LSQUARE RSQUARE

%start program
%type <Ast.program> program

%%

program:
   /* nothing */   { [] }
| program func_def { $2 :: $1 }

expr:

/* Returns Number */
  ID    { Id  ($1) }
| INT   { Real($1, true) }
| REAL  { Real($1, false) }
| IMAG  { Imag(String.sub $1 0 ((String.length $1)-1)) }
| MINUS expr %prec negate_op { Negate($2) }
```

(parser.mly con't)

```
| expr PLUS   expr { Binop($1, Add, $3) }
| expr MINUS  expr { Binop($1, Sub, $3) }
| expr TIMES  expr { Binop($1, Mul, $3) }
| expr DIVIDE expr { Binop($1, Div, $3) }
| expr LT     expr { Binop($1, Lt , $3) }
| expr LTE    expr { Binop($1, Lte, $3) }
| expr GT     expr { Binop($1, Gt , $3) }
| expr GTE    expr { Binop($1, Gte, $3) }
| expr EQL    expr { Binop($1, Eq , $3) }
| expr NEQL   expr { Binop($1, Neq, $3) }
| expr EXPON  expr { Binop($1, Pow, $3) }
| expr LSQUARE expr RSQUARE { GetElem($1, $3) }
| expr LSQUARE expr RSQUARE LARR expr { PutElem($1, $3, $6) }
| ID LARR expr { Assign($1, $3) }

/* Returns array of Number */
| ID LCURLY expr RCURLY     { Map(Named($1), $3) }
| lambda LCURLY expr RCURLY { Map(Lambda($1), $3) }
| ID LCURLY expr COMMA expr RCURLY     { Reduce(Named($1), $3, $5) }
| lambda LCURLY expr COMMA expr RCURLY { Reduce(Lambda($1), $3, $5) }
| expr RANGE expr                { Range($1, $3, Real("1",true)) }
| expr RANGE expr BACKSLASH expr { Range($1, $3, $5) }

/* Returns Number or array of Number */
| LPAREN expr RPAREN { $2 }
| ID LPAREN invoke_arg_opt RPAREN    { Invoke(Named($1), List.rev $3) }
| lambda LPAREN invoke_arg_list RPAREN { Invoke(Lambda($1), List.rev
$3) }
| expr SEMI expr { Sequence($1, $3) }
| IF expr THEN expr ELSE expr { IfThenElse($2, $4, $6) }

invoke_arg_opt:
  /* nothing */    { [] }
| invoke_arg_list  { $1 }

invoke_arg_list:
  expr { [$1] }
| invoke_arg_list COMMA expr { $3 :: $1 }

func_arg_opt:
  /* nothing */  { [] }
| func_arg_list  { $1 }

func_arg_list:
  ID { [$1] }
| func_arg_list COMMA ID { $3 :: $1 }

lambda:
  LPAREN func_arg_list RARR expr RPAREN { { fname = "_lambda" ^
(string_of_int (next_lambda_num ()));
                                          fargs = List.rev $2;
                                          fbody = $4; } }
```

(parser.mly con't)

```
func_def:
  ID LPAREN func_arg_opt RPAREN RARR expr { { fname = $1;
                                              fargs = List.rev $3;
                                              fbody = $6; } }
```

8.5 types.ml

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

type typ =
  Num
| NumArr
| Func of typ * typ list
| Tbd
| Var of typ ref

let fresh () = Var(ref Tbd);;

(* strip off any Var's *)
let rec baretyp t =
  match t with
    Var({contents = inner}) -> baretyp inner
  | _ -> t
;;

let rec string_of_type = function
    Num -> "Num"
  | NumArr -> "NumArr"
  | Func(rt, ats) -> "Func(" ^ string_of_type rt ^ ", " ^ String.concat
", " (List.map string_of_type ats) ^ ")"
  | Tbd -> "?"
  | Var({contents = inner}) -> "Var(" ^ (string_of_type inner) ^ ")"
;;

(* strip off any Var's *)
let rec string_of_type_clean = function
    Num -> "Num"
  | NumArr -> "NumArr"
  | Func(rt, ats) -> "Func(" ^ string_of_type rt ^ ", " ^ String.concat
", " (List.map string_of_type ats) ^ ")"
  | Tbd -> "?"
  | Var({contents = inner}) -> string_of_type_clean inner
;;
```

8.6 sym.ml

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

open Types

(* type checking exceptions *)
exception Symbol_not_found of string
exception Symbol_redefinition of string * typ
exception Type_mismatch of typ * typ

(* symbol table *)
type symbol_table = {
   parent: symbol_table option;
   context: string;
   mutable scope: symbol list
}
and symbol = {
    sname: string;
    st: typ
}

(* add symbol to current environment *)
let addsym env n t =
    if (not (List.exists (fun sy -> sy.sname=n) env.scope))
        then (env.scope <- {sname=n; st=t} :: env.scope; t)
        else raise (Symbol_redefinition(n, t));;

let rec find_env_typ env n =
    try
        (* look in current scope *)
        env, (List.find (fun sy -> sy.sname=n) env.scope).st
    with Not_found -> match env.parent with
                        (* in top scope - symbol not found *)
                        None -> raise (Symbol_not_found(n))
                        (* search parent scope *)
                      | Some(parent_env) -> find_env_typ parent_env n;;

(* find a type for a given symbol - search current and then parent
environments *)
let rec findtyp env n = snd (find_env_typ env n)

(* find a environment for a given symbol - search current and then
parent environments *)
let rec findenv env n = fst (find_env_typ env n)

let symbol_names env = List.map (fun symbol -> symbol.sname) env.scope

let dump env =
    List.iter (fun symbol -> Printf.printf "{sname=%s, st=%s}\n"
symbol.sname (string_of_type symbol.st)) env.scope;;

(* create a new subordinate scope - keep reference to parent scope *)
let newscope c env =
    {parent=Some(env); context=c; scope=[]};;
```

8.7 semantics.mli

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

val check : Ast.func_def list -> Sast.func_def list
val addbuiltin : string -> Types.typ -> unit
```

8.8 semantics.ml

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

open Types
open Sym
open Printf

let debug = false

(* global environment that sticks around for type checking of all modules *)
let genv = ref {parent=None; context="*GLOBAL*"; scope=[]}

let addbuiltin name t =
    ignore(addsym (!genv) name t)


(* Based on Algorithm 6.16, "Type inference for polymorphic functions", page 393,
    and Algorithm 6.19 "Unification of a pair of nodes in a type graph", page 397, from
    Golden Dragon Book.  Instead of returning true or false, return unit type and
    throw exception on failure. *)
let rec unify s t =

    (* This method is used to check for cycles before one type variable
             is assigned to another.   This is important for when recursion is
             used in COAL.  Use of this method was prompted by cycles discovered
             during the semantic check of the gcd function in the recursion tests. *)
    let rec contains a b =
      if debug then printf "does %s contain %s?\n" (string_of_type a) (string_of_type b);
      let res =
        (a==b) ||
        match a with
          Var({contents=inner_a}) -> contains inner_a b
        | _ -> (a==b)
      in
      if debug then printf "%s.\n" (if res then "yes" else "no");
      res

    in
    if debug then printf "unify %s %s\n" (string_of_type s) (string_of_type t);
    match s, t with

      (* physically the same type node or one contains the other *)
      _, _ when (s==t) -> ()

      (* same basic type*)
    | Types.Num, Types.Num
    | Types.NumArr, Types.NumArr -> ()

      (* function "op-node" *)
    | Types.Func(rt1, at1), Types.Func(rt2, at2) ->
        (* the union of these Func type operators will happen implicitly through the
union
                        of their children *)

        (* unify argument lists *)
        (
          try
            List.iter2 unify at1 at2
```

(semantics.ml con't)

```
          with Invalid_argument(_) -> raise (Type_mismatch(Types.Func(rt1, at1),
Types.Func(rt2, at2)))
        );

        (* unify return types *)
        unify rt1 rt2;

      (* s or t represents a variable *)
    | Types.Var({contents=Types.Tbd} as var_s), _ ->
        (* union where s becomes t *)
        if (not (contains t s)) then var_s := t

    | _, Types.Var({contents=Types.Tbd} as var_t) ->
        (* union where t becomes s *)
        if (not (contains s t)) then var_t := s


