SOS Project Report

Terric Abella, Sitong Feng, G Pershing, Sheron Wang

April 27, 2021

Contents

1 Language White Paper

1.1 Introduction

The SOS (Shape Open System) Language is an imperative language with some functional elements designed to render 2D images, especially mathematically interesting images, for example fractals. It is strictly typed and takes the very best syntax from C and Ocaml, but transforms into more elegant and concise language. It also employs OpenGL library to provide essential functions for shape lovers to create, manipulate, and enjoy 2D images rather easily. The goal of SOS is to provide a simple and intuitive tool for users to experiment with computer graphics efficiently, while having the language syntax well-documented and easy to learn.

1.2 Flexibility

The SOS has a non-restrictive syntax for programmer to write program with their unique styling, while following the SOS style guidelines. Also, since SOS renders the image very efficiently, it would be beneficial for programmers to iterate the code based on the visual result. So it would be very quick and simple to test the program, change the code, and achieve optimized performance.

1.3 Usability

SOS is very simple in the sense that it does not add complication to programmers who only want to do simple manipulation of images. The programming hides away the mathematical details and allows users to take all the easy-to-use but yet powerful standard library function for granted. These include math and graphics manipulation. The syntax also has good readability, which would be helpful to share and understand other's code.

1.4 Error Handling

Although SOS does not force the user to write code in a certain way, some generic error checking rules still hold. For example, int type variable cannot be reassigned to a struct. To help the users generate good-to-use code, we have a large volume of test cases to cover a wide range of problems.

1.5 Data Type

SOS has three basic types: bool, int, and float. We also have three reference types: arrays, structures, and functions. Structures of all floats or all ints are treated like mathematical vectors, meaning they can be added and scaled, and you can also take their dot product. SOS also features basic function definition, including recursion. Structures of all floats or all ints are treated like mathematical vectors, meaning they can be added and scaled, and you can also take their dot product.

Matrix multiplication is even supported for square matrices and column vectors.Because in graphics you often need to iterate on a large array of data, we have a powerful implicit array iteration syntax. For example, a function or an operation can be applied to an array to iterate on all its elements, returning a new array of the results.

1.6 Import Library

To increase our language's extensiblily, we implemented naive import which works similar to #include in c. Since it is very straightforward, we encourage users to use this feature to create more powerful and complicated projects. As mentioned, the SOS language has built-in several standard libraries on math operations and graphics.

1.7 Conclusion

With the goal of helping those in need of creative computational manipulation of graphics, SOS has turned out to be the most suitable language to graphics starters, as it is easy to user, and great for advanced users

who long for efficiency and elegance in code. As it being developed further, more support on graphics will allow the users to achieve a lot more through easy manipulation of the 2D, or even 3D graphics.

2 Language Tutorial

2.1 Variables

Variables in SOS associate an *identifier* with a *value* of a certain *type*. This value can then be accessed and modified using the name given. A variable's value is first defined by some expression, for instance a constant value or another variable. Here is an example of variable declarations:

```
a : int = -1
b : float = 3.1415
c : float = b
```

The first symbol is the identifier, which can be any string starting with a letter and containing only letters, underscores, and numbers. The second is the type, here the built-in types of `int`, an integer value, and `float`, a real number. After the equals sign is the variable's initial value: this must be present in every variable definition.

The value of a variable can be checked using a *print* statement, which will output the value to the console's standard output. SOS has two print statements: `print` for integers, and `printf` for floating point values.

```
print(a) // prints "-1"
printf(b) // prints "3.1415"
```

Notice the use of `//` to denote a *comment*. Commented code is not parsed by the compiler, and is only used to enhance readability. A comment starting with `//` with end at the next newline. A comment can also be introduced by `/*`, in which case the comment will end at the next `*/`.

With only this information, a complete program can be made in SOS. However, it cannot do anything interesting.

2.2 Basic Operators

To begin creating more intricate programs, we need to use *operators*. The most fundamental operator is the *assignment* operator, which can re-assign a declared variable.

```
a : int = 1
a = 3
print(a) // prints "3"
```

Values can be combined with the *arithmetic operators* `+`, `-`, `*`, `/`, and `%`. The first four are addition, subtraction, multiplication, and division, as in written arithmetic. The last operator is the *modulo* or *remainder* operator, which gives the remainder after dividing a value by some quotient. The subtraction operator may used both to subtract to numbers and to negate a single number.

```
print(1+2)      // prints "3"
printf(1.5*3.4) // prints "5.1"
print(7 % 2)    // prints "1"
```

In addition to integers and floats, SOS also has a `bool` type, representing either "true" or "false". These values can be combined using the *boolean operators* `&&` (and), `||` (or), and `!` (not). There is no print function for bools, but they can be printed as integers, where `true=1` and `false=0`.

```
print(true || false) // prints "1", or true
print(true && false) // prints "0", or false
print(!true)         // prints "0", or false
```

Boolean values are mainly used to compare other values using the *comparison operators* `==`, `!=`, `<`, `>`, `<=`, `>=`. These mean, in order, is equal to, is not equal to, less than, greater than, less than or equal to, and greater than or equal to.

```
print(1 < 2)  // prints "1"
print(2 == 3) // prints "0"
```

2.3 Expressions

In SOS, most statements are *expressions*, meaning that they have a *value*. For instance, a constant value like `1` is an expression with value 1, and an operator forms an expression with value depending on its operands. Expressions can be nested: for instance, the operands of an operator can both be expressions.

Both variable declarations and assignments are expressions, where the expression value is the value being assigned. This means assignments can be chained:

```
a : int = 1
b : int = a = 2 // a and b are both 2
```

Notice that the rightmost `=` is being processed first, because `=` is *right-associative*. Other operations, such as the arithmetic operations, are *left-associative*. Associativity may be overridden using parentheses, as an expression within parentheses is always evaluated before it is used in other expressions.

```
print(5 - 3 - 1)   // prints 1
print((5 - 3) - 1) // prints 1
print(5 - (3 - 1)) // prints 3
```

In addition to associativity, operators have different *precedence*. Operators of higher precedence are evaluated below operators of lower precedence. For instance, multiplication has higher precedence than addition:

```
print(5 * 3 + 1) // prints 16
print(5 * (3 + 1)) // prints 20
```

The following is the complete list of symbol associativity and precedence. Some of these symbols will be discussed in future sections.

Symbol	Associativity	
`( )`	None	(highest precedence)
`.`	Left	
`!`	Right	
`[ ] { } ( )`	None	(parens in function application)
`* / %`	Left	
`**`	Right	
`+ -`	Left	
`@`	Left	
`of`	Left	
`< > <= >=`	Left	
`== !=`	Left	
`&& ||`	Left	
`=`	Right	
`;`	Left	
`,`	Left	
`if then else`	None	(lowest precedence)

The if/else expression allows for a different expression to be evaluated based on a boolean value. It has three component expressions: the boolean condition, the expression if true, and the expression if false. The conditional expressions must have the same type, so that the whole expression has a consistent type.

```
print(if true then 1 else 0) // prints "1"
```

2.4 Functions

Any complicated program will have computations that must be performed several times. Instead of writing this code over and over, SOS allows the declaration of *functions*, which can be *called* on different inputs to perform computations many times. We have already seen two functions: the predefined `print` and `printf`. A new function can be declared with a function statement:

```
double : (n : int) -> int = 2 * n
print(double(3)) // prints 6
```

Like a variable definition, a function has an identifier, given by the first string. It also has zero or more *arguments*, listed, separated by commas, between parentheses. Finally, it has a *return type* given after the arrow `->`. Then, after the equals is the function *body*, which is an expression. This expression may contain the function's arguments as variables, and must evaluate to a value of the given return type. Just as with `print`, the function is called using parentheses. If there are several arguments, they are separated with commas.

Often, a function needs to perform more operations than can be cleanly fit into one expression. In this case, the *sequencing* operator `;` can be used to combine several expressions into one. These expressions are evaluated in order, left to right.

```
id : (i : int) -> int =
    (j : int = i + 1); j - 1
print(id(2)) // prints "2"
```

Notice that the expression may be freely written on an additional line, or even more than one line, for readability. Also note that the variable definition must be within parentheses, otherwise the `;` would be interpreted as part of the definition.

Functions can also be stored as variables and passed as arguments. When a function is defined as a variable, it can only refer to an existing function, but can be re-assigned to a different function. Functions that are defined with a function statement cannot be re-assigned. The type of a function is denoted by the `func` keyword, a comma-separated list of its argument types, an arrow `->`, and is return type.

```
lt : (i : int, j : int) -> bool = i < j
comp: (i: int, j: int, c: func int, int -> bool) -> bool =
  c(i, j)
less_than : func int, int -> bool = lt
print(comp(1, 2, less_than)) // prints "1"
```

2.5 Arrays

SOS has two ways to create types from other types: *arrays* and *structures*. An array is an ordered list of values, all of the same type, that can be accessed and modified. An array type is denoted as `array` t, where t is some other type. Arrays can be accessed using `[]` brackets and an integer index, where the first element is index 0. An array's length can be accessed using `.length`, which gives the integer length. Attempting to access a negative index or an index beyond the array's length will result in undefined behavior.

```
A : array int = [1, 2, 3]
first : int = A[0]
A[1] = 4
print(first)    // prints "1"
```

```
print(A[1])     // prints "4"
print(A.length) // prints "3"
```

Unlike SOS's primitive types, an array is a *reference* type, meaning that what is stored by the variable is actually just a reference to the memory location of the data. Therefore, assigning an array does not copy the data, instead this must be done with the special function `copy`.

```
A : array int = [1, 2, 3]
B : array int = A
C : array int = copy(A)
A[0] = 5
print(B[0]) //prints "5"
print(C[0]) //prints "1"
```

By default, an array's memory exists for as long as the program runs, even if it cannot be accessed any more. In larger programs where memory management is an issue, array memory can be freed using the `free` function. After being free, attempting to access an array will result in an error.

Array types can be nested as much as desired. With nested array types, `copy` and `free` only act on the uppermost level.

```
A : array array int = [[1], [2, 2]]
B : array array int = copy(A)
A[0] = [5]
A[1][0] = 5
print(B[0][0]) // prints "1"
print(B[1][0]) // prints "5"
```

Arrays have two unique operators: `@` concatenation and `of`. `@` creates a new array by appending the elements of one array onto the end of another. `of` concatenates an array with itself some integer number of times.

```
A : array int = 2 of [1, 2]
B : array int = [3] @ A
// B is now [3, 1, 2, 1, 2]
```

Because array type names can get quite long, it can be useful to have shorter names. An *alias* statement creates a string that can stand in place of another type name (not just array types).

```
alias ints = array int
A : ints = [1, 2]
// identical to A : array int = [1, 2]
```

2.6 Structures

A *structure* is a collection of several variables, called *fields*, of different types. A new structure type is defined with a structure definition statement, after which point it can be used freely. The fields are accessed by name using the `.` operator. As with arrays, structures are reference types, and be used in `copy` and `free`.

```
struct point = {x : float, y : float}
p : point = {1.0, 3.0}      // implicit type
q : point = point{2.0, 3.0} // explicit type
printf(p.x) // prints "1.0"
```

If two structures share the same types in the same order, they can be used interchangeably. However, the field names are determined from the type of the individual variable.

```
struct point = {x : float, y : float}
struct twofloats = {first : float, second : float}
p : point = {1.0, 2.0}
q : twofloats = p
printf(q.first) // prints 1.0
// q.x would give a compiler error
```

2.7 Arithmetic Structures

If a structure has fields that are all integers or all floats, it is a special struct called an *arithmetic structure*. These structures can be used like vectors and matrices. The `+` and `-` operators can add or subtract two structs component-wise, `*` and `/` can scale a struct by a scalar, and `*` can also take the dot product of two structs. In all cases, only one type may be involved: integer structs cannot be mixed with floating point structs, and structs of different lengths cannot be operated on together.

```
struct vector = {x : float, y : float}
v : vector = {1.0, 2.0}
u : vector = {3.0, 4.0}
w : vector = v + (3.0 * u)
printf(w.x)   // prints "10"
printf(v * u) // prints "11"
```

In addition, the matrix multiplication operator `**` only works on arithmetic structures. The expression `A ** B` can have two meanings. If `A` and `B` are both of length n^2, the result is the matrix product of `A` and `B`, where fields are assumed to be listed by the first column, then the second, and so on. If `B` is of length n while `A` is of length n^2, then `B` is instead treated as a column vector, and the result is a column vector.

```
struct mat2 = {a11 : float, a21 : float, a12 : float, a22 : float}
struct vec2 = {x : float, y : float}
ccw : mat2 = {0.0, 1.0, -1.0, 0.0}
cw : mat2 = {0.0, -1.0, 1.0, 0.0}
p : vec2 = {3.2, 1.1}
q : vec2 = cw ** ccw ** p
// q is now {3.2, 1.1}
```

2.8 Array Iteration

Often, it is useful to apply a function or an operation to a group of elements in an array. SOS features a powerful, concise notation for iterating on arrays. For any operator besides `@` and `of`, if one of the arguments is an array, the operation is performed on each element in turn, and an array is returned. This can even be done many times in the same operation, in which case the first operand is always fully expanded before the second.

```
A : array int = 1 + [2, 3] // = [3, 4]
B : array array int = [1, 2] + [3, 4]
// = [1 + [3, 4], 2 + [3, 4]]
// = [[4, 5], [5, 6]]
C : array array int = [[1, 2], [3, 4]] + 5
// = [[1, 2] + 5, [3, 4] + 5]
// = [[6, 7], [8, 9]]
```

In addition, if a function is applied to an argument that is an array of the expected argument type, the function is applied to each element of the array, and the array of results is returned. Unlike with operators, this cannot go more than one level deep. However, it can simultaneously iterate over two arrays. In this case, the resulting array is of the same length as the first iterated array, so care must be taken to ensure that the array lengths are appropriate.

```
sum_three : (a: int, b: int, c: int) -> int = a+b+c
A : array int = sum_three([1, 2], 3, [4, 5, 678])
// A = [8, 10]
```

2.9 Scope and Recursion

In SOS, not all variables may be accessed in all contexts. We refer to the variables that can be accessed as the *scope* of a certain context. Outside of a function, any variable previously defined that is also not in a function is in scope. Inside of a function, only the function's arguments and any locally defined variables are accessible.

```
a : int = 0
f : () -> int = a // this is invalid
g : () -> int = (b : int = 3); b
b = 3 // this is invalid
```

If a variable is re-defined, the new definition obscures any previous definitions in the same scope.

```
a : int = 0
a : float = 1.0
// the integer a is now inaccessible
```

The only exception to these scoping rules are function names. Functions that are defined (not function variables) are accessible *anywhere* in a program after they are defined, including in their own definitions. This means that a function may recursively call itself.

```
ints : (n : int, i : int) -> array int =
    if i < n then [i] @ ints(n, i+1)
    else []
A : array int = ints(10, 0)
// Gives the array [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

(Note: while concise, this algorithm is not ideal because each `@` allocates a new array. There are better ways to implement this same function without this problem).

2.10 Advanced Details

While the following sections describe the majority of SOS, there are some significant details that are omitted for the sake of being concise. This section intends to cover all these details that are not necessary for most code but can provide some extra functionality.

2.10.1 Function References

The first detail is just an edge case not mentioned before: the functions `print`, `printf`, `copy`, and `free` are all more limited that other functions. They can never be stored as a reference, passed to a function, or used in array iteration. This is because they are handled internally differently from other functions. If these functionalities are required, a "wrapper" function may be defined, and then used as normal.

2.10.2 Value Access

In most cases, array access, struct access, and function application are done directly from the variable names. However, there are cases where one may want these features without storing a value with a name. In addition to access from names, all three of these statements may chained from any other:

```
f : () -> array point = [{1.0, 3.0}]
printf(f()[0].x) // This is valid!
```

However, in any other context, these operations may not be done directly. In these cases, to make the syntax unambiguous, the value access modifier `$( )` must be used. For instance:

```
printf($(p : point = {1.0, 3.0}).x) // prints "1"
xx : float = $(p+p).x               // xx = 2
```

This should usually be used as a last resort, as it leads to messy code and may hide memory allocation.

2.10.3 Starting Expressions

Not all expressions can form a complete statement on their own. Only variable definition, assignments, function application, and if/else statements are allowed as statements. No other statement would do anything, so this should not affect any actual programs. However, it means that a statement like `a+1` on its own would give a syntax error.

2.10.4 Reserved Identifiers

The following is the list of strings that can never be used as identifers. In addition, any defined function names become reserved identifiers.

`print printf copy free of if then else array func true false import`

2.11 Import and Standard Libraries

To increase our language's extendibility, we implement one naive import which works similar to #include in c. We encourage users of our language to use this feature to create more powerful and complicated projects.

The syntax is as follows:

```
import math.sos
```

We have written several standard libraries regards of math and graphics, just to make life easier. Please check the lib folder for more details.

2.12 Setting up Environment

All the dependencies needed are inside of the Dockerfile we provided. There are two ways to get the docker image: build or pull. Either way will get the same docker image on your computer. The docker image is 7GB.

```
./docker_image_fetching.sh pull # or
./docker_image_fetching.sh build
```

Then, you could connect to the docker environment using the script below. (It will pull the new docker image if you haven't done that yet.)

```
./docker_connect.sh
```

Then, compile everything needed by make.

```
make
```

After writing any *.sos scripts, just run the shell script below to get one executable and run it automatically:

```
./compile_exec.sh *.sos
```

3 SOS Style Guide

The SOS language has a very flexible syntax, which gives the programmer many options for enhancing readability. However, it also allows for quite monstrous code if used poorly. This style guide is meant to establish a standard for readable SOS.

3.1 Declarations

Include spaces before and after the `:`, `->`, and `=`. If the function is short, then it should be on the same line. Otherwise, the expression should start on the next line and be indented by four spaces.

```
var : type = expr...
fxn : (arg1: type, arg2: type, ...) -> type = expr
fxn : (arg1: type, arg2: type, ...) -> type =
    long expr...
```

3.2 Expressions

Most of an expression should fit on a single line. If a expression that should be a single line is very long, any additional lines needed should be indented by four spaces. The expressions that do not require indenting are if/else expressions and sequencing.

For sequencing, two short expressions may share a line. In this case, the `;` should be preceded and followed by a space. Otherwise, the second expression should be on a new line at the same indentation, while the semicolon is at the end of the first expression.

For if/else, the then expression and the else expression may be on their own lines. If so, the `then` may share a line with `if` or the beginning of the then expression, and the `else` may share a line with the beginning of the else expression. The `if` should always share a line with the if expression.

```
if expr then
longer expr
else longer expr
```

3.3 Spacing

Spaces between identifiers and keywords are encouraged in most cases. Using more than one space is generally undesirable unless this is used to align similar elements in different lines for readability. No space may be used for simple arithmetic. The main exceptions are struct and array access, function application, and unary operators, where the identifier and symbol should always be directly adjacent.

```
struct.field       // not struct . field
array[element]     // not array [ element ]
function(args...)  // not function ( args )
```

Statements should all be on separate lines, and should be grouped into related blocks of statements that are separated by a blank line. For instance, several associated variable definitions may be on adjacent lines, followed by a blank line.

3.4 Identifiers

Identifiers should be lowercase, and can either uses underscores or camel case to distinguish words (camel case is preferred for graphics calls, but discouraged elsewhere).

4 Language Reference

4.1 Introduction

This manual describes the SOS syntax and the meaning of SOS statements. It is intended to be a complete description of the language features.

4.2 Lexical Conventions

A program in SOS (Sad Oblique Shapes) is first interpreted by parsing it as a string of *tokens*. The following subsections describe exactly what tokens are allowed in SOS.

4.2.1 Identifiers

Most tokens in SOS are identifiers, which refer to types, variables, and functions. An identifier is a string containing only letters, numbers, and underscores, beginning with a letter. The following identifiers are predefined and cannot be overwritten:

```
int  float  bool  void  copy  free  true  false
```

In addition, the following strings are keywords that can never be used for identifiers:

```
if  then  else  of  alias  struct  array  func
```

Keywords, like identifiers, are treated as one token.

4.2.2 Constants

Any string consisting of only numbers and at most one decimal point is a numeric constant. Constants are interpreted as a single token. In addition, either `e` or `E` may be used to indicate an integer exponent for a number represented in scientific notation.

Some `int` literals: `0 3 -7 4378`
Some `float` literals: `0.0 -5.778 4.0e2`

4.2.3 Comments

`/*`, `*/`, and `//` are special tokens that are used to indicate comments. They are not passed as tokens by the scanner, but rather denote a segment of text to be ignored.

4.2.4 Symbols

All other characters are interpreted as symbols, like operators and delimiters. Symbols are interpreted maximally, e.g. `<=` is always interpreted as one symbol, not a separate `<` and `=`. The following is the complete list of symbolic tokens:

```
+   -   *   /   %   **   @
=   ==  !=  <   >   <=   >=
!   ||  &&  ,   :   ;    ->
(   )   [   ]   {   }    $
```

4.2.5 Whitespace

Whitespace is used to separate tokens. In many cases, it is not necessary, but certain tokens (for instance, two identifiers) must be separated by whitespace or else they will be interpreted as one. It is generally advised to separate all tokens with a space for readability. In addition, all whitespace–spaces, tabs, and newlines–are interchangeable.

4.3 Types

In SOS (Stylistically Observed Structures), every variable is strictly typed. There are some options for casting between types. This subsection outlines what types exist and what casts are possible.

4.3.1 Basic Types

SOS has three basic types: `bool`, `int`, and `float`. `bool` is a single bit representing either `true` or `false`. `int` represents a 32-bit integer. `float` represents a 32-bit floating point number. All other types are built using these three types.

All the basic types are stored as values and passed by value.

The `void` type can be seen as the fourth basic type. It can only be used to indicate the return type of a function.

4.3.2 Derived Types

A new type can be created in three ways: as an array, as a structure, or as a function. An *array* is a block of memory with many variables of a specific type, along with an integer giving the length of the array. A *structure* consists of some number of *fields*, each with their own identifier and type. This allows associated information to be stored and processed together. A *function* is a stored expression with one or more *arguments* that can be used within the expression.

All the derived types are stored as pointers and passed as pointers. A simple assignment will replace the initial pointer, instead of writing to the location of the pointer. There are ways to modify arrays and structures after they are defined, but a function is immutable.

4.3.3 Type Identifiers

The names `bool`, `int`, `float`, `void` are all the identifiers for their respective types.

A `struct` type is referred to by the name given in its struct definition, and `struct` itself is not used as an identifier.

The string `array` *type-name* is the identifier for an `array` of a certain type. This can be nested, for example `array array int` is an array of integer arrays.

A function type is represented by `func` *arg-type-1* `, ...  ->` *return-type*, which is the type of a function with argument(s) of the specified type(s) and the specified return type. This notation is not commonly used, but is required to use a function as the argument in another function, for instance.

4.3.4 Type Conversions

Many inbuilt types will automatically be converted to a different type when required. This will always happen as late in any calculation as possible. In each of the following subsections, all the possible conversions from a given type will be listed, as well as their meanings. User-defined structs and arrays are never converted. In most cases, such as variable assignment or function application, one statement will expect a certain type, and the required cast will be clear. The only exception is comparisons, where in general `float` is preferred over `int` which is preferred over `bool`, and `point` is preferred over `vector`.

- **bool**

  A `bool` can be cast to an `int` or `float`, by the map `false` $\to 0$ and `true` $\to 1$.

- **int**

  An `int` can be cast to a `bool`, where 0 maps to `false` and all other values map to `true`.

  An `int` can also be cast to a `float` by the injection $x \to x$.

- **float**

  A `float` can be cast to a `int` by truncation. This is not to be confused with the floor operation. For instance, 0.5 and -0.5 both truncate to 0. It can also be cast to a `bool` in the same way as an `int`, although it should be noted that this may have unexpected behavior due to floating point precision.

- **struct**

  Any structs with the same fields in the same order may be cast interchangeably. However, this cast will not occur when accessing the struct's fields, so make sure that the fields are consistent with the specific struct name assigned to any given variable.

- **array, function**

  Casting never occurs for arrays or functions.

- **void**

  All types can be cast to `void`, meaning that their types are simply discarded. `void` cannot be cast to any type besides itself.

4.4 Syntax and Expressions

An SOS (Shapes of Sorrow) program is formed by a series of *statements*. This subsection outlines all possible SOS statements and their meanings.

4.4.1 Statements

There are three forms of statements in SOS: *type definitions*, *function definitions*, and *starting expressions*. Starting expressions and function definitions can contain other expressions, while type definitions are always concrete. Only a starting expression may form a statement; all other expressions can only be used within a starting expression or function definition.

Throughout this subsection, the production rules for the SOS grammar will be listed in the following format:

symbol:
production ...

Italic characters represent another symbol, typewriter strings and symbols represent specific tokens. Symbols may be followed by numbers to distinguish them. Ellipses `...` are indicating that more symbols of the same form can be given. Here is the production rule for a statement:

statement:
type-definition
function-definition
starting-expression

In addition, the following production rule is important:

expression:
starting-expression

Meaning any starting expression may be used in any place that calls for an expression.

4.4.2 Type Definition

The only type of statement that is not an expression is a type definition. As such, these statements do not include any expressions and cannot be included in any other expressions.

There are two ways to make a new type identifier:

type-definition:
`alias` *id* `=` *typeid*
`struct` *id0* `=` { *id1* `:` *typeid1* `,` `...`}

The first format defines a new type name that is an alias for an existing type. Internally, this means that every instance of the new type name will simply be treated as the old type name. It is possible for the *typeid* to be a combination of types, like an `array` or a function type.

The second format defines a new `struct` type with name *id0* and fields named *id1*, ...

A type id is given by the following production:

typeid:
id
`array` *typeid*
`func` *typeid1, ...* `->` *typeid0*

Which indicates, in order, a named type (such as a primitive type, alias, or struct), an array type, or a function type.

```
alias year = int
struct person = {id: int, age: int}
```

4.4.3 Function Definition

A new function can be created using the function definition statement:

function-definition:
id0 `:` (*id1* `:` *typeid1*, ...) `->` *typeid0* = *expression*

The function is named *id0*, which cannot be an already defined function name. The function's type is determined by the argument types and return type *typeid0*. Notice that the `func` keyword is not used to define a function. These keyword is instead use to indicate variables that store pointers to pre-defined functions.

4.4.4 Declarations and Assignments

In SOS, there is no such thing as a declaration without an assignment. That is to say, when a variable is introduced it must also have a value attached. Any declaration defines a *name* by which a variable can be referred to, the *type* of the variable, and the *value* of the variable.

Declarations are of the form:

starting-expression:
id `:` *typeid* = *expression*

This defines a variable of the given type and assigns its value to the given expression. The variable can be of a functional type, but this can only be used to create a new reference to an existing function, not a new function altogether. The expression must be of a type that can be cast to the specified type.

If a variable or function has already been declared, it can be re-assigned as follows:

starting-expression:
id = *expression*

A function cannot be re-assigned this way, although a pointer to a function may be.

```
var1 : int = 3
double : (n : int) -> int = 2 * n
double_double : func int -> int = double
var1 = double(var1)
```

4.4.5 Operators

The main tools for building meaningful expressions from other expressions are operators. The type of an operator expression depends on the types of the expression(s) it acts upon, which will be specified later. All operator expressions are of one of two forms:

expression:
unary-operator expression
expression binary-operator expression

Logical Operators

The logical operators are `!`, `||`, and `&&`. `!` is a unary operator which takes a `bool` and returns its negation. `||` and `&&` are binary operators on two `bool`s representing logical OR and AND, respectively. They are both left associative.

Comparison Operators

The comparison operators are `==`, `!=`, `<`, `>`, `<=`, and `>=`. All are binary operators that take two expressions of the same type that return a `bool`. `==` returns `true` if the expressions have the same value, `false` otherwise; `!=` does the opposite. Only primitive types and arithmetic structs (see below) can be equated. On `int`s and `float`s, the other four operators represent less than, greater than, less then or equal, and greater than or equal.

Mathematical Operators

The mathematical operators are `+`, `-`, `*`, `/`, and `%`. They can act on integers or floats. If one argument is a float, the other is promoted to a float, if necessary. The exception is the modulo operator % which can only act on integers. All these operators are left-associative.

In addition, `-` can be a unary operator, which returns the negation of its arguments.

Arithmetic Structures

Mathematical operators can also be used on *arithmetic structures*, which refers to any struct whose fields are all floats or all integers. For two arithmetic structures of the same size, `+` and `-` perform component-wise addition or subtraction, and `*` computes the dot product, that is, the sum of the component-wise products.

An arithmetic struct can be scaled by an number using `*` or `/`.

In addition, the matrix multiplication operator, `**`, is reserved for arithmetic structs. In an expression `A ** B`, `A` must have a square number of fields, it is treated as a square matrix, listed by the first column from the top down, then the second, and so on. `B` can either be struct of the same size, which is treated as another matrix, or of the square root of the same size, in which case it is treated as a column vector. Unlike the other mathematical operators, `**` is right-associative.

Array Operators

Arrays have two unique operators, the `of` operator and the concatenation operator, `@`. For two arrays `A` and `B`, `A @ B` indicates a new array with A's elements followed by B's. If `n` is an integer and `A` an array, `n of A` indicates A concatenated with itself n times. Both are left-associative.

Sequencing

`;` is a special binary operator that allows for sequencing. Both expressions are evaluated, and the value of the second expression is kept as the value of this expression. This is left associative.

4.4.6 Function Application

Function application is used to execute the expressions associated with a given function. It is written as:

starting-expression:
value (*expression1*, ...)

Where *value* is a special symbol of one of the following forms:
value:
id
value `.` *id*
value `[` *expression* `]`
value (*expression1*, ...)
`$(` *expression* `)`

A value symbol is used to contain an expression that can be accessed, either through function application, struct access, or array access. The `$` access symbol allows for access into other expressions, such struct arithmetic. In the case of function application, *value* should resolve to a function with the given number of arguments, in which case this statement evaluates the function and returns its value, which will be of the function's return type.

```
double : (a : int) -> int = a * 2
four : int = double(2)
```

4.4.7 Conditionals

The conditional expression is formated as follows:

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

Where *expression1* must resolve to type `bool`, and *expression2* and *expression3* must resolve to the same type. The whole expression will have this second type, and it will have the value of *expression2* if *expresion1* is `true`, and the value of *expression3* otherwise.

```
x : int = 100
var3 : int = if x>0 then 1 else 0
```

4.4.8 Construction

An array can be created using the array construction expression:

expression:
`[` *expression-1* `, ...]`

Where each expression must resolve to the type of the first expression. The whole expression will then be an array of that type. If no expressions are given, an empty array will be given, which can be cast to any array type.

A struct can be created using either struct construction expression:

expression:
`id{`*expression1*`, ...}`
`{`*expression1*`, ...}`

In the first case, the struct type is given by the *id*, otherwise it is determined implicitly. These are usually identical, except for the fact that the fields of an anonymous struct cannot be accessed until it has been cast to a named struct.

```
arr : array int = [1, 3+5, 7, 9]
struct person = {id : int, age : int}
tom : person = {12345, 45}
```

4.4.9 Variable Reference

Variable reference is always a terminal expression. It is notated simply as:

expression:
id

And has the same type as the specified variable. In addition, the fields of a struct can be referenced with dot notation:

expression:
value`.`*id*

An element of an array can be accessed by index, where index 0 represents the first element. The syntax is as follows:

expression:
value`[`*expression*`]`

Both struct access and array access can also be used in place of *variable-name* in an assignment expression.

starting-expression:
value`.`*id* `=` *expression*
value`[`*expression1*`]` `=` *expression2*

4.4.10 Literals

Numeric literals are terminal expressions. They are expressed in base 10. Without a decimal point or scientific notation exponent, they will be interpreted as `int`s, otherwise they will be interpreted as `float`s. Commas cannot be used in any way, either for separating thousands or for the decimal point.

expression:
integer-literal
float-literal
bool-literal

4.4.11 Parentheses

Parentheses can be used to clarify or alter the order of operations. A parenthetical expression is as follows:

expression:
`(` *expression* `)`

The whole expression has the same type as the inner expression.
Notice the difference with the value expression that uses the access symbol: `(x).y` is not a valid struct access, while `$(x).y` is (assuming `x` is a struct with field `y`).

4.4.12 Array iteration

SOS contains several powerful notations for performing operations on all elements of an array. For any operator (besides sequencing), if one of its arguments is an array, the operator will be performed on each element of the array, returning a new array with the results. This can be done recursively, with arrays within arrays or arrays on both sides of the operator. The left operand is expanded first, so it will give the length of the resulting array.

```
A : array array int = [1, 2] + [3, 4]
// = [1 + [3, 4], 2 + [3, 4]
// = [[4, 5], [5, 6]]
```

Arrays can also be iterated on a function. If a function has an argument of type t and the argument given is an array of t, the function is applied to each element of the array, and an array of the results is returned. If several iterated arrays are given, they are all iterated on simultaneously, up to the length of the first iterated array. Unlike with operators, this not recursive. If the function returns void, the iterated function also returns void (not an array of void, which is not possible).

```
add : (a: int, b: int, c: int) -> int = a + b + c
B : array int = add([1, 2], 3, [4, 5, 1000])
// = [8, 10]
```

4.4.13 Scope

The *scope* of an expression refers to the variables that can be accessed within that expression. Functions that are defined using the function definition are *global*, meaning they can be referred to at any point after they are defined, even within their own definition. Variables are always *local*. This means that a function

may only access the variables given by its formaal arguments. Any variable defined within a function may be accessed for the remainder of that function. This means that variables defined outside of any function are not accessible within a function. The only expression that introduces a local scope is the if/else statement: variables defined within the then or else expressions will not be accessible after the statement or in the other case.

Variables may always be re-named, even if they are already defined within a certain scope. Doing so prevents the original variable from being accessed, so this should be avoided. In separate scopes, variables can share names with no confusion.

In the following example, notice how `pow` can refer to itself within its defintion. There is no conflict between the argument `n` and the variable `n`.

```
pow : (n : int, x : int) -> int =
    if x == 0 then 1 else n * pow(n, x-1)
n : int = pow (3,2)
```

4.4.14 Operator Precedence

In expressions without parentheses, operators of higher precedence are interpreted first. Furthermore, in a chain of operators of the same precedence, the operators are interpreted either left-to-right or right-to-left, depending on if the operator is left- or right-associative. The following table completely describes operator precedence.

Symbol	Associativity	
`( )`	None	(highest precedence)
`.`	Left	
`!`	Right	
`[ ] { } ( )`	None	(parens for function application)
`* / %`	Left	
`**`	Right	
`+ -`	Left	
`@`	Left	
`of`	Left	
`< > <= >=`	Left	
`== !=`	Left	
`&& ||`	Left	
`=`	Right	
`;`	Left	
`,`	Left	
`if then else`	None	(lowest precedence)

4.4.15 Importing Libraries

SOS has only one preprocessor directive: lines with the form

```
import filename.extension
```

at the beginning of a file is replaced by the file *filename.extension*. The characters in the name of *filename* must not include newline or `/*`. Before scanning the files in OS, SOS will first try to find if there is any standard library with the *filename*. Even though the file imported is usually a **.sos* file, extension is still required.

As we are going to replace the line, declaring a new variable that is already in imported file is prohibited, but you could override it when needed. Also, the import graph cannot contain a cycle, for example, import A in file B and import B in file A is prohibited.

4.4.16 Comments

Any text written between the symbols `/*` and `*/` will be ignored. Comments can be nested this way. Additionally, any text between the symbol `//` and the next newline will be ignored.

```
// a single line comment
/* a multi line /* nested */ comment */
```

4.4.17 Complete grammar

typeid:
id
`array` *typeid*
`func` *typeid1, ...* `->` *typeid0*

statement:
type-definition
function-definition
starting-expression

type-definition:
`alias` *id* `=` *typeid*
`struct` *id0* `=` { *id1* `:` *typeid1* `,` ...}

function-definition:
id0 `:` (*id1* `:` *typeid1*, ...) `->` *typeid0* = *expression*

value:
id
value `.` *id*
value `[` *expression* `]`
value (*expression1*, ...)
`$(` *expression* `)`

starting-expression:
id `:` *typeid* = *expression*
id = *expression*
value`.`*id* `=` *expression*
value`[`*expression1*`]` `=` *expression2*
value (*expression1*, ...)
`if` *expression1* `then` *expression2* `else` *expression3*

expression:
integer-literal
float-literal
bool-literal
unary-operator expression
expression binary-operator expression
`(` *expression* `)`
`id`{*expression1* `,` ...}
{*expression1* `,` ...}
`[` *expression-1* `,` ...]

id
value . *id*
value [*expression*]
starting-expression

4.5 SOS Standard Library

The library functions are written with the SOS (SOS Object System) language in separate files which can be used with `import`. Certain library functions employ the external OpenGL library for its graphics utilities, and support extensive graphical operations.

4.5.1 Math

The file `math.sos` contains useful mathematical functions.

`sqrt : float -> float`
Computes the square root of a number.

`sin, cos, tan : float -> float`
Compute the trigonometric functions sine, cosine, and tangent for an angle in radians.

`asin, acos, atan : float -> float`
Compute the inverse trigonometric functions, returning an angle in radians.

`toradians : float -> float`
Converts an angle from degrees to radians.

`floor, ceil : float -> float`
Computes the smallest integer greater (floor) or the largest integer less than (ceil) a given float, returning the result as a float.

`frac : float -> float`
Computes the fractional part of a number from 0 (inclusive) to 1 (exclusive).

`max, min: float, float -> float`
Computes either the maximum or minimum of two numbers.

`clamp : float, float, float -> float`
`clamp(x,m,M)` clamps x to the range [m, M]

`abs : float -> float`
Computes the absolute value of a number.

`modf : float, float -> float`
Like `frac`, but for an arbitrary range. `frac(x, m) = y` means that $0 \leq y \leq m$ and $x + nm = y$ for some integer n.

4.5.2 Point

The file `point.sos` contains functions for dealing with points and vectors.

```
struct point = {x:  float, y:  float}
struct point3 ={x:  float, y:  float, z:  float}
```

`sqrMagnitude, magnitude :  point -> float`
Determines the squared magnitude or magnitude of a point (i.e., the distance to the origin).

`sqrDistance, distance :  point, point -> float`
Computes the squared distance or distance between two points.

4.5.3 Vector

For now, `vector.sos` just contains the useful alias:

```
struct vector = {x:  float, y:  float}
```

4.5.4 Shape

The file `shape.sos` contains functions for dealing with collections of points, such as lines, curves, and shapes.

```
alias path = array point
alias shape = array point
```

`copy_path :  path -> path; free_path :  path -> void`
Convenient shorthand for copying or freeing a path and all its points.

`append :  path, path, float -> path`
Appends one path onto another. The float parameter gives a distance below which the last point of the first path and first point of the last point will be merged.

`reversed :  path -> path`
Returns a reversed version of a path without changing the original.

`reverse :  path -> void`
Reverses a path in place.

4.5.5 Color

The file `color.sos` contains functions for dealing with color.

```
struct color = {r:  float, g:  float, b:  float, a:  float}
alias colors = array color
```

`rgb :  float, float, float -> color`
Creates a color with the given red, blue, and green channels (all from 0 to 1). Assumes alpha = 1.

`hsv : float, float, float -> color`
Creates a color with the given hue, saturation, and value (all from 0 to 1). Assumes alpha = 1.

4.5.6 Affine

The file `affine.sos` contains functions for manipulating affine transformations.

```
struct mat2 = {...} // Matrix fields are floats labeled .aij
struct mat3 = {...}
alias affine = mat3
```

`affine_mul : affine, point, float -> point`
Applies the affine to the given point with specified homogeneous coordinate.

`scale : float, float -> mat2`
Returns a matrix representing a scale by the given factors.

`rotation : float -> mat2`
Returns a matrix representing a counterclockwise rotation, with the angle given in radians.

`translate : float, float -> affine`
Returns an affine representing a translation by the given vector.

`rotation_aff, scale_aff ... -> affine`
Creates either a rotation or scale matrix, and creates a corresponding affine.

4.5.7 Renderer

The file `renderer.sos` contains the main functions for interfacing with OpenGL.

`struct canvas = {width: int, height: int, file_number: int}`

`drawPoints : path, colors -> void`
Draws the points on the path as dots with the given colors.

`drawPath : path, colors, int -> void`
Draws the path with connected lines. The int indicates how to interpolate colors along the lines.

`drawShape : path, colors, int, int -> void`
Draws the path as a closed loop. The ints indicate the color interpolation and whether to fill the shape.

`startCanvas : canvas -> void`
Sets up the given canvas for rendering. None of the draw methods can be called until this method has been called.

`endCanvas : canvas -> void`
Renders and saves a canvas as an image.

4.5.8 Array

For now, `array.sos` just contains this useful array generator:

`ints : n -> int array`
Creates an array of the n integers $0, 1, \ldots, n$ in order.

4.5.9 Random

For now, `random.sos` just contains an implementation of the Wichmann-Hill random number generator:

`struct rng = {s1: int, s2: int, s3: int}`

`randf : rng -> float`
Creates a random number from 0 to 1 and updates the seed states of the given rng. The seed values must all be non-zero.

5 Project Plan

SOS language was designed and developed as a project for the class "Programming Languages and Translators - 4115" taught by Prof. Stephen Edwards in Spring 2021 at Columbia University. We host our project in a Github repository to use Git as a version control system, with Project Kanban enabled to manage the project. We also had bi-weekly Zoom meetings on Friday to get updates from team members and ask questions, then, we held meetings twice a week before any important deadlines. Besides Zoom meetings, WhatsApp is our main instant messaging application used. We sent importatnt updates, bugs found and some funny jokes about our own language via WhatsApp. We also meet our TA Harry Choi one week before the due days to report our progress and ask clarification questions.

5.1 Contributors

The members of the project team and their roles are listed below:

- Tojo Abella: Test Engineer
- Sitong Feng: Project Manager
- G Pershing: Language Guru
- Sheron Wang: Systems Architect

5.2 Style Guide

Ocaml: Ocaml was the main source that made up our underlying architecture to compile our language. We strictly follow the convention of indenting with 2 spaces, indenting nested let expressions, and commenting above the code. We also tried to avoid verbose code expressions.

C: Even though C only contributes to 5% of our program, we still set up several basic style instructions such as 4 spaces as tab, left curly bracket on the same line but with one space before that, and new line for else.

SOS: As we are using our own language to write standard library, we set up a detailed style guide for SOS, please check SOS Style Guide after the tutorial section for more details.

5.3 Timeline

- Feb 3
  - Project Proposal Complete
- Feb 24
  - Scanner, Parser, and LRM Complete
- Mar 24
  - Hello World milestone
    - work entails codegen.ml, semant and more
- April 15
  - Add third-party library support
    - include `OpenGL` for graphics operation
- April 26
  - Language Project Complete

5.4 Future Work

The SOS language aims to develop more powerful and easy to use graphics features in the following aspects:

- Incorporate more `OpenGL` utilities such as varied line type
- Add basic built-in shapes for creating composite shapes
- Add stdlib function `drawCurve` for more smooth paths
- Add memory management
- Add function scope

More advanced features will include:

- Add 3D Object/Plot Diagram support
- Link other third-party APIs
- Add animating function and gif file support
- Allow real-time interactivity

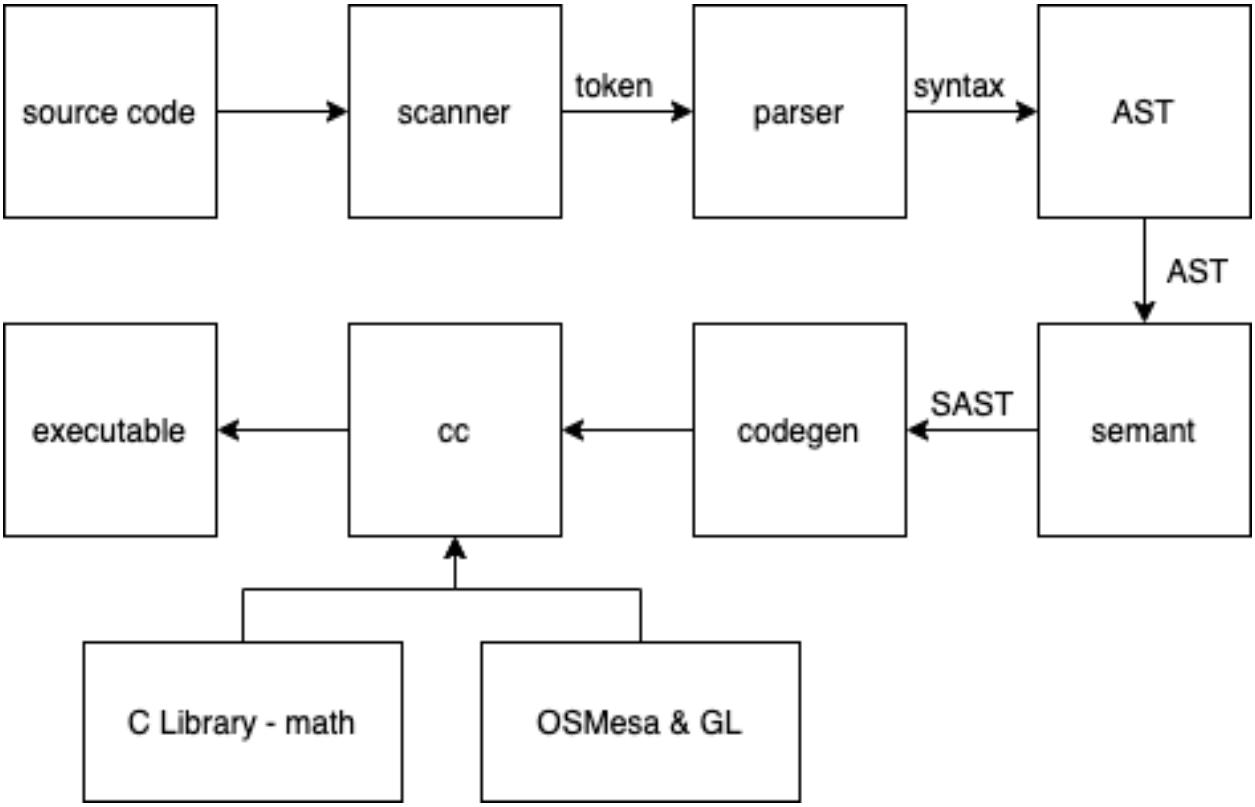
5.5 Git Log

Please check https://github.com/tojoabella/SOS, if needed.

6 Architectural Design

6.1 Block Diagram

SOS has a relatively traditional scanner, parser, ast, semant and codegen architecture. Then, C math library, OpenGL and Off-Screen Rendering Mesa library are added to cc compiler to compile to one executable file.



sos.ml by Sheron. The top level of our compiler, serves as one entry point for the source code to go through scanner, parser, AST, semant, codegen and then generate LLVM IR.

scanner.mll by Sheron and G. Receives the source code and uses OCaml Lexer to process it to tokens.

parser.mly by G and Sheron. Reads from scanner, generate AST from tokens. Reject when there is an error.

AST

ast.mli by G and Sheron. This file only preserves the abstract syntax tree structure.

astprint.ml by G. Contains helper functions to generate code from one AST and print it out. Use flag "-a" in compiler to call the function while compiling.

semant.ml by G, Tojo and Sheron. Receives one AST and check if it is semantically correct. We also store the information of all external functions at the beginning of the semant to check, and pass to codegen for further use.

SAST

sast.mli by G. This file only preserves the semantically checked abstract syntax tree structure.

sastprint.ml by G. Contains helper functions to generate code from one SAST and print it out. Use flag "-s" in compiler to call the function while compiling.

Codegen.ml by G, Sheron and Sitong. This file contains all the instructions to generate LLVM IR and call external functions from C and OpenGL libraries.

External Functions

util_math.c by Sheron. As we usually call functions of C Math library directly, there is only one function called "toradians" in this file.

util_opengl.c by Tojo, Sheron. Takes in pointers allocated in SOS and call the function such as start and stop render or draw points, path and shape. Standard libraries written in SOS provide more user friendly wrappers of these C functions. For the writing to file and canvas initialization part, we consulted the code from mesa-demos, an official collection of OpenGL / Mesa demos and test programs.

Compile

Makefile by G, Sheron, Tojo and Sitong. Make, test and clean SOS native file.

Dockerfile by Sheron. To install all dependencies for compiler and OpenGL based on Ubuntu 20.04 LTS. See next subsection for more details.

shell scripts by G and Sheron. Those are for pulling and connecting docker image, or compiling executables.

miscellaneous tests by Tojo. Check Test Plan section for more details.

Standard Libraries by G, Tojo, Sheron and Sitong. All written in pure SOS and could be imported. Extends the functionality of SOS language, and provides wrappers to external functions. See LRM for more details.

6.2 External Libraries

We link C Math Library and Mesa Library to LLVM IR code in CC compiler directly.

For OpenGL, we are using The Mesa 3D Graphics Library, which is one open source software implementation. We choose Mesa because it has one off-screen rendering interface. Because it is off-screen, it renders into main memory without any sort of window system or operating system dependencies. Thus, it could run inside of a docker image provided by us smoothly. In the end, SOS writes the buffer in memory to a file when user decides to end rendering, which is more practical than window system rendering.

The Dockerfile provides instructions of how to download and compile all dependencies needed in Ubuntu 20.04 LTS. Most of the dependencies could be installed by package management tools such as apt-get, pip and opam. However, we need to compile Mesa library by downloading and compiling. There are two choices to play with the docker image: compile (might cost about 1 hour on my computer) or pull(around 7 GB).

7 Test Plan

The end-to-end integration tests can be found in the test/ directory. An an end-to-end integration test suite, it was used to rigorously test the functionality of SOS, including lower level lexical, syntactic, and semantic checks.

A complete set of tests can be found in Appendix 9.4 Integration Tests Files (Negative tests) and Appendix 9.4 Integration Tests Files (Positive tests).

Tojo, Sheron, and Sitong created the tests/test-* and tests/fail-* test cases to test specific features of SOS. G, Sitong, and Tojo created sample programs as end-to-end tests.

7.1 Test Automation

While in the docker environment, running ./testall in the SOS directory will run all of the tests and say whether each test passed with "OK" or failed with "FAILED". Each test case that should successfully pass and produce output follow the naming pattern of using test-*.sos for the test program and a corresponding (in file name represented by *) test-*.out for the expected output of the program. Similarly, each negative test case used fail-*.sos for the test program that should fail and fail-*.err (instead of .out) for the expected error message.

For our integration tests, we modified the testall.sh script provided by Professor Edwards for the MicroC compiler. The script works by compiling and executing all of the test-*.sos and fail-*.sos programs. For each test case (test and fail), the script compares the produced outputs (.out) or errors (.err) with their corresponding references made by us, to see if what was produced matches their expected values. If the produced output or error files fail to match what is expected, the script will print out 'FAILED' next to the test name along with an error message. A more detailed error message can be found by looking in testall.log. One can refer to either the produced .out/.err file and compare it to the reference, or the .diff file, which summarizes the differences between the two files. If the test is successful, the script prints an 'OK', and the compiled .out/.err files are removed from the directory for cleanliness.

For larger end-to-end tests, (i.e. sample programs), we created a separate directory called "sample_programs" to hold these programs. These tests can be run using the compile_exec.sh script, which takes the file path to the program to run as an argument.

Example: ./compile_exec.sh sample_programs/dragon.sos

testall.sh and compile_exec.sh can be found in Appendix 9.4 testall.sh and Appendix 9.4 compile_exec.sh, respectively.

7.2 Testing Suite

See Appendices 9.4 for the testing suite.

7.3 Sample Programs

7.3.1 Square: test-helloworld.sos

```
import renderer.sos

p1: point = {-0.5, -0.5}
p2: point = {-0.5, 0.5}
p3: point = {0.5, 0.5}
p4: point = {0.5, -0.5}
point_arr : path = [p1, p2, p3, p4]

c1 : color = {255.0, 0.0, 0.0, 0.8}
c2 : color = {0.0, 255.0, 0.0, 0.8}
```

```
c3 : color = {0.0, 0.0, 255.0, 0.8}
c4 : color = {100.0, 100.0, 0.0, 0.8}
color_arr : colors = [c1, c2, c3, c4]

//create first canvas, draw square, and save and end canvas
canvas1 : canvas = {400, 400, 0}

startCanvas(canvas1)
drawShape(point_arr, color_arr, 0, 1)
endCanvas(canvas1)

//create a second canvas, draw same shape, and close this canvas
//there should now be two images of the same drawing saved in root
   directory
canvas2 : canvas = {400, 400, 1}

startCanvas(canvas2)
drawShape(point_arr, color_arr, 0, 1)
endCanvas(canvas2)
```

See Appendix 9.5.1 for test-helloworld.ll

7.3.2 Dragon: dragon.sos

```
import renderer.sos
import vector.sos
import transform.sos
import array.sos
import math.sos

// Creates a dragon curve of depth n
dragon: (n: int) -> path =
    if n == 0 // Base case
    then [point{0.0, 0.0}, point{1.0, 0.0}]
    else
    // Create two copies of the previous depths
    d1: path = dragon(n-1) ;
    d2: path = copy_path(d1) ;

    // Position d1
    s: float = sqrt(2.0)/2.0 ;
    rotate(d1, toradians(45.0), -1, {0.0, 0.0}) ;
    scale(d1, s, s) ;

    // Position d2
    rotate(d2, toradians(135.0), -1, {0., 0.}) ;
    scale(d2,s,s) ;
    trans(d2, {1., 0.}) ;
    reverse(d2) ;

    // Merge the paths
    r: path = append(d1, d2, 1.0) ;
    free_path(d1); free_path(d2); r

// Creates a rainbow color effect
```

```
rainbow: (r: int, len: int) -> color =
    h: float = (1.0*r)/len ;
    hsv(h, 0.8, 0.8)

// Render a 400px by 400px canvas, name the image pic0
my_canvas: canvas = {400, 400, 0}

// Start render
startCanvas(my_canvas)
d: path = dragon(7)
// Position the curve (0.4, 0.2 is approximately the center of mass of the
//curve for large n)
trans(d, {-0.4, -0.2})

// Draw it
drawPath(d, rainbow(ints(d.length), d.length), 0)
endCanvas(my_canvas)
```

See Appendix 9.5.2 for dragon.ll

7.3.3 Drunk walk: drunk.sos

```
import renderer.sos
import random.sos

n: int = 100
p: path = n of [{0.0, 0.0}]
c: colors = n of [{0.5, 0.5, 0.5, 0.5}]
r: rng = {1,2,3}

drunk_walk : (i: int, p: path, c: colors, r: rng) -> void =
    if i < p.length then
      theta: float = randf(r) * 6.28319;
      d: float = randf(r)*0.1 + 0.02;
      dx: float = cos(theta)*d ; dy: float = sin(theta)*d ;
      p[i] = {p[i-1].x+dx, p[i-1].y+dy} ;
      dc: float = 0.1 ;
      dr: float = (randf(r) - 0.5) * dc ;
      dg: float = (randf(r) - 0.5) * dc ;
      db: float = (randf(r) - 0.5) * dc ;
      c[i] = rgb(c[i-1].r + dr, c[i-1].g+dg, c[i-1].b+db) ;
      drunk_walk(i+1,p,c,r)

    else void


my_canvas: canvas = {400, 400, 0}
startCanvas(my_canvas)

draw_walks : (count: int, p: path, c: colors, r: rng) -> void =
    if count > 0 then
    p[0] = {randf(r)*0.5-0.25, randf(r)*0.5-0.25};
    c[0] = hsv(randf(r), 0.8, 0.8) ;
    drunk_walk(1, p, c, r) ;
```

