Programming Languages and Translators

ART

Animation Rendering Tool

Bret Jervey - baj2125
Gedion Metaferia - gym2103
Natan Kibret - nfk2105
Soul Joshi - srj2120

December 20, 2016
1 Introduction

The ART programming language is a general-purpose, statically typed, imperative language focused on vector based 2D animation rendering. Its design combines imperative programming with simple object oriented programming in order to provide developers a simple interface for rendering animations. It has an concise syntax derived from C. However, some syntactic choices are more similar to Java and C++. ART has automatic storage allocation making it convenient for use.

ART utilizes the OpenGL utility toolkit, GLUT, in order to render and update the visual representations of objects. GLUT is an API interface for OpenGL that is geared towards the construction of small to medium sized OpenGL programs.

In this report, we will provide a simple tutorial focusing on the main components of the language followed by the Language Reference Manual. The tutorial and Language Reference Manual will be the core documentation for new programmers to refer to. After these, we will provide the process followed while building the language, the architectural design of the language, and the test plan that allowed us to check the accuracy of the language. We have also included important lessons learned during the construction of the language for future language creators, and appendices that include the source codes of our language.

1.1 What sorts of Programs are meant to be written in ART?

We envision 2D animation programs to be written using ART. Developers utilizing ART will control where objects are drawn and how their movements are simulated.

Below are some examples of how ART can be used:

- **To model the planetary system**: With the help of ART, a developer can simulate the solar system as well as any hypothetical situation where the physical properties of the sun and the planets (such as mass) are different.
- **As an interface to simulate properties of physics**: If we consider a physics professor that is trying to simulate the motion of a free falling ball with given mass and gravitational pull, said professor can use ART to simulate the motion of the ball so their students can see the laws of physics in action.
- **As a visualization tool for Algorithms**: ART can be utilized to represent algorithms in two dimensional animations. Some sample algorithm represented through ART include, the Towers of Hanoi, tree traversals and sorting algorithms.

2 Tutorial

2.1 Using the Compiler

- In Terminal, navigate to the src directory.
- Here, type `make` in order to make the ART compiler that can read in your ART program source file.
- Then, type `bash compile path_to_file/name_of_file.art` to compile the program. If you see no errors, then your program has compiled.
• If you would like to check the output of your program, type the following bash command instead: `bash compile path_to_file/name_of_file.art abc`
Following this, type `./abc` to check your output.

Let us look at a sample code that previews the mandatory `main` function. Inside the main function, we will call the built-in `prints` function which takes an argument of type `string`.

**Example 1:** Compilable program called helloworld.art

```plaintext
1. int main()
2. {
3.     string s = "Hello World?";
4.     prints(s);
5. }
```

You can compile the above program as follows if you are in the same directory:

```
$ bash compile path_of_file/name_of_file.art abc
```

To view the output, type the following:

```
$ ./abc
Hello world?
```

### 2.2 Primitives

ART’s declaration and assignments work as follows:

- A newly-declared variable is preceded by its type such as, `char, double, int, string` and `vec`
- A declaration of a variable can be written as `vec v1;` or can be initialized with a value like `string s = “hello”;` If the assignment for a variable is done on the same line as the initialization, we will consider that to be part of the declaration.
- List declarators are acceptable as in `int i, x;` which declares both i and x.

**Example 2:** Declaration and assignment of primitives.

```plaintext
1. int main() // this is how you write a comment
2. /*this is another way of writing a comment*/
3. {
4.     string s = "Hello";
5.     vec v1;
6.     vec v2;
7.     int i, x;
8.     char c = 'Q';
9.     v1 = <10.5, 12.5>;
10.    i = 5;
11.    x = 7 + 1;
12.    prints(s); printc(\n');
13.    printi(i); printc(\n');
14.    printi(x); printc(\n');
15.    printc(c); printc(\n');
```
The above program has the following output:
Hello
5
8
0
10.5
12.5

2.3 Derived Types

2.3.1 Arrays

Arrays are indexed contiguous sequences of objects of a given type. Elements can be of any primitive or derived types.

- The dimensions and size of an array are fixed during declaration.
  - The size of the outer most dimension can be omitted but that requires the use of a full array initializer to be provided from which the dimension can be inferred.
- Partial array initializers (ending with a comma) can be provided for arrays that have fully defined size. The parts of the array for which the partial initializer doesn’t provide values are zeroed (see f in line 15 in Example 3).

Example 3: Declaration, assignment and manipulation of Arrays

```c
1. /*Array example*/
2. // function to print an integer, then skip a line
3. void println(int x)
4. {
5.     printi(x);printc('\n');
6. }
7. 
8. int main()
9. {
10.    string s = "hello";
11.    vec v1;
12.    int[] z = {1, 2, 3} , w = {4, 3, 5};
13.    int[4] b = {1,2,3,4} , f = {1};
14.    int[][3] a = { {1,2,3}, {3,4,5}};
15.    a[1][2] = 10
16.    println (w[0]);
17.    println (f[2]); println (f[3]);
18.    println (a[0][1]); println (a[1][2]);
19. }
20. }
```
The above program has the following output:
4
0
0
2
10

2.3.2 Struct

A struct is a sequence of named members of various types and a set of associated member functions (methods).

- The name of a struct must be unique and along with keyword struct, forms the type-declaration for that specific struct.
- The body of a struct contains any number of variable declarations with a type and a name that belong within the scope of the structure.
- A variable cannot be assigned a value in the struct definition.
- Elements of a struct can be of any type and do not need to be of the same type.
- A struct cannot have a member variable to itself.

One can access a struct’s variables via dot notation.

Note: A default struct called **color** is built-in for the convenience of the user. It is defined as:

```c
struct color
{
    double r, g, b;
}
```

This struct can be used as a member variable of struct/shapes, and can be used in the **setColor** built-in function to specify the color of the shapes being rendered. Shapes are explained in the next section.

**Example 4:** Declaration, assignment and manipulation of structs

```c
1. /*test to see if strings can be assigned from inside a struct*/
2. 
3. struct stringPair {
   4.     string s1;
   5.     string s2;
   6. }
7. 
8. struct intCharStruct {
   9.     int x;
10.     char y;
11.     struct stringPair z;
12. }
13. 
14. int main ()
15. {
```
The above program has the following output:

77
c
string inside a struct
string inside a struct
CHANGED string inside a struct

2.3.3 Shape

Shapes follow all the same conventions of a struct. However, shapes require a draw member function to be defined. A draw method must have return type void and takes no arguments.

Example 5: Declaration, assignment and manipulation of shapes

```c
/* Test for Shapes */
shape circle
{
    int radius;
    int px;
    int py;
    int points;
}
circle::circle(int r, int x, int y, int p)
{
    radius=r;
    px=x;
    py=y;
    points=p;
}
// below is the required draw member function
```
void circle::draw()
{
    int i, slice, angle, npx, npy;
    slice = 2 * points;
    for (i = 0; i < points; i += 1)
    {
        angle = slice * i;
        npy = py + radius * angle;
        npx = px + radius * angle;
        printi(npx);
        printi(npy);
    }
}

int main()
{
    shape circle cir, cir2;
    cir = circle(5, 3, 6, 20);
    cir2 = circle(10, 7, 9, 20);
    cir.draw();
    cir2.draw();
}

While this program will compile, there is no Timeloop here so the call to the draw method will have no effect. Therefore, this above program does not have an output.

2.4 Functions

Functions allow for flexible modular programming.

2.4.1 Global Functions

Global functions are declared as follows:

    type function_name ( parameter_list opt )
    {
        Function body
    }

Global functions are accessible from the global scope. The following example demonstrates how to define a function and call it.

Example 6: Defining and calling a global function

1. // function to print an integer, then print a character
2. void printic(int x, char c)
3. {
4.     printi(x); printc(c);
5. }
6.
7. int main()
8. {
9.     printic(1, ‘a’);
The above program has the following output:

```
1a2b
```

### 2.4.2 Methods (Member Functions)

Methods are functions that belong within the scope of a struct/shape. They are defined as follows:

```c
    type struct_name::function_name ( parameter_listopt )
    {
        Function body
    }
```

Inside a body of a struct member function, the struct's member names (both member variables and member functions) can be accessed without qualifiers. The members' variable names used in the methods correspond to the particular instance of the struct and member function calls are applied on the instance.

A special case is the constructor for the struct/shape which returns a newly created instance of the struct/shape, and is defined as follows:

```c
    struct_name::struct_name ( parameter_listopt )
    {
        Function body
    }
```

The following example demonstrates how to define a member function and a constructor and call them.

**Example 7:** Defining and calling a method (member function) and a constructor

```c
1. /*Member functions (Methods) Example*/
2. 
3. struct point
4. {
5.     int x;
6.     int y;
7. }
8. 
9. // define the constructor of struct point
10. point::point(int a, int b)
11. {
12.     x = a;
13.     y = b;
14. }
15. 
16. // define the method print_test_point() within the struct point
17. void point::print_test_point()
18. {
19.     printi(x);
20.     printi(y);
```
The above program does not have an output because the drawing commands (time loops) are not included, but it does compile.
2.5 Control Flow

ART utilizes if/else conditionals, for loops, while loops, as well as return statements the same way as in C. However, there are added control flow tools that are used for animation rendering. These tools are timeloop and frameloop.

Animation Loops:

The two Animation Loops are timeloop and frameloop. They will be discussed as subsections below, but first let us list some essential factors for generating animation

- Use addshape to add shapes to register shapes to be drawn on the next timeloop/frameloop.
- Drawing both static and moving images requires the use of timeloop or frameloop.

2.5.1 Time loop:

Time loop is an animation specific control flow. The following are the key components of timeloop:

- In the first expression, assign a render period (in seconds) to dt (dt is simply an identifier, you can use any variable here).
- In the second expression, assign a total time that the animation runs for (in seconds) to end (end is simply an identifier, you can use any variable here).
- If we are drawing a static image (without any movement) the contents of the timeloop end here (no statements within the timeloop). If we are rendering a moving image, the contents of the timeloop will have statements that use dt and end, but any changes made to them are reset on the start of the next iteration.

Example provided after Frame Loop is explained.

2.5.2 Frame loop:

Frame loop is an animation specific control flow. The frame loop is very similar to the time loop with three key differences:

- In the first expression, assign the number of frames rendered per second to fps (fps is simply an identifier, you can use any variable here).
- In the second expression, assign the total number of frames for the animation to frames (frames is simply an identifier, you can use any variable here).
- During smooth animation rendering, the effects of frameloop is the same as timeloop. However, during heavy load animation rendering, the time loop prioritizes the time requirements of the animation while the frame loop guarantees the number of frames rendered.

2.5.3 Drawpoint and Context Markers:

drawpoint is a builtin function that takes a vector parameter that contains the position of the point and passes it to the animation renderer. drawpoint is defined as follows: drawpoint (vec).
Contexts are used to dictate how the `drawpoint` call is interpreted. When outside the scope of a context marker, the program will be in point context therefore all `drawpoint` calls will draw individual points. When the `<< >>` context marker is used, three consecutive `drawpoint` calls specify the vertices of a triangle, and when the context marker `[ ]` is used two consecutive `drawpoints` calls create a line. Contexts can be nested but when a new context is created it closes the old one.

The following example is long, but is essential to show how one can successfully render a simple animation. Please pay attention to the comments as they are made very extensive intentionally.

**Example 8: Rendering a Static and Moving Image**

```cpp
1. /* Timeloop Example */
2. 
3. struct color color;
4. 
5. /* A circle with origin at x,y with radius r */
6. 
7. shape circle // a shape is declared, therefore, there must be a draw method
8. {
9.   double r;
10.  vec o;
11. }
12. 
13. circle::circle(vec origin, double radius)
14. {
15.   o = origin;
16.   r = radius;
17. }
18. 
19. // This is the function from the example
20. 
21. void circle::draw() // this is the draw method required for shape circle
22. {
23.  setcolor(color);
24.  
25.  // Single circle is 1000 points YIKES
26.  [  // Notice that this instates a circle context for drawpoint
27.  // Lines can be used to draw curves
28.  for (double theta = 0.0; theta <= 2 * PI; theta += .05 * PI) {
29.    drawpoint(o + r*cos(theta - .05*PI), sin(theta-0.05*PI));
30.    drawpoint(o + r*cos(theta), sin(theta));
31.  }
32.  
33.  drawpoint(o);
34. }  
35. } 
36. 
37. shape disk
38. {
39.   double r;
40.   vec o;
41. }
42. 
43. disk::disk(vec origin, double radius)
44. {
45.   o = origin;
46.   r = radius;
47. }
48. 
```
void disk::draw()  // this is the draw method required for shape disk
{
    setcolor(color);
    <<  // Notice that this instates a triangle context for drawpoint
    // Triangles can be used to draw surfaces
    for (double theta = 0.0; theta <= 2 * PI; theta += .05 * PI) {
        drawpoint(o);
        drawpoint(o + r*<cos(theta - 0.05*PI), sin(theta-0.05*PI)>);
        drawpoint(o + r*<cos(theta), sin(theta)>);
    }
    >>
}

// this is a path shape
// a drawable circular buffer of points
shape path
{
    int f, b;
    vec[128] p;
}

path::path() { f = 0; b = 0; }

int MOD(int x)
{
    return x % 128;
}

void path::draw()
{
    setcolor(color);
    
    if ( f != b)
        for (int i = MOD(f) + 1; MOD(i) != MOD(b); i++)
        {
            drawpoint(p[MOD(i-1)]);
            drawpoint(p[MOD(i)]);
        }
    
}

void path::add_point(vec x)
{
    p[MOD(b)] = x;
    b+=1;

    // if we run out of space
    if (MOD(b) == MOD(f))
        f+=1;
}

void path::reset() { f = b = 0;}

int main()
{
    shape circle c = circle(<0.0, 0.0>, 0.25);
    shape disk d = disk(<0.0, 0.0>, 0.15);
    shape path p = path();
    double theta = 0.0;
    vec o = <0.0,0.0>;

    addshape(c);
    addshape(p);
% 114. // first in red
% 115. color.r = 1.0; color.b = color.g = 0.0;
% 116. printf(theta);
% 117. printc(’\n’);
% 118. //this time loop runs for 10 secs and excutes body every millisec.
% 119. timeloop (dt = 0.001 ; end = 10.0) {
% 120. //produces static image
% 121. }
% 122. printf(theta);
% 123. printc(’\n’);
% 124. // then in blue
% 125. color.b = 1.0; color.r = color.g = 0.0;
% 126. // Readd shapes since they removed after the end of a time loop
% 127. addshape(d);
% 128. addshape(p);
% 129. p.reset();
% 130. //This frameloop runs around the same time as the above time loop
% 131. frameloop (fps = 1000; frames = 10000) {
% 132. theta += .010 * PI;
% 133. d.o = o + 1.3 * d.r * <cos(theta),sin(theta)>;
% 134. p.add_point(d.o);
% 135. }
% 136. return 0;
% 137. }

The following window (Figure 1) will open for 10 seconds to show a static image:

![Figure 1: a static circle rendered from Example 8](image-url)
Then the following window (Figure 2) opens for 10 seconds. Here we have a static image as a preview, but the result is a blue disk which rotates in a circular path and is followed by a curved line which appears to be attached to its center. Here is a link to the gif file which shows the animation: http://giphy.com/gifs/l4JyVCqIYiL3VkgqA

Figure 2: a moving disk and tail rendered from Example 8

3 Language Reference Manual

3.1 Lexical Conventions

An ART program consists of a single source file with function, method, shape and struct definitions and variable declaration. All programs must define a main function which serves as an entry point to the program.

3.1.1 Tokens

The language is composed of the following types of tokens: identifiers, keywords, literals, operators and other separators.

3.1.2 Comments

ART allows for both block and single line comments. The characters /* introduce a block comment, which terminates with the characters */. Block comments do not nest. The characters
// introduce a line comment which terminates at a new line character. Comments can not occur within character literals. Example below:

    /* double line comment here
    Double line comment continues here*/
    // single line comment here

3.1.3 Identifiers

A valid identifier consists of any sequence of letters (an underscore counts as a letter) and digits. An identifier cannot begin with a digit. They can be of any length and case. Identifiers are case sensitive; for example, "abc" is not the same as "Abc".

3.1.4 Keywords

The language has the following reserved words that may not be used for any other purpose:

User Defined Structures:
    struct
    shape

Control Flows:
    timeloop
    frameloop
    for
    while
    if
    else
    return

Types:
    char
    double
    int
    void
    vec
    string

3.1.5 Literals

An integer literal can be one of the following:
- A decimal literal: a sequence of digits that does not begin with a zero.
- An octal literal: a sequence of digits that begins with zero and is composed of only the digits 0 to 7.
- A hexadecimal literal: '0X' or '0x' followed by a sequence of case insensitive hex digits.
Character literals can be one of the following:

- A printable character in single quotes. E.g: `x'
- One of the following escape sequences: `\n', `\t', `\v', `\b', `\r', `\f', `\a', `\", `\?', `\', `\', `\''`
- A backslash followed by 1,2 or 3 octal digits in single quotes: E.g: `\0'
- A backslash followed by the letter x and a sequence of hex digits in single quotes: E.g: `\x7f'

For the hex and octal escape sequences, the behavior is undefined if the resulting value exceeds that of the largest character.

A double literal consists of an integer part, a decimal point, a fraction part, an \texttt{e} or \texttt{E}, and an optionally signed integer exponent. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the \texttt{e} and the exponent (not both) may be missing.

A vector literal consists of two floating literals separated by a comma enclosed by a matching set of angular brackets(\texttt{< >}). Any white space separating these components is ignored.

### 3.2 Meaning of Identifiers

Identifiers can be used to refer to functions, structures, members of structures and variables.

A brief description of types:

#### 3.2.1 Basic Types

Integers (**int**)

An integer is a 32-bit signed 2's complement series of digits with the maximum range of 2147483647.

Doubles (**double**)

A double is a 64-bit double precision number.

Characters (**char**)

Characters occupy 8-bits and come from the ASCII set of characters.

Vector (**vec**)

A vector is a tuple of two doubles. The components can be accessed with the indexing operator `[]`.

Void (**void**)

The void type is used to declare a function that returns nothing.

### 3.2.2 Derived Types

All types, with the exception of **void**, can be used to define the following derived types:
Arrays

Arrays are contiguous sequences of objects of a given type. They can be declared as:

\textbf{type-name \{\}}(note this needs an intitalizer) or \textbf{type-name [size]}, where \textbf{size} is a constant expression.

Structures (\textbf{struct})

A structure is a sequence of named members of various types and a set of associated member functions (methods).

Shapes (\textbf{shape})

Shapes are structures that need to implement a \textbf{draw} method.

\section*{3.3 Conversion}

Explicit type casting is not allowed in the language. And the only conversion that occurs is the promotion of an integer to the equivalent double value when an integer is provided where a double is expected. This includes arithmetic operations between int and double types and assignment of an int to a variable of type double. Therefore, if int is used where double is expected, it will be automatically promoted to a double.

\section*{3.4 Operators}

The following operators are allowed in the language:

\begin{itemize}
  \item \textbf{+} addition
  \item \textbf{-} subtraction
  \item \textbf{*} multiplication
  \item \textbf{/} division
  \item \textbf{%} modulo
  \item \textbf{<} less than
  \item \textbf{>} greater than
  \item \textbf{<=} less than or equal to
  \item \textbf{>=} greater than or equal to
  \item \textbf{==} equivalence
  \item \textbf{!=} inequality
  \item \textbf{=} assignment
  \item \textbf{+=} plus assignment
  \item \textbf{-=} subtraction assignment
  \item \textbf{/=} division assignment
  \item \textbf{*=} multiplication assignment
  \item \textbf{++} increment operator (prefix and postfix)
  \item \textbf{--} decrement operator (prefix and postfix)
  \item \textbf{+} unary plus
  \item \textbf{-} negation
\end{itemize}
3.4.1 Arithmetic Operators

+  addition  
-  subtraction  
*  multiplication  
/  division  
%  modulo

ART supports the basic arithmetic operators: addition, subtraction, multiplication, division and modulo (remainder operator). The left and right operands for the operators have to be the same type after int to double conversion if necessary. The exceptional case is for vector double multiplication and division. The operators evaluate to the same type as their operands.

Integers can be used with all the arithmetic operators. The division operator performs integer division (decimal truncated from result). The meaning of modulo operator for integers is such that:

\[
a == (a / b) \times b + (a \% b).
\]

Doubles can be used with all arithmetic operators except modulo. Vectors can be used with all arithmetic operators except modulo. Addition, subtraction, multiplication and division between vectors results in a component wise sum, subtraction, multiplication and division. Vectors can be multiplied with and divided by doubles with the same meaning as vector-scalar multiplication. The operations take the following form: \texttt{scalar * vector}, \texttt{vector * scalar} and \texttt{vector / scalar}. The two forms for multiplication are equivalent.

All arithmetic operations are left-associative with multiplication, division and modulo having higher precedence than addition and subtraction.

3.4.2 Relational operators

<  less than  
>  greater than  
\leq  less than or equal to  
\geq  greater than or equal to

The relational operators include: less than, greater than, less than or equal to, and greater than or equal to. These operators can only be applied to types int and double. Since there is no boolean type, the operators return 0 for false and 1 for true.
3.4.3 Equality Operators:

\[
\begin{align*}
== & \quad \text{equivalence} \\
!= & \quad \text{inequality}
\end{align*}
\]

The equality operators include equivalence and inequality. They can be used with integers, doubles and vectors. Like relational operators they return 0 for false and 1 for true.

3.4.4 Assignment operator:

\[
\begin{align*}
= & \quad \text{assignment} \\
+= & \quad \text{plus assignment} \\
-= & \quad \text{subtraction assignment} \\
/= & \quad \text{division assignment} \\
*= & \quad \text{multiplication assignment} \\
%= & \quad \text{modulo assignment}
\end{align*}
\]

The assignment operators are principally the basic assignment operator = and the compound assignment operators with the form \( op= \) where \( op \) is one of the arithmetic operators.