      (* s or t has been defined.  Simply strip off Var and unify. *)
    | Types.Var({contents=inner_s}), _ -> unify inner_s t
    | _, Types.Var({contents=inner_t}) -> unify s inner_t

      (* cannot unify *)
    | _, _ ->
    raise (Type_mismatch(s, t))


(* check expressions *)
let rec check_exp env = function

  (* leaf expression nodes *)
    Ast.Real(fps, isint) ->
    if debug then printf "%s\n" fps;
    Sast.Real(fps, isint), Types.Num

  | Ast.Imag(fps) ->
    if debug then printf "%s\n" fps;
    Sast.Imag(fps), Types.Num

  | Ast.Id(name) ->
        if debug then printf "Id(%s)\n" name;
    Sast.Id(name), findtyp env name

  | Ast.Negate(e) ->
    if debug then printf "Negate\n";
    let se = check_exp env e in
    unify Types.Num (snd se);
    Sast.Negate(se), Types.Num

  (* a + b *)
  | Ast.Binop(e1, op, e2) ->
    if debug then printf "Binop\n";
    let se1 = check_exp env e1
    and se2 = check_exp env e2 in
    unify Types.Num (snd se1);
    unify Types.Num (snd se2);
    Sast.Binop(se1, op, se2), Types.Num

  (* a <- 1 *)
  | Ast.Assign(id, e) ->
    if debug then printf "Assign\n";
    (* be sure to check rhs first so we can check for illegal usages of *)
    (* assigned variable before it is assigned *)
    let se = check_exp env e
    and idt =
        (* if this symbol hasn't been defined yet, them add it to environment *)
        try
            findtyp env id
        with Symbol_not_found(_) -> addsym env id (Types.fresh ())
    in
```

(semantics.ml con't)

```
    unify idt (snd se);
    Sast.Assign(id, se), (snd se)

  (* a[1] *)
  | Ast.GetElem(e1, e2) ->
    if debug then printf "GetElem\n";
    let se1 = check_exp env e1
    and se2 = check_exp env e2 in
    unify Types.NumArr (snd se1);
    unify Types.Num (snd se2);
    Sast.GetElem(se1, se2), Types.Num

  (* a[1] <- 5 *)
  | Ast.PutElem(e1, e2, e3) ->
    if debug then printf "PutElem\n";
    let se1 = check_exp env e1
    and se2 = check_exp env e2
    and se3 = check_exp env e3 in
    unify Types.NumArr (snd se1);
    unify Types.Num (snd se2);
    unify Types.Num (snd se3);
    Sast.PutElem(se1, se2, se3), Types.Num

  (* f{a} *)
  | Ast.Map(a_callee, e) ->
    if debug then printf "Map\n";
    let (id, s_callee) = handle_lambda env a_callee in
    let idt = findtyp env id in
    let se = check_exp env e in
    unify (Types.Func(Types.Num, [Types.Num])) idt;
    unify Types.NumArr (snd se);
    Sast.Map(s_callee, se), Types.NumArr

  (* a..b\c *)
  | Ast.Range(e1, e2, e3) ->
    if debug then printf "Range\n";
    let se1 = check_exp env e1
    and se2 = check_exp env e2
    and se3 = check_exp env e3 in
    unify Types.Num (snd se1);
    unify Types.Num (snd se2);
    unify Types.Num (snd se3);
    Sast.Range(se1, se2, se3), Types.NumArr

  (* f(1,2) *)
  | Ast.Invoke(a_callee, actual_elist) ->
    if debug then printf "Invoke\n";
    let (id, s_callee) = handle_lambda env a_callee in
    (* check actual argument expressions *)
    let actual_selist = List.map (fun e -> check_exp env e) actual_elist in
    let rt = (Types.fresh ()) in
    let invoke_ft = Types.Func(rt, List.map (fun se -> snd se) actual_selist) in
    (* retrieve formal argument types and compare with actuals *)
    let def_ft = findtyp env id in
    unify def_ft invoke_ft;
    Sast.Invoke(s_callee, actual_selist), rt

  (* f{0, a} *)
  | Ast.Reduce(a_callee, e1, e2) ->
    if debug then printf "Reduce\n";
    let (id, s_callee) = handle_lambda env a_callee in
    let idt = findtyp env id
    and accum_t = (Types.fresh ())
    and se1 = check_exp env e1
    and se2 = check_exp env e2 in
    unify (Types.Func(accum_t, [accum_t; Types.Num])) idt;
    unify accum_t (snd se1);
    unify Types.NumArr (snd se2);
    Sast.Reduce(s_callee, se1, se2), accum_t
```

(semantics.ml con't)

```
  (* if a then b else c *)
  | Ast.IfThenElse(e1, e2, e3) ->
    if debug then printf "IfThenElse\n";
    let se1 = check_exp env e1
    and se2 = check_exp env e2
    and se3 = check_exp env e3 in
    if debug then printf "--unify condition\n";
    unify Types.Num (snd se1);
    if debug then printf "--unify condition done\n";
    if debug then printf "--unify then/else\n";
    unify (snd se2) (snd se3);
    if debug then printf "--unify then/else done\n";
    Sast.IfThenElse(se1, se2, se3), (snd se3)

  (* a;b;c;d... *)
  | Ast.Sequence(e1, e2) ->
    if debug then printf "Sequence\n";
    let se1 = check_exp env e1
    and se2 = check_exp env e2 in
    Sast.Sequence(se1, se2), (snd se2)

(* check function definition and recurse down body *)
and check_func_def env ast_fdef =
       if debug then printf "Func %s\n" ast_fdef.Ast.fname;
    (* function argument name/type pairs *)
    let n_t_pairs = List.fold_left (fun pairs arg -> (arg, (Types.fresh ())) :: pairs) []
ast_fdef.Ast.fargs in
    let n_t_pairs = List.rev n_t_pairs in
    let rt = (Types.fresh ()) in
    let ft = Types.Func(rt, List.map (fun p -> snd p) n_t_pairs) in

    (* store function type in global sym table so that we may use it to unify function
invocations across modules *)
    let _ = addsym env ast_fdef.Ast.fname ft in

    (* store arguments in local sym table *)
    let env = newscope ast_fdef.Ast.fname env in
    let _ = List.iter (fun (n, t) -> ignore(addsym env n t)) n_t_pairs in

    (* type check body of function definition *)
    let sast_body = check_exp env ast_fdef.Ast.fbody in

    (* unify return type with that of type of body expression *)
    unify rt (snd sast_body);

    {Sast.fname = ast_fdef.Ast.fname;
     Sast.fargs = ast_fdef.Ast.fargs;
     Sast.fbody = sast_body;
     Sast.flocals = env }

and handle_lambda env = function
    (* not a lambda *)
    Ast.Named(fn) -> (fn, Sast.Named(fn))

    (* establish definition for lambda *)
  | Ast.Lambda(ast_fdef) ->
        let sast_fdef = check_func_def env ast_fdef in
        (sast_fdef.Sast.fname, Sast.Lambda(sast_fdef));;

(* generate list of sast function definitions from ast function definitions - *)
(* this is the entry point for module *)
let rec check ast_fdefs =
try
   List.map (check_func_def (!genv)) (List.rev ast_fdefs)
with e -> if debug then dump (!genv); raise e
```

8.9 cbackend.mli

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

val emit : Coalopts.compile_opts    (* options *)
           -> string list           (* builtins *)
           -> Sast.func_def list    (* program definition *)
           -> unit
```