```
    drawPath(p,c,0) ;
    draw_walks(count-1,p,c,r)
    else void

draw_walks(20, p, c, r)

endCanvas(my_canvas)
```

8 Lessons Learned

8.1 Sheron

Taking PLT is a really pleasant but painful experience to me. I thought that I could handle it at the beginning, while just writing some shitty scanner and parser. However, I got totally overwhelmed after I realized that we need to finish Hello World within one month. I feel like I am one idiot that can work hard on something but not accomplish anything. I got panicked for a while in March, struggled to read the slides and go over the lecture recordings without any progress. As a result, I just went to build the docker file and set up the environment for a while. Even though that part is also tedious, the accomplishment made me come back with a calm mind to deal with all those messy compiler things. If you dive into it, you'll get familiar with it (even though it is still hard). In the end, it is just so delightful to play with our toy language and write standard libraries, so just don't give up!

The lessons I learned are:

- Yeah, I am one idiot.
- Don't panic. Start from things that you could do. It is really easy to give up in the middle.
- GO TO THE OFFICE HOURS. I wish I did that more.
- I am not saying that Professor's lectures are bad, but I did find some online tutorials also helpful - it builds the view of a compiler from a different perspective, which is really helpful.
- It is hard to build one language but fun to play with it.

8.2 G

- Syntax exists for a reason. We set out aiming for a pretty light syntax but found out the hard why very few languages do that. I'd recommend others to aim for their ideal syntax but have some backups in mind: note that more distinct symbols will make parsing much easier.
- Codegen is where a language goes to die. All of your previous mistakes will be revealed to the world. I wish that I had spent some time learning about LLVM before we even started on the reference manual, to know what would be possible down the line.
- Remember that LLVM is designed to work like C. I had the most success when I thought about our features as existing in C. Plus, this will make connecting to C libraries easier!

8.3 Sitong

- This experience of working through the process of how (a simple) programming language is formed, from the stage of ideation to actual usage is so transforming, especially to the way we consider the trade-offs of languages and how to better utilize what we have at hand.
- OCaml is fun until we need to use it extensively for unworldly purposes.
- Finding good teammates is very crucial, especially during the time of crisis. A huge thanks for my teammate's support. For the future, I would wish to discuss the projects in person, as the work can get overwhelming, and makes one(e.g. me) doubt if they really understand anything.

8.4 Tojo

Functional programming was a completely new paradigm for me. Learning Ocaml was difficult. It was so traumatic that I still haven't come to appreciate it. On a brighter note, learning about how languages and compilers work was super interesting. Some project-related lessons:

- Go to office hours. Even after spending a lot of time trying to learn LLVM and going through multiple tutorials, I still did not have the understanding necessary to apply it to our project. It is indeed

possible to struggle for weeks and make absolutely no progress. And you do not have that kind of time to be stuck.

- Get started early and prepare to do a ton of research on your own. The project deadlines are quick, many times quicker than what is covered in class.
- Really understand the different components of MicroC, from the shell scripts to using external libraries.

9 Appendix

9.1 SOS Interpreter

9.1.1 sos.ml

```
(* Top-level of the SOS compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module
   Reference: The MicroC compiler *)
(* Written by Sheron *)

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./sos.native [-a|-s|-l|-c] [file.sos]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename)
   usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Scanner.token lexbuf in
  match !action with
    Ast -> Astprint.basic_print ast
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> Sastprint.basic_print sast
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate
   sast))
    | Compile -> let m = Codegen.translate sast in
  Llvm_analysis.assert_valid_module m;
  print_string (Llvm.string_of_llmodule m)
```

9.1.2 scanner.mll

```
(* SOS Scanner *)
(* Written primarily by Sheron, later polish by G *)

{ open Parser

  let find_file file =
      if Sys.file_exists file then file
      else if Sys.file_exists ("lib/"^file) then ("lib/"^file)
      else raise (Failure ("Could not find file "^file))
```

```
  let import_table = Hashtbl.create 10
}

(* Definitions *)
let digit = ['0'-'9']
let digits = digit+

(* Rules *)

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*"     { comment 0 lexbuf }           (* Comments *)
| "//"     { single_comment lexbuf }            (* Single line comments *)
| "import " ([^'\n']+".sos" as file) {
  let file = find_file file in
  if Hashtbl.mem import_table file then IMPORT [] (* Ignore *)
  else (
  Hashtbl.add import_table file ();
  let read = Lexing.from_channel (open_in file) in
  let parsed = Parser.program token read in
  IMPORT parsed ) }
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACK }
| ']'      { RBRACK }
| '$'      { DOLLAR }
| ','      { COMMA }
| ':'      { COLON }
| '.'      { DOT }
| "->"     { TO }
| '+'      { ADD }
| '-'      { SUB }
| "**"     { MMUL }
| '*'      { MUL }
| '/'      { DIV }
| '%'      { MOD }
| '@'      { CONCAT }
| ';'      { SEQ }
| "=="     { EQEQ }
| "!="     { NEQ }
| '!'      { NOT }
| '='      { EQ }
| '<'      { LT }
| '>'      { GT }
| "<="     { LTEQ }
| ">="     { GTEQ }
| "&&"     { AND }
| "||"     { OR }
| "of"     { OF }
| "if"     { IF }
| "then"   { THEN }
```

```
| "else"   { ELSE }
| "struct" { STRUCT }
| "alias"  { ALIAS }
| "array"  { ARRAY }
| "func"   { FUNC }
| digits as lxm { INTLIT(int_of_string lxm) }
| digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLOATLIT(
   lxm) }
| "true"   { BOOLLIT(true) }
| "false"  { BOOLLIT(false) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*     as lxm { VAR(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }


and comment depth = parse
  "*/" { if depth==0 then token lexbuf else comment (depth-1) lexbuf }
| "/*" { comment (depth+1) lexbuf }
| _    { comment depth lexbuf }

and single_comment = parse
  '\n' { token lexbuf }
| _    { single_comment lexbuf }
```

9.1.3 parser.mly

```
/* SOS Parser */
/* Written primarily by Sheron, later polish by G */

%{ open Ast %}

/* Declarations */

/* %token statements... */
%token ADD SUB MUL MMUL DIV MOD SEQ
%token NOT EQ LT GT LTEQ GTEQ EQEQ NEQ AND OR
%token CONCAT OF
%token DOT COMMA COLON DOLLAR
%token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
%token IF THEN ELSE
%token STRUCT ALIAS ARRAY FUNC TO
%token <Ast.program> IMPORT
%token <int> INTLIT
%token <string> FLOATLIT
%token <bool> BOOLLIT
%token <string> VAR
%token EOF

%start program
%type <Ast.program> program

/* Associativity and Precedence */
%right VAR
```

```
%nonassoc IF THEN ELSE
%left COMMA
%left SEQ
%right EQ
%left AND OR
%left EQEQ NEQ
%left LT GT LTEQ GTEQ
%left OF
%left CONCAT
%left ADD SUB
%right MMUL
%left MUL DIV MOD
%nonassoc LBRACK RBRACK LPAREN RPAREN LBRACE RBRACE
%right NOT
%left DOT


%%

/* rules */
typeid:
    VAR { TypeID($1) }
  | ARRAY typeid { ArrayTypeID($2) }
  | FUNC types TO typeid { FxnTypeID($2, $4) }

value:
    VAR { Var ($1) }
  | value DOT VAR { StructField($1, $3) }
  | value LBRACK expr RBRACK { ArrayAccess($1, $3) }
  | DOLLAR LPAREN expr RPAREN { $3 }
  | fxn_app { $1 }

fxn_app:
    value LPAREN args RPAREN { FxnApp($1, $3) }

stexpr:
    VAR COLON typeid EQ expr { VarDef($3, $1, $5) }
  | VAR EQ expr { Assign ($1, $3) }
  | value DOT VAR EQ expr { AssignStruct($1, $3, $5) }
  | value LBRACK expr RBRACK EQ expr { AssignArray($1, $3, $6) }
  | IF expr THEN expr ELSE expr { IfElse($2,$4,$6) }
  | fxn_app { $1 }

expr:
    INTLIT { IntLit($1) }
  | FLOATLIT { FloatLit($1) }
  | BOOLLIT { BoolLit($1) }
  | NOT expr { Uop(Not,$2) }
  | SUB expr { Uop(Neg,$2) }
  | expr ADD expr { Binop($1,Add,$3) }
  | expr SUB expr { Binop($1,Sub,$3) }
  | expr MUL expr { Binop($1,Mul,$3) }
  | expr MMUL expr { Binop($1,MMul,$3) }
  | expr DIV expr { Binop($1,Div,$3) }
```

```
  | expr MOD expr { Binop($1,Mod,$3) }
  | expr EQEQ expr { Binop($1,Eq,$3) }
  | expr NEQ expr { Binop($1,Neq,$3) }
  | expr LT expr { Binop($1,Less,$3) }
  | expr GT expr { Binop($1,Greater,$3) }
  | expr LTEQ expr { Binop($1,LessEq,$3) }
  | expr GTEQ expr { Binop($1,GreaterEq,$3) }
  | expr AND expr { Binop($1,And,$3) }
  | expr OR expr { Binop($1,Or,$3) }
  | expr SEQ expr { Binop($1,Seq,$3) }
  | expr CONCAT expr {Binop($1,Concat,$3) }
  | expr OF expr { Binop($1,Of,$3) }
  | LPAREN expr RPAREN { $2 }
  | VAR LBRACE args RBRACE { NamedStruct($1, $3) }
  | LBRACE args RBRACE { AnonStruct($2) }
  | LBRACK args RBRACK { ArrayCon($2) }
  | VAR { Var ($1) }
  | value DOT VAR { StructField($1, $3) }
  | value LBRACK expr RBRACK {ArrayAccess($1, $3) }
  | stexpr { $1 }

fxn_args:
    /* nothing */ { [] }
  | fxn_args_list {List.rev $1}

fxn_args_list:
    VAR COLON typeid { [($3,$1)] }
  | fxn_args_list COMMA VAR COLON typeid { ($5,$3) :: $1 }

args:
    /* nothing */ { [] }
  | args_list {List.rev $1}

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

types:
    /* nothing */ { [] }
  | rev_types {List.rev $1}

rev_types:
    typeid { [$1] }
  | rev_types COMMA typeid { $3 :: $1 }

typedef:
    ALIAS VAR EQ typeid { Alias($2,$4) }
  | STRUCT VAR EQ LBRACE fxn_args RBRACE { StructDef($2,$5) }

stmt:
    typedef { Typedef($1) }
  | VAR COLON LPAREN fxn_args RPAREN TO typeid EQ expr {FxnDef($7,$1,$4,$9
   )}
  | stexpr { Expression($1) }
```

```
stmts:
    stmt { [$1] }
  | stmts stmt { $2:: $1 }

program:
    stmts EOF { List.rev $1 }
  | IMPORT program { $1 @ $2 }
```

9.1.4 ast.mli

```
(* Abstract syntax tree for SOS *)
(* Written by G *)

type operator =
(*num operators*)
Add | Sub | Mul | Div | Mod | MMul
(*relational operators*)
| Eq | Neq | Less | Greater | LessEq | GreaterEq
(*boolean operators*)
| And | Or
(* array combination *)
| Concat | Of
(*sequencing*)
| Seq


type uop = Not | Neg

type id = string (* non-type id *)
type tid = (* type id *)
  TypeID of string
| ArrayTypeID of tid
| FxnTypeID of tid list * tid
type import = string

(* type name pair *)
type argtype = tid * id

(* all possible expression statements, found in LRM sec 4 *)
and expr =
  VarDef of tid * id * expr                (* type name = val *)
| Assign of id * expr                      (* id = val *)
| AssignStruct of expr * id * expr         (* struct.field = val *)
| AssignArray of expr * expr * expr          (* id[expr] = expr *)
| Uop of uop * expr                        (* uop expr *)
| Binop of expr * operator * expr          (* expr op expr *)
| FxnApp of expr * expr list
| IfElse of expr * expr * expr             (* if expr then expr else expr
   *)
| ArrayCon of expr list                    (* [expr, ...] *)
| AnonStruct of expr list                  (* {expr, ...} *)
| NamedStruct of id * expr list           (* name{expr, ...} *)
| Var of id                                (* name *)
```

```
| ArrayAccess of expr * expr                  (* name[expr] *)
| StructField of expr * id                   (* struct.id *)
| IntLit of int                              (* int *)
| FloatLit of string                         (* float *)
| BoolLit of bool                            (* bool *)

type typedef =
  Alias of id * tid                         (* alias name = type *)
| StructDef of id * argtype list            (* struct name = {type name,
   ...} *)

type stmt =
  Typedef of typedef
| Expression of expr
| FxnDef of tid * id * argtype list * expr

type program = stmt list
```

9.1.5 astprint.ml

```
(* Very basic """pretty""" printer for the AST *)
(* Written by G *)
open Ast

let rec comma_list_str f l = match l with
  [] -> ""
| hd :: tl -> match tl with
    [] -> f hd
  | _ -> f hd ^ ", " ^ comma_list_str f tl

let rec typeid_str t = match t with
  TypeID(s) -> s
| ArrayTypeID(p) -> "array " ^ typeid_str p
| FxnTypeID(l, t) -> "func " ^ comma_list_str typeid_str l ^" -> "^
   typeid_str t

let basic_print prog =
  let rec print_stmt = function
    Typedef(t) -> let print_tdef = function
      Alias(a, b) -> print_endline ("alias " ^ a ^ " " ^ typeid_str(b))
    | StructDef(a, b) -> print_endline ("struct " ^ a ^ " = {" ^
   comma_list_str (fun (a, b) -> typeid_str(a) ^ " " ^ b) b ^ "}")
    in print_tdef t
  | FxnDef(a, b, c, d) -> print_endline (typeid_str(a) ^ " " ^ b ^ "(" ^
   comma_list_str (fun (a, b) -> typeid_str(a) ^ " " ^ b) c ^ ") = ");
   print_stmt (Expression(d))
  | Expression(e) -> let rec expr_str = function
      VarDef(a, b, c) -> typeid_str(a) ^ " " ^ b ^ " = " ^ expr_str c
    | Assign(a, b) -> a ^ " = " ^ expr_str b
    | AssignStruct(a, b, c) -> expr_str a ^ "." ^ b ^ " = " ^ expr_str c
    | AssignArray(a, b, c) -> expr_str a^"["^expr_str b ^"] = "^ expr_str
   c
    | ArrayAccess(nm, idx) -> expr_str nm^"["^expr_str idx^"]"
    | Uop(a, b) -> let uoperator_str = function Not -> "!" | Neg -> "-" in
```

```
    uoperator_str a ^ expr_str b
    | Binop(a, b, c) -> let operator_str = function
        Add -> "+"
      | Sub -> "-"
      | Mul -> "*"
      | MMul -> "**"
      | Div -> "/"
      | Mod -> "%"
      | Eq -> "="
      | Neq -> "!="
      | Less -> "<"
      | Greater -> ">"
      | LessEq -> "<="
      | GreaterEq -> ">="
      | And -> "&&"
      | Or -> "||"
      | Of -> "of"
      | Concat -> "@"
      | Seq -> ";" in
      "(" ^ expr_str a ^ " " ^ operator_str b ^ " " ^ expr_str c ^ ")"
    | FxnApp(a, b) ->
        expr_str a ^ "(" ^ comma_list_str expr_str b ^ ")"
    | IfElse(a, b, c) -> "if (" ^ expr_str a ^ ")\n then (" ^ expr_str b ^
    ")\n else (" ^ expr_str c ^")\n"
    | ArrayCon(a) -> "[" ^ comma_list_str expr_str a ^ "]"
    | AnonStruct(a) -> "{" ^ comma_list_str expr_str a ^ "}"
    | NamedStruct(a, b) -> a ^ "{" ^ comma_list_str expr_str b ^ "}"
    | Var(a) -> a
    | StructField(a, b) -> expr_str a ^ "." ^ b
    | IntLit(i) -> string_of_int i
    | FloatLit(f) -> f
    | BoolLit(true) -> "true"
    | BoolLit(false) -> "false"
    in print_endline(expr_str e)
  in
  List.iter print_stmt prog

(*let _ =
  let lexbuf = Lexing.from_channel stdin in
  let prog = Parser.program Scanner.token lexbuf in
  basic_print prog *)
```

9.1.6 semant.ml

```
(* Semantic checking for the SOS compiler *)
(* Written primarily by G *)

open Ast
open Sast

(* import map for global variables (VarDef, FxnDef, Alias, StructDef) *)
(* we don't have scope defined so a string * tid is enough? *)
module StringMap = Map.Make(String)
module StringSet = Set.Make(String)
```

```
(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.
   Check each statement *)

(* Environment type for holding all the bindings currently in scope *)
type environment = {
  typemap : typeid StringMap.t;
  fxnnames : StringSet.t;
  varmap : typeid StringMap.t;
}

(* External function signatures *)
(* This is re-used in Codegen *)
(* type, name that to be called in SOS, name in c file *)
let external_functions : (typeid * string) list =
[ Func([Float], Float), "sqrtf" ;
  Func([Float], Float), "sinf" ;
  Func([Float], Float), "cosf" ;
  Func([Float], Float), "tanf" ;
  Func([Float], Float), "asinf" ;
  Func([Float], Float), "acosf" ;
  Func([Float], Float), "atanf" ;
  Func([Float], Float), "toradiansf" ;
  Func([Int; Int], Void), "gl_startRendering" ;
  Func([Int; Int; Int], Void), "gl_endRendering" ;
  Func([Array(Float); Array(Float); Int], Void), "gl_drawCurve" ;
  Func([Array(Float); Array(Float); Int; Int], Void), "gl_drawShape" ;
  Func([Array(Float); Array(Float); Int], Void), "gl_drawPoint" ;
]


let raisestr s = raise (Failure s)

let check prog =

  (* add built-in function such as basic printing *)
  let built_in_decls =
    let add_bind map (name, ty) = StringMap.add name (Func([ty], Void))
   map
    in List.fold_left add_bind StringMap.empty [ ("print", Int);
                                                 ("printf", Float) ]
  in
  (* add external functions *)
  let built_in_decls = List.fold_left
    (fun map (decl, nm) -> StringMap.add nm decl map)
    built_in_decls external_functions
  in
  let starting_fxns = List.fold_left
    (fun map (_, nm) -> StringSet.add nm map)
    StringSet.empty external_functions
  in
  let starting_fxns = StringSet.add "print" starting_fxns in
  let starting_fxns = StringSet.add "printf" starting_fxns in
```

```
(*  (* add math functions *)
  let built_in_decls = List.fold_left
    (fun map (decl, nm) -> StringMap.add nm decl map)
    built_in_decls math_functions
  in *)

  (* add built-in types such as int, float *)
  let built_in_types = (
    let add_type map (name, ty) = StringMap.add name ty map
    in List.fold_left add_type StringMap.empty [("int", Int); ("bool",
   Bool); ("float", Float); ("void", Void)] )
  in

  (* Void id *)
  let built_in_decls = StringMap.add "void" Void built_in_decls in

  (* Initial environment containing built-in types and functions *)
  let global_env = { typemap = built_in_types; varmap = built_in_decls
  ; fxnnames = starting_fxns }
  in

  (* resolve the type of a tid to a typeid *)
  let rec resolve_typeid t map = match t with
    TypeID(s) -> if StringMap.mem s map
      then StringMap.find s map
      else raisestr ("Could not resolve type id "^s)
  | ArrayTypeID(s) -> Array(resolve_typeid s map)
  | FxnTypeID(l, r) -> Func(List.map (fun tt -> resolve_typeid tt map) l,
          resolve_typeid r map)
  in

  (* should add a function to add three things above dynamically *)
  (* let add_id_type = ()
  in *)

  (* function to lookup *)
  let type_of_id s map =
    if StringMap.mem s map then StringMap.find s map
    else raisestr ("Unknown variable name "^s)
  in

  (* function to lookup the type of a struct field *)
  let type_of_field stype f =
    match stype with
      Struct(sargs) ->
      let rec find_field f = function
        (ft, fn) :: tl -> if fn = f then ft else find_field f tl
        | _ -> raisestr ("Could not find field "^f)
      in find_field f sargs
     | _ -> raisestr ("Cannot access fields for a non-struct variable")
  in

  let add_typedef td map =
    match td with
```

```
    Alias(nm, t) -> if StringMap.mem nm map
      then raisestr ("Cannot create an alias with preexisting name " ^
 nm)
      else StringMap.add nm (resolve_typeid t map) map
  | StructDef(nm, l) -> let sargl = List.map (fun (t, i) -> (
 resolve_typeid t map, i)) l in
    StringMap.add nm (Struct(sargl)) map
in

let rec add_formals args vmap tmap = match args with
  (typ, nm) :: tl -> add_formals tl (StringMap.add nm (resolve_typeid
 typ tmap) vmap) tmap
  | _ -> vmap
in

(* Matches a struct type component-wise without names *)
(* Can also work within arrays or other structs *)
let rec match_str_type t1 t2 =
  match (t1, t2) with
    (Int, Int) -> true
  | (Float, Float) -> true
  | (Bool, Bool) -> true
  | (Void, Void) -> true
  | (Array(a1), Array(a2)) -> match_str_type a1 a2
  | (Struct(s1), Struct(s2)) ->
      if List.length s1 != List.length s2 then false
      else
      List.fold_left2
        (fun b (st1, _) (st2, _) -> if match_str_type st1 st2 then
           b else false) true s1 s2
  | _ -> false
in

(* Returns a sexp that casts sexp to typ, if possible. *)
(* Returns sexp if no cast is required *)
let cast_to typ sexp err_str =
 let (expt, sx) = sexp in
 if expt=typ then sexp else
 if match_str_type expt typ then (typ, sx) else
 if expt=EmptyArray then
  match typ with
    Array(t) -> (Array(t), SArrayCon([]))
  | _ -> raisestr ("Cannot cast empty array to non-array type")
 else
 (
 (match (typ, expt) with
   (Int, Float)   -> ()
 | (Int, Bool)    -> ()
 | (Bool, Int)    -> ()
 | (Bool, Float)  -> ()
 | (Float, Int)   -> ()
 | (Void, _)      -> ()
 | _ -> raisestr err_str );
 (typ, SCast(sexp)))
```

```
in

(* Function with the same signature as cast_to
 * Used to ignore casting checks *)
let no_cast typ sexp err_str =
  ignore(err_str);
  let (expt, _) = sexp in
  if expt=typ then sexp else
  raisestr ("No type casting allowed within arrays")
in

(* Converts a type to a string *)
let rec type_str = function
  Int        -> "int"
| Float      -> "float"
| Bool       -> "bool"
| Void       -> "void"
| Array(t)   -> "array "^type_str t
| Struct(sl) -> "struct {"^
  (let rec struct_typ_str = function
    (hdt, _) :: (h::t) -> type_str hdt ^ ", " ^struct_typ_str (h::t)
  | (hdt, _) :: _ -> type_str hdt
  | _ -> ""
  in struct_typ_str sl)^"}"
| Func(al, rt) -> "func "^
  (let rec func_typ_str = function
    (hdt) :: (h::t) -> type_str hdt ^", "^func_typ_str (h::t)
  | (hdt) :: _ -> type_str hdt
  | _ -> "" in func_typ_str al)^" -> "^type_str rt
| EmptyArray -> "[]"
in

(* Identifies structs with only ints or only floats *)
let arith_struct t1 at =
    match t1 with
      Struct(l) ->
        List.fold_left (fun b (ft, _) -> if ft=at then b else false)
          true l
    | _ -> false
in
let either_struct t1 t2 =
  match t1 with Struct(_) -> true
  | _ -> match t2 with Struct(_) -> true
  | _ -> false
in
let is_struct t1 = match t1 with Struct(_) -> true | _ -> false
in

let assert_arith t1 =
  if arith_struct t1 Float then () else
  if arith_struct t1 Int then () else
  raisestr ("Can only operate on arithmetic structs")
in
```

```
let sop_type t1 =
  if arith_struct t1 Float then Float else
  if arith_struct t1 Int then Int else
  Void
in

let addsub_expr env exp1 op exp2 cast =
    let (t1, _) = exp1 in let (t2, _) = exp2 in
    (* Can add structs component-wise *)
    if either_struct t1 t2 then
      if match_str_type t1 t2 then
        (assert_arith t1 ;
        (t1, SBinop(exp1, op, exp2)), env)
      else
      raisestr ("Can only add or subtract structs of matching type")

    else
    let err = "Cannot add or subtract "^type_str t1^" and "^type_str t2
 in
    match (t1, t2) with
      (Float, _) -> (Float, SBinop(exp1, op, cast Float exp2 err)), env
    | (_, Float) -> (Float, SBinop(cast Float exp1 err, op, exp2)), env
    | (Int, _)   -> (Int,   SBinop(exp1, op, cast Int   exp2 err)), env
    | (_, Int)   -> (Int,   SBinop(cast Int exp1 err, op,   exp2)), env
    | _ -> raisestr (err)
in

let mul_expr env exp1 op exp2 cast =
  let (t1, _) = exp1 in let (t2, _) = exp2 in
  (* Can scale structs and take the dot product *)
  if either_struct t1 t2 then
    if match_str_type t1 t2 then
      (assert_arith t1 ;
      (sop_type t1, SBinop(exp1, op, exp2)), env)

    else if is_struct t1 then
      (assert_arith t1 ;
      (t1, SBinop(exp1, op, cast (sop_type t1) exp2
        "Cannot scale a struct by a non-scalar")), env)
    else
      (assert_arith t2 ;
      (t2, SBinop(exp2, op, cast (sop_type t2) exp1
        "Cannot scale a struct by a non-scalar")), env)
  else
  let err = "Cannot multiply "^type_str t1^" and "^type_str t2 in
  match (t1, t2) with
    (Float, _) -> (Float, SBinop(exp1, op, cast Float exp2 err)), env
  | (_, Float) -> (Float, SBinop(cast Float exp1 err, op, exp2)), env
  | (Int, _)   -> (Int,   SBinop(exp1, op, cast Int   exp2 err)), env
  | (_, Int)   -> (Int,   SBinop(cast Int exp1 err, op,   exp2)), env
  | _ -> raisestr ("Cannot multiply "^type_str t1^" and "^type_str t2)
in

let mmul_expr env exp1 op exp2 cast =
```

```
  ignore(cast);
  let (t1, _) = exp1 in let (t2, _) = exp2 in
  (* Can multiply two n * n matrices OR
     Can multiply an n*n matrix with an n*1 vector *)
  match (t1, t2) with
    (Struct(l1), Struct(l2)) ->
    let n1 = List.length l1 in let n2 = List.length l2 in
    let int_sqrt n =
      let rec int_sqrt_inner n m =
        if m * m = n then Some(m)
        else if m * m < n then int_sqrt_inner n (m+1)
        else None
      in int_sqrt_inner n 1
    in
    let sq1 = int_sqrt n1 in (match sq1 with
      | Some(m1) ->
        if n1 = n2 then
          (Struct(l1), SBinop(exp1, op, exp2)), env

        else if m1 = n2 then
          (Struct(l2), SBinop(exp1, op, exp2)), env

        else raisestr ("Can only multiply a "^string_of_int m1^" by "^
 string_of_int m1^" matrix with a square matrix or vector of the same
 height")
      | None -> raisestr ("Can only multiply square matrices")
    )

  | _ -> raisestr ("Cannot matrix multiply non-structs")
in

let div_expr env exp1 op exp2 cast =
    let (t1, _) = exp1 in let (t2, _) = exp2 in
    (* Can scale structs *)
    if is_struct t1 then
      (assert_arith t1 ;
      (t1, SBinop(exp1, op, cast (sop_type t1) exp2
        "Cannot scale a struct by a non-scalar")), env)
    else
    let err = "Cannot divide "^type_str t1^" and "^type_str t2 in
    match (t1, t2) with
      (Float, _) -> (Float, SBinop(exp1, op, cast Float exp2 err)), env
    | (_, Float) -> (Float, SBinop(cast Float exp1 err, op, exp2)), env
    | (Int, _)   -> (Int,   SBinop(exp1, op, cast Int   exp2 err)), env
    | (_, Int)   -> (Int,   SBinop(cast Int exp1 err, op,   exp2)), env
    | _ -> raisestr ("Cannot divide "^type_str t1^" and "^type_str t2)
in

let mod_expr env exp1 op exp2 cast =
    ignore (cast);
    let (t1, _) = exp1 in let (t2, _) = exp2 in
    match (t1, t2) with
      (Int, Int) -> (Int, SBinop(exp1, op, exp2)), env
    | _ -> raisestr ("Can only take the modulo with integers")
```

```
in

let eq_expr env exp1 op exp2 cast =
    let (t1, _) = exp1 in let (t2, _) = exp2 in
    (* Can equate arith structs *)
    if either_struct t1 t2 then
      if match_str_type t1 t2 then
        (assert_arith t1 ;
        (Bool, SBinop(exp1, op, exp2)), env)
      else
      raisestr ("Can only equate structs of matching type")

    else
    let err = "Cannot equate "^type_str t1^" and "^type_str t2 in
    match (t1, t2) with
      (Float, _) -> (Bool, SBinop(exp1, op, cast Float exp2 err)), env
    | (_, Float) -> (Bool, SBinop(cast Float exp1 err, op, exp2)), env
    | (Int, _)   -> (Bool, SBinop(exp1, op, cast Int   exp2 err)), env
    | (_, Int)   -> (Bool, SBinop(cast Int exp1 err, op,   exp2)), env
    | _ -> raisestr (err)
in

let comp_expr env exp1 op exp2 cast =
    let (t1, _) = exp1 in let (t2, _) = exp2 in
    let err = "Cannot compare "^type_str t1^" and "^type_str t2 in
    match (t1, t2) with
      (Float, _) -> (Bool, SBinop(exp1, op, cast Float exp2 err)), env
    | (_, Float) -> (Bool, SBinop(cast Float exp1 err, op, exp2)), env
    | (Int, _)   -> (Bool, SBinop(exp1, op, cast Int exp2 err  )), env
    | (_, Int)  -> (Bool, SBinop(cast Int exp1 err, op, exp2  )), env
    | _ -> raisestr ("Cannot compare "^type_str t1^" and "^type_str t2)
in

let logic_expr env exp1 op exp2 cast =
    let err_str = "Could not resolve boolean operands to boolean values"
  in
    (Bool, SBinop(cast Bool exp1 err_str, op, cast Bool exp2 err_str)),
 env
in

let array_expr env exp1 op exp2 =
    let (t1, _) = exp1 in let (t2, _) = exp2 in
    (match t2 with Array(_) -> ()
     | _ -> raisestr ("Cannot perform array operations on non-array type
  "^type_str t2)      ) ;
    match op with
      Concat     -> if t1 = t2 then (t2, SBinop(exp1, op, exp2)), env
      else raisestr ("Cannot concatenate arrays of different types")
    | _ (* Of *) -> (t2, SBinop((cast_to Int exp1
                  "First operand of of operator must be an int"),
                op, exp2)), env
in

let rec binop_expr env exp1 op exp2 cast =
```

```
  if op = Of || op = Concat then array_expr env exp1 op exp2
  else
  let e = SVar("empty") in
  let t1, _ = exp1 in
  match t1 with Array(t) ->
    let (ot, _), _ = binop_expr env (t, e) op exp2 no_cast in
    (Array(ot), SBinop(exp1, op, exp2)), env
  | _ ->
  let t2, _ = exp2 in
  match t2 with Array(t) ->
    let (ot, _), _ = binop_expr env exp1 op (t, e) no_cast in
    (Array(ot), SBinop(exp1, op, exp2)), env
  | _ ->
  match op with
    Add -> addsub_expr env exp1 op exp2 cast
  | Sub -> addsub_expr env exp1 op exp2 cast
  | Mul -> mul_expr    env exp1 op exp2 cast
  | MMul-> mmul_expr   env exp1 op exp2 cast
  | Div -> div_expr    env exp1 op exp2 cast
  | Mod -> mod_expr    env exp1 op exp2 cast
  | Eq  -> eq_expr     env exp1 op exp2 cast
  | Neq -> eq_expr     env exp1 op exp2 cast
  | Less      -> comp_expr env exp1 op exp2 cast
  | Greater   -> comp_expr env exp1 op exp2 cast
  | LessEq    -> comp_expr env exp1 op exp2 cast
  | GreaterEq -> comp_expr env exp1 op exp2 cast
  | Or ->  logic_expr env exp1 op exp2 cast
  | And -> logic_expr env exp1 op exp2 cast
  | _ -> raisestr ("Special case, this should never happen")
in

(* Takes a pair of sexprs and makes their types agree by adding casts,
 if possible. *)
let agree_type e1 e2 err_str =
  let ((t1, _), (t2, _)) = (e1, e2) in
  if t1=t2 then (e1, e2) else
  (match (t1, t2) with
   (* Priority is Float -> Int -> Bool *)
    (Void, _)  -> (e1, cast_to t1 e2 err_str)
  | (_, Void)  -> (cast_to t2 e1 err_str, e2)
  | (Float, _) -> (e1, cast_to t1 e2 err_str)
  | (_, Float) -> (cast_to t2 e1 err_str, e2)
  | (Int, _)   -> (e1, cast_to t1 e2 err_str)
  | (_, Int)   -> (cast_to t2 e1 err_str, e2)
  | (Bool, _)  -> (e1, cast_to t1 e2 err_str)
  | (_, Bool)  -> (cast_to t2 e1 err_str, e2)
  | _ -> raisestr err_str )
in

let rec assert_nonvoid = function
  Void -> raisestr ("Cannot use a void type in this context")
| Array(t) -> assert_nonvoid t
| _ -> ()
in
```

```
let assert_non_reserved env name =
  if name="copy" || name="free" then
  raisestr ("Cannot create an identifier with reserved name "^name)
  else
  if StringSet.mem name env.fxnnames then
  raisestr ("Cannot create an identifier with defined function name "^
 name)
  else ()
in

let rec expr env = function

    VarDef (tstr, name, exp) ->
      assert_non_reserved env name;
      let (sexp, _) = expr env exp in
      let t = resolve_typeid tstr env.typemap in
      assert_nonvoid t ;
      let (exptype, _) = sexp in
      ((t, SVarDef(t, name, cast_to t sexp
                   ("Could not resolve type when defining "^name^
                    "(Found "^type_str exptype^", expected "^type_str t
 ^")"))),
       { env with varmap = StringMap.add name t env.varmap } )

  | Assign (name, exp) ->
       if StringSet.mem name env.fxnnames then
       raisestr ("Cannot assign a defined (non-variable) function")
       else
       let ((exptype, sexp), _) = expr env exp in
       let t = type_of_id name env.varmap in
       ((t, SAssign(name, cast_to t (exptype, sexp)
                ("Could not match type when assigning variable "^name^
                " (Found "^type_str exptype^", expected "^type_str t^")
 "))),env)

  | AssignStruct (struct_exp, field, exp) ->
       let ((exptype, sexp), env) = expr env exp in
       let (struct_sexp, env)  = expr env struct_exp in
       let (strt, _) = struct_sexp in
       let t = type_of_field strt field in
       ((t, SAssignStruct(struct_sexp, field, cast_to t (exptype, sexp)
              ("Could not match type when assigning field "^field^
               " (Found "^type_str exptype^", expected "^type_str t^")")
 )), env)

  | AssignArray (array_exp, idx, exp) ->
      let ((exptype, sexp), env) = expr env exp in
      let (array_sexp, env) = expr env array_exp in
      let (arrt, _) = array_sexp in
      let (sidx, _) = expr env idx in
      (match sidx with (Int, _) -> () | _ ->
         raisestr ("Array index must be an integer ") );
      let eltype = match arrt with Array(el) -> el | _ -> Void in
```

```
     ((eltype, SAssignArray(array_sexp, sidx, cast_to eltype (exptype,
sexp)
 ("Could not match type when assigning array "^
 "(Found "^type_str exptype^", expected"^type_str eltype^")"))), env)

 | Uop(op, exp) ->
     let (sexp, env) = expr env exp in (
     match op with
       Not -> (Bool, SUop(op, cast_to Bool sexp
         "Could not resolve expression to bool")), env
     | Neg -> let (t, _) = sexp in
       (match t with
         Int -> ()
       | Float -> ()
       | _ -> raisestr "Cannot negate non-arithmetic types" );
       (t, SUop(op, sexp)), env )

 | Binop(exp1, op, exp2) ->
   if op = Seq then
      (* Need to pass new environments *)
      (* jk this happens anyways. but seq still gets to feel special *)
      let (e1, env) = expr env exp1 in
      let (e2, env) = expr env exp2 in
      let (t, _) = e2 in
      (t, SBinop(e1, Seq, e2)), env
   else
      let (e1, env) = expr env exp1 in
      let (e2, env) = expr env exp2 in
      binop_expr env e1 op e2 cast_to

 | FxnApp (exp, args) ->
      if exp=Var("copy") then (* Copy constructor *)
      (match args with
        [ex] ->
         let (sexp, _) = expr env ex in
         let (t, _) = sexp in
         (match t with
           Array(_) -> ()
         | Struct(_) -> ()
         | _ -> raisestr ("Can only use Copy constructor on reference
types"));
          ((t, SFxnApp((Func([t], t), SVar("copy")), [sexp])), env)

      | _ -> raisestr ("Too many arguments for Copy constructor")
      )

      else if exp=Var("free") then (* Free instr *)
      (match args with
        [ex] ->
          let (sexp, _) = expr env ex in
          let (t, _) = sexp in
          (match t with
            Array(_) -> ()
          | Struct(_) -> ()
```

```
          | _ -> raisestr ("Can only free memory of struct and array
types")) ;
          ((Void, SFxnApp((Func([t], t), SVar("free")), [sexp])), env)
       | _ -> raisestr ("Too many arguments for free()")
       )

      else (* All other functions *)
      let fxn, env = expr env exp in
      let sargs, base_rt = match fxn with (Func(l, t), _) -> l, t
       | _ -> raisestr ("Could not resolve expression to a function")
in

      let check_args sigl expl env =
        if (List.length sigl) != (List.length expl) then
        raisestr ("Incorrect number of arguments for function")
        else
        let (l, b) = List.fold_left2
        (fun (l, arr) typ e ->
         let ((exptype, sexp), _) = expr env e in
         if exptype = Array(typ) then (exptype, sexp) :: l, true
         else (cast_to typ (exptype, sexp)
             ("Could not match type of argument")) :: l, arr )
         ([], false) sigl expl
        in (List.rev l, b)
      in
      let cargs, arrmode = check_args sargs args env in
      if arrmode then
      (( (if base_rt=Void then Void else Array(base_rt)), SIterFxnApp(
fxn, cargs)), env)
      else ((base_rt, SFxnApp(fxn, cargs)), env)

 | IfElse (eif, ethen, eelse) ->
     let (sif, env) = expr env eif in
     let scif = cast_to Bool sif
      "Could not resolve if condition to a bool" in
     let (sthen, _) = expr env ethen in
     let (selse, _)  = expr env eelse in
     let (scthen, scelse) = agree_type sthen selse
       ("Could not reconcile types of then and else clauses ("^
      (let (t,_) = sthen in type_str t)^", "^
      (let (t,_) = selse in type_str t)^")") in
     let (t, _) = scthen in
     ((t, SIfElse(scif, scthen, scelse)), env)

 | ArrayCon l -> (match l with
   hd :: tl ->
     let ((exptype, sexp), env) = expr env hd in
     assert_nonvoid exptype ;
     let rev_sexprs, env = List.fold_left
       (fun (l, env) ex -> let (se, env) = expr env ex in
        (cast_to exptype se
          "Could not agree types of array literal") :: l, env)
       ([(exptype, sexp)], env) tl in
     ((Array(exptype), SArrayCon(List.rev rev_sexprs)), env)
```

```
    | [] -> ((EmptyArray, SArrayCon([])), env)
    )

  | AnonStruct l ->
    let rec create_anon_struct env n = function
      e :: tl -> let ((exptype, sexp), env) = expr env e in
        let (typel, expl), env = create_anon_struct env (n+1) tl in
        ((exptype, "x"^string_of_int n) :: typel, (exptype, sexp) ::
expl), env
    | _ -> ([], []), env
    in
    let (typel, expl), env = create_anon_struct env 1 l in
    ((Struct(typel), SStruct("anon", expl)), env)

  | NamedStruct (name, l)  ->
    let st = resolve_typeid (TypeID(name)) env.typemap in
    (match st with
      Struct(sargs) ->
      let rec create_named_struct env argl = function
        e :: tl -> let (sexp, env) = expr env e in
          (match argl with (t, nm) :: argtl ->
            let stl, env = create_named_struct env argtl tl in
            cast_to t sexp
              ("Could not resolve type of struct field "^nm)
             :: stl, env
          | _ -> raisestr ("Too many arguments for struct "^name))
        | [] -> (match argl with
          [] -> [], env
         | _ -> raisestr ("Not enough arguments for struct "^name))
      in
      let sexprs, env = create_named_struct env sargs l
      in ((st, SStruct(name, sexprs)), env)
    | _ -> raisestr ("Cannot resolve the struct name "^name)
   )

  | Var i -> ((type_of_id i env.varmap, SVar(i)), env)
  | ArrayAccess (arr, idx) ->
    let (sarr, env) = expr env arr in
    let (sidx, env) = expr env idx in
    let (t, _) = sarr in
    let el_t = (match t with Array(e) -> e
      | _ -> raisestr ("Cannot access elements of non-array variable"))
    in
    ((el_t, SArrayAccess(sarr, cast_to Int sidx
       "Could not cast array index to an integer")), env)
  | StructField (str, fl) ->
     let (sstr, env) = expr env str in
     let (t, _) = sstr in
     (match t with
       Struct(_) ->
        ((type_of_field t fl, SStructField(sstr ,fl)), env)
     | Array(_) -> if fl="length" then
        (Int, SArrayLength(sstr)), env
        else raisestr ("Cannot access fields for a non-struct variable")
```

```
      | _ -> raisestr ("Cannot access fields for a non-struct variable")
     )

   | IntLit i -> ((Int, SIntLit i), env)
   | FloatLit f -> ((Float, SFloatLit f), env)
   | BoolLit b -> ((Bool, SBoolLit b), env)
 in

 let make_stypedef env = function
     Alias(nm, tp) -> SAlias(nm, resolve_typeid tp env.typemap)
   | StructDef(nm , l) -> SStructDef(nm, List.map (fun (t, i) -> let tt =
   resolve_typeid t env.typemap in assert_nonvoid tt; (tt, i)) l)
 in

 (* check a single statement and update the environment *)
 let stmt env = function
     Expression(e) -> let (se, en) = expr env e in (SExpression (se), en)
   | Typedef(td) -> (STypeDef(make_stypedef env td), {env with typemap =
  add_typedef td env.typemap})
   | FxnDef (tstr, name, args, exp) ->
       assert_non_reserved env name ;
       let t = resolve_typeid tstr env.typemap in
       let sargs = List.map (fun (tp, nm) -> resolve_typeid tp env.
  typemap, nm)
         args in
       let argtypes = List.map (fun (tp, _) -> assert_nonvoid tp; tp)
  sargs in
       let newvarmap = StringMap.add name (Func(argtypes, t)) env.varmap
  in
       let env = { env with varmap = newvarmap; fxnnames = StringSet.add
           name env.fxnnames } in
       let ((exptype, sx), _) = expr { env with
          varmap = add_formals args env.varmap env.typemap; } exp
       in
       (SFxnDef(t, name, sargs, cast_to t (exptype, sx)
                    ("Incorrect return type for function "^name
                    ^" (Found "^type_str exptype^", expected "^type_str t
  ^")")))),
        env

 in

 let rec stmts env = function
   hd :: tl -> let (st, en) = stmt env hd in st :: stmts en tl
 | _ -> []
 in stmts global_env prog
```

9.1.7 sast.mli

```
(* Semantically-checked AST *)

open Ast

(* Detailed type meaning *)
```

```
type typeid =
  Int
| Float
| Bool
| Void
| Array of typeid
| Struct of sargtype list
| Func of typeid list * typeid
| EmptyArray (* The empty array constructor, [] *)
and sargtype = typeid * id

(* Detailed function binding *)
type func_bind = {
  ftype : typeid;
  formals : sargtype list;
}

type sexpr = typeid * sx
and sx =
  SVarDef of typeid * id * sexpr                (* type name = val *)
| SAssign of id * sexpr                        (* id = val *)
| SAssignStruct of sexpr * id * sexpr          (* id.field = val *)
| SAssignArray of sexpr * sexpr * sexpr        (* id[expr] = expr *)
| SUop of uop * sexpr                          (* uop expr *)
| SBinop of sexpr * operator * sexpr           (* expr op expr *)
| SFxnApp of sexpr * sexpr list
| SIterFxnApp of sexpr * sexpr list
| SIfElse of sexpr * sexpr * sexpr             (* if expr then expr else
   expr *)
| SArrayCon of sexpr list                      (* [expr, ...] *)
(* | SAnonStruct of sexpr list                 (* {expr, ...} *)
| SNamedStruct of id * sexpr list             (* name{expr, ...} *) *)
| SStruct of id * sexpr list
| SVar of id                                   (* name *)
| SArrayAccess of sexpr * sexpr                (* name[expr] *)
| SArrayLength of sexpr                        (* name.length *)
| SStructField of sexpr * id                   (* name.id *)
| SIntLit of int                               (* int *)
| SFloatLit of string                          (* float *)
| SBoolLit of bool                             (* bool *)
| SCast of sexpr                               (* type casting *)

type stypedef =
  SAlias of id * typeid
| SStructDef of id * sargtype list

type sstmt =
  STypeDef of stypedef
| SExpression of sexpr
| SFxnDef of typeid * id * sargtype list * sexpr (* type id (type name,
   ...) = val *)

type sprogram = sstmt list
```

9.1.8 sastprint.ml

```
(* Very basic """pretty""" printer for SAST *)
(* Written by G *)
open Sast
open Ast

let basic_print sast =
  let rec sargl_string l =
    List.fold_left (fun str (t, id) -> (if str = "" then "" else str^", ")
   ^typeid_string t ^" "^id) "" l
  and typeid_string = function
    Int -> "int"
  | Float -> "float"
  | Bool -> "bool"
  | Void -> "void"
  | Array(t) -> "array "^typeid_string t
  | Struct(l) -> "{"^sargl_string l^"}"
  | Func(_) -> "func"
  | EmptyArray -> "[]"
  in

  let print_typedef = function
    SAlias(nm, al) -> print_string("alias "^nm^" = "^typeid_string al^"\n
   ")
  | SStructDef(nm,  l) -> print_string("struct "^nm^" = {"^sargl_string l
   ^"}\n")
  in

  let rec explstr l = List.fold_left
    (fun s e -> (if s="" then s else s^", ")^sexp_string e) "" l

  and sexp_string (t, e) = match e with
    SVarDef(_, var, exp) -> typeid_string t ^" "^var^" = "^sexp_string exp
   ^"\n"
  | SAssign(var, exp) -> "("^typeid_string t^") "^var^" = "^sexp_string
   exp^"\n"
  | SAssignStruct(var, f, exp) -> "("^typeid_string t^") "^sexp_string var
   ^"."^f^" = "^sexp_string exp^"\n"
  | SAssignArray(var, e1, e2) -> "("^typeid_string t^") "^sexp_string var
   ^"["^sexp_string e1^"] = "^sexp_string e2^"\n"
  | SUop(op, exp) -> "("^typeid_string t^")"^
    (match op with Not -> "!" | Neg -> "-")^sexp_string exp
  | SBinop (e1, op, e2) -> let opstr  = match op with
      Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/" | Mod -> "%" |
      MMul -> "**" |
      Eq -> "==" | Neq -> "!=" | Less -> "<" | Greater -> ">" | LessEq ->
   "<=" |
      GreaterEq -> ">=" | And -> "&&" | Or -> "||" | Seq -> ";" |
      Of -> "of" | Concat -> "@" in
    "("^typeid_string t^") ("^sexp_string e1^" "^opstr^" "^sexp_string e2
   ^")"
  | SIterFxnApp (fe, expl)
  | SFxnApp (fe, expl) -> "("^typeid_string t^") "^sexp_string fe^"("^
```