The basic assignment operator stores the value of the right operand in the memory location corresponding to the left operand. This implies the left operand must be an expression that refers to an object in memory. Moreover, the left and right operands must be of the same type after the necessary promotions.

The meaning of a compound assignment operator ‘\( l \ op= r \)’ is the same as ‘\( l = l \ op \ r \)’ but with the expression ‘\( l \)’ evaluated only once. Moreover the operation ‘\( l \ op \ r \)’ must be defined.

The assignment operators are right associative operators.

3.4.5 Increment/decrement operators:

\[
\begin{align*}
++ & \quad \text{increment operator (prefix and postfix)} \\
-- & \quad \text{decrement operator (prefix and postfix)}
\end{align*}
\]

These operators are a shorthand form of the expression \( x=x+1 \) or \( x=x-1 \) but with the expression \( x \) evaluated only once. They both have prefix and postfix forms. The prefix form \( ++x \) evaluates to the value of \( x \) after incrementing. The postfix form \( x++ \) evaluates to the value of \( x \) before incrementing. The same holds true for prefix and postfix decrement. Neither of these expressions can be used on the left side of an assignment. These operators are defined only for integers.

3.4.6 Unary Operators:

\[
\begin{align*}
+ & \quad \text{Unary Plus}
\end{align*}
\]
- negation
! logical not

Includes unary plus, negation and logical not. Unary plus and negation can take int, double and vec types as operands. Logical not applies only to int types.

The negation operator is equivalent to multiplying by negative. For vectors, this implies component wise negation. The unary plus operator is added for symmetry.

Logical not results in 0 for non-zero values and a 1 when applied to 0.

3.4.7 Logical Operators:

|| logical OR
&& logical AND

These are the OR and AND operators. The operators take in two int operands and return 0 or 1. OR returns 0 if both operands are 0, and 1 otherwise. AND returns 1 if both operands are 1 and 0 otherwise. Both operators are short-circuited. That implies operands are evaluated left to right and the rightmost operand is not evaluated if the result can be determined from the leftmost one.

3.4.8 Subscript Operator:

[] subscript

The subscript operator is used to access individual objects in arrays and the components of vectors at the given index (eg. x[5] gets the object at index 5). Accessing an index higher than the number of objects in a given array (which can be any number) or vector (always size 2) results in undefined behavior.

3.4.9 Scope Operator:

:: scope

The scope operator is only used for a member function (method) definition where the left operand is an identifier for a struct or shape type name and the right is the method name.

3.4.10 Pass by Reference Operator:

& pass by reference

The pass by reference is a special operator which is used only in the argument list of a function definition. When this operator is applied to a valid type, the argument passed is passed by reference
rather than by value as ART is a pass-by-value language by default. In this way, the argument is not copied but directly used by the function. This operator cannot be used anywhere else. L-value expressions, which represent objects defined by the user, are the only values that can be passed by reference. L-values include: global or local variables, members of arrays, member variable of struct instances/ shape instances that themselves are L-values. Constructor calls are not L-values.

3.5 Functions

3.5.1 Function Definition

A basic function follows this format:

```c
    type name(parameter_list opt)
    {
        Function body
    }
```

The type signifies the return type of the function which can be any of the basic types or struct/shape types. Arrays cannot be used as the return type of a function. Functions that don’t return a value are defined with return type void. The name of the function needs to be unique.

The parameter list can contain any number of parameters in the form of type name, where type is any valid type excluding void and name is any valid identifier which is used by the function to access the argument’s value.

The parameter can also have the pass-by-reference operator (which is appended to the type) that signals to the function to pass arguments by reference rather than by value (ART is by default a pass-by-value language). The pass-by-reference means that the argument is not copied but the argument’s name acts as its alias. The pass-by-reference operator can only appear in the function definition and not the function call and only lvalues can be passed in as the argument for a pass-by-reference parameter.

```c
    void exampleByRef(int& x) { x = 7; }
    void exampleByValue(int x) { x = 10; }

    int x = 5;
    exampleByValue(x); // x is unchanged
    exampleByRef(x);   // x is now set to 7
    exampleByValue(3); // can do this
    exampleByRef(3);   // compiler error
```

The function body is the actual code that is executed when a function call is performed. The expression in the return statement must match the return type.
3.5.2 Function Calls

Function calls are in the form of \texttt{name (argument\_list)} where \texttt{name} is a name of a function that has been previously defined. The length and types of arguments in the argument list of a function call must match exactly the length and types of the parameter list in the function definition. The arguments of a function are evaluated from left to right.

A function call (to a non-void function) will evaluate to a value of the type declared in the definition. This value is a copy of the value of the expression in the corresponding \texttt{return} statement in the function body. A function call to a \texttt{void} function has no value or equivalently has value \texttt{void}.

3.5.3 Builtin-functions/definitions

\texttt{drawpoint(vec)}

\texttt{drawpoint} takes a vector parameter that contains position of the point and passes it to the animation renderer. The effects depend on contexts (discussed below).

\texttt{addshape(shape)}

Add takes one argument of a shape type and adds to the list of shapes to be drawn by the animation loops.

\texttt{addshape(shape, ...)}

This version of \texttt{addshape} works the same way as the single parameter version but can take in any number of shape arguments.

\texttt{setcolor(struct color)}

\texttt{setcolor} takes one argument of type \texttt{struct color} which contains values corresponding to the red, green and blue color values, respectively. It sets the color of the shape being rendered.

ART also includes several math and printing functions/definitions which are detailed in the Appendix.

3.6 Structs and Shapes

3.6.1 Struct Definition

A \texttt{struct} definition follows this format:

\begin{verbatim}
struct name
{
    type member\_names;
\end{verbatim}
The name of the \texttt{struct} must be unique and along with keyword \texttt{struct}, forms the type-declaration for that specific struct.

\begin{verbatim}
struct point { int x; int y}

struct point pt1; // declares variable pt1 of type struct point
\end{verbatim}

The body of the \texttt{struct} contains any number of variable declarations with a type and a name and belong within the scope of the structure. A variable cannot be assigned a value in the struct definition.

3.6.2 Member Access

The way to access variables and methods of struct is by using the post-fix dot notation expression as illustrated in the following example

\begin{verbatim}
struct point pt1; // variable pt1 with type struct point

pt1.x = 1; // variable x in instance of struct point pt1 has value of 1

pt1.y = 2; // variable y is set to 2
\end{verbatim}

3.6.3 Defining Member Functions

A member function (method) is a function that belongs within the scope of a \texttt{struct} and is defined as:

\begin{verbatim}
<type> <struct-name>::<function name>(parameter-list)
{ function body }
\end{verbatim}

Since a member function is in the scope of a \texttt{struct}, it can directly refer to the struct’s variables in its body. It can also call other member functions. The member variables/functions referred to in a method body correspond to the member variables of the object on which the method is called. In other words the struct variable is an implicit argument to the member function.

Member function calls are written in the format of:

\begin{verbatim}
<variable of type struct>..<name of member function>(parameter-list).
\end{verbatim}

Example:

\begin{verbatim}
struct point { int x; int y};
\end{verbatim}
vec point::getPoint(){ vec temp; temp[0] = x ; temp[1] = y; return temp}

struct point pt1
pt1.x= 1;
Pt1.y=1;

pt1.getPoint(); // returns a vectors with components 1,2

Note: A default struct called color is built-in for the convenience of the user. It is defined as:

    struct color
    {
        double r, g, b ;
    }

This struct can be used as a member variable of shapes, and is required by the setcolor built-in function to specify the color of the shapes being rendered.

3.6.4 Defining Constructors and Struct Initialization

A constructor is a special method that initializes and returns an instance of a structure. A constructor has the same name as the struct it returns and hence a constructor definition has no return type. This also implies that there is only one constructor as function overloading is not supported.

The body of constructor has access to the members of the struct like other methods. A constructor call creates a new object and the body of the constructor is executed with the newly created object provided as an implicit argument. The body of the constructor does not have a return statement.

A constructor is called as if it were a function that had the same name as the struct. A constructor call evaluates to a newly initialized object and can be used anywhere an expression of the struct type is legal. For example, it can be assigned to other struct variables, passed to functions/methods and returned from functions/methods.

    point::point(int pt1, int pt2t) { x = pt1; y =pt2; }

    struct point pt = point(5,6); // pt has its x variable set to 5 and y to 6

The other way to initialize a struct is to list the values in braces with the same number of listed values as fields in the structure being initialized

    struct point { int x; int y;}

    struct point pt1 = {1,2}; /* x and y in struct point are now 1 and 2 respectively */
If a variable in a `struct` is not initialized there is no guarantee to what the variable will contain as a value.

Structures can also be nested. The list initializers can be nested to initialize structs with nested structs.

```c
struct rectangle { struct point top, bottom; }

struct rectangle r1 = { {1,2}, {3,4} }
```

Aside: List initializers can also be used to initialize arrays. The nested form of list initializes can be used to initialize arrays of arrays.

### 3.6.5 Shapes:

Shapes follow all the same conventions of structure but have the additional requirement of needing to have a `draw` member function defined.

```c
shape circle{ vec center; double radius} // creates new shape circle
```

The `draw` member function dictates how the shape will be drawn when used in a `timeloop` or `frameloop`.

The `draw` function usually contains either the logic that creates the values that are passed into `drawPoint` or calls the `draw` methods of its member shapes.

### 3.7 Statements

Statements are the basic units of executions. The following types of statements are defined:

- `expression-statement`
- `compound-statement`
- `selection-statement`
- `iteration-statement`
- `jump-statement`

#### 3.7.1 Expression Statements

These are statements of the form `expression;`. The value of the expression is evaluated and any side effects the expression may have takes effect before the next statement begins.

#### 3.7.2 Compound Statements

Compound statements have the following form:

```c
{ declaration-statement-list }
```
The form of compound statements implies that variables declarations have to come before any statements in blocks (as well as function bodies).

The variables that are defined in a block only exist and are accessible within the body of the block after the point in which they are defined. This is elaborated further in the scopes and declarations section.

**Context:**

The following compound statements also define contexts:

```
[ declaration-statement-list ]
<< declaration-statement-list >>
```

Contexts are used to dictate how the `drawpoint` call is interpreted. When outside the scope of a context marker, the program will be in point context therefore all `drawpoint` calls will draw individual points. When the `<< >>` context marker is used, three consecutive `drawpoint` calls specify the vertices of a triangle, and when the context marker `[ ]` is used two consecutive `drawpoints` calls create a line. The context markers also delineate blocks just as the `{ }`. Contexts can be nested, that includes lexical nesting as well as nesting across function calls. That means that if you call a function inside one context that function can instantiate a new context. This is possible because context define program states which can be carried along function calls. When a new context is created the old one is closed. When this new context closes, the default context (point context) is reinstated. However, the scope defined by the context markers of the outer context is still intact. This discrepancy arises because scoping is handled at compile time but context states is maintained at runtime.

Example:

```
void draw_line()
{
    // This marker opens a line context and closes the block context
    [ int x = 3; // variable in the block delineated by the context marker
    // draw a line segment with the specified endpoints
    drawpoint(<0.0,0.0>);
    drawpoint(<0.1,0.1>);
]
    // block terminates and hence x is not defined
}

void draw_something()
{
    // start a triangle context
    << int x;
    // draws the triangle formed by these three vertices
    drawpoint(<0.0,0.0>);
    drawpoint(<0.1,0.1>);
```
3.7.3 Selection Statements

The selection statement has the following forms in the language:

\[
\text{selection-statement:} \\
\quad \text{if ( expression ) statement} \\
\quad \text{if ( expression ) statement else statement}
\]

\text{if else statement:}

The language supports if else statements as selection statements.

If the expression, which must be of type \text{int}, evaluates to a non-zero value, the first substatement (the \text{if} statement) is executed. The second substatement (the \text{else} statement) is executed if the expression is evaluated to zero. Nesting of the if else statements is also supported.

To resolve the dangling else ambiguity, the else is associated to the nearest \text{if}.

3.7.4 Iteration Statements

Iteration statements specify loops. Iteration statements have the following forms in the language:

\[
\text{iteration-statement:} \\
\quad \text{while ( expression ) statement} \\
\quad \text{for ( expression; expression; expression) statement} \\
\quad \text{for ( declaration expression; expression) statement} \\
\quad \text{timeloop ( identifier = expression; identifier = expression) statement} \\
\quad \text{frameloop ( identifier = expression; identifier = expression) statement}
\]

\text{while statement:}

The language supports \text{while} statements as iteration statements.
In the `while` statement, the expression specifies a test. The substatement is executed repeatedly as long as the value of the expression, which must be of type `int`, is not equal to `zero`. The test, including all side-effects of the expression, takes place before each execution of the statement.

**for statement:**

The language supports `for` statements as iteration statements.

In the first form of the `for` statement, the first expression, which can be of any type, is evaluated only once, and specifies initialization for the loop. The second expression, which must be of type `int`, specifies a test which is evaluated before each iteration of the loop. The for loop is terminated if the second expression evaluates to zero. The third expression, which can be of any type, specifies a re-initialization for the loop as it is evaluated at the end of each iteration. Typically, the third expression specifies an incrementation.

The second form simply substitutes the first expression for a declaration. The variables defined in the declaration are local to the loop scope.

The first form of the for statement is equivalent to:

```
expression1 ;
while ( expression2 )
{
    statement
    expression3 ;
}
```

**Example:** For loop that corresponds to the first form (declaration outside the forloop).

```c
1. void printiln(int x)
2. {
3.     printi(x);printc("\n");
4. }
5. int main()
6. {
7.     int i;
8.     for(i = 0; i < 10; i = i + 1)
9.     {
10.    printiln(i);
11. }
12. }
```

The second form is equivalent to:

```
{
    declaration
    while ( expression2 )
    {
        statement
        expression3 ;
    }
}
```
Example: For loop that corresponds to the second form (declaration in the for statement):

```c
1. void println(int x)
2. {
3.     println(x);printc(‘\n’);
4. }
5. int main()
6. {
7.     for(int i =0; i < 10; i = i + 1)
8.         println(i);
9. }
10. }
```

Any of the three expressions (including the declaration) may be dropped. A dropped second expression makes the implied test equivalent to testing a non-zero constant, which results in an infinite loop.

**Time loop:**

Time loop is one of the two animation specific control flow statements. In the first expression, the variable `dt` of type double (`dt` is simply an identifier, any variable can be used here) is assigned a value that represents the render period (in seconds) that the animation is using. The second expression sets the variable `end` of type double (`end` is simply an indentifer, any variable can be used here) to a value representing the total time (in seconds) that the animation runs for. When the timeloop begins, a window is created, which displays the rendered shapes. It checks the time elapsed and if that time doesn’t exceed the time specified by the end variable, it executes the associated statements. It then pauses for `dt` seconds and continues with the time condition check and iteration from the beginning. When the timeloop finishes, it closes the window. The effect of animation is created by the alternating of the drawing operations executed between the timeloop pauses and the shape updates done in the loop body.

The statement that follows can use `dt` and `end` but cannot change `dt` or `end`, or create a variable called `dt` or `end`. If the variable names `dt` and `end` are defined outside the timeloop, they are masked inside the timeloop statement as in any other block.

At the end of each iteration of the time loop, the runtime makes a call to the renderer to redraw all the shapes.

```c
1. timeloop (dt = 0.001 ; end = 10.0)
2. {
3.     planet.x += v * dt;
4.     // double end = 55.5; // this is illegal
5.     end = 55.5; // this is legal, modifiers end but is reset on the next iteration
6.     {
7.         double end; // this is legal, masks the outer end
8.     }
9. }
```

**Frame Loop:**
Frame loop is the second animation specific control structure. The first expression sets the double variable \texttt{fps} (\texttt{fps} is simply an identifier, any variable can be used here) to the number of frames that are rendered per second. The second expression sets the double variable \texttt{frames} (\texttt{frames} is simply an identifier, any variable can be used here) to the total number of frames for the animation. Other than this, timeloop and frameloop are equivalent. In general, a frameloop that runs with \texttt{fps} frames per second for \texttt{frames} number of frames is equivalent to a timeloop that runs for \texttt{fps*frames} seconds and with a delay of $\frac{1}{\texttt{fps}}$ seconds. This holds during smooth animations (the loop finishes well below the delay time). However, during heavy load animation rendering, where a single iteration of the loop may take longer than the delay time, the timeloop prioritizes the time requirements of the animation while the frameloop guarantees the number of frames rendered. The timeloop will attempt to stop iteration when the time exceeds the limit as soon as it yields control from the renderer. This may result in less frames being drawn than the smooth animation case. On the opposite end, the frameloop may pass the \texttt{fps*frames} seconds mark but all the frames will be drawn.

3.7.5 Jump Statements:

Jump statements transfer control unconditionally. Jump statements have the following forms in the language:

\[
\text{jump-statement:} \\
\text{return expression}; \\
\text{return statement:}
\]

A function returns to its caller by the \texttt{return} statement. If an expression is provided, the value of the expression is return to the caller of the function. The expression must match the type returned by the function in which it appears, the only exception being the case where an \texttt{int} is automatically promoted to a \texttt{double} to match the return type. The no expression return statement corresponds to functions that have \texttt{void} return type. Falling off the end of the function is equivalent to a \texttt{return} statement with no expression.

3.8 Variable Declaration

Declarations follow the following format:

\[
\text{declaration:} \\
\text{type-name init-declarator-list ;}
\]

\[
\text{init-declarator-list:} \\
\text{init-declarator} \\
\text{init-declarator-list , init-declarator}
\]

\[
\text{init-declarator:} \\
\text{identifier} \\
\text{identifier = initializer}
\]
Type name corresponds to a non-void type or arrays thereof. The following are examples of valid declarations:

```
int x, y = 3, z = 3;
int[] z = {1, 2, 3}, w = {4, 3, 5};
int[4] b = {1,2,3,4};
int[4] f = {1,}; // partial array initializer
Int[][3] a = { {1,2,3}, {3,4,5}};
```

The declaration can optionally be initialized. The effect of uninitialized variable declarations depends on whether the declarations happens in local or global (outside functions) scope. Local variables are left uninitialized while global variables are zeroed (at the byte level).

For array declaration, the size of the outermost dimension can be omitted but that requires the use of a full array initializer to be provided from which the dimension can be inferred.

Partial array initializers (ending with a comma) can be provided for arrays that have fully defined size. The parts of the array for which the partial initializer doesn’t provide values are zeroed.

If the length of the initializer exceeds the size of the array, the extra elements are cut-off.

### 3.9 Program Structure:

An ART program consists of struct/shape definitions, method definitions, function definitions and global variable declarations.

The entry point for an ART program is the `main()` function and must be defined for all programs. It must have return value `int` which is used to indicate program status to the calling environment. Animation control structures in an ART program can only be used in the main function.

### 3.10 Scoping Rules and Object Lifetimes:

A program will be kept in one source file. All regular functions, member functions, structs and shapes, are immediately accessible anywhere in the source file after being defined. The order of member function definitions does not matter. A function defined after a particular struct can call the member functions of the struct even if the member function definitions appear later in the source.

Unlike declaration of other variables and functions, the order of struct definitions matter. Method definitions can be defined freely once the struct is defined.

Variables declared outside of any function have global scope and are visible at any point in the source after their declaration (definition). This variables persist through the duration of the program.
Member functions (functions that are within the scope of a struct/shape) have access to all the member variables and member function of that struct/shape.

Variables declared within blocks (including function/method bodies) are visible throughout the body of the block. The lifetime of this variables is up to the point they go out of scope.

In particular, the scope of a parameter of a function is throughout the block defining the function and these variables only live during the function call. The exceptions are variables passed through reference which live outside the function. The second form of the for statement defines its own scope where variables declared in the for statement declaration are persistent and visible. Similarly timeloops and frameloops form their own scopes where the corresponding variables $\texttt{dt}$, $\texttt{end}$, $\texttt{fps}$, $\texttt{frames}$ (which can be any variables) are persistent and visible.

Variables names in a nested scope can have the same name as variables/functions names in the outer scope. But the new names hide the old names. Otherwise, names in the nesting scope are accessible from the nested scopes.

### 3.11 Animation Framework

The task of displaying and updating animations (renderings) is distributed among the following components: the shape structures and their draw method which calls the underlying drawing functions, the internal shape list which maintains a list of shapes to be drawn, the timeloop and frameloop constructs which update the animations state maintained in the shapes/programs and the renderer which draws (by calling the respective) draw method of the shapes in the shapelist. The semantics of the timeloop/frameloop is discussed above; this discussion focuses on their interaction with the animation framework.

#### The Draw Method

This is a required method for all shapes which is called when the renderer needs to draw them (provided they are in the shape list). This method has to be active/called by the renderer for any draw operations to take place. The draw method may delegate its drawpoint calls to other functions or methods but the calls to these functions have to originate from a call tree that is rooted on the draw method for the draw operations to have any effect. When a draw method is called by the renderer the draw state of the program is enabled. If the draw state is not enabled, draw operations (like calling draw point) have no effect when called. This also implies calls to draw methods outside of this state have no effect. Eg, what `draw` from main.

#### The List of Shapes

A shape is added to this list by calling the addshape method. Shapes in this list are rendered during the execution of the next timeloop/frameloop. Once a frameloop/timeloop completes execution all shapes in this list are removed.