8.10 cbackend.ml

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

open Coalopts
open Types
open Sym
open Sast
open Printf

let debug = false

let rec to_c_type = function
    Num -> "coal_num"
  | NumArr -> "coal_arr"
  | Tbd -> "coal_num"
  | Func(_,_) -> "/*Func not supported*/"
  | Var({contents = inner}) -> to_c_type inner;;

let to_c_var_decl env id =
    let t = findtyp env id in
    (to_c_type t) ^ " " ^ id;;

let to_c_proto fdef =
    let beg = sprintf "%s %s(" (to_c_type (snd fdef.fbody)) fdef.fname in
    let args = List.map (to_c_var_decl fdef.flocals) fdef.fargs in
    beg ^ (String.concat ", " args) ^ ")";;

let string_of_op = function
    Ast.Add -> "_cl_add"
  | Ast.Sub -> "_cl_sub"
  | Ast.Mul -> "_cl_mul"
  | Ast.Div -> "_cl_div"
  | Ast.Pow -> "_cl_expon"
  | Ast.Eq  -> "_cl_eq"
  | Ast.Neq -> "_cl_neq"
  | Ast.Lt  -> "_cl_lt"
  | Ast.Lte -> "_cl_lte"
  | Ast.Gt  -> "_cl_gt"
  | Ast.Gte -> "_cl_gte";;

let emit_header_beg copts doth =
    (* idempotency *)
    let ppdef = sprintf "_%s_h" copts.outfileroot in
    fprintf doth "#ifndef %s\n" ppdef;
    fprintf doth "#define %s\n\n" ppdef;;

let emit_header_end doth =
    fprintf doth "#endif\n";;

let emit_internals copts doth dotc =

    (* .h *)
    fprintf doth "#define CL_ERR_ILLEGAL_INDEX 0x1\n";
    fprintf doth "#define CL_ERR_ARRAY_IMMUTABLE 0x2\n";
    fprintf doth "#define CL_ERR_OUT_OF_MEMORY 0x3\n";
    fprintf doth "#define CL_ERR_STEP_IS_ZERO 0x4\n";
    fprintf doth "#define CL_ERR_STEP_WRONG_SIGN 0x5\n";
```

(cbackend.ml con't)

```
    fprintf doth "#define CL_ERR_BAD_ARRAY 0x6\n\n";
    fprintf doth "typedef struct\n\
                  {\n\
                  \   double re;\n\
                  \   double im;\n\
                  } coal_num;\n\n" ;
    fprintf doth "typedef union \n\
                  {\n\
                  \   struct\n\
                  \   {\n\
                  \       unsigned int kind: 2;\n\
                  \       unsigned int era: 14;\n\
                  \       unsigned int num: 16;\n\
                  \       coal_num* parr;\n\
                  \   } cp;\n\
                  \   struct\n\
                  \   {\n\
                  \       unsigned int kind: 2;\n\
                  \       unsigned int reserved: 14;\n\
                  \       unsigned int num: 16;\n\
                  \       double* parr;\n\
                  \   } re;\n\
                  \   struct\n\
                  \   {\n\
                  \       unsigned int kind: 2;\n\
                  \       unsigned int reserved: 14;\n\
                  \       unsigned int num: 16;\n\
                  \       short start;\n\
                  \       short step;\n\
                  \   } rng;\n\
                  } coal_arr;\n\n" ;
    fprintf doth "extern void cl_free();\n";
    fprintf doth "extern int cl_last_err();\n";
    fprintf doth "#define cl_valid_num(x) (!isnan(x.re))\n";
    fprintf doth "#define cl_dbl_re(x) (x.re)\n";
    fprintf doth "#define cl_dbl_im(x) (x.im)\n";
    fprintf doth "extern int cl_valid_arr(coal_arr);\n";
    fprintf doth "extern coal_num cl_get_elem(coal_arr, int);\n";
    fprintf doth "extern coal_num cl_put_elem(coal_arr, int, coal_num);\n";
    fprintf doth "#define cl_len(x) (x.cp.num)\n";
    fprintf doth "#define cl_last(x) (x.cp.num-1)\n";
    fprintf doth "extern coal_num cl_num(double re, double im);\n";
    fprintf doth "extern coal_arr cl_real_arr(double* re, int size);\n";
    fprintf doth "extern coal_arr cl_cplx_arr(double* re, int size);\n";

    (* .c *)
    fprintf dotc "#include <math.h>\n";
    fprintf dotc "#include <setjmp.h>\n";
    if (copts.emode=Trace) then
        fprintf dotc "#include <stdio.h>\n";
    fprintf dotc "#include \"%s.h\"\n\n" copts.outfileroot;
    fprintf dotc "#define CL_STORE_SIZE %d\n" copts.storesize;
    fprintf dotc "#define CL_MAX_ARR_SIZE 65535\n";
    fprintf dotc "#define CL_ARR_CPX_INT 0\n";
    fprintf dotc "#define CL_ARR_CPX_EXT 1\n";
    fprintf dotc "#define CL_ARR_RE_EXT 2\n";
    fprintf dotc "#define CL_ARR_RNG 3\n\n";

    fprintf dotc "static coal_num _cl_zero = {0.0, 0.0};\n";
    fprintf dotc "static coal_num _cl_one = {1.0, 0.0};\n\n";

    (* _cl_handle_err needs forward declaration *)
    if (copts.emode=Trace) then fprintf dotc "char* _cl_err_to_str(int err);\n";
```

(cbackend.ml con't)

```
(* error handler macro for client visible functions *)
    fprintf dotc
"#define _cl_handle_err(ret)\\\n\
if (_cl_call_lev==0) \\\n\
{\\\n\
\ if ((_cl_last_err=setjmp(_cl_top))!=0)\\\n\
\ {\\\n";

    (* output any encountered run time error in trace mode *)
    if (copts.emode=Trace) then
        fprintf dotc
"   printf(\"%s\", _cl_err_to_str(_cl_last_err));\\\n" "%s.\\n";

    fprintf dotc
"\   _cl_call_lev = -1;\\\n\
\   return ret;\\\n\
\ }\\\n\
}\n\n";

    fprintf dotc "static int _cl_last_err = 0;\n";
    fprintf dotc "static int _cl_call_lev = -1;\n";
    fprintf dotc "static jmp_buf _cl_top;\n\n";
    fprintf dotc "static coal_num _cl_arr_store[CL_STORE_SIZE];\n";
    fprintf dotc "static int _cl_used = 0;\n";
    fprintf dotc "static int _cl_era = 0;\n\n";

    (* C optimizer should be able to remove this *)
    fprintf dotc "void _cl_compile_asserts()\n\
                 {\n\
                 #define COAL_CT_ASSERT(e) ((void)sizeof(char[1 - 2*!(e)]))\n\
                 COAL_CT_ASSERT(sizeof(double)==8u);\n\
                 COAL_CT_ASSERT(sizeof(unsigned int)==4u);\n\
                 }\n\n";

    fprintf dotc "coal_num _cl_invalid_num()\n\
                  {\n\
                  \   unsigned int nan[4];\n\
                  \   nan[0]=0xFFFFFFFF;\n\
                  \   nan[1]=0xFFFFFFFF;\n\
                  \   nan[2]=0xFFFFFFFF;\n\
                  \   nan[3]=0xFFFFFFFF;\n\
                  \   return (*((coal_num*)(&nan)));\n\
                  }\n\n";
    fprintf dotc "coal_arr _cl_invalid_arr()\n\
                  {\n\
                  \   coal_arr bad;\n\
                  \   bad.cp.kind = 0;\n\
                  \   bad.cp.era = 0;\n\
                  \   bad.cp.num = 0;\n\
                  \   bad.cp.parr = 0;\n\
                  \   return bad;\n\
                  }\n\n";
    fprintf dotc "#define _cl_check_arr(ca) { if
((ca.cp.parr==0)||((ca.cp.kind==CL_ARR_CPX_INT) && (ca.cp.era!=_cl_era))) longjmp(_cl_top,
CL_ERR_BAD_ARRAY); } \n\n";
    fprintf dotc "#define _cl_check_idx(idx,ca) { if ((idx<0)||(idx>=ca.cp.num))
longjmp(_cl_top, CL_ERR_ILLEGAL_INDEX); }\n\n";
    fprintf dotc "int _cl_is_zero(coal_num x)\n\
                  {\n\
                  \   return ((x.re==0.0) && (x.im==0.0));\n\
                  }\n\n";
    fprintf dotc "int _cl_is_true(coal_num x)\n\
                  {\n\
                  \   return ((x.re>=0.5)||(x.re<=-0.5));\n\
                  }\n\n";
    fprintf dotc "coal_num _cl_rect(double re, double im)\n\
                  {\n\
                  \   coal_num r;\n\
                  \   r.re = re; r.im = im;\n\
```

(cbackend.ml con't)