```
    (explstr expl)^")\n"
  | SIfElse(e1, e2, e3) -> "("^typeid_string t^") if "^sexp_string e1^"\
   nthen "^sexp_string e2^"\nelse "^sexp_string e3^"\n"
  | SArrayCon(expl) ->
      "("^typeid_string t^") ["^explstr expl^"]"
  | SStruct(nm, expl) ->
      "("^typeid_string t^") "^nm^"{"^explstr expl^"}"
  | SVar(id) -> "("^typeid_string t^") "^id
  | SStructField(id, f) -> "("^typeid_string t^") "^sexp_string id^"."^f
  | SIntLit(i) -> string_of_int i
  | SFloatLit(f) -> f
  | SBoolLit(b) -> string_of_bool b
  | SArrayAccess(nm, idx) -> "("^typeid_string t^") "^sexp_string nm^"["^
   sexp_string idx^"]"
  | SArrayLength(nm) -> sexp_string nm^".length"
  | SCast(exp) -> sexp_string exp
  in

  let print_sstmt = function
    STypeDef(td) -> print_typedef td
  | SExpression(sexpr) -> print_string(sexp_string sexpr)
  | SFxnDef(_, var, sargl, exp) ->
      print_string(var^"("^sargl_string sargl^") = "^sexp_string exp^"\n")

  in List.iter print_sstmt sast

(*
let _ =
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  let sast = Semant.check ast in
  basic_print sast *)
```

9.1.9 codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR *)
(* Written primarily by G, some initial setup by Sitong & Sheron *)

module L = Llvm
open Ast
open Sast

module StringMap = Map.Make(String)
module StringSet = Set.Make(String)

type environment = {
ebuilder : L.llbuilder;
evars : L.llvalue StringMap.t; (* The storage associated with a given var
   *)
efxns : StringSet.t; (* Which names are original fxn definitions *)
esfxns : (L.llvalue * func_bind) StringMap.t; (* Decls for struct op fxns
   *)
ecurrent_fxn : L.llvalue; (* The current function *)
```

```
}

(* translate : Sast.program -> Llvm.module *)
let translate prog =
  let context    = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "SOS" in

  (* Get types from the context *)
  let i32_t      = L.i32_type     context
  and i8_t       = L.i8_type      context
  and i1_t       = L.i1_type      context
  and float_t    = L.float_type   context
  and void_t     = L.void_type    context
  and ptr_t      = L.pointer_type
  and struct_t   = L.struct_type  context in

  (* Convenient notation for GEP instructions, etc *)
  let l0         = L.const_int i32_t 0 in
  let l1         = L.const_int i32_t 1 in

  (* Return the LLVM type for a SOS type *)
  let rec ltype_of_typ = function
      Int      -> i32_t
    | Bool     -> i1_t
    | Float    -> float_t
    | Void     -> void_t
    (* An array is a pointer to a struct containing an array (as a pointer
   )
      and its length, an int *)
    | Array(t) -> ptr_t (struct_t [|ptr_t (ltype_of_typ t); i32_t|])
    | Struct(l) -> ptr_t (struct_t
       (Array.of_list (List.map (fun (tid, _) -> ltype_of_typ tid) l)))
    | Func(l, r) -> ptr_t (L.function_type (ltype_of_typ r) (Array.of_list
    (List.map ltype_of_typ l)))
    | EmptyArray-> raise (Failure "Unexpected empty array")

  in

  let printf_t : L.lltype =
      L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func : L.llvalue =
      L.declare_function "printf" printf_t the_module in

  (* External Functions *)
  let add_external_fxn env (decl, name) =
      let formals, rt = match decl with Func(formals, rt) -> formals, rt
        | _ -> raise (Failure "Unexpected external function decl") in
      let ftype = L.function_type (ltype_of_typ rt)
        (Array.of_list (List.map (fun t -> ltype_of_typ t) formals)) in
      let lldecl = L.declare_function name ftype the_module in
      { env with evars = StringMap.add name lldecl env.evars ;
```

```
        efxns = StringSet.add name env.efxns }
in

(* Setup main function *)
let main =
    L.define_function "main" (L.function_type i32_t [||]) the_module
in

(* Add a variable llvalue to environment.evars *)
let add_variable env nm lv =
     {env with evars = StringMap.add nm lv env.evars }
in

(* Get a variable's llvalue from environment.evars *)
let get_variable env nm = if StringMap.mem nm env.evars then StringMap.
 find nm env.evars
  else raise (Failure ("Unexpected variable name "^nm))
in

(* Add a function declaration to environment.efxns *)
let add_function env nm llv =
    {env with evars = StringMap.add nm llv env.evars;
     efxns = StringSet.add nm env.efxns }
in

(* Add a formal argument llvalue to environment.evars *)
let add_formal env (ty, nm) param =
  L.set_value_name nm param;
  let local = L.build_alloca (ltype_of_typ ty) nm env.ebuilder in
  ignore (L.build_store param local env.ebuilder);
  add_variable env nm local
in

 (* Operator Maps *)
 let opstr = function
   Add -> "Add"
 | Sub -> "Sub"
 | Mul -> "Mul"
 | MMul-> "MMul"
 | Div -> "Div"
 | Mod -> "Mod"
 | Eq -> "Eq"
 | Neq -> "Neq"
 | Less -> "Less"
 | Greater -> "Greater"
 | LessEq -> "LessEq"
 | GreaterEq -> "GreaterEq"
 | And -> "And"
 | Or -> "Or"
 | Of -> "Of"
 | Concat -> "Concat"
 | Seq -> "Seq"
 in
```

```
let make_opmap l =
  List.fold_left (fun map (op, fxn) -> StringMap.add (opstr op) fxn map
)
   StringMap.empty l
in

let int_map = make_opmap
[(Add, L.build_add); (Sub, L.build_sub);
 (Mul, L.build_mul); (Div, L.build_sdiv);
 (Eq, L.build_icmp L.Icmp.Eq); (Neq, L.build_icmp L.Icmp.Ne);
 (Mod, L.build_srem);
 (Less, L.build_icmp L.Icmp.Slt); (Greater, L.build_icmp L.Icmp.Sgt);
 (LessEq, L.build_icmp L.Icmp.Sle); (GreaterEq, L.build_icmp L.Icmp.Sge
)
  ]
in

let float_map = make_opmap
[(Add, L.build_fadd); (Sub, L.build_fsub);
 (Mul, L.build_fmul); (Div, L.build_fdiv);
 (Eq, L.build_fcmp L.Fcmp.Oeq); (Neq, L.build_fcmp L.Fcmp.One);
 (Less, L.build_fcmp L.Fcmp.Olt); (Greater, L.build_fcmp L.Fcmp.Ogt);
 (LessEq, L.build_fcmp L.Fcmp.Ole); (GreaterEq, L.build_fcmp L.Fcmp.Oge
)
  ]
in

(* Creates a loop that increments i by 1 each iteration,
 * and branches if i >= length.
 * i_addr : the address of the int to be iterated on
 * length : the value of i to branch at
 * nm     : the name of the iterated variable, to make the LL readable
 * build  : a llvalue -> llbuilder -> llbuilder that builds all the
statements using i
 * loop_bb: the basic block to build in
 * end_bb : the basic block to go to *)
let build_loop i_addr length nm loop_bb end_bb build =
  let builder = L.builder_at_end context loop_bb in
  let i = L.build_load i_addr "i" builder in
  let builder = build i builder in
  ignore(L.build_store (L.build_add i l1 nm builder) i_addr builder);
  let i = L.build_load i_addr nm builder in
  ignore (L.build_cond_br (L.build_icmp L.Icmp.Slt i length "tmp"
builder)
    loop_bb end_bb builder);
in

(* General shorthand *)
let build_zero builder nm =
  let addr = L.build_alloca i32_t nm builder in
  ignore(L.build_store l0 addr builder);
  addr
in
```

```
let build_param builder decl typ n nm =
    let param = (Array.get (L.params decl) n) in
    L.set_value_name nm param ;
    let local = L.build_alloca typ nm builder in
    ignore (L.build_store param local builder);
    L.build_load local nm builder
 in

(* Array operataions *)
let build_array_load data idx nm builder =
  let elref = L.build_gep data [|idx|] (nm^"ref") builder in
  L.build_load elref nm builder
in

let build_array_store data idx llv builder =
  let ref = L.build_gep data [|idx|] "storeref" builder in
  ignore (L.build_store llv ref builder)
in

let build_array_struct lltyp data length nm env =
  let arr_struct = L.build_malloc (L.element_type lltyp) nm env.
ebuilder in
  let data_addr = L.build_gep arr_struct [|l0; l0|] (nm^"data") env.
ebuilder in
  let len_addr  = L.build_gep arr_struct [|l0; l1|] (nm^"len")  env.
ebuilder in
  ignore (L.build_store data  data_addr env.ebuilder);
  ignore (L.build_store length len_addr env.ebuilder);
  arr_struct
in

let build_array_data builder lv nm =
  let data_ref = L.build_gep lv [|l0; l0|] (nm^"ref") builder in
  L.build_load data_ref nm builder
in

let build_array_len builder lv nm =
  let lenref = L.build_gep lv [|l0; l1|]
   (nm^"ref") builder in
  L.build_load lenref nm  builder
in

let build_of t2 ll1 ll2 env =
  (* Get ll2's length *)
  let len = build_array_len env.ebuilder ll2 "len" in

  (* Compute new length *)
  let n = L.build_mul ll1 len "oflen" env.ebuilder in
  (* Pre-GEP the array *)
  let old_data = build_array_data env.ebuilder ll2 "olddata" in

  (* Create a new array *)
  let el_typ = match t2 with Array(et) -> et | _ -> Void in
  let data = L.build_array_malloc (ltype_of_typ el_typ) n
```

```
    "arrdata" env.ebuilder in
  (* Set up loop *)
  let i_addr = build_zero env.ebuilder "i" in
  let j_addr = build_zero env.ebuilder "j" in
  let loop_bb = L.append_block context "loop" env.ecurrent_fxn in
  let inner_bb = L.append_block context "inner" env.ecurrent_fxn in
  let continue_bb = L.append_block context "continue" env.ecurrent_fxn
in
  ignore (L.build_br inner_bb env.ebuilder);

  (* Inner loop *)
  build_loop j_addr len "j" inner_bb loop_bb
    (fun j builder ->
     let i = L.build_load i_addr "i" builder in
     let el = build_array_load old_data j "el" builder in
     build_array_store data i el builder ;
     ignore(L.build_store (L.build_add i l1 "i" builder) i_addr builder
);
     builder
    ) ;

  (* Outer loop *)
  let builder = L.builder_at_end context loop_bb in
  let i = L.build_load i_addr "i" builder in
  ignore (L.build_store l0 j_addr builder);
  ignore (L.build_cond_br (L.build_icmp L.Icmp.Slt i n "tmp" builder)
    inner_bb continue_bb builder);

  (* Continue *)
  let builder = L.builder_at_end context continue_bb in
  let env = { env with ebuilder = builder } in

  (* Create array struct *)
  let arr_struct = build_array_struct (ltype_of_typ t2) data n "arr"
env in
  arr_struct, env
in

let build_concat t2 ll1 ll2 env =
  (* Get lengths *)
  let len1 = build_array_len env.ebuilder ll1 "len1" in
  let len2 = build_array_len env.ebuilder ll2 "len2" in
  let n = L.build_add len1 len2 "n" env.ebuilder in

  (* Pre-GEP the arrays *)
  let data1 = build_array_data env.ebuilder ll1 "data1" in
  let data2 = build_array_data env.ebuilder ll2 "data2" in

  (* Create a new array *)
  let el_typ = match t2 with Array(et) -> et | _ -> Void in
  let data = L.build_array_malloc (ltype_of_typ el_typ) n
    "data" env.ebuilder in
  (* Set up loop *)
  let i_addr = build_zero env.ebuilder "i" in
```

```
  let j_addr = build_zero env.ebuilder "j" in

  let loop1 = L.append_block context "loop1" env.ecurrent_fxn in
  let inbtw = L.append_block context "inbtw" env.ecurrent_fxn in
  let loop2 = L.append_block context "loop2" env.ecurrent_fxn in
  let contb = L.append_block context "contb" env.ecurrent_fxn in
  ignore (L.build_br loop1 env.ebuilder);

  let make_concat_loop sbb tbb from_data len =
    build_loop j_addr len "j" sbb tbb
    (fun j builder ->
    let i = L.build_load i_addr "i" builder in
    let el = build_array_load from_data j "el" builder in
    build_array_store data i el builder ;
    ignore (L.build_store (L.build_add i l1 "tmp" builder)
            i_addr builder) ;
    builder )
  in
  (* Loop 1 *)
  make_concat_loop loop1 inbtw data1 len1 ;

  let builder = L.builder_at_end context inbtw in
  ignore(L.build_store l0 j_addr builder);
  ignore(L.build_br loop2 builder);
  (* Loop 2 *)
  make_concat_loop loop2 contb data2 len2 ;

  (* Continue *)
  let builder = L.builder_at_end context contb in
  let env = { env with ebuilder = builder } in
  (* Create array struct *)
  let arr_struct = build_array_struct (ltype_of_typ t2) data n "arr"
env in
  arr_struct, env
in

(* Struct ops *)
(* Finds the integer field index *)
let find_field struct_typ field =
    let sargl = match struct_typ with Struct(l) -> l | _  -> [] in
    let rec find_field_inner sargl field n = match sargl with sarg ::
tl ->
      let (_, nm) = sarg in
      if nm = field then n else find_field_inner tl field n+1
    | _ -> raise (Failure "Field not found")
    in
    find_field_inner sargl field 0
in

let build_struct_field builder lv n nm =
  let ref = L.build_gep lv [|l0; L.const_int i32_t n|] (nm^"ref")
builder in
  L.build_load ref nm builder
in
```

```
let build_struct_store builder lv s_lv n =
  let ref = L.build_gep s_lv [|l0; L.const_int i32_t n|] "ref" builder
in
  ignore (L.build_store lv ref builder)
in

let dot_product stype slist env =
  let atype = match slist with
    (hd, _) :: _ -> if hd = Float then Float else Int
  | _ -> Float in
  let len = List.length slist in

  let name = "__dot"^(if atype=Float then "f" else "i")^(string_of_int
len) in

  if StringMap.mem name env.esfxns then
    StringMap.find name env.esfxns, env
  else (* Make new function *)
    let ltype = ltype_of_typ stype in
    let formals = [|ltype; ltype|] in
    let ftype = L.function_type (ltype_of_typ atype) formals in
    let decl = L.define_function name ftype the_module in
    let builder = L.builder_at_end context (L.entry_block decl) in

    let bind = { ftype = atype; formals = [stype, "a"; stype, "b"] } in

    let a = build_param builder decl ltype 0 "a" in
    let b = build_param builder decl ltype 1 "b" in

    let res = L.build_alloca (ltype_of_typ atype) "dot" builder in
    let tmp = L.build_alloca (ltype_of_typ atype) "tmp" builder in
    ignore(L.build_store (if atype=Float then L.const_float (
ltype_of_typ Float) 0.0 else L.const_int i32_t 0) res builder) ;
    let map = (if atype=Float then float_map else int_map) in
    (if StringMap.mem "Mul" map && StringMap.mem "Add" map then () else
 raise (Failure("Could not find operator in dot_product"))) ;
    let opmul = StringMap.find "Mul" map in
    let opadd = StringMap.find "Add" map in
    let rec dot_prod n =
      if n < len then
        let aval = build_struct_field builder a n "aval" in
        let bval = build_struct_field builder b n "bval" in
        ignore (L.build_store (opmul aval bval "tmp" builder) tmp
builder);
        let tmpv = L.build_load tmp "tmp" builder in
        let resv = L.build_load res "res" builder in
        ignore (L.build_store (opadd tmpv resv "tmp" builder) res
builder);
        dot_prod (n+1)
      else ()
    in
    dot_prod 0 ;
```

```
    (* Return *)
    let resv = L.build_load res "res" builder in
    ignore (L.build_ret resv builder) ;
    (decl, bind), { env with esfxns = StringMap.add name (decl, bind)
env.esfxns }
in

let struct_sum stype slist op env =
  let atype = match slist with
    (hd, _) :: _ -> if hd = Float then Float else Int
  | _ -> Float in
  let len = List.length slist in

  let name = "__"^(if op=Add then "add" else "sub")^
   (if atype=Float then "f" else "i")^(string_of_int len) in

  if StringMap.mem name env.esfxns then
    StringMap.find name env.esfxns, env
  else (* Make new function *)
    let ltype = ltype_of_typ stype in
    let formals = [|ltype; ltype|] in
    let ftype = L.function_type ltype formals in
    let decl = L.define_function name ftype the_module in
    let builder = L.builder_at_end context (L.entry_block decl) in

    let bind = { ftype = stype; formals = [stype, "a"; stype, "b"] } in

    let a = build_param builder decl ltype 0 "a" in
    let b = build_param builder decl ltype 1 "b" in

    let struc = L.build_malloc (L.element_type ltype) "ret" builder in
    let map = (if atype=Float then float_map else int_map) in
    (if StringMap.mem (opstr op) map then () else raise (Failure("Could
 not find op "^(opstr op)^" in struct_sum map")) );
    let sumop = StringMap.find (opstr op) map in
    let rec strsum n =
      if n < len then
        let aval = build_struct_field builder a n "aval" in
        let bval = build_struct_field builder b n "bval" in
        build_struct_store builder (sumop aval bval "tmp" builder)
struc n;
        strsum (n+1)
      else ()
    in
    strsum 0 ;

    (* Return *)
    ignore (L.build_ret struc builder) ;
    (decl, bind), { env with esfxns = StringMap.add name (decl, bind)
env.esfxns }
in

let struct_scale stype slist op env =
  let atype = match slist with
```

```
    (hd, _) :: _ -> if hd = Float then Float else Int
  | _ -> Float in
  let len = List.length slist in

  let name = "__"^(if op=Mul then "mul" else "div")^
   (if atype=Float then "f" else "i")^(string_of_int len) in

  if StringMap.mem name env.esfxns then
    StringMap.find name env.esfxns, env
  else (* Make new function *)
    let ltype = ltype_of_typ stype in
    let altype = ltype_of_typ atype in
    let formals = [|ltype; altype|] in
    let ftype = L.function_type ltype formals in
    let decl = L.define_function name ftype the_module in
    let builder = L.builder_at_end context (L.entry_block decl) in

    let bind = { ftype = stype; formals = [stype, "a"; stype, "b"] } in

    let a = build_param builder decl ltype 0 "a" in
    let b = build_param builder decl altype 1 "b" in

    let struc = L.build_malloc (L.element_type ltype) "ret" builder in
    let map = (if atype=Float then float_map else int_map) in
    (if StringMap.mem (opstr op) map then () else raise (Failure("Could
 not find op "^(opstr op)^" in struct_scale map")) );
    let sumop = StringMap.find (opstr op) map in
    let rec strscl n =
      if n < len then
        let aref = L.build_gep a [|l0; L.const_int i32_t n|] "aref"
builder in
        let aval = L.build_load aref "aval" builder in
        build_struct_store builder (sumop aval b "tmp" builder) struc n
;
        strscl (n+1)
      else ()
    in
    strscl 0 ;

    (* Return *)
    ignore (L.build_ret struc builder) ;
    (decl, bind), { env with esfxns = StringMap.add name (decl, bind)
env.esfxns }
in

let struct_eq stype slist op env =
  let atype = match slist with
    (hd, _) :: _ -> if hd = Float then Float else Int
  | _ -> Float in
  let len = List.length slist in

  let name = "__"^(if op=Eq then "eq" else "neq")^
    (if atype=Float then "f" else "i")^(string_of_int len) in
```

```
  if StringMap.mem name env.esfxns then
    StringMap.find name env.esfxns, env
  else (* Make new function *)
    let ltype = ltype_of_typ stype in
    let formals = [|ltype; ltype|] in
    let ftype = L.function_type (ltype_of_typ Bool) formals in
    let decl = L.define_function name ftype the_module in
    let builder = L.builder_at_end context (L.entry_block decl) in

    let bind = { ftype = Bool; formals =
        [stype, "a"; stype, "b"] } in

    let a = build_param builder decl ltype 0 "a" in
    let b = build_param builder decl ltype 1 "b" in

    let ret = L.build_alloca (ltype_of_typ Bool) "ret" builder in
    let eq = StringMap.find (opstr op) (if atype=Float then float_map
       else int_map) in
    let combine = if op=Eq then L.build_and else L.build_or in
    ignore(L.build_store (L.const_int i1_t (if op=Eq then 1 else 0))
ret builder);
    let rec streq n =
      if n < len then
        let aval = build_struct_field builder a n "aval" in
        let bval = build_struct_field builder b n "bval" in
        let rval = L.build_load ret "rval" builder in
        ignore(L.build_store
          (combine rval (eq aval bval "eq" builder) "ret" builder) ret
builder);
        streq (n+1)
      else ()
    in streq 0 ;

    (* Return *)
    let rv = L.build_load ret "rval" builder in
    ignore(L.build_ret rv builder) ;
    (decl, bind), { env with esfxns = StringMap.add name (decl, bind)
env.esfxns }
in

let mat_mul rtype slist1 slist2 env =
  let atype = match slist1 with
    (hd, _) :: _ -> if hd = Float then Float else Int
  | _ -> Float in
  let size = List.length slist1 in
  let int_sqrt n =
    let rec int_sqrt_inner n m =
      if m * m = n then m
      else if m * m < n then int_sqrt_inner n (m+1)
      else raise (Failure "Unexpected struct size")
    in int_sqrt_inner n 1
  in
  let n = int_sqrt size in
  let m = List.length slist2 in
```

```
  let name = "__"^(if m=n then "vec" else "mat")^
    (if atype=Float then "f" else "i")^(string_of_int n) in
  if StringMap.mem name env.esfxns then
    StringMap.find name env.esfxns, env
  else (* Make new function *)
    let rltype = ltype_of_typ rtype in
    let ltype1 = ltype_of_typ (Struct(slist1)) in
    let ltype2 = ltype_of_typ (Struct(slist2)) in
    let altype = ltype_of_typ atype in
    let formals = [|ltype1; ltype2|] in
    let ftype = L.function_type rltype formals in
    let decl = L.define_function name ftype the_module in
    let builder = L.builder_at_end context (L.entry_block decl) in

    let bind =
     { ftype = rtype; formals = [Struct(slist1), "A"; Struct(slist2), "
B"] } in

    let a = build_param builder decl ltype1 0 "A" in
    let b = build_param builder decl ltype2 1 "B" in

    let struc = L.build_malloc (L.element_type rltype) "ret" builder in
    let map = (if atype=Float then float_map else int_map) in
    let sumop = StringMap.find (opstr Add) map in
    let mulop = StringMap.find (opstr Mul) map in
    let height = n in
    let width = (if m=n then 1 else n) in
    let tmp = L.build_alloca altype "tmp" builder in
    let rec mmul i j =
      if i < width then
      if j < height then (
      ignore(L.build_store (if atype=Float then L.const_float altype
0.0 else L.const_int altype 0) tmp builder) ;
      let rec mmul_inner k =
        if k < height then (
        let aval = build_struct_field builder a (j+k*n) "aval" in
        let bval = build_struct_field builder b (k+i*n) "bval" in
        let mval = mulop aval bval "tmp2" builder in
        let tval = L.build_load tmp "tval" builder in
        ignore (L.build_store (sumop mval tval "tmp" builder) tmp
builder);
        mmul_inner (k+1))
        else ()
      in mmul_inner 0 ;
      let tval = L.build_load tmp "tval" builder in
      build_struct_store builder tval struc (j+i*n);
      mmul (i+1) j )
      else (* j >= height *) ()
      else (* i >= width *)  mmul 0 (j+1)
    in
    mmul 0 0 ;

    (* Return *)
```

```
    ignore (L.build_ret struc builder) ;
    (decl, bind), { env with esfxns = StringMap.add name (decl, bind)
env.esfxns }
in

(* Binops *)
let rec binop op rt t1 t2 ll1 ll2 env =
  if op = Of then build_of t2 ll1 ll2 env
  else if op = Concat then build_concat t2 ll1 ll2 env
  else
  (match (t1, t2) with
    (Bool, Bool) ->
      let llop = match op with
        Or -> L.build_or
      | And -> L.build_and
      | _ -> raise(Failure "Unexpected boolean operator") in
      llop ll1 ll2 "tmp" env.ebuilder, env
  | (Int, Int) -> StringMap.find (opstr op) int_map ll1 ll2 "tmp"
      env.ebuilder, env
  | (Float, Float) -> StringMap.find (opstr op) float_map ll1 ll2
      "tmp" env.ebuilder, env
  | (Struct(l1), Struct(l2)) -> let (decl, _), env =
    (match op with
       Mul -> dot_product t1 l1 env
     | MMul-> mat_mul rt l1 l2 env
     | Eq  -> struct_eq t1 l1 op env
     | Neq -> struct_eq t1 l1 op env
     | Add -> struct_sum t1 l1 op env
     | Sub -> struct_sum t1 l1 op env
     | _ -> raise (Failure("Unexpected struct operator "^(opstr op))) )
 in
    L.build_call decl [|ll1; ll2|] "result" env.ebuilder, env
  | (Array(el_typ), _) ->
    let len = build_array_len env.ebuilder ll1 "len" in

    (* Pre-GEP the array *)
    let argdata = build_array_data env.ebuilder ll1 "argdata" in

    (* Create a new array *)
    let rtel_typ = match rt with Array(et) -> et | _ -> Void in
    let data = L.build_array_malloc (ltype_of_typ rtel_typ) len
      "arrdata" env.ebuilder in
    (* Set up loop *)
    let i_addr = build_zero env.ebuilder "i" in
    let loop_bb = L.append_block context "loop" env.ecurrent_fxn in
    let cont_bb = L.append_block context "cont" env.ecurrent_fxn in
    ignore (L.build_br loop_bb env.ebuilder);
    (* Build loop *)
    build_loop i_addr len "i" loop_bb cont_bb (
      fun i builder ->
      let v = build_array_load argdata i "v" builder in
      let fenv = { env with ebuilder = builder } in
      let llv, fenv = binop op rtel_typ el_typ t2 v ll2 fenv in
      let builder = fenv.ebuilder in
```

```
      build_array_store data i llv builder;
      builder ) ;
    (* Continue *)
    let builder = L.builder_at_end context cont_bb in
    let env = { env with ebuilder = builder } in
    (* Create array struct *)
    let arr_struct = build_array_struct (ltype_of_typ rt) data len "arr
" env in
    arr_struct, env

  | (_, Array(el_typ)) ->
    let len = build_array_len env.ebuilder ll2 "len" in

    (* Pre-GEP the array *)
    let argdata = build_array_data env.ebuilder ll2 "argdata" in

    (* Create a new array *)
    let rtel_typ = match rt with Array(et) -> et | _ -> Void in
    let data = L.build_array_malloc (ltype_of_typ rtel_typ) len
      "arrdata" env.ebuilder in
    (* Set up loop *)
    let i_addr = build_zero env.ebuilder "i" in
    let loop_bb = L.append_block context "loop" env.ecurrent_fxn in
    let cont_bb = L.append_block context "cont" env.ecurrent_fxn in
    ignore (L.build_br loop_bb env.ebuilder);
    (* Build loop *)
    build_loop i_addr len "i" loop_bb cont_bb (
      fun i builder ->
      let v = build_array_load argdata i "v" builder in
      let fenv = { env with ebuilder = builder } in
      let llv, fenv = binop op rtel_typ t1 el_typ ll1 v fenv in
      let builder = fenv.ebuilder in
      build_array_store data i llv builder;
      builder ) ;
    (* Continue *)
    let builder = L.builder_at_end context cont_bb in
    let env = { env with ebuilder = builder } in
    (* Create array struct *)
    let arr_struct = build_array_struct (ltype_of_typ rt) data len "arr
" env in
    arr_struct, env
  | (Struct(l1), _) -> let (decl, _), env = struct_scale t1 l1 op env
    in L.build_call decl [|ll1; ll2|] "result" env.ebuilder, env
  | _ -> raise (Failure "Unsupported operation")
  )
in

(* Construct code for an expression
   Return its llvalue and the updated builder *)
let rec expr env sexpr =
  let (t, e) = sexpr in match e with
  (* Literals *)
  SIntLit(i) -> L.const_int i32_t i, env
| SFloatLit(f) -> L.const_float_of_string float_t f, env
```

```
  | SBoolLit(b) -> L.const_int i1_t (if b then 1 else 0), env
  | SArrayCon(expl) ->
    let n = List.length expl in
    let el_typ = match t with Array(et) -> et | _ -> Void in
    (* Create data *)
    let data = L.build_array_malloc
      (ltype_of_typ el_typ)
      (L.const_int i32_t n) "arrdata" env.ebuilder in
    let (_, env) = List.fold_left
      (fun (n, env) sx ->
        let (lv, env) = expr env sx in
        build_array_store data (L.const_int i32_t n) lv env.ebuilder;
        (n+1, env) ) (0, env) expl in
    (* Create struct *)
    let arr_struct = build_array_struct (ltype_of_typ t) data
      (L.const_int i32_t n) "arr" env in
    arr_struct, env

  | SStruct(nm, expl) ->
    let struc = L.build_malloc (L.element_type (ltype_of_typ t)) nm env.
  ebuilder in
    let rec set_fields env n = function
      exp :: tl ->
        let fieldaddr = L.build_gep struc [|l0; L.const_int i32_t n|]
          "fieldaddr" env.ebuilder in
        let (lv, env) = expr env exp in
        ignore(L.build_store lv fieldaddr env.ebuilder);
        set_fields env (n+1) tl
      | [] -> env
    in
    let env = set_fields env 0 expl in
    struc, env

    (* Access *)
  | SVar(nm) ->
    if nm="void" then l0, env else (* Void id *)
    if StringSet.mem nm env.efxns then (* Global fxn name, don't load *)
    get_variable env nm, env
    else (* All other variables *)
    (L.build_load (get_variable env nm) nm env.ebuilder),env

  | SArrayAccess(arr_exp, idx) ->
    let (arr, env) = expr env arr_exp in
    let (idx_lv, env) = expr env idx in
    (* Access the struct pointer, then the field *)
    let elref = L.build_gep arr [|l0; l0|]
      ("dataref") env.ebuilder in
    (* Access data *)
    let d = L.build_load elref ("data") env.ebuilder in
    build_array_load d idx_lv "el" env.ebuilder, env

  | SArrayLength (arr_exp) -> let (arr, env) = expr env arr_exp in
    build_array_len env.ebuilder arr "len", env
```

```
  | SStructField (str_exp, fl) ->
      let (stype, _) = str_exp in
      let (struc, env) = expr env str_exp in
      let idx = find_field stype fl in
      let adr = L.build_gep struc [|l0; L.const_int i32_t idx|]
        "fieldadr" env.ebuilder in
        L.build_load adr (fl) env.ebuilder, env

    (* Definitions *)
  | SVarDef(ty, nm, ex) ->
      let var = L.build_alloca (ltype_of_typ ty) nm env.ebuilder in
      let env = add_variable env nm var in
      expr env (t, SAssign(nm, ex)) (* Bootstrap off Assign *)

    (* Assignments *)
  | SAssign(nm, ex) ->
      let ex' = expr env ex in
      let (lv, env) = ex' in
      ignore(L.build_store lv (get_variable env nm) env.ebuilder); ex'

  | SAssignStruct (str_exp, fl, ex) ->
      let (stype, _) = str_exp in
      let (struc, env) = expr env str_exp in
      let ex' = expr env ex in
      let (lv, env) = ex' in
      (* Find field index *)
      let idx = find_field stype fl in
      build_struct_store env.ebuilder lv struc idx; ex'

  | SAssignArray (arr_exp, idx, ex) ->
      let (arr, env) = expr env arr_exp in
      let data = build_array_data env.ebuilder arr "dataref" in
      let (i, env) = expr env idx in
      let (el, env) = expr env ex in
      build_array_store data i el env.ebuilder; (el, env)

    (* Operators *)
  | SUop (op, exp) ->
      let (l, env) = expr env exp in
      (match op with
        Neg when t = Float -> L.build_fneg
      | Neg                -> L.build_neg
      | Not                -> L.build_not) l "tmp" env.ebuilder, env

  | SBinop(exp1, op, exp2) ->
      (match op with
        Seq ->
          let (_, env) = expr env exp1 in
          expr env exp2
      | _ ->
        let (t1, _) = exp1 in let (t2, _) = exp2 in
        let (ll1, env) = expr env exp1 in
        let (ll2, env) = expr env exp2 in
        binop op t t1 t2 ll1 ll2 env
```

```
    )

  (* Function application *)
 (* Special functions *)
| SFxnApp((_, SVar("printf")), [e]) ->
   let float_format_str =
    L.build_global_stringptr "%g\n" "fmt" env.ebuilder in
   let arg, env  = expr env e in
   let dval = L.build_fpext arg (L.double_type context) "fmtp" env.
ebuilder in
   L.build_call printf_func [| float_format_str; dval |]
     "printf" env.ebuilder, env
| SFxnApp((_, SVar("print")), [e]) ->
   let int_format_str =
    L.build_global_stringptr "%d\n" "fmt" env.ebuilder in
   let arg, env = expr env e in
   L.build_call printf_func [| int_format_str ; arg |]
     "printf" env.ebuilder, env


| SFxnApp((_, SVar("copy")), [e]) ->
   let (ctype, _) = e in
   let arg, env = expr env e in
   (
   match ctype with
     Array(atype)    ->
       let n = build_array_len env.ebuilder arg "len"in
       (* Pre-GEP the array *)
       let cdata = build_array_data env.ebuilder arg "cdata" in

       (* Create a new array *)
       let data = L.build_array_malloc (ltype_of_typ atype) n
         "arrdata" env.ebuilder in
       (* Set up loop *)
       let i_addr = build_zero env.ebuilder "i" in
       let loop_bb = L.append_block context "loop" env.ecurrent_fxn in
       let continue_bb = L.append_block context "continue" env.
ecurrent_fxn in
       ignore (L.build_br loop_bb env.ebuilder);

       (* Loop *)
       build_loop i_addr n "i" loop_bb continue_bb
         (fun i builder ->
          let el = build_array_load cdata i "el" builder in
          build_array_store data i el builder ; builder ) ;

      (* Continue *)
      let builder = L.builder_at_end context continue_bb in
      let env = { env with ebuilder = builder } in
      (* Create array struct *)
      let arr_struct = build_array_struct (ltype_of_typ t) data n
        "arr" env in
      arr_struct, env
```

```
    | Struct(sfields) ->
       let len = List.length sfields in
       let name = "__copy"^(string_of_int len) in

       let (decl, _), env = if StringMap.mem name env.esfxns
       then StringMap.find name env.esfxns, env
       else (* Make a new copy fxns *)
         let formals = [|ltype_of_typ ctype|] in
         let ftype = L.function_type (ltype_of_typ ctype) formals in
         let decl = L.define_function name ftype the_module in
         let builder = L.builder_at_end context (L.entry_block decl) in

         let bind = { ftype = ctype; formals = [ctype, "to_copy"] } in

         let tocopy = build_param builder decl (ltype_of_typ ctype) 0 "
to_copy" in

         (* Create a new struct *)
         let struc = L.build_malloc (L.element_type (ltype_of_typ ctype)
)
           "struct" builder in
         let rec copy_struct n =
           if n < len then
           let fl = build_struct_field builder tocopy n "fl" in
           build_struct_store builder fl struc n;
           copy_struct (n+1)
           else ()
         in
         copy_struct 0 ;

         (* Return *)
         ignore (L.build_ret struc builder) ;
         (decl, bind), { env with esfxns = StringMap.add name (decl,
bind) env.esfxns }
       in
       L.build_call decl [|arg|] "copied" env.ebuilder, env


    | _ -> raise (Failure "Copy constructor only works on reference
types")
    )

(* Free instruction *)
| SFxnApp((_, SVar("free")), [e]) ->
    let (ctype, _) = e in
    let arg, env = expr env e in
    (
    match ctype with
      Array(_)    ->
       (* Need to free data as well as the structure *)
       let dataref = build_array_data env.ebuilder arg "data" in
       ignore(L.build_free dataref env.ebuilder);
    | _ -> () ) ;
    (* Free structure *)
```

```
    ignore(L.build_free arg env.ebuilder) ;
    l0, env

  (* General functions *)
| SFxnApp(fexp, args) ->
    let fdef, env = expr env fexp in
    let (fxntype, _) = fexp in
    let _, rt = match fxntype with Func(l, t) -> l, t | _ ->
      raise (Failure "Unexpected function type") in

    (* Get llvalues of args and accumualte env *)
    let (llargs_rev, env) = List.fold_left
      (fun (l, env) a -> let (ll, e) = expr env a in (ll::l, e))
      ([], env) args in
    let llargs = List.rev llargs_rev in
    let result = (match rt with
                    Void -> ""
                  | _ -> "fxn_result") in
    (* Normal function application *)
    L.build_call fdef (Array.of_list llargs) result env.ebuilder, env

  | SIterFxnApp(fexp, args) ->
    let fdef, env = expr env fexp in
    let (fxntype, _) = fexp in
    let fargs, rt = match fxntype with Func(l, t) -> l, t | _ ->
      raise (Failure "Unexpected function type") in

    (* Get llvalues of args and accumualte env *)
    let (llargs_rev, env) = List.fold_left
      (fun (l, env) a -> let (ll, e) = expr env a in (ll::l, e))
      ([], env) args in
    let llargs = List.rev llargs_rev in
    let result = (match rt with
                    Void -> ""
                  | _ -> "fxn_result") in

    (* Iterated fxn application *)
    let arr_args = List.map2 (fun (ty, _) fty -> ty=Array(fty) )
         args fargs in
    let rec find_first bools = function
      (hd :: tl) -> if List.hd bools then hd else find_first (List.tl
bools) tl
    | _ -> raise (Failure "Unexpected arguments")
    in let first = find_first arr_args llargs in
    let len = build_array_len env.ebuilder first "len" in
    (* Pre-GEP all the arrays *)
    let datalist = List.map2
      (fun llarg b -> if b then
        Some(build_array_data env.ebuilder llarg "data")
       else None) llargs arr_args in

    (* Create a new array *)
    let data = if t=Void then None else
      Some(L.build_array_malloc (ltype_of_typ rt) len
```

```
     "arrdata" env.ebuilder) in
   (* Set up loop *)
   let i_addr = build_zero env.ebuilder "i" in
   let loop_bb = L.append_block context "loop" env.ecurrent_fxn in
   let cont_bb = L.append_block context "continue" env.ecurrent_fxn in
   ignore (L.build_br loop_bb env.ebuilder);

   (* Loop *)
   build_loop i_addr len "i" loop_bb cont_bb
     (fun i builder ->
       let llargs_i = List.map2
        (fun llarg data_opt -> match data_opt with
          Some(data) -> build_array_load data i "el" builder
        | None -> llarg ) llargs datalist in
       let ret = L.build_call fdef (Array.of_list llargs_i) result
builder in
       (match data with
        Some(d) -> build_array_store d i ret builder |_->()) ; builder
) ;

  (* Continue *)
  let builder = L.builder_at_end context cont_bb in
  let env = { env with ebuilder = builder } in
  (* Create array struct *)
  (match data with Some(d) ->
  let arr_struct = build_array_struct (ltype_of_typ t) d len
    "arr" env in
  arr_struct, env
  | _ -> l0, env )

 (* Control flow *)
| SIfElse (eif, ethen, eelse) ->
   let (cond, env) = expr env eif in
   (* Memory to store the value of this expression *)
   let ret = (if t != Void then
     Some(L.build_alloca (ltype_of_typ t) "if_tmp" env.ebuilder)
     else None) in
   let merge_bb = L.append_block context "merge" env.ecurrent_fxn in
   let then_bb = L.append_block context "then" env.ecurrent_fxn in
   let else_bb = L.append_block context "else" env.ecurrent_fxn in
   ignore (L.build_cond_br cond then_bb else_bb env.ebuilder);

   let (thenv, then_env) = expr {env with ebuilder=(L.builder_at_end
context then_bb)} ethen in
   (match ret with Some(rv) ->
     ignore (L.build_store thenv rv then_env.ebuilder)
     | None -> () );
   ignore (L.build_br merge_bb then_env.ebuilder);

   let (elsev, else_env) = expr {env with ebuilder=(L.builder_at_end
context else_bb)} eelse in
   (match ret with Some(rv) ->
     ignore (L.build_store elsev rv else_env.ebuilder)
     | None -> () );
```

```
    ignore (L.build_br merge_bb else_env.ebuilder);

    let env ={env with ebuilder=(L.builder_at_end context merge_bb)} in
    (match ret with
      Some(rv) -> (L.build_load rv "if_tmp" env.ebuilder), env
    | None -> (L.const_int (ltype_of_typ Bool) 0), env)

  (* Type casting *)
| SCast (ex) -> let t_to = t in let (t_from, _) = ex in
    let normal_cast command =
      let (lv, env) = expr env ex in
      (command lv (ltype_of_typ t_to) "cast" env.ebuilder, env)
    in
    let il i = (Int, SIntLit(i)) in
    let fl f = (Float, SFloatLit(f)) in
    (
    match (t_to, t_from) with
      (Int, Float)  -> normal_cast L.build_fptosi
    | (Int, Bool)   -> expr env (Int, SIfElse(ex, il 1, il 0))
    | (Float, Int)  -> normal_cast L.build_sitofp
    | (Bool, Int)   -> expr env (Bool, SBinop(ex, Neq, il 0))
    | (Bool, Float) -> expr env (Bool, SBinop(ex, Neq, fl "0"))
    | (Void, _)     -> expr env ex
    | _             -> raise (Failure "Unknown type cast")
    )

in

(* Builds an SOS statement and returns the updated environment *)
let build_stmt env = function
  STypeDef(_) -> env (* Everything handled in semant *)
| SExpression(ex) -> let (_, env) = expr env ex
  in env
| SFxnDef(ty, nm, args, ex) ->
  let formal_types =
      Array.of_list (List.map (fun (t, _) -> ltype_of_typ t) args) in
  let ftype = L.function_type (ltype_of_typ ty) formal_types in
  let decl = L.define_function nm ftype the_module in
  let new_builder = L.builder_at_end context (L.entry_block decl) in

  (* Add function to fxns set *)
  let env = add_function env nm decl in

  let new_env = { env with ebuilder=new_builder } in
  let new_env = List.fold_left2 add_formal new_env args (
         Array.to_list (L.params decl)) in
  let new_env = {new_env with ebuilder = new_builder; ecurrent_fxn=decl
} in
  let (lv, ret_env) = expr new_env ex in

  (* End with a return statement *)
  ( if ty=Void then
  ignore (L.build_ret_void ret_env.ebuilder)
  else
```

```
    ignore (L.build_ret lv ret_env.ebuilder) );

    env
  in

  (* Build the main function, the entry point for the whole program *)
  let build_main stmts =
    (* Init the builder at the beginning of main() *)
    let builder = L.builder_at_end context (L.entry_block main) in
    (* Init the starting environment from exeternal functions *)
    let start_env = { ebuilder = builder; evars = StringMap.empty;
      efxns = StringSet.empty; esfxns = StringMap.empty;
      ecurrent_fxn = main } in
    let start_env = List.fold_left add_external_fxn start_env
      Semant.external_functions in
    (* Build the program *)
    let end_env = List.fold_left build_stmt start_env stmts in
    (* Add a return statement *)
    L.build_ret (L.const_int i32_t 0) end_env.ebuilder

  in
  ignore(build_main prog);
  the_module
```

9.1.10 util_math.c

```
#include <math.h>

float toradiansf(float x) {
    return (float) x * (M_PI / 180.0);
}

#ifdef BUILD_TEST
int main() {
    toradiansf(12.5);
    return 0;
}
#endif
```

9.1.11 util_opengl.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "GL/osmesa.h"

/* reference:
   https://github.com/freedesktop/mesa-demos/blob/master/src/osdemos/
   osdemo.c
   (mainly for gl_startRendering, write_ppm and gl_endRendering)
*/

#define maxpoints 50000
```

```
struct array {
    float *arr;
    int length;
};

OSMesaContext ctx;
void *buffer;

static void rendering_helper_init() {
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glMatrixMode(GL_MODELVIEW);
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glEnableClientState(GL_VERTEX_ARRAY);
   glEnableClientState(GL_COLOR_ARRAY);
   glColor4f(1.0, 1.0, 1.0, 1.0); //initalize color as white
}

static void rendering_helper_close() {
    glFinish();
}

/*
 * startRendering: an initalization that must be called before drawing
 * any image. Creates Mesa and OpenGL contexts and image buffer.
 */
void gl_startRendering(int width, int height) {
    ctx = OSMesaCreateContextExt(OSMESA_RGBA, 16, 0, 0, NULL);
    if (!ctx) {
        printf("OSMesaCreateContext failed!\n");
    }

    buffer = malloc( width * height * 4 * sizeof(GLubyte) );
    if (!buffer) {
        printf("Alloc image buffer failed!\n");
    }

    // Bind the buffer to the context and make it current
    if (!OSMesaMakeCurrent(ctx, buffer, GL_UNSIGNED_BYTE, width, height))
   {
        printf("OSMesaMakeCurrent failed!\n");
    }
    printf("startRendering...\n");
    rendering_helper_init();
}

/*
 * drawCurve: draws segments between a list of points,
 * meaning n-1 segments for n points
 *
 * points: an array of points, with a point (x,y) located at [2i, 2i+1]
 * colors: an array of colors, with a the RGBA values of a point located
   at [4i, 4i+1, 4i+2, 4i+3]
```

```
 * size_arr: the number of points
 * color_mode: 0 -> between points i and i+1, the color of the segment is
   the color of point i+1
 *             1 -> each point has its own color. The segment between each
    point is a gradient between point colors
 */
void gl_drawCurve(struct array *spoints, struct array *scolors, int
   color_mode) {
    if (spoints->length!=scolors->length/2) {
        fprintf( stderr, "Unable to draw: The length of points array and
   colors array mismatched!\n" );
        return;
    }
    glPushMatrix();

    if (color_mode == 0) {
        glShadeModel(GL_FLAT);
    }
    else {
        glShadeModel(GL_SMOOTH);
    }

    float points[maxpoints];
    memcpy(points, spoints->arr, sizeof(float)*spoints->length);
    float colors[maxpoints];
    memcpy(colors, scolors->arr, sizeof(float)*scolors->length);
    int size_arr = spoints->length;
    size_arr = size_arr/2;

    glVertexPointer(2, GL_FLOAT, 0, points);
    glColorPointer(4, GL_FLOAT, 0, colors);
    glDrawArrays(GL_LINE_STRIP, 0, size_arr);

    glPopMatrix();
}

/*
 * drawShape: draws segments between a list of points,
 * including the segment connecting the first and last point
 *
 * points: an array of points, with a point (x,y) located at [2i, 2i+1]
 * colors: an array of colors, with a the RGBA values of a point located
   at [4i, 4i+1, 4i+2, 4i+3]
 * size_arr: the number of points
 * color_mode: 0 -> between points i and i+1, the color of the segment is
   the color of point i+1
 *             1 -> each point has its own color. The segment between each
    point is a gradient between point colors
 * filed: 0 -> shape is not filled with color
 *        1 -> shape will be filled with color
 */
void gl_drawShape(struct array *spoints, struct array *scolors, int
   color_mode, int filled) {
    if (spoints->length!=scolors->length/2) {
```

```
        printf("%d %d", spoints->length, scolors->length);
        fprintf( stderr, "Unable to draw: The length of points array and
   colors array mismatched!\n" );
        return;
    }
    glPushMatrix();

    if (color_mode == 0) {
        glShadeModel(GL_FLAT);
    }
    else {
        glShadeModel(GL_SMOOTH);
    }

    float points[maxpoints];
    memcpy(points, spoints->arr, sizeof(float)*spoints->length);
    float colors[maxpoints];
    memcpy(colors, scolors->arr, sizeof(float)*spoints->length);
    int size_arr = spoints->length;;
    size_arr = size_arr/2;

    glVertexPointer(2, GL_FLOAT, 0, points);
    glColorPointer(3, GL_FLOAT, 0, colors);

    if (filled==1) {
        glDrawArrays(GL_POLYGON, 0, size_arr);
    }
    else {
        glDrawArrays(GL_LINE_LOOP, 0, size_arr);
    }

    glPopMatrix();
}

/*
 * drawPoint: draws all points without creating segments
 *
 * points: an array of points, with a point (x,y) located at [2i, 2i+1]
 * colors: an array of colors, with a the RGBA values of a point located
   at [4i, 4i+1, 4i+2, 4i+3]
 * size_arr: the number of points
 * point_size: the size of each point
 */
void gl_drawPoint(struct array *spoints, struct array *scolors, int
   point_size) {
    if (spoints->length!=scolors->length/2) {
        fprintf( stderr, "Unable to draw: The length of points array and
   colors array mismatched!\n" );
        return;
    }
    glPushMatrix();

    float points[maxpoints];
    memcpy(points, spoints->arr, sizeof(float)*spoints->length);
```

```
    float colors[maxpoints];
    memcpy(colors, scolors->arr, sizeof(float)*spoints->length);
    int size_arr = spoints->length;
    size_arr = size_arr/2;

    glVertexPointer(2, GL_FLOAT, 0, points);
    glColorPointer(3, GL_FLOAT, 0, colors);
    glPointSize(point_size);
    glDrawArrays(GL_POINTS, 0, size_arr);
    glPopMatrix();
}

static void gl_clearCanvas() {
    glMatrixMode(GL_MODELVIEW);
    glClear(GL_COLOR_BUFFER_BIT);
}

/*
 * write_ppm: saves drawing
 *
 * filename: file name
 * buffer:
 * width: canvas width
 * height: canvas height
 */
static void write_ppm(int fileNumber, const GLubyte *buffer, int width,
   int height) {
    char filename[50];
    char filenumasstr[50];
    sprintf(filenumasstr, "%d.ppm", fileNumber);
    strcpy(filename, "pic");
    strcat(filename, filenumasstr);
    const int binary = 0;
    FILE *f = fopen( filename, "w" );
    if (f) {
       int i, x, y;
       const GLubyte *ptr = buffer;
       if (binary) {
          fprintf(f,"P6\n");
          fprintf(f,"# ppm-file created by util_opengl.c\n");
          fprintf(f,"%i %i\n", width,height);
          fprintf(f,"255\n");
          fclose(f);
          f = fopen( filename, "ab" );  /* reopen in binary append mode */
          for (y=height-1; y>=0; y--) {
             for (x=0; x<width; x++) {
                i = (y*width + x) * 4;
                fputc(ptr[i], f);   /* write red */
                fputc(ptr[i+1], f); /* write green */
                fputc(ptr[i+2], f); /* write blue */
             }
          }
       }
       else {
```

```
          /*ASCII*/
          int counter = 0;
          fprintf(f,"P3\n");
          fprintf(f,"# ascii ppm file created by util_opengl.c\n");
          fprintf(f,"%i %i\n", width, height);
          fprintf(f,"255\n");
          for (y=height-1; y>=0; y--) {
             for (x=0; x<width; x++) {
                i = (y*width + x) * 4;
                fprintf(f, " %3d %3d %3d", ptr[i], ptr[i+1], ptr[i+2]);
                counter++;
                if (counter % 5 == 0)
                   fprintf(f, "\n");
             }
          }
       }
       fclose(f);
    }
}

/*
 * endRendering: closes OpenGL and Mesa contexts and saves drawing
 * by calling write_ppm
 */
void gl_endRendering(int width, int height, int fileNumber) {
    rendering_helper_close();
    write_ppm(fileNumber, buffer, width, height);
    free(buffer);
    OSMesaDestroyContext(ctx);
    printf("endRendering...\n");
}
/*
//sample program
//#ifdef BUILD_TEST
int main(int argc, char *argv[]) {
    startRendering();

    float p[]= {-.5, 0, .5, 0, 0, .5};
    struct array points;
    memcpy(points.arr, p, sizeof(p));
    points.length = 6;

    float c[] = {1.0, 0.5, 1.0, 1.0, 1.0, 0.5, 0, 1.0,  0.5, 1.0, 1.0,
   1.0};
    struct array colors;
    memcpy(colors.arr, c, sizeof(c));
    colors.length = 12;

    int fileNumber = 1;

    startRendering();

    drawCurve(points, colors, 1);
```

```
    glTranslatef(-.2,-.2,0);
    drawCurve(points, colors, 0);

    glTranslatef(-.2, -.2, 0);
    drawShape(points, colors, 1, 1);

    glTranslatef(.6, 0 , 0);
    drawShape(points, colors, 0, 0);

    glTranslatef(-.3, -.2, 0);
    drawPoint(points, colors, 10);

    endRendering(1);

    return 0;
}
//#endif
*/
```

9.1.12 Makefile

```
.PHONY : all
all: sos.native util_math.o util_opengl.o

# Creates the main compiler
sos.native:
  opam config exec -- \
  ocamlbuild -use-ocamlfind sos.native

util_math: util_math.c
  cc -lm -o util_math -DBUILD_TEST util_math.c

util_opengl: util_opengl.c
  cc -o util_opengl -DBUILD_TEST util_opengl.c -I/usr/local/include/ -L/
   user/local/lib/ -lOSMesa -lGLU -lm

.PHONY : clean

test:
  ./testall.sh

clean :
  ocamlbuild -clean
  rm util_math.o util_opengl.o
```

9.1.13 Dockerfile

```
# Sheron Wang
# Based on 20.04 LTS
FROM ubuntu:focal

# Set timezone:
RUN ln -snf /usr/share/zoneinfo/$CONTAINER_TIMEZONE /etc/localtime && echo
    $CONTAINER_TIMEZONE > /etc/timezone
```

```
RUN apt-get -yq update && \
    apt-get -y upgrade && \
    apt-get -yq --no-install-suggests --no-install-recommends install \
    ocaml \
    menhir \
    llvm-10 \
    llvm-10-dev \
    m4 \
    git \
    aspcud \
    ca-certificates \
    python \
    pkg-config \
    cmake \
    opam \
    python3 \
    python3-distutils \
    ninja-build

######################################################################
# for building MESA
######################################################################
# add environment variable
RUN export PATH="/usr/bin/python:$PATH"

# add Mako and meson dependency from python
RUN wget https://bootstrap.pypa.io/get-pip.py
RUN python3 get-pip.py
RUN pip install Mako
RUN pip install meson
RUN apt-get install libpciaccess-dev -y

# download and install newest libdrm
RUN wget https://dri.freedesktop.org/libdrm/libdrm-2.4.105.tar.xz
RUN tar xf libdrm-2.4.105.tar.xz && rm libdrm-2.4.105.tar.xz
WORKDIR libdrm-2.4.105/
RUN meson build/ && cd build && ninja && ninja install
WORKDIR ../
RUN rm -r libdrm-2.4.105/

# download mesa
RUN wget https://archive.mesa3d.org//mesa-20.3.5.tar.xz
RUN tar xf mesa-20.3.5.tar.xz && rm mesa-20.3.5.tar.xz
WORKDIR mesa-20.3.5

# add things to sources.list
RUN cp /etc/apt/sources.list /etc/apt/sources.list~
RUN sed -Ei 's/^# deb-src /deb-src /' /etc/apt/sources.list
RUN apt-get update

# add dependencies
RUN apt-get install -y libdrm-dev libxxf86vm-dev libxt-dev xutils-dev flex
    bison xcb libx11-xcb-dev libxcb-glx0 libxcb-glx0-dev xorg-dev libxcb-
```

```
   dri2-0-dev
RUN apt-get install -y libelf-dev libunwind-dev valgrind libwayland-dev
   wayland-protocols libwayland-egl-backend-dev
RUN apt-get install -y libxcb-shm0-dev libxcb-dri3-dev libxcb-present-dev
   libxshmfence-dev
RUN apt-get build-dep mesa -y

# build, compile and install
RUN meson build/ -Dosmesa=classic && ninja -C build/ && ninja -C build/
   install
WORKDIR ../

# add OSMesa to path
RUN echo "export LD_LIBRARY_PATH=/usr/local/lib/x86_64-linux-gnu/:
   $LD_LIBRARY_PATH" >> ~/.bashrc

# install MESA GLU if you need more advanced features
# RUN git clone https://gitlab.freedesktop.org/mesa/glu.git
# RUN cd glu && ./autogen.sh && ./configure --enable-osmesa --prefix=/usr/
   local/ && make && make install
# RUN rm -r glu mesa-20.3.5 get-pip.py

# add GLU to path
# RUN echo "export LD_LIBRARY_PATH=/usr/local/lib/:$LD_LIBRARY_PATH" >>
   ~/.bashrc

# install vim for testing
# RUN apt-get install vim -y

#####################################################################
# for building LLVM & others
#####################################################################
RUN ln -s /usr/bin/lli-10 /usr/bin/lli
RUN ln -s /usr/bin/llc-10 /usr/bin/llc

RUN opam init --disable-sandboxing -y
RUN opam install \
    llvm.10.0.0 \
    ocamlfind \
    ocamlbuild -y
RUN eval `opam config env`

WORKDIR /root

ENTRYPOINT ["opam", "config", "exec", "--"]

CMD ["bash"]
```

9.1.14 compile_exec.sh

```
#!/bin/sh

# Builds a .sos file to an executable file
# Requires that ./sos.native has been built
```

```
# Path to the LLVM interpreter
LLI="lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the SOS compiler
SOS="./sos.native"

if [ ! -f util_math.o ]
then
    echo "Could not find util_math.o"
    echo "Try \"make util_math.o\""
    exit 1
fi

filename=$1
basename=${filename%.sos}

$SOS $filename >${basename}.ll
$LLC -relocation-model=pic ${basename}.ll >${basename}.s
$CC -o ${basename}.exe ${basename}.s util_math.o -lm util_opengl.o -I/usr/
   local/include/ -L/user/local/lib/ -lOSMesa

rm ${basename}.ll ${basename}.s

./${basename}.exe
```

9.1.15 docker_connect.sh

```
#!/bin/sh

docker run --rm -it -v `pwd`:/home/sos -w=/home/sos sheronw1174/sos-env:
   latest
```

9.1.16 docker_image_fetching.sh

```
#!/bin/sh

if [ "$1" = "build" ]
then
    docker build . -t sheronw1174/sos-env
elif [ "$1" = "pull" ]
then
    docker pull sheronw1174/sos-env
else
    echo "usage: $0 [build|pull]"
fi
```

9.1.17 testall.sh

```
#!/bin/sh

# Regression testing script for SOS
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
#LLI="/usr/local/Cellar/llvm/9.0.1_2/bin/lli"
LLI="lli"

# Path to the LLVM compiler
#LLC="/usr/local/Cellar/llvm/9.0.1_2/bin/lli"
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the SOS compiler.  Usually "./sos.native"
# Try "_build/sos.native" if ocamlbuild was unable to create a symbolic
   link.
SOS="./sos.native"

OPENGL_FLAGS="-lOSMesa -lm"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.sos files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
  echo "FAILED"
  error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to
```

```
   difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
  SignalError "$1 differs"
  echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
  SignalError "$1 failed on $*"
  return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
  SignalError "failed: $* did not report an error"
  return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.sos//'`
    reffile=`echo $1 | sed 's/.sos$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${
   basename}.exe ${basename}.out" &&
    Run "$SOS" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s
   " &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "util_opengl.o" "
   util_math.o" "$OPENGL_FLAGS" &&
    Run "./${basename}.exe" > "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff
```

```
    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
  echo "OK"
  echo "###### SUCCESS" 1>&2
    else
  echo "###### FAILED" 1>&2
  globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.sos//'`
    reffile=`echo $1 | sed 's/.sos$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$SOS" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
  echo "OK"
  echo "###### SUCCESS" 1>&2
    else
  echo "###### FAILED" 1>&2
  globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
  k) # Keep intermediate files
      keep=1
      ;;
  h) # Help
      Usage
```

```
      ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in
   testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail

if [ ! -f util_opengl.o ]
then
    echo "Could not find util_opengl.o"
    echo "Try \"make util_opengl.o\""
    exit 1
fi

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.sos tests/fail-*.sos"
fi

for file in $files
do
    case $file in
  *test-*)
      Check $file 2>> $globallog
      ;;
  *fail-*)
      CheckFail $file 2>> $globallog
      ;;
  *)
      echo "unknown file type $file"
      globalerror=1
      ;;
    esac
done

exit $globalerror
```

9.1.18 helloworld.sos

```
import renderer.sos

a: int = 5
print(a)
```

```
p1: point = {-0.9, -0.9}
p2: point = {-0.9, -0.7}
p3: point = {-0.7, -0.7}
p4: point = {-0.7, -0.9}

point_arr : path = [p1, p2, p3, p4]

c1 : color = {255.0, 0.0, 0.0, 0.8}
c2 : color = {0.0, 255.0, 0.0, 0.8}
c3 : color = {0.0, 0.0, 255.0, 0.8}
c4 : color = {100.0, 100.0, 0.0, 0.8}
color_arr : colors = [c1, c2, c3, c4]

my_canvas : canvas = {400, 400, 2}

startCanvas(my_canvas)
drawShape(point_arr, color_arr, 0, 1)
endCanvas(my_canvas)
```

9.2 Library Programs

9.2.1 lib/affine.sos

```
import point.sos

struct mat2 = {a11: float, a21: float, a12: float, a22: float}
struct mat3 = {a11: float, a21: float, a31: float,
               a12: float, a22: float, a32: float,
               a13: float, a23: float, a33: float}
alias affine = mat3

affine_mul : (A: affine, v: point, w: float) -> point =
    to: point3 = {v.x, v.y, w};
    res: point3 = A ** to;
    free(to);
    if w == 0 || res.z == 1 then
    ret: point = {res.x, res.y};
    free(res) ; ret
    else
    ret: point = {res.x/res.z, res.y/res.z};
    free(res) ; ret

rotation : (r: float) -> mat2 =
    c: float = cos(r) ; s: float = sin(r);
    {c,s,-s,c}

scale : (sx: float, sy: float) -> mat2 =
    {sx,0.0,0.0,sy}

translate : (dx: float, dy: float) -> affine =
    {1.0,0.0,0.0, 0.0,1.0,0.0, dx,dy,1.0}