#### Loops and Renderers

The loops can’t be nested since there is only one render state. This implies a nested timeloop is ignored.
3.12 Grammar

translation-unit:
    external-declaration opt
    translation-unit external-declaration

external-declaration:
    function-definition
    method-definition
    declaration
    struct-or-shape-definition

method-definition:
    method-declarator ( parameter-list opt ) function-block

function-definition:
    function-declarator ( parameter-list opt ) function-block

function-block:
    { declaration-statement-list }

declaration:
    type-name init-declarator-list ;

init-declarator-list:
    init-declarator
    init-declarator-list , init-declarator

init-declarator:
    identifier
    identifier = initializer

struct-or-shape-specifier:
    struct-or-shape identifier

struct-or-shape-definition:
    struct-or-shape identifier { struct-declaration-list }

struct-or-shape:
    struct
    shape

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration

struct-declaration:
    type-name struct-declarator-list ;
struct-declarator-list:
   identifier
   struct-declarator-list , identifier

type-name:
   type-specifier array-declarator_opt

type-specifier: one of
   void char int double vec string struct-or-shape-specifier

array-declarator:
   array-declarator_opt [ constant-expression_opt ]

method-declarator:
   type-name_opt identifier :: identifier

function-declarator:
   type-name identifier

parameter-list:
   parameter-declaration
   parameter-list , parameter-declaration

parameter-declaration:
   type-name identifier
   type-name & identifier

initializer:
   expression
   { initializer-list }
   { initializer-list , }

initializer-list:
   initializer
   initializer-list , initializer

statement:
   expression-statement
   compound-statement
   selection-statement
   iteration-statement
   jump-statement

expression-statement:
   expression_opt ;

compound-statement:
   { declaration-statement-list }
   [ declaration-statement-list ]
<< declaration-statement-list >>

declaration-statement-list:
declaration_{opt}
statement_{opt}
declaration-statement-list declaration
declaration-statement-list statement

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

selection-statement:
if ( expression ) statement
if ( expression ) statement else statement

iteration-statement:
while ( expression ) statement
for ( expression_{opt} ; expression_{opt} ; expression_{opt} ) statement
for ( declaration expression_{opt} ; expression_{opt} ) statement
timeloop ( identifier = expression ; identifier = expression ) statement
frameloop ( identifier = expression ; identifier = expression ) statement

jump-statement:
return expression_{opt} ;

expression:
logical-OR-expression
postfix-expression assignment-operator expression

assignment-operator: one of
= *= /= %= += -=

constant-expression:
logical-OR-expression

logical-OR-expression:
logical-AND-expression
logical-OR-expression || logical-AND-expression

logical-AND-expression:
equality-expression
logical-AND-expression && equality-expression

equality-expression:
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

relational-expression:
  additive-expression
  relational-expression < additive-expression
  relational-expression > additive-expression
  relational-expression <= additive-expression
  relational-expression >= additive-expression

additive-expression:
  multiplicative-expression
  additive-expression + multiplicative-expression
  additive-expression - multiplicative-expression

multiplicative-expression:
  prefix-expression
  multiplicative-expression * unary-expression
  multiplicative-expression / unary-expression
  multiplicative-expression % unary-expression

prefix-expression:
  unary-expression
  ++ unary-expression
  -- unary-expression

unary-expression:
  postfix-expression
  unary-operator unary-expression

unary-operator: one of
  + - !

postfix-expression:
  primary-expression
  postfix-expression [ expression ]
  postfix-expression ( argument-expression-listopt )
  postfix-expression . identifier
  postfix-expression ++
  postfix-expression --

primary-expression:
  identifier
  literal
  vecexpr
  ( expression )
4 Project Plan

4.1 Planning Process

The ART team met up two times a week during the initial planning stages. One of the meetings was usually held on Sunday and the other on Tuesday, with our I.A, Rachel Gordon. At our first meeting, we assigned the four overarching roles: Manager (Natan), Language Guru (Brett), System Architect (Soul) and Tester (Gedion) and defined the main requirements of those roles. While these roles provided a leader for each aspect of the development, it did not mean that only the leader would work on each part. We also set a tentative deadline for each part of the project according to our own time restrictions. Following this, We had multiple brainstorming sessions and explored different types of languages from parallel languages, to 3D modeling to graph oriented languages. At the end, we decided on an animation rendering language and named it ART: Animation Rendering Tool.

4.2 Specification Process

After deciding on the abstract idea of our language, we set out to identify its details. We layed out the following main parts of the language to guide us in the development process:

- overarching description of the language we plan to implement
- explanation of the types of programs to be written with the language
- the different parts of our language and their purpose
- syntax inspirations
- an estimate of what the simplest and most complex code in our language could look like
- a sample source code showing what we envision our language to look like

Finally, we compiled a Language Reference Manual for our language.
4.3 Development Process

Using our Language Reference Manual, we worked on each individual part of the compiler structure starting with the lexer. After the lexer, we wrote the parser followed by the code generator. The code generator took care of some semantic checks, but for the sake of consistent and readable code, we transferred all semantic checking to a new semantic checker.

Throughout the whole development process, we maintained a GitHub branch flow for code collaboration and utilized a group messageboard for efficient communication.

4.4 Testing Process

Tests were compiled after each stage of the development was completed and after each feature of the language was implemented. For example, tests were written to see that the lexer and parser were working separately. Similarly, individual tests were written to make sure every aspect of each feature such as primitive types, arrays, structs and shapes were working as expected. Although we attempted to make test errors as detailed as possible, it was still challenging to identify the exact reason for failures with complex codes. Hence, we decided to write simple tests that focus on individual features and specific functionalities of those features. Since some features are dependent on others, we planned to test the most independent features first and build up to more dependent functionalities.

4.5 Team Responsibilities

Although roles were assigned at the beginning of the semester, members of the team worked on many different parts of the project. The following table shows a rough distribution of the main roles each team member had.

<table>
<thead>
<tr>
<th>Team Member</th>
<th>Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brett Jervey</td>
<td>Semantics, Testing, Code Generation</td>
</tr>
<tr>
<td>Gedion Metaferia</td>
<td>Lexer, Parser, Code Generation, Semantics</td>
</tr>
<tr>
<td>Natan Kibret</td>
<td>Testing, Documentation, Report</td>
</tr>
<tr>
<td>Soul Joshi</td>
<td>Testing, Semantics, Code Generation, Report</td>
</tr>
</tbody>
</table>

4.6 Project Timeline

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 28</td>
<td>Project Proposal completed</td>
</tr>
<tr>
<td>October 8</td>
<td>First commit/development architecture setup</td>
</tr>
<tr>
<td>October 25</td>
<td>Language Reference Manual completed</td>
</tr>
<tr>
<td>October 30</td>
<td>Lexer and parser completed</td>
</tr>
<tr>
<td>November 13</td>
<td>Code generation partially completed</td>
</tr>
<tr>
<td>November 15</td>
<td>Hello world code compiles</td>
</tr>
<tr>
<td>December 17</td>
<td>Semantic checking complete</td>
</tr>
<tr>
<td>December 18</td>
<td>Code generation fully completed</td>
</tr>
<tr>
<td>December 19</td>
<td>Test suite (comprehensive) complete</td>
</tr>
<tr>
<td>December 19</td>
<td>Final report complete</td>
</tr>
</tbody>
</table>
4.7 Project Log

<table>
<thead>
<tr>
<th>Date</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 19</td>
<td>Language identified</td>
</tr>
<tr>
<td>October 8</td>
<td>Language proposal completed</td>
</tr>
<tr>
<td>October 15</td>
<td>Language features adjusted according to feedback</td>
</tr>
<tr>
<td>October 25</td>
<td>Language reference manual completed</td>
</tr>
<tr>
<td>October 30</td>
<td>Lexer and parser completed</td>
</tr>
<tr>
<td>October 30</td>
<td>Tests for lexer and parser passed</td>
</tr>
<tr>
<td>November 13</td>
<td>Code generation partially completed</td>
</tr>
<tr>
<td>November 14</td>
<td>Test Suite Framework completed</td>
</tr>
<tr>
<td>November 22</td>
<td>Hello World Demo Code Done</td>
</tr>
<tr>
<td>December 5</td>
<td>Basic semantic checking started</td>
</tr>
<tr>
<td>December 17</td>
<td>Semantic checking complete</td>
</tr>
<tr>
<td>December 18</td>
<td>Merged all GitHub branches to a final branch</td>
</tr>
<tr>
<td>December 19</td>
<td>All test suits completed</td>
</tr>
<tr>
<td>December 19</td>
<td>Finan report completed</td>
</tr>
<tr>
<td>December 20</td>
<td>Official demo of project.</td>
</tr>
</tbody>
</table>

4.8 Software Development Environment

The following software development environments/tools and languages were used:

- Development tools: Sublime text, vim.
- Programming languages: Ocaml 4.02.3 for creation of the lexer, parser and semantics. Used LLVM ocaml bindings to create code generator and ocaml build tools (automated via a Makefile) to build the compiler.
- Libraries: GLUT from OpenGL for animation rendering and C standard library for the various print functions and math functions.
- Collaborative platform: Git repository on GitHub.

4.9 Programming Style Guide

The following programming styles were followed as much as possible.

Formatting, Indentation and Spaces

Every member of the team used Sublime Text to edit code; therefore, indentation remained consistent. We did not specify an indentation method between tabs and spaces for the sake of freedom of choice.

For variable naming, underscore notation was used more dominantly, but it was not a requirement.

There were no line length restrictions either.

Comments and documentation:

Extensive comments were used when possible. The amount of comments across all code is not consistent, but a minimum needed to clarify the code was required.
5 Architectural Design

5.1 Compiler

ART utilizes a Lexer, Parser, Semantic checker, and Code Generator. Figure 3 shows the ART compiler architecture:

![Figure 3: ART Compiler Architecture](image)

5.2 Lexer

File: scanner.mll

Scanner.mll takes the ART source codes and converts the sequence of characters into tokens identified by our language. These include keywords, identifiers and literals. Additionally, it identifies and removes unnecessary whitespace and all comments.

5.3 Parser and AST

Files: parser.mly, ast.ml
Parser makes sure that the code is accurate in terms of syntax. It receives tokens from the lexer and makes an Abstract Syntax Tree (AST) using ART’s grammar specifications. Tokens usually become the nodes of the AST. Different parts of the code, such as conditionals, loops, operations and more are assembled into their syntactic architecture.

5.4 Semantic Checker

File: semant.ml

Semantic checker ensures that the source code is semantically accurate. While parser checks for syntactic validity, semantic checker makes sure it follows the conventions of our language as expected by our code generator. Semant.ml goes through the AST and applies semantic analysis in order to produce the SAST, which is an abstract syntax tree which has been syntactically checked.

5.5 Code Generator

File: codegen.ml

Code Generator traverses through the SAST and generates the corresponding LLVM byte code (*.bc). This is also where we integrate the GLUT functions into the code.

6 Test Plan

Tests were conducted throughout the development process. Each part of the architecture was tested as it was being developed. The lexer and parser were tested by writing sample ART source codes and examining the output files generated by a pretty-printer to make sure that the AST was built properly. To test the semantic checker, we wrote test cases that should fail according to the conventions of the language to make sure that the errors were being caught. For the code generator, we compiled the program and checked that the expected output was being produced. We also checked the generated LLVM IR code for any errors.

6.1 Sample Test Source Codes

Let us look at some sample test source codes that are part of the automated test suite:

Source code 1: drawing a red circle in motion and a blue disc in motion (with a curved line trailing the path it passes through.

Fail Test Example: Test that Draw Method should not take any arguments

```
1. // draw must have return type void and no parameters
2.
3. shape circle
4. {
5.   double r;
6. }
7.
8. circle::circle(double x)
9. {
10.   r = x;
11.   printf(r);
12. }
```
void circle::draw(int x) {
    //
    shape circle new = circle(10.0);
    printc('n');
    prints("Hello");
    printc('n');
}

Output:
Fatal error: exception Failure("draw method must have return type void, and no parameters")

Pass Test Example: Test that Integer can be assigned to a Doubles

// Test integer promotion in assinging int to double
int main()
{
    double a = 5;
    double b = 10.5;
    printf(a);
    printf(\n);
    printf(b);
    printf(\n);
    a = 9.5;
    b = 4;
    printf(a);
    printf(\n);
    printf(b);
    printf(\n);
}

Output:
5.000000
10.500000
9.500000
4.000000

6.2 Test Suites

A new test case was developed for every new feature that was implemented in our language. Each test case had one sourcefile that was named “test-name’of test.art” that will successfully compile and one source code that was named “fail-name’of test.art” that would fail. Each success case has a corresponding “test-name’of test.out” which includes the expected output of the program and each fail case has a corresponding “fail-name’of test.err” which includes the expected error message of the program. There are some test cases that were created without an expected output file since they included animation rendering. These test cases do not begin with “test” or “fail.”
6.3 Test Automation

In order to efficiently test our code everytime a feature was added, we implemented a test automation structure that runs all the test cases at once. After compiling the code, we run `bash testall.sh` to execute the suite.

If the success cases do not match with their corresponding `.out` files or if the fail cases do not match with their corresponding `.err` files, then the error `FAILED` is previewed next to the test case and the difference between the expected output and the actual output is saved in a `results` folder as `test-file-name.diff` or `fail-file-name.diff` files. If the test case has the expected outcome, the message `OK` is previewed next to the test case.

Below is a sample run of testall.sh. It is not the final version of the project but a good representation of the error messages.
Terminal Example: Output of bash testall.sh

Notice that there is a path to the results directory where you can find a .diff file for the failed expectation.
6.4 Roles of Members

Soul and Gedion setup the test suite structure and the automation script testall.sh. Brett, Gedion, Natan, and Soul wrote tests for different parts of the code as well as for each feature implementation throughout the development process. Additionally, the team reconvened before submission date and wrote more test cases.

7 Lessons Learned

7.1 Bretton Jervey

Having a project made up so many separate but interconnected parts, I learned the importance of version control and the need to communicate when major changes were made. Even one change can radically effect other people's work so having the ability to revert or have changes made in a sandbox is critical. Github proved to be an excellent but initially daunting version system and learning to use it was a bit of a problem. My suggestion would be become familiar with Github way before PLT since its applications beyond this class are immeasurable.

7.2 Gedion Metaferia

As part of the ART team, I learned the value of test oriented programming and the significance of various coding conventions and their benefits. Although I enjoyed working with the team to produce a great project, I did realize that from time to time, good communication skills come in handy to make sure that effort is not duplicated and everyone is clear on their roles. I loved the experience I gained from working with Ocaml. For future classes, I suggest good communication, an early start and a positive attitude when meeting with their teammates.

7.3 Natan Kibret

As part of the ART team, I learned that it is important to communicate specific details of our code and thought process through comments as well as meeting times. It was very helpful that we started the project very early, but I've learned that it is hard to keep the momentum going as a group for the whole semester, so it is essential to make steady progress week after week. Formatting and descriptive content takes a lot longer than anyone expects, so allocate more than estimated time for final report.

7.4 Soul Joshi

Working on a semester-long project like ART, I realized the importance of proper planning, team communication and work distribution. I also learnt that, even though the work may be distributed among team members, it is important to go through each other's work to stay updated with what everyone is doing and not lose track of the whole picture by restricting yourself to just your part of the work. Code walkthrough with all members present were very helpful for this. I also realized the important of regression testing throughout the development process. The automated test suite was very useful to ensure that new changes did not break old working features. Starting early was helpful and I learnt that it is sometimes difficult to maintain steady progress throughout a long project like ART.
8 Appendix

8.1 Built-in functions/definitions

\[
\begin{align*}
\text{sin} & \quad \text{cos} & \quad \text{tan} & \quad \text{log} & \quad \text{log2} & \quad \text{log10} & \quad \text{abs} \\
\text{exp} & \quad \text{sqrt} & \quad \text{asin} & \quad \text{acos} & \quad \text{atan} & \quad \text{sinh} & \quad \text{cosh} \\
\text{tanh} & \quad \text{asinh} & \quad \text{acosh} & \quad \text{atanh} & \quad \text{PI} & \quad \text{E} & \quad \text{pow}
\end{align*}
\]

All the math functions take the one `double` argument except `pow`, which takes two `double` arguments.

`PI` and `E` are constants.

The following functions are used for printing:

\[
\begin{align*}
\text{printi} & \quad \text{printf} & \quad \text{printc} & \quad \text{prints}
\end{align*}
\]

8.2 Lexer or Scanner

File Name: `scanner.mll`

1. (* Ocamlex scanner for ART *)
2. { open Parser
3. 5. (* Converts escape sequences into characters *)
4. 7. let char_esc = function
8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19. | "\n" -> Char.chr(0XA) | "\t" -> Char.chr(0X9) | "\v" -> Char.chr(0X8) | "\b" -> Char.chr(0X8) | "\f" -> Char.chr(0XC) | "\a" -> Char.chr(0X7) | "\\\\" -> '\\\'
23. 24. 25. 26. 27. 28. 29. 30. | "\\?" -> '?'
27. 28. 29. 30. 31. 32. 33. 34. | "\\" -> '\'
28. 29. 30. 31. 32. 33. 34. 35. | "\\\\\\" -> ' ' 
31. 32. 33. 34. 35. 36. 37. | e -> raise (Failure("illegal escape " ^ e))
35. \}
36. 22. 23. 24. 25. let oct = ['0' - '7'] (* Octal digits *)
37. let dec = ['0' - '9'] (* Decimal digits *)
38. 27. let hex = dec | ['A' - 'F' 'a' - 'f'] (* Hex digits *)
39. 28. 29. 30. 31. let printable = [' ' - '~'] (* All printable *)
32. 33. 34. 35. 36. 37. let escapes = "\n" | "\t" | "\v" (* Escaped chars *)
36. 47
```
48
[301x49]48
[88x762]31.  "\\b"  "\\n"  "\\f"
32.  "\\a"  "\\\"  "\\?"
33.  "\\"  "\\"  "\\"
34.  let octescp = (oct | oct oct | oct oct oct) (* Octal escapes *)
35.  let hexescp = hex+ (* Hex escapes *)
36.
37.  let exp = 'e' ('+' | '-')? dec+ (* Floating point exponent *)
38.  let double = '.:' dec* exp?
39.  (* Floating point Literal *)
40.
41.  rule token = parse
42.  '
'  {Lexing.new_line lexbuf; (* Keep track of new lines for error printing *)
43.  }  token lexbuf
44.  '/'
45.  '{'  {LPAREN }
46.  '}'  {RPAREN }
47.  ':'  {LBRACE }
48.  ']'  {LRBRAKE }
49.  '.'  {DOT }
50.  '?'  {QMARK }
51.  '&.'  {AMPS }
52.  '::'  {DCOLON }
53.  '...'  {SEMI }
54.  '..'  {COMMA }
55.  '+'  {PLUS }
56.  '-'  {MINUS }
57.  '+'  {TIMES }
58.  '/'  {DIVIDE }
59.  '%'  {MOD }
60.  '='  {ASSIGN }
61.  '+='  {PLUSASSIGN }
62.  '-='  {MINUSASSIGN }
63.  '*='  {TIMESASSIGN }
64.  '/='  {DIVAS }
65.  '%='  {MODASSIGN }
66.  '++'  {PLUSPLUS }
67.  '--'  {MINUSMINUS }
68.  '=='  {EQ }
69.  '!='  {NEQ }
70.  '<'  {LT }
71.  '<='  {LEQ }
72.  '>'  {GT }
73.  '>='  {GEQ }
74.  '&&'  {AND }
75.  '||'  {OR }
76.  '!'  {NOT }
77.  '::'  {DCOLON }
78.  '<<'  {LTLT }
79.  '>>'  {GTGT }
80.  'if'  {IF }
81.  'else'  {ELSE }
82.  'for'  {FOR }
83.  'while'  {WHILE }
84.  'timeloop'  {TLOOP }
85.  'frameloop'  {FLOOP }
86.  ('*'| "break"  { BREAK }
87.  ("continue"  { CONTINUE })
88.  ("return"  { RETURN })
89.  (* Built-in Types *)
```
96. | "void" | { VOID }
97. | "int" | { INT }
98. | "char" | { CHAR }
99. | "double" | { DOUBLE }
100. | "vec" | { VEC }
101. | "string" | {STRING}
102. (* Type keywords *)
103. | "struct" | { STRUCT }
104. | "shape" | { SHAPE }
105. (* Integer Literals *)
106. | '0' oct* as lxm | { INTLIT( int_of_string("0o"^lxm))] (* reads octal and converts to int *)
107. | '0' ('x' | 'X') hex* as lxm | { INTLIT( int_of_string lxm)} (*reads hex and converts to int *)
108. | ['1'-'9'] dec* as lxm | { INTLIT(int_of_string lxm) } (* reads int *)
109. (* Identifier *)
110. | ['a'-'z' 'A'-'Z' '_'][-'a'-'z' 'A'-'Z' '0'-'9' '_'] * as lxm | { ID(lxm) }
111. (* Character Literals *)
112. | \\'' (printable as lex) "\\" | { CHARLIT (lex) } (* printable within single quotes *)
113. | \\'' (escapes as lex) \\'' | { CHARLIT (char_esc lex) } (* escapes single quotes *)
114. | \\'''(octescp as lex)'''' | { CHARLIT (Char.chr(int_of_string("0o"^lex)))}
115. (* oct escapes *)
116. | \\'''(hexescp as lex)'''' | { CHARLIT (Char.chr(int_of_string("0x"^lex)))}
117. (* hex escapes *)
118. | \\'''(" printable as lex)'''' | { CHARLIT (char_esc lex) } (* Catch invalid escapes *)
119. (* Double Literal *)
120. | double as lex | { FLOATLIT (float_of_string lex)}
121. (* Handle strings as a sequence of character tokens *)
122. | \\'' | { read_string (Buffer.create 2048) lexbuf} 
123. (* Vector Literal *)
124. | \\'<' spc* (double as lex1) spc* \\'>' | { VECTORLIT(float_of_string lex1, float_of_string lex2)}
125. (* end of line rule *)
126. | _ as char | { raise (Failure("illegal character " ^ Char.escaped char)) }
127. and block_comment = parse
128. | "*/" | { token lexbuf }
129. | \\n | { lexing.new_line lexbuf; block_comment lexbuf}
130. | _ | { block_comment lexbuf }
131. and line_comment = parse
132. | \\n | { Lexing.new_line lexbuf; token lexbuf }
133. | _ | { line_comment lexbuf }
134. and read_string buf = parse
135. | "\\" (printable as lex) | { STRINGLIT (Buffer.contents buf) }
136. | (escapes as lex) | { Buffer.add_char buf (lex); read_string buf lexbuf} (* printable *)
137. | (octescp as lex) | { Buffer.add_char buf (char_esc lex); read_string buf lexbuf} (* oct escapes *)
138. | (hexescp as lex) |
139. | (double as lex1) spc* (double as lex2) spc* \\'>' |