```
                \   return r;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_polar(double m, double theta)\n\
                {\n\
                \   coal_num r;\n\
                \   r.re = m * cos(theta);\n\
                \   r.im = m * sin(theta);\n\
                \   return r;\n\
                }\n\n";
    fprintf dotc "double _cl_mag_dbl(coal_num x)\n\
                {\n\
                \   return sqrt(x.re*x.re + x.im*x.im);\n\
                }\n\n";
    fprintf dotc "double _cl_phase_dbl(coal_num x)\n\
                {\n\
                \   return atan2(x.im, x.re);\n\
                }\n\n";
    fprintf dotc "coal_num _cl_negate(coal_num x)\n\
                {\n\
                \   x.re = -x.re; x.im = -x.im;\n\
                \   return x;\n\
                }\n";
    fprintf dotc "coal_num _cl_add(coal_num a, coal_num b)\n\
                {\n\
                \   coal_num r;\n\
                \   r.re = a.re + b.re;\n\
                \   r.im = a.im + b.im;\n\
                \   return r;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_sub(coal_num a, coal_num b)\n\
                {\n\
                \   coal_num r;\n\
                \   r.re = a.re - b.re;\n\
                \   r.im = a.im - b.im;\n\
                \   return r;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_mul(coal_num a, coal_num b)\n\
                {\n\
                \   coal_num r;\n\
                \   r.re = (a.re * b.re - a.im * b.im);\n\
                \   r.im = (a.im * b.re + a.re * b.im);\n\
                \   return r;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_div(coal_num a, coal_num b)\n\
                {\n\
                \   coal_num r;\n\
                \   double denom;\n\n\
                \   if (_cl_is_zero(a))\n\
                \   {\n\
                \       r.re = 0.0;\n\
                \       r.im = 0.0;\n\
                \       return r;\n\
                \   }\n\n\
                \   if (_cl_is_zero(b))\n\
                \   {\n\
                \       r.re = HUGE_VAL;\n\
                \       r.im = HUGE_VAL;\n\
                \       return r;\n\
                \   }\n\n\
                \   denom = (b.re * b.re + b.im * b.im);\n\n\
                \   r.re = (a.re * b.re + a.im * b.im) / denom;\n\
                \   r.im = (b.re * a.im - a.re * b.im) / denom;\n\
                \   return r;\n\
                }\n\n";
```

(cbackend.ml con't)

```
(*
           x ^ p

           x = m * e^(i*theta)
           x = e ^ (ln m + i*theta)
           x ^ p = e ^ ((ln m + i*theta) * p)

           q = (ln m + i*theta) * p

           x ^ p = (e ^ re{q}) *(e ^ i*im{q})

           m ' = (e ^ re{q})
           theta' = im{q}

           x ^ p = m' * e ^ i*theta'
         *)
    fprintf dotc "coal_num _cl_expon(coal_num x, coal_num p)\n\
                 {\n\
                 \    coal_num r;\n\
                 \    double ln_m;\n\
                 \    double theta;\n\
                 \    coal_num q;\n\
                 \    double m_prime;\n\
                 \    double theta_prime;\n\n\
                 \    if (_cl_is_zero(x)) return x;\n\n\
                 \    if (_cl_is_zero(p)) return _cl_rect(1.0, 0.0);\n\n\
                 \    ln_m = log(_cl_mag_dbl(x));\n\
                 \    theta = _cl_phase_dbl(x);\n\
                 \    q = _cl_mul(_cl_rect(ln_m, theta), p);\n\n\
                 \    m_prime = exp(q.re);\n\
                 \    theta_prime = q.im;\n\n\
                 \    r = _cl_polar(m_prime, theta_prime);\n\n\
                 \    return r;\n\
                 }\n\n";
    fprintf dotc "coal_num _cl_eq(coal_num a, coal_num b)\n\
                 {\n\
                 \   return (a.re==b.re && a.im==b.im)?_cl_one:_cl_zero;\n\
                 }\n\n";
    fprintf dotc "coal_num _cl_neq(coal_num a, coal_num b)\n\
                 {\n\
                 \   return (a.re!=b.re || a.im!=b.im)?_cl_one:_cl_zero;\n\
                 }\n\n";
    fprintf dotc "coal_num _cl_lt(coal_num a, coal_num b)\n\
                 {\n\
                 \   return (a.re<b.re)?_cl_one:_cl_zero;\n\
                 }\n\n";
    fprintf dotc "coal_num _cl_lte(coal_num a, coal_num b)\n\
                 {\n\
                 \   return (a.re<=b.re)?_cl_one:_cl_zero;\n\
                 }\n\n";
    fprintf dotc "coal_num _cl_gt(coal_num a, coal_num b)\n\
                 {\n\
                 \   return (a.re>b.re)?_cl_one:_cl_zero;\n\
                 }\n\n";
    fprintf dotc "coal_num _cl_gte(coal_num a, coal_num b)\n\
                 {\n\
                 \   return (a.re>=b.re)?_cl_one:_cl_zero;\n\
                 }\n\n";
```

(cbackend.ml con't)

```
    fprintf dotc "#define _cl_ge2(ca, cnidx) cl_get_elem(ca, (int)cnidx.re)\n\n";
    fprintf dotc "#define _cl_pe2(ca, cnidx, val) cl_put_elem(ca,(int)cnidx.re,val)\n\n";
    fprintf dotc "coal_arr _cl_new_arr(int N)\n\
                {\n\
                \   coal_arr nca;\n\
                \   if (N > (CL_STORE_SIZE - _cl_used)) longjmp(_cl_top,
CL_ERR_OUT_OF_MEMORY);\n\
                \   nca.cp.kind = CL_ARR_CPX_INT;\n\
                \   nca.cp.era = _cl_era;\n\
                \   nca.cp.num = N;\n\
                \   nca.cp.parr = _cl_arr_store + _cl_used;\n\
                \   _cl_used += N;\n\
                \   return nca;\n\
                }\n\n";
    fprintf dotc "coal_arr _cl_map(coal_num (*f)(coal_num), coal_arr ca)\n\
                {\n\
                \   int idx;\n\
                \   coal_arr nca;\n\
                \   _cl_check_arr(ca);\n\
                \   nca = _cl_new_arr(cl_len(ca));\n\
                \   for (idx=0; idx<cl_len(ca); ++idx)\n\
                \   {\n\
                \       nca.cp.parr[idx] = f(cl_get_elem(ca, idx));\n\
                \   }\n\
                \   return nca;\n\
                }\n\n";
    fprintf dotc "coal_arr _cl_rng1(int start, int stop, int step)\n\
                {\n\
                \   coal_arr nca;\n\
                \   int diff;\n\
                \   if (step == 0) longjmp(_cl_top, CL_ERR_STEP_IS_ZERO);\n\
                \   if (  ((start > stop) && (step > 0))\n\
                \       ||((start < stop) && (step < 0)) ) longjmp(_cl_top,
CL_ERR_STEP_WRONG_SIGN);\n\
                \   nca.rng.kind = CL_ARR_RNG;\n\
                \   nca.rng.num = ((stop - start)/step + 1);\n\
                \   nca.rng.start = (short) start;\n\
                \   nca.rng.step = (short) step;\n\
                \   return nca;\n\
                }\n\n";
    fprintf dotc "#define _cl_rng2(start, stop, step) _cl_rng1((int)start.re,
(int)stop.re, (int)step.re)\n\n";
    fprintf dotc "coal_num _cl_reduce_num(coal_num (*f)(coal_num, coal_num), coal_num acc,
coal_arr ca)\n\
                {\n\
                \    int idx;\n\
                \    _cl_check_arr(ca);\n\
                \    for (idx=0; idx<cl_len(ca); ++idx)\n\
                \    {\n\
                \        acc = f(acc, cl_get_elem(ca, idx));\n\
                \    }\n\
                \    return acc;\n\
                }\n\n";
    fprintf dotc "coal_arr _cl_reduce_arr(coal_arr (*f)(coal_arr, coal_num), coal_arr acc,
coal_arr ca)\n\
                {\n\
                \    int idx;\n\
                \    _cl_check_arr(ca);\n\
                \    _cl_check_arr(acc);\n\
                \    for (idx=0; idx<cl_len(ca); ++idx)\n\
                \    {\n\
                \        acc = f(acc, cl_get_elem(ca, idx));\n\
                \    }\n\
                \    return acc;\n\
                }\n\n";