// For internal use only
```

```
// Functions not normally free their arguments like this
make_affine__ : (m: mat2) -> affine =
    ret: affine = {m.a11, m.a21, 0.0,
                   m.a12, m.a22, 0.0,
                   0.0, 0.0, 1.0};
    free(m) ; ret

rotation_aff : (r: float) -> affine = make_affine__(rotation(r))
scale_aff : (sx: float, sy: float) -> affine = make_affine__(scale(sx,sy))
```

9.2.2 lib/array.sos

```


fill_ints : (a: array int, i: int) -> void =
    if i < a.length then a[i] = i; fill_ints(a, i+1)
    else 0

// A very useful function that makes an array of consecutive ints
// Use with implicit iteration for very nice results
ints : (n : int) -> array int =
  if n <= 0 then (a: array int = [])
  else arr : array int = n of [0];
  fill_ints(arr, 0) ; arr
```

9.2.3 lib/color.sos

```
import math.sos

struct color = {r: float, g: float, b: float, a: float}

alias colors = array color

// Standard: r,g,b in [0, 1]
rgb: (r: float, g: float, b: float) -> color = {r,g,b,1.0}

// Hue/saturation/value: hsv in [0, 1]
hsv: (h: float, s: float, v: float) -> color =
   c: float = v * s ;
   hfac: float = modf(h*6.0, 2.0) ;
   x: float = c * (1.0-abs(hfac-1.0)) ;
   m: float = v - c ;
   hh: float = h*6.0 ;

   if hh < 1.0 then rgb(v,x+m,m) else
   if hh < 2.0 then rgb(x+m,v,m) else
   if hh < 3.0 then rgb(m,v,x+m) else
   if hh < 4.0 then rgb(m,x+m,v) else
   if hh < 5.0 then rgb(x+m,m,v) else
                    rgb(v,m,x+m)
```

9.2.4 lib/math.sos

```
// Some floating point ops

floor : (x: float) -> float =
    z: float = (y: int = x);
    if z <= x then z
    else z - 1.0

ceil : (x: float) -> float = -floor(-x)

frac : (x: float) -> float = x - floor(x)

max : (a: float, b: float) -> float = if a<b then b else a
min : (a: float, b: float) -> float = if a<b then a else b

clamp : (x: float, m: float, M: float) -> float = min(M, max(x, m))

abs : (x: float) -> float = if x < 0 then -x else x

// Returns the value y between 0 and m such that y = x+mn for an integer n
modf : (x: float, m: float) -> float = m * frac(x/m)

sin : (x: float) -> float = sinf(x)
cos : (x: float) -> float = cosf(x)
tan : (x: float) -> float = tanf(x)
asin : (x: float) -> float = asinf(x)
acos : (x: float) -> float = acosf(x)
atan : (x: float) -> float = atanf(x)
sqrt : (x: float) -> float = sqrtf(x)
toradians : (x: float) -> float = toradiansf(x)
```

9.2.5 lib/point.sos

```
import math.sos

struct point = {x: float, y: float}
struct point3 = {x: float, y: float, z: float}

sqrMagnitude : (p: point) -> float = p * p
magnitude : (p: point) -> float = sqrt(sqrMagnitude(p))

sqrDistance : (a: point, b: point) -> float =
   p: point = a - b;
   d: float = sqrMagnitude(p);
   free(p) ; d
distance : (a: point, b: point) -> float = sqrt(sqrDistance(a,b))
```

9.2.6 lib/random.sos

```
// Wichmann-Hill PRNG
// en.wikipedia.org/wiki/Wichmann-Hill
import math.sos
```

```
struct rng = {s1: int, s2: int, s3: int}

randf : (r: rng) -> float =
   r.s1 = 171 * r.s1 % 30269;
   r.s2 = 172 * r.s2 % 30307;
   r.s3 = 170 * r.s3 % 30323;
   frac(r.s1/30269.0 + r.s2/30307.0 + r.s3/30323.0)
```

9.2.7 lib/renderer.sos

```
import shape.sos
import color.sos

struct canvas = {width: int, height: int, file_number: int}

startCanvas: (c : canvas) -> void = gl_startRendering(c.width, c.height)

cvoid : () -> void = 0

drawHelper : (point_structs: path, color_structs: array color, numOfPoints
   : int, i: int,  points: array float, colors: array float) -> void =
    if i >= numOfPoints /*i>=numOfPoints-1?*/
    then cvoid()
    else
        px : float = point_structs[i].x;
        py : float= point_structs[i].y;
        points[2*i] = px;
        points[2*i+1] = py;

        cr : float = color_structs[i].r;
        cg : float = color_structs[i].g;
        cb : float= color_structs[i].b;
        ca : float= color_structs[i].a;
        colors[4*i] = cr;
        colors[4*i+1] = cg;
        colors[4*i+2] = cb;
        colors[4*i+3] = ca;

        drawHelper(point_structs, color_structs, numOfPoints, i + 1,
   points, colors)

drawPoints : (point_structs : path, color_structs : array color) -> void =
        numOfPoints : int = point_structs.length;
        points : array float = numOfPoints*2 of [0.0];
        colors : array float = numOfPoints*4 of [0.0];

        drawHelper(point_structs, color_structs, numOfPoints, 0,  points,
   colors);

        gl_drawPoint(points, colors, 2)

drawPath : (point_structs : path, color_structs : array color,  colorMode
```

```
   : int) -> void =
        numOfPoints : int= point_structs.length;
        points : array float= numOfPoints*2 of [0.0];
        colors : array float= numOfPoints*4 of [0.0];

        drawHelper(point_structs, color_structs, numOfPoints, 0,  points,
   colors);

        gl_drawCurve(points, colors, colorMode)


drawShape : (point_structs : path, color_structs : array color,  colorMode
    : int, filled : int) ->
    void =
        numOfPoints : int = point_structs.length;
        points : array float = numOfPoints*2 of [0.0];
        colors : array float = numOfPoints*4 of [0.0];

        drawHelper(point_structs, color_structs, numOfPoints, 0,  points,
   colors);

        gl_drawShape(points, colors, colorMode, filled)


endCanvas : (c : canvas) -> void = gl_endRendering(c.width, c.height, c.
   file_number)
```

9.2.8 lib/shape.sos

```
import point.sos

alias path = array point
alias shape = array point

// Wrappers to enable array iteration
copy_point : (p: point) -> point = copy(p)
free_point : (p: point) -> void = free(p)

// More convenient names
copy_path : (p: path) -> path = copy_point(p)
free_path : (p: path) -> void = free_point(p)

appendhelp_copyin : (in: path, from: path, i: int) -> void =
    if i<in.length then
    in[i] = copy(from[i+1]);
    appendhelp_copyin (in, from, i+1)
    else 0

appendhelp_tail : (p: path) -> path =
    tail: path = p.length-1 of [{0.0, 0.0}];
    appendhelp_copyin(tail, p, 0); tail

// Appends two paths, merging them at their endpoints, if needed
```

```
// Epsilon is the max distance that can be merged
append : (p1: path, p2: path, epsilon: float) -> path =
    if p1.length == 0 then copy_path(p2)
    else if p2.length == 0 then copy_path(p1)
    else
    merge: bool = sqrDistance(p1[p1.length-1], p2[0]) < epsilon*epsilon;
    p2c: path = (if merge then appendhelp_tail(p2) else p2);
    ret: path = copy_path(p1) @ copy_path(p2c);
    ret

reversedhelp: (in: path, from: path, i: int) -> void =
    if i < in.length then
    in[i].x = from[in.length-1-i].x;
    in[i].y = from[in.length-1-i].y;
    reversedhelp(in, from, i+1)
    else 0

// Creates a new array that is p reversed
reversed: (p : path) -> path =
    newpath : path = p.length of [{0.0, 0.0}];
    reversedhelp(newpath, p, 0);
    newpath

reversehelp : (p: path, i: int) -> void =
    if i < p.length/2 then
    q: point = p[i];
    p[i] = p[p.length-1-i];
    p[p.length-1-i] = q;
    reversehelp (p, i+1)
    else 0

// Reverses p in-place
reverse: (p: path) -> void = reversehelp(p, 0)
```

9.2.9 lib/size.sos

```
import shape.sos

struct minPoint = {float x, float y}

// scale rightward downward
// should based on a rectangular box aound the object and get that vertex?
    or just use opengl

min(a: point, b: minPoint) -> void =
    if a.x < b.x and a.y < b.y then b.x = a.x and b.y = a.y
    else 0

// shape size(float multiple) =
```

9.2.10 lib/std.sos

```
import vector.sos
import affine.sos
import renderer.sos
```

9.2.11 lib/transform.sos

```
import shape.sos
import math.sos
import vector.sos

// Rotates the given point by angle radians about the given point
// Either rotate clockwise (1) or counterclockwise (-1)
rotate : (p: point, angle: float, direction: int, about: point) -> void =
    //shifted
    px : float = p.x - about.x;
    py : float = p.y - about.y;

    if direction == -1
    then //counterclockwise
        p.x = (px*cos(angle) - py*sin(angle)) + about.x;
        p.y = (px*sin(angle) + py*cos(angle)) + about.y
    else if direction == 1
    then //clockwise
        p.x = (px*cos(angle) + py*sin(angle)) + about.x;
        p.y = (-px*sin(angle)+ py*cos(angle)) + about.y
    else //do no rotation
        p.x = p.x;
        p.y = p.y

// Translates the given point by the given vector
trans : (p: point, direction: vector) -> void =
    p.x = p.x + direction.x ; p.y = p.y + direction.y

// Scales the point by (sx, sy)
scale : (p: point, sx: float, sy: float) -> void =
    p.x = p.x * sx ; p.y = p.y * sy

// Performs rotate() on a new point
rotated : (p: point, angle: float, direction: int, about: point) -> point
   =
    q: point = copy(p) ;
    rotate(q, angle, direction, about) ;
    q

// Performs translate() on a new point
translated : (p: point, direction: vector) -> point =
    q: point = copy(p) ;
    trans(q, direction) ;
    q

scaled : (p: point, sx: float, sy: float) -> point =
    q: point = copy(p) ;
    scale(p, sx, sy) ;
```

```
    q
```

9.2.12 lib/vector.sos

```
struct vector = {x: float, y: float}
```

9.3 Example Programs

9.3.1 sample_programs/dragon.sos

```
import renderer.sos
import vector.sos
import transform.sos
import array.sos
import math.sos

// Creates a dragon curve of depth n
dragon: (n: int) -> path =
    if n == 0 // Base case
    then [point{0.0, 0.0}, point{1.0, 0.0}]
    else
    // Create two copies of the previous depths
    d1: path = dragon(n-1) ;
    d2: path = copy_path(d1) ;

    // Position d1
    s: float = sqrt(2.0)/2.0 ;
    rotate(d1, toradians(45.0), -1, {0.0, 0.0}) ;
    scale(d1, s, s) ;

    // Position d2
    rotate(d2, toradians(135.0), -1, {0., 0.}) ;
    scale(d2,s,s) ;
    trans(d2, {1., 0.}) ;
    reverse(d2) ;

    // Merge the paths
    r: path = append(d1, d2, 1.0) ;
    free_path(d1); free_path(d2); r

// Creates a rainbow color effect
rainbow: (r: int, len: int) -> color =
    h: float = (1.0*r)/len ;
    hsv(h, 0.8, 0.8)

// Render a 400px by 400px canvas, name the image pic0
my_canvas: canvas = {400, 400, 0}

// Start render
startCanvas(my_canvas)
d: path = dragon(7)
// Position the curve (0.4, 0.2 is approximately the center of mass of the
    curve for large n)
```

```
trans(d, {-0.4, -0.2})

// Draw it
drawPath(d, rainbow(ints(d.length), d.length), 0)
endCanvas(my_canvas)
```

9.3.2 sample_programs/drunk.sos

```
import renderer.sos
import random.sos

n: int = 100
p: path = n of [{0.0, 0.0}]
c: colors = n of [{0.5, 0.5, 0.5, 0.5}]
r: rng = {1,2,3}

drunk_walk : (i: int, p: path, c: colors, r: rng) -> void =
    if i < p.length then
      theta: float = randf(r) * 6.28319;
      d: float = randf(r)*0.1 + 0.02;
      dx: float = cos(theta)*d ; dy: float = sin(theta)*d ;
      p[i] = {p[i-1].x+dx, p[i-1].y+dy} ;
      dc: float = 0.1 ;
      dr: float = (randf(r) - 0.5) * dc ;
      dg: float = (randf(r) - 0.5) * dc ;
      db: float = (randf(r) - 0.5) * dc ;
      c[i] = rgb(c[i-1].r + dr, c[i-1].g+dg, c[i-1].b+db) ;
      drunk_walk(i+1,p,c,r)

    else void


my_canvas: canvas = {400, 400, 0}
startCanvas(my_canvas)

draw_walks : (count: int, p: path, c: colors, r: rng) -> void =
    if count > 0 then
    p[0] = {randf(r)*0.5-0.25, randf(r)*0.5-0.25};
    c[0] = hsv(randf(r), 0.8, 0.8) ;
    drunk_walk(1, p, c, r) ;
    drawPath(p,c,0) ;
    draw_walks(count-1,p,c,r)
    else void

draw_walks(20, p, c, r)

endCanvas(my_canvas)
```

9.3.3 sample_programs/lorenz.sos

```
import renderer.sos
import array.sos
```

```
alias path3 = array point3

create_lorenz : (p: path3, i: int, sigma: float, rho: float, beta: float)
   -> void =
    if i == 0 then p[i] = {0.1,0.1,0.1}; create_lorenz(p,i+1,sigma,rho,
   beta)
    else if i < p.length then
      q: point3 = p[i-1] ;
      dx: float = sigma * (q.y - q.x) ;
      dy: float = q.x * (rho - q.z) - q.y ;
      dz: float = q.x*q.y - beta * q.z ;
      dt: float = 0.005 ;
      p[i] = {q.x + dx*dt, q.y + dy*dt, q.z + dz*dt} ;
      create_lorenz(p,i+1,sigma,rho,beta)
    else void

len: int = 5000
l3: path3 = len of [{0.0,0.0,0.0}]
create_lorenz (l3, 0, 10, 28, 8.0/3)
rainbow: (r: int, len: int) -> color =
    h: float = (1.0*r)/len ;
    hsv(h, 0.8, 0.8)
c: colors = rainbow(ints(len), len+1000)


reduce : (p: point3) -> point = {p.x/30.0, p.y/30.0}

l: path = reduce(l3)

my_canvas : canvas = {400,400,1}

startCanvas(my_canvas)
drawPath(l, c, 0)
endCanvas(my_canvas)
```

9.3.4 sample_programs/square.sos

```
import renderer.sos

a: int = 5
print(a)

p1: point = {-0.5, -0.5}
p2: point = {-0.5, 0.5}
p3: point = {0.5, 0.5}
p4: point = {0.5, -0.5}
point_arr : path = [p1, p2, p3, p4]

c1 : color = {255.0, 0.0, 0.0, 0.8}
c2 : color = {0.0, 255.0, 0.0, 0.8}
c3 : color = {0.0, 0.0, 255.0, 0.8}
c4 : color = {100.0, 100.0, 0.0, 0.8}
color_arr : colors = [c1, c2, c3, c4]
```

```
canvas1 : canvas = {400, 400, 0}

startCanvas(canvas1)
drawShape(point_arr, color_arr, 0, 1)
endCanvas(canvas1)

canvas2 : canvas = {400, 400, 1}

startCanvas(canvas2)
drawShape(point_arr, color_arr, 0, 1)
endCanvas(canvas2)
```

9.3.5 sample_programs/tree.sos

```
import affine.sos
import renderer.sos

branchHelper : (p: path, i: int, A: affine) -> void =
    if i < p.length then
    p[i] = affine_mul(A, p[i-1]-p[i-2], 0) + p[i-1];
    branchHelper(p,i+1,A)
    else void

makeBranch : (n: int, A: affine) -> path =
    p: path = n of [{0.0,0.0}];
    p[1] = {0.0, 0.2};
    branchHelper(p, 2, A); p

powersHelper : (arr: array affine, i: int, A: affine) -> void =
    if i < arr.length then
    arr[i] = A ** arr[i-1];
    powersHelper(arr,i+1,A)
    else void

powers : (n: int, A: affine) -> array affine =
    arr: array affine = n of [copy(A)];
    powersHelper(arr, 1, A) ; arr

A: affine = translate(0, 0.2) ** rotation_aff(0.13) ** scale_aff(0.8, 0.8)
As: array affine = powers(8, A)
L: affine = scale_aff(0.36, 0.36) ** rotation_aff(0.66)
R: affine = scale_aff(0.49,0.49) ** rotation_aff(-0.78)
base: path = makeBranch(10, A)

white: colors = 10 of [rgb(1.0,1.0,1.0)]
green: colors = 10 of [rgb(0.0,0.8,0.0)]

my_canvas: canvas = {400,400,0}
startCanvas(my_canvas)

drawTree : (base: path, T: affine, A: affine, As: array affine, L: affine,
    R: affine, depth: int, c1: colors, c2: colors)
  -> void =
```

```
  if depth==0 then drawPath(affine_mul(T,base,1), c2, 0) else
  drawPath(affine_mul(T,base,1), c1, 0) ;
  drawTree(base, T ** As ** L, A, As, L, R, depth-1, c1, c2) ;
  drawTree(base, T ** As ** R, A, As, L, R, depth-1, c1, c2)

T: affine = {1.8,0.0,0.0,0.0,1.8,0.0,0.0,-0.8,1.0}
drawTree (base, T, A, As, L, R, 3, white, green)

endCanvas(my_canvas)
```

9.3.6 sample_programs/web.sos

```
import renderer.sos
import random.sos
import array.sos
import math.sos

build_ring : (i: int, r: int, radius: float) -> point =
    angle: float = (i*6.2831852)/r  + 0.134;
    {cos(angle) * radius, sin(angle) * radius}

build_rings : (n: int, r: int, radius_interval: float, rr: rng) -> array
   point =
    build_ring(ints(r), r, radius_interval * n * (randf(rr)*0.18+0.91))

build_points : (n : int, r : int, max_radius: float, rr: rng) -> array
   array point =
    build_rings (ints(n), r, max_radius/(n-1), rr)

connect_ring_inner : (i: int, a: array point, c: colors) -> void =
    if  i==a.length-1 then drawPath([a[i], a[0]], c, 0)
    else drawPath([a[i], a[i+1]], c, 0); connect_ring_inner(i+1,a,c)

connect_ring : (a: array point, c: colors) -> void =
    connect_ring_inner(0, a, c)

ring_iter : (w: array array point, c: colors, i: int, j: int, r: rng,
        f: func array array point, colors, int, int, rng -> void) -> void
   =
    if i < w.length then if j < w[i].length then
    f(w,c,i,j,r) ; ring_iter(w,c,i,j+1,r,f)
    else ring_iter(w,c,i+1,0,r,f) else void

connect_lines_in : (w: array array point, c: colors, i: int, j: int, r:
   rng) -> void =
    if i < w.length - 1 then
    drawPath([w[i][j], w[i+1][j]], c, 0)
    else drawPath([w[i][j], 5*w[i][j]], c, 0)

connect_lines : (w: array array point, c: colors, r: rng) -> void =
    ring_iter(w,c,0,0,r, connect_lines_in)

random_connections_in : (w: array array point, c: colors, i: int, j: int,
```

```
   r: rng) -> void =
    if i < w.length - 1 then
    (if randf(r) > 0.65 then drawPath([w[i][j], w[i+1][(j+1)%w[0].length
   ]], c, 0) else void);
    if randf(r) > 0.65 then
      drawPath([w[i][j], w[i+1][(j-1+w[0].length)%w[0].length]], c, 0)
   else void
    else void

random_connections : (w: array array point, c: colors, r: rng) -> void =
   ring_iter(w,c,1,0,r, random_connections_in)

perturb_in : (w: array array point, c: colors, i: int, j: int, r: rng) ->
   void =
    s: float = 0.1 ;
    dx: float = randf(r) * s + 1.0 - s/2;
    dy: float = randf(r) * s + 1.0 - s/2;
    w[i][j].x = w[i][j].x * dx ;
    w[i][j].y = w[i][j].y * dy

perturb : (w: array array point, c: colors, r: rng) -> void =
   ring_iter(w,c,1,0,r, perturb_in)

my_canvas: canvas = {400, 400, 3}
r: rng = {35,62,21}
startCanvas(my_canvas)

w: array array point = build_points(7, 7, 0.95, r)
perturb(w, [], r)
connect_ring(w, [rgb(1,1,1), rgb(1,1,1)])
connect_lines(w, [rgb(1,1,1), rgb(1,1,1)], r)
random_connections(w, [rgb(1,1,1), rgb(1,1,1)], r)

endCanvas(my_canvas)
```

9.4 Test Programs

9.4.1 tests/fail-alias.err

```
Fatal error: exception Failure("Cannot create an alias with preexisting
   name int")
```

9.4.2 tests/fail-alias.sos

```
//aliasing type id is already defined. could have problems if aliasing
   float as int, etc.
alias int = int
```

9.4.3 tests/fail-alias2.err

```
Fatal error: exception Stdlib.Parsing.Parse_error
```

9.4.4 tests/fail-alias2.sos

```
//an alias of an undefined type id

alias x: notatype
```

9.4.5 tests/fail-array-construction2.err

```
Fatal error: exception Failure("Could not resolve type when defining a(
   Found array float, expected array int)")
```

9.4.6 tests/fail-array-construction2.sos

```
//if first element in array construction does not match array type,
   automatic type conversion does not take place
a: array int = [2.2, 1]
```

9.4.7 tests/fail-array-of.err

```
Fatal error: exception Failure("First operand of of operator must be an
   int")
```

9.4.8 tests/fail-array-of.sos

```
struct point = {x: float, y: float}
p1: point = {1.2, 2.3}

a: array float = p1 of [0.0]
```

9.4.9 tests/fail-array-void.err

```
Fatal error: exception Stdlib.Parsing.Parse_error
```

9.4.10 tests/fail-array-void.sos

```
//arrays cannot be of type void
array void a = []
```

9.4.11 tests/fail-bool-arith.err

```
Fatal error: exception Failure("Cannot add or subtract bool and bool")
```

9.4.12 tests/fail-bool-arith.sos

```
t: bool = true
f: bool = false

sum: bool = t + f
```

9.4.13 tests/fail-bool-comparison.err

```
Fatal error: exception Failure("Cannot equate bool and bool")
```

9.4.14 tests/fail-bool-comparison.sos

```
b: bool = true == true
```

9.4.15 tests/fail-bool.err

```
Fatal error: exception Failure("Cannot equate bool and bool")
```

9.4.16 tests/fail-bool.sos

```
b: bool = true == true
```

9.4.17 tests/fail-comparison-bool-float.err

```
Fatal error: exception Failure("Cannot compare bool and float")
```

9.4.18 tests/fail-comparison-bool-float.sos

```
b: bool = true > 0.1
```

9.4.19 tests/fail-definitions-in-definitions.err

```
Fatal error: exception Stdlib.Parsing.Parse_error
```

9.4.20 tests/fail-definitions-in-definitions.sos

```
//struct defined in function definition
func2: (x: int) -> int =
    struct s = {field: int};
    temp: s = {x};
    s.field
```

9.4.21 tests/fail-definitions-in-definitions2.err

```
Fatal error: exception Stdlib.Parsing.Parse_error
```

9.4.22 tests/fail-definitions-in-definitions2.sos

```
//defining function in function
bar: () -> int = 5
baruser: (f: func  -> int) -> int =
    sum: (x: int, y: int) -> int = x+y; temp: int = f(); sum(temp, 5)

barvar: func -> int = bar
print(baruser(barvar))
```

9.4.23 tests/fail-derived-comparison.err

```
Fatal error: exception Failure("Can only equate structs of matching type")
```

9.4.24 tests/fail-derived-comparison.sos

```
x: bool = {1,2} == {3,4,5}
```

9.4.25 tests/fail-float.err

```
Fatal error: exception Stdlib.Parsing.Parse_error
```

9.4.26 tests/fail-float.sos

```
f: float = .1
```

9.4.27 tests/fail-func-param-type.err

```
Fatal error: exception Failure("Could not resolve type when defining p(
   Found struct {int, int}, expected struct {float, float})")
```

9.4.28 tests/fail-func-param-type.sos

```
struct point = {x: float, y: float}
p: point = {1,2}

sum: (x: int, y: int) -> int = x+y

sum(3, p)
```

9.4.29 tests/fail-func-return-type.err

```
Fatal error: exception Failure("Incorrect return type for function f (
   Found int, expected struct {float, float})")
```

9.4.30 tests/fail-func-return-type.sos

```
struct point = {x: float, y: float}

f: (x: int) -> point = if x == 1 then 1 else 2
```

9.4.31 tests/fail-func.err

```
Fatal error: exception Failure("Unknown variable name sum")
```

9.4.32 tests/fail-func.sos

```
//undefined func

x: int = sum(1, 2)
```

9.4.33 tests/fail-if-elseless.err

```
Fatal error: exception Stdlib.Parsing.Parse_error
```

9.4.34 tests/fail-if-elseless.sos

```
x: int = if 1==2 then 1
```

9.4.35 tests/fail-if-expr-type.err

```
Fatal error: exception Failure("Could not reconcile types of then and else
    clauses (int, array int)")
```

9.4.36 tests/fail-if-expr-type.sos

```
x: int =
if 1 == 1
then 1
else [2]
```

9.4.37 tests/fail-memory.err

```
Fatal error: exception Failure("Can only free memory of struct and array
   types")
```

9.4.38 tests/fail-memory.sos

```
add: (x: int, y: int) -> int =
    x + 7

squareafteradd: (x: int, y: int) -> int =
    temp: int = add(x, y);
    ret: int = temp*temp;
    free(temp);
    ret
```

9.4.39 tests/fail-negate.err

```
Fatal error: exception Failure("Cannot negate non-arithmetic types")
```

9.4.40 tests/fail-negate.sos

```
x: int = -true
```

9.4.41 tests/fail-struct-arith.err

```
Fatal error: exception Failure("Can only add or subtract structs of
   matching type")
```

9.4.42 tests/fail-struct-arith.out

```
Can only add or subtract structs of matching type
```

9.4.43 tests/fail-struct-arith.sos

```
struct s1 = {a: int, b: float}

struct s2 = {a: int, b: int}

a: s1 = {1, 1.0}
b: s2 = {1, 1}

c: s3 = a+b
```

9.4.44 tests/fail-struct-arith2.err

```
Fatal error: exception Failure("Can only add or subtract structs of
   matching type")
```

9.4.45 tests/fail-struct-arith2.out

```
Can only operate on arithmetic structs
```

9.4.46 tests/fail-struct-arith2.sos

```
//arithmetic on structs that do not match types

struct s1 = {a: float, b: float}

struct s2 = {a: float, b: bool}

a: s1 = {1.4, 2.3}
b: s2 = {1.0, false}

c: s1 = a + b
```

9.4.47 tests/fail-struct-construction.err

```
Fatal error: exception Failure("Could not resolve type when defining v2(
   Found struct {float, int, int}, expected struct {float, float})")
```

9.4.48 tests/fail-struct-construction.sos

```
struct vector = {x: float, y: float}
v2: vector = {1.0, 2, 0}
```

9.4.49 tests/fail-struct-field.err

```
Fatal error: exception Failure("Could not find field y")
```

9.4.50 tests/fail-struct-field.sos

```
//accessing an undefined struct field

struct a = {x: int}

s: a = {1}

b: int = s.y
```

9.4.51 tests/fail-struct-field2.err

```
Fatal error: exception Failure("Cannot access fields for a non-struct
   variable")
```

9.4.52 tests/fail-struct-field2.out

```
Cannot access fields for a non-struct variable
```

9.4.53 tests/fail-struct-field2.sos

```
//attempting to access a field of a non-struct

struct s = {x: int}
b: int = 1
y: int = b.x
```

9.4.54 tests/fail-type-conversions.err

```
Fatal error: exception Failure("Could not resolve type when defining p2(
   Found struct {float, float}, expected struct {float, float, float})")
```

9.4.55 tests/fail-type-conversions.sos

```
struct point = {x: float, y: float}

struct point3d = {x: float, y: float, z: float}

p1: point = {1.0, 2.0}

p2: point3d = p1
```

9.4.56 tests/fail-unop.err

```
Fatal error: exception Failure("Cannot negate non-arithmetic types")
```

9.4.57 tests/fail-unop.sos

```
//fixed b to bool type. no type conversion

b: bool = -true
```

9.4.58 tests/fail-unop2.err

```
Fatal error: exception Failure("Cannot negate non-arithmetic types")
```

9.4.59 tests/fail-unop2.sos

```
x: int= -true
print(x)
```

9.4.60 tests/fail-value-access-modifier.err

```
Fatal error: exception Failure("Could not find field c")
```

9.4.61 tests/fail-value-access-modifier.sos

```
import color.sos
printf($(c : color = {0.0, 255.0, 255.0, 0.8}).c)
```

9.4.62 tests/fail-vardecl-as-expr-scoping.err

```
Fatal error: exception Failure("Unknown variable name x")
```

9.4.63 tests/fail-vardecl-as-expr-scoping.sos

```
y: int = 5
y= (x: int = 5)

print(y) //5
print(x) //5
```

9.4.64 tests/fail-vardef.err

```
Fatal error: exception Failure("Could not resolve type id notatype")
```

9.4.65 tests/fail-vardef.sos

```
//defining a variable with an undefined type id

x: notatype = 10
```

9.4.66 tests/fail-vardef2.err

```
Fatal error: exception Failure("Unknown variable name x")
```

9.4.67 tests/fail-vardef2.sos

```
//assignment before variable declaration

x = 10
```

9.4.68 tests/fail-vardef3.err

```
Fatal error: exception Stdlib.Parsing.Parse_error
```

9.4.69 tests/fail-vardef3.sos

```
//variable declaration without an initialization to a value
x: int
x = 5
```

9.4.70 tests/fail-void.err

```
Fatal error: exception Failure("Cannot use a void type in this context")
```

9.4.71 tests/fail-void.sos

```
x: void = 0
```

9.4.72 tests/test-E.out

```
200
```

9.4.73 tests/test-E.sos

```
x: float = 2.0E2
print(x)
```

9.4.74 tests/test-alias.out

```
2021
3
20
2022
2022
2024
2022
4042
2024
2041
56
1
1
1
1
2021
3
20
1
1
0
1.2
87
```

```
90
1
0
90
4
```

9.4.75 tests/test-alias.sos

```
/*alias of int */
alias year = int
alias month = int

//alias of alias
alias day = month

present_year: year = 2021
present_month: month = 3
present_day: day = 20
print(present_year) //2021
print(present_month) //3
print(present_day) //20


/*alias operations */

//alias operator primitive -> alias
//year + int -> year
test: year = present_year + 1
print(test) //2021+1=2022

//alias operator primitive -> alias2
//year + int -> month
test2: month = present_year + 1
print(test2) //2021+1=2022

//alias operator primitive -> aliasofalias
//year + int -> day
test3: day = present_year + 3
print(test3) //2021+1=2022

//alias operator primitive -> primitive
//year + int -> int
test4: int = present_year + 1
print(test4) //2021+1=2022

//alias operator alias -> primitive
//year + year -> int
test5: int = present_year + present_year
print(test5) //2021+2021=4042

//alias operator alias2 -> primitive
//year + month -> int
test6: int = present_year + present_month
```

```
print(test6) //2021+3=2024

//alias operator aliasofalias -> primitive
//year + month -> month
test7: int = present_year + present_day
print(test7) //2021+20=2041

//mixing int operators and aliases
test8: int = present_month*present_day-present_year/10%9
print(test8) //3*20-(2021/10)%9=56

/*alias comparison */
//primitive vs alias
i1: int = 1
y1: year = 1
result: bool = i1 == y1
print(result) //True

//alias vs alias
y2: year = 1
result2: bool = y1 == y2
print(result2) //True

//alias vs alias2
m1: month = 1
result3: bool = y1 == m1
print(result3) //True

//alias vs aliasofalias
d1: day = 1
result4: bool = y1 == d1
print(result4) //True

/*alias of struct*/
struct date = {day: int, yr: year, mth: month}
present_day: date = {20, present_year, present_month}
print(present_day.yr) //2021

//alias of struct field
present_day.yr = present_month
print(present_day.yr) //3

//alias of struct
alias new_date = date
newer: new_date = present_day
print(newer.day) //20

newer2: new_date = {1, 2, 3}
print(newer2.day) //1

result5: bool = newer == present_day
print(result5) //True (pass by reference)
result6: bool = newer == newer2
print(result6) //False
```

```
/*alias of float*/
alias x_dist = float
x: x_dist = 1.2
printf(x) //1.2

/*alias array*/
//alias of array int
alias scores = array int
class1: scores = [87,93,70]
print(class1[0]) //87
class1 = class1 + 3
print(class1[0]) //90

//alias of alias of array int
alias marks = scores
class2: marks = class1
class3: marks = [87, 93, 70]
result7: array array bool = class2 == class1
print(result7[0][0]) //True
result8: array array bool = class3 == class1
print(result8[0][0]) //False

//alias of array of alias of array int: array array int
alias class_scores = array scores
highschool1: class_scores = [class1, [1,2,3]]
print(highschool1[0][0]) //90

//alias of array array array int
alias school_scores = array array array int
district1: school_scores = [highschool1, [[4,5,6], [7,8,9]], [[10,11],
   [12,13,14,15]]]
print(district1[1][0][0]) //4
```

9.4.76 tests/test-array-access.out

```
1
2
3
1
2
2
4
6
```

9.4.77 tests/test-array-access.sos

```
mult_2 : (i: int, arr: array int) -> array int =
    if i == -1
    then arr
    //array access and assignment
    else
```

```
        arr[i] = arr[i]*2;
        mult_2(i - 1, arr)

printingarray : (arr: array int, length: int, current: int) -> void =
    if current == length
    then void
    else
        print(arr[current]);
        printingarray(arr, length, current+1)

a: array int = [1,2,3]
n: int = a[0]
b: array int = copy(a)
b = mult_2(3, b)

printingarray(a, 3, 0) //1, 2, 3
print(n) //1
print(a[0+1]) //2
printingarray(b, 3, 0) //2, 4, 6
```

9.4.78 tests/test-array-arith-operators.out

```
1
2
3
4
5
6
7
8
9
10
-1
0
1
2
3
2
4
6
8
10
0
1
1
2
2
```

9.4.79 tests/test-array-arith-operators.sos

```
printing: (x: int) -> void = print(x)
```

```
arr: array int = [1,2,3,4,5]
printing(arr)

arr2: array int = 5 + arr
printing(arr2)

arr3: array int = arr - 2
printing(arr3)

arr4: array int = 2 * arr
printing(arr4)

arr5: array int = arr/2
printing(arr5)
```

9.4.80 tests/test-array-concat.out

```
1
2
3
4
```

9.4.81 tests/test-array-concat.sos

```
printing: (x: int) -> void = print(x)

x: array int = [1, 2]
y: array int = [3, 4]

z: array int = x @ y

printing(z)
```

9.4.82 tests/test-array-construction.out

```
1
2.4
0
1
2
1
3
1
```

9.4.83 tests/test-array-construction.sos

```
/*int array*/
arrr: array int = [1]
print(arrr[0]) //1
```

```
/*float array*/
arrr2: array float = [1.2, 2.4]
printf(arrr2[1]) //2.4

/*bool array*/
arrr3: array bool = [false]
print(arrr3[0]) //False

/*struct array in test-array-of-struct.sos*/

/*nested arrays*/
nested: array array int = [[1], [2,3], [4,5,6]]
print(nested[0][0]) //1

/*func arrays*/
add1: (k: int) -> int = k+1
minus1: (k: int) -> int = k-1
f1: func int -> int = add1
f2: func int -> int = minus1

f_usage: (i: int, f: func int -> int) -> int =
    f(i)
arr_name: array func int -> int = [f1, f2]
print(f_usage(1, arr_name[0]))

/*construction with automatic type conversion*/
printing: (x: int) -> void = print(x)
a: array int = [1, 3.4, true]
printing(a)
```

9.4.84 tests/test-array-iteration.out

```
1
4
3
6
3
8
```

9.4.85 tests/test-array-iteration.sos

```
x: int = 1
y: int = 3
z: int = 8

many_x: array int = [x, 4, y, 6, 3, z]

printing: (x: int) -> void = print(x)

printing(many_x)
```

9.4.86 tests/test-array-of-struct.out

```
5
10
1
6
1
2
```

9.4.87 tests/test-array-of-struct.sos

```
//The structs within array of structs only need to have the same property
   types

struct x = {i: int}
struct y = {i: int}

distance1_x: x = {5}
distance2_x: x = {6}
distance1_y: y = {1}

b: array x = [distance1_x, {10}, distance1_y]

print(b[0].i) //5
print(b[1].i) //10
print(b[2].i) //1

b[0] = distance2_x
print(b[0].i) //6

b[0] = distance1_y
print(b[0].i) //1

b[0].i = 2
print(b[0].i) //2
```

9.4.88 tests/test-assign.out

```
5
6
1
2.5
1
0
1
2
4
4.4
1
3.3
5
```

9.4.89 tests/test-assign.sos

```
//int
a: int = 5
print(a) //5
a = 6
print(a) //6


//float
b: float = 1.0
printf(b) //1.0
b = 2.5
printf(b) //2.5


//bool
c: bool = true
print(c) //True
c = false
print(c) //False


//array
d: array int = [1, 2]
print(d[0]) //1
temp: int = d[1]
d[1] = d[0]
d[0] = temp
print(d[0]) //2


//struct
struct point = {x: float, y: float}
struct e = {v: int, w: float, x: bool, y: point, z: array int }

es: e = {4, 4.4, true, {3.3, 4.5}, [5, 6, 7]}
print(es.v) //4
printf(es.w) //4.4
print(es.x) //True
printf(es.y.x) //3.3
print(es.z[0]) //5
```

9.4.90 tests/test-associativity.out

```
1
2
2
1
0
1
1
1
```

```
3
2
2
50
```

9.4.91 tests/test-associativity.sos

```
/*
right associative: =, !, **
left associative: ., *, /, %, +, -, @, of, (comparison), (boolean), ",", ;
*/

/*right*/
//=
a: int = 1
print(a) //1
b: int = a = 2 //b = (a=2) = 2
print(a) //2
print(b) //2

//!
b: bool = true
print(b) //1
print(!b) //0
print(!!b) //1

//left
print(5 - 3 - 1) //2-1 = 1
print((5 - 3) - 1) //2-1 = 1
print(5 - (3 - 1)) //5-2=3

print(100/10/5) // 10/5 = 2
print((100/10)/5) //10/5 = 2
print(100/(10/5)) //100/2 = 50
```

9.4.92 tests/test-conditional.out

```
1
0
```

9.4.93 tests/test-conditional.sos

```
x: int = 1

//if then
y: int = if x == 1 then 1 else 0
print(y) //1

//else
z: int = if y != 1 then 1 else 0
print(z) //0
```

9.4.94 tests/test-derived-comparison.out

```
1
0
1
1
0
```

9.4.95 tests/test-derived-comparison.sos

```
/* structs, not arrays, allows for comparions */

//comparison of unnamed structs
x: bool = {1,2} == {1,2}
print(x) //true
x = {1, 3} == {2, 4}
print(x)

//comparison of named structs
struct point = {x: float, y: float}
p1: point = {1.0, 2.0}
p2: point = {1.0, 2.0}
p3: point = {0.0, 0.0}
x = p1 == p2
print(x) //true
x = p1 == p2
print(x) //true
x = p1 == p3
print(x) //false
```

9.4.96 tests/test-dot-product.out

```
11
```

9.4.97 tests/test-dot-product.sos

```
struct point = {x: int, y: int}

p1: point = {1, 2}
p2: point = {3, 4}

dotted: int = p1*p2
print(dotted) //1*3 + 2*4 = 11
```

9.4.98 tests/test-fibb.out

```
5
```

9.4.99 tests/test-fibb.sos

```
fib: (n: int) -> int =
if n <= 1
then n
else fib(n-1) + fib(n-2)

x: int = 5
x = fib(5)
print(x) //5
```

9.4.100 tests/test-func-of-struct.out

```
5
```

9.4.101 tests/test-func-of-struct.sos

```
//predefining struct in outer scope
struct normal = {field: int}

func1: (x: int) -> normal =
    temp: normal  = {x}; temp

a: normal = func1(5)
print(a.field) //5
```

9.4.102 tests/test-func1.out

```
5
```

9.4.103 tests/test-func1.sos

```
//basic func, no parameters

foo: () -> int= 5

print(foo()) //5
```

9.4.104 tests/test-func2.out

```
5
```

9.4.105 tests/test-func2.sos

```
//function with parameter

foo: (x: int) -> int = 5
```

```
a: int = foo(1)
print(a) //5
```

9.4.106 tests/test-func3.out

```
5
```

9.4.107 tests/test-func3.sos

```
//function with body dependent on parameter

foo: (x: int) -> int = x

a: int = foo(5)
print(a) //5
```

9.4.108 tests/test-func4.out

```
1
1
```

9.4.109 tests/test-func4.sos

```
//nested function application

first: () -> int = 1
second: (x: int) -> int = x

a: int = second(first())
print(a) //1

b: int = second(second(second(first())))
print(b) //1
```

9.4.110 tests/test-func5.out

```
5
```

9.4.111 tests/test-func5.sos

```
//multiple parameters

add: (x: int, y: int) -> int = x+y

var: int = add(2,3)
print(var) //8
```

9.4.112 tests/test-func6.out

```
0
0
```

9.4.113 tests/test-func6.sos

```
//more complex function bodies: conditionals

foo: (x: int) -> int =
    if x == 0
    then x
    else (y: int = x); y-x

a: int = foo(0)
b: int = foo(5)
print(a) //0
print(b) //0
```

9.4.114 tests/test-func7.out

```
5
```

9.4.115 tests/test-func7.sos

```
//using func keyword to turn a function into a variable

bar: () -> int = 5
baruser: (f: func  -> int) -> int = f()

barvar: func -> int = bar
print(baruser(barvar))
```

9.4.116 tests/test-helloworld.out

```
5
startRendering...
endRendering...
startRendering...
endRendering...
```

9.4.117 tests/test-helloworld.sos

```
import renderer.sos

a: int = 5
print(a)
```

```
p1: point = {-0.9, -0.9}
p2: point = {-0.9, -0.7}
p3: point = {-0.7, -0.7}
p4: point = {-0.7, -0.9}
point_arr : path = [p1, p2, p3, p4]

c1 : color = {255.0, 0.0, 0.0, 0.8}
c2 : color = {0.0, 255.0, 0.0, 0.8}
c3 : color = {0.0, 0.0, 255.0, 0.8}
c4 : color = {100.0, 100.0, 0.0, 0.8}
color_arr : colors = [c1, c2, c3, c4]

canvas1 : canvas = {400, 400, 0}

startCanvas(canvas1)
drawShape(point_arr, color_arr, 0, 1)
endCanvas(canvas1)

canvas2 : canvas = {400, 400, 1}

startCanvas(canvas2)
drawShape(point_arr, color_arr, 0, 1)
endCanvas(canvas2)
```

9.4.118 tests/test-if.out

```
1
0
```

9.4.119 tests/test-if.sos

```
x: int = 1

//catch if-then
y: int =
if (x == 1)
then 1
else 0

print(y) //1

//catch else
z: int =
if (y != 1)
then 1
else 0

print(z) //0
```

9.4.120 tests/test-import-protection.out

```
0
```

9.4.121 tests/test-import-protection.sos

```
import point.sos
import point.sos
p1: point = {0.0, 0.0}
printf(p1.x)

/* can duplicate a file import.
Usage example: dragon.sos imports renderer.sos and transform.sos, and both
    import shape.sos
*/
```

9.4.122 tests/test-import.out

```
0
```

9.4.123 tests/test-import.sos

```
import point.sos

p1: point = {0.0, 0.0}

printf(p1.x)
```

9.4.124 tests/test-logical-operators.out

```
1
1
0
0
1
1
0
0
1
```

9.4.125 tests/test-logical-operators.sos

```
/* logical operators: AND, OR, NOT  */

t: bool = true
f: bool = false


print(t && t)
```

```
//and
and_ans: bool = t && t
print(and_ans)              //T AND T = T
and_ans = t && f
print(and_ans)              //T AND F = F
and_ans = f && f
print(and_ans)              //F AND F = F

//or
or_ans: bool = t || f
print(or_ans)               //T OR T = T
or_ans = t || f
print(or_ans)               //T OR F = T
or_ans = f || f
print(or_ans)               //F OR F = F

//not
not_ans: bool = !t
print(not_ans)              //NOT TRUE = F
not_ans = !f
print(not_ans)              //NOT FALSE = T
```

9.4.126 tests/test-memory.out

```
5
```

9.4.127 tests/test-memory.sos

```
struct point = {x: float, y: float}

add: (x: point, y: point) -> point =
    x+y
dotafteradd: (x: point, y: point) -> float =
    temp: point = add(x, y);
    ret: float = temp*temp;
    free(temp);
    ret

print(dotafteradd({0.0, 1.0}, {1.0, 1.0})) //{1, 2}*{1, 2} = 1+4 = 5
```

9.4.128 tests/test-precedence.out

```
16
20
```

9.4.129 tests/test-precedence.sos

```
print(5*3+1) //16
print(5*(3+1)) //20
```

9.4.130 tests/test-primitive-arith-operators.out

```
3
-1
2
0
2
3.2
1
2.31
1.90909
```

9.4.131 tests/test-primitive-arith-operators.sos

```
/* arithmetic operators on primitives int, float */

add: (x: int, y: int) -> int = x + y
sub: (x: int, y: int) -> int = x - y
mult: (x: int, y: int) -> int = x * y
div: (x: int, y: int) -> int = x/y
mod: (x: int, y: int) -> int = x%y

addf: (x: float, y: float) -> float = x + y
subf: (x: float, y: float) -> float = x - y
multf:(x: float, y: float) -> float = x * y
divf: (x: float, y: float) -> float = x/y

x: int = add(1,2)
print(x) //3
x = sub(1,2)
print(x) //-1
x = mult(1,2)
print(x) //2
x = div(1,2)
print(x) //0
x = mod(10,8)
print(x) //2

y: float = addf(1.1, 2.1)
printf(y) //3.2
y = subf(2.1, 1.1)
printf(y) //1.0
y = multf(1.1, 2.1)
printf(y) //2.31
y = divf(2.1, 1.1)
printf(y) //1.9090...
```

9.4.132 tests/test-primitive-comparison.out

```
1
1
0
1
1
1
1
1
1
1
0
1
1
1
1
1
```

9.4.133 tests/test-primitive-comparison.sos

```
/*
==, !=, <, <=, >, >=
primitives are passed by value
*/

/*int*/
a: int = 5
b: int = 5
c: int = 7

out1: bool  = a == a
print(out1) //True

out2: bool  = a == b
print(out2) //True

out3: bool = a == c
print(out3) //False


x: bool = 5 != 6
print(x)

x = 5 <= 6
print(x)

x = 6 >= 6
print(x)

x = 6 > 5
print(x)

x = 5 < 6
```

```
print(x)



/*float*/
d: float = 1.2
e: float = 1.2
f: float = 2.3

out4: bool = d == d
print(out4) //True

out5: bool = d == e
print(out5) //True

out6: bool = e == f
print(out6) //False


x = 5.0 != 6.0
print(x)

x = 5.0 <= 6.0
print(x)

x = 6.0 >= 6.0
print(x)

x = 6.0 > 5.0
print(x)

x = 5.0 < 6.0
print(x)
```

9.4.134 tests/test-print.out

```
1
1
1.1
1
```

9.4.135 tests/test-print.sos

```
/* Int: print */
x: int = 1
print(1) //1
print(x) //1

/* Float: printf */
y: float = 1.0
printf(1.1) //1.1
printf(y) //1
```

9.4.136 tests/test-recursion.out

```
15
```

9.4.137 tests/test-recursion.sos

```
add: (n: int) -> int =
if n == 0
then 0
else add(n-1) + n

x: int = 5
x = add(x)
print(x) //5+4+3+2+1+0=15
```

9.4.138 tests/test-scoping-if.out

```
1
1
3
0
4
```

9.4.139 tests/test-scoping-if.sos

```
y: int = 1
x: int = 1

print(x)
print(y)

x =
if 1==1
then x = 3; y = 0; print(x); print(y); 4
else 0

print(x) //4
```

9.4.140 tests/test-scoping.out

```
6
12
6
```

9.4.141 tests/test-scoping.sos

```
//global x vs x in function
```

```
double: (x: int) -> int = x+x

x: int = double(3)
print(x) //6

x = double(x)
print(x) //12

func2: () -> int = x: int = 3; x+x

x = func2()
print(x) //6
```

9.4.142 tests/test-struct-access.out

```
2
3
```

9.4.143 tests/test-struct-access.sos

```
//construction
struct person = {identifier: int, age: int, isMarried: bool}
tom: person = {2, 20, false}

//accessing struct field
tom_id: int = tom.identifier
print(tom_id) //2

//changing struct field
tom.identifier=3
tom_id = tom.identifier
print(tom_id) //3
```

9.4.144 tests/test-struct-arith-diff-names.out

```
4
6
```

9.4.145 tests/test-struct-arith-diff-names.sos

```
struct s1 = {a: int, b: int}

struct s2 = {c: int, d: int}

first: s1 = {1, 2}

second: s2 = {3, 4}

first = first + second
```

```
print(first.a)
print(first.b)
```

9.4.146 tests/test-struct-construction.out

```
2
20
```

9.4.147 tests/test-struct-construction.sos

```
//construction
struct person = {identifier: int, age: int}

p1: person = {2, 20}
print(p1.identifier) //2
print(p1.age) //20
```

9.4.148 tests/test-struct-dot-assoc.out

```
2
```

9.4.149 tests/test-struct-dot-assoc.sos

```
struct more = {field1: int}

struct stuff = {field1: int, field2: more}

temp1: more = {2}
temp2: stuff = {1, temp1}

ans: int = temp2.field2.field1
print(ans) //2
```

9.4.150 tests/test-struct-dot-product.out

```
11
```

9.4.151 tests/test-struct-dot-product.sos

```
struct s1 = {a: int, b: int}

first: s1 = {1, 2}

second: s1 = {3, 4}

dotted: int = first*second
print(dotted)
```

9.4.152 tests/test-struct-of-array.out

```
3
```

9.4.153 tests/test-struct-of-array.sos

```
struct first = {arr1: array int}

struct second = {arr2: array first}

struct third = {arr3: array second}

s1: first = {[1,2]}
s1_2: first = {[3,4]}
s2: second = {[s1, s1_2]}
s2_2: second = {[{[-1, 0]}, {[-2, -1]}]}
s1: third = {[s2, s2_2]}

x: int = s1.arr3[0].arr2[1].arr1[0]
print(x) //3
```

9.4.154 tests/test-struct-of-same-types.out

```
-2.5
-4
1
2
```

9.4.155 tests/test-struct-of-same-types.sos

```
struct point = {x: float, y: float}
struct vector = {a: float, b: float}


p: point = {-2.5, -4.0}
v: vector = p
printf(v.a) //-2.5
printf(v.b) //-4.0

v2: vector = {1.0, 2.0}
p2: point = v2
printf(p2.x) //1.0
printf(p2.y) //2.0
```

9.4.156 tests/test-struct-of-struct.out

```
2
```

9.4.157 tests/test-struct-of-struct.sos

```
//predefining field struct in outer scope
struct a = {field: int}
struct b = {
    field: int,
    field2: a}

test1: b = {1, temp: a = {2}; temp}
print(test1.field2.field)

/*
//field struct defined in outer struct
struct c = {
    int field,
    struct d = {int field} field2
}

c test2 = {1, d temp2 = {2}; temp2}
int temp2 = test2.field2.field
*/
```

9.4.158 tests/test-struct-scaling.out

```
6
-9
1
-1
```

9.4.159 tests/test-struct-scaling.sos

```
struct point = {a: int, b: int}

p1: point = {2, -3}

p_mult: point = 3*p1

print(p_mult.a) //6
print(p_mult.b) //-9

p_div: point = p1/2

print(p_div.a) //1
print(p_div.b) //-1
```

9.4.160 tests/test-type-conversions-comp.out

```
1
1
1
```

9.4.161 tests/test-type-conversions-comp.sos

```
/*float over int*/
b: bool = 3.5 > 3
print(b) //True

b = 3 < 3.5
print(b) //True

/*int over bool*/
b = 2 > true
print(b) //True
```

9.4.162 tests/test-type-conversions-func.out

```
4
```

9.4.163 tests/test-type-conversions-func.sos

```
add_int: (x: int, y: int) -> int = x + y

x: float = 1.0
y: float = 3.0

sum: int = add_int(x, y)
print(sum)
```

9.4.164 tests/test-type-conversions.out

```
10
1
1
0
0
0
0
2
-2
1
1
1
0
4
2
3
```

9.4.165 tests/test-type-conversions.sos

```
/*bool to int*/
```

```
//Type conversion during variable assignment
x: bool = true
y: int = x
y = x + 9
print(y) //10

//Type conversion during function application
sum: (x: int, y: int) -> int = x + y
summed: int = sum(true, false)
print(summed) //1


/*int to bool*/
//Type conversion during variable assignment
m: int = 1
n: int = 0
o: bool = m
p: bool = n
print(o) //True
print(p) //False

//Type conversion based on unary operator
p = !1
print(p) //False

//Type conversion during function application
not: (x: bool) -> bool = !x
notted: bool  = not(1)
print(notted) //False


/*float to int (via truncation) */
a: float = -0.5
b: int = a
print(b) //0

a = 2.9
b = a
print(b) //2

a = -2.9
b = a
print(b) //-2


/*float to bool: anything other than 0.0 is true */
b: bool = 1.1 //1
print(b)
b = -1.1 //1
print(b)
b = 0.1
print(b) //1
b = 0.0
print(b) // 0
```

```
//Type conversion during function application (use sum() on line 15)
summed2: int = sum(1.2, 3.5)
print(summed2) //4


/*int to float (via injection) */
c: int = 2
d: float = c
printf(d) //2.0

//Type conversion during function application
floatsum: (x: float, y: float) -> float = x + y
e: float = floatsum(1, 2)
printf(e) //3.0
```

9.4.166 tests/test-unop.out

```
5
-5
-3
5
-5
2
1
0
0
```

9.4.167 tests/test-unop.sos

```
/*negation of int and float: neg*/
//int
a: int = 5
b: int = -a
print(a) //5
print(b) //-5
print(-1+-2) //-3

//float
c: float = 5.0
d: float = -c
printf(c) //5.0
printf(d) //-5.0
printf(-1.0*-2.0) //2.0


/*negation of bool: not*/
e: bool = true
f: bool = !e
print(e) //True
print(f) //False
print(!true) //False
```

9.4.168 tests/test-value-access-modifier.out

```
255
```

9.4.169 tests/test-value-access-modifier.sos

```
import color.sos
printf($(c : color = {0.0, 255.0, 255.0, 0.8}).g)
```

9.4.170 tests/test-var-assign-as-expr.out