152. | "\\x" (hexescp as lex)  { Buffer.add_char buf (Char.chr(int_of_string ("0x" ^ lex))); read_string buf lexbuf} (* hex escapes *)
153. | ("\\" printable as lex)  { Buffer.add_char buf (char_esc lex); read_string buf lexbuf} (* Catch invalid escapes *)
154. | \n { raise (Failure ("Multiline string not supported")) }
155.
156. | _ { raise (Failure ("Illegal string character: " ^ Lexing.lexeme lexbuf)) }
157. | eof { raise (Failure ("String is not terminated")) }
8.3 Parser

File Name: parser.mly

```ocaml
/* Ocamlyacc parser for ART */

open Ast

let make_dec_list s l =
  List.map (fun (a,b) ->
    (s, a, b)) l ;; (* if there is a list of a,b, then s is appended infront *)

let make_struct_dec_list s l =
  List.map (fun a -> (s, a)) l ;;

(* Build array of type t from the list of dimensions *)

let build_array t =
  function [] -> t
  | [i] -> Array(t, i)
  | i::l -> List.fold_left (fun at e -> Array(at,e)) (Array(t, i)) l

(*let handle_mixed_scoping*)

type mixed_decl = {l: vdecl list; t: vdecl list * stmt list *}

let handle_dup ml (dl,sl) =

let rec helper = function
  (_,n1,i) :: (_,n2,_) :: _ when n1=n2 -> [(t,n1,i)]
  _ :: tl -> helper tl
  [] ->[] in

let ml = (List.sort (fun (_,n1,_) (_,n2,_) -> compare n1 n2) ml) in

match helper ml with
  [s] -> (s::dl,sl)
  _ -> (dl,sl)

%

%token VOID INT CHAR DOUBLE VEC STRING
%token LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE
%token DOT QMARK COLON COMMA SEMI AMPS DCOLON
%token PLUS MINUS TIMES DIVIDE MOD
%token ASSIGN PLUSASSIGN MINUSASSIGN TIMESASSIGN
%token DIVASSIGN MODASSIGN PLUSPLUS MINUSMINUS
%token RETURN IF ELSE FOR WHILE
%token STRUCT SHAPE
%token TLOOP FLOOP
%token <int> INTLIT
%token <char> CHARLIT
%token <float> FLOATLIT
%token <float * float> VECTORLIT
%token <string> STRINGLIT
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN PLUSASSIGN MINUSASSIGN TIMESASSIGN DIVASSIGN MODASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right PRE /* prefix increment/decrement */
%right NOT NEG POS
%left INDEX CALL MEMB /* member */ POST /* postfix decrement/increment */
%nonassoc VECEXPR
```
%start program
%type <Ast.prog> program
%
program:
  decls EOF {{s = List.rev ($1.s); f = List.rev ($1.f); v = $1.v }}

decls:
  /* empty */ { [s = []; f = []; v = []] } /* { structs, funcs, vars } */
  | decls struct_or_shape_definition {{ s = $2::$1.s; f = $1.f; v = $1.v }}
  | decls fdecl {{ s = $1.s; f = $2::$1.f; v = $1.v }}
  | decls mdecl {{ s = $1.s; f = $2::$1.f; v = $1.v }}
  /*| decls mdecl {List.find}*/

fdecl:
  function_declarator LPAREN parameter_list RPAREN func_block
    { { rettyp = fst $1;
      fname = snd $1;
      params = List.rev $3;
      locals = fst $5;
      body = snd $5;
      typ = Func;
      owner= "" } }

function_declarator:
  vdecl_typ ID {($1, $2)}
  | VOID ID {Void, $2} 

mdecl:
  method_declarator LPAREN parameter_list RPAREN func_block
    { { rettyp = fst (fst $1);
      fname = snd (fst $1);
      params = List.rev $3;
      locals = fst $5;
      body = snd $5;
      typ = snd (snd $1);
      owner= fst (snd $1) } }

method_declarator:
  vdecl_typ ID DCOLON ID {($1, $4), ($2, Method)} /* (rettyp, fname) (owner, typ)*/
  | VOID ID DCOLON ID {Void, $4), ($2, Method) /* (rettyp, fname) (owner, typ)*/ 
  | ID DCOLON ID {Void, $3), ($1, Constructor}) /* Constructor */

func_block:
  /* Function Block */
  LBRACE decl_list_stmt_list RBRACE {$2}

parameter_list:
  /* No parameter case */ {[]}
  | parameter_declaration {[$1]}

parameter_declaration:
  vdecl_typ ID {{($1, $2, Value)}}
  | vdecl_typ AMPS ID {($1, $3, Ref)}

struct_or_shape:
  STRUCT { StructType }
  | SHAPE { ShapeType }

struct_or_shape_specifier:
  struct_or_shape ID { UserType($2, $1)}
struct_or_shape_definition:
  struct_or_shape ID LBRACE struct_declaration_list RBRACE
  { { ss = $1; sname = $2; decls = $4; ctor = default_ctr ""; methods = []} }

struct_declaration_list:
  struct_declaration
  { $1}
  | struct_declaration_list struct_declaration
  { $1 @ $2 }

struct_declaration:
  ID
  {[$1]}
  | struct_declaration_list COMMA ID
  { $3 :: $1}