    (* builtins *)
    fprintf dotc "coal_num _cl_re(coal_num x)\n\
                {\n\
```

(cbackend.ml con't)

```
                \   x.im = 0.0;\n\
                \   return x;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_im(coal_num x)\n\
                {\n\
                \   x.re = x.im;\n\
                \   x.im = 0.0;\n\
                \   return x;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_conj(coal_num x)\n\
                {\n\
                \   x.im = -x.im;\n\
                \   return x;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_mag(coal_num x)\n\
                {\n\
                \   x.re = _cl_mag_dbl(x);\n\
                \   x.im = 0.0;\n\
                \   return x;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_phase(coal_num x)\n\
                {\n\
                \   x.re = _cl_phase_dbl(x);\n\
                \   x.im = 0.0;\n\
                \   return x;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_sin(coal_num x)\n\
                {\n\
                \   x.re = sin(x.re);\n\
                \   x.im = 0.0;\n\
                \   return x;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_cos(coal_num x)\n\
                {\n\
                \   x.re = cos(x.re);\n\
                \   x.im = 0.0;\n\
                \   return x;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_tan(coal_num x)\n\
                {\n\
                \   x.re = tan(x.re);\n\
                \   x.im = 0.0;\n\
                \   return x;\n\
                }\n\n";
    fprintf dotc "coal_num _cl_sqrt(coal_num x)\n\
                {\n\
                \   double m;\n\
                \   double theta;\n\
                \   coal_num r;\n\
                \   m = sqrt(_cl_mag_dbl(x));\n\
                \   theta = _cl_phase_dbl(x) / 2.0;\n\
                \   r = _cl_polar(m, theta);\n\
                \   return r;\n\
                }\n\n";
    (*
            e ^ x

            x = (a + i*b)

            e ^ (a + i*b)
            e^a * e^(i*b)
            e^a * (cos b + i*sin b)
          *)
    fprintf dotc "coal_num _cl_exp(coal_num x)\n\
                {\n\
                \   double exp_a;\n\
                \   double b;\n\
                \   coal_num r;\n\
                \   if (_cl_is_zero(x)) return _cl_one;\n\
```

(cbackend.ml con't)

```
               \   exp_a = exp(x.re);\n\
               \   b = x.im;\n\
               \   r = _cl_rect(exp_a*cos(b), exp_a*sin(b));\n\
               \   return r;\n\
               }\n\n";

   fprintf dotc "coal_num _cl_distance(coal_num a, coal_num b)\n\
               {\n\
               \   return _cl_mag(_cl_sub(a,b));\n\
               }\n\n";
   fprintf dotc "#define _cl_len(x) _cl_rect(cl_len(x),0.0)\n\n";
   fprintf dotc "#define _cl_last(x) _cl_rect(cl_last(x),0.0)\n\n";
   fprintf dotc "coal_num _cl_not(coal_num x)\n\
               {\n\
               \   return (fabs(x.re)<.5?_cl_one:_cl_zero);\n\
               }\n\n";

   (* externally visible helpers *)
   fprintf dotc "void cl_free()\n\
                {\n\
                \   _cl_used = 0;\n\
                \   _cl_era = 0x3FFF&(_cl_era+1);\n\
                }\n\n";
   fprintf dotc "int cl_last_err()\n\
                {\n\
                \   return _cl_last_err;\n\
                }\n\n";
   fprintf dotc "int cl_valid_arr(coal_arr ca)\n\
                {\n\
                \   if (ca.cp.parr == 0)\n\
                \   {\n\
                \       return 0;\n\
                \   }\n\
                \   else if ((ca.cp.kind==CL_ARR_CPX_INT) && (ca.cp.era!=_cl_era))\n\
                \   {\n\
                \       return 0;\n\
                \   }\n\
                \   else\n\
                \   {\n\
                \       return 1;\n\
                \   }\n\
                }\n\n";
   fprintf dotc "coal_num cl_get_elem(coal_arr ca, int idx)\n\
               {\n\
               \   coal_num _ret;\n\
               \   ++_cl_call_lev;\n\
               \   _cl_handle_err(_cl_invalid_num());\n\
               \   _cl_check_idx(idx,ca);\n\
               \   switch (ca.cp.kind)\n\
               \   {\n\
               \       case(CL_ARR_CPX_INT):\n\
               \       case(CL_ARR_CPX_EXT):\n\
               \         _ret = ca.cp.parr[idx];\n\
               \         break;\n\
               \       case(CL_ARR_RE_EXT):\n\
               \         _ret = _cl_rect(ca.re.parr[idx], 0.0);\n\
               \         break;\n\
               \       case(CL_ARR_RNG):\n\
               \         _ret = _cl_rect((ca.rng.start + idx * ca.rng.step), 0.0);\n\
               \         break;\n\
               \       default:\n\
               \         longjmp(_cl_top, CL_ERR_BAD_ARRAY);\n\
               \   }\n\
               \   --_cl_call_lev;\n\
               \   return _ret;\n\
               }\n\n";
```

(cbackend.ml con't)

```
fprintf dotc "coal_num cl_put_elem(coal_arr ca, int idx, coal_num val)\n\
                {\n\
                \   ++_cl_call_lev;\n\
                \   _cl_handle_err(_cl_invalid_num());\n\
                \   _cl_check_idx(idx,ca);\n\
                \   switch (ca.cp.kind)\n\
                \   {\n\
                \       case(CL_ARR_CPX_INT):\n\
                \       case(CL_ARR_CPX_EXT):\n\
                \         ca.cp.parr[idx] = val;\n\
                \         break;\n\
                \       case(CL_ARR_RE_EXT):\n\
                \         ca.re.parr[idx] = val.re;\n\
                \         break;\n\
                \       case(CL_ARR_RNG):\n\
                \         longjmp(_cl_top, CL_ERR_ARRAY_IMMUTABLE);\n\
                \         break;\n\
                \       default:\n\
                \         longjmp(_cl_top, CL_ERR_BAD_ARRAY);\n\
                \   }\n\
                \   --_cl_call_lev;\n\
                \   return val;\n\
                }\n\n";
    fprintf dotc "coal_num cl_num(double re, double im)\n\
                {\n\
                \    return _cl_rect(re, im);\n\
                }\n\n";
    fprintf dotc "coal_arr cl_real_arr(double* re, int size)\n\
                {\n\
                \    coal_arr r;\n\
                \    if ((re == 0)||(size<1)||(size>CL_MAX_ARR_SIZE))\n\
                \    {\n\
                \        _cl_last_err = CL_ERR_BAD_ARRAY;\n\
                \        r.re.parr = 0;\n\
                \    }\n\
                \    else\n\
                \    {\n\
                \        r.re.kind = CL_ARR_RE_EXT;\n\
                \        r.re.num = size;\n\
                \        r.re.parr = re;\n\
                \    }\n\
                \    return r;\n\
                }\n\n";
    fprintf dotc "coal_arr cl_cplx_arr(double* re, int size)\n\
                {\n\
                \    coal_arr r;\n\
                \    if ((re == 0)||(size<2)||(size>(2*CL_MAX_ARR_SIZE)))\n\
                \    {\n\
                \        _cl_last_err = CL_ERR_BAD_ARRAY;\n\
                \        r.cp.parr = 0;\n\
                \    }\n\
                \    else\n\
                \    {\n\
                \        r.cp.kind = CL_ARR_CPX_EXT;\n\
                \        r.cp.era = 0;\n\
                \        r.cp.num = size/2;\n\
                \        r.cp.parr = (coal_num*)re;\n\
                \    }\n\
                \    return r;\n\
                }\n\n";

    (* trace support *)
    if (copts.emode=Trace) then
        (fprintf dotc "void _cl_dump(int indent, char *msg)\n\
                    {\n\
                    \   printf(\"%s\", indent, \"\", msg);\n\
                    }\n\n" "%*.s%s\\n";
         fprintf dotc "void _cl_dump_num(int indent, coal_num x)\n\
                    {\n\
```

(cbackend.ml con't)