```
6
6
```

9.4.171 tests/test-var-assign-as-expr.sos

```
a: int = 2

print(a = 3 + 3) //6
print(a) //6
```

9.4.172 tests/test-vardecl-as-expr.out

```
10
```

9.4.173 tests/test-vardecl-as-expr.sos

```
//variable declaration for x is an expression for assignment of y
y: int = 5
y= (x: int = 10)

print(y) //10
```

9.4.174 tests/test-vardecl-twice.out

```
5
1
```

9.4.175 tests/test-vardecl-twice.sos

```
/* SOS allows variables to be redeclared to different types */

a: int = 5
print(a) //5

a: bool = true
print(a) //1
```

9.4.176 tests/test-vardecl.out

```
0
0
1
```

9.4.177 tests/test-vardecl.sos

```
/*variable definitions must have a type id and an expression.
This test case is for primitives only. Reference type definitions
are tested in "construction" test cases.*/

x: int = 0
print(x)

y: float = 0.0
printf(y)

z: bool = true
print(z)
```

9.4.178 tests/test-vardecl2.out

```
0
0
```

9.4.179 tests/test-vardecl2.sos

```
/*variables must start with a letters, then can have any
number of letters, underscores, and numbers*/

a_d1: int = 0
a000___: float = 0.0
print(a_d1)
print(a000___)
```

9.5 Example LL Outputs

9.5.1 helloworld.ll

test-helloworld.ll

```
; ModuleID = 'SOS'
source_filename = "SOS"

declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
  %p1 = alloca { float, float }*
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64
    ptrtoint (float* getelementptr (float, float* null, i32 1) to
```

```
  i64), i64 2) to i32))
%anon = bitcast i8* %malloccall to { float, float }*
%fieldaddr = getelementptr { float, float }, { float, float }*
  %anon, i32 0, i32 0
store float -5.000000e-01, float* %fieldaddr
%fieldaddr1 = getelementptr { float, float }, { float, float }*
  %anon, i32 0, i32 1
store float -5.000000e-01, float* %fieldaddr1
store { float, float }* %anon, { float, float }** %p1
%p2 = alloca { float, float }*
%malloccall2 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64
  ptrtoint (float* getelementptr (float, float* null, i32 1) to
  i64), i64 2) to i32))
%anon3 = bitcast i8* %malloccall2 to { float, float }*
%fieldaddr4 = getelementptr { float, float }, { float, float }*
  %anon3, i32 0, i32 0
store float -5.000000e-01, float* %fieldaddr4
%fieldaddr5 = getelementptr { float, float }, { float, float }*
  %anon3, i32 0, i32 1
store float 5.000000e-01, float* %fieldaddr5
store { float, float }* %anon3, { float, float }** %p2
%p3 = alloca { float, float }*
%malloccall6 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64
  ptrtoint (float* getelementptr (float, float* null, i32 1) to
  i64), i64 2) to i32))
%anon7 = bitcast i8* %malloccall6 to { float, float }*
%fieldaddr8 = getelementptr { float, float }, { float, float }*
  %anon7, i32 0, i32 0
store float 5.000000e-01, float* %fieldaddr8
%fieldaddr9 = getelementptr { float, float }, { float, float }*
  %anon7, i32 0, i32 1
store float 5.000000e-01, float* %fieldaddr9
store { float, float }* %anon7, { float, float }** %p3
%p4 = alloca { float, float }*
%malloccall10 = tail call i8* @malloc(i32 trunc (i64 mul nuw
  (i64 ptrtoint (float* getelementptr (float, float* null,
  i32 1) to i64), i64 2) to i32))
%anon11 = bitcast i8* %malloccall10 to { float, float }*
%fieldaddr12 = getelementptr { float, float }, { float, float }*
  %anon11, i32 0, i32 0
store float 5.000000e-01, float* %fieldaddr12
%fieldaddr13 = getelementptr { float, float }, { float, float }*
  %anon11, i32 0, i32 1
store float -5.000000e-01, float* %fieldaddr13
store { float, float }* %anon11, { float, float }** %p4
%point_arr = alloca { { float, float }**, i32 }*
%malloccall14 = tail call i8* @malloc(i32 mul (i32 ptrtoint (i1**
  getelementptr (i1*, i1** null, i32 1) to i32), i32 4))
%arrdata = bitcast i8* %malloccall14 to { float, float }**
%p115 = load { float, float }*, { float, float }** %p1
%storeref = getelementptr { float, float }*, { float, float }**
  %arrdata, i32 0
store { float, float }* %p115, { float, float }** %storeref
%p216 = load { float, float }*, { float, float }** %p2
%storeref17 = getelementptr { float, float }*, { float, float }**
  %arrdata, i32 1
store { float, float }* %p216, { float, float }** %storeref17
%p318 = load { float, float }*, { float, float }** %p3
%storeref19 = getelementptr { float, float }*, { float, float }**
```

```
  %arrdata, i32 2
store { float, float }* %p318, { float, float }** %storeref19
%p420 = load { float, float }*, { float, float }** %p4
%storeref21 = getelementptr { float, float }*, { float, float }**
  %arrdata, i32 3
store { float, float }* %p420, { float, float }** %storeref21
%malloccall22 = tail call i8* @malloc(i32 ptrtoint ({ { float, float
  }**, i32 }* getelementptr ({ { float, float }**, i32 }, { { float,
  float }**, i32 }* null, i32 1) to i32))
%arr = bitcast i8* %malloccall22 to { { float, float }**, i32 }*
%arrdata23 = getelementptr { { float, float }**, i32 }, { { float,
  float }**, i32 }* %arr, i32 0, i32 0
%arrlen = getelementptr { { float, float }**, i32 }, { { float, float }**,
  i32 }* %arr, i32 0, i32 1
store { float, float }** %arrdata, { float, float }*** %arrdata23
store i32 4, i32* %arrlen
store { { float, float }**, i32 }* %arr, { { float, float }**, i32 }**
  %point_arr
%c1 = alloca { float, float, float, float }*
%malloccall24 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
  (float* getelementptr (float, float* null, i32 1) to i64), i64 4) to i32))
%anon25 = bitcast i8* %malloccall24 to { float, float, float, float }*
%fieldaddr26 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon25, i32 0, i32 0
store float 2.550000e+02, float* %fieldaddr26
%fieldaddr27 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon25, i32 0, i32 1
store float 0.000000e+00, float* %fieldaddr27
%fieldaddr28 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon25, i32 0, i32 2
store float 0.000000e+00, float* %fieldaddr28
%fieldaddr29 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon25, i32 0, i32 3
store float 0x3FE99999A0000000, float* %fieldaddr29
store { float, float, float, float }* %anon25, { float, float, float, float
  }** %c1
%c2 = alloca { float, float, float, float }*
%malloccall30 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
  (float* getelementptr (float, float* null, i32 1) to i64), i64 4) to i32))
%anon31 = bitcast i8* %malloccall30 to { float, float, float, float }*
%fieldaddr32 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon31, i32 0, i32 0
store float 0.000000e+00, float* %fieldaddr32
%fieldaddr33 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon31, i32 0, i32 1
store float 2.550000e+02, float* %fieldaddr33
%fieldaddr34 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon31, i32 0, i32 2
store float 0.000000e+00, float* %fieldaddr34
%fieldaddr35 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon31, i32 0, i32 3
store float 0x3FE99999A0000000, float* %fieldaddr35
store { float, float, float, float }* %anon31, { float, float, float, float
  }** %c2
%c3 = alloca { float, float, float, float }*
%malloccall36 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
  (float* getelementptr (float, float* null, i32 1) to i64), i64 4) to i32))
%anon37 = bitcast i8* %malloccall36 to { float, float, float, float }*
%fieldaddr38 = getelementptr { float, float, float, float }, { float, float,
```

```
  float, float }* %anon37, i32 0, i32 0
store float 0.000000e+00, float* %fieldaddr38
%fieldaddr39 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon37, i32 0, i32 1
store float 0.000000e+00, float* %fieldaddr39
%fieldaddr40 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon37, i32 0, i32 2
store float 2.550000e+02, float* %fieldaddr40
%fieldaddr41 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon37, i32 0, i32 3
store float 0x3FE99999A0000000, float* %fieldaddr41
store { float, float, float, float }* %anon37, { float, float, float, float
  }** %c3
%c4 = alloca { float, float, float, float }*
%malloccall42 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
  (float* getelementptr (float, float* null, i32 1) to i64), i64 4) to i32))
%anon43 = bitcast i8* %malloccall42 to { float, float, float, float }*
%fieldaddr44 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon43, i32 0, i32 0
store float 1.000000e+02, float* %fieldaddr44
%fieldaddr45 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon43, i32 0, i32 1
store float 1.000000e+02, float* %fieldaddr45
%fieldaddr46 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon43, i32 0, i32 2
store float 0.000000e+00, float* %fieldaddr46
%fieldaddr47 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %anon43, i32 0, i32 3
store float 0x3FE99999A0000000, float* %fieldaddr47
store { float, float, float, float }* %anon43, { float, float, float, float
  }** %c4
%color_arr = alloca { { float, float, float, float }**, i32 }*
%malloccall48 = tail call i8* @malloc(i32 mul (i32 ptrtoint (i1**
  getelementptr (i1*, i1** null, i32 1) to i32), i32 4))
%arrdata49 = bitcast i8* %malloccall48 to { float, float, float, float }**
%c150 = load { float, float, float, float }*, { float, float, float, float
  }** %c1
%storeref51 = getelementptr { float, float, float, float }*, { float, float,
  float, float }** %arrdata49, i32 0
store { float, float, float, float }* %c150, { float, float, float, float
  }** %storeref51
%c252 = load { float, float, float, float }*, { float, float, float,
  float }** %c2
%storeref53 = getelementptr { float, float, float, float }*, { float,
  float,
  float, float }** %arrdata49, i32 1
store { float, float, float, float }* %c252, { float, float, float,
  float }** %storeref53
%c354 = load { float, float, float, float }*, { float, float, float,
  float }** %c3
%storeref55 = getelementptr { float, float, float, float }*, { float,
float, float, float }** %arrdata49, i32 2
store { float, float, float, float }* %c354, { float, float, float,
  float }** %storeref55
%c456 = load { float, float, float, float }*, { float, float, float,
  float }** %c4
%storeref57 = getelementptr { float, float, float, float }*, { float,
  float,
  float, float }** %arrdata49, i32 3
```

```
store { float, float, float, float }* %c456, { float, float, float,
  float }** %storeref57
%malloccall58 = tail call i8* @malloc(i32 ptrtoint ({ { float, float,
  float, float }**, i32 }* getelementptr ({ { float, float, float,
  float }**, i32 }, { { float, float, float, float }**, i32 }* null,
  i32 1) to i32))
%arr59 = bitcast i8* %malloccall58 to { { float, float, float,
  float }**, i32 }*
%arrdata60 = getelementptr { { float, float, float, float }**,
  i32 }, { { float, float, float, float }**, i32 }* %arr59,
  i32 0, i32 0
%arrlen61 = getelementptr { { float, float, float, float }**, i32 }, {
  { float, float, float, float }**, i32 }* %arr59, i32 0, i32 1
store { float, float, float, float }** %arrdata49, { float,
  float, float, float }*** %arrdata60
store i32 4, i32* %arrlen61
store { { float, float, float, float }**, i32 }* %arr59, { { float,
  float, float, float }**, i32 }** %color_arr
%canvas1 = alloca { i32, i32, i32 }*
%malloccall62 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
  (i32* getelementptr (i32, i32* null, i32 1) to i64), i64 3) to i32))
%anon63 = bitcast i8* %malloccall62 to { i32, i32, i32 }*
%fieldaddr64 = getelementptr { i32, i32, i32 }, { i32, i32, i32 }*
  %anon63, i32 0, i32 0
store i32 400, i32* %fieldaddr64
%fieldaddr65 = getelementptr { i32, i32, i32 }, { i32, i32, i32 }*
  %anon63, i32 0, i32 1
store i32 400, i32* %fieldaddr65
%fieldaddr66 = getelementptr { i32, i32, i32 }, { i32, i32, i32 }*
  %anon63, i32 0, i32 2
store i32 0, i32* %fieldaddr66
store { i32, i32, i32 }* %anon63, { i32, i32, i32 }** %canvas1
%canvas167 = load { i32, i32, i32 }*, { i32, i32, i32 }** %canvas1
call void @startCanvas({ i32, i32, i32 }* %canvas167)
%point_arr68 = load { { float, float }**, i32 }*, { { float, float
  }**, i32 }** %point_arr
%color_arr69 = load { { float, float, float, float }**, i32 }*,
  { { float, float, float, float }**, i32 }** %color_arr
call void @drawShape({ { float, float }**, i32 }* %point_arr68,
  { { float, float, float, float }**, i32 }* %color_arr69,
  i32 0, i32 1)
%canvas170 = load { i32, i32, i32 }*, { i32, i32, i32 }** %canvas1
call void @endCanvas({ i32, i32, i32 }* %canvas170)
%canvas2 = alloca { i32, i32, i32 }*
%malloccall71 = tail call i8* @malloc(i32 trunc (i64 mul nuw
  (i64 ptrtoint (i32* getelementptr (i32, i32* null, i32 1)
  to i64), i64 3) to i32))
%anon72 = bitcast i8* %malloccall71 to { i32, i32, i32 }*
%fieldaddr73 = getelementptr { i32, i32, i32 }, { i32, i32,
  i32 }* %anon72, i32 0, i32 0
store i32 400, i32* %fieldaddr73
%fieldaddr74 = getelementptr { i32, i32, i32 }, { i32, i32,
  i32 }* %anon72, i32 0, i32 1
store i32 400, i32* %fieldaddr74
%fieldaddr75 = getelementptr { i32, i32, i32 }, { i32, i32,
  i32 }* %anon72, i32 0, i32 2
store i32 1, i32* %fieldaddr75
store { i32, i32, i32 }* %anon72, { i32, i32, i32 }** %canvas2
%canvas276 = load { i32, i32, i32 }*, { i32, i32, i32 }** %canvas2
```

```
  call void @startCanvas({ i32, i32, i32 }* %canvas276)
  %point_arr77 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %point_arr
  %color_arr78 = load { { float, float, float, float }**, i32 }*,
    { { float, float, float, float }**, i32 }** %color_arr
  call void @drawShape({ { float, float }**, i32 }* %point_arr77,
    { { float, float, float, float }**, i32 }* %color_arr78,
    i32 0, i32 1)
  %canvas279 = load { i32, i32, i32 }*, { i32, i32, i32 }** %canvas2
  call void @endCanvas({ i32, i32, i32 }* %canvas279)
  ret i32 0
}

declare float @sqrtf(float)

declare float @sinf(float)

declare float @cosf(float)

declare float @tanf(float)

declare float @asinf(float)

declare float @acosf(float)

declare float @atanf(float)

declare float @toradiansf(float)

declare void @gl_startRendering(i32, i32)

declare void @gl_endRendering(i32, i32, i32)

declare void @gl_drawCurve({ float*, i32 }*, { float*, i32 }*, i32)

declare void @gl_drawShape({ float*, i32 }*, { float*, i32 }*, i32, i32)

declare void @gl_drawPoint({ float*, i32 }*, { float*, i32 }*, i32)

define float @floor(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %z = alloca float
  %y = alloca i32
  %x2 = load float, float* %x1
  %cast = fptosi float %x2 to i32
  store i32 %cast, i32* %y
  %cast3 = sitofp i32 %cast to float
  store float %cast3, float* %z
  %z4 = load float, float* %z
  %x5 = load float, float* %x1
  %tmp = fcmp ole float %z4, %x5
  %if_tmp = alloca float
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %if_tmp9 = load float, float* %if_tmp
  ret float %if_tmp9
```

```
then:                                             ; preds = %entry
  %z6 = load float, float* %z
  store float %z6, float* %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %z7 = load float, float* %z
  %tmp8 = fsub float %z7, 1.000000e+00
  store float %tmp8, float* %if_tmp
  br label %merge
}

define float @ceil(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %tmp = fneg float %x2
  %fxn_result = call float @floor(float %tmp)
  %tmp3 = fneg float %fxn_result
  ret float %tmp3
}

define float @frac(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %x3 = load float, float* %x1
  %fxn_result = call float @floor(float %x3)
  %tmp = fsub float %x2, %fxn_result
  ret float %tmp
}

define float @max(float %a, float %b) {
entry:
  %a1 = alloca float
  store float %a, float* %a1
  %b2 = alloca float
  store float %b, float* %b2
  %a3 = load float, float* %a1
  %b4 = load float, float* %b2
  %tmp = fcmp olt float %a3, %b4
  %if_tmp = alloca float
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %if_tmp7 = load float, float* %if_tmp
  ret float %if_tmp7

then:                                             ; preds = %entry
  %b5 = load float, float* %b2
  store float %b5, float* %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %a6 = load float, float* %a1
  store float %a6, float* %if_tmp
```

```
  br label %merge
}

define float @min(float %a, float %b) {
entry:
  %a1 = alloca float
  store float %a, float* %a1
  %b2 = alloca float
  store float %b, float* %b2
  %a3 = load float, float* %a1
  %b4 = load float, float* %b2
  %tmp = fcmp olt float %a3, %b4
  %if_tmp = alloca float
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %if_tmp7 = load float, float* %if_tmp
  ret float %if_tmp7

then:                                             ; preds = %entry
  %a5 = load float, float* %a1
  store float %a5, float* %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %b6 = load float, float* %b2
  store float %b6, float* %if_tmp
  br label %merge
}

define float @clamp(float %x, float %m, float %M) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %m2 = alloca float
  store float %m, float* %m2
  %M3 = alloca float
  store float %M, float* %M3
  %M4 = load float, float* %M3
  %x5 = load float, float* %x1
  %m6 = load float, float* %m2
  %fxn_result = call float @max(float %x5, float %m6)
  %fxn_result7 = call float @min(float %M4, float %fxn_result)
  ret float %fxn_result7
}

define float @abs(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %tmp = fcmp olt float %x2, 0.000000e+00
  %if_tmp = alloca float
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %if_tmp6 = load float, float* %if_tmp
  ret float %if_tmp6
```

```
then:                                             ; preds = %entry
  %x3 = load float, float* %x1
  %tmp4 = fneg float %x3
  store float %tmp4, float* %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %x5 = load float, float* %x1
  store float %x5, float* %if_tmp
  br label %merge
}

define float @modf(float %x, float %m) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %m2 = alloca float
  store float %m, float* %m2
  %m3 = load float, float* %m2
  %x4 = load float, float* %x1
  %m5 = load float, float* %m2
  %tmp = fdiv float %x4, %m5
  %fxn_result = call float @frac(float %tmp)
  %tmp6 = fmul float %m3, %fxn_result
  ret float %tmp6
}

define float @sin(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @sinf(float %x2)
  ret float %fxn_result
}

define float @cos(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @cosf(float %x2)
  ret float %fxn_result
}

define float @tan(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @tanf(float %x2)
  ret float %fxn_result
}

define float @asin(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
```

```
  %fxn_result = call float @asinf(float %x2)
  ret float %fxn_result
}

define float @acos(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @acosf(float %x2)
  ret float %fxn_result
}

define float @atan(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @atanf(float %x2)
  ret float %fxn_result
}

define float @sqrt(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @sqrtf(float %x2)
  ret float %fxn_result
}

define float @toradians(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @toradiansf(float %x2)
  ret float %fxn_result
}

define float @sqrMagnitude({ float, float }* %p) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %p2 = load { float, float }*, { float, float }** %p1
  %p3 = load { float, float }*, { float, float }** %p1
  %result = call float @__dotf2({ float, float }* %p2,
    { float, float }* %p3)
  ret float %result
}

define float @__dotf2({ float, float }* %a, { float, float }* %b) {
entry:
  %a1 = alloca { float, float }*
  store { float, float }* %a, { float, float }** %a1
  %a2 = load { float, float }*, { float, float }** %a1
  %b3 = alloca { float, float }*
  store { float, float }* %b, { float, float }** %b3
  %b4 = load { float, float }*, { float, float }** %b3
```

```
  %dot = alloca float
  %tmp = alloca float
  store float 0.000000e+00, float* %dot
  %avalref = getelementptr { float, float }, { float,
    float }* %a2, i32 0, i32 0
  %aval = load float, float* %avalref
  %bvalref = getelementptr { float, float }, { float,float
    }* %b4, i32 0, i32 0
  %bval = load float, float* %bvalref
  %tmp5 = fmul float %aval, %bval
  store float %tmp5, float* %tmp
  %tmp6 = load float, float* %tmp
  %res = load float, float* %dot
  %tmp7 = fadd float %tmp6, %res
  store float %tmp7, float* %dot
  %avalref8 = getelementptr { float, float }, { float, float
    }* %a2, i32 0, i32 1
  %aval9 = load float, float* %avalref8
  %bvalref10 = getelementptr { float, float }, { float, float
    }* %b4, i32 0, i32 1
  %bval11 = load float, float* %bvalref10
  %tmp12 = fmul float %aval9, %bval11
  store float %tmp12, float* %tmp
  %tmp13 = load float, float* %tmp
  %res14 = load float, float* %dot
  %tmp15 = fadd float %tmp13, %res14
  store float %tmp15, float* %dot
  %res16 = load float, float* %dot
  ret float %res16
}

define float @magnitude({ float, float }* %p) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %p2 = load { float, float }*, { float, float }** %p1
  %fxn_result = call float @sqrMagnitude({ float, float }*
    %p2)
  %fxn_result3 = call float @sqrt(float %fxn_result)
  ret float %fxn_result3
}

define float @sqrDistance({ float, float }* %a, { float,
    float }* %b) {
entry:
  %a1 = alloca { float, float }*
  store { float, float }* %a, { float, float }** %a1
  %b2 = alloca { float, float }*
  store { float, float }* %b, { float, float }** %b2
  %p = alloca { float, float }*
  %a3 = load { float, float }*, { float, float }** %a1
  %b4 = load { float, float }*, { float, float }** %b2
  %result = call { float, float }* @__subf2({ float, float
    }* %a3, { float, float }* %b4)
  store { float, float }* %result, { float, float }** %p
  %d = alloca float
  %p5 = load { float, float }*, { float, float }** %p
  %fxn_result = call float @sqrMagnitude({ float, float }* %p5)
  store float %fxn_result, float* %d
```

```
  %p6 = load { float, float }*, { float, float }** %p
  %0 = bitcast { float, float }* %p6 to i8*
  tail call void @free(i8* %0)
  %d7 = load float, float* %d
  ret float %d7
}

define { float, float }* @__subf2({ float, float }* %a,
    { float, float }* %b) {
entry:
  %a1 = alloca { float, float }*
  store { float, float }* %a, { float, float }** %a1
  %a2 = load { float, float }*, { float, float }** %a1
  %b3 = alloca { float, float }*
  store { float, float }* %b, { float, float }** %b3
  %b4 = load { float, float }*, { float, float }** %b3
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul
    nuw (i64 ptrtoint (float*
    getelementptr (float, float* null, i32 1) to i64), i64 2) to i32))
  %ret = bitcast i8* %malloccall to { float, float }*
  %avalref = getelementptr { float, float }, { float,
    float }* %a2, i32 0, i32 0
  %aval = load float, float* %avalref
  %bvalref = getelementptr { float, float }, { float,
    float }* %b4, i32 0, i32 0
  %bval = load float, float* %bvalref
  %tmp = fsub float %aval, %bval
  %ref = getelementptr { float, float }, { float, float
    }* %ret, i32 0, i32 0
  store float %tmp, float* %ref
  %avalref5 = getelementptr { float, float }, { float,
    float }* %a2, i32 0, i32 1
  %aval6 = load float, float* %avalref5
  %bvalref7 = getelementptr { float, float }, { float,
    float }* %b4, i32 0, i32 1
  %bval8 = load float, float* %bvalref7
  %tmp9 = fsub float %aval6, %bval8
  %ref10 = getelementptr { float, float }, { float,
    float }* %ret, i32 0, i32 1
  store float %tmp9, float* %ref10
  ret { float, float }* %ret
}

declare noalias i8* @malloc(i32)

declare void @free(i8*)

define float @distance({ float, float }* %a, { float,
    float }* %b) {
entry:
  %a1 = alloca { float, float }*
  store { float, float }* %a, { float, float }** %a1
  %b2 = alloca { float, float }*
  store { float, float }* %b, { float, float }** %b2
  %a3 = load { float, float }*, { float, float }** %a1
  %b4 = load { float, float }*, { float, float }** %b2
  %fxn_result = call float @sqrDistance({ float, float
    }* %a3, { float, float }* %b4)
  %fxn_result5 = call float @sqrt(float %fxn_result)
```

```
  ret float %fxn_result5
}

define { float, float }* @copy_point({ float, float }* %p) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %p2 = load { float, float }*, { float, float }** %p1
  %copied = call { float, float }* @__copy2({ float,
    float }* %p2)
  ret { float, float }* %copied
}

define { float, float }* @__copy2({ float, float }* %to_copy) {
entry:
  %to_copy1 = alloca { float, float }*
  store { float, float }* %to_copy, { float, float
    }** %to_copy1
  %to_copy2 = load { float, float }*, { float, float
    }** %to_copy1
  %malloccall = tail call i8* @malloc(i32 trunc (i64
    mul nuw (i64 ptrtoint (float*
    getelementptr (float, float* null, i32 1) to i64)
        , i64 2) to i32))
  %struct = bitcast i8* %malloccall to { float, float }*
  %flref = getelementptr { float, float }, { float,
    float }* %to_copy2, i32 0, i32 0
  %fl = load float, float* %flref
  %ref = getelementptr { float, float }, { float, float
    }* %struct, i32 0, i32 0
  store float %fl, float* %ref
  %flref3 = getelementptr { float, float }, { float, float
    }* %to_copy2, i32 0, i32 1
  %fl4 = load float, float* %flref3
  %ref5 = getelementptr { float, float }, { float, float
    }* %struct, i32 0, i32 1
  store float %fl4, float* %ref5
  ret { float, float }* %struct
}

define void @free_point({ float, float }* %p) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %p2 = load { float, float }*, { float, float }** %p1
  %0 = bitcast { float, float }* %p2 to i8*
  tail call void @free(i8* %0)
  ret void
}

define { { float, float }**, i32 }* @copy_path({ { float,
    float }**, i32 }* %p) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p, { { float,
    float }**, i32 }** %p1
  %p2 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %p1
  %lenref = getelementptr { { float, float }**, i32 },
```

```
    { { float, float }**, i32 }* %p2, i32 0, i32 1
  %len = load i32, i32* %lenref
  %dataref = getelementptr { { float, float }**, i32 },
    { { float, float }**, i32 }* %p2, i32 0, i32 0
  %data = load { float, float }**, { float, float }***
    %dataref
  %mallocsize = mul i32 %len, ptrtoint (i1** getelementptr
    (i1*, i1** null, i32 1) to i32)
  %malloccall = tail call i8* @malloc(i32 %mallocsize)
  %arrdata = bitcast i8* %malloccall to { float, float }**
  %i = alloca i32
  store i32 0, i32* %i
  br label %loop

loop:                                             ; preds = %loop, %entry
  %i3 = load i32, i32* %i
  %elref = getelementptr { float, float }*, { float,
    float }** %data, i32 %i3
  %el = load { float, float }*, { float, float }** %elref
  %fxn_result = call { float, float }* @copy_point({
    float, float }* %el)
  %storeref = getelementptr { float, float }*, { float,
    float }** %arrdata, i32 %i3
  store { float, float }* %fxn_result, { float, float
    }** %storeref
  %i4 = add i32 %i3, 1
  store i32 %i4, i32* %i
  %i5 = load i32, i32* %i
  %tmp = icmp slt i32 %i5, %len
  br i1 %tmp, label %loop, label %continue

continue:                                         ; preds = %loop
  %malloccall6 = tail call i8* @malloc(i32 ptrtoint
    ({ { float, float }**, i32 }*
    getelementptr ({ { float, float }**, i32 }, { {
        float, float }**, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall6 to { { float, float
    }**, i32 }*
  %arrdata7 = getelementptr { { float, float }**, i32
    }, { { float, float }**, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { { float, float }**, i32 },
    { { float, float }**, i32 }* %arr, i32 0, i32 1
  store { float, float }** %arrdata, { float, float
    }*** %arrdata7
  store i32 %len, i32* %arrlen
  ret { { float, float }**, i32 }* %arr
}

define void @free_path({ { float, float }**, i32 }* %p) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p, { { float,
    float }**, i32 }** %p1
  %p2 = load { { float, float }**, i32 }*, { {
    float, float }**, i32 }** %p1
  %lenref = getelementptr { { float, float }**,
    i32 }, { { float, float }**, i32 }* %p2, i32 0, i32 1
  %len = load i32, i32* %lenref
  %dataref = getelementptr { { float, float }**,
```

```
    i32 }, { { float, float }**,
    i32 }* %p2, i32 0, i32 0
  %data = load { float, float }**, { float,
    float }*** %dataref
  %i = alloca i32
  store i32 0, i32* %i
  br label %loop

loop:                                             ; preds = %loop, %entry
  %i3 = load i32, i32* %i
  %elref = getelementptr { float, float }*,
    { float, float }** %data, i32 %i3
  %el = load { float, float }*, { float,
    float }** %elref
  call void @free_point({ float, float }* %el)
  %i4 = add i32 %i3, 1
  store i32 %i4, i32* %i
  %i5 = load i32, i32* %i
  %tmp = icmp slt i32 %i5, %len
  br i1 %tmp, label %loop, label %continue

continue:                                         ; preds = %loop
  ret void
}

define void @appendhelp_copyin({ { float, float }**,
    i32 }* %in, { { float, float }**, i32 }*
    %from, i32 %i) {
entry:
  %in1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %in, { {
    float, float }**, i32 }** %in1
  %from2 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %from, { {
    float, float }**, i32 }** %from2
  %i3 = alloca i32
  store i32 %i, i32* %i3
  %i4 = load i32, i32* %i3
  %in5 = load { { float, float }**, i32 }*, { {
    float, float }**, i32 }** %in1
  %lenref = getelementptr { { float, float }**,
    i32 }, { { float, float }**, i32 }* %in5, i32 0,
    i32 1
  %len = load i32, i32* %lenref
  %tmp = icmp slt i32 %i4, %len
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  ret void

then:                                             ; preds = %entry
  %in6 = load { { float, float }**, i32 }*, { {
    float, float }**, i32 }** %in1
  %datarefref = getelementptr { { float, float
    }**, i32 }, { { float, float
    }**, i32 }* %in6, i32 0, i32 0
  %dataref = load { float, float }**, { float,
    float }*** %datarefref
  %i7 = load i32, i32* %i3
```

```
  %from8 = load { { float, float }**, i32 }*, {
    { float, float }**, i32 }** %from2
  %i9 = load i32, i32* %i3
  %tmp10 = add i32 %i9, 1
  %dataref11 = getelementptr { { float, float }**,
    i32 }, { { float, float }**, i32 }* %from8, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref11
  %elref = getelementptr { float, float }*, { float,
    float }** %data, i32 %tmp10
  %el = load { float, float }*, { float, float }** %elref
  %copied = call { float, float }* @__copy2.1({ float,
    float }* %el)
  %storeref = getelementptr { float, float }*, { float,
    float }** %dataref, i32 %i7
  store { float, float }* %copied, { float, float }** %storeref
  %in12 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %in1
  %from13 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %from2
  %i14 = load i32, i32* %i3
  %tmp15 = add i32 %i14, 1
  call void @appendhelp_copyin({ { float, float }**,
    i32 }* %in12, { { float, float }**, i32 }* %from13, i32 %tmp15)
  br label %merge

else:                                             ; preds = %entry
  br label %merge
}

define { float, float }* @__copy2.1({ float, float }*
    %to_copy) {
entry:
  %to_copy1 = alloca { float, float }*
  store { float, float }* %to_copy, { float, float
    }** %to_copy1
  %to_copy2 = load { float, float }*, { float, float
    }** %to_copy1
  %malloccall = tail call i8* @malloc(i32 trunc (i64
    mul nuw (i64 ptrtoint (float* getelementptr (float,
    float* null, i32 1) to i64), i64 2) to i32))
  %struct = bitcast i8* %malloccall to { float, float }*
  %flref = getelementptr { float, float }, { float,
    float }* %to_copy2, i32 0, i32 0
  %fl = load float, float* %flref
  %ref = getelementptr { float, float }, { float,
    float }* %struct, i32 0, i32 0
  store float %fl, float* %ref
  %flref3 = getelementptr { float, float }, {
    float, float }* %to_copy2, i32 0, i32 1
  %fl4 = load float, float* %flref3
  %ref5 = getelementptr { float, float }, {
    float, float }* %struct, i32 0, i32 1
  store float %fl4, float* %ref5
  ret { float, float }* %struct
}

define { { float, float }**, i32 }*
    @appendhelp_tail({ { float, float }**, i32 }* %p) {
entry:
```

```
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p,
    { { float, float }**, i32 }** %p1
  %tail = alloca { { float, float }**, i32 }*
  %p2 = load { { float, float }**, i32 }*,
    { { float, float }**, i32 }** %p1
  %lenref = getelementptr { { float, float
    }**, i32 }, { { float, float }**, i32 }*
    %p2, i32 0, i32 1
  %len = load i32, i32* %lenref
  %tmp = sub i32 %len, 1
  %malloccall = tail call i8* @malloc(i32 ptrtoint
    (i1** getelementptr (i1*,i1** null, i32 1) to i32))
  %arrdata = bitcast i8* %malloccall to { float, float }**
  %malloccall3 = tail call i8* @malloc(i32 trunc (i64 mul
    nuw (i64 ptrtoint (float* getelementptr (float, float*
    null, i32 1) to i64), i64 2) to i32))
  %anon = bitcast i8* %malloccall3 to { float, float }*
  %fieldaddr = getelementptr { float, float }, { float,
    float }* %anon, i32 0, i32 0
  store float 0.000000e+00, float* %fieldaddr
  %fieldaddr4 = getelementptr { float, float }, { float,
    float }* %anon, i32 0, i32 1
  store float 0.000000e+00, float* %fieldaddr4
  %storeref = getelementptr { float, float }*, { float,
    float }** %arrdata, i32 0
  store { float, float }* %anon, { float, float }** %storeref
  %malloccall5 = tail call i8* @malloc(i32 ptrtoint
    ({ { float, float }**, i32}* getelementptr
    ({ { float, float }**, i32 }, { { float, float }**, i32 }*
    null, i32 1) to i32))
  %arr = bitcast i8* %malloccall5 to { { float, float }**, i32 }*
  %arrdata6 = getelementptr { { float, float
    }**, i32 }, { { float, float }**, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { { float, float }**, i32 },
    { { float, float }**, i32 }* %arr, i32 0, i32 1
  store { float, float }** %arrdata, { float,
    float }*** %arrdata6
  store i32 1, i32* %arrlen
  %lenref7 = getelementptr { { float, float }**,
    i32 }, { { float, float }**, i32 }* %arr, i32 0, i32 1
  %len8 = load i32, i32* %lenref7
  %oflen = mul i32 %tmp, %len8
  %olddataref = getelementptr { { float, float }**,
    i32 }, { { float, float }**, i32 }* %arr, i32 0, i32 0
  %olddata = load { float, float }**, { float, float }*** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (i1**
    getelementptr (i1*, i1** null, i32 1) to i32)
  %malloccall9 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata10 = bitcast i8* %malloccall9 to
    { float, float }**
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner

loop:                                             ; preds = %inner
  %i18 = load i32, i32* %i
```

```
  store i32 0, i32* %j
  %tmp19 = icmp slt i32 %i18, %oflen
  br i1 %tmp19, label %inner, label %continue

inner:
    ; preds = %loop, %inner, %entry
  %i11 = load i32, i32* %j
  %i12 = load i32, i32* %i
  %elref = getelementptr { float, float }*, { float,
    float }** %olddata, i32 %i11
  %el = load { float, float }*, { float, float }**
    %elref
  %storeref13 = getelementptr { float, float }*,
    { float, float }** %arrdata10, i32 %i12
  store { float, float }* %el, { float, float }**
    %storeref13
  %i14 = add i32 %i12, 1
  store i32 %i14, i32* %i
  %j15 = add i32 %i11, 1
  store i32 %j15, i32* %j
  %j16 = load i32, i32* %j
  %tmp17 = icmp slt i32 %j16, %len8
  br i1 %tmp17, label %inner, label %loop

continue:                                         ; preds = %loop
  %malloccall20 = tail call i8* @malloc(i32 ptrtoint
    ({ { float, float }**, i32 }* getelementptr ({
        { float, float }**, i32 }, { { float, float }**,
        i32 }* null, i32 1) to i32))
  %arr21 = bitcast i8* %malloccall20 to { { float,
    float }**, i32 }*
  %arrdata22 = getelementptr { { float, float }**,
    i32 }, { { float, float }**, i32 }* %arr21,
    i32 0, i32 0
  %arrlen23 = getelementptr { { float, float }**,
    i32 }, { { float, float }**,
    i32 }* %arr21, i32 0, i32 1
  store { float, float }** %arrdata10, { float,
    float }*** %arrdata22
  store i32 %oflen, i32* %arrlen23
  store { { float, float }**, i32 }* %arr21, {
    { float, float }**, i32 }** %tail
  %tail24 = load { { float, float }**, i32 }*, {
    { float, float }**, i32 }** %tail
  %p25 = load { { float, float }**, i32 }*, { {
    float, float }**, i32 }** %p1
  call void @appendhelp_copyin({ { float, float
    }**, i32 }* %tail24, { { float,
    float }**, i32 }* %p25, i32 0)
  %tail26 = load { { float, float }**, i32 }*,
    { { float, float }**, i32 }** %tail
  ret { { float, float }**, i32 }* %tail26
}

define { { float, float }**, i32 }* @append({
    { float, float }**, i32 }* %p1,
    { { float, float }**, i32 }* %p2, float %epsilon) {
entry:
  %p11 = alloca { { float, float }**, i32 }*
```

```
  store { { float, float }**, i32 }* %p1, { {
    float, float }**, i32 }** %p11
  %p22 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p2, { {
    float, float }**, i32 }** %p22
  %epsilon3 = alloca float
  store float %epsilon, float* %epsilon3
  %p14 = load { { float, float }**, i32 }*, { {
    float, float }**, i32 }** %p11
  %lenref = getelementptr { { float, float }**,
    i32 }, { { float, float }**, i32 }* %p14, i32 0, i32 1
  %len = load i32, i32* %lenref
  %tmp = icmp eq i32 %len, 0
  %if_tmp = alloca { { float, float }**, i32 }*
  br i1 %tmp, label %then, label %else

merge:
    ; preds = %merge11, %then
  %if_tmp66 = load { { float, float }**, i32 }*,
    { { float, float }**, i32 }**
    %if_tmp
  ret { { float, float }**, i32 }* %if_tmp66

then:
    ; preds = %entry
  %p25 = load { { float, float }**, i32 }*, { {
    float, float }**, i32 }** %p22
  %fxn_result = call { { float, float }**, i32 }*
    @copy_path({ { float, float }**, i32 }* %p25)
  store { { float, float }**, i32 }* %fxn_result,
    { { float, float }**, i32 }** %if_tmp
  br label %merge

else:
    ; preds = %entry
  %p26 = load { { float, float }**, i32 }*, {
    { float, float }**, i32 }** %p22
  %lenref7 = getelementptr { { float, float }**,
    i32 }, { { float, float }**, i32 }* %p26, i32 0, i32 1
  %len8 = load i32, i32* %lenref7
  %tmp9 = icmp eq i32 %len8, 0
  %if_tmp10 = alloca { { float, float }**, i32 }*
  br i1 %tmp9, label %then12, label %else13

merge11:
    ; preds = %contb, %then12
  %if_tmp65 = load { { float, float }**, i32 }*,
    { { float, float }**, i32 }** %if_tmp10
  store { { float, float }**, i32 }* %if_tmp65,
    { { float, float }**, i32}** %if_tmp
  br label %merge

then12:                                           ; preds = %else
  %p114 = load { { float, float }**, i32 }*,
    { { float, float }**, i32 }** %p11
  %fxn_result15 = call { { float, float }**,
    i32 }* @copy_path({ { float, float }**, i32 }* %p114)
  store { { float, float }**, i32 }* %fxn_result15,
    { { float, float }**, i32 }** %if_tmp10
```

```
  br label %merge11

else13:                                           ; preds = %else
  %merge16 = alloca i1
  %p117 = load { { float, float }**, i32 }*,
    { { float, float }**, i32 }** %p11
  %p118 = load { { float, float }**, i32 }*,
    { { float, float }**, i32 }** %p11
  %lenref19 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %p118, i32 0, i32 1
  %len20 = load i32, i32* %lenref19
  %tmp21 = sub i32 %len20, 1
  %dataref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %p117, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref
  %elref = getelementptr { float, float }*, { float, float }** %data, i32
    %tmp21
  %el = load { float, float }*, { float, float }** %elref
  %p222 = load { { float, float }**, i32 }*, { { float, float }**, i32 }**
    %p22
  %dataref23 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %p222, i32 0, i32 0
  %data24 = load { float, float }**, { float, float }*** %dataref23
  %elref25 = getelementptr { float, float }*, { float, float }** %data24,
    i32 0
  %el26 = load { float, float }*, { float, float }** %elref25
  %fxn_result27 = call float @sqrDistance({ float, float }* %el, { float,
    float }* %el26)
  %epsilon28 = load float, float* %epsilon3
  %epsilon29 = load float, float* %epsilon3
  %tmp30 = fmul float %epsilon28, %epsilon29
  %tmp31 = fcmp olt float %fxn_result27, %tmp30
  store i1 %tmp31, i1* %merge16
  %p2c = alloca { { float, float }**, i32 }*
  %merge32 = load i1, i1* %merge16
  %if_tmp33 = alloca { { float, float }**, i32 }*
  br i1 %merge32, label %then35, label %else36

merge34:
    ; preds = %else36, %then35
  %if_tmp40 = load { { float, float }**, i32 }*, { { float, float
    }**, i32 }** %if_tmp33
  store { { float, float }**, i32 }* %if_tmp40, { { float,
    float }**, i32 }** %p2c
  %ret = alloca { { float, float }**, i32 }*
  %p141 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %p11
  %fxn_result42 = call { { float, float }**, i32 }* @copy_path({
    { float, float }**, i32 }* %p141)
  %p2c43 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %p2c
  %fxn_result44 = call { { float, float }**, i32 }*
    @copy_path({ { float, float }**, i32 }* %p2c43)
  %len1ref = getelementptr { { float, float }**, i32 },
    { { float, float }**, i32 }* %fxn_result42, i32 0, i32 1
  %len1 = load i32, i32* %len1ref
  %len2ref = getelementptr { { float, float }**, i32 },
    { { float, float }**, i32 }* %fxn_result44, i32 0, i32 1
  %len2 = load i32, i32* %len2ref
```

```
  %n = add i32 %len1, %len2
  %data1ref = getelementptr { { float, float }**, i32
    }, { { float, float }**, i32 }* %fxn_result42, i32
    0, i32 0
  %data1 = load { float, float }**, { float, float
    }*** %data1ref
  %data2ref = getelementptr { { float, float }**,
    i32 }, { { float, float
    }**, i32 }* %fxn_result44, i32 0, i32 0
  %data2 = load { float, float }**, { float, float }*** %data2ref
  %mallocsize = mul i32 %n, ptrtoint (i1** getelementptr
    (i1*, i1** null, i32 1) to i32)
  %malloccall = tail call i8* @malloc(i32 %mallocsize)
  %data45 = bitcast i8* %malloccall to { float, float }**
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %loop1

then35:                                           ; preds = %else13
  %p237 = load { { float, float }**, i32 }*, { { float, float }**, i32
    }** %p22
  %fxn_result38 = call { { float, float }**, i32 }* @appendhelp_tail({
    { float, float }**, i32 }* %p237)
  store { { float, float }**, i32 }* %fxn_result38, { { float, float
    }**, i32 }** %if_tmp33
  br label %merge34

else36:                                           ; preds = %else13
  %p239 = load { { float, float }**, i32 }*, { { float, float }**, i32 }**
    %p22
  store { { float, float }**, i32 }* %p239, { { float, float }**, i32 }**
    %if_tmp33
  br label %merge34

loop1:
    ; preds = %loop1, %merge34
  %i46 = load i32, i32* %j
  %i47 = load i32, i32* %i
  %elref48 = getelementptr { float, float }*, { float, float }**
    %data1, i32 %i46
  %el49 = load { float, float }*, { float, float }** %elref48
  %storeref = getelementptr { float, float }*, { float, float }**
    %data45, i32 %i47
  store { float, float }* %el49, { float, float }** %storeref
  %tmp50 = add i32 %i47, 1
  store i32 %tmp50, i32* %i
  %j51 = add i32 %i46, 1
  store i32 %j51, i32* %j
  %j52 = load i32, i32* %j
  %tmp53 = icmp slt i32 %j52, %len1
  br i1 %tmp53, label %loop1, label %inbtw

inbtw:                                            ; preds = %loop1
  store i32 0, i32* %j
  br label %loop2

loop2:                                            ; preds = %loop2, %inbtw
```

```
  %i54 = load i32, i32* %j
  %i55 = load i32, i32* %i
  %elref56 = getelementptr { float, float }*, { float, float }** %data2,
    i32 %i54
  %el57 = load { float, float }*, { float, float }** %elref56
  %storeref58 = getelementptr { float, float }*, { float, float }**
    %data45, i32 %i55
  store { float, float }* %el57, { float, float }** %storeref58
  %tmp59 = add i32 %i55, 1
  store i32 %tmp59, i32* %i
  %j60 = add i32 %i54, 1
  store i32 %j60, i32* %j
  %j61 = load i32, i32* %j
  %tmp62 = icmp slt i32 %j61, %len2
  br i1 %tmp62, label %loop2, label %contb

contb:                                            ; preds = %loop2
  %malloccall63 = tail call i8* @malloc(i32 ptrtoint ({ { float, float
    }**, i32 }* getelementptr ({ { float, float }**, i32 }, { { float,
    float }**, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall63 to { { float, float }**, i32 }*
  %arrdata = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr, i32 0, i32 1
  store { float, float }** %data45, { float, float }*** %arrdata
  store i32 %n, i32* %arrlen
  store { { float, float }**, i32 }* %arr, { { float, float }**,
    i32 }** %ret
  %ret64 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %ret
  store { { float, float }**, i32 }* %ret64, { { float, float }**,
    i32 }** %if_tmp10
  br label %merge11
}

define void @reversedhelp({ { float, float }**, i32 }* %in, { {
    float, float }**, i32 }* %from, i32 %i) {
entry:
  %in1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %in, { { float, float
    }**, i32 }** %in1
  %from2 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %from, { { float, float
    }**, i32 }** %from2
  %i3 = alloca i32
  store i32 %i, i32* %i3
  %i4 = load i32, i32* %i3
  %in5 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %in1
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %in5, i32 0, i32 1
  %len = load i32, i32* %lenref
  %tmp = icmp slt i32 %i4, %len
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  ret void
```

```
then:                                             ; preds = %entry
  %in6 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %in1
  %i7 = load i32, i32* %i3
  %dataref = getelementptr { { float, float }**, i32 },
    { { float, float
    }**, i32 }* %in6, i32 0, i32 0
  %data = load { float, float }**, { float, float }***
    %dataref
  %elref = getelementptr { float, float }*, { float,
    float }** %data, i32 %i7
  %el = load { float, float }*, { float, float }** %elref
  %from8 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %from2
  %in9 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %in1
  %lenref10 = getelementptr { { float, float }**, i32 },
    { { float, float}**, i32 }* %in9, i32 0, i32 1
  %len11 = load i32, i32* %lenref10
  %tmp12 = sub i32 %len11, 1
  %i13 = load i32, i32* %i3
  %tmp14 = sub i32 %tmp12, %i13
  %dataref15 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %from8, i32 0, i32 0
  %data16 = load { float, float }**, { float, float }***
    %dataref15
  %elref17 = getelementptr { float, float }*, { float,
    float }** %data16, i32 %tmp14
  %el18 = load { float, float }*, { float, float }** %elref17
  %fieldadr = getelementptr { float, float }, { float,
    float }* %el18, i32 0, i32 0
  %x = load float, float* %fieldadr
  %ref = getelementptr { float, float }, { float, float }*
    %el, i32 0, i32 0
  store float %x, float* %ref
  %in19 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %in1
  %i20 = load i32, i32* %i3
  %dataref21 = getelementptr { { float, float }**, i32 },
    { { float,
    float }**, i32 }* %in19, i32 0, i32 0
  %data22 = load { float, float }**, { float, float }***
    %dataref21
  %elref23 = getelementptr { float, float }*, { float,
    float }** %data22, i32 %i20
  %el24 = load { float, float }*, { float, float }**
    %elref23
  %from25 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %from2
  %in26 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %in1
  %lenref27 = getelementptr { { float, float }**, i32 },
    { { float,
    float }**, i32 }* %in26, i32 0, i32 1
  %len28 = load i32, i32* %lenref27
  %tmp29 = sub i32 %len28, 1
  %i30 = load i32, i32* %i3
  %tmp31 = sub i32 %tmp29, %i30
  %dataref32 = getelementptr { { float, float }**, i32 }, { { float,
```

```
    float }**, i32 }* %from25, i32 0, i32 0
  %data33 = load { float, float }**, { float, float }*** %dataref32
  %elref34 = getelementptr { float, float }*, { float, float }** %data33,
    i32 %tmp31
  %el35 = load { float, float }*, { float, float }** %elref34
  %fieldadr36 = getelementptr { float, float }, { float, float }* %el35,
    i32 0, i32 1
  %y = load float, float* %fieldadr36
  %ref37 = getelementptr { float, float }, { float, float
    }* %el24, i32 0, i32 1
  store float %y, float* %ref37
  %in38 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %in1
  %from39 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %from2
  %i40 = load i32, i32* %i3
  %tmp41 = add i32 %i40, 1
  call void @reversedhelp({ { float, float }**, i32 }* %in38, { { float,
    float }**, i32 }* %from39, i32 %tmp41)
  br label %merge

else:                                             ; preds = %entry
  br label %merge
}

define { { float, float }**, i32 }* @reversed({ { float,
    float }**, i32 }* %p) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p, { { float, float }**,
    i32 }** %p1
  %newpath = alloca { { float, float }**, i32 }*
  %p2 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %p1
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %p2, i32 0, i32 1
  %len = load i32, i32* %lenref
  %malloccall = tail call i8* @malloc(i32 ptrtoint (i1**
    getelementptr (i1*, i1** null, i32 1) to i32))
  %arrdata = bitcast i8* %malloccall to { float, float }**
  %malloccall3 = tail call i8* @malloc(i32 trunc (i64 mul nuw
    (i64 ptrtoint(float* getelementptr (float, float* null,
    i32 1) to i64), i64 2) to i32))
  %anon = bitcast i8* %malloccall3 to { float, float }*
  %fieldaddr = getelementptr { float, float }, { float, float
    }* %anon, i32 0, i32 0
  store float 0.000000e+00, float* %fieldaddr
  %fieldaddr4 = getelementptr { float, float }, { float, float
    }* %anon, i32 0, i32 1
  store float 0.000000e+00, float* %fieldaddr4
  %storeref = getelementptr { float, float }*, { float, float
    }** %arrdata, i32 0
  store { float, float }* %anon, { float, float }** %storeref
  %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ { float, float }**,
    i32 }* getelementptr ({ { float, float }**, i32 }, { { float, float }**,
    i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall5 to { { float, float }**, i32 }*
  %arrdata6 = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %arr, i32 0, i32 0
```

```
  %arrlen = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %arr, i32 0, i32 1
  store { float, float }** %arrdata, { float, float }*** %arrdata6
  store i32 1, i32* %arrlen
  %lenref7 = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %arr, i32 0, i32 1
  %len8 = load i32, i32* %lenref7
  %oflen = mul i32 %len, %len8
  %olddataref = getelementptr { { float, float }**, i32 },
    { { float, float }**, i32 }* %arr, i32 0, i32 0
  %olddata = load { float, float }**, { float, float
    }*** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (i1**
    getelementptr (i1*, i1** null, i32 1) to i32)
  %malloccall9 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata10 = bitcast i8* %malloccall9 to { float, float }**
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner

loop:                                             ; preds = %inner
  %i17 = load i32, i32* %i
  store i32 0, i32* %j
  %tmp18 = icmp slt i32 %i17, %oflen
  br i1 %tmp18, label %inner, label %continue

inner:                                            ; preds = %loop, %inner, %entry
  %i11 = load i32, i32* %j
  %i12 = load i32, i32* %i
  %elref = getelementptr { float, float }*, { float, float
    }** %olddata, i32 %i11
  %el = load { float, float }*, { float, float }** %elref
  %storeref13 = getelementptr { float, float }*, { float, float
    }** %arrdata10, i32 %i12
  store { float, float }* %el, { float, float }** %storeref13
  %i14 = add i32 %i12, 1
  store i32 %i14, i32* %i
  %j15 = add i32 %i11, 1
  store i32 %j15, i32* %j
  %j16 = load i32, i32* %j
  %tmp = icmp slt i32 %j16, %len8
  br i1 %tmp, label %inner, label %loop

continue:                                         ; preds = %loop
  %malloccall19 = tail call i8* @malloc(i32 ptrtoint
    ({ { float, float }**, i32}* getelementptr ({ { float,
    float }**, i32 }, { { float, float }**, i32
    }* null, i32 1) to i32))
  %arr20 = bitcast i8* %malloccall19 to { { float,
    float }**, i32 }*
  %arrdata21 = getelementptr { { float, float }**,
    i32 }, { { float, float }**, i32 }* %arr20, i32 0, i32 0
  %arrlen22 = getelementptr { { float, float }**, i32
    }, { { float, float }**, i32 }* %arr20, i32 0, i32 1
  store { float, float }** %arrdata10, { float, float
    }*** %arrdata21
  store i32 %oflen, i32* %arrlen22
```

```
  store { { float, float }**, i32 }* %arr20,
    { { float, float }**, i32 }** %newpath
  %newpath23 = load { { float, float }**, i32 }*,
    { { float, float }**, i32 }** %newpath
  %p24 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %p1
  call void @reversedhelp({ { float, float }**, i32 }*
    %newpath23, { { float, float }**, i32 }* %p24, i32 0)
  %newpath25 = load { { float, float }**, i32 }*,
    { { float, float }**, i32 }** %newpath
  ret { { float, float }**, i32 }* %newpath25
}

define void @reversehelp({ { float, float }**, i32 }* %p, i32 %i) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p, { { float, float }**, i32 }** %p1
  %i2 = alloca i32
  store i32 %i, i32* %i2
  %i3 = load i32, i32* %i2
  %p4 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %p4, i32 0, i32 1
  %len = load i32, i32* %lenref
  %tmp = sdiv i32 %len, 2
  %tmp5 = icmp slt i32 %i3, %tmp
  br i1 %tmp5, label %then, label %else

merge:                                            ; preds = %else, %then
  ret void

then:                                             ; preds = %entry
  %q = alloca { float, float }*
  %p6 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %i7 = load i32, i32* %i2
  %dataref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %p6, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref
  %elref = getelementptr { float, float }*, { float, float }** %data, i32 %i7
  %el = load { float, float }*, { float, float }** %elref
  store { float, float }* %el, { float, float }** %q
  %p8 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %datarefref = getelementptr { { float, float }**, i32 }, { { float,
    float }**,i32 }* %p8, i32 0, i32 0
  %dataref9 = load { float, float }**, { float, float }*** %datarefref
  %i10 = load i32, i32* %i2
  %p11 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %p12 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %lenref13 = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %p12, i32 0, i32 1
  %len14 = load i32, i32* %lenref13
  %tmp15 = sub i32 %len14, 1
  %i16 = load i32, i32* %i2
  %tmp17 = sub i32 %tmp15, %i16
  %dataref18 = getelementptr { { float, float }**, i32 }, { { float,
    float }**,i32 }* %p11, i32 0, i32 0
  %data19 = load { float, float }**, { float, float }*** %dataref18
  %elref20 = getelementptr { float, float }*, { float, float }**
    %data19, i32 %tmp17
```