declarator_list:
  struct_or_shape_specifier { $1}

declarator:
  ID
  { ($1, Noinit) }
  | ID ASSIGN init
  { ($1, $3) }
```plaintext
189. init:
190.  expr  {Exprinit($1)}
191.  | LBRACE init_list RBRACE {Listinit( List.rev $2)}
192.  | LBRACE init_list COMMA RBRACE {Listinit( List.rev $2 )}
193.  
194. init_list:
195.  init  {[ $1 ]} /* usefull for 2d arrays */
196.  | init_list COMMA init  {$3 :: $1}
197.  
198. stmt_list: /* inverted list of the statements */
199.  stmt  {[ $1 ]}
200.  | stmt_list stmt  {$2 :: $1}
201.  
202. stmt:
203.  expr_opt SEMI  { Expr $1 }
204.  | RETURN SEMI  { Return (Noexpr,Void) }
205.  | RETURN expr SEMI  { Return $2 }
206.  
207.  /* Block */
208.  | stmt_block  {$1 } /* defined in stmt_block: */
209.  
210.  | IF LPAREN expr RPAREN %prec NOELSE  { If($3, $5, Block([],[],PointContext)) }
211.  | IF LPAREN expr RPAREN ELSE stmt  { If($3, $5, $7) }
212.  
213.  | FOR LPAREN expr_opt SEMI for_cond SEMI expr_opt RPAREN stmt
214.  { For($3, $5, $7, $9) }
215.  /* Deal with for with declaration */
216.  | FOR LPAREN declaration for_cond SEMI expr_opt RPAREN stmt
217.  { ForDec($3, $4, $6, $8) }
218.  | WHILE LPAREN expr RPAREN stmt  { While($3, $5) }
219.  
220.  | TLOOP LPAREN ID ASSIGN expr SEMI ID ASSIGN expr RPAREN stmt
221.  { Timeloop($3, $5, $7, $9, $11) }
222.  | FLOOP LPAREN ID ASSIGN expr SEMI ID ASSIGN expr RPAREN stmt
223.  { Frameloop($3, $5, $7, $9, $11) }
224.  
225.  stmt_block:
226.  /* Block */
227.  | LBRACE decl_list_stmt_list RBRACE  { Block(fst $2, snd $2, PointContext) }
228.  | LBRACK decl_list_stmt_list RBRACK  { Block(fst $2, snd $2, LineContext) }
229.  | LT LT decl_list_stmt_list GTGT  { Block(fst $2, snd $2, TriangleContext) }
230.  
231.  decl_list_stmt_list:
232.  /* Empty Block */  {([],[])}
233.  | stmt_list  {([], List.rev $1)} /* stmt_list needs to be reversed */
234.  
235.  cdlist_slist_pair_list  {$1}
236.  | stmt_list cdlist_slist_pair_list  {([],List.rev( (Block (fst $2, snd $2, PointContext))::($1) )))}
237.  
238. cdlist_slist_pair_list:
239.  dlist_slist_pair_list  {handle_dup $1.1 $1.t}
240.  
241. dlist_slist_pair_list:
242.  dlist_slist_pair  {$1}
243.  | declaration_list  {l=$1; t=($1,[])}
244.  | dlist_slist_pair dlist_slist_pair_list  {l=($1.1)@($2.1); t=(fst $1.t, List.rev( Block(fst $2.t, snd $2.t, PointContext)::(List.rev(snd $1.t)) ))}
245.  
246.  /* A pair of declaration List , statement List */
247.  dlist_slist_pair:
248.  ```
declaration_list stmt_list {{l=$1; t=(t($1, List.rev $2))} /* declaration doesn't need reversing */
expr_opt:
  /* nothing */ { (Noexpr,Void) }
  | expr { $1 }
for_cond:
  /* nothing */ { (IntLit 1,Int) }
  | expr { $1 }
expr:
  bexpr { $1 }
  /* postfix expressions */
  posexpr {$1}
  /* unary */
  | PLUS addexpr %prec POS { (Unop(Add, $2), Void) }
  | MINUS addexpr %prec NEG { (Unop(Sub, $2), Void) }
  | NOT addexpr { (Unop(Not, $2), Void) }
  | PLUSPLUS addexpr %prec PRE { (Unop(Preinc, $2), Void) }
  | MINUSMINUS addexpr %prec PRE { (Unop(Predec, $2), Void) }
  /* multiplicative */
  | addexpr TIMES addexpr { (Binop(Mult, $3), Void) }
  | addexpr DIVIDE addexpr { (Binop(Div, $3), Void) }
  | addexpr MOD addexpr { (Binop(Mod, $3), Void) }
  /* additive */
  | addexpr PLUS addexpr { (Binop(Add, $3), Void) }
  | addexpr MINUS addexpr { (Binop(Sub, $3), Void) }
  /* Literals */
}
INTLIT { (IntLit($1), Int) }
CHARLIT { (CharLit($1), Char) }
FLOATLIT { (FloatLit($1), Float) }
VECTORLIT { (VecLit($1), Vec) }
stringlit_list { (StringLit($1), String) }
/* Vector expression */
LT addexpr COMMA addexpr GT %prec VECEXPR { (Vecexpr($2, $4), Vec) }
/* primary expression */
ID { (Id($1),Void) }
LPAREN expr RPAREN { $2 }
/* postfix expression */
posexpr LBRACK expr RBRACK %prec INDEX { (Index($1, $3), Void) }
posexpr LPAREN arg_list RPAREN %prec CALL { (Call($1, List.rev $3), Void) }
stringlit_list:
STRINGLIT {$1}
arg_list:
/* nothing */ {[]}
expr_list {[$1]}
/* Inverted List */
expr_list:
expr {[$1]}
expre_list COMMA expr {[$3::$1]}
/* List.rev is used because in arg_list, expr_list is build from the back cause it is more efficient*/
posexpr DOT ID %prec MEMB { (Member($1, $3), Void) }
posexpr PLUSPLUS %prec POST { (Posop(Postinc, $1), Void) }
posexpr MINUSMINUS %prec POST { (Posop(Postdec, $1), Void) }
arg_list:
EXPR_LIST {[$1]}
/* Inverted List */
expr_list:
expr {[$1]}
expre_list COMMA expr {[$3::$1]}
/* Vector expression */
LT addexpr COMMA addexpr GT %prec VECEXPR { (Vecexpr($2, $4), Vec) }
/* primary expression */
ID { (Id($1),Void) }
LPAREN expr RPAREN { $2 }
/* postfix expression */
posexpr LBRACK expr RBRACK %prec INDEX { (Index($1, $3), Void) }
posexpr LPAREN arg_list RPAREN %prec CALL { (Call($1, List.rev $3), Void) }
stringlit_list:
STRINGLIT {$1}
arg_list:
/* nothing */ {[]}
expr_list {[$1]}
/* Inverted List */
expr_list:
expr {[$1]}
expre_list COMMA expr {[$3::$1]}
/* List.rev is used because in arg_list, expr_list is build from the back cause it is more efficient*/
posexpr DOT ID %prec MEMB { (Member($1, $3), Void) }
posexpr PLUSPLUS %prec POST { (Posop(Postinc, $1), Void) }
posexpr MINUSMINUS %prec POST { (Posop(Postdec, $1), Void) }
arg_list:
EXPR_LIST {[$1]}
/* Inverted List */
expr_list:
expr {[$1]}
expre_list COMMA expr {[$3::$1]}
8.4 Abstract Syntax Tree

File Name: ast.ml

1. (* Abstract Syntax Tree and functions for printing it *)
2. (* binary operations *)
3. type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater | Geq |
4. And | Or |
5. (* Assignment operations *)
6. type asnop = Asn | CmpAsn of op (* Compound Assignment operator op= *)
7. (* unary operators *)
8. type uop = Neg | Not | Pos | Preinc | Predec
9. (* postfix operation*)
10. type pop = Postinc | Postdec
11. (*type typ = Int | Char | Float | Void | Array of typ * expr | UserType of string*)
12. type stosh = StructType | ShapeType
13. (* these are the types you can use to declare an object *)
14. type typ = Int | Char | Float | Vec | Void | Array of typ * expr | UserType of string
15. g*stosh | String
16. and baseexpr =
17. | IntLit of int
18. | CharLit of char
19. | StringLit of string
20. | FloatLit of float
21. | VecLit of (float * float)
22. | Id of string
23. | Vecexpr of expr * expr
24. | Binop of expr * op * expr
25. | Asnop of expr * asnop * expr (* Assignment operation *)
26. | Unop of uop * expr
27. | Posop of pop * expr
28. | Call of expr * expr list (* expr list = list of arguments *)
29. | Index of expr * expr (* more general than it needs to be, needs to be checked for symantec *)
30. | Member of expr * string
31. | Promote of expr
32. | Noexpr
33. and expr = baseexpr * typ
34. type initer = Exprinit of expr | Listinit of initer list | Noinit
35. (* bind is for variable declaration *)
36. type bind = typ * string
37. (*you have to specify if it is passed by reference or by value*)
38. type pass = Value | Ref
39. (* methods stored as functions *)
40. type fbind = typ * string * pass
41. (* variable declaration *)
42. type vdecl = typ * string * initer
43. (* Context types *)
57. type context = PointContext | LineContext | TriangleContext
58.
59. type stmt =
60.   Block of vdecl list * stmt list * context
61.   | Expr of expr
62.   | Return of expr
63.   | If of expr * stmt * stmt
64.   | For of expr * expr * expr * stmt
65.   | ForDec of vdecl list * expr * expr * stmt
66.   | While of expr * stmt
67.   | Timeloop of string * expr * string * expr * stmt
68.   | Frameloop of string * expr * string * expr * stmt
69.
70. (* types of functions *)
71. type ftyp = Func | Method | Constructor
72.
73. (* allows us to store all function data in one struct *)
74. type fdecl = {
75.   rettyp : typ;
76.   fname : string;
77.   params : fbind list;
78.   locals : vdecl list;
79.   body : stmt list;
80.   typ : ftyp;
81.   owner: string ; (* Refers to owning struct/shape *)
82. }
83.
84. (* Actually filled in the semantic step *)
85. type usrtype = {
86.   ss : stosh; (* struct/shape? *)
87.   sname : string;
88.   decls : bind list; (* member var declarations *)
89.   ctor : fdecl; (* constructor *)
90.   methods : fdecl list;
91.   }
92.
93. (* Type of program: user type, function declaration and variable declaration *)
94. type prog = {
95.   s : usrtype list;
96.   f : fdecl list;
97.   v : vdecl list;
98.   }
99.
100. (* Pretty-printing functions *)
101.
102. let string_of_op = function
103.  Add -> "+
104.  | Sub -> "-" 
105.  | Mult -> "*" 
106.  | Div -> "/" 
107.  | Mod -> "%" 
108.  | Equal -> "==" 
109.  | Neq -> "!=" 
110.  | Less -> "<" 
111.  | Leq -> "<=" 
112.  | Greater -> ">" 
113.  | Geq -> ">=" 
114.  | And -> "&&" 
115.  | Or -> "||" 
116.
117. let string_of_asnop = function
118.  Asn -> "=
119.  | CmpAsn o -> string_of_op o ^ "\"=
120.  |
let string_of_uop = function
| Neg -> "-"
| Pos -> "+
| Not -> "!"
| Preinc -> "++"
| Predec -> "--"

let string_of_pop = function
| Postinc -> "++"
| Postdec -> "--"

let string_of_chr = function
| 'b' | 't' | 'n' | 'r' as c -> Char.escaped c
| c when Char.code(c) > 31 && Char.code(c) < 127 -> Char.escaped c
| c -> "\" ^ Printf.sprintf "%a" (Char.code c)

let rec list_of_arr = function
| Array(Array(_,_) as a, i) -> let (t,l) = list_of_arr a in (t, i::l)
| Array(t, i) -> (t, [i])
| t -> (t, []) (* Syntactically Required but not used *)

let string_of_stosh = function
| rec STRUCTType -> "struct"
| ShapeType -> "shape"

let rec string_of_typ = function
| Int -> "int"
| Char -> "char"
| Void -> "void"
| Float -> "double"
| Vec -> "vec"
| String -> "string"
| UserType(n,ss) -> string_of_stosh ss ^ " " ^ n
| Array(_,_) as a -> let (t,l) = list_of_arr a in string_of_typ t ^ String.concat "" (List.map (fun e - > "[" ^ string_of_expr e ^ "]") l)

(* Uncomment the next comment for full parenthesized *)

and string_of_baseexpr (*e = "(" ^ paren_of_expr e ^ ")")

and paren_of_expr *) = function
| IntLit(l) -> string_of_int l
| CharLit(l) -> "" ^ (string_of_chr l) ^ ""
| FloatLit(l) -> string_of_float l
| StringLit(s) -> "" ^ s^""""^"
| VecLit(a,b) - > "< " ^ (string_of_float a) ^ " , " ^ (string_of_float b) ^ " >"
| Id(s) -> s
| Vecexpr(e1,e2) -> " < " ^ string_of_expr e1 ^ " , " ^ string_of_expr e2 ^ " >"
| Binop(e1, o, e2) - > string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
| Asnop(e1, o, e2) - > string_of_expr e1 ^ " " ^ string_of_asnop o ^ " " ^ string_of_expr e2
| Unop(o, e) -> string_of_uop o ^ string_of_expr e
| Posop(e, o) -> string_of_expr e ^ string_of_pop o
| Call(f, e1) ->
| string_of_expr f ^ "(" ^ String.concat ", " ^ (List.map string_of_expr e1) ^ "")"
| Index(e1, e2) -> string_of_expr e1 ^ "][" ^ string_of_expr e2 ^ "]"
| Member(e1, s) -> string_of_expr e1 ^ "]" ^ s
| Promote e -> string_of_expr e
| Noexpr -> ""

and string_of_expr = function
| (Promote (e,t1), t) when t1<>t - > (string_of_typ t)^"\"_of("\"(string_of_baseexpr e)\")"

59
let string_of_bind (t,s) =
    string_of_typ t "^" "^ s
let string_of_fbind (t,s, v) =
    match v with
    | Value -> string_of_typ t "^" "^ s
    | Ref -> string_of_typ t "^& "^ s

let rec string_of_initer = function
    | Exprinit(e) -> string_of_expr e
    | (* Already Reversed *)
    | Listinit(el) -> "{" ^ String.concat ", " (List.map string_of_initer (el)) ^ "}"

let string_of_vdecl (t, id, i) =
    match i with
    | Noinit -> string_of_typ t "^ "^ id ^"
    | _ -> string_of_typ t "^ "^ id ^" = "^ string_of_initer i ^"

let rec string_of_stmt = function
    | Block(decns, stmts, ctxt) ->
      let enclosers = function PointContext -> ("{","}"") | LineContext -> ("[","]")
      | TriangleContext -> ("<"",">"") in
      let opener, closer = enclosers ctxt in
      opener ^ "\n" ^ String.concat "" (List.map string_of_vdecl (decns)) ^ "\n"
      ^ String.concat "" (List.map string_of_stmt stmts) ^ closer ^ "\n"
      | Expr(expr) -> string_of_expr expr ^ "\n"
      | Return(expr) -> "return " ^ string_of_expr expr ^ "\n"

| If(e, s, Block([],[],_,_)) ->
  | If(e, s1, s2) -> "if " ^ string_of_expr e ^ "\n"
  | string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2

| For(e1, e2, e3, s) ->
  | ForDec(d1, e2, e3, s) ->
    "for " ^ string_of_expr e1 ^ " ; " ^ string_of_expr e2 ^ " ; " ^ string_of_expr e3 ^ "\n"
    ^ string_of_stmt s

| While(e, s) -> "while " ^ string_of_expr e ^ "\n"

| Timeloop(id1, e1, id2, e2, s) ->
  | FrameLoop(id1, e1, id2, e2, s) ->
    "timeloop " ^ id1 ^ " ; " ^ string_of_expr e2 ^ "\n"
    ^ string_of_stmt s

let string_rettyp f =
    match f.typ with
    | Constructor -> ""
    | _ -> string_of_typ f.rettyp

let string_fname f =
    match f.typ with
    | Func -> f.fname
    | _ -> f.owner ^":"^ f.fname

let string_of_fdecl f =
    string_rettyp f ^ " " ^ string_fname f ^ " ( " ^ String.concat "" (List.map string_of_fbind f.params) ^ " )\n"
let string_of_stmt s = string_of_token s
^ String.concat "" (List.map string_of_bind s.decls) ^ "" ^ String.concat "" (List.map string_of_fdecl (s.ctor::s.methods))

let string_of_utype s = string_of_stosh s.ss ^ "" ^ String.concat "" (List.map string_of_bind s.decls) ^ "" ^ String.concat "" (List.map string_of_fdecl (s.ctor::s.methods))

let string_of_program p =
  String.concat "" (List.map string_of_vdecl p.v) ^
  String.concat "" (List.map string_of_utype p.s) ^
  String.concat "" (List.map string_of_fdecl p.f)

let default_ctr n = { rettype = Void; fname = n; params = []; locals = []; body = []; typ = Constructor; owner = n }

(* Const expression evaluation *)

let get_int = function
  | IntLit(i) -> 1
  | e -> raise(Failure((string_of_baseexpr e)^" is not an int literal")

let get_char = function
  | CharLit(c) -> 1
  | e -> raise(Failure((string_of_baseexpr e)^" is not a char literal")

let get_float = function
  | FloatLit(f) -> 1
  | e -> raise(Failure((string_of_baseexpr e)^" is not a double literal")

let get_string = function
  | StringLit(s) -> s
  | e -> raise(Failure((string_of_baseexpr e)^" is not an string literal")

let get_vec = function
  | VecLit(a,b) -> (a,b)
  | e -> raise(Failure((string_of_baseexpr e)^" is not an vec literal")

(* convert one type of literal to another *)

let rec do_lit_promote (e,_) src trg = match (src,trg) with
  | (Int, Float) -> (FloatLit(float(get_int e)),Float)
  | (Int, Vec) -> (do_lit_promote (do_lit_promote (e,Int) Int Float) Float Vec)
  | (Float, Vec) -> (VecLit(get_float e, get_float e), Vec)
  | (t1, t2) when t1 = t2 -> (e,t1) (* No op promotion *)

  | _ ->
    raise(Failure("Can not convert literal of type "^(string_of_typ src)^" to "^(string_of_typ trg)")

let fail_op t op = raise(Failure("No operator "^(string_of_op op)^" defined for typ e "^(string_of_typ t))

let fail_uop t op = raise(Failure("No operator "^(string_of_op op)^" defined for t ype "^(string_of_typ t))

let bti b = if b then 1 else 0

(* binop on two const expressions of the same type *)

let do_binop (e1,t1) op (e2,_) =
  match(op) with
  | Add -> if t1 = Int then (IntLit((get_int e1) + (get_int e2)),Int)
  | else if t1 = Float then (FloatLit((get_float e1) +. (get_float e2)),Float)
  | else if t1 = Vec then (let v1 = get_vec e1 and v2 = get_vec e2 in
    VecLit( (fst v1) +. (fst v2) , (snd v1) +. (snd v2) ),Vec)
    else fail_op t1 op
  | Sub -> if t1 = Int then (IntLit((get_int e1) - (get_int e2)),Int)
  | else if t1 = Float then (FloatLit((get_float e1) -. (get_float e2)),Float)
  | else if t1 = Vec then (let v1 = get_vec e1 and v2 = get_vec e2 in
    VecLit( (fst v1) -. (fst v2) , (snd v1) -. (snd v2) ),Vec)
    else fail_op t1 op
  | Mult -> if t1 = Int then (IntLit((get_int e1) * (get_int e2)),Int)
    else fail_op t1 op
  | Div -> if t1 = Int then (IntLit((get_int e1) ~/ (get_int e2)),Int)
    else fail_op t1 op
  | Rem -> if t1 = Int then (IntLit((get_int e1) % (get_int e2)),Int)
    else fail_op t1 op
  | _ -> raise(Failure("Can not convert literal of type "^(string_of_typ src)^" to "^(string_of_typ trg)")
else if t1 = Float then (FloatLit((get_float e1) *. (get_float e2)), Float)
else if t1 = Vec then (let v1 = get_vec e1 and v2 = get_vec e2 in
(Veclit((fst v1) *. (fst v2), (snd v1) *. (snd v2)), Vec)
else fail_op t1 op
| Div -> if t1 = Int then (IntLit((get_int e1) / (get_int e2)), Int)
else if t1 = Float then (FloatLit((get_float e1) /. (get_float e2)), Float)
else if t1 = Vec then (let v1 = get_vec e1 and v2 = get_vec e2 in
(Veclit((fst v1) /. (fst v2), (snd v1) /. (snd v2)), Vec)
else fail_op t1 op
| Mod -> if t1 = Int then (IntLit((get_int e1) mod (get_int e2)), Int)
else fail_op t1 op
| Equal -> if t1 = Int then (IntLit(bti((get_int e1) = (get_int e2))), Int)
else if t1 = Float then (IntLit(bti((get_float e1) = (get_float e2))), Int)
else if t1 = Vec then (let v1 = get_vec e1 and v2 = get_vec e2 in
(IntLit(bti((fst v1) = (fst v2)) && ((snd v1) = (snd v2))), Int)
else fail_op t1 op
| Neq -> if t1 = Int then (IntLit(bti((get_int e1) <> (get_int e2))), Int)
else if t1 = Float then (IntLit(bti((get_float e1) <> (get_float e2))), Int)
else if t1 = Vec then (let v1 = get_vec e1 and v2 = get_vec e2 in
(IntLit(bti((fst v1) <> (fst v2)) || ((snd v1) <> (snd v2))), Int)
else fail_op t1 op
| Less -> if t1 = Int then (IntLit(bti((get_int e1) < (get_int e2))), Int)
else if t1 = Float then (IntLit(bti((get_float e1) < (get_float e2))), Int)
else fail_op t1 op
| Leq -> if t1 = Int then (IntLit(bti((get_int e1) <= (get_int e2))), Int)
else if t1 = Float then (IntLit(bti((get_float e1) <= (get_float e2))), Int)
else fail_op t1 op
| Greater -> if t1 = Int then (IntLit(bti((get_int e1) > (get_int e2))), Int)
else if t1 = Float then (IntLit(bti((get_float e1) > (get_float e2))), Int)
else fail_op t1 op
| Geq -> if t1 = Int then (IntLit(bti((get_int e1) >= (get_int e2))), Int)
else if t1 = Float then (IntLit(bti((get_float e1) >= (get_float e2))), Int)
else fail_op t1 op
| And -
  > if t1 = Int then (IntLit(bti((get_int e1) & (get_int e2))), Int)
  else fail_op t1 op
| Or -
  > if t1 = Int then (IntLit(bti((get_int e1) | (get_int e2))), Int)
  else fail_op t1 op

(* unary operators on const_expr *)
let do_uop op (e1,t1) = match op with
| Neg -> if t1 = Int then (IntLit((-get_int e1))), Int)
| else if t1 = Float then (FloatLit(-.(get_float e1)), Float)
| else if t1 = Vec then (let v1 = get_vec e1 in
  (Veclit(-.(fst v1), -.(snd v1)), Vec)
| ) else fail_uop t1 op
| Pos -> (e1,t1)
| Not -> if t1 = Int then ((IntLit(if (get_int e1) = 0 then 1 else 0)), Int)
| else fail_uop t1 op
| _ -> raise(Failure("Non-const expression "^\"(string_of_expr (e1,t1))\" where constexpr is expected."))

(* Evaluate const_expr *)
let rec const_expr = function
  (IntLit(_, as i, _)) -> (i, Int)
| (CharLit(_, as c, _)) -> (c, Char)
| (StringLit(_, as s, _)) -> (s, String)
| (FloatLit(_, as f, _)) -> (f, Float)
| (Veclit(_, as v, _)) -> (v, Vec)
| (Vecexpr(e1, e2, _)) ->
  let (e1', _) = const_expr (Promote(const_expr e1), Float)
  and (e2', _) = const_expr (Promote(const_expr e2), Float)
  in (Veclit(get_float e1', get_float e2'), Vec)
| (Binop(e1, t1, op, e2, t2, _)) ->
let (e1,t1) = const_expr(e1,t1) and (e2,t2) = const_expr(e2,t2) in
  if t1 = t2 then let e1' = const_expr (e1,t1)
                   and e2' = const_expr (e2,t2) in do_binop e1' op e2'
  else if (t1=Int && t2=Float) then let e1' = const_expr (Promote(const_expr (e1,
                                               t1)),t2)
                   and e2' = const_expr (e2,t2) in do_binop e1' op e2'
  else if (t2=Vec && t1=Float) then let e2' = const_expr (Promote(const_expr (e2,
                                               t2)),t1)
                   and e1' = const_expr (e1,t1) in do_binop e1' op e2'
  else if (t2=Vec && op=Mult) then let e2' = const_expr (Promote(const_expr (e2,t
                                               2)),t1)
                   and e1' = const_expr (e1,t1) in do_binop e1' op e2'
  else raise(Failure("No operator "^(string_of_op op)^" defined for types "^(string_of_typ t1)^" and "^(string_of_typ t2))")

| (Unop (op,e), _) -> do_uop op (const_expr e)
| (Index(le,rt),_,t) -
  > let (le',lt) = (const_expr le) and (re',rt) = (const_expr re) in
  > if rt = Int then
    > let i = (get_int re') in
    > (match le' with
    >     VecLit (a,b) -
    >       > if i = 0 then (FloatLit(a),Float) else if i = 1 then (FloatLit(b),Float) else raise(Failure("Index expression "^(string_of_typ (le',lt))" where const_expr is expected."))
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |
    >     |...
8.5 Code Generator

File name: codegen.ml

1. (* Translate takes AST and produces LLVM IR *)
2. 3. module L = LLVM
4. module A = Ast
5. 6. module StringMap = Map.Make(String)
7. module StringSet = Set.Make( struct let compare = Pervasives.compare type t = string end )
8. 9. (* scope types *)
10. type scope = GlobalScope | LocalScope | StructScope | ClosureScope
11. 12. let translate prog =
13. 14. (* Get the global variables and functions *)
15. let globals = prog.A.v (* Use this format when referencing records in other modules *)
16. and functions = prog.A.f
17. and structs = prog.A.s in
18. 19. (* Set up LLVM module and context *)
20. let context = L.global_context () in
21. let the_module = L.create_module context "ART" and i32_t = L.i32_type context
22. and i64_t = L.i64_type context and i8_t = L.i8_type context
23. and void_t = L.void_type context and double_t = L.double_type context
24. in
25. let string_t = L.pointer_type i8_t and i8ptr_t = L.pointer_type i8_t and i8ptrptr_t = L.pointer_type (L.pointer_type i8_t)
26. and vec_t = L.vector_type double_t 2
27. in
28. 29. (* General version of llttype_of_typ that takes a struct map *)
30. (* The struct map helps to get member types for structs in terms of previously *)
31. 32. (* defined structs *)
33. let rec _ltype_of_typ m = function
34. | A.Int -> i32_t
35. | A.Char -> i8_t
36. | A.Void -> void_t
37. | A.Float -> double_t
38. | A.String -> string_t
39. | A.Vec -> vec_t
40. | A.Array(t,e) -> (match (A.const_expr e) with
41. | (A.IntLit(i),_) -> L.array_type (_ltype_of_typ m t) i
42. | _ -> raise(Failure "Arrays declaration requires int expression"))
43. | A.UserType(s,_) -> StringMap.find s m
44. in
45. 46. (* Defining each of the structs *)
47. (* struct_ltypes is a map from struct names to their corresponding LLVM type. *)
48. (* It's used by ltype_of_typ. It has to be defined this way to allow structs to *)
49. (* have member whose type is of previously defined structs *)
50. let struct_ltypes =
51. let struct_ltype m st =
(*) Ocaml Array containing llvm lltype of the member variables *)
let decls_array = Array.of_list (List.rev (List.fold_left
  (fun l (t,_) -> (const_of_typ m t)::l) [] st.A.decls)
(* Define the llvm struct type *)
in let named_struct = L.named_struct_type context st.A.sname
in L.struct_set_body named_struct decls_array false; (* false -
  > not packed *)
StringMap.add st.A.sname named_struct m in