```
                    \   printf(\"%s\", indent, \"\", x.re, x.im);\n\
                    }\n\n" "%*.s(%f,%f)\\n";
        fprintf dotc "char* _cl_arr_kind_to_str(coal_arr ca)\n\
                    {\n\
                    \   switch(ca.cp.kind)\n\
                    \   {\n\
                    \       case(CL_ARR_CPX_INT): return \"%s\";\n\
                    \       case(CL_ARR_CPX_EXT): return \"%s\";\n\
                    \       case(CL_ARR_RE_EXT): return \"%s\";\n\
                    \       case(CL_ARR_RNG): return \"%s\";\n\
                    \       default: return \"UNKNOWN ARRAY KIND\";\n\
                    \   }\n\
                    }\n\n" "CL_ARR_CPX_INT" "CL_ARR_CPX_EXT" "CL_ARR_RE_EXT"
"CL_ARR_RNG";
        fprintf dotc "char* _cl_err_to_str(int err)\n\
                    {\n\
                    \   switch(err)\n\
                    \   {\n\
                    \       case(CL_ERR_ILLEGAL_INDEX): return \"%s\";\n\
                    \       case(CL_ERR_ARRAY_IMMUTABLE): return \"%s\";\n\
                    \       case(CL_ERR_OUT_OF_MEMORY): return \"%s\";\n\
                    \       case(CL_ERR_STEP_IS_ZERO): return \"%s\";\n\
                    \       case(CL_ERR_STEP_WRONG_SIGN): return \"%s\";\n\n\
                    \       case(CL_ERR_BAD_ARRAY): return \"%s\";\n\n\
                    \       default: return \"UNKNOWN ERROR\";\n\n\
                    \   }\n\
                    }\n\n" "CL_ERR_ILLEGAL_INDEX" "CL_ERR_ARRAY_IMMUTABLE"
"CL_ERR_OUT_OF_MEMORY" "CL_ERR_STEP_IS_ZERO" "CL_ERR_STEP_WRONG_SIGN" "CL_ERR_BAD_ARRAY";
        fprintf dotc "void _cl_dump_arr(int indent, coal_arr ca)\n\
                    {\n\
                    \   int idx;\n\
                    \   printf(\"%s\", indent, \"\", _cl_arr_kind_to_str(ca));\n\
                    \   if (ca.cp.kind!=CL_ARR_RNG)\n\
                    \   {\n\
                    \       for(idx=0; idx<cl_len(ca); ++idx)\n\
                    \       {\n\
                    \           _cl_dump_num(indent, cl_get_elem(ca, idx));\n\
                    \       }\n\
                    \   }\n\
                    \   else\n\
                    \   {\n\
                    \       printf(\"%s\", indent, \"\", ca.rng.num, ca.rng.start,
ca.rng.step);\n\
                    \   }\n\
                    }\n\n" "%*.s%s\\n" "%*.snum=%d start=%d step=%d\\n")

let emit_func_def copts dotc builtins link_from link_to fdef =

    let extract_int_lits selist =
        List.rev (
        List.fold_left (fun lits se ->
              match se with
                Real(fps, true), _ -> fps::lits
              | _, _ -> lits
              ) [] selist
        )
    in
    let rec emit_exp = function

      (* leaf expression nodes *)
        Real(fps, _), _ -> fprintf dotc "_cl_rect(%s, 0.0)" fps

      | Imag(fps), _ -> fprintf dotc "_cl_rect(0.0, %s)" fps

      | Id(name), _ ->
        let parent_env = findenv fdef.flocals name in
        if (parent_env.context = fdef.fname) (* local variable? *)
            then fprintf dotc "%s" name (* locally defined *)
            else fprintf dotc "*p%s" name (* imported via static link *)
```

(cbackend.ml con't)

```
    | Negate(se), _ ->
      fprintf dotc "_cl_negate(\n";
      emit_exp se;
      fprintf dotc ")\n"

    | Binop(se1, op, se2), _ ->
      fprintf dotc "%s(" (string_of_op op);
      emit_exp se1;
      fprintf dotc ",\n";
      emit_exp se2;
      fprintf dotc ")\n"

    | Assign(id, se), _ ->
      fprintf dotc "%s=" id;
      emit_exp se

    | GetElem(se1, se2), _ ->
      (match (extract_int_lits [se2]) with
        [l2] ->
          fprintf dotc "cl_get_elem(";
          emit_exp se1;
          fprintf dotc ",\n%s)\n" l2
      | _ ->
          fprintf dotc "_cl_ge2(";
          emit_exp se1;
          fprintf dotc ",\n";
          emit_exp se2;
          fprintf dotc ")\n")

    | PutElem(se1, se2, se3), _ ->
      (match (extract_int_lits [se2]) with
          [l2] ->
          fprintf dotc "cl_put_elem(";
          emit_exp se1;
          fprintf dotc ",\n%s,\n" l2
        | _ ->
          fprintf dotc "_cl_pe2(";
          emit_exp se1;
          fprintf dotc ",\n";
          emit_exp se2;
          fprintf dotc ",\n");
      emit_exp se3;
      fprintf dotc ")\n"

    | Map(callee, se), _ ->
      fprintf dotc "_cl_map(";
      (match callee with
          Named(name) ->
          fprintf dotc "%s" (if (List.mem name builtins) then "_cl_"^name else name)
        | Lambda(fdef) ->
          fprintf dotc "%s" fdef.fname);
      fprintf dotc ",\n";
      emit_exp se;
      fprintf dotc ")\n"

    | Range(se1, se2, se3), _ ->
      (match (extract_int_lits [se1;se2;se3]) with
          (* all int literals *)
          [l1; l2; l3] ->

          fprintf dotc "_cl_rng1(%s,%s,%s)\n" l1 l2 l3

            (* 2 or fewer int literals *)
       |  _ ->

          fprintf dotc "_cl_rng2(";
          emit_exp se1;
          fprintf dotc ",\n";
```

(cbackend.ml con't)

```
            emit_exp se2;
            fprintf dotc ",\n";
            emit_exp se3;
            fprintf dotc ")\n"
        )

      | Invoke(callee, actuals), _ ->
        (match callee with
            Named(name) ->
            fprintf dotc "%s" (if (List.mem name builtins) then "_cl_"^name else name)
          | Lambda(fdef) ->
            fprintf dotc "%s" fdef.fname);
        fprintf dotc "(";
        (match actuals with
            [] -> ()
          | _  ->
            (emit_exp (List.hd actuals);
            (List.iter (fun a -> fprintf dotc ", "; emit_exp a)(List.tl actuals))
                     ) );
        fprintf dotc ")\n"

      | Reduce(callee, se1, se2), _ ->
        let rec which_reduce = function
            Num -> fprintf dotc "_cl_reduce_num("
          | Tbd -> fprintf dotc "_cl_reduce_num("
          | NumArr -> fprintf dotc "_cl_reduce_arr("
          | Var({contents=inner})-> which_reduce inner
          | Func(_) -> fprintf dotc "/* Func not supported */"
        in
        which_reduce (snd se1);
        (match callee with
            Named(name) ->
            fprintf dotc "%s,\n" (if (List.mem name builtins) then "_cl_"^name else name)
          | Lambda(fdef) ->
            fprintf dotc "%s,\n" fdef.fname);

        emit_exp se1;
        fprintf dotc ",\n";
        emit_exp se2;
        fprintf dotc ")\n"

      | IfThenElse(se1, se2, se3), _ ->
        fprintf dotc "(_cl_is_true(";
        emit_exp se1;
        fprintf dotc ")?";
        emit_exp se2;
        fprintf dotc ":";
        emit_exp se3;
        fprintf dotc ")\n"

      | Sequence(se1, se2), _ ->

        let rec flatten_sequence selist se =
        match se with
           Sequence(se1, se2), _ -> flatten_sequence (se2::selist) se1
         | non_seq_se -> non_seq_se::selist in

        let selist = flatten_sequence [se2] se1 in
        fprintf dotc "(";
        emit_exp (List.hd selist);
        (List.iter (fun a -> fprintf dotc ",\n "; emit_exp a) (List.tl selist));
        fprintf dotc ")"

    in

    if debug then printf "emitting %s\n" fdef.fname;