```
  %el21 = load { float, float }*, { float, float }** %elref20
  %storeref = getelementptr { float, float }*, { float, float }**
    %dataref9, i32 %i10
  store { float, float }* %el21, { float, float }** %storeref
  %p22 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %p1
  %datarefref23 = getelementptr { { float, float }**, i32 }, { {
    float, float}**, i32 }* %p22, i32 0, i32 0
  %dataref24 = load { float, float }**, { float, float }***
    %datarefref23
  %p25 = load { { float, float }**, i32 }*, { { float, float
    }**, i32 }** %p1
  %lenref26 = getelementptr { { float, float }**, i32 }, {
    { float, float }**,i32 }* %p25, i32 0, i32 1
  %len27 = load i32, i32* %lenref26
  %tmp28 = sub i32 %len27, 1
  %i29 = load i32, i32* %i2
  %tmp30 = sub i32 %tmp28, %i29
  %q31 = load { float, float }*, { float, float }** %q
  %storeref32 = getelementptr { float, float }*,
    { float, float }** %dataref24,i32 %tmp30
  store { float, float }* %q31, { float, float }** %storeref32
  %p33 = load { { float, float }**, i32 }*, {
    { float, float }**, i32 }** %p1
  %i34 = load i32, i32* %i2
  %tmp35 = add i32 %i34, 1
  call void @reversehelp({ { float, float }**,
    i32 }* %p33, i32 %tmp35)
  br label %merge

else:                                             ; preds = %entry
  br label %merge
}

define void @reverse({ { float, float }**, i32 }* %p) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p, { {
    float, float }**, i32 }** %p1
  %p2 = load { { float, float }**, i32 }*,
    { { float, float }**, i32 }** %p1
  call void @reversehelp({ { float, float }**,
    i32 }* %p2, i32 0)
  ret void
}

define { float, float, float, float }* @rgb(float
    %r, float %g, float %b) {
entry:
  %r1 = alloca float
  store float %r, float* %r1
  %g2 = alloca float
  store float %g, float* %g2
  %b3 = alloca float
  store float %b, float* %b3
  %malloccall = tail call i8* @malloc(i32 trunc
    (i64 mul nuw (i64 ptrtoint (float* getelementptr
    (float, float* null, i32 1) to i64), i64 4) to i32))
  %anon = bitcast i8* %malloccall to { float, float,
```

```
    float, float }*
  %fieldaddr = getelementptr { float, float, float,
    float }, { float, float,
    float, float }* %anon, i32 0, i32 0
  %r4 = load float, float* %r1
  store float %r4, float* %fieldaddr
  %fieldaddr5 = getelementptr { float, float, float,
    float }, { float, float,
    float, float }* %anon, i32 0, i32 1
  %g6 = load float, float* %g2
  store float %g6, float* %fieldaddr5
  %fieldaddr7 = getelementptr { float, float,float, float
    }, { float, float, float, float }* %anon, i32 0, i32 2
  %b8 = load float, float* %b3
  store float %b8, float* %fieldaddr7
  %fieldaddr9 = getelementptr { float, float, float, float
    }, { float, float,float, float }* %anon, i32 0, i32 3
  store float 1.000000e+00, float* %fieldaddr9
  ret { float, float, float, float }* %anon
}

define { float, float, float, float }* @hsv(float %h,
    float %s, float %v) {
entry:
  %h1 = alloca float
  store float %h, float* %h1
  %s2 = alloca float
  store float %s, float* %s2
  %v3 = alloca float
  store float %v, float* %v3
  %c = alloca float
  %v4 = load float, float* %v3
  %s5 = load float, float* %s2
  %tmp = fmul float %v4, %s5
  store float %tmp, float* %c
  %hfac = alloca float
  %h6 = load float, float* %h1
  %tmp7 = fmul float %h6, 6.000000e+00
  %fxn_result = call float @modf(float %tmp7, float 2.000000e+00)
  store float %fxn_result, float* %hfac
  %x = alloca float
  %c8 = load float, float* %c
  %hfac9 = load float, float* %hfac
  %tmp10 = fsub float %hfac9, 1.000000e+00
  %fxn_result11 = call float @abs(float %tmp10)
  %tmp12 = fsub float 1.000000e+00, %fxn_result11
  %tmp13 = fmul float %c8, %tmp12
  store float %tmp13, float* %x
  %m = alloca float
  %v14 = load float, float* %v3
  %c15 = load float, float* %c
  %tmp16 = fsub float %v14, %c15
  store float %tmp16, float* %m
  %hh = alloca float
  %h17 = load float, float* %h1
  %tmp18 = fmul float %h17, 6.000000e+00
  store float %tmp18, float* %hh
  %hh19 = load float, float* %hh
  %tmp20 = fcmp olt float %hh19, 1.000000e+00
```

```
  %if_tmp = alloca { float, float, float, float }*
  br i1 %tmp20, label %then, label %else

merge:                                            ; preds = %merge30, %then
  %if_tmp85 = load { float, float, float, float }*, { float, float, float,
    float }** %if_tmp
  ret { float, float, float, float }* %if_tmp85

then:                                             ; preds = %entry
  %v21 = load float, float* %v3
  %x22 = load float, float* %x
  %m23 = load float, float* %m
  %tmp24 = fadd float %x22, %m23
  %m25 = load float, float* %m
  %fxn_result26 = call { float, float, float, float }* @rgb(float %v21,
    float %tmp24, float %m25)
  store { float, float, float, float }* %fxn_result26, { float, float,
    float, float }** %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %hh27 = load float, float* %hh
  %tmp28 = fcmp olt float %hh27, 2.000000e+00
  %if_tmp29 = alloca { float, float, float, float }*
  br i1 %tmp28, label %then31, label %else32

merge30:
    ; preds = %merge42,%then31
  %if_tmp84 = load { float, float, float, float }*, { float,
    float, float,float }** %if_tmp29
  store { float, float, float, float }* %if_tmp84, { float,
    float, float,float }** %if_tmp
  br label %merge

then31:                                           ; preds = %else
  %x33 = load float, float* %x
  %m34 = load float, float* %m
  %tmp35 = fadd float %x33, %m34
  %v36 = load float, float* %v3
  %m37 = load float, float* %m
  %fxn_result38 = call { float, float, float, float }* @rgb(float %tmp35,
    float %v36, float %m37)
  store { float, float, float, float }* %fxn_result38, { float, float,
    float, float }** %if_tmp29
  br label %merge30

else32:                                           ; preds = %else
  %hh39 = load float, float* %hh
  %tmp40 = fcmp olt float %hh39, 3.000000e+00
  %if_tmp41 = alloca { float, float, float, float }*
  br i1 %tmp40, label %then43, label %else44

merge42:                                          ; preds = %merge54, %then43
  %if_tmp83 = load { float, float, float, float }*, { float, float,
    float, float }** %if_tmp41
  store { float, float, float, float }* %if_tmp83, { float, float,
    float, float }** %if_tmp29
  br label %merge30
```

```
then43:                                           ; preds = %else32
  %m45 = load float, float* %m
  %v46 = load float, float* %v3
  %x47 = load float, float* %x
  %m48 = load float, float* %m
  %tmp49 = fadd float %x47, %m48
  %fxn_result50 = call { float, float, float, float }* @rgb(float %m45,
    float %v46, float %tmp49)
  store { float, float, float, float }* %fxn_result50, { float, float,
    float, float }** %if_tmp41
  br label %merge42

else44:                                           ; preds = %else32
  %hh51 = load float, float* %hh
  %tmp52 = fcmp olt float %hh51, 4.000000e+00
  %if_tmp53 = alloca { float, float, float, float }*
  br i1 %tmp52, label %then55, label %else56

merge54:                                          ; preds = %merge66, %then55
  %if_tmp82 = load { float, float, float, float }*, { float, float,
    float, float }** %if_tmp53
  store { float, float, float, float }* %if_tmp82, { float, float,
    float, float }** %if_tmp41
  br label %merge42

then55:                                           ; preds = %else44
  %m57 = load float, float* %m
  %x58 = load float, float* %x
  %m59 = load float, float* %m
  %tmp60 = fadd float %x58, %m59
  %v61 = load float, float* %v3
  %fxn_result62 = call { float, float, float, float }* @rgb(float %m57,
    float %tmp60, float %v61)
  store { float, float, float, float }* %fxn_result62, { float, float,
    float, float }** %if_tmp53
  br label %merge54

else56:                                           ; preds = %else44
  %hh63 = load float, float* %hh
  %tmp64 = fcmp olt float %hh63, 5.000000e+00
  %if_tmp65 = alloca { float, float, float, float }*
  br i1 %tmp64, label %then67, label %else68

merge66:                                          ; preds = %else68, %then67
  %if_tmp81 = load { float, float, float, float }*, { float, float,
    float, float }** %if_tmp65
  store { float, float, float, float }* %if_tmp81, { float, float,
    float, float }** %if_tmp53
  br label %merge54

then67:                                           ; preds = %else56
  %x69 = load float, float* %x
  %m70 = load float, float* %m
  %tmp71 = fadd float %x69, %m70
  %m72 = load float, float* %m
  %v73 = load float, float* %v3
  %fxn_result74 = call { float, float, float, float }* @rgb(float
    %tmp71, float %m72, float %v73)
  store { float, float, float, float }* %fxn_result74, { float,
```

```
    float, float, float }** %if_tmp65
  br label %merge66

else68:                                           ; preds = %else56
  %v75 = load float, float* %v3
  %m76 = load float, float* %m
  %x77 = load float, float* %x
  %m78 = load float, float* %m
  %tmp79 = fadd float %x77, %m78
  %fxn_result80 = call { float, float, float, float }* @rgb(float
    %v75, float %m76, float %tmp79)
  store { float, float, float, float }* %fxn_result80, { float,
    float, float, float }** %if_tmp65
  br label %merge66
}

define void @startCanvas({ i32, i32, i32 }* %c) {
entry:
  %c1 = alloca { i32, i32, i32 }*
  store { i32, i32, i32 }* %c, { i32, i32, i32 }** %c1
  %c2 = load { i32, i32, i32 }*, { i32, i32, i32 }** %c1
  %fieldadr = getelementptr { i32, i32, i32 }, { i32, i32, i32 }* %c2, i32
    0, i32 0
  %width = load i32, i32* %fieldadr
  %c3 = load { i32, i32, i32 }*, { i32, i32, i32 }** %c1
  %fieldadr4 = getelementptr { i32, i32, i32 }, { i32, i32, i32 }* %c3, i32
    0, i32 1
  %height = load i32, i32* %fieldadr4
  call void @gl_startRendering(i32 %width, i32 %height)
  ret void
}

define void @cvoid() {
entry:
  ret void
}

define void @drawHelper({ { float, float }**, i32 }* %point_structs,
    { { float, float, float, float }**, i32 }* %color_structs,
    i32 %numOfPoints, i32 %i, { float*, i32 }* %points, {
    float*, i32 }* %colors) {
entry:
  %point_structs1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %point_structs, { { float,
    float }**, i32 }** %point_structs1
  %color_structs2 = alloca { { float, float, float, float }**, i32 }*
  store { { float, float, float, float }**, i32 }* %color_structs,
    { { float, float, float, float }**, i32 }** %color_structs2
  %numOfPoints3 = alloca i32
  store i32 %numOfPoints, i32* %numOfPoints3
  %i4 = alloca i32
  store i32 %i, i32* %i4
  %points5 = alloca { float*, i32 }*
  store { float*, i32 }* %points, { float*, i32 }** %points5
  %colors6 = alloca { float*, i32 }*
  store { float*, i32 }* %colors, { float*, i32 }** %colors6
  %i7 = load i32, i32* %i4
  %numOfPoints8 = load i32, i32* %numOfPoints3
  %tmp = icmp sge i32 %i7, %numOfPoints8
```

```
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  ret void

then:                                             ; preds = %entry
  call void @cvoid()
  br label %merge

else:                                             ; preds = %entry
  %px = alloca float
  %point_structs9 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %point_structs1
  %i10 = load i32, i32* %i4
  %dataref = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %point_structs9, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref
  %elref = getelementptr { float, float }*, { float, float
    }** %data, i32 %i10
  %el = load { float, float }*, { float, float }** %elref
  %fieldadr = getelementptr { float, float }, { float, float
    }* %el, i32 0, i32 0
  %x = load float, float* %fieldadr
  store float %x, float* %px
  %py = alloca float
  %point_structs11 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %point_structs1
  %i12 = load i32, i32* %i4
  %dataref13 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %point_structs11, i32 0, i32 0
  %data14 = load { float, float }**, { float, float }*** %dataref13
  %elref15 = getelementptr { float, float }*, { float, float }**
    %data14, i32 %i12
  %el16 = load { float, float }*, { float, float }** %elref15
  %fieldadr17 = getelementptr { float, float }, { float, float }*
    %el16, i32 0, i32 1
  %y = load float, float* %fieldadr17
  store float %y, float* %py
  %points18 = load { float*, i32 }*, { float*, i32 }** %points5
  %datarefref = getelementptr { float*, i32 }, { float*, i32 }*
    %points18, i32 0, i32 0
  %dataref19 = load float*, float** %datarefref
  %i20 = load i32, i32* %i4
  %tmp21 = mul i32 2, %i20
  %px22 = load float, float* %px
  %storeref = getelementptr float, float* %dataref19, i32 %tmp21
  store float %px22, float* %storeref
  %points23 = load { float*, i32 }*, { float*, i32 }** %points5
  %datarefref24 = getelementptr { float*, i32 }, { float*, i32 }*
    %points23, i32 0, i32 0
  %dataref25 = load float*, float** %datarefref24
  %i26 = load i32, i32* %i4
  %tmp27 = mul i32 2, %i26
  %tmp28 = add i32 %tmp27, 1
  %py29 = load float, float* %py
  %storeref30 = getelementptr float, float* %dataref25, i32 %tmp28
  store float %py29, float* %storeref30
  %cr = alloca float
  %color_structs31 = load { { float, float, float, float }**, i32 }*,
```

```
  { { float, float, float, float }**, i32 }** %color_structs2
%i32 = load i32, i32* %i4
%dataref33 = getelementptr { { float, float, float, float }**,i32 },
  { { float, float, float, float }**, i32 }* %color_structs31, i32 0,
  i32 0
%data34 = load { float, float, float, float }**, { float, float,
  float, float }*** %dataref33
%elref35 = getelementptr { float, float, float, float }*, { float,
  float, float, float }** %data34, i32 %i32
%el36 = load { float, float, float, float }*, { float, float,
  float, float }** %elref35
%fieldadr37 = getelementptr { float, float, float, float }, { float,
  float, float, float }* %el36, i32 0, i32 0
%r = load float, float* %fieldadr37
store float %r, float* %cr
%cg = alloca float
%color_structs38 = load { { float, float, float, float }**, i32 }*,
  { { float, float, float, float }**, i32 }** %color_structs2
%i39 = load i32, i32* %i4
%dataref40 = getelementptr { { float, float, float, float }**, i32 },
  { { float, float, float, float }**, i32 }* %color_structs38, i32
  0, i32 0
%data41 = load { float, float, float, float }**, { float, float,
  float, float }*** %dataref40
%elref42 = getelementptr { float, float, float, float }*, { float,
  float, float, float }** %data41, i32 %i39
%el43 = load { float, float, float, float }*, { float, float, float,
  float }** %elref42
%fieldadr44 = getelementptr { float, float, float, float }, { float,
  float, float, float }* %el43, i32 0, i32 1
%g = load float, float* %fieldadr44
store float %g, float* %cg
%cb = alloca float
%color_structs45 = load { { float, float, float, float }**, i32 }*,
  { { float, float, float, float }**, i32 }** %color_structs2
%i46 = load i32, i32* %i4
%dataref47 = getelementptr { { float, float, float, float }**, i32 },
  { { float, float, float, float }**, i32 }* %color_structs45,
  i32 0, i32 0
%data48 = load { float, float, float, float }**, { float, float,
  float, float }*** %dataref47
%elref49 = getelementptr { float, float, float, float }*, { float,
  float, float, float }** %data48, i32 %i46
%el50 = load { float, float, float, float }*, { float, float, float,
  float }** %elref49
%fieldadr51 = getelementptr { float, float, float, float }, { float,
  float, float, float }* %el50, i32 0, i32 2
%b = load float, float* %fieldadr51
store float %b, float* %cb
%ca = alloca float
%color_structs52 = load { { float, float, float, float }**, i32 }*,
  { { float, float, float, float }**, i32 }** %color_structs2
%i53 = load i32, i32* %i4
%dataref54 = getelementptr { { float, float, float, float }**, i32 },
  { { float, float, float, float }**, i32 }* %color_structs52,
  i32 0, i32 0
%data55 = load { float, float, float, float }**, { float, float,
  float, float }*** %dataref54
%elref56 = getelementptr { float, float, float, float }*, { float,
```

```
  float, float, float }** %data55, i32 %i53
%el57 = load { float, float, float, float }*, { float, float, float,
  float }** %elref56
%fieldadr58 = getelementptr { float, float, float, float }, { float,
  float, float, float }* %el57, i32 0, i32 3
%a = load float, float* %fieldadr58
store float %a, float* %ca
%colors59 = load { float*, i32 }*, { float*, i32 }** %colors6
%datarefref60 = getelementptr { float*, i32 }, { float*, i32 }*
  %colors59, i32 0, i32 0
%dataref61 = load float*, float** %datarefref60
%i62 = load i32, i32* %i4
%tmp63 = mul i32 4, %i62
%cr64 = load float, float* %cr
%storeref65 = getelementptr float, float* %dataref61, i32 %tmp63
store float %cr64, float* %storeref65
%colors66 = load { float*, i32 }*, { float*, i32 }** %colors6
%datarefref67 = getelementptr { float*, i32 }, { float*, i32 }*
  %colors66, i32 0, i32 0
%dataref68 = load float*, float** %datarefref67
%i69 = load i32, i32* %i4
%tmp70 = mul i32 4, %i69
%tmp71 = add i32 %tmp70, 1
%cg72 = load float, float* %cg
%storeref73 = getelementptr float, float* %dataref68, i32 %tmp71
store float %cg72, float* %storeref73
%colors74 = load { float*, i32 }*, { float*, i32 }** %colors6
%datarefref75 = getelementptr { float*, i32 }, { float*, i32 }*
  %colors74, i32 0, i32 0
%dataref76 = load float*, float** %datarefref75
%i77 = load i32, i32* %i4
%tmp78 = mul i32 4, %i77
%tmp79 = add i32 %tmp78, 2
%cb80 = load float, float* %cb
%storeref81 = getelementptr float, float* %dataref76, i32 %tmp79
store float %cb80, float* %storeref81
%colors82 = load { float*, i32 }*, { float*, i32 }** %colors6
%datarefref83 = getelementptr { float*, i32 }, { float*, i32 }*
  %colors82, i32 0, i32 0
%dataref84 = load float*, float** %datarefref83
%i85 = load i32, i32* %i4
%tmp86 = mul i32 4, %i85
%tmp87 = add i32 %tmp86, 3
%ca88 = load float, float* %ca
%storeref89 = getelementptr float, float* %dataref84, i32 %tmp87
store float %ca88, float* %storeref89
%point_structs90 = load { { float, float }**, i32 }*, { { float,
  float }**, i32 }** %point_structs1
%color_structs91 = load { { float, float, float, float }**, i32 }*,
  { { float, float, float, float }**, i32 }** %color_structs2
%numOfPoints92 = load i32, i32* %numOfPoints3
%i93 = load i32, i32* %i4
%tmp94 = add i32 %i93, 1
%points95 = load { float*, i32 }*, { float*, i32 }** %points5
%colors96 = load { float*, i32 }*, { float*, i32 }** %colors6
call void @drawHelper({ { float, float }**, i32 }* %point_structs90,
  { { float, float, float, float }**, i32 }* %color_structs91, i32
      %numOfPoints92, i32 %tmp94, { float*, i32 }* %points95,
      { float*, i32 }* %colors96)
```

```
  br label %merge
}

define void @drawPoints({ { float, float }**, i32 }* %point_structs,
    { { float, float, float, float }**, i32 }* %color_structs) {
entry:
  %point_structs1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %point_structs, { { float,
    float }**, i32 }** %point_structs1
  %color_structs2 = alloca { { float, float, float, float }**, i32 }*
  store { { float, float, float, float }**, i32 }* %color_structs,
    { { float, float, float, float }**, i32 }** %color_structs2
  %numOfPoints = alloca i32
  %point_structs3 = load { { float, float }**, i32 }*, {
    { float, float }**, i32 }** %point_structs1
  %lenref = getelementptr { { float, float }**, i32 }, {
    { float, float }**, i32 }* %point_structs3, i32 0, i32 1
  %len = load i32, i32* %lenref
  store i32 %len, i32* %numOfPoints
  %points = alloca { float*, i32 }*
  %numOfPoints4 = load i32, i32* %numOfPoints
  %tmp = mul i32 %numOfPoints4, 2
  %malloccall = tail call i8* @malloc(i32 ptrtoint (float*
    getelementptr(float, float* null, i32 1) to i32))
  %arrdata = bitcast i8* %malloccall to float*
  %storeref = getelementptr float, float* %arrdata, i32 0
  store float 0.000000e+00, float* %storeref
  %malloccall5 = tail call i8* @malloc(i32 ptrtoint
    ({ float*, i32 }* getelementptr ({ float*, i32 },
    { float*, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall5 to { float*, i32 }*
  %arrdata6 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { float*, i32 }, { float*,
    i32 }* %arr, i32 0, i32 1
  store float* %arrdata, float** %arrdata6
  store i32 1, i32* %arrlen
  %lenref7 = getelementptr { float*, i32 }, { float*, i32 }*
    %arr, i32 0, i32 1
  %len8 = load i32, i32* %lenref7
  %oflen = mul i32 %tmp, %len8
  %olddataref = getelementptr { float*, i32 }, { float*,
    i32 }* %arr, i32 0, i32 0
  %olddata = load float*, float** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (float* getelementptr
    (float,float* null, i32 1) to i32)
  %malloccall9 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata10 = bitcast i8* %malloccall9 to float*
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner

loop:
    ; preds = %inner
  %i18 = load i32, i32* %i
  store i32 0, i32* %j
  %tmp19 = icmp slt i32 %i18, %oflen
```

```
  br i1 %tmp19, label %inner, label %continue

inner:
    ; preds = %loop, %inner, %entry
  %i11 = load i32, i32* %j
  %i12 = load i32, i32* %i
  %elref = getelementptr float, float* %olddata, i32 %i11
  %el = load float, float* %elref
  %storeref13 = getelementptr float, float* %arrdata10, i32 %i12
  store float %el, float* %storeref13
  %i14 = add i32 %i12, 1
  store i32 %i14, i32* %i
  %j15 = add i32 %i11, 1
  store i32 %j15, i32* %j
  %j16 = load i32, i32* %j
  %tmp17 = icmp slt i32 %j16, %len8
  br i1 %tmp17, label %inner, label %loop

continue:
    ; preds = %loop
  %malloccall20 = tail call i8* @malloc(i32 ptrtoint
    ({ float*, i32 }* getelementptr ({ float*, i32 },
    { float*, i32 }* null, i32 1) to i32))
  %arr21 = bitcast i8* %malloccall20 to { float*,
    i32 }*
  %arrdata22 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr21, i32 0, i32 0
  %arrlen23 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr21, i32 0, i32 1
  store float* %arrdata10, float** %arrdata22
  store i32 %oflen, i32* %arrlen23
  store { float*, i32 }* %arr21, { float*, i32 }**
    %points
  %colors = alloca { float*, i32 }*
  %numOfPoints24 = load i32, i32* %numOfPoints
  %tmp25 = mul i32 %numOfPoints24, 4
  %malloccall26 = tail call i8* @malloc(i32 ptrtoint
    (float* getelementptr (float, float* null, i32 1)
    to i32))
  %arrdata27 = bitcast i8* %malloccall26 to float*
  %storeref28 = getelementptr float, float* %arrdata27, i32 0
  store float 0.000000e+00, float* %storeref28
  %malloccall29 = tail call i8* @malloc(i32 ptrtoint
    ({ float*, i32 }* getelementptr ({ float*, i32 },
    { float*, i32 }* null, i32 1) to i32))
  %arr30 = bitcast i8* %malloccall29 to { float*, i32 }*
  %arrdata31 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr30, i32 0, i32 0
  %arrlen32 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr30, i32 0, i32 1
  store float* %arrdata27, float** %arrdata31
  store i32 1, i32* %arrlen32
  %lenref33 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr30, i32 0, i32 1
  %len34 = load i32, i32* %lenref33
  %oflen35 = mul i32 %tmp25, %len34
  %olddataref36 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr30, i32 0, i32 0
  %olddata37 = load float*, float** %olddataref36
```

```
  %mallocsize38 = mul i32 %oflen35, ptrtoint (float*
    getelementptr (float,
    float* null, i32 1) to i32)
  %malloccall39 = tail call i8* @malloc(i32 %mallocsize38)
  %arrdata40 = bitcast i8* %malloccall39 to float*
  %i41 = alloca i32
  store i32 0, i32* %i41
  %j42 = alloca i32
  store i32 0, i32* %j42
  br label %inner44

loop43:
    ; preds = %inner44
  %i55 = load i32, i32* %i41
  store i32 0, i32* %j42
  %tmp56 = icmp slt i32 %i55, %oflen35
  br i1 %tmp56, label %inner44, label %continue45

inner44:
    ; preds = %loop43, %inner44, %continue
  %i46 = load i32, i32* %j42
  %i47 = load i32, i32* %i41
  %elref48 = getelementptr float, float* %olddata37, i32 %i46
  %el49 = load float, float* %elref48
  %storeref50 = getelementptr float, float* %arrdata40, i32 %i47
  store float %el49, float* %storeref50
  %i51 = add i32 %i47, 1
  store i32 %i51, i32* %i41
  %j52 = add i32 %i46, 1
  store i32 %j52, i32* %j42
  %j53 = load i32, i32* %j42
  %tmp54 = icmp slt i32 %j53, %len34
  br i1 %tmp54, label %inner44, label %loop43

continue45:
    ; preds = %loop43
  %malloccall57 = tail call i8* @malloc(i32
    ptrtoint ({ float*, i32 }*getelementptr
    ({ float*, i32 }, { float*, i32}* null, i32 1) to i32))
  %arr58 = bitcast i8* %malloccall57 to { float*, i32 }*
  %arrdata59 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr58, i32 0, i32 0
  %arrlen60 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr58, i32 0, i32 1
  store float* %arrdata40, float** %arrdata59
  store i32 %oflen35, i32* %arrlen60
  store { float*, i32 }* %arr58, { float*, i32 }** %colors
  %point_structs61 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %point_structs1
  %color_structs62 = load { { float, float, float, float }**, i32 }*,
    { { float, float, float, float }**, i32 }** %color_structs2
  %numOfPoints63 = load i32, i32* %numOfPoints
  %points64 = load { float*, i32 }*, { float*, i32 }** %points
  %colors65 = load { float*, i32 }*, { float*, i32 }** %colors
  call void @drawHelper({ { float, float }**, i32 }* %point_structs61,
    { { float, float, float, float }**, i32 }* %color_structs62, i32
    %numOfPoints63, i32 0, { float*, i32 }* %points64,
    { float*, i32 }* %colors65)
  %points66 = load { float*, i32 }*, { float*, i32 }** %points
```

```
  %colors67 = load { float*, i32 }*, { float*, i32 }** %colors
  call void @gl_drawPoint({ float*, i32 }* %points66, { float*, i32
    }* %colors67, i32 2)
  ret void
}

define void @drawPath({ { float, float }**, i32 }* %point_structs,
    { { float, float, float, float }**, i32 }* %color_structs,
    i32 %colorMode) {
entry:
  %point_structs1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %point_structs, { { float,
    float }**, i32 }** %point_structs1
  %color_structs2 = alloca { { float, float, float, float }**, i32 }*
  store { { float, float, float, float }**, i32 }* %color_structs,
    { { float, float, float, float }**, i32 }** %color_structs2
  %colorMode3 = alloca i32
  store i32 %colorMode, i32* %colorMode3
  %numOfPoints = alloca i32
  %point_structs4 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %point_structs1
  %lenref = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %point_structs4, i32 0, i32 1
  %len = load i32, i32* %lenref
  store i32 %len, i32* %numOfPoints
  %points = alloca { float*, i32 }*
  %numOfPoints5 = load i32, i32* %numOfPoints
  %tmp = mul i32 %numOfPoints5, 2
  %malloccall = tail call i8* @malloc(i32 ptrtoint (float*
    getelementptr (float, float* null, i32 1) to i32))
  %arrdata = bitcast i8* %malloccall to float*
  %storeref = getelementptr float, float* %arrdata, i32 0
  store float 0.000000e+00, float* %storeref
  %malloccall6 = tail call i8* @malloc(i32 ptrtoint ({
    float*, i32 }* getelementptr ({ float*, i32 }, { float*,
    i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall6 to { float*, i32 }*
  %arrdata7 = getelementptr { float*, i32 }, { float*, i32
    }* %arr, i32 0, i32 0
  %arrlen = getelementptr { float*, i32 }, { float*,
    i32 }* %arr, i32 0, i32 1
  store float* %arrdata, float** %arrdata7
  store i32 1, i32* %arrlen
  %lenref8 = getelementptr { float*, i32 },
    { float*, i32 }* %arr, i32 0, i32 1
  %len9 = load i32, i32* %lenref8
  %oflen = mul i32 %tmp, %len9
  %olddataref = getelementptr { float*, i32 },
    { float*, i32 }* %arr, i32 0, i32 0
  %olddata = load float*, float** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (float*
    getelementptr (float, float* null, i32 1) to i32)
  %malloccall10 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata11 = bitcast i8* %malloccall10 to float*
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner
```

```
loop:
    ; preds = %inner
  %i19 = load i32, i32* %i
  store i32 0, i32* %j
  %tmp20 = icmp slt i32 %i19, %oflen
  br i1 %tmp20, label %inner, label %continue

inner:
    ; preds = %loop, %inner, %entry
  %i12 = load i32, i32* %j
  %i13 = load i32, i32* %i
  %elref = getelementptr float, float* %olddata, i32 %i12
  %el = load float, float* %elref
  %storeref14 = getelementptr float, float* %arrdata11, i32 %i13
  store float %el, float* %storeref14
  %i15 = add i32 %i13, 1
  store i32 %i15, i32* %i
  %j16 = add i32 %i12, 1
  store i32 %j16, i32* %j
  %j17 = load i32, i32* %j
  %tmp18 = icmp slt i32 %j17, %len9
  br i1 %tmp18, label %inner, label %loop

continue:                                         ; preds = %loop
  %malloccall21 = tail call i8* @malloc(i32
    ptrtoint ({ float*, i32 }*
    getelementptr ({ float*, i32 }, { float*,
    i32 }* null, i32 1) to i32))
  %arr22 = bitcast i8* %malloccall21 to { float*, i32 }*
  %arrdata23 = getelementptr { float*, i32 },
    { float*, i32 }* %arr22, i32 0, i32 0
  %arrlen24 = getelementptr { float*, i32 },
    { float*, i32 }* %arr22, i32 0, i32 1
  store float* %arrdata11, float** %arrdata23
  store i32 %oflen, i32* %arrlen24
  store { float*, i32 }* %arr22, { float*, i32
    }** %points
  %colors = alloca { float*, i32 }*
  %numOfPoints25 = load i32, i32* %numOfPoints
  %tmp26 = mul i32 %numOfPoints25, 4
  %malloccall27 = tail call i8* @malloc(i32
    ptrtoint (float*
    getelementptr (float, float* null, i32 1) to i32))
  %arrdata28 = bitcast i8* %malloccall27 to float*
  %storeref29 = getelementptr float, float* %arrdata28, i32 0
  store float 0.000000e+00, float* %storeref29
  %malloccall30 = tail call i8* @malloc(i32 ptrtoint
    ({ float*,i32 }* getelementptr ({ float*, i32 },
    { float*, i32 }* null, i32 1) to i32))
  %arr31 = bitcast i8* %malloccall30 to { float*, i32 }*
  %arrdata32 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr31, i32 0, i32 0
  %arrlen33 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr31, i32 0, i32 1
  store float* %arrdata28, float** %arrdata32
  store i32 1, i32* %arrlen33
  %lenref34 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr31, i32 0, i32 1
```

```
  %len35 = load i32, i32* %lenref34
  %oflen36 = mul i32 %tmp26, %len35
  %olddataref37 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr31, i32 0, i32 0
  %olddata38 = load float*, float** %olddataref37
  %mallocsize39 = mul i32 %oflen36, ptrtoint (float*
    getelementptr(float, float* null, i32 1) to i32)
  %malloccall40 = tail call i8* @malloc(i32 %mallocsize39)
  %arrdata41 = bitcast i8* %malloccall40 to float*
  %i42 = alloca i32
  store i32 0, i32* %i42
  %j43 = alloca i32
  store i32 0, i32* %j43
  br label %inner45

loop44:                                           ; preds = %inner45
  %i56 = load i32, i32* %i42
  store i32 0, i32* %j43
  %tmp57 = icmp slt i32 %i56, %oflen36
  br i1 %tmp57, label %inner45, label %continue46

inner45:
    ; preds = %loop44, %inner45, %continue
  %i47 = load i32, i32* %j43
  %i48 = load i32, i32* %i42
  %elref49 = getelementptr float, float* %olddata38, i32 %i47
  %el50 = load float, float* %elref49
  %storeref51 = getelementptr float, float* %arrdata41, i32 %i48
  store float %el50, float* %storeref51
  %i52 = add i32 %i48, 1
  store i32 %i52, i32* %i42
  %j53 = add i32 %i47, 1
  store i32 %j53, i32* %j43
  %j54 = load i32, i32* %j43
  %tmp55 = icmp slt i32 %j54, %len35
  br i1 %tmp55, label %inner45, label %loop44

continue46:                                       ; preds = %loop44
  %malloccall58 = tail call i8* @malloc(i32 ptrtoint
    ({ float*, i32 }*
    getelementptr ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr59 = bitcast i8* %malloccall58 to { float*, i32 }*
  %arrdata60 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr59, i32 0, i32 0
  %arrlen61 = getelementptr { float*, i32 }, { float*,
    i32 }* %arr59, i32 0, i32 1
  store float* %arrdata41, float** %arrdata60
  store i32 %oflen36, i32* %arrlen61
  store { float*, i32 }* %arr59, { float*, i32 }** %colors
  %point_structs62 = load { { float, float }**, i32 }*,
    { { float, float}**, i32 }** %point_structs1
  %color_structs63 = load { { float, float, float, float }**, i32 }*,
    { { float, float, float, float }**, i32 }** %color_structs2
  %numOfPoints64 = load i32, i32* %numOfPoints
  %points65 = load { float*, i32 }*, { float*, i32 }** %points
  %colors66 = load { float*, i32 }*, { float*, i32 }** %colors
  call void @drawHelper({ { float, float }**, i32 }* %point_structs62,
    { { float, float, float, float }**, i32 }* %color_structs63, i32
    %numOfPoints64, i32 0, { float*, i32 }* %points65, { float*, i32 }
```

```
    * %colors66)
  %points67 = load { float*, i32 }*, { float*, i32 }** %points
  %colors68 = load { float*, i32 }*, { float*, i32 }** %colors
  %colorMode69 = load i32, i32* %colorMode3
  call void @gl_drawCurve({ float*, i32 }* %points67, { float*, i32 }*
    %colors68, i32 %colorMode69)
  ret void
}

define void @drawShape({ { float, float }**, i32 }* %point_structs, {
    { float, float, float, float }**, i32 }* %color_structs,
    i32 %colorMode, i32 %filled) {
entry:
  %point_structs1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %point_structs,
    { { float, float }**, i32 }** %point_structs1
  %color_structs2 = alloca { { float, float, float, float }**, i32 }*
  store { { float, float, float, float }**, i32 }* %color_structs, {
    { float, float, float, float }**, i32 }** %color_structs2
  %colorMode3 = alloca i32
  store i32 %colorMode, i32* %colorMode3
  %filled4 = alloca i32
  store i32 %filled, i32* %filled4
  %numOfPoints = alloca i32
  %point_structs5 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %point_structs1
  %lenref = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %point_structs5, i32 0, i32 1
  %len = load i32, i32* %lenref
  store i32 %len, i32* %numOfPoints
  %points = alloca { float*, i32 }*
  %numOfPoints6 = load i32, i32* %numOfPoints
  %tmp = mul i32 %numOfPoints6, 2
  %malloccall = tail call i8* @malloc(i32 ptrtoint (float*
    getelementptr (float, float* null, i32 1) to i32))
  %arrdata = bitcast i8* %malloccall to float*
  %storeref = getelementptr float, float* %arrdata, i32 0
  store float 0.000000e+00, float* %storeref
  %malloccall7 = tail call i8* @malloc(i32 ptrtoint ({ float*,
    i32 }* getelementptr ({ float*, i32 }, { float*, i32 }*
    null, i32 1) to i32))
  %arr = bitcast i8* %malloccall7 to { float*, i32 }*
  %arrdata8 = getelementptr { float*, i32 }, { float*, i32 }*
    %arr, i32 0, i32 0
  %arrlen = getelementptr { float*, i32 }, { float*, i32 }*
    %arr, i32 0, i32 1
  store float* %arrdata, float** %arrdata8
  store i32 1, i32* %arrlen
  %lenref9 = getelementptr { float*, i32 }, { float*, i32 }*
    %arr, i32 0, i32 1
  %len10 = load i32, i32* %lenref9
  %oflen = mul i32 %tmp, %len10
  %olddataref = getelementptr { float*, i32 }, { float*, i32
    }* %arr, i32 0, i32 0
  %olddata = load float*, float** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (float* getelementptr
    (float, float* null, i32 1) to i32)
  %malloccall11 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata12 = bitcast i8* %malloccall11 to float*
```

```
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner

loop:
    ; preds = %inner
  %i20 = load i32, i32* %i
  store i32 0, i32* %j
  %tmp21 = icmp slt i32 %i20, %oflen
  br i1 %tmp21, label %inner, label %continue

inner:
    ; preds = %loop, %inner, %entry
  %i13 = load i32, i32* %j
  %i14 = load i32, i32* %i
  %elref = getelementptr float, float* %olddata, i32 %i13
  %el = load float, float* %elref
  %storeref15 = getelementptr float, float* %arrdata12, i32 %i14
  store float %el, float* %storeref15
  %i16 = add i32 %i14, 1
  store i32 %i16, i32* %i
  %j17 = add i32 %i13, 1
  store i32 %j17, i32* %j
  %j18 = load i32, i32* %j
  %tmp19 = icmp slt i32 %j18, %len10
  br i1 %tmp19, label %inner, label %loop

continue:                                         ; preds = %loop
  %malloccall22 = tail call i8* @malloc(i32 ptrtoint
    ({ float*,
    i32 }* getelementptr ({ float*, i32 }, { float*,
    i32 }* null, i32 1) to i32))
  %arr23 = bitcast i8* %malloccall22 to { float*, i32 }*
  %arrdata24 = getelementptr { float*, i32 }, {
    float*, i32 }* %arr23, i32 0, i32 0
  %arrlen25 = getelementptr { float*, i32 },
    { float*, i32 }* %arr23, i32 0, i32 1
  store float* %arrdata12, float** %arrdata24
  store i32 %oflen, i32* %arrlen25
  store { float*, i32 }* %arr23, { float*,
    i32 }** %points
  %colors = alloca { float*, i32 }*
  %numOfPoints26 = load i32, i32* %numOfPoints
  %tmp27 = mul i32 %numOfPoints26, 4
  %malloccall28 = tail call i8* @malloc(i32 ptrtoint (float*
    getelementptr (float, float* null, i32 1) to i32))
  %arrdata29 = bitcast i8* %malloccall28 to float*
  %storeref30 = getelementptr float, float* %arrdata29, i32 0
  store float 0.000000e+00, float* %storeref30
  %malloccall31 = tail call i8* @malloc(i32
    ptrtoint ({ float*, i32 }* getelementptr
    ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr32 = bitcast i8* %malloccall31 to { float*, i32 }*
  %arrdata33 = getelementptr { float*, i32 },
    { float*, i32 }* %arr32, i32 0, i32 0
  %arrlen34 = getelementptr { float*, i32 },
    { float*, i32 }* %arr32, i32 0, i32 1
```

```
  store float* %arrdata29, float** %arrdata33
  store i32 1, i32* %arrlen34
  %lenref35 = getelementptr { float*, i32 },
    { float*, i32 }* %arr32, i32 0, i32 1
  %len36 = load i32, i32* %lenref35
  %oflen37 = mul i32 %tmp27, %len36
  %olddataref38 = getelementptr { float*, i32 },
    { float*, i32 }* %arr32, i32 0, i32 0
  %olddata39 = load float*, float** %olddataref38
  %mallocsize40 = mul i32 %oflen37, ptrtoint (float* getelementptr
    (float, float* null, i32 1) to i32)
  %malloccall41 = tail call i8* @malloc(i32 %mallocsize40)
  %arrdata42 = bitcast i8* %malloccall41 to float*
  %i43 = alloca i32
  store i32 0, i32* %i43
  %j44 = alloca i32
  store i32 0, i32* %j44
  br label %inner46

loop45:                                           ; preds = %inner46
  %i57 = load i32, i32* %i43
  store i32 0, i32* %j44
  %tmp58 = icmp slt i32 %i57, %oflen37
  br i1 %tmp58, label %inner46, label %continue47

inner46:
    ; preds = %loop45, %inner46, %continue
  %i48 = load i32, i32* %j44
  %i49 = load i32, i32* %i43
  %elref50 = getelementptr float, float* %olddata39, i32 %i48
  %el51 = load float, float* %elref50
  %storeref52 = getelementptr float, float* %arrdata42, i32 %i49
  store float %el51, float* %storeref52
  %i53 = add i32 %i49, 1
  store i32 %i53, i32* %i43
  %j54 = add i32 %i48, 1
  store i32 %j54, i32* %j44
  %j55 = load i32, i32* %j44
  %tmp56 = icmp slt i32 %j55, %len36
  br i1 %tmp56, label %inner46, label %loop45

continue47:
    ; preds = %loop45
  %malloccall59 = tail call i8* @malloc(i32
    ptrtoint ({ float*, i32 }*getelementptr
    ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr60 = bitcast i8* %malloccall59 to { float*, i32 }*
  %arrdata61 = getelementptr { float*, i32 },
    { float*, i32 }* %arr60, i32 0, i32 0
  %arrlen62 = getelementptr { float*, i32 },
    { float*, i32 }* %arr60, i32 0, i32 1
  store float* %arrdata42, float** %arrdata61
  store i32 %oflen37, i32* %arrlen62
  store { float*, i32 }* %arr60, { float*,
    i32 }** %colors
  %point_structs63 = load { { float, float }**,
    i32 }*, { { float, float }**, i32 }** %point_structs1
  %color_structs64 = load { { float, float, float,
    float }**, i32 }*, {
```

```
    { float, float, float, float }**, i32 }** %color_structs2
  %numOfPoints65 = load i32, i32* %numOfPoints
  %points66 = load { float*, i32 }*, { float*,
    i32 }** %points
  %colors67 = load { float*, i32 }*, { float*,
    i32 }** %colors
  call void @drawHelper({ { float, float }**, i32 }* %point_structs63, {
    { float, float, float, float }**, i32 }*
        %color_structs64, i32
    %numOfPoints65, i32 0, { float*, i32 }*
        %points66, { float*, i32 }* %colors67)
  %points68 = load { float*, i32 }*, { float*,
    i32 }** %points
  %colors69 = load { float*, i32 }*, { float*,
    i32 }** %colors
  %colorMode70 = load i32, i32* %colorMode3
  %filled71 = load i32, i32* %filled4
  call void @gl_drawShape({ float*, i32 }*
    %points68, { float*, i32 }* %colors69, i32 %colorMode70, i32 %filled71)
  ret void
}

define void @endCanvas({ i32, i32, i32 }* %c) {
entry:
  %c1 = alloca { i32, i32, i32 }*
  store { i32, i32, i32 }* %c, { i32, i32, i32 }** %c1
  %c2 = load { i32, i32, i32 }*, { i32, i32, i32 }** %c1
  %fieldadr = getelementptr { i32, i32, i32 }, { i32, i32, i32 }* %c2, i32
    0, i32 0
  %width = load i32, i32* %fieldadr
  %c3 = load { i32, i32, i32 }*, { i32, i32, i32 }** %c1
  %fieldadr4 = getelementptr { i32, i32, i32 }, { i32, i32, i32 }* %c3, i32
    0, i32 1
  %height = load i32, i32* %fieldadr4
  %c5 = load { i32, i32, i32 }*, { i32, i32, i32 }** %c1
  %fieldadr6 = getelementptr { i32, i32, i32 }, { i32, i32, i32 }* %c5, i32
    0, i32 2
  %file_number = load i32, i32* %fieldadr6
  call void @gl_endRendering(i32 %width, i32 %height, i32 %file_number)
  ret void
}
```

9.5.2 dragon.ll

dragon.sos

```
import renderer.sos
import vector.sos
import transform.sos
import array.sos
import math.sos

// Creates a dragon curve of depth n
dragon: (n: int) -> path =
    if n == 0 // Base case
    then [point{0.0, 0.0}, point{1.0, 0.0}]
    else
    // Create two copies of the previous depths
```

```
    d1: path = dragon(n-1) ;
    d2: path = copy_path(d1) ;

    // Position d1
    s: float = sqrt(2.0)/2.0 ;
    rotate(d1, toradians(45.0), -1, {0.0, 0.0}) ;
    scale(d1, s, s) ;

    // Position d2
    rotate(d2, toradians(135.0), -1, {0., 0.}) ;
    scale(d2,s,s) ;
    trans(d2, {1., 0.}) ;
    reverse(d2) ;

    // Merge the paths
    r: path = append(d1, d2, 1.0) ;
    free_path(d1); free_path(d2); r

// Creates a rainbow color effect
rainbow: (r: int, len: int) -> color =
    h: float = (1.0*r)/len ;
    hsv(h, 0.8, 0.8)

// Render a 400px by 400px canvas, name the image pic0
my_canvas: canvas = {400, 400, 0}

// Start render
startCanvas(my_canvas)
d: path = dragon(7)
// Position the curve (0.4, 0.2 is approximately the center of mass of the
//curve for large n)
trans(d, {-0.4, -0.2})

// Draw it
drawPath(d, rainbow(ints(d.length), d.length), 0)
endCanvas(my_canvas)
```

dragon.ll

```
 ModuleID = 'SOS'
source_filename = "SOS"

declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
  %my_canvas = alloca { i32, i32, i32 }*
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
    (i32* getelementptr (i32, i32* null, i32 1) to i64), i64 3) to i32))
  %anon = bitcast i8* %malloccall to { i32, i32, i32 }*
  %fieldaddr = getelementptr { i32, i32, i32 }, { i32, i32, i32
    }* %anon, i32 0, i32 0
  store i32 400, i32* %fieldaddr
  %fieldaddr1 = getelementptr { i32, i32, i32 }, { i32, i32, i32
    }* %anon, i32 0, i32 1
  store i32 400, i32* %fieldaddr1
```

```
  %fieldaddr2 = getelementptr { i32, i32, i32 }, { i32, i32, i32
    }* %anon, i32 0, i32 2
  store i32 0, i32* %fieldaddr2
  store { i32, i32, i32 }* %anon, { i32, i32, i32 }** %my_canvas
  %my_canvas3 = load { i32, i32, i32 }*, { i32, i32, i32 }** %my_canvas
  call void @startCanvas({ i32, i32, i32 }* %my_canvas3)
  %d = alloca { { float, float }**, i32 }*
  %fxn_result = call { { float, float }**, i32 }* @dragon(i32 7)
  store { { float, float }**, i32 }* %fxn_result, { { float,
    float }**, i32 }** %d
  %d4 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %d
  %malloccall5 = tail call i8* @malloc(i32 trunc (i64 mul nuw
    (i64 ptrtoint(float* getelementptr (float, float* null,
    i32 1) to i64), i64 2) to i32))
  %anon6 = bitcast i8* %malloccall5 to { float, float }*
  %fieldaddr7 = getelementptr { float, float }, { float,
    float }* %anon6, i32 0, i32 0
  store float 0xBFD99999A0000000, float* %fieldaddr7
  %fieldaddr8 = getelementptr { float, float }, { float,
    float }* %anon6, i32 0, i32 1
  store float 0xBFC99999A0000000, float* %fieldaddr8
  %lenref = getelementptr { { float, float }**, i32 }, { {
    float, float }**, i32 }* %d4, i32 0, i32 1
  %len = load i32, i32* %lenref
  %dataref = getelementptr { { float, float }**, i32 }, {
    { float, float }**, i32 }* %d4, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref
  %i = alloca i32
  store i32 0, i32* %i
  br label %loop

loop:                                             ; preds = %loop, %entry
  %i9 = load i32, i32* %i
  %elref = getelementptr { float, float }*, { float, float }** %data,
    i32 %i9
  %el = load { float, float }*, { float, float }** %elref
  call void @trans({ float, float }* %el, { float, float }* %anon6)
  %i10 = add i32 %i9, 1
  store i32 %i10, i32* %i
  %i11 = load i32, i32* %i
  %tmp = icmp slt i32 %i11, %len
  br i1 %tmp, label %loop, label %continue

continue:                                         ; preds = %loop
  %d12 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %d
  %d13 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %d
  %lenref14 = getelementptr { { float, float }**, i32 },
    { { float, float }**i32 }* %d13, i32 0, i32 1
  %len15 = load i32, i32* %lenref14
  %fxn_result16 = call { i32*, i32 }* @ints(i32 %len15)
  %d17 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %d
  %lenref18 = getelementptr { { float, float }**, i32 },
    { { float, float }**,i32 }* %d17, i32 0, i32 1
  %len19 = load i32, i32* %lenref18
  %lenref20 = getelementptr { i32*, i32 }, { i32*, i32 }*
    %fxn_result16, i32 0, i32 1
```

```
  %len21 = load i32, i32* %lenref20
  %dataref22 = getelementptr { i32*, i32 }, { i32*, i32 }*
    %fxn_result16, i32 0, i32 0
  %data23 = load i32*, i32** %dataref22
  %mallocsize = mul i32 %len21, ptrtoint (i1**
    getelementptr (i1*, i1** null,i32 1) to i32)
  %malloccall24 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata = bitcast i8* %malloccall24 to { float, float,
    float, float }**
  %i25 = alloca i32
  store i32 0, i32* %i25
  br label %loop26

loop26:
    ; preds = %loop26, %continue
  %i28 = load i32, i32* %i25
  %elref29 = getelementptr i32, i32* %data23, i32 %i28
  %el30 = load i32, i32* %elref29
  %fxn_result31 = call { float, float, float, float
    }* @rainbow(i32 %el30, i32 %len19)
  %storeref = getelementptr { float, float, float,
    float }*, { float,
    float, float, float }** %arrdata, i32 %i28
  store { float, float, float, float }* %fxn_result31,
    { float, float, float, float }** %storeref
  %i32 = add i32 %i28, 1
  store i32 %i32, i32* %i25
  %i33 = load i32, i32* %i25
  %tmp34 = icmp slt i32 %i33, %len21
  br i1 %tmp34, label %loop26, label %continue27

continue27:                                       ; preds = %loop26
  %malloccall35 = tail call i8* @malloc(i32 ptrtoint
    ({ { float, float, float, float }**, i32 }* getelementptr
    ({ { float, float, float, float }**, i32 }, {
    { float, float, float, float }**, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall35 to { { float, float,
    float, float
    }**, i32 }*
  %arrdata36 = getelementptr { { float, float, float,
    float }**, i32 }, { { float, float, float,
    float }**, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { { float, float, float,float }**, i32 },
    { { float, float, float, float }**, i32 }* %arr, i32 0, i32 1
  store { float, float, float, float }** %arrdata, { float, float,
    float, float }*** %arrdata36
  store i32 %len21, i32* %arrlen
  call void @drawPath({ { float, float }**, i32 }* %d12, { { float,
    float, float, float }**, i32 }* %arr, i32 0)
  %my_canvas37 = load { i32, i32, i32 }*, { i32, i32, i32 }** %my_canvas
  call void @endCanvas({ i32, i32, i32 }* %my_canvas37)
  ret i32 0
}

declare float @sqrtf(float)

declare float @sinf(float)

declare float @cosf(float)
```

```
declare float @tanf(float)

declare float @asinf(float)

declare float @acosf(float)

declare float @atanf(float)

declare float @toradiansf(float)

declare void @gl_startRendering(i32, i32)

declare void @gl_endRendering(i32, i32, i32)

declare void @gl_drawCurve({ float*, i32 }*, { float*, i32 }*, i32)

declare void @gl_drawShape({ float*, i32 }*, { float*, i32 }*, i32, i32)

declare void @gl_drawPoint({ float*, i32 }*, { float*, i32 }*, i32)

define float @floor(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %z = alloca float
  %y = alloca i32
  %x2 = load float, float* %x1
  %cast = fptosi float %x2 to i32
  store i32 %cast, i32* %y
  %cast3 = sitofp i32 %cast to float
  store float %cast3, float* %z
  %z4 = load float, float* %z
  %x5 = load float, float* %x1
  %tmp = fcmp ole float %z4, %x5
  %if_tmp = alloca float
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %if_tmp9 = load float, float* %if_tmp
  ret float %if_tmp9

then:                                             ; preds = %entry
  %z6 = load float, float* %z
  store float %z6, float* %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %z7 = load float, float* %z
  %tmp8 = fsub float %z7, 1.000000e+00
  store float %tmp8, float* %if_tmp
  br label %merge
}

define float @ceil(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
```

```
  %tmp = fneg float %x2
  %fxn_result = call float @floor(float %tmp)
  %tmp3 = fneg float %fxn_result
  ret float %tmp3
}

define float @frac(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %x3 = load float, float* %x1
  %fxn_result = call float @floor(float %x3)
  %tmp = fsub float %x2, %fxn_result
  ret float %tmp
}

define float @max(float %a, float %b) {
entry:
  %a1 = alloca float
  store float %a, float* %a1
  %b2 = alloca float
  store float %b, float* %b2
  %a3 = load float, float* %a1
  %b4 = load float, float* %b2
  %tmp = fcmp olt float %a3, %b4
  %if_tmp = alloca float
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %if_tmp7 = load float, float* %if_tmp
  ret float %if_tmp7

then:                                             ; preds = %entry
  %b5 = load float, float* %b2
  store float %b5, float* %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %a6 = load float, float* %a1
  store float %a6, float* %if_tmp
  br label %merge
}

define float @min(float %a, float %b) {
entry:
  %a1 = alloca float
  store float %a, float* %a1
  %b2 = alloca float
  store float %b, float* %b2
  %a3 = load float, float* %a1
  %b4 = load float, float* %b2
  %tmp = fcmp olt float %a3, %b4
  %if_tmp = alloca float
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %if_tmp7 = load float, float* %if_tmp
  ret float %if_tmp7
```