List.fold_left struct_ltype StringMap.empty structs
in
(* Function takes ast types and returns corresponding llvm type *)
let ltype_of_typ t = _ltype_of_typ struct_ltypes t
in
(* llvm value of a literal expression *)
let lvalue_of_lit = function
  A.Intlit i -> L.const_int i32_t i
  | A.CharLit c -> L.const_int i8_t (int_of_char c) (* 1 byte characters *)
  | A.Noexpr -> L.const_int i32_t 0 (* No expression is 0 *)
  | A.StringLit s -
    > let t = L.define_global "unnamed." (L.const_stringz context s) the_module in
    L.const_bitcast (L.const_gep i1 (L.const_int i32_t 0)) i8pt
    r_t
  | A.FloatLit f -> L.const_int double_t f
  | A.VecLit(f1, f2) -
    > L.const_vec [|(L.const_float double_t f1); (L.const_float double_t f2)]
  | _ -> raise(Failure("Attempt to initialize global variable with a non-
    const"))
let const_null t = if t = A.String then (lvalue_of_lit (A.StringLit "") else L.
  const_null(ltype_of_typ t)
let rec construct_initer t = function
  A.Exprbind e -> lvalue_of_lit (fst(A.const_expr e))
  | A.Listinit il -> (match t with
    A.Array(t2,e) -> let len = A.get_int(fst(A.const_expr e)) in
    l1 = List.map (construct_initer t2) il in
    let (i,l) = List.fold_left (fun (c,il) m -
      if c < len then (c+1,m::il) else (c,il)) (0,[]) l in
    let il = expand_list i len (List.rev l) t2 in
    L.const_array (ltype_of_typ t2)(Array.of_list l)
  |
    A.UserType(n,_) ->
      (* type of members *)
    let dtl = List.map (fun (t,_)-> t) ((List.find (fun s -
      > s.A.sname = n) prog.A.ss).A.decls ) in
    L.const_named_struct (ltype_of_typ t) (Array.of_list(List.map2 cons-
    struct_initer dtl il))
    | _ -> raise(Failure("Nested initializer cannot be used with "^A.string_of_typ t)))
  )
| A.Noinit -> const_null t
in
let global_vars =
  let expand_list i max l t = A._expand_list i max l const_null
in
let rec construct_initer t = function
  A.Exprbind e -> lvalue_of_lit (fst(A.const_expr e))
  | A.Listinit il -> (match t with
    A.Array(t2,e) -> let len = A.get_int(fst(A.const_expr e)) in
    l1 = List.map (construct_initer t2) il in
    let (i,l) = List.fold_left (fun (c,il) m -
      if c < len then (c+1,m::il) else (c,il)) (0,[]) l in
    let il = expand_list i len (List.rev l) t2 in
    L.const_array (ltype_of_typ t2)(Array.of_list l)
  |
    A.UserType(n,_) ->
      (* type of members *)
    let dtl = List.map (fun (t,_)-> t) ((List.find (fun s -
      > s.A.sname = n) prog.A.ss).A.decls ) in
    L.const_named_struct (ltype_of_typ t) (Array.of_list(List.map2 cons-
    struct_initer dtl il))
    | _ -> raise(Failure("Nested initializer cannot be used with "^A.string_of_typ t)))
  )
| A.Noinit -> const_null t
in
let global_vars =
  let expand_list i max l t = A._expand_list i max l const_null
in
let rec construct_initer t = function
  A.Exprbind e -> lvalue_of_lit (fst(A.const_expr e))
  | A.Listinit il -> (match t with
    A.Array(t2,e) -> let len = A.get_int(fst(A.const_expr e)) in
    l1 = List.map (construct_initer t2) il in
    let (i,l) = List.fold_left (fun (c,il) m -
      if c < len then (c+1,m::il) else (c,il)) (0,[]) l in
    let il = expand_list i len (List.rev l) t2 in
    L.const_array (ltype_of_typ t2)(Array.of_list l)
  |
    A.UserType(n,_) ->
      (* type of members *)
    let dtl = List.map (fun (t,_)-> t) ((List.find (fun s -
      > s.A.sname = n) prog.A.ss).A.decls ) in
    L.const_named_struct (ltype_of_typ t) (Array.of_list(List.map2 cons-
    struct_initer dtl il))
    | _ -> raise(Failure("Nested initializer cannot be used with "^A.string_of_typ t)))
  )
| A.Noinit -> const_null t
in
let timeval_struct_t = let g = L.named_struct_type context "timeval"

in ignore(L.struct_set_body g [[i64_t; i64_t]] false); g

(* Declare printf() *)
(* Allowing a print builtin function for debuggin purposes *)
let printf_t = L.var_arg_function_type i32_t [ L.pointer_type i8_t ]

(* GLUT FUNCTION ARG TYPES *)
and glut_init_t = L.function_type void_t [ L.pointer_type i32_t; L.pointer_type
(L.pointer_type i8_t)]
and glut_crwin_t = L.function_type i32_t [ L.pointer_type i8_t ]
and get_tday_t = L.function_type i32_t [ L.pointer_type timeval_struct_t; L.
pointer_type i8_t[]]
and void_void_t = L.function_type void_t[[]]
and void_i8p_t = L.function_type void_t[i8ptr_t[]]
and void_int_t = L.function_type void_t[i32_t[]]
and void_int_int_t = L.function_type void_t[i32_t; i32_t[]]
and void_2d_t = L.function_type void_t[double_t; double_t[]]
and void_3d_t = L.function_type void_t[double_t; double_t; double_t[]]
and double_double_t = L.function_type double_t[double_t[]]
and double_2d_t = L.function_type double_t[double_t; double_t[]]

in let void_callback_t = L.function_type void_t[i32_t]

(* END OF GLUT ARG TYPES *)
let printf_func = L.declare_function "printf" printf_t the_module
and get_tday_func = L.declare_function "gettimeofday" get_tday_t the_module

(* GLUT FUNCTION DECLARATIONS *)
and glutinit_func = L.declare_function "glutInit" glut_init_t the_module
and glutinitdmode_func = L.declare_function "glutInitDisplayMode" void_int_t the_module
and glutinitwpos_func = L.declare_function "glutInitWindowPosition" void_int_i
nt_t the_module
and glutinitwsiz_func = L.declare_function "glutInitWindowSize" void_int_t the_module
and glutcreatwin_func = L.declare_function "glutCreateWindow" glut_crwin_t the
module
and glutdisplay_func = L.declare_function "glutDisplayFunc" void_callback_t the
module
and gluIdle_func = L.declare_function "glutIdleFunc" void_callback_t the
module
and glutsetopt_func = L.declare_function "glutSetOption" void_int_t the_module
and glutmainloop_func = L.declare_function "glutMainLoop" void_void_t the_module

(* NON BOILER PLATE FUNCTIONS *)
and glColor_func = L.declare_function "glColor3d" void_3d_t the_module
and glBegin_func = L.declare_function "glBegin" void_int_t the_module
and glVertex_func = L.declare_function "glVertex2d" void_2d_t the_module
and glEnd_func = L.declare_function "glEnd" void_void_t the_module
and glclear_func = L.declare_function "glClear" void_int_t the_module
and glswap_func = L.declare_function "glutSwapBuffers" void_void_t the_module
and glutlvmain_func = L.declare_function "glutLeaveMainLoop" void_void_t the_module
and glutrepost_func = L.declare_function "glutPostRedisplay" void_void_t the_module

/* technically not glut */
and sin_func = L.declare_function "sin" double_double_t the_module
and cos_func = L.declare_function "cos" double_double_t the_module
and tan_func = L.declare_function "tan" double_double_t the_module
and logfunc = L.declare_function "log" double_double_t the_module
and log2func = L.declare_function "log2" double_double_t the_module
and log10func = L.declare_function "log10" double_double_t the_module
and absfunc = L.declare_function "fabs" double_double_t the_module
and expfunc = L.declare_function "exp" double_double_t the_module
and sqrtfunc = L.declare_function "sqrt" double_double_t the_module
and asin_func = L.declare_function "asin" double_double_t the_module
and acos_func = L.declare_function "acos" double_double_t the_module
and atan_func = L.declare_function "atan" double_double_t the_module
and sinh_func = L.declare_function "sinh" double_double_t the_module
and cosh_func = L.declare_function "cosh" double_double_t the_module
and tanh_func = L.declare_function "tanh" double_double_t the_module
and asinh_func = L.declare_function "asinh" double_double_t the_module
and acosh_func = L.declare_function "acosh" double_double_t the_module
and atanh_func = L.declare_function "atanh" double_double_t the_module
and powfunc = L.declare_function "pow" double_2d_t the_module

(* END OF GLUT FUNCTION DECLARATIONS *)

(* Defining each of the declared functions *)

(* Function decs is a map from function names to tuples of llvm function representation *)
and Ast declarations *)
let function_decls =
  let function_decl m fdecl =
    let name = fdecl.A.fname
    and formal_types = (* Types of parameters in llvm type repr *)
    Array.of_list (List.map (fun (t,_,pass) ->
      let lt = ltype_of_typ t in
      match pass with
      | A.Value -> lt
      | A.Ref -
      > L.pointer_type lt (* If pass by reference use pointers *)
    ) fdecl.A.params)
    in let ftyper = L.function_type (ltype_of_typ fdecl.A.rettyp) formal_types
    StringMap.add name (L.define_function_name ftyper the_module, fdecl) m in

(* ADD THE GLUT FUNCTIONS HERE WITH DECLARATION *)

let glut_decls =
  let glut_decl m artname def = StringMap.add artname m.def m in
  List.fold_left2 glut_decl StringMap.empty
  ["setcolor";"vertex";"sin";"cos";"tan";"log";"log2";"log10";"abs";"exp";"sqrt";
  "asin";"acos";"atan";"sinh";"cosh";"tanh";"asinh";"acosh";"atanh";"pow";"drawpoint"]
    [glcolor_func;glvertex_func;sin_func;cos_func;tan_func;log_func;
    log2_func;log10_func;abs_func;exp_func;sqrt_func;asin_func;acos_func;atan_func;...]
```plaintext
216.   \text{sinh\_func};\text{cosh\_func};\text{tanh\_func};\text{asinh\_func};\text{acosh\_func};\text{atanh\_func};\text{pow\_func};\text{L}\.\text{constant_32_t}\ 0]
217.   \text{in}
218.   \text{(* No type checking done *)}
219.   \text{let do\_glut\_func}\ f\text{decl}\  act\_builder =
220.     \text{if} \text{gcolor\_func} \Rightarrow f\text{decl} \text{then} \text{L\_build\_call} \text{gcolor\_func} \, [[\text{act}(\theta) ; \text{act}(1) ; \text{act}(2)]] "" \text{builder}
221.     \text{else if} \text{glvertex\_func} \Rightarrow f\text{decl} \text{then} \text{L\_build\_call} \text{glvertex\_func} \, [[\text{act}(\theta) ; \text{act}(1) ]] "" \text{builder}
222.     \text{else if} \text{pow\_func} \Rightarrow f\text{decl} \text{then} \text{L\_build\_call} f\text{decl} \, [[\text{act}(\theta) ; \text{act}(1) ] "" \text{builder}
223.     \text{else if} \text{L\_constant_32_t}\ 0 \Rightarrow f\text{decl} \text{then} \text{L\_build\_call} f\text{decl} \, [[\text{act}(\theta) ] "" \text{builder}
224.     \text{(* Draw Point - draws vertices *)}
225.     \text{else} \text{L\_build\_call} \text{glvertex\_func} \, [[\text{act}(0) ; \text{act}(1) ] "" \text{builder}
226.   \text{in}
227.   \text{let do\_glut\_init}\ \text{argc}\ \text{argv}\ \text{title}\ \text{draw\_func}\ \text{idle\_func}\ \text{builder} =
228.     \text{let const} = \text{L\_constant_32_t}
229.     \text{in} \text{ignore(\text{L\_build\_call} \text{glutinit\_func} \, [[\text{argc}; \text{argv}]])} "" \text{builder}
230.     \text{ignore(\text{L\_build\_call} \text{glutinit\_mode\_func} \, [[\text{const} 2]])} "" \text{builder}
231.     \text{ignore(\text{L\_build\_call} \text{glutinit\_twpos\_func} \, [[\text{const} 100; \text{const} 200]])} "" \text{builder}
232.     \text{ignore(\text{L\_build\_call} \text{glutinit\_wsiz\_func} \, [[\text{const} 800; \text{const} 600]])} "" \text{builder}
233.     \text{ignore(\text{L\_build\_call} \text{glutcreate\_win\_func} \, [[\text{title}])} "" \text{builder}
234.     \text{ignore(\text{L\_build\_call} \text{glutdisplay\_func} \, [[\text{draw\_func}])} "" \text{builder}
235.     \text{ignore(\text{L\_build\_call} \text{glut\_idle\_func} \, [[\text{idle\_func}])} "" \text{builder}
236.     \text{ignore(\text{L\_build\_call} \text{glut\_set\_opt\_func} \, [[\text{const} 0x01F9; \text{const} 1])} "" \text{builder}
237.     \text{L\_build\_call} \text{glut\_main\_loop\_func} \, [[\text{}]) "" \text{builder}
238.   \text{in}
239.   \text{(* Map from struct names to tuples (member variable map, methods map, ll\_ltype) *)}
240.   \text{let struct\_decls} =
241.     \text{(* struct\_decl takes a map and an ast sdecl and returns a map which contains}
242.     \text{sdecl added *)}
243.     \text{let struct\_decl} m sdecl =
244.     \text{let \text{l\_ltype} = l\_type\_of\_typ (A\_User\_Type(sdecl.A.sname, sdecl.A.ss))} \text{in}
245.     \text{(* Map from struct member variable name to (ast type, index) *)}
246.     \text{(A\_User\_Type(sdecl.A.sname, sdecl.A.ss))} \text{in}
247.     \text{(* Note: members in the ast have variant type "bind = typ * string" *)}
248.     \text{let var\_map =}
249.     \text{let var\_index = List\_rev ( snd (List\_fold\_left (fun (i,l) (t,n) -
250.       ( i+1,(n,t,i)::l) ) (\theta[,][]) sdecl.A.decls ) \text{in}}
251.     \text{List\_fold\_left ( fun vm (n,t,i) -
252.       \text{StringMap\_add n (t,i) vm }) StringMap\_empty var\_index}
253.     \text{in}
254.     \text{(* Map from method/construct name to (ll\_vm func type, fdecl).*)}
255.     \text{(* Similar to the function\_decls map.*)}
256.     \text{let method\_map =}
257.     \text{let method\_decl} m fdecl =
258.     \text{let name = fdecl.A.fname}
259.     \text{(* Append a pointer to the current struct type to parameter list.}
260.     \text{It will be used as a this pointer. Method calls always implicitly}
261.     \text{pass the caller as first argument *)}
262.     \text{and formal\_types = Array\_of\_list}
```
264.  ((L.pointer_type lstype)::(List.map (fun t, _, pass) ->
265.    let lt = ltype_of_typ t in
266.    match pass with
267.    | A.Value -> lt
268.    | A.Ref -
269.    > L.pointer_type lt (* If pass by reference use pointers *)
270.    (* NOTE: Return type for constructor is Void *)
271.    let ret_type = ltype_of_typ fdecl.A.rettyp
272.    in let ftype = L.function_type ret_type formal_types in
273.    StringMap.add name (L.define_function name ftype the_module, fdecl) m in
274.    List.fold_left method_decl StringMap.empty (sdecl.A.ctor::sdecl.A.methods)
275.    in
276.    StringMap.add sdecl.A.sname (varmap, methodmap, lstype) m in
277.    List.fold_left struct_decl StringMap.empty structs
278.    in
279.    (* Returns (fdef, fdecl) for a method in constructor *)
280.    let lookup_method sname fname =
281.    let (_,_,methodmap,_) = StringMap.find fname struct_decls in
282.    StringMap.find sname methodmap
283.    in
284.    (* Returns index of member memb in struct named sname *)
285.    let memb_index_type sname memb =
286.    (* Obtain map from struct_decls map *)
287.    let (varmap, _,_) = try StringMap.find sname struct_decls
288.    with Not_found -
289.    > raise (Failure("Varmap not found for : ":"^sname^":"^memb"))
290.    in
291.    (* Obtain index from varmap *)
292.    try StringMap.find memb varmap
293.    with Not_found -> raise (Failure("Index not found for : ":"^sname^":"^memb"))
294.    in
295.    let memb_index sname memb = snd (memb_index_type sname memb)
296.    in
297.    let memb_type sname memb = fst (memb_index_type sname memb)
298.    in
299.    (* Fill in the body of the given function *)
300.    (* This is general as it takes lets user specify function_decls map *)
301.    let rec _build_function_body fdecl function_decls closure_scopes =
302.    (* Get the corresponding llvm function value *)
303.    let (the_function, _) = (match fdecl.A.typ with
304.      | A.Func -> StringMap.find fdecl.A.fname function_decls
305.      | _ -> lookup_method fdecl.A.owner fdecl.A.fname
306.    ) in
307.    (* Get an instruction builder that will emit instructions in the current function *)
308.    let builder = L.builder_at_end context (L.entry_block the_function) in
309.    (* Checks if function is a draw shape method *)
310.    let is_draw_shape fdecl = (fdecl.A.typ = A.Method) && (fdecl.A.fname = "draw")
311.    ((let s = List.find (fun s -
312.      > s.A.sname = fdecl.A.owner) structs in s.A.ss = A.ShapeType)
313.    in
314.    (* call a closing glend in a shape_draw before before emitting return *)
315.    let build_custom_ret_void builder =
316.      ( if is_draw_shape fdecl
317.        then ignore(L.build_call glend_func [ | | ] " builder) else ());
318.    L.build_ret_void builder
(" For unique globals, use a name that ends with '.', *")

(* NOTE: there can only be one variable name "foo." that is a unique global *)

let unique_global n init = match (L.lookup_global n the_module) with
  Some g -> g
| None -> L.define_global n init the_module

(* For unique global stringptrs, use a name that ends with '.' *)

(* NOTE: there can only be one variable name "foo." that is a unique global stringptr *)

let unique_global_stringptr str n = match (L.lookup_global n the_module) with
  Some g -> L.const_bitcast g i8ptr_t
| None -> L.build_global_stringptr str n builder

in closbl = unique_global "closbl." (L.const_null i8ptrptr_t)

in

(* Format strings for printf call *)

let int_format_str = unique_global_stringptr "%d" "ifmt." in

let char_format_str = unique_global_stringptr "%c" "cfmt." in

let string_format_str = unique_global_stringptr "%s" "sfmt." in

let float_format_str = unique_global_stringptr "%f" "ffmt." in

(* GLUT RELATED *)

let time_value = L.define_global "tv" (L.const_null timevlaue_struct_t) the_module in

(* Need to define an actual argc. dummy_arg_1 is now the address of argc *)

let dummy_arg_1 = L.define_global "argc" (L.const_int i32_t 1) the_module in

(* The first element of argv *)

let glut_argv_0 = L.const_bitcast (unique_global_stringptr "ART" "glutstr. ") i8ptr_t

in

(* Second element of argv *)

let glut_argv_1 = (L.const_null i8ptr_t) in

(* The argv object itself *)

let dummy_arg_2 = L.define_global "glutargv" (L.const_array i8ptr_t [glut argv_0; glut argv_1]) the_module

in

let g_last_time = unique_global "g_last_time." (L.const_null double_t) in

let g_delay = unique_global "g_delay." (L.const_null double_t) in

let g_steps = unique_global "g_steps." (L.const_int i32_t 0) in

let g_maxiter = unique_global "g_maxiter." (L.const_int i32_t 0) in

(* END OF GLUT RELATED *)

(* Add shape array *)

let shape struct =

  let named_struct = L.named_struct_type context "shape_struct." in
  L.struct_set_body named_struct ([ i8ptr_t ; L.pointer_type void i8p_t ])
 | false ; named struct

in

let shape_list = unique_global "shape_list." (L.const_array shape_struct (Array.make 1000 (L.const_null shape structure)))

in

let shape_list_ind = unique_global "shape_list_ind." (L.const_int i32_t 0)

in

let tloop_on = unique_global "tloop_on." (L.const_int i32_t 0)

in

let draw_enabled = unique_global "draw_enabled." (L.const_int i32_t 0)

in
let do_seconds_call builder =
  let secptr = ignore(builder | L.build_call get_tday_func [
    [time_value; l.const_int 1] builder];
  L.build_gep time_value [
    [l.const_int 132_t 0; l.const_int 132_t 0]]"sec"
  builder in
  let usecptr = l.build_gep time_value [
    [l.const_int 132_t 0; l.const_int 1]
  32_t 1]"usec"
  builder in
  let sec = l.build_sitofp (l.build_load secptr "tmp" builder) double_t "tm"p
  builder
  and usec = l.build_sitofp (l.build_load usecptr "tmp" builder) double_t "tmp"
  builder
  in
  let usecsec = l.build_fmul usec (l.const_float double_t 1.0e-6) "tmp" builder in
  L.build_fadd sec usecsec "seconds" builder

in
let get_unique_draw_func () =
  let draw_func = L.define_function "draw." void_void_t the_module in
  let builder = L.builder_at_end (L.entry_block draw_func) in
  ignore(L.build_store (l.const_int 132_t 1) draw_enabled builder);
  let i = ignore(L.build_call glclear_func [
    [l.const_int 132_t 0]
x4000]"" builder);
  L.build_alloca 132_t "drawl" builder in
  ignore (L.build_store (l.const_int 132_t 0) i builder);
  let pred_bb = L.append_block context "while" draw_func in
  ignore (L.build_br pred_bb builder); (* builder is in block that contains while stm *)

  (* Body Block *)
  let body_bb = L.append_block context "while_body" draw_func in
  let body_builder = L.builder_at_end context body_bb in
  let ib = L.build_load i "ib" body_builder in
  (* get the shape *)
  let shape = L.build_load (L.build_gep shape_list [] [l.const_int 132_t 0; ib]
    [l.const_int 132_t 0]) "slp" body_builder in
  (* get the function pointer *)
  let drawshape = L.build_load (L.build_gep shape_list [] [l.const_int 132_t 0; ib]
    [l.const_int 132_t 1]) "slfp" body_builder in
  ignore(L.build_call drawshape [] shape []"" body_builder);
  (* increment i *)
  ignore (L.build_store (L.build_add ib (l.const_int 132_t 1) "ib" body_builder) i body_builder);
  ignore(L.build_br pred_bb body_builder);

  let pred_builder = L.builder_at_end context pred_bb in
  let bool_val =
    let nmax = L.build_load shape_list_ind "sindex" pred_builder in
  let ip = L.build_load i "ip" pred_builder in
  L.build icmp l.1 icmp.1lt ip nmax "tp" pred_builder
  in
  let merge_bb = L.append_block context "merge" draw_func in
  ignore (L.build_call draw_func body_builder merge_bb pred_builder);
  let merge_builder = L.builder_at_end context merge_bb in
  ignore(L.build_call glswap_func [] [] merge_builder);
  ignore(L.build_store (l.const_int 132_t 0) draw_enabled merge_builder);
  ignore(L.build_ret_void merge_builder); draw_func

  (match (L.lookup_function "draw." the_module) with
    Some f -> f
    | None -> get_draw_func)

in
let get_idle_func looptp loop_func =
  let idle_stop_condition steps idle_func target_bb builder =
    let bool_val = L.buildicmp L.Icmp.Sge steps (L.buildload g_maxiter "maxiter" builder) "idlestcond" builder in
    let merge_bb = L.append_block context "merge" idle_func in (* Merge block *)
  let merge_builder = L.builderatend context merge_bb in
  let then_bb = L.appendblock context "then" idle_func in
  let then_builder = L.builderatend context then_bb in
  ignore (L.buildcondbr bool_val then_bb merge_bb builder);
  ignore (L.buildcall glutlvmain_func [[]] "then" builder);
  ignore (L.buildbr merge_bb then_builder);
  ignore (L.buildcall glutrepost_func [[]] "merge_builder" builder);
  ignore (L.buildbr target_bb merge_builder);
  merge_bb

  let merge_bb = L.appendblock context "merge" idle_func in (* Merge block *)
  let merge_builder = L.builderatend context merge_bb in
  let then_bb = L.appendblock context "then" idle_func in
  let then_builder = L.builderatend context then_bb in
  ignore (L.buildcondbr bool_val then_bb merge_bb builder);
  ignore (L.buildcall glutlvmain_func [[]] "then" builder);
  ignore (L.buildbr merge_bb then_builder);
  ignore (L.buildcall glutrepost_func [[]] "merge_builder" builder);
  ignore (L.buildbr target_bb merge_builder);
  merge_bb

  let merge_bb = L.appendblock context "merge" idle_func in (* Merge block *)
  let merge_builder = L.builderatend context merge_bb in
  let then_bb = L.appendblock context "then" idle_func in
  let then_builder = L.builderatend context then_bb in
  ignore (L.buildcondbr bool_val then_bb merge_bb builder);
  ignore (L.buildcall glutlvmain_func [[]] "then" builder);
  ignore (L.buildbr merge_bb then_builder);
  ignore (L.buildcall glutrepost_func [[]] "merge_builder" builder);
  ignore (L.buildbr target_bb merge_builder);
  merge_bb

  let merge_bb = L.appendblock context "merge" idle_func in (* Merge block *)
  let merge_builder = L.builderatend context merge_bb in
  let then_bb = L.appendblock context "then" idle_func in
  let then_builder = L.builderatend context then_bb in