    (* definition prototype *)
    fprintf dotc "%s\n{\n" (to_c_proto fdef);
```

(cbackend.ml con't)

```
    (* emit locals *)
    List.iter
        (fun vn -> if (   (* not a function argument... *)
                          (not (List.mem vn fdef.fargs))
                          (* ... and a variable type *)
                       && (match baretyp (findtyp fdef.flocals vn) with
                           Num | NumArr -> true | _ -> false) )
                    then fprintf dotc "%s;\n" (to_c_var_decl fdef.flocals vn)
                  ) (symbol_names fdef.flocals);

    (* temporary for return expression - see below *)
    fprintf dotc "%s _ret;\n" (to_c_type (snd fdef.fbody));

    (* emit link from variable *)
    List.iter
        (fun (ctxt, vn, idx) ->
            if (ctxt = fdef.fname)
                then fprintf dotc "_links[%d]=(void*)&%s;\n" idx vn
                ) link_from;

    (* emit link to variable *)
    List.iter
      (fun (ctxt, vn, idx) ->
         if (ctxt = fdef.fname)
             then let ctypename = to_c_type (findtyp fdef.flocals vn)
             in
             fprintf dotc "%s* p%s=(%s*)_links[%d];\n" ctypename vn ctypename idx
             ) link_to;

    let rt = baretyp (snd fdef.fbody) in

    let is_named = (((String.length fdef.fname)<7)||(String.sub fdef.fname 0 7) <>
"_lambda")
    in

    (* error handling - named (top level) functions only *)
    if is_named then (
            (* increment current call level so that setjmp only called once at top *)
            fprintf dotc "++_cl_call_lev;\n";
            fprintf dotc "_cl_handle_err(%s);\n"
                        (if (rt = NumArr)
                             then "_cl_invalid_arr()"
                             else "_cl_invalid_num()") );
    (* trace for debug *)
    if (copts.emode=Trace) then
        fprintf dotc "_cl_dump(_cl_call_lev, \"%s\");\n" (fdef.fname^" begin");

    (* emit body *)
    fprintf dotc "_ret = \n";
    emit_exp fdef.fbody;
    fprintf dotc ";\n";

    (* trace for debug *)
    if (copts.emode=Trace) then
       (fprintf dotc "_cl_dump(_cl_call_lev, \"%s\");\n" (fdef.fname^" end with");
        fprintf dotc "%s(_cl_call_lev, _ret);\n" (if (rt=NumArr) then "_cl_dump_arr" else
"_cl_dump_num"));

    (* decrement call level indicating that we're going back up the stack *)
    if is_named then fprintf dotc "--_cl_call_lev;\n";
    fprintf dotc "return _ret;\n}\n\n";;

let emit_lambda_support o fdefs =

    let link_exists env id links =
        List.exists (fun (ctxt, vn, _) -> ctxt=env.context && vn=id) links
    in
```

(cbackend.ml con't)

```
    (* Extract static link info as well as lambda definitions. *)
    (* Returns: maximum static link index, map of variables to static link indexes, list
of lamda definitions. *)
    let rec extract_lambda_info res env = function

        Id(name), _ ->

        let (curr_idx, link_from, link_to, lfdefs) = res in

        let parent_env, t = find_env_typ env name in

        if debug then
             printf "found %s in %s. def in %s\n w/ type %s.\n"
                name env.context parent_env.context (to_c_type t);

        (match (baretyp t) with
            (* we've come across a variable while descending SAST  *)
            (* figure out if we need to link from or to a variable across parent/child
scopes  *)
            Num
          | NumArr when (env.context <> parent_env.context) ->
            if debug then printf "link %s from %s to %s\n" name parent_env.context
env.context;
            let curr_idx, link_from =

              if (link_exists parent_env name link_from)
                  (* scope and var combo exists.  pass on link index and link_from list *)
                  then curr_idx, link_from
                  (* first time for this scope and var combo.  create new link index &
pass on with updated link_from list *)
                  else curr_idx+1, (parent_env.context,name,curr_idx+1)::link_from

            in
            let link_to =

              if (link_exists env name link_to)
                   (* have already come across this imported var from above scope *)
                   then link_to
                   (* first time to have come across imported var.  record and pass on *)
                   else (env.context,name,curr_idx)::link_to

            in
            (curr_idx, link_from, link_to, lfdefs)

          | _ -> res (* nothing to do *)
        )

      | Negate(se), _ ->
        extract_lambda_info res env se

      | Binop(se1, _, se2), _ ->
        let res = extract_lambda_info res env se1 in
        extract_lambda_info res env se2

      | Assign(_, se), _ ->
        extract_lambda_info res env se

      | GetElem(se1, se2), _ ->
        let res = extract_lambda_info res env se1 in
        extract_lambda_info res env se2

      | PutElem(se1, se2, se3), _ ->
        let res = extract_lambda_info res env se1 in
        let res = extract_lambda_info res env se2 in
        extract_lambda_info res env se3

      | Map(callee, se), _ ->
        let res = descend_lambda res callee in
        extract_lambda_info res env se
```

(cbackend.ml con't)

```
      | Range(se1, se2, se3), _ ->
        let res = extract_lambda_info res env se1 in
        let res = extract_lambda_info res env se2 in
        extract_lambda_info res env se3

      | Reduce(callee, se1, se2), _ ->
        let res = descend_lambda res callee in
        let res = extract_lambda_info res env se1 in
        extract_lambda_info res env se2

      | IfThenElse(se1, se2, se3), _ ->
        let res = extract_lambda_info res env se1 in
        let res = extract_lambda_info res env se2 in
        extract_lambda_info res env se3

      | Sequence(se1, se2), _ ->
        let res = extract_lambda_info res env se1 in
        extract_lambda_info res env se2

      | Invoke(callee, actuals), _ ->
        let res = descend_lambda res callee in
        List.fold_left (fun res se -> extract_lambda_info res env se) res actuals

      | _, _ -> res (* nothing to do *)

   (* Used for when we hit a lambda definition - recurse into definition. *)
   and descend_lambda res = function
        Named(_) -> res (* nothing to do *)
      | Lambda(fdef) ->
        let curr_idx, link_from, link_to, lfdefs = res in
        (* descend into lambda with new scope *)
        extract_lambda_info (curr_idx,link_from,link_to,(fdef::lfdefs)) fdef.flocals
fdef.fbody

   in
   (* collect static link info for lambdas by descending into all named functions *)
   (* max_idx - maximum index ever needed at one time by static link table *)
   (* link_from - which variables are exported to child lambda scopes *)
   (* link_to - which variables are imported from parent scopes *)
   (* lfdefs - all lambda definitions encountered *)
   let max_idx, link_from, link_to, lfdefs
        = List.fold_left
            (fun (curr_idx, link_from, link_to, lfdefs) fdef ->
                let curr_idx', link_from', link_to', lfdefs'
                  = extract_lambda_info (-1,[],[],[]) fdef.flocals fdef.fbody
                in
                (max curr_idx' curr_idx,
                 (List.rev link_from') @ link_from,
                 (List.rev link_to') @ link_to,
                 lfdefs' @ lfdefs)
                )
            (-1,[],[],[]) fdefs
   in

   (* If required, declare static array used for establishing static links between parent
and  *)
   (* subordinate scopes. *)
   if (max_idx>=0) then fprintf o "static void* _links[%d];\n\n" (max_idx + 1);

   (* output forward declarations for lambda functions so that named functions can
reference them: *)
   (* lambda definitions will be placed after named function definitions *)
   List.iter (fun fdef -> fprintf o "static %s;\n" (to_c_proto fdef)) lfdefs;

   fprintf o "\n";

   (* return the static link index map and the Lambda definitions *)
   (link_from, link_to, lfdefs)
```

(cbackend.ml con't)

```
let emit copts builtins fdefs =

    let hdr, impl = (copts.outfileroot ^ ".h"), (copts.outfileroot ^ ".c") in
    let doth, dotc = (open_out hdr), (open_out impl) in

    (* header declarations and internals *)
    emit_header_beg copts doth;
    emit_internals copts doth dotc;