```
then:                                             ; preds = %entry
  %a5 = load float, float* %a1
  store float %a5, float* %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %b6 = load float, float* %b2
  store float %b6, float* %if_tmp
  br label %merge
}

define float @clamp(float %x, float %m, float %M) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %m2 = alloca float
  store float %m, float* %m2
  %M3 = alloca float
  store float %M, float* %M3
  %M4 = load float, float* %M3
  %x5 = load float, float* %x1
  %m6 = load float, float* %m2
  %fxn_result = call float @max(float %x5, float %m6)
  %fxn_result7 = call float @min(float %M4, float %fxn_result)
  ret float %fxn_result7
}

define float @abs(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %tmp = fcmp olt float %x2, 0.000000e+00
  %if_tmp = alloca float
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %if_tmp6 = load float, float* %if_tmp
  ret float %if_tmp6

then:                                             ; preds = %entry
  %x3 = load float, float* %x1
  %tmp4 = fneg float %x3
  store float %tmp4, float* %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %x5 = load float, float* %x1
  store float %x5, float* %if_tmp
  br label %merge
}

define float @modf(float %x, float %m) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %m2 = alloca float
  store float %m, float* %m2
```

```
  %m3 = load float, float* %m2
  %x4 = load float, float* %x1
  %m5 = load float, float* %m2
  %tmp = fdiv float %x4, %m5
  %fxn_result = call float @frac(float %tmp)
  %tmp6 = fmul float %m3, %fxn_result
  ret float %tmp6
}

define float @sin(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @sinf(float %x2)
  ret float %fxn_result
}

define float @cos(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @cosf(float %x2)
  ret float %fxn_result
}

define float @tan(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @tanf(float %x2)
  ret float %fxn_result
}

define float @asin(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @asinf(float %x2)
  ret float %fxn_result
}

define float @acos(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @acosf(float %x2)
  ret float %fxn_result
}

define float @atan(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
```

```
  %fxn_result = call float @atanf(float %x2)
  ret float %fxn_result
}

define float @sqrt(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @sqrtf(float %x2)
  ret float %fxn_result
}

define float @toradians(float %x) {
entry:
  %x1 = alloca float
  store float %x, float* %x1
  %x2 = load float, float* %x1
  %fxn_result = call float @toradiansf(float %x2)
  ret float %fxn_result
}

define float @sqrMagnitude({ float, float }* %p) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %p2 = load { float, float }*, { float, float }** %p1
  %p3 = load { float, float }*, { float, float }** %p1
  %result = call float @__dotf2({ float, float }* %p2, { float, float }* %p3)
  ret float %result
}

define float @__dotf2({ float, float }* %a, { float, float }* %b) {
entry:
  %a1 = alloca { float, float }*
  store { float, float }* %a, { float, float }** %a1
  %a2 = load { float, float }*, { float, float }** %a1
  %b3 = alloca { float, float }*
  store { float, float }* %b, { float, float }** %b3
  %b4 = load { float, float }*, { float, float }** %b3
  %dot = alloca float
  %tmp = alloca float
  store float 0.000000e+00, float* %dot
  %avalref = getelementptr { float, float }, { float, float }*
    %a2, i32 0, i32 0
  %aval = load float, float* %avalref
  %bvalref = getelementptr { float, float }, { float, float }*
    %b4, i32 0, i32 0
  %bval = load float, float* %bvalref
  %tmp5 = fmul float %aval, %bval
  store float %tmp5, float* %tmp
  %tmp6 = load float, float* %tmp
  %res = load float, float* %dot
  %tmp7 = fadd float %tmp6, %res
  store float %tmp7, float* %dot
  %avalref8 = getelementptr { float, float }, { float, float }*
    %a2, i32 0, i32 1
  %aval9 = load float, float* %avalref8
  %bvalref10 = getelementptr { float, float }, { float, float }*
```

```
    %b4, i32 0, i32 1
  %bval11 = load float, float* %bvalref10
  %tmp12 = fmul float %aval9, %bval11
  store float %tmp12, float* %tmp
  %tmp13 = load float, float* %tmp
  %res14 = load float, float* %dot
  %tmp15 = fadd float %tmp13, %res14
  store float %tmp15, float* %dot
  %res16 = load float, float* %dot
  ret float %res16
}

define float @magnitude({ float, float }* %p) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %p2 = load { float, float }*, { float, float }** %p1
  %fxn_result = call float @sqrMagnitude({ float, float }* %p2)
  %fxn_result3 = call float @sqrt(float %fxn_result)
  ret float %fxn_result3
}

define float @sqrDistance({ float, float }* %a, { float,
    float }* %b) {
entry:
  %a1 = alloca { float, float }*
  store { float, float }* %a, { float, float }** %a1
  %b2 = alloca { float, float }*
  store { float, float }* %b, { float, float }** %b2
  %p = alloca { float, float }*
  %a3 = load { float, float }*, { float, float }** %a1
  %b4 = load { float, float }*, { float, float }** %b2
  %result = call { float, float }* @__subf2({ float, float }*
    %a3, { float, float }* %b4)
  store { float, float }* %result, { float, float }** %p
  %d = alloca float
  %p5 = load { float, float }*, { float, float }** %p
  %fxn_result = call float @sqrMagnitude({ float, float }* %p5)
  store float %fxn_result, float* %d
  %p6 = load { float, float }*, { float, float }** %p
  %0 = bitcast { float, float }* %p6 to i8*
  tail call void @free(i8* %0)
  %d7 = load float, float* %d
  ret float %d7
}

define { float, float }* @__subf2({ float, float }* %a, { float, float }* %b) {
entry:
  %a1 = alloca { float, float }*
  store { float, float }* %a, { float, float }** %a1
  %a2 = load { float, float }*, { float, float }** %a1
  %b3 = alloca { float, float }*
  store { float, float }* %b, { float, float }** %b3
  %b4 = load { float, float }*, { float, float }** %b3
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64
    ptrtoint (float* getelementptr (float, float* null, i32 1) to
    i64), i64 2) to i32))
  %ret = bitcast i8* %malloccall to { float, float }*
  %avalref = getelementptr { float, float }, { float, float }*
```

```
    %a2, i32 0, i32 0
  %aval = load float, float* %avalref
  %bvalref = getelementptr { float, float }, { float, float }*
    %b4, i32 0, i32 0
  %bval = load float, float* %bvalref
  %tmp = fsub float %aval, %bval
  %ref = getelementptr { float, float }, { float, float }* %ret,
    i32 0, i32 0
  store float %tmp, float* %ref
  %avalref5 = getelementptr { float, float }, { float, float }*
    %a2, i32 0, i32 1
  %aval6 = load float, float* %avalref5
  %bvalref7 = getelementptr { float, float }, { float, float }*
    %b4, i32 0, i32 1
  %bval8 = load float, float* %bvalref7
  %tmp9 = fsub float %aval6, %bval8
  %ref10 = getelementptr { float, float }, { float, float }* %ret,
    i32 0, i32 1
  store float %tmp9, float* %ref10
  ret { float, float }* %ret
}

declare noalias i8* @malloc(i32)

declare void @free(i8*)

define float @distance({ float, float }* %a, { float, float }* %b) {
entry:
  %a1 = alloca { float, float }*
  store { float, float }* %a, { float, float }** %a1
  %b2 = alloca { float, float }*
  store { float, float }* %b, { float, float }** %b2
  %a3 = load { float, float }*, { float, float }** %a1
  %b4 = load { float, float }*, { float, float }** %b2
  %fxn_result = call float @sqrDistance({ float, float }* %a3,
    { float, float }* %b4)
  %fxn_result5 = call float @sqrt(float %fxn_result)
  ret float %fxn_result5
}

define { float, float }* @copy_point({ float, float }* %p) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %p2 = load { float, float }*, { float, float }** %p1
  %copied = call { float, float }* @__copy2({ float, float }* %p2)
  ret { float, float }* %copied
}

define { float, float }* @__copy2({ float, float }* %to_copy) {
entry:
  %to_copy1 = alloca { float, float }*
  store { float, float }* %to_copy, { float, float }** %to_copy1
  %to_copy2 = load { float, float }*, { float, float }** %to_copy1
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64
    ptrtoint (float* getelementptr (float, float* null, i32 1) to
    i64), i64 2) to i32))
  %struct = bitcast i8* %malloccall to { float, float }*
  %flref = getelementptr { float, float }, { float, float }* %to_copy2,
```

```
    i32 0, i32 0
  %fl = load float, float* %flref
  %ref = getelementptr { float, float }, { float, float }*
    %struct, i32 0, i32 0
  store float %fl, float* %ref
  %flref3 = getelementptr { float, float }, { float, float }*
    %to_copy2, i32 0, i32 1
  %fl4 = load float, float* %flref3
  %ref5 = getelementptr { float, float }, { float, float }*
    %struct, i32 0, i32 1
  store float %fl4, float* %ref5
  ret { float, float }* %struct
}

define void @free_point({ float, float }* %p) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %p2 = load { float, float }*, { float, float }** %p1
  %0 = bitcast { float, float }* %p2 to i8*
  tail call void @free(i8* %0)
  ret void
}

define { { float, float }**, i32 }* @copy_path({ { float,
    float }**, i32 }* %p) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p, { { float, float }**, i32 }** %p1
  %p2 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %p2, i32 0, i32 1
  %len = load i32, i32* %lenref
  %dataref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %p2, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref
  %mallocsize = mul i32 %len, ptrtoint (i1** getelementptr (i1*, i1**
    null, i32 1) to i32)
  %malloccall = tail call i8* @malloc(i32 %mallocsize)
  %arrdata = bitcast i8* %malloccall to { float, float }**
  %i = alloca i32
  store i32 0, i32* %i
  br label %loop

loop:                                             ; preds = %loop, %entry
  %i3 = load i32, i32* %i
  %elref = getelementptr { float, float }*, { float,
    float }** %data, i32 %i3
  %el = load { float, float }*, { float, float }** %elref
  %fxn_result = call { float, float }* @copy_point({ float,
    float }* %el)
  %storeref = getelementptr { float, float }*, { float, float
    }** %arrdata, i32 %i3
  store { float, float }* %fxn_result, { float, float }** %storeref
  %i4 = add i32 %i3, 1
  store i32 %i4, i32* %i
  %i5 = load i32, i32* %i
  %tmp = icmp slt i32 %i5, %len
  br i1 %tmp, label %loop, label %continue
```

```
continue:                                         ; preds = %loop
  %malloccall6 = tail call i8* @malloc(i32 ptrtoint ({ {
    float, float }**,
    i32 }* getelementptr ({ { float, float }**, i32 }, { { float, float }**,
    i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall6 to { { float, float }**, i32 }*
  %arrdata7 = getelementptr { { float, float }**, i32 },
    { { float, float }**,i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { { float, float }**, i32 },
    { { float, float }**, i32 }* %arr, i32 0, i32 1
  store { float, float }** %arrdata, { float,
    float }*** %arrdata7
  store i32 %len, i32* %arrlen
  ret { { float, float }**, i32 }* %arr
}

define void @free_path({ { float, float }**, i32 }* %p) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p, { { float,
    float }**, i32 }** %p1
  %p2 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %p1
  %lenref = getelementptr { { float, float }**, i32 },
    { { float, float }**,
    i32 }* %p2, i32 0, i32 1
  %len = load i32, i32* %lenref
  %dataref = getelementptr { { float, float }**, i32 },
    { { float, float }**,
    i32 }* %p2, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref
  %i = alloca i32
  store i32 0, i32* %i
  br label %loop

loop:                                             ; preds = %loop, %entry
  %i3 = load i32, i32* %i
  %elref = getelementptr { float, float }*, { float,
    float }** %data, i32 %i3
  %el = load { float, float }*, { float, float }** %elref
  call void @free_point({ float, float }* %el)
  %i4 = add i32 %i3, 1
  store i32 %i4, i32* %i
  %i5 = load i32, i32* %i
  %tmp = icmp slt i32 %i5, %len
  br i1 %tmp, label %loop, label %continue

continue:                                         ; preds = %loop
  ret void
}

define void @appendhelp_copyin({ { float, float }**, i32 }* %in, { { float,
    float }**, i32 }* %from, i32 %i) {
entry:
  %in1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %in, { { float, float }**, i32 }** %in1
  %from2 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %from, { { float, float }**, i32 }** %from2
```

```
  %i3 = alloca i32
  store i32 %i, i32* %i3
  %i4 = load i32, i32* %i3
  %in5 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %in1
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %in5, i32 0, i32 1
  %len = load i32, i32* %lenref
  %tmp = icmp slt i32 %i4, %len
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  ret void

then:                                             ; preds = %entry
  %in6 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %in1
  %datarefref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %in6, i32 0, i32 0
  %dataref = load { float, float }**, { float, float }*** %datarefref
  %i7 = load i32, i32* %i3
  %from8 = load { { float, float }**, i32 }*, { { float, float }**, i32
    }** %from2
  %i9 = load i32, i32* %i3
  %tmp10 = add i32 %i9, 1
  %dataref11 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %from8, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref11
  %elref = getelementptr { float, float }*, { float, float }** %data, i32 %tmp10
  %el = load { float, float }*, { float, float }** %elref
  %copied = call { float, float }* @__copy2.1({ float, float }* %el)
  %storeref = getelementptr { float, float }*, { float, float }** %dataref,
    i32 %i7
  store { float, float }* %copied, { float, float }** %storeref
  %in12 = load { { float, float }**, i32 }*, { { float, float }**, i32 }**
    %in1
  %from13 = load { { float, float }**, i32 }*, { { float, float }**, i32 }**
    %from2
  %i14 = load i32, i32* %i3
  %tmp15 = add i32 %i14, 1
  call void @appendhelp_copyin({ { float, float }**, i32 }* %in12, {
    { float, float }**, i32 }* %from13, i32 %tmp15)
  br label %merge

else:                                             ; preds = %entry
  br label %merge
}

define { float, float }* @__copy2.1({ float, float }* %to_copy) {
entry:
  %to_copy1 = alloca { float, float }*
  store { float, float }* %to_copy, { float, float }** %to_copy1
  %to_copy2 = load { float, float }*, { float, float }** %to_copy1
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
    (float* getelementptr (float, float* null, i32 1) to i64), i64 2) to i32))
  %struct = bitcast i8* %malloccall to { float, float }*
  %flref = getelementptr { float, float }, { float, float }* %to_copy2, i32 0, i32
    0
  %fl = load float, float* %flref
  %ref = getelementptr { float, float }, { float, float }* %struct, i32 0, i32 0
  store float %fl, float* %ref
```

```
  %flref3 = getelementptr { float, float }, { float, float }* %to_copy2, i32 0,
    i32 1
  %fl4 = load float, float* %flref3
  %ref5 = getelementptr { float, float }, { float, float }* %struct, i32 0, i32 1
  store float %fl4, float* %ref5
  ret { float, float }* %struct
}

define { { float, float }**, i32 }* @appendhelp_tail({ { float, float }**, i32
    }* %p) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p, { { float, float }**, i32 }** %p1
  %tail = alloca { { float, float }**, i32 }*
  %p2 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32}* %p2, i32 0, i32 1
  %len = load i32, i32* %lenref
  %tmp = sub i32 %len, 1
  %malloccall = tail call i8* @malloc(i32 ptrtoint (i1** getelementptr
    (i1*, i1** null, i32 1) to i32))
  %arrdata = bitcast i8* %malloccall to { float, float }**
  %malloccall3 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
    (float* getelementptr (float, float* null, i32 1) to i64), i64 2)to i32))
  %anon = bitcast i8* %malloccall3 to { float, float }*
  %fieldaddr = getelementptr { float, float }, { float, float }* %anon,
    i32 0, i32 0
  store float 0.000000e+00, float* %fieldaddr
  %fieldaddr4 = getelementptr { float, float }, { float, float }* %anon,
    i32 0, i32 1
  store float 0.000000e+00, float* %fieldaddr4
  %storeref = getelementptr { float, float }*, { float, float }** %arrdata, i32 0
  store { float, float }* %anon, { float, float }** %storeref
  %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ { float, float
    }**, i32 }* getelementptr ({ { float, float }**, i32 }, { { float,
    float }**, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall5 to { { float, float }**, i32 }*
  %arrdata6 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr, i32 0, i32 1
  store { float, float }** %arrdata, { float, float }*** %arrdata6
  store i32 1, i32* %arrlen
  %lenref7 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr, i32 0, i32 1
  %len8 = load i32, i32* %lenref7
  %oflen = mul i32 %tmp, %len8
  %olddataref = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr, i32 0, i32 0
  %olddata = load { float, float }**, { float, float }*** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (i1** getelementptr (i1*,
    i1** null, i32 1) to i32)
  %malloccall9 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata10 = bitcast i8* %malloccall9 to { float, float }**
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner
```

```
loop:                                             ; preds = %inner
  %i18 = load i32, i32* %i
  store i32 0, i32* %j
  %tmp19 = icmp slt i32 %i18, %oflen
  br i1 %tmp19, label %inner, label %continue

inner:
    ; preds = %loop, %inner, %entry
  %i11 = load i32, i32* %j
  %i12 = load i32, i32* %i
  %elref = getelementptr { float, float }*, { float, float }** %olddata,
    i32 %i11
  %el = load { float, float }*, { float, float }** %elref
  %storeref13 = getelementptr { float, float }*, { float, float }**
    %arrdata10, i32 %i12
  store { float, float }* %el, { float, float }** %storeref13
  %i14 = add i32 %i12, 1
  store i32 %i14, i32* %i
  %j15 = add i32 %i11, 1
  store i32 %j15, i32* %j
  %j16 = load i32, i32* %j
  %tmp17 = icmp slt i32 %j16, %len8
  br i1 %tmp17, label %inner, label %loop

continue:                                         ; preds = %loop
  %malloccall20 = tail call i8* @malloc(i32 ptrtoint ({ { float,
    float }**, i32 }* getelementptr ({ { float, float }**, i32 },
    { { float, float }**, i32 }* null, i32 1) to i32))
  %arr21 = bitcast i8* %malloccall20 to { { float, float }**, i32 }*
  %arrdata22 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr21, i32 0, i32 0
  %arrlen23 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr21, i32 0, i32 1
  store { float, float }** %arrdata10, { float, float }*** %arrdata22
  store i32 %oflen, i32* %arrlen23
  store { { float, float }**, i32 }* %arr21, { { float, float }**,
    i32 }** %tail
  %tail24 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %tail
  %p25 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %p1
  call void @appendhelp_copyin({ { float, float }**, i32 }* %tail24,
    { { float, float }**, i32 }* %p25, i32 0)
  %tail26 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %tail
  ret { { float, float }**, i32 }* %tail26
}

define { { float, float }**, i32 }* @append({ { float, float }**, i32
    }* %p1, { { float, float }**, i32 }* %p2, float %epsilon) {
entry:
  %p11 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p1, { { float, float }**, i32
    }** %p11
  %p22 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p2, { { float, float }**, i32
    }** %p22
  %epsilon3 = alloca float
```

```
  store float %epsilon, float* %epsilon3
  %p14 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %p11
  %lenref = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %p14, i32 0, i32 1
  %len = load i32, i32* %lenref
  %tmp = icmp eq i32 %len, 0
  %if_tmp = alloca { { float, float }**, i32 }*
  br i1 %tmp, label %then, label %else

merge:
    ; preds = %merge11, %then
  %if_tmp66 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %if_tmp
  ret { { float, float }**, i32 }* %if_tmp66

then:
    ; preds = %entry
  %p25 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %p22
  %fxn_result = call { { float, float }**, i32 }* @copy_path({
    { float, float }**, i32 }* %p25)
  store { { float, float }**, i32 }* %fxn_result, { { float,
    float }**, i32 }** %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %p26 = load { { float, float }**, i32 }*, { { float, float
    }**, i32 }** %p22
  %lenref7 = getelementptr { { float, float }**, i32 }, { {
    float, float }**, i32 }* %p26, i32 0, i32 1
  %len8 = load i32, i32* %lenref7
  %tmp9 = icmp eq i32 %len8, 0
  %if_tmp10 = alloca { { float, float }**, i32 }*
  br i1 %tmp9, label %then12, label %else13

merge11:
    ; preds = %contb, %then12
  %if_tmp65 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %if_tmp10
  store { { float, float }**, i32 }* %if_tmp65, { { float,
    float }**, i32 }** %if_tmp
  br label %merge

then12:                                           ; preds = %else
  %p114 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %p11
  %fxn_result15 = call { { float, float }**, i32 }* @copy_path({
    { float, float }**, i32 }* %p114)
  store { { float, float }**, i32 }* %fxn_result15, { { float,
    float }**, i32 }** %if_tmp10
  br label %merge11

else13:                                           ; preds = %else
  %merge16 = alloca i1
  %p117 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %p11
  %p118 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %p11
```

```
  %lenref19 = getelementptr { { float, float }**, i32 }, { {
    float, float }**, i32 }* %p118, i32 0, i32 1
  %len20 = load i32, i32* %lenref19
  %tmp21 = sub i32 %len20, 1
  %dataref = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %p117, i32 0, i32 0
  %data = load { float, float }**, { float, float }***
    %dataref
  %elref = getelementptr { float, float }*, { float, float
    }** %data, i32 %tmp21
  %el = load { float, float }*, { float, float }** %elref
  %p222 = load { { float, float }**, i32 }*, { { float,
    float }**, i32 }** %p22
  %dataref23 = getelementptr { { float, float }**, i32 },
    { { float,float }**, i32 }* %p222, i32 0, i32 0
  %data24 = load { float, float }**, { float, float
    }*** %dataref23
  %elref25 = getelementptr { float, float }*, { float, float }** %data24, i32 0
  %el26 = load { float, float }*, { float, float }** %elref25
  %fxn_result27 = call float @sqrDistance({ float,
    float }* %el, { float, float }* %el26)
  %epsilon28 = load float, float* %epsilon3
  %epsilon29 = load float, float* %epsilon3
  %tmp30 = fmul float %epsilon28, %epsilon29
  %tmp31 = fcmp olt float %fxn_result27, %tmp30
  store i1 %tmp31, i1* %merge16
  %p2c = alloca { { float, float }**, i32 }*
  %merge32 = load i1, i1* %merge16
  %if_tmp33 = alloca { { float, float }**, i32 }*
  br i1 %merge32, label %then35, label %else36

merge34:
    ; preds = %else36, %then35
  %if_tmp40 = load { { float, float }**, i32 }*,
    { { float, float }**,i32 }** %if_tmp33
  store { { float, float }**, i32 }* %if_tmp40, {
    { float, float }**,
    i32 }** %p2c
  %ret = alloca { { float, float }**, i32 }*
  %p141 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p11
  %fxn_result42 = call { { float, float }**, i32 }* @copy_path({ { float,
    float }**, i32 }* %p141)
  %p2c43 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p2c
  %fxn_result44 = call { { float, float }**, i32 }* @copy_path({ { float,
    float }**, i32 }* %p2c43)
  %len1ref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %fxn_result42, i32 0, i32 1
  %len1 = load i32, i32* %len1ref
  %len2ref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %fxn_result44, i32 0, i32 1
  %len2 = load i32, i32* %len2ref
  %n = add i32 %len1, %len2
  %data1ref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %fxn_result42, i32 0, i32 0
  %data1 = load { float, float }**, { float, float }*** %data1ref
  %data2ref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %fxn_result44, i32 0, i32 0
  %data2 = load { float, float }**, { float, float }*** %data2ref
  %mallocsize = mul i32 %n, ptrtoint (i1** getelementptr (i1*, i1** null,
```

```
    i32 1) to i32)
  %malloccall = tail call i8* @malloc(i32 %mallocsize)
  %data45 = bitcast i8* %malloccall to { float, float }**
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %loop1

then35:                                           ; preds = %else13
  %p237 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p22
  %fxn_result38 = call { { float, float }**, i32 }* @appendhelp_tail({
    { float, float }**, i32 }* %p237)
  store { { float, float }**, i32 }* %fxn_result38, { { float, float
    }**, i32 }** %if_tmp33
  br label %merge34

else36:                                           ; preds = %else13
  %p239 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %p22
  store { { float, float }**, i32 }* %p239, { { float, float }**,
    i32 }** %if_tmp33
  br label %merge34

loop1:                                            ; preds = %loop1, %merge34
  %i46 = load i32, i32* %j
  %i47 = load i32, i32* %i
  %elref48 = getelementptr { float, float }*, { float, float }**
    %data1, i32 %i46
  %el49 = load { float, float }*, { float, float }** %elref48
  %storeref = getelementptr { float, float }*, { float, float }**
    %data45, i32 %i47
  store { float, float }* %el49, { float, float }** %storeref
  %tmp50 = add i32 %i47, 1
  store i32 %tmp50, i32* %i
  %j51 = add i32 %i46, 1
  store i32 %j51, i32* %j
  %j52 = load i32, i32* %j
  %tmp53 = icmp slt i32 %j52, %len1
  br i1 %tmp53, label %loop1, label %inbtw

inbtw:                                            ; preds = %loop1
  store i32 0, i32* %j
  br label %loop2

loop2:                                            ; preds = %loop2, %inbtw
  %i54 = load i32, i32* %j
  %i55 = load i32, i32* %i
  %elref56 = getelementptr { float, float }*, { float, float }**
    %data2, i32 %i54
  %el57 = load { float, float }*, { float, float }** %elref56
  %storeref58 = getelementptr { float, float }*, { float, float }**
    %data45, i32 %i55
  store { float, float }* %el57, { float, float }** %storeref58
  %tmp59 = add i32 %i55, 1
  store i32 %tmp59, i32* %i
  %j60 = add i32 %i54, 1
  store i32 %j60, i32* %j
  %j61 = load i32, i32* %j
```

```
  %tmp62 = icmp slt i32 %j61, %len2
  br i1 %tmp62, label %loop2, label %contb

contb:                                            ; preds = %loop2
  %malloccall63 = tail call i8* @malloc(i32 ptrtoint ({ { float, float
    }**, i32 }* getelementptr ({ { float, float }**, i32 }, { {
    float, float }**, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall63 to { { float, float }**, i32 }*
  %arrdata = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr, i32 0, i32 1
  store { float, float }** %data45, { float, float }*** %arrdata
  store i32 %n, i32* %arrlen
  store { { float, float }**, i32 }* %arr, { { float, float }**,
    i32 }** %ret
  %ret64 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %ret
  store { { float, float }**, i32 }* %ret64, { { float, float }**,
    i32 }** %if_tmp10
  br label %merge11
}

define void @reversedhelp({ { float, float }**, i32 }* %in, { {
    float, float }**, i32 }* %from, i32 %i) {
entry:
  %in1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %in, { { float, float }**,
    i32 }** %in1
  %from2 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %from, { { float, float }**, i32 }** %from2
  %i3 = alloca i32
  store i32 %i, i32* %i3
  %i4 = load i32, i32* %i3
  %in5 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %in1
  %lenref = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %in5, i32 0, i32 1
  %len = load i32, i32* %lenref
  %tmp = icmp slt i32 %i4, %len
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  ret void

then:                                             ; preds = %entry
  %in6 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %in1
  %i7 = load i32, i32* %i3
  %dataref = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %in6, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref
  %elref = getelementptr { float, float }*, { float, float }** %data, i32 %i7
  %el = load { float, float }*, { float, float }** %elref
  %from8 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %from2
  %in9 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %in1
  %lenref10 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %in9, i32 0, i32 1
  %len11 = load i32, i32* %lenref10
```

```
  %tmp12 = sub i32 %len11, 1
  %i13 = load i32, i32* %i3
  %tmp14 = sub i32 %tmp12, %i13
  %dataref15 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %from8, i32 0, i32 0
  %data16 = load { float, float }**, { float, float }*** %dataref15
  %elref17 = getelementptr { float, float }*, { float, float }** %data16, i32
    %tmp14
  %el18 = load { float, float }*, { float, float }** %elref17
  %fieldadr = getelementptr { float, float }, { float, float }* %el18, i32 0, i32
    0
  %x = load float, float* %fieldadr
  %ref = getelementptr { float, float }, { float, float }* %el, i32 0, i32 0
  store float %x, float* %ref
  %in19 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %in1
  %i20 = load i32, i32* %i3
  %dataref21 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %in19, i32 0, i32 0
  %data22 = load { float, float }**, { float, float }*** %dataref21
  %elref23 = getelementptr { float, float }*, { float, float }** %data22, i32 %i20
  %el24 = load { float, float }*, { float, float }** %elref23
  %from25 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %from2
  %in26 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %in1
  %lenref27 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %in26, i32 0, i32 1
  %len28 = load i32, i32* %lenref27
  %tmp29 = sub i32 %len28, 1
  %i30 = load i32, i32* %i3
  %tmp31 = sub i32 %tmp29, %i30
  %dataref32 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %from25, i32 0, i32 0
  %data33 = load { float, float }**, { float, float }*** %dataref32
  %elref34 = getelementptr { float, float }*, { float, float }**
    %data33, i32 %tmp31
  %el35 = load { float, float }*, { float, float }** %elref34
  %fieldadr36 = getelementptr { float, float }, { float, float }*
    %el35, i32 0, i32 1
  %y = load float, float* %fieldadr36
  %ref37 = getelementptr { float, float }, { float, float }* %el24,
    i32 0, i32 1
  store float %y, float* %ref37
  %in38 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %in1
  %from39 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %from2
  %i40 = load i32, i32* %i3
  %tmp41 = add i32 %i40, 1
  call void @reversedhelp({ { float, float }**, i32 }* %in38, { {
    float, float }**, i32 }* %from39, i32 %tmp41)
  br label %merge

else:                                             ; preds = %entry
  br label %merge
}

define { { float, float }**, i32 }* @reversed({ { float, float }**, i32 }* %p) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
```

```
  store { { float, float }**, i32 }* %p, { { float, float }**, i32 }** %p1
  %newpath = alloca { { float, float }**, i32 }*
  %p2 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %lenref = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %p2, i32 0, i32 1
  %len = load i32, i32* %lenref
  %malloccall = tail call i8* @malloc(i32 ptrtoint (i1** getelementptr
    (i1*, i1** null, i32 1) to i32))
  %arrdata = bitcast i8* %malloccall to { float, float }**
  %malloccall3 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64
    ptrtoint (float* getelementptr (float, float* null, i32 1) to i64), i64 2)
    to i32))
  %anon = bitcast i8* %malloccall3 to { float, float }*
  %fieldaddr = getelementptr { float, float }, { float, float }* %anon, i32 0,
    i32 0
  store float 0.000000e+00, float* %fieldaddr
  %fieldaddr4 = getelementptr { float, float }, { float, float }* %anon, i32 0,
    i32 1
  store float 0.000000e+00, float* %fieldaddr4
  %storeref = getelementptr { float, float }*, { float, float }** %arrdata, i32
    0
  store { float, float }* %anon, { float, float }** %storeref
  %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ { float,
    float }**, i32 }* getelementptr ({ { float, float }**, i32 },
    { { float, float }**, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall5 to { { float, float }**, i32 }*
  %arrdata6 = getelementptr { { float, float }**, i32 }, { { float,
    float }**, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { { float, float }**, i32 }, { {
    float, float }**, i32 }* %arr, i32 0, i32 1
  store { float, float }** %arrdata, { float, float }*** %arrdata6
  store i32 1, i32* %arrlen
  %lenref7 = getelementptr { { float, float }**, i32 }, { {
    float, float }**, i32 }* %arr, i32 0, i32 1
  %len8 = load i32, i32* %lenref7
  %oflen = mul i32 %len, %len8
  %olddataref = getelementptr { { float, float }**, i32 }, { {
    float, float }**, i32 }* %arr, i32 0, i32 0
  %olddata = load { float, float }**, { float, float }*** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (i1** getelementptr (i1*,
    i1** null, i32 1) to i32)
  %malloccall9 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata10 = bitcast i8* %malloccall9 to { float, float }**
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner

loop:                                             ; preds = %inner
  %i17 = load i32, i32* %i
  store i32 0, i32* %j
  %tmp18 = icmp slt i32 %i17, %oflen
  br i1 %tmp18, label %inner, label %continue

inner:                                            ; preds = %loop, %inner, %entry
  %i11 = load i32, i32* %j
  %i12 = load i32, i32* %i
  %elref = getelementptr { float, float }*, { float, float }** %olddata, i32 %i11
```

```
  %el = load { float, float }*, { float, float }** %elref
  %storeref13 = getelementptr { float, float }*, { float, float }** %arrdata10,
    i32 %i12
  store { float, float }* %el, { float, float }** %storeref13
  %i14 = add i32 %i12, 1
  store i32 %i14, i32* %i
  %j15 = add i32 %i11, 1
  store i32 %j15, i32* %j
  %j16 = load i32, i32* %j
  %tmp = icmp slt i32 %j16, %len8
  br i1 %tmp, label %inner, label %loop

continue:                                         ; preds = %loop
  %malloccall19 = tail call i8* @malloc(i32 ptrtoint ({ { float, float }**,
    i32 }* getelementptr ({ { float, float }**, i32 }, { { float, float }**,
    i32 }* null, i32 1) to i32))
  %arr20 = bitcast i8* %malloccall19 to { { float, float }**, i32 }*
  %arrdata21 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %arr20, i32 0, i32 0
  %arrlen22 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %arr20, i32 0, i32 1
  store { float, float }** %arrdata10, { float, float }*** %arrdata21
  store i32 %oflen, i32* %arrlen22
  store { { float, float }**, i32 }* %arr20, { { float, float }**, i32 }**
    %newpath
  %newpath23 = load { { float, float }**, i32 }*, { { float, float }**, i32 }**
    %newpath
  %p24 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  call void @reversedhelp({ { float, float }**, i32 }* %newpath23, { { float,
    float }**, i32 }* %p24, i32 0)
  %newpath25 = load { { float, float }**, i32 }*, { { float, float }**, i32 }**
    %newpath
  ret { { float, float }**, i32 }* %newpath25
}

define void @reversehelp({ { float, float }**, i32 }* %p, i32 %i) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p, { { float, float }**, i32 }** %p1
  %i2 = alloca i32
  store i32 %i, i32* %i2
  %i3 = load i32, i32* %i2
  %p4 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %p4, i32 0, i32 1
  %len = load i32, i32* %lenref
  %tmp = sdiv i32 %len, 2
  %tmp5 = icmp slt i32 %i3, %tmp
  br i1 %tmp5, label %then, label %else

merge:                                            ; preds = %else, %then
  ret void

then:                                             ; preds = %entry
  %q = alloca { float, float }*
  %p6 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %i7 = load i32, i32* %i2
  %dataref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %p6, i32 0, i32 0
```

```
  %data = load { float, float }**, { float, float }*** %dataref
  %elref = getelementptr { float, float }*, { float, float }** %data, i32 %i7
  %el = load { float, float }*, { float, float }** %elref
  store { float, float }* %el, { float, float }** %q
  %p8 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %datarefref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %p8, i32 0, i32 0
  %dataref9 = load { float, float }**, { float, float }*** %datarefref
  %i10 = load i32, i32* %i2
  %p11 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %p12 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %lenref13 = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %p12, i32 0, i32 1
  %len14 = load i32, i32* %lenref13
  %tmp15 = sub i32 %len14, 1
  %i16 = load i32, i32* %i2
  %tmp17 = sub i32 %tmp15, %i16
  %dataref18 = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %p11, i32 0, i32 0
  %data19 = load { float, float }**, { float, float }*** %dataref18
  %elref20 = getelementptr { float, float }*, { float, float }** %data19, i32
    %tmp17
  %el21 = load { float, float }*, { float, float }** %elref20
  %storeref = getelementptr { float, float }*, { float, float }** %dataref9,
    i32 %i10
  store { float, float }* %el21, { float, float }** %storeref
  %p22 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %datarefref23 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %p22, i32 0, i32 0
  %dataref24 = load { float, float }**, { float, float }*** %datarefref23
  %p25 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %lenref26 = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %p25, i32 0, i32 1
  %len27 = load i32, i32* %lenref26
  %tmp28 = sub i32 %len27, 1
  %i29 = load i32, i32* %i2
  %tmp30 = sub i32 %tmp28, %i29
  %q31 = load { float, float }*, { float, float }** %q
  %storeref32 = getelementptr { float, float }*, { float, float }** %dataref24,
    i32 %tmp30
  store { float, float }* %q31, { float, float }** %storeref32
  %p33 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  %i34 = load i32, i32* %i2
  %tmp35 = add i32 %i34, 1
  call void @reversehelp({ { float, float }**, i32 }* %p33, i32 %tmp35)
  br label %merge

else:                                             ; preds = %entry
  br label %merge
}

define void @reverse({ { float, float }**, i32 }* %p) {
entry:
  %p1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %p, { { float, float }**, i32 }** %p1
  %p2 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %p1
  call void @reversehelp({ { float, float }**, i32 }* %p2, i32 0)
  ret void
}
```

```
define { float, float, float, float }* @rgb(float %r, float %g, float %b) {
entry:
  %r1 = alloca float
  store float %r, float* %r1
  %g2 = alloca float
  store float %g, float* %g2
  %b3 = alloca float
  store float %b, float* %b3
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint (float*
    getelementptr (float, float* null, i32 1) to i64), i64 4) to i32))
  %anon = bitcast i8* %malloccall to { float, float, float, float }*
  %fieldaddr = getelementptr { float, float, float, float }, { float, float, float
    ,
    float }* %anon, i32 0, i32 0
  %r4 = load float, float* %r1
  store float %r4, float* %fieldaddr
  %fieldaddr5 = getelementptr { float, float, float, float }, { float, float,
    float,
    float }* %anon, i32 0, i32 1
  %g6 = load float, float* %g2
  store float %g6, float* %fieldaddr5
  %fieldaddr7 = getelementptr { float, float, float, float }, { float, float,
    float,
    float }* %anon, i32 0, i32 2
  %b8 = load float, float* %b3
  store float %b8, float* %fieldaddr7
  %fieldaddr9 = getelementptr { float, float, float, float }, { float, float,
    float,
    float }* %anon, i32 0, i32 3
  store float 1.000000e+00, float* %fieldaddr9
  ret { float, float, float, float }* %anon
}

define { float, float, float, float }* @hsv(float %h, float %s, float %v) {
entry:
  %h1 = alloca float
  store float %h, float* %h1
  %s2 = alloca float
  store float %s, float* %s2
  %v3 = alloca float
  store float %v, float* %v3
  %c = alloca float
  %v4 = load float, float* %v3
  %s5 = load float, float* %s2
  %tmp = fmul float %v4, %s5
  store float %tmp, float* %c
  %hfac = alloca float
  %h6 = load float, float* %h1
  %tmp7 = fmul float %h6, 6.000000e+00
  %fxn_result = call float @modf(float %tmp7, float 2.000000e+00)
  store float %fxn_result, float* %hfac
  %x = alloca float
  %c8 = load float, float* %c
  %hfac9 = load float, float* %hfac
  %tmp10 = fsub float %hfac9, 1.000000e+00
  %fxn_result11 = call float @abs(float %tmp10)
  %tmp12 = fsub float 1.000000e+00, %fxn_result11
  %tmp13 = fmul float %c8, %tmp12
```

```
  store float %tmp13, float* %x
  %m = alloca float
  %v14 = load float, float* %v3
  %c15 = load float, float* %c
  %tmp16 = fsub float %v14, %c15
  store float %tmp16, float* %m
  %hh = alloca float
  %h17 = load float, float* %h1
  %tmp18 = fmul float %h17, 6.000000e+00
  store float %tmp18, float* %hh
  %hh19 = load float, float* %hh
  %tmp20 = fcmp olt float %hh19, 1.000000e+00
  %if_tmp = alloca { float, float, float, float }*
  br i1 %tmp20, label %then, label %else

merge:                                            ; preds = %merge30, %then
  %if_tmp85 = load { float, float, float, float }*, { float, float, float, float
    }** %if_tmp
  ret { float, float, float, float }* %if_tmp85

then:                                             ; preds = %entry
  %v21 = load float, float* %v3
  %x22 = load float, float* %x
  %m23 = load float, float* %m
  %tmp24 = fadd float %x22, %m23
  %m25 = load float, float* %m
  %fxn_result26 = call { float, float, float, float }* @rgb(float %v21, float %
    tmp24
    , float %m25)
  store { float, float, float, float }* %fxn_result26, { float, float, float,
    float
    }** %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %hh27 = load float, float* %hh
  %tmp28 = fcmp olt float %hh27, 2.000000e+00
  %if_tmp29 = alloca { float, float, float, float }*
  br i1 %tmp28, label %then31, label %else32

merge30:                                          ; preds = %merge42, %then31
  %if_tmp84 = load { float, float, float, float }*, { float, float, float, float
    }** %if_tmp29
  store { float, float, float, float }* %if_tmp84, { float, float, float, float
    }** %if_tmp
  br label %merge

then31:                                           ; preds = %else
  %x33 = load float, float* %x
  %m34 = load float, float* %m
  %tmp35 = fadd float %x33, %m34
  %v36 = load float, float* %v3
  %m37 = load float, float* %m
  %fxn_result38 = call { float, float, float, float }* @rgb(float %tmp35, float
    %v36, float %m37)
  store { float, float, float, float }* %fxn_result38, { float, float, float,
    float }** %if_tmp29
  br label %merge30
```

```
else32:                                           ; preds = %else
  %hh39 = load float, float* %hh
  %tmp40 = fcmp olt float %hh39, 3.000000e+00
  %if_tmp41 = alloca { float, float, float, float }*
  br i1 %tmp40, label %then43, label %else44

merge42:                                          ; preds = %merge54, %then43
  %if_tmp83 = load { float, float, float, float }*, { float, float, float,
    float }** %if_tmp41
  store { float, float, float, float }* %if_tmp83, { float, float, float,
    float }** %if_tmp29
  br label %merge30

then43:                                           ; preds = %else32
  %m45 = load float, float* %m
  %v46 = load float, float* %v3
  %x47 = load float, float* %x
  %m48 = load float, float* %m
  %tmp49 = fadd float %x47, %m48
  %fxn_result50 = call { float, float, float, float }* @rgb(float %m45, float
    %v46, float %tmp49)
  store { float, float, float, float }* %fxn_result50, { float, float, float,
    float }** %if_tmp41
  br label %merge42

else44:                                           ; preds = %else32
  %hh51 = load float, float* %hh
  %tmp52 = fcmp olt float %hh51, 4.000000e+00
  %if_tmp53 = alloca { float, float, float, float }*
  br i1 %tmp52, label %then55, label %else56

merge54:                                          ; preds = %merge66, %then55
  %if_tmp82 = load { float, float, float, float }*, { float, float, float,
    float }** %if_tmp53
  store { float, float, float, float }* %if_tmp82, { float, float, float,
    float }** %if_tmp41
  br label %merge42

then55:                                           ; preds = %else44
  %m57 = load float, float* %m
  %x58 = load float, float* %x
  %m59 = load float, float* %m
  %tmp60 = fadd float %x58, %m59
  %v61 = load float, float* %v3
  %fxn_result62 = call { float, float, float, float }* @rgb(float %m57,
    float %tmp60, float %v61)
  store { float, float, float, float }* %fxn_result62, { float, float,
    float, float }** %if_tmp53
  br label %merge54

else56:                                           ; preds = %else44
  %hh63 = load float, float* %hh
  %tmp64 = fcmp olt float %hh63, 5.000000e+00
  %if_tmp65 = alloca { float, float, float, float }*
  br i1 %tmp64, label %then67, label %else68

merge66:                                          ; preds = %else68, %then67
  %if_tmp81 = load { float, float, float, float }*, { float, float,
    float, float }** %if_tmp65
```

```
  store { float, float, float, float }* %if_tmp81, { float, float,
    float, float }** %if_tmp53
  br label %merge54

then67:                                           ; preds = %else56
  %x69 = load float, float* %x
  %m70 = load float, float* %m
  %tmp71 = fadd float %x69, %m70
  %m72 = load float, float* %m
  %v73 = load float, float* %v3
  %fxn_result74 = call { float, float, float, float }* @rgb(float
    %tmp71, float %m72, float %v73)
  store { float, float, float, float }* %fxn_result74, { float,
    float, float, float }** %if_tmp65
  br label %merge66

else68:                                           ; preds = %else56
  %v75 = load float, float* %v3
  %m76 = load float, float* %m
  %x77 = load float, float* %x
  %m78 = load float, float* %m
  %tmp79 = fadd float %x77, %m78
  %fxn_result80 = call { float, float, float, float }* @rgb(float %v75,
    float %m76, float %tmp79)
  store { float, float, float, float }* %fxn_result80, { float, float,
    float, float }** %if_tmp65
  br label %merge66
}

define void @startCanvas({ i32, i32, i32 }* %c) {
entry:
  %c1 = alloca { i32, i32, i32 }*
  store { i32, i32, i32 }* %c, { i32, i32, i32 }** %c1
  %c2 = load { i32, i32, i32 }*, { i32, i32, i32 }** %c1
  %fieldadr = getelementptr { i32, i32, i32 }, { i32, i32, i32 }* %c2,
    i32 0, i32 0
  %width = load i32, i32* %fieldadr
  %c3 = load { i32, i32, i32 }*, { i32, i32, i32 }** %c1
  %fieldadr4 = getelementptr { i32, i32, i32 }, { i32, i32, i32 }* %c3,
    i32 0, i32 1
  %height = load i32, i32* %fieldadr4
  call void @gl_startRendering(i32 %width, i32 %height)
  ret void
}

define void @cvoid() {
entry:
  ret void
}

define void @drawHelper({ { float, float }**, i32 }* %point_structs, { {
    float, float, float, float }**, i32 }* %color_structs, i32 %numOfPoints,
    i32 %i, { float*, i32 }* %points, { float*, i32 }* %colors) {
entry:
  %point_structs1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %point_structs, { { float, float }**, i32
    }** %point_structs1
  %color_structs2 = alloca { { float, float, float, float }**, i32 }*
  store { { float, float, float, float }**, i32 }* %color_structs, { {
```

```
    float, float, float, float }**, i32 }** %color_structs2
  %numOfPoints3 = alloca i32
  store i32 %numOfPoints, i32* %numOfPoints3
  %i4 = alloca i32
  store i32 %i, i32* %i4
  %points5 = alloca { float*, i32 }*
  store { float*, i32 }* %points, { float*, i32 }** %points5
  %colors6 = alloca { float*, i32 }*
  store { float*, i32 }* %colors, { float*, i32 }** %colors6
  %i7 = load i32, i32* %i4
  %numOfPoints8 = load i32, i32* %numOfPoints3
  %tmp = icmp sge i32 %i7, %numOfPoints8
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  ret void

then:                                             ; preds = %entry
  call void @cvoid()
  br label %merge

else:                                             ; preds = %entry
  %px = alloca float
  %point_structs9 = load { { float, float }**, i32 }*, { { float, float
    }**, i32 }** %point_structs1
  %i10 = load i32, i32* %i4
  %dataref = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %point_structs9, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref
  %elref = getelementptr { float, float }*, { float, float }** %data, i32 %i10
  %el = load { float, float }*, { float, float }** %elref
  %fieldadr = getelementptr { float, float }, { float, float }* %el, i32 0, i32 0
  %x = load float, float* %fieldadr
  store float %x, float* %px
  %py = alloca float
  %point_structs11 = load { { float, float }**, i32 }*, { { float, float
    }**, i32 }** %point_structs1
  %i12 = load i32, i32* %i4
  %dataref13 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %point_structs11, i32 0, i32 0
  %data14 = load { float, float }**, { float, float }*** %dataref13
  %elref15 = getelementptr { float, float }*, { float, float }** %data14,
    i32 %i12
  %el16 = load { float, float }*, { float, float }** %elref15
  %fieldadr17 = getelementptr { float, float }, { float, float }* %el16,
    i32 0, i32 1
  %y = load float, float* %fieldadr17
  store float %y, float* %py
  %points18 = load { float*, i32 }*, { float*, i32 }** %points5
  %datarefref = getelementptr { float*, i32 }, { float*, i32 }* %points18,
    i32 0, i32 0
  %dataref19 = load float*, float** %datarefref
  %i20 = load i32, i32* %i4
  %tmp21 = mul i32 2, %i20
  %px22 = load float, float* %px
  %storeref = getelementptr float, float* %dataref19, i32 %tmp21
  store float %px22, float* %storeref
  %points23 = load { float*, i32 }*, { float*, i32 }** %points5
  %datarefref24 = getelementptr { float*, i32 }, { float*, i32 }* %points23,
```

```
  i32 0, i32 0
%dataref25 = load float*, float** %datarefref24
%i26 = load i32, i32* %i4
%tmp27 = mul i32 2, %i26
%tmp28 = add i32 %tmp27, 1
%py29 = load float, float* %py
%storeref30 = getelementptr float, float* %dataref25, i32 %tmp28
store float %py29, float* %storeref30
%cr = alloca float
%color_structs31 = load { { float, float, float, float }**, i32 }*, { {
  float, float, float, float }**, i32 }** %color_structs2
%i32 = load i32, i32* %i4
%dataref33 = getelementptr { { float, float, float, float }**, i32 }, { {
  float, float, float, float }**, i32 }* %color_structs31, i32 0, i32 0
%data34 = load { float, float, float, float }**, { float, float, float,
  float }*** %dataref33
%elref35 = getelementptr { float, float, float, float }*, { float, float,
  float, float }** %data34, i32 %i32
%el36 = load { float, float, float, float }*, { float, float, float, float
  }** %elref35
%fieldadr37 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %el36, i32 0, i32 0
%r = load float, float* %fieldadr37
store float %r, float* %cr
%cg = alloca float
%color_structs38 = load { { float, float, float, float }**, i32 }*, { {
  float, float, float, float }**, i32 }** %color_structs2
%i39 = load i32, i32* %i4
%dataref40 = getelementptr { { float, float, float, float }**, i32 }, { {
  float, float, float, float }**, i32 }* %color_structs38, i32 0, i32 0
%data41 = load { float, float, float, float }**, { float, float, float,
  float }*** %dataref40
%elref42 = getelementptr { float, float, float, float }*, { float, float,
  float, float }** %data41, i32 %i39
%el43 = load { float, float, float, float }*, { float, float, float, float
  }** %elref42
%fieldadr44 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %el43, i32 0, i32 1
%g = load float, float* %fieldadr44
store float %g, float* %cg
%cb = alloca float
%color_structs45 = load { { float, float, float, float }**, i32 }*, { { float,
  float, float, float }**, i32 }** %color_structs2
%i46 = load i32, i32* %i4
%dataref47 = getelementptr { { float, float, float, float }**, i32 }, { { float,
  float, float, float }**, i32 }* %color_structs45, i32 0, i32 0
%data48 = load { float, float, float, float }**, { float, float, float, float
  }*** %dataref47
%elref49 = getelementptr { float, float, float, float }*, { float, float, float,
  float }** %data48, i32 %i46
%el50 = load { float, float, float, float }*, { float, float, float, float }**
  %elref49
%fieldadr51 = getelementptr { float, float, float, float }, { float, float,
  float, float }* %el50, i32 0, i32 2
%b = load float, float* %fieldadr51
store float %b, float* %cb
%ca = alloca float
%color_structs52 = load { { float, float, float, float }**, i32 }*, { { float,
  float, float, float }**, i32 }** %color_structs2
```

```
%i53 = load i32, i32* %i4
%dataref54 = getelementptr { { float, float, float, float }**, i32 }, { { float,
  float, float, float }**, i32 }* %color_structs52, i32 0, i32 0
%data55 = load { float, float, float, float }**, { float, float, float, float
  }*** %dataref54
%elref56 = getelementptr { float, float, float, float }*, { float, float, float,
  float }** %data55, i32 %i53
%el57 = load { float, float, float, float }*, { float, float, float, float }**
  %elref56
%fieldadr58 = getelementptr { float, float, float, float }, { float, float,
  float,float }* %el57, i32 0, i32 3
%a = load float, float* %fieldadr58
store float %a, float* %ca
%colors59 = load { float*, i32 }*, { float*, i32 }** %colors6
%datarefref60 = getelementptr { float*, i32 }, { float*, i32 }* %colors59,
  i32 0, i32 0
%dataref61 = load float*, float** %datarefref60
%i62 = load i32, i32* %i4
%tmp63 = mul i32 4, %i62
%cr64 = load float, float* %cr
%storeref65 = getelementptr float, float* %dataref61, i32 %tmp63
store float %cr64, float* %storeref65
%colors66 = load { float*, i32 }*, { float*, i32 }** %colors6
%datarefref67 = getelementptr { float*, i32 }, { float*, i32 }* %colors66,
  i32 0, i32 0
%dataref68 = load float*, float** %datarefref67
%i69 = load i32, i32* %i4
%tmp70 = mul i32 4, %i69
%tmp71 = add i32 %tmp70, 1
%cg72 = load float, float* %cg
%storeref73 = getelementptr float, float* %dataref68, i32 %tmp71
store float %cg72, float* %storeref73
%colors74 = load { float*, i32 }*, { float*, i32 }** %colors6
%datarefref75 = getelementptr { float*, i32 }, { float*, i32 }* %colors74,
  i32 0, i32 0
%dataref76 = load float*, float** %datarefref75
%i77 = load i32, i32* %i4
%tmp78 = mul i32 4, %i77
%tmp79 = add i32 %tmp78, 2
%cb80 = load float, float* %cb
%storeref81 = getelementptr float, float* %dataref76, i32 %tmp79
store float %cb80, float* %storeref81
%colors82 = load { float*, i32 }*, { float*, i32 }** %colors6
%datarefref83 = getelementptr { float*, i32 }, { float*, i32 }* %colors82,
  i32 0, i32 0
%dataref84 = load float*, float** %datarefref83
%i85 = load i32, i32* %i4
%tmp86 = mul i32 4, %i85
%tmp87 = add i32 %tmp86, 3
%ca88 = load float, float* %ca
%storeref89 = getelementptr float, float* %dataref84, i32 %tmp87
store float %ca88, float* %storeref89
%point_structs90 = load { { float, float }**, i32 }*, { { float, float }**,
  i32 }** %point_structs1
%color_structs91 = load { { float, float, float, float }**, i32 }*, { { float,
  float, float, float }**, i32 }** %color_structs2
%numOfPoints92 = load i32, i32* %numOfPoints3
%i93 = load i32, i32* %i4
%tmp94 = add i32 %i93, 1
```

```
  %points95 = load { float*, i32 }*, { float*, i32 }** %points5
  %colors96 = load { float*, i32 }*, { float*, i32 }** %colors6
  call void @drawHelper({ { float, float }**, i32 }* %point_structs90, { { float,
    float, float, float }**, i32 }* %color_structs91, i32 %numOfPoints92, i32 %
    tmp94,
    { float*, i32 }* %points95, { float*, i32 }* %colors96)
  br label %merge
}

define void @drawPoints({ { float, float }**, i32 }* %point_structs, { { float,
    float, float, float }**, i32 }* %color_structs) {
entry:
  %point_structs1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %point_structs, { { float, float }**,
    i32 }** %point_structs1
  %color_structs2 = alloca { { float, float, float, float }**, i32 }*
  store { { float, float, float, float }**, i32 }* %color_structs, { { float,
    float, float, float }**, i32 }** %color_structs2
  %numOfPoints = alloca i32
  %point_structs3 = load { { float, float }**, i32 }*, { { float, float }**, i32
    }** %point_structs1
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float }**, i32
    }* %point_structs3, i32 0, i32 1
  %len = load i32, i32* %lenref
  store i32 %len, i32* %numOfPoints
  %points = alloca { float*, i32 }*
  %numOfPoints4 = load i32, i32* %numOfPoints
  %tmp = mul i32 %numOfPoints4, 2
  %malloccall = tail call i8* @malloc(i32 ptrtoint (float* getelementptr (float,
    float* null, i32 1) to i32))
  %arrdata = bitcast i8* %malloccall to float*
  %storeref = getelementptr float, float* %arrdata, i32 0
  store float 0.000000e+00, float* %storeref
  %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr
    ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall5 to { float*, i32 }*
  %arrdata6 = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 1
  store float* %arrdata, float** %arrdata6
  store i32 1, i32* %arrlen
  %lenref7 = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 1
  %len8 = load i32, i32* %lenref7
  %oflen = mul i32 %tmp, %len8
  %olddataref = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 0
  %olddata = load float*, float** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (float* getelementptr (float, float* null
    ,
    i32 1) to i32)
  %malloccall9 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata10 = bitcast i8* %malloccall9 to float*
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner

loop:                                             ; preds = %inner
  %i18 = load i32, i32* %i
```