  ignore (L.buildcondbr bool_val then_bb merge_bb builder);
  ignore (L.buildcall glutlvmain_func [[]] "then" builder);
  ignore (L.buildbr merge_bb then_builder);
  ignore (L.buildcall glutrepost_func [[]] "merge_builder" builder);
  ignore (L.buildbr target_bb merge_builder);
  merge_bb

  let merge_bb = L.appendblock context "merge" idle_func in (* Merge block *)
  let merge_builder = L.builderatend context merge_bb in
  let then_bb = L.appendblock context "then" idle_func in
  let then_builder = L.builderatend context then_bb in
  ignore (L.buildcondbr bool_val then_bb merge_bb builder);
  ignore (L.buildcall glutlvmain_func [[]] "then" builder);
  ignore (L.buildbr merge_bb then_builder);
  ignore (L.buildcall glutrepost_func [[]] "merge_builder" builder);
  ignore (L.buildbr target_bb merge_builder);
  merge_bb

  let merge_bb = L.appendblock context "merge" idle_func in (* Merge block *)
  let merge_builder = L.builderatend context merge_bb in
  let then_bb = L.appendblock context "then" idle_func in
  let then_builder = L.builderatend context then_bb in
  ignore (L.buildcondbr bool_val then_bb merge_bb builder);
  ignore (L.buildcall glutlvmain_func [[]] "then" builder);
  ignore (L.buildbr merge_bb then_builder);
  ignore (L.buildcall glutrepost_func [[]] "merge_builder" builder);
  ignore (L.buildbr target_bb merge_builder);
  merge_bb
528.  | (locls, LocalScope) -
529.    > (try ignore(StringMap.find n locls);LocalScope
530.      > scope_iter n (List.tl scopes))
531.        with Not_found -
532.      > (_ , StructScope) -
533.      > raise (Failure ("No struct scope in closure"))
534.    > (try ignore(StringMap.find n locls);ClosureScope
535.      > scope_iter n (List.tl scopes))
536.        with Not_found -
537.      |
538.    in
539.    let rec non_local_expr (e,_) = non_local_baseexpr e
540.    and non_local_baseexpr = function
541.    A.Id s -
542.      > (try (if (scope_iter s scopes)>LocalScope then StringSet.empty else StringSet.singleton s)
543.      |
544.        with Not_found -> StringSet.singleton s)
545.    > StringSet.union (non_local_expr e1) (non_local_expr e2)
546.      > A.Binop(e1,_,e2) -
547.      > StringSet.union (non_local_expr e1) (non_local_expr e2)
548.      > A.Asno(e1,_,e2) -
549.      > StringSet.union (non_local_expr e1) (non_local_expr e2)
550.      > A.Unop(_,e) -> non_local_expr e
551.      > A.Posop(_,e) -> non_local_expr e
552.      > A.Call(e1,e1) -> StringSet.union (match e with (A.Id _,_) -
553.      > StringSet.empty | _ -> non_local_expr e1)
554.      |
555.      (List.fold_left (fun set e-
556.    > StringSet.union (non_local_expr e) StringSet.empty e1)
557.      > A.Index(e1, e2) -
558.      > StringSet.union (non_local_expr e1) (non_local_expr e2)
559.      > A.Member(e, _) | A.Promote(e) -> non_local_expr e
560.      > _ -> StringSet.empty
561.      |
562.    in
563.    let rec non_local = function
564.    A.Expr(e) -> non_block_locals decls stmts scopes
565.    A.Return(e) -> non_local_expr e
566.    A.If(e,s1,s2) -
567.      > StringSet.union (non_local_expr e) (StringSet.union (non_local s1) (non_local s2)
568.    )
569.      > A.For(e1,e2,e3,s) -
570.      > StringSet.union (StringSet.union (non_local_expr e1) (non_local_expr e2))
571.      (StringSet.union (non_local_expr e3) (non_local l s))
572.      > A.ForDec(decls,e2,e3,s)-
573.      > non_local ( A.Block(decls, [A.For((A.Noexpr,A.Void), e2, e3, s)],A.PointContext))
574.      |
575.    A.While(e, s) -> StringSet.union (non_local_expr e) (non_local s)
576.      > StringSet.union (non_local_expr e1) (non_local l s))
577.      > A.Timeloop (_,e1,_,e2,s) -> StringSet.union (non_local_expr e1)
578.      (StringSet.union (non_local_expr e2) (non_local l s))
579.      |
580.    in
581.    let rec non_local_initer = function
582.    A.Exprinit e -> non_local_expr e
583.    A.Listinit il -> List.fold_left (fun set i-
584.      > StringSet.union set (non_local_initer i)) StringSet.empty il
585.      > A.Noinit -> StringSet.empty
586.      |
587.    in
588.    StringSet.union
589.      (List.fold_left (fun set s-
590.      > StringSet.union set (non_local s)) StringSet.empty stmt_list)
529. (list.fold_left (fun set (_,i) ->
   > StringSet.union set (non_local_initer i)) StringSet.empty local_decls)
530.     in
531.   (* Returns (fdef, fdecl) for a function call in method/function *)
532.   (* Handles case of constructor call, a method call without dot operator and
533.      normal function call *)
534.   (* Second case can happen only within struct scope when calling a struct's
535.      method
536.      from another method of the same struct *)
537.   let lookup_function f =
538.      (* First try to find a matching constructor *)
539.      try lookup_method f f
540.      (* If that fails try to find a method.
541.      this is guaranteed to fail in a normal function *)
542.      with Not_found -> (try lookup_method fdecl.A.owner f
543.      (* Finally look for normal function *)
544.      with Not_found -> StringMap.find f function_decls)
545.   in
546.   (* Construct the function's formal arguments. Allocate each on the stack, i
547.      initialize their
548.      value, and remember their values in the "formals" map *)
549.   (* NOTE: the functions top level local vars are constructed in the build_b
550.      ock_body. This means formal vars (params)
551.      are check after top level local var during lookup, even though they are s
552.      emantically at the same level. While local hiding
553.      formal is technically possible it is prohibited during the semantic check
554.      stage. *)
555.   let formal_vars =
556.     (* Arguments: map (type, name, pass) param (llvm of params]*)
557.     (* Add formal m (t, n,pass) p = L.set_value_name n p;
558.     (* name appended with nums as necessary: eg a1, a2 *)
559.     (* Look at microc lecture: pg 39 *)
560.     let local = match pass with
561.     A.Value -
562.     > L.build_alloc (ltype_of_typ t) n builder (* allocate stack for value params *)
563.     | A.Ref -> p
564.     in
565.     ignore (match pass with
566.     A.Value -
567.     > ignore(l.build_store p local builder); (* Store the value param in the allocated p
568.    lace *)
569.     | A.Ref -> ()
570.     StringMap.add n (local,t) m in (* We add the stack version *)
571.     let lparams = (match fdecl.A.typ with
572.     (* LLVM type list for the params *)
573.     let lparams = (match fdecl.A.typ with
574.      A.Func -> Array.to_list (l.params the_function)
575.      (* For Method/Const drop the "this" param as it needs to be inacc
576.      essible to user *)
577.      _ -> List.tl (Array.to_list (l.params the_function)))
578.     in
579.     List.fold_left2 add_formal StringMap.empty fdecl.A.params lparams
580.     in
581.     (* Invoke "f builder" if the current block doesn't already
582.      have a terminal (e.g., a branch). *)
583.     let add_terminal builder f =
584.     match L.block_terminator (l.insertion_block builder) with
585.     Some _ -> ()
586.     | None -> ignore (f builder)
(* Build the body of a code execution block and return builder *)
let rec build_block_body (local_decls, stmt_list) builder scopes =

(* Prepend the block local variables to the scopes list *)
(* Prepend any initializers to the statement list *)
let (stmt_list, scopes) =
  (* takes the Ivalue ast expression le to be evaluated and the initializer *)
  let rec construct_initer (le,t) = function
  A.Exprinit e -> let e = (try A.const_expr e with Failure _ - e) in
  [A.Expr( A.Asnop((le,t),A.Asn, e),t)]

  A.Array(t2,e) -
  > let arrlen = A.get_int(fst(A.const_expr e)) in
  (* Expand the initializer as appropriate *)
  let il = A._expand_l_list (List.length il) arrlen il t2 nul
  l_initer in
  (* Construct each initer upto the length of the array *)
  fst (List.fold_left (fun (il2,c) -> if c < arrlen then (il2@construct_initer (A.Index((le,t),(A.IntLit c,A.Int)),t2) init ) , c+1)
  else (il2,c)) ([]0) il

  (* decls list*)
  let ml = ((List.find ( fun s - > s.A.sname = n ) prog.A.s).A.decls ) in
  List.fold_left2 (fun il2 (t2,n) init - > il2@construct_initer (A.Member((le,t),n),t2) init )) [] ml il

  raise(Failure("Nested initializer cannot be used with ",^A.s.string_of_typ t))

  | (*A.Noinit*)_ -> []

  in
  let add_local m (t,n) =
  (* Currently all block local variables are allocated at the entry block
  This prevents multiple allocas for loop local variables. The only issue with this
  method is that in the output file, variables are allocated in reverse order *)
  (* NOTE this translation should be moved to the semantic part of the code *)
  let builder = L.builder_at context (L.instr_begin(L.entry_block the_function)) in
  let local_var = L.buildalloca (ltype_of_typ t) n builder (* allocate space for local *)
  in StringMap.add n (local_var,t) m

  in
  let stmt_list = (List.fold_left (fun l (t,n,i) - > l@construct_initer (A.Id(n),t)) [] local_decls)@stmt_list
  and local_decls = List.map (fun (t,n,_) -> (t,n)) local_decls in
  (*List.fold_left

  (* Handle expression initters by adding them as assignment exp
  ression statments *)

  (fun (sl, ld) (t,n,i) ->
    ( match i with A.Exprinit e - > A.Expr( A.Asnop((A.Id(n),t),A.Asn, e),t)::sl , (t,n)::ld
    | _ - > sl, (t,n)::ld (* Silently ignore NoInit and ListInit *)

    )
  ) (stmt_list, [])
(* Need to reverse since we are pushing to top of stmt_list. Luckily, the fold unreversed local_decls *)

(List.rev local_decls)

let locals = List.fold_left add_local StringMap.empty local_decls

in stmt_list,(locals, LocalScope)::scopes

let string_create s builder = L.build_global_stringptr s "temp" builder

in

(* Return the value for a variable by going through the scope chain *)

let rec _lookup n builder scopes =
  let succ ci =(*successor of const_int i32_t in int*)
    let int_of_const ci = (match (L.int64_of_const ci) with
      Some i -> Int64.to_int(i)
      | None -> 0 )
    in (int_of_const ci) + 1
  in
  let hd = List.hd scopes in
  (match hd with
    (globs, GlobalScope) -> fst(StringMap.find n globs)
    | (locls, LocalScope) -> (try fst(StringMap.find n locls)
        with Not_found ->
        _lookup n builder (List.tl scopes) )
    | (_, StructScope) -> (try {
      let ind = try memb_index fdecl.A.owner n with Failure _ -
        raise Not_found in
      (* If member, get its pointer by dereferencing the first argument
        corresponding to the "this" pointer *)
      let e' = L.param (fst (lookup_function fdecl.A.fname)) 0 in
      L.build_gep e' [L.const_int i32_t 0; L.const_int i32_t ind | ] "tmp" builder
    ) with Not_found -> _lookup n builder (List.tl scopes) )
    | (locls, ClosureScope) -> try
      let (i, t) = StringMap.find n locls in
      let closbl = L.build_bitcast (L.build_load ("clostbptr" builder) (L.pointer_type(L.array_type i8ptr_t (succ i))) "clostblarr" builder in
      let ppt = L.build_load (L.build_gep closbl [ L.const_int i32_t 0; i[]] "valppt" builder) "valppt" builder in
      L.build_bitcast ppt (L.pointer_type (ltype_o f_typ t)) "valppt" builder
    ) with Not_found -
    _lookup n builder (List.tl scopes)
  )

in

(* Return the type for a variable by going through the scope chain *)

let rec _lookup_type n scopes =
  let hd = List.hd scopes in
  (match hd with
    (globs, GlobalScope) -> snd(StringMap.find n globs)
    | (locls, LocalScope) -> (try snd(StringMap.find n locls)
        with Not_found ->
        _lookup_type n (List.tl scopes) )
    | (_, StructScope) -> try memb_type fdecl.A.owner n
        with Failure _ -
    _lookup_type n (List.tl scopes)
  )
let lookup n builder = _lookup n builder scopes

let lookup_type n = _lookup_type n scopes

(* Like string_of_typ but prints "circle" instead of "shape circle" *)

let string_of_typ2 = function
  A.UserType(s, _) -> s
  | t -> A.string_of_typ t

(* Returns a tuple (type name, ast type) for an expression *)
(* In final code [with semantic analysis] the ast type should be part of expr *)

let expr_type (_,t) = (string_of_typ2 t, t)

(* Adds an int_cast to the llvmop *)
let bit_to_int llvmop v1 v2 s builder =
  let bv = llvmop v1 v2 s builder in
  let i1 = L.build_extractelement bv (L.const_int i32_t 0) "i1" builder
  and i2 = L.build_extractelement bv (L.const_int i32_t 1) "i2" builder in
  let ir = iop i1 i2 "ir" builder in
  L.build_zext_or_bitcast ir i32_t s builder

if typ=A.Vec
  then match op with
  | A.Add -> L.build_fadd
  | A.Sub -> L.build_fsub
  | A.Mult -> L.build_fmul
  | A.Div -> L.build_fdiv
  | A.Equal -> vec_cmp (L.build_fcmpeq L.Fcmp.Oeq) L.build_and
  | A.Neq -> vec_cmp (L.build_fcmpeq L.Fcmp.One) L.build_or
  | _ -> raise (Failure ("Unsupported binary operation for vec: ":"^A.string_of_op(op)))

else if typ=A.Float
  then match op with
  | A.Add -> L.build_fadd
  | A.Sub -> L.build_fsub
  | A.Mult -> L.build_fmul
  | A.Div -> L.build_fdiv
  | A.Mod -> raise (Failure "Cannot mod a float")
  (* Need to think about these ops *)
  | A.Equal -> bit_to_int (L.build_fcmpeq L.Fcmp.Oeq)
  | A.Neq -> bit_to_int (L.build_fcmpeq L.Fcmp.One)
  | A.Less -> bit_to_int (L.build_fcmpeq L.Fcmp.Olt)
  | A.Leq -> bit_to_int (L.build_fcmpeq L.Fcmp.Ole)
  | A.Greater -> bit_to_int (L.build_fcmpeq L.Fcmp.Ogt)
  | A.Geq -> bit_to_int (L.build_fcmpeq L.Fcmp.Oge)
  | _ -> raise (Failure ("Unsupported binary operation for float: ":"^A.string_of_op(op)))

else match op with
A.Add -> L.build_add
A.Sub -> L.build_sub
A.Mult -> L.build_mul
A.Div -> L.build_sdiv
A.And -> L.build_and
A.Or -> L.build_or
A.Mod -> L.build_srem
A.Equal -> bit_to_int (L.build_icmp L.Icmp.Eq)
A.Neq -> bit_to_int (L.build_icmp L.Icmp.Ne)
A.Less -> bit_to_int (L.build_icmp L.Icmp.Slt)
A.Leq -> bit_to_int (L.build_icmp L.Icmp.Sle)
A.Greater -> bit_to_int (L.build_icmp L.Icmp.Sgt)
A.Geq -> bit_to_int (L.build_icmp L.Icmp.Sge)
A.FloorDiv -> L.build_sdiv

(* Construct code for an lvalue; return a pointer to access object *)
let rec lexpr builder (e,_) = lbaseexpr builder e
and lbaseexpr builder = function
| A.Id s -> lookup s builder
| A.Index(e1,e2) -> let e2' = expr builder e2 in
  (match e1 with
    (A.Id _,_); (A.Index(_,_),_) | (A.Member(_,_),_) ->
      L.build_gep (lexpr builder e1) [[L.const_int 132_t 0; e2']]
      "tmp" builder
| _  -> let e1' = expr builder e1 in (* e1 should be indexible *)
  let tmp = L.buildalloca (l.type_of e1') "indtmp" builder in
  ignore (L.build_store e1' tmp builder);
  L.build_gep tmp [[L.const_int 132_t 0; e2']] "tmp" builder
| A.Member(e, s) -> let e' = slexpr builder e in
  (* Obtain index of s in the struct type of expression e *)
  let (sname, _) = expr_type e in let i = mem_index sname s in
  L.build_gep e' [[L.const_int 132_t 0; L.const_int 132_t i]] "tmp" builder
| e  -> raise (Failure ("r-value provided where l-value expected: "^ (A.string_of_baseexpr e)))

(* Special handling for member access to struct calls *)
and slexpr builder = function
| A.Call(_,_),t as e ->
  (* Create local temporary to hold newly created struct *)
  let loc = L.buildalloca (ltype_of_typ t) "tmp" builder in
  ignore(L.build_store (expr builder e) loc builder);
  (* Return the adress of local temporary *)
  loc
| e -> lexpr builder e

(* Construct code for an expression; return its value *)
and expr builder = function
| A.Promote (e,tt),t) ->
  (match (st,tt) with
    (a,b) when a==b -> expr builder (e,tt)
    | (A.Int,A.Float) ->
      L.build_sitofp (expr builder (e,tt)) double_t "tmp" builder
    | (A.Int,A.Bool) ->
      L.build_store (expr builder (e,tt)) tt
    | (A.Float,A.Bool) -> let e' = (expr builder (e,tt)) in
      L.build_sdiv double_t 0.0 [] in
    | (L.const float double_t 0.0 [] in
      let i1 = L.build_insertelement vec e' (L.const_int 132_t 0) "tmp1" builder in
    | (A.Bool,A.Bool) ->
      let e1 = (expr builder (e,tt)) in
      L.Icmp.Eq e1 e
    | (A.Bool,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Float) ->
      L.Icmp.Eq e1 e
    | (A.Float,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Bool) ->
      L.Icmp.Eq e1 e
    | (A.Bool,A.Int) ->
      L.Icmp.Eq e1 e
    | (A.Int,A.Bool) ->
      L.Icm...
L.build_insertelement il e' (L.const_int i32_t 1) "tmp2" builder

    | _ -
    -> raise (Failure ("Unsupported promotion from "^\(A.string\_of\_typ\_st\)^" to "^\(A.string\_of\_typ\_tt\)))

    | (e_,) -> baseexpr builder e
    and baseexpr builder = function (* Takes args builder and Ast.expr *)
    | A.IntLit i -> L.const_int i32_t i
    | A.CharLit c -
    > L.const_int i8_t (int_of_char c) (* 1 byte characters *)
    | A.Noexpr -> L.const_int i32_t 0 (* No expression is 0 *)
    | A.StringLit s -> string_create s builder
    | A.FloatLit f -> L.const_float double_t f
    | A.VecLit(f1, f2) -
    > L.const_vector [(L.const_float double_t f1); (L.const_float double_t f2)]
    | A.Vecexpr(e1, e2) ->
    | let e1' = expr builder e1
    and e2' = expr builder e2
    in
    | let tmp_vec = L.const_vector [(L.const_float double_t 0.0); L.const_float double_t 0.0]
    in
    | let insert_element1 = L.build_insertelement tmp_vec e1' (L.const_int i32_t 0) "tmp1" builder
    in
    | L.build_insertelement insert_element1 e2' (L.const_int i32_t 1) "tmp2" builder
    | A.Id s -
    > L.build_load (lookup s builder) s builder (* Load the variable into register and register *)
    | A.Binop (e1, op, e2) ->
    | let e1' = expr builder e1
    and e2' = expr builder e2
    in binop_of_type (expr_type e1) op e1' e2' "tmp" builder
    | A.Index(_, _) as arr -
    > L.build_load (lbaseexpr builder arr) "tmp" builder
    | A.Member(_, _) as mem -
    > L.build_load (lbaseexpr builder mem) "tmp" builder
    | A.Asnop (el, op, er) ->
    | let el' = lexpr builder el in
    (match op with
    | A.Asn -> let e' = expr builder er in
    ignore (L.build_store e' el' builder); e'
    | A.CmpAsn bop ->
    | let e' =
    | let el' = L.build_load el' "ltmp" builder
    and e2' = expr builder er
    in binop_of_type (expr_type el) bop e1' e2' "tmp" builder
    in ignore (L.build_store e' el' builder); e'
    )
    | A.Unop(op, e) ->
    (match op with
    | A.Neg ->
    (match snd(expr_type e) with A.Int -> L.build_neg
    | A.Float | A.Vec -> L.build_fneg | t -
    -> raise (Failure ("Negation not supported for "^\(A.string\_of\_typ(t))
    ) (expr builder e) "tmp" builder
    | A.Not -
    -> expr builder (A.Binop((A.IntLit(0), A.Int), A.Equal, e), A.Int)
844. | A.Pos  ->  expr builder e
845. | A.Preinc | A.Predec  ->  let e' = lexpr builder e in
846.       let ev = L.build_load e' "tmp" builder in
847.       let ev = (if op = A.Preinc then L.build_add else L.build_sub) ev (L.const_int i32_t 1) "tmp" builder in
848.       ignore(L.build_store ev e' builder); ev
849.       )
850. | A.Posop(op,e)  ->  let e' = lexpr builder e in
851.       let ev = L.build_load e' "tmp" builder in
852.       let ev2 = (if op = A.Postinc then L.build_add else L.build_sub) ev (L.const_int i32_t 1) "tmp" builder in
853.       ignore(L.build_store ev2 e' builder); ev
854.       (* This ok only for few built in functions *)
855.       | A.Call ((A.Id "printf", _),[e]) -
856.       > L.build_call printf_func [[int_format_str ; (expr builder e) ]] "printf" builder
857.       | A.Call ((A.Id "printf", _),[e]) -
858.       > L.build_call printf_func [[char_format_str ; (expr builder e) ]] "printf" builder
859.       | A.Call ((A.Id "addshape", _),el) ->
860.       let add_one_shape ex =
861.       let fdef' = L.const_bitcast (fst(lookup_method (fst(expr_type ex))) "draw") (L.pointer_type void_i8p_t) in
862.       let i = L.build_load shape_list_ind "index" builder in
863.       let ex' = L.build_bitcast (slexpr builder ex) i8p_t "shp" builder in
864.       (* store the shape *)
865.       ignore( L.build_store ex' (L.build_gep shape_list [] [ L.const_int i32_t 0 ; i ; L.const_int i32_t 0 ] ) "slp" builder) builder);
866.       (* store the function *)
867.       ignore( L.build_store fdef' (L.build_gep shape_list [ [ L.const_int i32_t 0 ; i ; L.const_int i32_t 1 ] ] ) "slfp" builder) builder);
868.       (* increment i *)
869.       ignore( L.build_store (L.build_add i (L.const_int i32_t 1) "tmp" builder) shape_list_ind builder) ();
870.       in ignore(List.iter add_one_shape el); L.undef void_t
871.       (* A call without a dot expression refers to three possibilities. In order of precedence: *)
872.       (* constructor call, method call (within struct scope), function call *)
873.       | A.Call ((A.Id f,_,) , act) ->
874.       (* The llvm type array of the calling functions parameters
875.       Can be use to retrive the "this" argument *)
876.       (try let fdef = StringMap.find f glut_decls in
877.       let actuals =
878.       if f = "drawpoint" (* Convert vector into two arguments *)
879.       then let v = (List.hd act) in list_map (expr builder) [ (A.Index(v,(A.IntLit(0),A.Int)),A.Float); (A.Index(v,(A.IntLit(1),A.Int)), A.Float)]
880.       else List.rev (list_map (expr builder) (List.rev act)) in
881.       do glut_func fdef (Array.of_list actuals) builder
882.       with Not_found -> (]
883.       let myparams = L.params (fst (lookup_function fdecl.A.fname) ) in
884.       let (fdef, fdecl) = lookup_function f in
885.       (* Helper function for pass by value handling *)
886.       let arg_passer builder (,_,pass) = function
887.       (* Identifier, index, and member expressions may be passed by r
888.       Other types are required to be passed by value. *)
(A.Id(_,_)) | (A.Index(_,_),_)) | (A.Member(_,_),_) as e -
> (match pass with

A.Ref -
> lexpr builder e (* This gets the pointer to the variable *)
  | A.Value -> expr builder e
in
(* This makes right to left evaluation order. What order should we use? *)
> let actuals = List.rev (List.map2 (arg_passer builder) (List.rev fdecl.A.params)) (List.rev act)
> let result = (match fdecl.A.rettyp with A.Void -
> "" (* don't name result for void llvm issue")
> (* How the function is called depends on the type *)
> (match fdecl.A.typ with

A.Func -
> L.build_call fdef (Array.of_list actuals) result builder
(* For methods pass value of the callers "this" argument *)
> | A.Method -
> L.build_call fdef (Array.of_list (myparams.(0) :: actuals)) result builder
(* Constructors are called like methods but evaluate to their own type
not their return value as they don't have explicit return values *)
> | A.Constructor -
> let (_,_,lstype) = StringMap.find f struct_decls
(* Create local temporary to hold newly created struct *)
in let loc = L.build_alloca lstype "tmp" builder in
(* Pass the newly created struct as the "this" argument of constructor call *)
ignore( L.build_call fdef (Array.of_list (loc :: actuals)) result builder);
(* Return the initialized local temporary *)
L.build_load loc "tmp" builder
)
>