    (* export user functions *)
    List.iter (fun fdef ->
               fprintf doth "extern %s;\n" (to_c_proto fdef)
               ) fdefs;
    (* end of header *)
    emit_header_end doth;

    (* setup static linkage between lambdas and parent functions *)
    let link_from, link_to, lfdefs = emit_lambda_support dotc fdefs in

    (* emit function definitions *)
    List.iter (emit_func_def copts dotc builtins link_from link_to) fdefs;
    List.iter (emit_func_def copts dotc builtins link_from link_to) lfdefs;

    close_out doth;
    close_out dotc
```

8.11 printer.mli

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

val string_of_program : Sast.func_def list -> string
```

8.12 printer.ml

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

open Types
open Sast
open Sym

let string_of_op = function
    Ast.Add -> "+"
  | Ast.Sub -> "-"
  | Ast.Mul -> "*"
  | Ast.Div -> "/"
  | Ast.Pow -> "^"
  | Ast.Eq  -> "="
  | Ast.Neq -> "<>"
  | Ast.Lt  -> "<"
  | Ast.Lte -> "<="
  | Ast.Gt  -> ">"
  | Ast.Gte -> ">="

let rec string_of_expr = function
    Real(fps, _) -> fps

  | Imag(fps) -> fps ^ "i"

  | Id(name) -> name

  | Negate(se) ->
      "-" ^ (string_of_expr (fst se))

  | Binop(se1, op, se2) ->
      (string_of_expr (fst se1))
      ^ " "
      ^ (string_of_op op) ^ " "
      ^ (string_of_expr (fst se2))

  | Assign(name, se) ->
      (string_of_type_clean (snd se))
      ^ " "
      ^ name
      ^ " <- "
      ^ string_of_expr (fst se)

  | GetElem(se1, se2) ->
      (string_of_expr (fst se1))
      ^ "["
      ^ (string_of_expr (fst se2))
      ^ "]"

  | PutElem(se1, se2, se3) ->
      (string_of_expr (fst se1))
      ^ "["
      ^ (string_of_expr (fst se2))
      ^ "]<-"
      ^ (string_of_expr (fst se3))

  | Map(callee, se) ->
      (string_of_callee callee)
      ^ "{"
      ^ string_of_expr (fst se)
      ^ "}"
```

(printer.ml con't)

```
  | Range(se1, se2, se3) ->
      string_of_expr (fst se1)
      ^ ".." ^ string_of_expr (fst se2)
      ^ "\\" ^ string_of_expr (fst se3)

  | Invoke(callee, selist) ->
      (string_of_callee callee) ^ "("
      ^ String.concat ", " (List.map (fun se-> string_of_expr (fst se)) selist)
      ^ ")"

  | Reduce(callee, se1, se2) ->
      (string_of_callee callee)
      ^ "{"
      ^ string_of_expr (fst se1)
      ^ ", "
      ^ string_of_expr (fst se2)
      ^ "}"

  | IfThenElse(se1, se2, se3) ->
      "if " ^ string_of_expr (fst se1)
      ^ " then " ^ string_of_expr (fst se2)
      ^ " else " ^ string_of_expr (fst se3)

  | Sequence(se1, se2) ->
      (string_of_expr (fst se1))
      ^ ";\n"
      ^ (string_of_expr (fst se2))

and string_of_callee = function
    Named(fname) -> fname
  | Lambda(fdef) ->
    let args =
       String.concat ", "
         (List.map
               (fun name -> (string_of_type_clean (findtyp fdef.flocals name)) ^ " " ^
name)
               fdef.fargs
               ) in
    "(" ^ args ^ " -> " ^ string_of_expr (fst fdef.fbody) ^ ")";;

let string_of_fdef fdef =
    let args =
       String.concat ", "
         (List.map
               (fun name -> (string_of_type_clean (findtyp fdef.flocals name)) ^ " " ^
name)
               fdef.fargs
               ) in
    (string_of_type_clean (snd fdef.fbody)) ^ " " ^ fdef.fname ^ "(" ^ args ^ ") -> \n"
    ^ string_of_expr (fst fdef.fbody)
;;

(* convert SAST to a string *)
let string_of_program fdefs =
    String.concat "\n\n" (List.map string_of_fdef fdefs)
```

8.13 coalopts.mli

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

type compile_opts =
{
    emode : emit_mode;
    sources : string list;
    storesize : int;
    outfileroot : string
}

and emit_mode = Normal | Trace | Sast
```

8.14 coal.ml

```
(* COMS W4115, COAL, Eliot Scull, CUID: C000056091 *)

open Coalopts
open Arg
open Printf
open Types

let debug = false

(* entry point *)
let _ =
    if debug then printf "start\n";

    (* declare built in functions so that user code may refer to them *)
    let builtins =
    [
        ("re"       ,(Func(Num, [Num])));
        ("im"       ,(Func(Num, [Num])));
        ("conj"     ,(Func(Num, [Num])));
        ("mag"      ,(Func(Num, [Num])));
        ("phase"    ,(Func(Num, [Num])));
        ("sin"      ,(Func(Num, [Num])));
        ("cos"      ,(Func(Num, [Num])));
        ("tan"      ,(Func(Num, [Num])));
        ("sqrt"     ,(Func(Num, [Num])));
        ("exp"      ,(Func(Num, [Num])));
        ("distance" ,(Func(Num, [Num; Num])));
        ("len"      ,(Func(Num, [NumArr])));
        ("last"     ,(Func(Num, [NumArr])));
        ("not"      ,(Func(Num, [Num])))
    ]
    in
    if debug then printf "add builtins\n";
    (* add builtins to global namespace *)
    List.iter (fun (n,t) -> Semantics.addbuiltin n t) builtins;

    (* process command line arguments *)
    let copts =

      if debug then printf "cmd line opts\n";
      let emoderef = ref Normal in
      let outfilerootref = ref "coalout" in
      let storesizeref = ref 128 in
      let sourcesref = ref [] in
        Arg.parse
         (* options defined *)
         [
           ("-sast"
           , Arg.Unit(fun () -> emoderef:= Sast)
           , "Dump semantically checked abstract syntax tree for each specified .coal
file to standard out.  Inhibits generation of .c output.");
```

(coal.ml con't)

```
          ("-o"
          , Arg.String(fun name -> outfilerootref:=name )
          , "Specify root name of c output files.  If not specified, default is coalout,
resulting in coalout.h/c being generated.");

          ("-storesize"
          , Arg.Int(fun size -> if ((size < 1)||(size > 65535))
                        then raise (Bad("Illegal size " ^ (string_of_int size)))
                        else storesizeref:=size)
          ,"Specify the total number of elements in the array store used for dynamic
array creation.  Default is 128, minimum is 1, and maximum is 65535.");

           ("-trace"
           , Arg.Unit(fun () -> emoderef:= Trace)
           , "Generate C output with debug tracing.  Traces dumped to standard out via
printf.  This option ignored if -sast specified after.")
         ]
         (* callback to build list of files to compile *)
         (fun f -> if (Filename.check_suffix f ".coal")
                   then sourcesref := f::!sourcesref
                   else raise (Arg.Bad(f)))
         (* message to user output when there's a usage problem *)
         "usage: coal [options] <file1>.coal <file2>.coal ...";

        {emode = !emoderef
        ; sources = List.rev !sourcesref
        ; storesize = !storesizeref
        ; outfileroot = !outfilerootref}

    in

    let total_sast =

        (* build a source file *)
        let build total_sast file =
            let fin = open_in file in
            try
                (* analysis *)
                if debug then printf "build %s\n" file;
                let lexbuf = Lexing.from_channel fin in
                let ast = Parser.program Scanner.token lexbuf in
                let sast = Semantics.check ast in
                close_in fin;
                total_sast @ sast
            with e -> (close_in fin; raise e)
        in

        (* build all specified sources *)
        List.fold_left build [] copts.sources

    in

        if debug then printf "emit code\n";
        (* output generation *)
        match copts.emode with
          Sast -> print_string (Printer.string_of_program total_sast)
        | Normal
        | Trace ->  Cbackend.emit copts (List.map (fun pair->fst pair) builtins)
total_sast
```