```
  store i32 0, i32* %j
  %tmp19 = icmp slt i32 %i18, %oflen
  br i1 %tmp19, label %inner, label %continue

inner:                                            ; preds = %loop, %inner, %entry
  %i11 = load i32, i32* %j
  %i12 = load i32, i32* %i
  %elref = getelementptr float, float* %olddata, i32 %i11
  %el = load float, float* %elref
  %storeref13 = getelementptr float, float* %arrdata10, i32 %i12
  store float %el, float* %storeref13
  %i14 = add i32 %i12, 1
  store i32 %i14, i32* %i
  %j15 = add i32 %i11, 1
  store i32 %j15, i32* %j
  %j16 = load i32, i32* %j
  %tmp17 = icmp slt i32 %j16, %len8
  br i1 %tmp17, label %inner, label %loop

continue:                                         ; preds = %loop
  %malloccall20 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr
     ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr21 = bitcast i8* %malloccall20 to { float*, i32 }*
  %arrdata22 = getelementptr { float*, i32 }, { float*, i32 }* %arr21, i32 0, i32
    0
  %arrlen23 = getelementptr { float*, i32 }, { float*, i32 }* %arr21, i32 0, i32 1
  store float* %arrdata10, float** %arrdata22
  store i32 %oflen, i32* %arrlen23
  store { float*, i32 }* %arr21, { float*, i32 }** %points
  %colors = alloca { float*, i32 }*
  %numOfPoints24 = load i32, i32* %numOfPoints
  %tmp25 = mul i32 %numOfPoints24, 4
  %malloccall26 = tail call i8* @malloc(i32 ptrtoint (float* getelementptr (float,
    float* null, i32 1) to i32))
  %arrdata27 = bitcast i8* %malloccall26 to float*
  %storeref28 = getelementptr float, float* %arrdata27, i32 0
  store float 0.000000e+00, float* %storeref28
  %malloccall29 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr
     ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr30 = bitcast i8* %malloccall29 to { float*, i32 }*
  %arrdata31 = getelementptr { float*, i32 }, { float*, i32 }* %arr30, i32 0, i32
    0
  %arrlen32 = getelementptr { float*, i32 }, { float*, i32 }* %arr30, i32 0, i32 1
  store float* %arrdata27, float** %arrdata31
  store i32 1, i32* %arrlen32
  %lenref33 = getelementptr { float*, i32 }, { float*, i32 }* %arr30, i32 0, i32 1
  %len34 = load i32, i32* %lenref33
  %oflen35 = mul i32 %tmp25, %len34
  %olddataref36 = getelementptr { float*, i32 }, { float*, i32 }* %arr30, i32 0,
    i32 0
  %olddata37 = load float*, float** %olddataref36
  %mallocsize38 = mul i32 %oflen35, ptrtoint (float* getelementptr (float, float*
    null,
     i32 1) to i32)
  %malloccall39 = tail call i8* @malloc(i32 %mallocsize38)
  %arrdata40 = bitcast i8* %malloccall39 to float*
  %i41 = alloca i32
```

```
  store i32 0, i32* %i41
  %j42 = alloca i32
  store i32 0, i32* %j42
  br label %inner44

loop43:                                           ; preds = %inner44
  %i55 = load i32, i32* %i41
  store i32 0, i32* %j42
  %tmp56 = icmp slt i32 %i55, %oflen35
  br i1 %tmp56, label %inner44, label %continue45

inner44:
    ; preds = %loop43, %inner44, %continue
  %i46 = load i32, i32* %j42
  %i47 = load i32, i32* %i41
  %elref48 = getelementptr float, float* %olddata37, i32 %i46
  %el49 = load float, float* %elref48
  %storeref50 = getelementptr float, float* %arrdata40, i32 %i47
  store float %el49, float* %storeref50
  %i51 = add i32 %i47, 1
  store i32 %i51, i32* %i41
  %j52 = add i32 %i46, 1
  store i32 %j52, i32* %j42
  %j53 = load i32, i32* %j42
  %tmp54 = icmp slt i32 %j53, %len34
  br i1 %tmp54, label %inner44, label %loop43

continue45:                                       ; preds = %loop43
  %malloccall57 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr58 = bitcast i8* %malloccall57 to { float*, i32 }*
  %arrdata59 = getelementptr { float*, i32 }, { float*, i32 }* %arr58, i32 0, i32
    0
  %arrlen60 = getelementptr { float*, i32 }, { float*, i32 }* %arr58, i32 0, i32 1
  store float* %arrdata40, float** %arrdata59
  store i32 %oflen35, i32* %arrlen60
  store { float*, i32 }* %arr58, { float*, i32 }** %colors
  %point_structs61 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %point_structs1
  %color_structs62 = load { { float, float, float, float }**, i32 }*, { { float,
    float, float, float }**, i32 }** %color_structs2
  %numOfPoints63 = load i32, i32* %numOfPoints
  %points64 = load { float*, i32 }*, { float*, i32 }** %points
  %colors65 = load { float*, i32 }*, { float*, i32 }** %colors
  call void @drawHelper({ { float, float }**, i32 }* %point_structs61, { { float,
    float, float, float }**, i32 }* %color_structs62, i32 %numOfPoints63, i32 0,
    { float*, i32 }* %points64, { float*, i32 }* %colors65)
  %points66 = load { float*, i32 }*, { float*, i32 }** %points
  %colors67 = load { float*, i32 }*, { float*, i32 }** %colors
  call void @gl_drawPoint({ float*, i32 }* %points66, { float*, i32 }* %colors67,
    i32 2)
  ret void
}

define void @drawPath({ { float, float }**, i32 }* %point_structs, { { float,
    float, float, float }**, i32 }* %color_structs, i32 %colorMode) {
entry:
  %point_structs1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %point_structs, { { float, float }**, i32 }**
```

```
    %point_structs1
  %color_structs2 = alloca { { float, float, float, float }**, i32 }*
  store { { float, float, float, float }**, i32 }* %color_structs, { { float,
    float, float, float }**, i32 }** %color_structs2
  %colorMode3 = alloca i32
  store i32 %colorMode, i32* %colorMode3
  %numOfPoints = alloca i32
  %point_structs4 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %point_structs1
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %point_structs4, i32 0, i32 1
  %len = load i32, i32* %lenref
  store i32 %len, i32* %numOfPoints
  %points = alloca { float*, i32 }*
  %numOfPoints5 = load i32, i32* %numOfPoints
  %tmp = mul i32 %numOfPoints5, 2
  %malloccall = tail call i8* @malloc(i32 ptrtoint (float* getelementptr
    (float, float* null, i32 1) to i32))
  %arrdata = bitcast i8* %malloccall to float*
  %storeref = getelementptr float, float* %arrdata, i32 0
  store float 0.000000e+00, float* %storeref
  %malloccall6 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr
    ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall6 to { float*, i32 }*
  %arrdata7 = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 1
  store float* %arrdata, float** %arrdata7
  store i32 1, i32* %arrlen
  %lenref8 = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 1
  %len9 = load i32, i32* %lenref8
  %oflen = mul i32 %tmp, %len9
  %olddataref = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 0
  %olddata = load float*, float** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (float* getelementptr (float, float* null
    ,
    i32 1) to i32)
  %malloccall10 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata11 = bitcast i8* %malloccall10 to float*
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner

loop:                                             ; preds = %inner
  %i19 = load i32, i32* %i
  store i32 0, i32* %j
  %tmp20 = icmp slt i32 %i19, %oflen
  br i1 %tmp20, label %inner, label %continue

inner:                                            ; preds = %loop, %inner, %entry
  %i12 = load i32, i32* %j
  %i13 = load i32, i32* %i
  %elref = getelementptr float, float* %olddata, i32 %i12
  %el = load float, float* %elref
  %storeref14 = getelementptr float, float* %arrdata11, i32 %i13
  store float %el, float* %storeref14
  %i15 = add i32 %i13, 1
```

```
  store i32 %i15, i32* %i
  %j16 = add i32 %i12, 1
  store i32 %j16, i32* %j
  %j17 = load i32, i32* %j
  %tmp18 = icmp slt i32 %j17, %len9
  br i1 %tmp18, label %inner, label %loop

continue:                                         ; preds = %loop
  %malloccall21 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr22 = bitcast i8* %malloccall21 to { float*, i32 }*
  %arrdata23 = getelementptr { float*, i32 }, { float*, i32 }* %arr22, i32 0,
    i32 0
  %arrlen24 = getelementptr { float*, i32 }, { float*, i32 }* %arr22, i32 0,
    i32 1
  store float* %arrdata11, float** %arrdata23
  store i32 %oflen, i32* %arrlen24
  store { float*, i32 }* %arr22, { float*, i32 }** %points
  %colors = alloca { float*, i32 }*
  %numOfPoints25 = load i32, i32* %numOfPoints
  %tmp26 = mul i32 %numOfPoints25, 4
  %malloccall27 = tail call i8* @malloc(i32 ptrtoint (float* getelementptr
    (float,float* null, i32 1) to i32))
  %arrdata28 = bitcast i8* %malloccall27 to float*
  %storeref29 = getelementptr float, float* %arrdata28, i32 0
  store float 0.000000e+00, float* %storeref29
  %malloccall30 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr31 = bitcast i8* %malloccall30 to { float*, i32 }*
  %arrdata32 = getelementptr { float*, i32 }, { float*, i32 }* %arr31, i32 0,
    i32 0
  %arrlen33 = getelementptr { float*, i32 }, { float*, i32 }* %arr31, i32 0,
    i32 1
  store float* %arrdata28, float** %arrdata32
  store i32 1, i32* %arrlen33
  %lenref34 = getelementptr { float*, i32 }, { float*, i32 }* %arr31, i32 0,
    i32 1
  %len35 = load i32, i32* %lenref34
  %oflen36 = mul i32 %tmp26, %len35
  %olddataref37 = getelementptr { float*, i32 }, { float*, i32 }* %arr31, i32 0,
    i32 0
  %olddata38 = load float*, float** %olddataref37
  %mallocsize39 = mul i32 %oflen36, ptrtoint (float* getelementptr (float, float*
    null,i32 1) to i32)
  %malloccall40 = tail call i8* @malloc(i32 %mallocsize39)
  %arrdata41 = bitcast i8* %malloccall40 to float*
  %i42 = alloca i32
  store i32 0, i32* %i42
  %j43 = alloca i32
  store i32 0, i32* %j43
  br label %inner45

loop44:                                           ; preds = %inner45
  %i56 = load i32, i32* %i42
  store i32 0, i32* %j43
  %tmp57 = icmp slt i32 %i56, %oflen36
  br i1 %tmp57, label %inner45, label %continue46

inner45:
```

```
    ; preds = %loop44, %inner45, %continue
  %i47 = load i32, i32* %j43
  %i48 = load i32, i32* %i42
  %elref49 = getelementptr float, float* %olddata38, i32 %i47
  %el50 = load float, float* %elref49
  %storeref51 = getelementptr float, float* %arrdata41, i32 %i48
  store float %el50, float* %storeref51
  %i52 = add i32 %i48, 1
  store i32 %i52, i32* %i42
  %j53 = add i32 %i47, 1
  store i32 %j53, i32* %j43
  %j54 = load i32, i32* %j43
  %tmp55 = icmp slt i32 %j54, %len35
  br i1 %tmp55, label %inner45, label %loop44

continue46:                                       ; preds = %loop44
  %malloccall58 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr59 = bitcast i8* %malloccall58 to { float*, i32 }*
  %arrdata60 = getelementptr { float*, i32 }, { float*, i32 }* %arr59, i32 0, i32
    0
  %arrlen61 = getelementptr { float*, i32 }, { float*, i32 }* %arr59, i32 0, i32 1
  store float* %arrdata41, float** %arrdata60
  store i32 %oflen36, i32* %arrlen61
  store { float*, i32 }* %arr59, { float*, i32 }** %colors
  %point_structs62 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %point_structs1
  %color_structs63 = load { { float, float, float, float }**, i32 }*, { { float,
    float, float, float }**, i32 }** %color_structs2
  %numOfPoints64 = load i32, i32* %numOfPoints
  %points65 = load { float*, i32 }*, { float*, i32 }** %points
  %colors66 = load { float*, i32 }*, { float*, i32 }** %colors
  call void @drawHelper({ { float, float }**, i32 }* %point_structs62, { { float,
    float, float, float }**, i32 }* %color_structs63, i32 %numOfPoints64, i32 0,
    { float*, i32 }* %points65, { float*, i32 }* %colors66)
  %points67 = load { float*, i32 }*, { float*, i32 }** %points
  %colors68 = load { float*, i32 }*, { float*, i32 }** %colors
  %colorMode69 = load i32, i32* %colorMode3
  call void @gl_drawCurve({ float*, i32 }* %points67, { float*, i32 }* %colors68,
    i32 %colorMode69)
  ret void
}

define void @drawShape({ { float, float }**, i32 }* %point_structs, { { float,
    float, float, float }**, i32 }* %color_structs, i32 %colorMode, i32 %filled) {
entry:
  %point_structs1 = alloca { { float, float }**, i32 }*
  store { { float, float }**, i32 }* %point_structs, { { float, float }**, i32
    }** %point_structs1
  %color_structs2 = alloca { { float, float, float, float }**, i32 }*
  store { { float, float, float, float }**, i32 }* %color_structs, { { float,
    float, float, float }**, i32 }** %color_structs2
  %colorMode3 = alloca i32
  store i32 %colorMode, i32* %colorMode3
  %filled4 = alloca i32
  store i32 %filled, i32* %filled4
  %numOfPoints = alloca i32
  %point_structs5 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %point_structs1
```

```
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %point_structs5, i32 0, i32 1
  %len = load i32, i32* %lenref
  store i32 %len, i32* %numOfPoints
  %points = alloca { float*, i32 }*
  %numOfPoints6 = load i32, i32* %numOfPoints
  %tmp = mul i32 %numOfPoints6, 2
  %malloccall = tail call i8* @malloc(i32 ptrtoint (float* getelementptr
    (float, float* null, i32 1) to i32))
  %arrdata = bitcast i8* %malloccall to float*
  %storeref = getelementptr float, float* %arrdata, i32 0
  store float 0.000000e+00, float* %storeref
  %malloccall7 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall7 to { float*, i32 }*
  %arrdata8 = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 1
  store float* %arrdata, float** %arrdata8
  store i32 1, i32* %arrlen
  %lenref9 = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 1
  %len10 = load i32, i32* %lenref9
  %oflen = mul i32 %tmp, %len10
  %olddataref = getelementptr { float*, i32 }, { float*, i32 }* %arr, i32 0, i32 0
  %olddata = load float*, float** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (float* getelementptr (float, float* null
    ,
    i32 1) to i32)
  %malloccall11 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata12 = bitcast i8* %malloccall11 to float*
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner

loop:                                             ; preds = %inner
  %i20 = load i32, i32* %i
  store i32 0, i32* %j
  %tmp21 = icmp slt i32 %i20, %oflen
  br i1 %tmp21, label %inner, label %continue

inner:                                            ; preds = %loop, %inner, %entry
  %i13 = load i32, i32* %j
  %i14 = load i32, i32* %i
  %elref = getelementptr float, float* %olddata, i32 %i13
  %el = load float, float* %elref
  %storeref15 = getelementptr float, float* %arrdata12, i32 %i14
  store float %el, float* %storeref15
  %i16 = add i32 %i14, 1
  store i32 %i16, i32* %i
  %j17 = add i32 %i13, 1
  store i32 %j17, i32* %j
  %j18 = load i32, i32* %j
  %tmp19 = icmp slt i32 %j18, %len10
  br i1 %tmp19, label %inner, label %loop

continue:                                         ; preds = %loop
  %malloccall22 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr
```

```
    ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr23 = bitcast i8* %malloccall22 to { float*, i32 }*
  %arrdata24 = getelementptr { float*, i32 }, { float*, i32 }* %arr23, i32 0, i32
    0
  %arrlen25 = getelementptr { float*, i32 }, { float*, i32 }* %arr23, i32 0, i32 1
  store float* %arrdata12, float** %arrdata24
  store i32 %oflen, i32* %arrlen25
  store { float*, i32 }* %arr23, { float*, i32 }** %points
  %colors = alloca { float*, i32 }*
  %numOfPoints26 = load i32, i32* %numOfPoints
  %tmp27 = mul i32 %numOfPoints26, 4
  %malloccall28 = tail call i8* @malloc(i32 ptrtoint (float* getelementptr
    (float,float* null, i32 1) to i32))
  %arrdata29 = bitcast i8* %malloccall28 to float*
  %storeref30 = getelementptr float, float* %arrdata29, i32 0
  store float 0.000000e+00, float* %storeref30
  %malloccall31 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32 }*
    getelementptr({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr32 = bitcast i8* %malloccall31 to { float*, i32 }*
  %arrdata33 = getelementptr { float*, i32 }, { float*, i32 }* %arr32,
    i32 0, i32 0
  %arrlen34 = getelementptr { float*, i32 }, { float*, i32 }* %arr32,
    i32 0, i32 1
  store float* %arrdata29, float** %arrdata33
  store i32 1, i32* %arrlen34
  %lenref35 = getelementptr { float*, i32 }, { float*, i32 }* %arr32,
    i32 0, i32 1
  %len36 = load i32, i32* %lenref35
  %oflen37 = mul i32 %tmp27, %len36
  %olddataref38 = getelementptr { float*, i32 }, { float*, i32 }*
    %arr32, i32 0, i32 0
  %olddata39 = load float*, float** %olddataref38
  %mallocsize40 = mul i32 %oflen37, ptrtoint (float* getelementptr
    (float, float*null,i32 1) to i32)
  %malloccall41 = tail call i8* @malloc(i32 %mallocsize40)
  %arrdata42 = bitcast i8* %malloccall41 to float*
  %i43 = alloca i32
  store i32 0, i32* %i43
  %j44 = alloca i32
  store i32 0, i32* %j44
  br label %inner46

loop45:
    ; preds = %inner46
  %i57 = load i32, i32* %i43
  store i32 0, i32* %j44
  %tmp58 = icmp slt i32 %i57, %oflen37
  br i1 %tmp58, label %inner46, label %continue47

inner46:
    ; preds = %loop45, %inner46, %continue
  %i48 = load i32, i32* %j44
  %i49 = load i32, i32* %i43
  %elref50 = getelementptr float, float* %olddata39, i32 %i48
  %el51 = load float, float* %elref50
  %storeref52 = getelementptr float, float* %arrdata42, i32 %i49
  store float %el51, float* %storeref52
  %i53 = add i32 %i49, 1
  store i32 %i53, i32* %i43
```

```
  %j54 = add i32 %i48, 1
  store i32 %j54, i32* %j44
  %j55 = load i32, i32* %j44
  %tmp56 = icmp slt i32 %j55, %len36
  br i1 %tmp56, label %inner46, label %loop45

continue47:                                       ; preds = %loop45
  %malloccall59 = tail call i8* @malloc(i32 ptrtoint ({ float*, i32
    }* getelementptr
    ({ float*, i32 }, { float*, i32 }* null, i32 1) to i32))
  %arr60 = bitcast i8* %malloccall59 to { float*, i32 }*
  %arrdata61 = getelementptr { float*, i32 }, { float*, i32 }* %arr60,
    i32 0, i32 0
  %arrlen62 = getelementptr { float*, i32 }, { float*, i32 }* %arr60,
    i32 0, i32 1
  store float* %arrdata42, float** %arrdata61
  store i32 %oflen37, i32* %arrlen62
  store { float*, i32 }* %arr60, { float*, i32 }** %colors
  %point_structs63 = load { { float, float }**, i32 }*, { { float,
    float }**, i32}** %point_structs1
  %color_structs64 = load { { float, float, float, float }**, i32 }*,
    { { float,float, float, float }**, i32 }** %color_structs2
  %numOfPoints65 = load i32, i32* %numOfPoints
  %points66 = load { float*, i32 }*, { float*, i32 }** %points
  %colors67 = load { float*, i32 }*, { float*, i32 }** %colors
  call void @drawHelper({ { float, float }**, i32 }* %point_structs63, { { float,
    float, float, float }**, i32 }* %color_structs64, i32 %numOfPoints65,
    i32 0, { float*, i32 }* %points66, { float*, i32 }* %colors67)
  %points68 = load { float*, i32 }*, { float*, i32 }** %points
  %colors69 = load { float*, i32 }*, { float*, i32 }** %colors
  %colorMode70 = load i32, i32* %colorMode3
  %filled71 = load i32, i32* %filled4
  call void @gl_drawShape({ float*, i32 }* %points68, { float*, i32 }* %colors69,
    i32 %colorMode70, i32 %filled71)
  ret void
}

define void @endCanvas({ i32, i32, i32 }* %c) {
entry:
  %c1 = alloca { i32, i32, i32 }*
  store { i32, i32, i32 }* %c, { i32, i32, i32 }** %c1
  %c2 = load { i32, i32, i32 }*, { i32, i32, i32 }** %c1
  %fieldadr = getelementptr { i32, i32, i32 }, { i32, i32, i32 }* %c2, i32 0, i32
    0
  %width = load i32, i32* %fieldadr
  %c3 = load { i32, i32, i32 }*, { i32, i32, i32 }** %c1
  %fieldadr4 = getelementptr { i32, i32, i32 }, { i32, i32, i32 }* %c3, i32 0, i32
    1
  %height = load i32, i32* %fieldadr4
  %c5 = load { i32, i32, i32 }*, { i32, i32, i32 }** %c1
  %fieldadr6 = getelementptr { i32, i32, i32 }, { i32, i32, i32 }* %c5, i32 0, i32
    2
  %file_number = load i32, i32* %fieldadr6
  call void @gl_endRendering(i32 %width, i32 %height, i32 %file_number)
  ret void
}

define void @rotate({ float, float }* %p, float %angle, i32 %direction, { float,
    float }* %about) {
```

```
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %angle2 = alloca float
  store float %angle, float* %angle2
  %direction3 = alloca i32
  store i32 %direction, i32* %direction3
  %about4 = alloca { float, float }*
  store { float, float }* %about, { float, float }** %about4
  %px = alloca float
  %p5 = load { float, float }*, { float, float }** %p1
  %fieldadr = getelementptr { float, float }, { float, float }* %p5, i32 0, i32 0
  %x = load float, float* %fieldadr
  %about6 = load { float, float }*, { float, float }** %about4
  %fieldadr7 = getelementptr { float, float }, { float, float }* %about6, i32 0,
    i32 0
  %x8 = load float, float* %fieldadr7
  %tmp = fsub float %x, %x8
  store float %tmp, float* %px
  %py = alloca float
  %p9 = load { float, float }*, { float, float }** %p1
  %fieldadr10 = getelementptr { float, float }, { float, float }* %p9, i32 0, i32
    1
  %y = load float, float* %fieldadr10
  %about11 = load { float, float }*, { float, float }** %about4
  %fieldadr12 = getelementptr { float, float }, { float, float }* %about11, i32
    0, i32 1
  %y13 = load float, float* %fieldadr12
  %tmp14 = fsub float %y, %y13
  store float %tmp14, float* %py
  %direction15 = load i32, i32* %direction3
  %tmp16 = icmp eq i32 %direction15, -1
  %if_tmp = alloca float
  br i1 %tmp16, label %then, label %else

merge:                                            ; preds = %merge48, %then
  %if_tmp93 = load float, float* %if_tmp
  ret void

then:                                             ; preds = %entry
  %p17 = load { float, float }*, { float, float }** %p1
  %px18 = load float, float* %px
  %angle19 = load float, float* %angle2
  %fxn_result = call float @cos(float %angle19)
  %tmp20 = fmul float %px18, %fxn_result
  %py21 = load float, float* %py
  %angle22 = load float, float* %angle2
  %fxn_result23 = call float @sin(float %angle22)
  %tmp24 = fmul float %py21, %fxn_result23
  %tmp25 = fsub float %tmp20, %tmp24
  %about26 = load { float, float }*, { float, float }** %about4
  %fieldadr27 = getelementptr { float, float }, { float, float }* %about26, i32 0,
    i32 0
  %x28 = load float, float* %fieldadr27
  %tmp29 = fadd float %tmp25, %x28
  %ref = getelementptr { float, float }, { float, float }* %p17, i32 0, i32 0
  store float %tmp29, float* %ref
  %p30 = load { float, float }*, { float, float }** %p1
  %px31 = load float, float* %px
```

```
  %angle32 = load float, float* %angle2
  %fxn_result33 = call float @sin(float %angle32)
  %tmp34 = fmul float %px31, %fxn_result33
  %py35 = load float, float* %py
  %angle36 = load float, float* %angle2
  %fxn_result37 = call float @cos(float %angle36)
  %tmp38 = fmul float %py35, %fxn_result37
  %tmp39 = fadd float %tmp34, %tmp38
  %about40 = load { float, float }*, { float, float }** %about4
  %fieldadr41 = getelementptr { float, float }, { float, float }* %about40, i32 0,
    i32 1
  %y42 = load float, float* %fieldadr41
  %tmp43 = fadd float %tmp39, %y42
  %ref44 = getelementptr { float, float }, { float, float }* %p30, i32 0, i32 1
  store float %tmp43, float* %ref44
  store float %tmp43, float* %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %direction45 = load i32, i32* %direction3
  %tmp46 = icmp eq i32 %direction45, 1
  %if_tmp47 = alloca float
  br i1 %tmp46, label %then49, label %else50

merge48:                                          ; preds = %else50, %then49
  %if_tmp92 = load float, float* %if_tmp47
  store float %if_tmp92, float* %if_tmp
  br label %merge

then49:                                           ; preds = %else
  %p51 = load { float, float }*, { float, float }** %p1
  %px52 = load float, float* %px
  %angle53 = load float, float* %angle2
  %fxn_result54 = call float @cos(float %angle53)
  %tmp55 = fmul float %px52, %fxn_result54
  %py56 = load float, float* %py
  %angle57 = load float, float* %angle2
  %fxn_result58 = call float @sin(float %angle57)
  %tmp59 = fmul float %py56, %fxn_result58
  %tmp60 = fadd float %tmp55, %tmp59
  %about61 = load { float, float }*, { float, float }** %about4
  %fieldadr62 = getelementptr { float, float }, { float, float }* %about61, i32 0,
    i32 0
  %x63 = load float, float* %fieldadr62
  %tmp64 = fadd float %tmp60, %x63
  %ref65 = getelementptr { float, float }, { float, float }* %p51, i32 0, i32 0
  store float %tmp64, float* %ref65
  %p66 = load { float, float }*, { float, float }** %p1
  %px67 = load float, float* %px
  %angle68 = load float, float* %angle2
  %fxn_result69 = call float @sin(float %angle68)
  %tmp70 = fmul float %px67, %fxn_result69
  %tmp71 = fneg float %tmp70
  %py72 = load float, float* %py
  %angle73 = load float, float* %angle2
  %fxn_result74 = call float @cos(float %angle73)
  %tmp75 = fmul float %py72, %fxn_result74
  %tmp76 = fadd float %tmp71, %tmp75
  %about77 = load { float, float }*, { float, float }** %about4
```

```
  %fieldadr78 = getelementptr { float, float }, { float, float }* %about77, i32 0,
    i32 1
  %y79 = load float, float* %fieldadr78
  %tmp80 = fadd float %tmp76, %y79
  %ref81 = getelementptr { float, float }, { float, float }* %p66, i32 0, i32 1
  store float %tmp80, float* %ref81
  store float %tmp80, float* %if_tmp47
  br label %merge48

else50:                                           ; preds = %else
  %p82 = load { float, float }*, { float, float }** %p1
  %p83 = load { float, float }*, { float, float }** %p1
  %fieldadr84 = getelementptr { float, float }, { float, float }* %p83, i32 0, i32
    0
  %x85 = load float, float* %fieldadr84
  %ref86 = getelementptr { float, float }, { float, float }* %p82, i32 0, i32 0
  store float %x85, float* %ref86
  %p87 = load { float, float }*, { float, float }** %p1
  %p88 = load { float, float }*, { float, float }** %p1
  %fieldadr89 = getelementptr { float, float }, { float, float }* %p88, i32 0, i32
    1
  %y90 = load float, float* %fieldadr89
  %ref91 = getelementptr { float, float }, { float, float }* %p87, i32 0, i32 1
  store float %y90, float* %ref91
  store float %y90, float* %if_tmp47
  br label %merge48
}

define void @trans({ float, float }* %p, { float, float }* %direction) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %direction2 = alloca { float, float }*
  store { float, float }* %direction, { float, float }** %direction2
  %p3 = load { float, float }*, { float, float }** %p1
  %p4 = load { float, float }*, { float, float }** %p1
  %fieldadr = getelementptr { float, float }, { float, float }* %p4, i32 0, i32 0
  %x = load float, float* %fieldadr
  %direction5 = load { float, float }*, { float, float }** %direction2
  %fieldadr6 = getelementptr { float, float }, { float, float }* %direction5, i32
    0,
    i32 0
  %x7 = load float, float* %fieldadr6
  %tmp = fadd float %x, %x7
  %ref = getelementptr { float, float }, { float, float }* %p3, i32 0, i32 0
  store float %tmp, float* %ref
  %p8 = load { float, float }*, { float, float }** %p1
  %p9 = load { float, float }*, { float, float }** %p1
  %fieldadr10 = getelementptr { float, float }, { float, float }* %p9, i32 0, i32
    1
  %y = load float, float* %fieldadr10
  %direction11 = load { float, float }*, { float, float }** %direction2
  %fieldadr12 = getelementptr { float, float }, { float, float }* %direction11,
    i32 0,
    i32 1
  %y13 = load float, float* %fieldadr12
  %tmp14 = fadd float %y, %y13
  %ref15 = getelementptr { float, float }, { float, float }* %p8, i32 0, i32 1
  store float %tmp14, float* %ref15
```

```
  ret void
}

define void @scale({ float, float }* %p, float %sx, float %sy) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %sx2 = alloca float
  store float %sx, float* %sx2
  %sy3 = alloca float
  store float %sy, float* %sy3
  %p4 = load { float, float }*, { float, float }** %p1
  %p5 = load { float, float }*, { float, float }** %p1
  %fieldadr = getelementptr { float, float }, { float, float }* %p5, i32 0, i32 0
  %x = load float, float* %fieldadr
  %sx6 = load float, float* %sx2
  %tmp = fmul float %x, %sx6
  %ref = getelementptr { float, float }, { float, float }* %p4, i32 0, i32 0
  store float %tmp, float* %ref
  %p7 = load { float, float }*, { float, float }** %p1
  %p8 = load { float, float }*, { float, float }** %p1
  %fieldadr9 = getelementptr { float, float }, { float, float }* %p8, i32 0, i32 1
  %y = load float, float* %fieldadr9
  %sy10 = load float, float* %sy3
  %tmp11 = fmul float %y, %sy10
  %ref12 = getelementptr { float, float }, { float, float }* %p7, i32 0, i32 1
  store float %tmp11, float* %ref12
  ret void
}

define { float, float }* @rotated({ float, float }* %p, float %angle, i32 %
    direction,
    { float, float }* %about) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %angle2 = alloca float
  store float %angle, float* %angle2
  %direction3 = alloca i32
  store i32 %direction, i32* %direction3
  %about4 = alloca { float, float }*
  store { float, float }* %about, { float, float }** %about4
  %q = alloca { float, float }*
  %p5 = load { float, float }*, { float, float }** %p1
  %copied = call { float, float }* @__copy2.2({ float, float }* %p5)
  store { float, float }* %copied, { float, float }** %q
  %q6 = load { float, float }*, { float, float }** %q
  %angle7 = load float, float* %angle2
  %direction8 = load i32, i32* %direction3
  %about9 = load { float, float }*, { float, float }** %about4
  call void @rotate({ float, float }* %q6, float %angle7, i32 %direction8, { float
    ,
    float }* %about9)
  %q10 = load { float, float }*, { float, float }** %q
  ret { float, float }* %q10
}

define { float, float }* @__copy2.2({ float, float }* %to_copy) {
entry:
```

```
  %to_copy1 = alloca { float, float }*
  store { float, float }* %to_copy, { float, float }** %to_copy1
  %to_copy2 = load { float, float }*, { float, float }** %to_copy1
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint (float*
    getelementptr(float, float* null, i32 1) to i64), i64 2) to i32))
  %struct = bitcast i8* %malloccall to { float, float }*
  %flref = getelementptr { float, float }, { float, float }* %to_copy2, i32 0, i32
    0
  %fl = load float, float* %flref
  %ref = getelementptr { float, float }, { float, float }* %struct, i32 0, i32 0
  store float %fl, float* %ref
  %flref3 = getelementptr { float, float }, { float, float }* %to_copy2, i32 0,
    i32 1
  %fl4 = load float, float* %flref3
  %ref5 = getelementptr { float, float }, { float, float }* %struct, i32 0,
    i32 1
  store float %fl4, float* %ref5
  ret { float, float }* %struct
}

define { float, float }* @translated({ float, float }* %p, { float, float }*
    %direction) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %direction2 = alloca { float, float }*
  store { float, float }* %direction, { float, float }** %direction2
  %q = alloca { float, float }*
  %p3 = load { float, float }*, { float, float }** %p1
  %copied = call { float, float }* @__copy2.3({ float, float }* %p3)
  store { float, float }* %copied, { float, float }** %q
  %q4 = load { float, float }*, { float, float }** %q
  %direction5 = load { float, float }*, { float, float }** %direction2
  call void @trans({ float, float }* %q4, { float, float }* %direction5)
  %q6 = load { float, float }*, { float, float }** %q
  ret { float, float }* %q6
}

define { float, float }* @__copy2.3({ float, float }* %to_copy) {
entry:
  %to_copy1 = alloca { float, float }*
  store { float, float }* %to_copy, { float, float }** %to_copy1
  %to_copy2 = load { float, float }*, { float, float }** %to_copy1
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
    (float* getelementptr(float, float* null, i32 1) to i64), i64 2) to i32))
  %struct = bitcast i8* %malloccall to { float, float }*
  %flref = getelementptr { float, float }, { float, float }* %to_copy2,
    i32 0, i32 0
  %fl = load float, float* %flref
  %ref = getelementptr { float, float }, { float, float }* %struct, i32 0,
    i32 0
  store float %fl, float* %ref
  %flref3 = getelementptr { float, float }, { float, float }* %to_copy2,
    i32 0, i32 1
  %fl4 = load float, float* %flref3
  %ref5 = getelementptr { float, float }, { float, float }* %struct, i32 0,
    i32 1
  store float %fl4, float* %ref5
  ret { float, float }* %struct
```

```
}

define { float, float }* @scaled({ float, float }* %p, float %sx, float %sy) {
entry:
  %p1 = alloca { float, float }*
  store { float, float }* %p, { float, float }** %p1
  %sx2 = alloca float
  store float %sx, float* %sx2
  %sy3 = alloca float
  store float %sy, float* %sy3
  %q = alloca { float, float }*
  %p4 = load { float, float }*, { float, float }** %p1
  %copied = call { float, float }* @__copy2.4({ float, float }* %p4)
  store { float, float }* %copied, { float, float }** %q
  %p5 = load { float, float }*, { float, float }** %p1
  %sx6 = load float, float* %sx2
  %sy7 = load float, float* %sy3
  call void @scale({ float, float }* %p5, float %sx6, float %sy7)
  %q8 = load { float, float }*, { float, float }** %q
  ret { float, float }* %q8
}

define { float, float }* @__copy2.4({ float, float }* %to_copy) {
entry:
  %to_copy1 = alloca { float, float }*
  store { float, float }* %to_copy, { float, float }** %to_copy1
  %to_copy2 = load { float, float }*, { float, float }** %to_copy1
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
    (float* getelementptr(float, float* null, i32 1) to i64), i64 2) to i32))
  %struct = bitcast i8* %malloccall to { float, float }*
  %flref = getelementptr { float, float }, { float, float }* %to_copy2,
    i32 0, i32 0
  %fl = load float, float* %flref
  %ref = getelementptr { float, float }, { float, float }* %struct, i32 0,
    i32 0
  store float %fl, float* %ref
  %flref3 = getelementptr { float, float }, { float, float }* %to_copy2,
    i32 0, i32 1
  %fl4 = load float, float* %flref3
  %ref5 = getelementptr { float, float }, { float, float }* %struct, i32 0,
    i32 1
  store float %fl4, float* %ref5
  ret { float, float }* %struct
}

define void @fill_ints({ i32*, i32 }* %a, i32 %i) {
entry:
  %a1 = alloca { i32*, i32 }*
  store { i32*, i32 }* %a, { i32*, i32 }** %a1
  %i2 = alloca i32
  store i32 %i, i32* %i2
  %i3 = load i32, i32* %i2
  %a4 = load { i32*, i32 }*, { i32*, i32 }** %a1
  %lenref = getelementptr { i32*, i32 }, { i32*, i32 }* %a4, i32 0, i32 1
  %len = load i32, i32* %lenref
  %tmp = icmp slt i32 %i3, %len
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
```

```
  ret void

then:                                             ; preds = %entry
  %a5 = load { i32*, i32 }*, { i32*, i32 }** %a1
  %datarefref = getelementptr { i32*, i32 }, { i32*, i32 }* %a5, i32 0, i32 0
  %dataref = load i32*, i32** %datarefref
  %i6 = load i32, i32* %i2
  %i7 = load i32, i32* %i2
  %storeref = getelementptr i32, i32* %dataref, i32 %i6
  store i32 %i7, i32* %storeref
  %a8 = load { i32*, i32 }*, { i32*, i32 }** %a1
  %i9 = load i32, i32* %i2
  %tmp10 = add i32 %i9, 1
  call void @fill_ints({ i32*, i32 }* %a8, i32 %tmp10)
  br label %merge

else:                                             ; preds = %entry
  br label %merge
}

define { i32*, i32 }* @ints(i32 %n) {
entry:
  %n1 = alloca i32
  store i32 %n, i32* %n1
  %n2 = load i32, i32* %n1
  %tmp = icmp sle i32 %n2, 0
  %if_tmp = alloca { i32*, i32 }*
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %continue, %then
  %if_tmp30 = load { i32*, i32 }*, { i32*, i32 }** %if_tmp
  ret { i32*, i32 }* %if_tmp30

then:                                             ; preds = %entry
  %a = alloca { i32*, i32 }*
  %malloccall = tail call i8* @malloc(i32 0)
  %arrdata = bitcast i8* %malloccall to i32*
  %malloccall3 = tail call i8* @malloc(i32 ptrtoint ({ i32*, i32 }* getelementptr
    ({ i32*, i32 }, { i32*, i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall3 to { i32*, i32 }*
  %arrdata4 = getelementptr { i32*, i32 }, { i32*, i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { i32*, i32 }, { i32*, i32 }* %arr, i32 0, i32 1
  store i32* %arrdata, i32** %arrdata4
  store i32 0, i32* %arrlen
  store { i32*, i32 }* %arr, { i32*, i32 }** %a
  store { i32*, i32 }* %arr, { i32*, i32 }** %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %arr5 = alloca { i32*, i32 }*
  %n6 = load i32, i32* %n1
  %malloccall7 = tail call i8* @malloc(i32 ptrtoint (i32* getelementptr (i32, i32*
    null, i32 1) to i32))
  %arrdata8 = bitcast i8* %malloccall7 to i32*
  %storeref = getelementptr i32, i32* %arrdata8, i32 0
  store i32 0, i32* %storeref
  %malloccall9 = tail call i8* @malloc(i32 ptrtoint ({ i32*, i32 }* getelementptr
    ({
    i32*, i32 }, { i32*, i32 }* null, i32 1) to i32))
```

```
  %arr10 = bitcast i8* %malloccall9 to { i32*, i32 }*
  %arrdata11 = getelementptr { i32*, i32 }, { i32*, i32 }* %arr10, i32 0, i32 0
  %arrlen12 = getelementptr { i32*, i32 }, { i32*, i32 }* %arr10, i32 0, i32 1
  store i32* %arrdata8, i32** %arrdata11
  store i32 1, i32* %arrlen12
  %lenref = getelementptr { i32*, i32 }, { i32*, i32 }* %arr10, i32 0, i32 1
  %len = load i32, i32* %lenref
  %oflen = mul i32 %n6, %len
  %olddataref = getelementptr { i32*, i32 }, { i32*, i32 }* %arr10, i32 0, i32 0
  %olddata = load i32*, i32** %olddataref
  %mallocsize = mul i32 %oflen, ptrtoint (i32* getelementptr (i32, i32* null, i32
    1)
    to i32)
  %malloccall13 = tail call i8* @malloc(i32 %mallocsize)
  %arrdata14 = bitcast i8* %malloccall13 to i32*
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  br label %inner

loop:                                             ; preds = %inner
  %i22 = load i32, i32* %i
  store i32 0, i32* %j
  %tmp23 = icmp slt i32 %i22, %oflen
  br i1 %tmp23, label %inner, label %continue

inner:                                            ; preds = %loop, %inner, %else
  %i15 = load i32, i32* %j
  %i16 = load i32, i32* %i
  %elref = getelementptr i32, i32* %olddata, i32 %i15
  %el = load i32, i32* %elref
  %storeref17 = getelementptr i32, i32* %arrdata14, i32 %i16
  store i32 %el, i32* %storeref17
  %i18 = add i32 %i16, 1
  store i32 %i18, i32* %i
  %j19 = add i32 %i15, 1
  store i32 %j19, i32* %j
  %j20 = load i32, i32* %j
  %tmp21 = icmp slt i32 %j20, %len
  br i1 %tmp21, label %inner, label %loop

continue:                                         ; preds = %loop
  %malloccall24 = tail call i8* @malloc(i32 ptrtoint ({ i32*, i32 }* getelementptr
    ({ i32*, i32 }, { i32*, i32 }* null, i32 1) to i32))
  %arr25 = bitcast i8* %malloccall24 to { i32*, i32 }*
  %arrdata26 = getelementptr { i32*, i32 }, { i32*, i32 }* %arr25, i32 0, i32 0
  %arrlen27 = getelementptr { i32*, i32 }, { i32*, i32 }* %arr25, i32 0, i32 1
  store i32* %arrdata14, i32** %arrdata26
  store i32 %oflen, i32* %arrlen27
  store { i32*, i32 }* %arr25, { i32*, i32 }** %arr5
  %arr28 = load { i32*, i32 }*, { i32*, i32 }** %arr5
  call void @fill_ints({ i32*, i32 }* %arr28, i32 0)
  %arr29 = load { i32*, i32 }*, { i32*, i32 }** %arr5
  store { i32*, i32 }* %arr29, { i32*, i32 }** %if_tmp
  br label %merge
}

define { { float, float }**, i32 }* @dragon(i32 %n) {
```

```
entry:
  %n1 = alloca i32
  store i32 %n, i32* %n1
  %n2 = load i32, i32* %n1
  %tmp = icmp eq i32 %n2, 0
  %if_tmp = alloca { { float, float }**, i32 }*
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %continue89, %then
  %if_tmp103 = load { { float, float }**, i32 }*, { { float, float }**, i32 }**
    %if_tmp
  ret { { float, float }**, i32 }* %if_tmp103

then:                                             ; preds = %entry
  %malloccall = tail call i8* @malloc(i32 mul (i32 ptrtoint (i1** getelementptr
    (i1*, i1** null, i32 1) to i32), i32 2))
  %arrdata = bitcast i8* %malloccall to { float, float }**
  %malloccall3 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint (float
    *
    getelementptr (float, float* null, i32 1) to i64), i64 2) to i32))
  %point = bitcast i8* %malloccall3 to { float, float }*
  %fieldaddr = getelementptr { float, float }, { float, float }* %point,
    i32 0, i32 0
  store float 0.000000e+00, float* %fieldaddr
  %fieldaddr4 = getelementptr { float, float }, { float, float }* %point,
    i32 0, i32 1
  store float 0.000000e+00, float* %fieldaddr4
  %storeref = getelementptr { float, float }*, { float, float }** %arrdata,
    i32 0
  store { float, float }* %point, { float, float }** %storeref
  %malloccall5 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
    (float*getelementptr (float, float* null, i32 1) to i64), i64 2) to i32))
  %point6 = bitcast i8* %malloccall5 to { float, float }*
  %fieldaddr7 = getelementptr { float, float }, { float, float }* %point6,
    i32 0, i32 0
  store float 1.000000e+00, float* %fieldaddr7
  %fieldaddr8 = getelementptr { float, float }, { float, float }* %point6,
    i32 0, i32 1
  store float 0.000000e+00, float* %fieldaddr8
  %storeref9 = getelementptr { float, float }*, { float, float }** %arrdata,
    i32 1
  store { float, float }* %point6, { float, float }** %storeref9
  %malloccall10 = tail call i8* @malloc(i32 ptrtoint ({ { float, float }**,
    i32 }* getelementptr ({ { float, float }**, i32 }, { { float, float }**,
    i32 }* null, i32 1) to i32))
  %arr = bitcast i8* %malloccall10 to { { float, float }**, i32 }*
  %arrdata11 = getelementptr { { float, float }**, i32 }, { { float, float
    }**,
    i32 }* %arr, i32 0, i32 0
  %arrlen = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %arr, i32 0, i32 1
  store { float, float }** %arrdata, { float, float }*** %arrdata11
  store i32 2, i32* %arrlen
  store { { float, float }**, i32 }* %arr, { { float, float }**, i32 }**
    %if_tmp
  br label %merge

else:                                             ; preds = %entry
  %d1 = alloca { { float, float }**, i32 }*
```

```
  %n12 = load i32, i32* %n1
  %tmp13 = sub i32 %n12, 1
  %fxn_result = call { { float, float }**, i32 }* @dragon(i32 %tmp13)
  store { { float, float }**, i32 }* %fxn_result, { { float, float }**, i32
    }** %d1
  %d2 = alloca { { float, float }**, i32 }*
  %d114 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %d1
  %fxn_result15 = call { { float, float }**, i32 }* @copy_path({ { float, float
    }**, i32 }* %d114)
  store { { float, float }**, i32 }* %fxn_result15, { { float, float }**, i32
    }** %d2
  %s = alloca float
  %fxn_result16 = call float @sqrt(float 2.000000e+00)
  %tmp17 = fdiv float %fxn_result16, 2.000000e+00
  store float %tmp17, float* %s
  %d118 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %d1
  %fxn_result19 = call float @toradians(float 4.500000e+01)
  %malloccall20 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
    (float* getelementptr (float, float* null, i32 1) to i64), i64 2) to i32))
  %anon = bitcast i8* %malloccall20 to { float, float }*
  %fieldaddr21 = getelementptr { float, float }, { float, float }* %anon,
    i32 0, i32 0
  store float 0.000000e+00, float* %fieldaddr21
  %fieldaddr22 = getelementptr { float, float }, { float, float }* %anon,
    i32 0, i32 1
  store float 0.000000e+00, float* %fieldaddr22
  %lenref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %d118, i32 0, i32 1
  %len = load i32, i32* %lenref
  %dataref = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %d118, i32 0, i32 0
  %data = load { float, float }**, { float, float }*** %dataref
  %i = alloca i32
  store i32 0, i32* %i
  br label %loop

loop:                                             ; preds = %loop, %else
  %i23 = load i32, i32* %i
  %elref = getelementptr { float, float }*, { float, float }** %data, i32 %i23
  %el = load { float, float }*, { float, float }** %elref
  call void @rotate({ float, float }* %el, float %fxn_result19, i32 -1, {
    float, float }* %anon)
  %i24 = add i32 %i23, 1
  store i32 %i24, i32* %i
  %i25 = load i32, i32* %i
  %tmp26 = icmp slt i32 %i25, %len
  br i1 %tmp26, label %loop, label %continue

continue:                                         ; preds = %loop
  %d127 = load { { float, float }**, i32 }*, { { float, float }**,
    i32 }** %d1
  %s28 = load float, float* %s
  %s29 = load float, float* %s
  %lenref30 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %d127, i32 0, i32 1
  %len31 = load i32, i32* %lenref30
  %dataref32 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %d127, i32 0, i32 0
  %data33 = load { float, float }**, { float, float }*** %dataref32
```

```
  %i34 = alloca i32
  store i32 0, i32* %i34
  br label %loop35

loop35:                                           ; preds = %loop35, %continue
  %i37 = load i32, i32* %i34
  %elref38 = getelementptr { float, float }*, { float, float }** %data33,
    i32 %i37
  %el39 = load { float, float }*, { float, float }** %elref38
  call void @scale({ float, float }* %el39, float %s28, float %s29)
  %i40 = add i32 %i37, 1
  store i32 %i40, i32* %i34
  %i41 = load i32, i32* %i34
  %tmp42 = icmp slt i32 %i41, %len31
  br i1 %tmp42, label %loop35, label %continue36

continue36:                                       ; preds = %loop35
  %d243 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %d2
  %fxn_result44 = call float @toradians(float 1.350000e+02)
  %malloccall45 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
    (float* getelementptr (float, float* null, i32 1) to i64), i64 2) to i32))
  %anon46 = bitcast i8* %malloccall45 to { float, float }*
  %fieldaddr47 = getelementptr { float, float }, { float, float }* %anon46,
    i32 0, i32 0
  store float 0.000000e+00, float* %fieldaddr47
  %fieldaddr48 = getelementptr { float, float }, { float, float }* %anon46,
    i32 0, i32 1
  store float 0.000000e+00, float* %fieldaddr48
  %lenref49 = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %d243, i32 0, i32 1
  %len50 = load i32, i32* %lenref49
  %dataref51 = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %d243, i32 0, i32 0
  %data52 = load { float, float }**, { float, float }*** %dataref51
  %i53 = alloca i32
  store i32 0, i32* %i53
  br label %loop54

loop54:                                           ; preds = %loop54, %continue36
  %i56 = load i32, i32* %i53
  %elref57 = getelementptr { float, float }*, { float, float }** %data52,
    i32 %i56
  %el58 = load { float, float }*, { float, float }** %elref57
  call void @rotate({ float, float }* %el58, float %fxn_result44, i32 -1,
    { float, float }* %anon46)
  %i59 = add i32 %i56, 1
  store i32 %i59, i32* %i53
  %i60 = load i32, i32* %i53
  %tmp61 = icmp slt i32 %i60, %len50
  br i1 %tmp61, label %loop54, label %continue55

continue55:                                       ; preds = %loop54
  %d262 = load { { float, float }**, i32 }*, { { float, float }**, i32
    }** %d2
  %s63 = load float, float* %s
  %s64 = load float, float* %s
  %lenref65 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %d262, i32 0, i32 1
  %len66 = load i32, i32* %lenref65
```

```
  %dataref67 = getelementptr { { float, float }**, i32 }, { { float, float
    }**, i32 }* %d262, i32 0, i32 0
  %data68 = load { float, float }**, { float, float }*** %dataref67
  %i69 = alloca i32
  store i32 0, i32* %i69
  br label %loop70

loop70:                                           ; preds = %loop70, %continue55
  %i72 = load i32, i32* %i69
  %elref73 = getelementptr { float, float }*, { float, float }** %data68,
    i32 %i72
  %el74 = load { float, float }*, { float, float }** %elref73
  call void @scale({ float, float }* %el74, float %s63, float %s64)
  %i75 = add i32 %i72, 1
  store i32 %i75, i32* %i69
  %i76 = load i32, i32* %i69
  %tmp77 = icmp slt i32 %i76, %len66
  br i1 %tmp77, label %loop70, label %continue71

continue71:                                       ; preds = %loop70
  %d278 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %d2
  %malloccall79 = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
    (float* getelementptr (float, float* null, i32 1) to i64), i64 2) to i32))
  %anon80 = bitcast i8* %malloccall79 to { float, float }*
  %fieldaddr81 = getelementptr { float, float }, { float, float }* %anon80,
    i32 0, i32 0
  store float 1.000000e+00, float* %fieldaddr81
  %fieldaddr82 = getelementptr { float, float }, { float, float }* %anon80,
    i32 0, i32 1
  store float 0.000000e+00, float* %fieldaddr82
  %lenref83 = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %d278, i32 0, i32 1
  %len84 = load i32, i32* %lenref83
  %dataref85 = getelementptr { { float, float }**, i32 }, { { float, float }**,
    i32 }* %d278, i32 0, i32 0
  %data86 = load { float, float }**, { float, float }*** %dataref85
  %i87 = alloca i32
  store i32 0, i32* %i87
  br label %loop88

loop88:                                           ; preds = %loop88, %continue71
  %i90 = load i32, i32* %i87
  %elref91 = getelementptr { float, float }*, { float, float }** %data86, i32 %i90
  %el92 = load { float, float }*, { float, float }** %elref91
  call void @trans({ float, float }* %el92, { float, float }* %anon80)
  %i93 = add i32 %i90, 1
  store i32 %i93, i32* %i87
  %i94 = load i32, i32* %i87
  %tmp95 = icmp slt i32 %i94, %len84
  br i1 %tmp95, label %loop88, label %continue89

continue89:                                       ; preds = %loop88
  %d296 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %d2
  call void @reverse({ { float, float }**, i32 }* %d296)
  %r = alloca { { float, float }**, i32 }*
  %d197 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %d1
  %d298 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %d2
  %fxn_result99 = call { { float, float }**, i32 }* @append({ { float, float }**,
    i32 }* %d197, { { float, float }**, i32 }* %d298, float 1.000000e+00)
```

```
  store { { float, float }**, i32 }* %fxn_result99, { { float, float }**, i32 }**
    %r
  %d1100 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %d1
  call void @free_path({ { float, float }**, i32 }* %d1100)
  %d2101 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %d2
  call void @free_path({ { float, float }**, i32 }* %d2101)
  %r102 = load { { float, float }**, i32 }*, { { float, float }**, i32 }** %r
  store { { float, float }**, i32 }* %r102, { { float, float }**, i32 }** %if_tmp
  br label %merge
}

define { float, float, float, float }* @rainbow(i32 %r, i32 %len) {
entry:
  %r1 = alloca i32
  store i32 %r, i32* %r1
  %len2 = alloca i32
  store i32 %len, i32* %len2
  %h = alloca float
  %r3 = load i32, i32* %r1
  %cast = sitofp i32 %r3 to float
  %tmp = fmul float 1.000000e+00, %cast
  %len4 = load i32, i32* %len2
  %cast5 = sitofp i32 %len4 to float
  %tmp6 = fdiv float %tmp, %cast5
  store float %tmp6, float* %h
  %h7 = load float, float* %h
  %fxn_result = call { float, float, float, float }* @hsv(float %h7, float
    0x3FE99999A0000000, float 0x3FE99999A0000000)
  ret { float, float, float, float }* %fxn_result
}
```