(* Explicit method calls with dot operator *)
> | A.Call ((A.Member(e,s),_), act) ->
let (sname, _) = expr_type e in
let (fdef, fdecl) = lookup_method sname e in
(* Helper function for pass by value handling *)
(* Same us the above code *)
> let arg_passer builder (_,_,pass) = function
(A.Id(_,_),_)) | (A.Index(_,_,_) | (A.Member(_,_,_) as e -
> (match pass with

A.Ref -
> lexpr builder e (* This gets the pointer to the variable *)
  | A.Value -> expr builder e
in
(* This makes right to left evaluation order. What order should we use? *)
> let actuals = List.rev (List.map2 (arg_passer builder) (List.rev fdecl.A.params)) (List.rev act)
(* Append the left side of dot operator to arguments so it is used as a "this" argument *)
> let actuals = (slexpr builder e :: actuals in
let result = (match fdecl.A.rettyp with A.Void -
> "" (* don't name result for void llvm issue")
> L.build_call fdef (Array.of_list actuals) result builder
> raise (Failure ("Promote should be handled in expr not bexpr"))
in

(* Build the code for the given statement; return the builder for
the statement's successor *)
let rec stmt builder = function
A.Block (vl, sl, ctxt) ->
  (* Handle context start *)
let glcontext = function A.PointContext -> 0 | A.LineContext ->
  1 | A.TriangleContext -> 4 in
let contexti = L.const_int i32_t (glcontext ctxt) in
let builder =
  (* start a new context if necessary.*)
  (if ctxt = A.PointContext then () (* Point context is noop *)
    else (ignore(L.build_call glbegin_func []) " builder);
      ignore(L.build_call glbegin_func [L.const_int 132_t 0]
      ) " builder))
); build_block_body (vl, sl) builder scopes
in
  (if ctxt = A.PointContext then () (* Point context is noop *)
    else (ignore(L.build_call glbegin_func []) " builder);
      ignore(L.build_call glbegin_func [L.const_int 132_t 0]
      ) " builder))
); builder

  | A.Expr e -
  > ignore (expr builder e); builder (* Simply evaluate expression *)
  |
  | A.Return e -
  > ignore (match fdecl.A.rettyp with (* Different cases for void and non-void *)
  A.Void -> build_custom_ret_void builder
  | _ -> L.build_ret (expr builder e) builder;
  | A.If (predicate, then_stmt, else_stmt) ->
  let bool_val = (L.build_icmp L.Icmp.Ne)(expr builder predicate)
  (L.const_int i32_t 0) "itob" builder in
  let merge_bb = L.append_block context "merge" the_function in
  (* Merge block *)
  let then_bb = L.append_block context "then" the_function in
  (* Get the builder for the then block, emit then_stmt and then
  add branch statement to
  merge block *)
  add_terminal (stmt (L.builder_at_end context then_bb then_stmt
  )
  (L.build_br merge_bb);
  |
  let else_bb = L.append_block context "else" the_function in
  add_terminal (stmt (L.builder_at_end context else_bb else_stmt
  )
  (L.build_br merge_bb);
  (* add_terminal used to avoid insert two terminals to basic blo
  cks *)
  |
  (* builder is in block that contains if stmt *)
  ignore (L.build_cond_br bool_val then_bb else_bb builder);
  L.builder_at_end context merge_bb (* Return builder at end of m
  erge block *)
  |
  A.While (predicate, body) ->
  let pred_bb = L.append_block context "while" the_function in
  ignore (L.build_br pred_bb builder); (* builder is in block tha
t contains while stmt *)
  |
  let body_bb = L.append_block context "while_body" the_function in
  add_terminal (stmt (L.builder_at_end context body_bb body) body)
let pred_builder = L.builder_at_end context pred_bb in
let bool_val = (L.build_icip L.Icmp.Eq) (expr pred_builder predicate) (L.const_int i32_t 0) "itob" pred_builder in
let merge_bb = L.append_block context "merge" the_function in
ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder)

L.builder_at_end context merge_bb

(* make equivalent while *)
| A.For (e1, e2, e3, body) -> stmt builder
| A.Block ([], [ A.Expr e1; A.While (e2, A.Block ([]), [body; A.Expr e3], A.PointContext) ]) , A.PointContext)
| A.ForDec (vdecls, e2, e3, body) -> stmt builder (A.Block(vdecls, [A.For((A.Noexpr, A.Void), e2, e3, body)], A.PointContext))
| A.Timeloop(_, e1, _, e2, stmt) | A.FrameLoop(_, e1, _, e2, stmt) as s->

let loop = (match s with A.Timeloop(_,_,_,_,_) ->
> @ | --> 1) in (* @ for time and 1 for frame *)
let typedef = i1 define_function "timeloop." type the_module e in
let loopdecl = {A.retyp = A.Void ; A.fname = "timeloop." ; A.params = []}

A.locals = []({[(A.Float, s1,A.ExprInit( e1));(A.Float, s2, A.ExprInit(e2))]: this is done in semantic}*)
A.body = [stmt]; A.typ = A.Func ; A.owne r=""} in
let fdecls = StringMap.add "timeloop." (fdef, loopdecl) function_decls in
let outnames =
let non_block_locals loopdecl.A.locals
in

(* Check to see if the tloopen flag is active *)
let t_off = (L.build_icip L.Icmp.Eq)(L.build_load tloop_on "tflag" builder)

let e1' = expr builder e1 in let e2' = expr builder e2 in

let zero = L.const_float double_t 0.0 in

(* fps/dt => 0 *)
let cond1 = L.build_and t_off (L.build_fcmp L.Fcmp.Olt e1' zero "le1" builder) "cond1" builder in

(* frames/time > 0 *)
let cond2 = L.build_and cond1 (L.build_fcmp L.Fcmp.Oge e2' zero "le1" builder) "cond1" builder in

let merge_bb = L.append_block context "merge" the_function in
let then_bb = L.append_block context "then" the_function in

let else_bb = L.append_block context "else" the_function in

add_terminal (L.builder_at_end context else_bb) (L.builder_at_end context merge_bb)

(* add_terminal used to avoid insert two terminals to basic blocks *)

(* builder is in block that contains if stmt *)
ignore (L.build_cond_br cond2 then_bb else_bb builder);

let builder = L.builder_at_end context then_bb in

let table = (* Mark timeloop active flag *)
ignore(L.build_store (L.const_int i32_t 1) t
loop_on builder);

L.build_array_malloc i8ptr_t (L.const_int i3
2_t (List.length outnames)) "ctable" builder

let tablearr = L.build_bitcast table (L.pointer_type(L.a
rray_type i8ptr_t (List.length outnames))) "ctablearr" builder

let colstblptr = L.build_bitcast table i8ptrptr_t "toclo
stbl" builder

(* Closure build and teardown is expensive *)
let (closure_map, _) = ignore(L.build_store
clostblptr colstbl builder);

let add_to_closure (m,i) n =
    let valpointer = L.build_bitcast
(lookup n builder) i8ptr_t "valp" builder

in ignore(L.build_store valpointer pospointer builde
r);

(StringMap.add n (L.const_int i32_t i, lookup_type n
) m, i+1)
in List.fold_left add_to_closure (StringMap.empty, 0)

outnames in
ignore(_build_function_body loopdecl fdecls [(closure_m
ap, ClosureScope)]);

(* Setup the timer and stepper values *)
ignore(L.build_store (do_seconds_call builder) g_last_ti
me builder);

let delayflt = if loop = 0 then e1' else L.build_fdiv (L
.const_float double_t 1.0) e1' "dfstps" builder in

ignore(L.build_store delayflt g_delay builder);
ignore(L.build_store (L.const_int i32_t 0) g_steps build
er);

let stepsflt = if loop = 0 then L.build_fdiv e2' e1' "st
pft" builder else e2' in

ignore(L.build_store (L.build_fptoui stepsflt i32_t "ste
psint" builder) g_maxiter builder);

ignore(do_glut_init dummy_arg_1 (L.const_bitcast dummy_a
rg_2 i8ptrptr_t) glut_argv_0 (get_unique_draw_func()) (get_idl
e_func loop fdef)builder);

ignore(L.build_free table builder);
ignore(L.build_store (L.const_int i32_t 0) tloop_on buil
der);
ignore(L.build_store (L.const_int i32_t 0) shape_list_in
d builder);
add_terminal builder (L.build_br merge_bb);

L.builder_at_end context merge_bb (* Return builder at e
nd of merge block *)

(* Build the code for each statement in the block
and return the builder *)
List.fold_left stmt builder stmt_list

(* End of build_block_body *)
(* Construct the scopes list before calling build_block_body *)
let scopes_list = match fdecl.A.typ with
  A.Func -
    > closure_scopes@[(formal_vars, LocalScope); (global_vars, GlobalScope)]
  (_) -
    > closure_scopes@[(formal_vars, LocalScope); (StringMap.empty, StructScope); (global_vars, GlobalScope)]

(* Functions can't access members so no struct scope *)

So we are using an empty map *

|returns out of a draw shape if no timeloop is active *)

let handle_draw_shape builder =
  let bool_val = (* Check to see if the tloop_on flag is active *)
  (L.build_icmp L.Icmp.Eq)(L.build_load draw_enabled builder)

  (L.const_int i32_t 0) "d_on" builder

  let merge_bb = L.append_block context "merge" the_function in (* Merge block *)

  let then_bb = L.append_block context "then" the_function in

  add_terminal (L.builder_at_end context then_bb)

  (L.build_ret_void);

  (* add_terminal used to avoid insert two terminals to basic blocks *)

  let else_bb = L.append_block context "else" the_function in

  add_terminal (L.builder_at_end context else_bb)

  (L.build_br merge_bb);

  (* builder is in block that contains if stmt *)

  ignore (L.build_cond_br bool_val then_bb else_bb builder);

  L.builder_at_end context merge_bb

  in

  let builder =

  let builder' = if is_draw_shape fdecl then handle_draw_shape builder else builder

  in

  (* Add glbegin to beginning of shape draw methods *)

  (if (is_draw_shape fdecl)
    then ignore(L.build_call glbegin_func [|L.const_int i32 _t @ |] "" builder')
    else ()
  );

  build_block_body (fdecl.A.locals, fdecl.A.body) builder' scopes_list

  in

  (* Add a return if the last block falls off the end *)

  add_terminal builder (match fdecl.A.rettyp with
    A.Void -> build_custom_ret_void
    | t -> L.build_ret (const_null t))

  in

  (* old build_function_body *)

  let build_function_body fdecl = _build_function_body fdecl function_decls []

  in

  List.iter build_function_body functions;

  (* Build methods and constructors for each struct *)
List.iter (fun sdecl -
   > list.iter build_function_body (sdecl.A.ctor::sdecl.A.methods)) structs;
1127.
   the_module
8.6 Semantic Checker

File name: semant.mll

1. open Ast

2. module StringMap = Map.Make(String)

3. type scope = GlobalScope | LocalScope | StructScope

4. (* Semantic checking of a program. Returns possibly modified Ast if successful, throws an exception if something is wrong. *)

5. let report_dup exceptf list =
   let rec helper = function
     n1 :: n2 :: _ when n1=n2 -> raise (Failure(exceptf n1))
   | _ :: t -> helper t
   [] ->()

6. in helper(List.sort compare list)

7. (* lftype, rtype, errmsg *)

8. let check_ass lval rval promote err =
   if lval = rval || (promote && lval = Float && rval = Int)
   then lval else raise err

9. in

10. let struct_build prog =

11.   let globals = (Float,"PI",Exprinit (FloatLit 3.141592653589793,Float))::

12.     (Float,"E",Exprinit (FloatLit 2.718281828459045,Float))::prog.v

13. and functions = {

14.     rettyp = Void; fname="setcolor"; params=[[(UserType("color",StructType),"c",V

15.         alue)]; locals=[];

16.     body = [Expr (Call ( (Id "setcolor."),Void), [ (Member((Id "c", Void),"r"),Flo

17.         at); (Member((Id "c", Void),"g"),Float);(Member((Id "c", Void),"b"),Float))],Void]

18.         ;

19.     typ=Func;owner="None"::prog.f

20. and structlist = {

21.     ss = StructType;

22.     name = "color";

23.     decls = [(Float,"r"),(Float,"g"),(Float,"b")];

24.     ctor = { rettyp=Void; fname="color":params=[(Float,"ri",Value);(Float


26.       locals=[]; body = [Expr (Asnop (Id "r",Float), Asn,(Id "ri" ,Float ) ), Float ];

27.       Expr (Asnop (Id "g",Float), Asn,(Id "gi" ,Float ) ), Float );

28.       Expr (Asnop (Id "b",Float), Asn,(Id "bi" ,Float ) ), Float ]); typ=Constructor; owner="color" } ;

29. methods = [];

30. } :: prog.s (* adding color type here *)

31. in

32. (* A map of all struct/shape types *)

33. let structs = List.fold_left (fun m st -> report_dup(fun n -

34.     "Duplicate member variable named " ^n " in "]^ (string_of_stosh st.ss)^ "^st.name)(List.map (fun (_,n) -

35.     n)st.decls);

36. (List.iter(fun (t,_)->

37.     (match t with

38.         UserType(s,ss) -> if(st.name = s)

39.         then raise(Failure("Cannot nest "^(string_of_stosh st.ss)^ "^st

40.         .name" within itself")))

41.         else (try if (StringMap.find s m).ss != ss then raise Not_found w

42.                 ith Not_found ->

43.                 try if (StringMap.find s m).ss != ss then raise Not_found w
let structs,funcs = (* Refers to structs and non-member functions *)
(* Puts methods and constructors with appropriate struct and returns tuple
(map, bool) *)
let filter_func m f =
  match f.typ with
  | Func -> (m, true) (* true means keep function *)
  | Constructor ->
    let s = try,StringMap.find f.owner m
    with Not_found -
      > raise (Failure("Constructor of undefined struct/shape: " ^ f.owner^"::"^f.fname))
    in
      if f.fname <> f.owner then raise (Failure("Constructor must have same name as struct/shape: " ^ f.owner^"::"^f.fname))
      else if (s.ctor.fname='') then (StringMap.add s.sname
        {ss = s.ss;sname = s.sname; decls = s.decls; ctor = f; methods = s.methods} m , false)
      else
        raise(Failure("Multiple constructor definitions for "^s.fname))
    |
    Method -> let s = try,StringMap.find f.owner m
    with Not_found -
      > raise (Failure("Method of undefined struct/shape: " ^ f.owner^"::"^f.fname))
    in try ignore( list.find (fun f2 -
      > f2.fname = f.fname) s.methods);
    raise(Failure("Multiple definitions for method " ^f.fname))
  |
  try StringMap.add s.sname
    {ss = s.ss;sname = s.sname; decls = s.decls; ctor = s.ctor; methods = f::s.methods} m , false)
  in
  List.fold_left ( fun (m,l) f -> let (m, cond) = filter_func m f in
    if cond then (m, f::l) else (m, l) ) (structs, [] ) functions
  in
  { s = List.map (fun st -> let s = StringMap.find st.sname structs in
    (* If no constructor is defined add default *)
    {ss = s.ss;sname = s.sname; decls = s.decls; ctor = if (s.ctor.fname='')
      then defaultCtr s.sname else s.ctor;
    methods =
      if s.ss = ShapeType
    then
      let draw = try List.find (fun f2 -
        > f2.fname = "draw") s.methods
    with Not_found -
      > raise (Failure("draw method not defined in shape "^s.sname))
    in
      if (draw.rettype!=Void|draw.params!=[]) then raise(Failure("draw method must have return type void, id, and no parameters"))
        else s.methods
    else s.methods)
  ) structlist;
f = List.rev funcs ; v = globals}
let check prog =
(* Get the global variables and functions *)
let globals = prog.v
and functions = prog.f
and structs = prog.s
in
report_dup(fun n-> "Duplicate function name " ^n)(List.map (fun fd -
  > fd.fname)functions); (*Does pretty basic superficial checking if there exists dupli-
cate function names, global or structs*)
report_dup(fun n-> "Duplicate global variable " ^n)(List.map (fun (_,a,_) -
  > a)globals);
report_dup(fun n-> "Duplicate struct/shape name " ^n)(List.map(fun st -
  > st.sname)structs);

(* a list of the predefined functions special note about add shape it demands a nam-
e to construc the type *)
let builtinlist = [
  {rettyp=Void; fname="printi";params=[(Int, "x",Value)];locals=[];body=[];typ=Func;
  owner="None"};
  {rettyp=Void; fname="printf";params=[(Float, "x",Value)];locals=[];body=[];typ=
  Func;owner="None"};
  {rettyp=Void; fname="print";params=[(String, "x",Value)];locals=[];body=[];typ
  =Func;owner="None"};
  {rettyp=Void; fname="addshape";params=[(UserType(".shape",ShapeType),"x",Value)]
  ;locals=[];body[];typ=Func;owner="None"};
  {rettyp=Void;fname="setcolor.";params=[(Float,"x",Value);(Float,"x",Value);(Flo-
  at,"x",Value)];locals=[];body[];typ=Func;owner="None"};
  {rettyp=Void;fname="drawpoint";params=[(Vec,"x",Value)];locals=[];body[];typ=F
  unc;owner="None"};
  {rettyp=Void;fname="printc";params=[(Char,"x",Value)];locals=[];body[];typ=F
  unc;owner="None"};

  (* math funcs *)
  {rettyp=Float;fname="sin";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="cos";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="tan";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="log";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="log2";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="log10";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="abs";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="exp";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="sqrt";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="asin";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="acos";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="atan";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="sinh";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="cosh";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
  {rettyp=Float;fname="tanh";params=[(Float,"x",Value)];locals=[];body[];typ=Func
  ;owner="None"};
{retty=Float;fname="asinh";params=[(Float,"x",Value)];locals=[];body=[];typ=Func;owner="None"};
{retty=Float;fname="acosh";params=[(Float,"x",Value)];locals=[];body=[];typ=Func;owner="None"};
{retty=Float;fname="atanh";params=[(Float,"x",Value)];locals=[];body=[];typ=Func;owner="None"};
{retty=Float;fname="pow";params=[(Float,"x",Value);(Float,"x",Value)];locals=[ ];body=[];typ=Func;owner="None"};

let built_in_fun = List.fold_left (fun f m -> StringMap.add f.fname f m) StringMap.empty builtinlist
(*let function_decls =
  List.map(fun fd -> fd.fname) functions*)
let function_decls =
  List.iter(fun fd -
    -> StringMap.mem fd.fname built_in_fun in if x=true then raise(Failure("Built-in function " ^fd.fname^ " cannot be redefined")) else ()functions;
let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
  built_in_fun functions
in
let function_decl s = try StringMap.find s function_decls (*Builds a string map of name of func to function recorded*)
with Not_found -
  -> raise (Failure("Unrecognized function " ^ s ^". Did you forget to define it?"))
in
let main_check = let mn = (try function_decl "main" with Failure _ -
  -> raise( Failure "Must define main")
in if(mn.retty!=Int)
  then raise(Failure("main function must have return type int"))
  else ()(*Makes sure that main is defined*)
in
main_check;
(* Map of struct name to struct *)
let struct_name_list = List.fold_left(fun m =>
  -> StringMap.add m.fname m) (*creates a struct names list*)
StringMap.empty(structs)

(* convert int to float ignore others *)
let lit_promote (e,_) src trg = match (src,trg) with
  (Int,Float) -> (FloatLit(float(get_int e)),Float)
  | (t1, t2) when t1 = t2 -> (e,t1) (* No op promotion *)
  | _ -
  -> raise(Failure("Can not convert literal of type "^(string_of_typ src)^" to "^(string_of_typ trg)")

(* returns type and improved initializer *)
let rec check_global_initer t n = function
Exprinit e -> let (_,t') as e' = const_expr e in
  let e'' = (try lit_promote e' t' t with Failure _ -
    ->raise(Failure("Invalid initializer for global var "^n")))
in (t, Exprinit(e''))
| Listinit il -> (match t with
  Array(t2,e) ->
    let l = List.map ( fun i -> snd(check_global_initer t2 n i)) il in
    if e <> (Noexpr,Void) then (t, Listinit l) else ( Array(t2, (IntLit (List .length l), Int)), Listinit l)
  | UserType(_,_) ->

90
(* type of members *)
let dtl = List.map (fun (t,_) ->
  t) (StringMap.find s struct_name_list).decls
if (List.length dtl = List.length il) then
  (t, ListInit(List.map2 (fun t i ->
    snd(check_global_initer t n i)) dtl il))
else raise(Failure("Invalid initializer for global var "^n))

| _ -> raise(Failure("Brace initializer cannot be used with "^n))
| NoInit -> (t, NoInit)
in

(* Simultaneously modify globals list and build global_vars map *)
let (global_vars, globals') =
  List.fold_left(fun (m,gl) (t,n,i) ->
    (* check if type exists *)
    match t with
    | UserType(s,ss) -> ignore(
      try if (StringMap.find s struct_name_list).ss != ss then raise Not_found
      with Not_found ->
        raise(Failure((string_of_stosh ss)^" "^s " is not defined ")
    |
    Array(_,Noexpr,Void) when i<>Noinit -> ()
    | Array(_,e) ->
      (try if snd(const_expr e)<Int then raise Not_found with Not_found|Failure _ -> raise(Failure("Array declaration requires constant integer: "^n)))
    |
    _ -> ()
  );
  Array(_,_,) when i=Noinit ->
  raise(Failure("Incomplete array without initializer: "^n))
  |
  Array(_,_,Noexpr,Void) when i<>Noinit -> ()
  |
  Array(_,e) ->
  (try if snd(const_expr e)<Int then raise Not_found with Not_found|Failure _ -> raise(Failure("Array declaration requires constant integer: "^n)))
  |
  _ -> ()
); let globals = List.rev globals'
in

let function_check func =

(* Given struct give map of method names to method *)
let get_member_funcs name = let st = try StringMap.find name struct_name_list (creates a struct member function list*)
  with Not_found -> raise(Failure("Undefined struct/shape "^name))
in List.fold_left(fun m s -> StringMap.add s.fname s m) StringMap.empty st.methods
in

(* Get constructor *)
let get_member_constr name = let st = StringMap.find name struct_name_list (*creates a struct member function list*)
in st.ctor
in

(* Get map of memb_variables to their type *)
let get_struct_member_var name = let st = try StringMap.find name struct_name_list
  with Not_found -> raise(Failure("Undefined struct/shape "^name))
in StringMap.empty st.decls
(* Gets type of var in struct *)
let member_var_type name var =
  let temp = get_struct_member_var name
  in StringMap.find temp
  in
(* Gets mem_func *)
let get_mem_func_name func =
  let temp = get_member_funcs name
  in try StringMap.find func temp
      with Not_found -> raise(Failure(func ^ " is not a method of " ^name))
  in
let formal_vars = List.fold_left(fun m(t,n,_) -
    >StringMap.add n t m StringMap.empty (func.params)
  in
(* Top level local_vars *)
let local_vars = List.fold_left(fun m(t,n,_) -
  (match t with
      UserType(s,ss) -> ignore(
        try if (StringMap.find s struct_name_list).ss != ss then
        n raise Not_found with Not_found ->
        raise(Failure((string_of_stosh ss)^ "^s^ is not defined "))
      ); StringMap.add n t m
      _ -> StringMap.add n t m)
  )
*)
let lookup_function f =
  (* First try to find a matching constructor *)
try get_member_constr f
  (* If that fails try to find a method. 
this is guaranteed to fail in a normal function *)
with Not_found -> (try get_mem_func_name func.owner f
  (* Finally look for normal function *)
  with Not_found | Failure _ -> StringMap.find f function_decls)
  in
let rec check_block(local_decls, stmt_list) scopes =
  (* Prepend the block local variables to the scopes list *)
  (* Prepend any initializers to the statement list *)
let (stmt_list, scopes) =
  report_dup(fun n -
    > "Duplicate local variable " ^n ^ " in " ^func.fname)(List.map (fun (_,a,_) -
    > a)func.params); (*Checks is there exists duplicate parameter names and function
  n local name*)
  report_dup(fun n -> "Duplicate local variable " ^n ^ " in " ^func.fname)
  ((List.map (fun (_,a,_) -> a)func.params)@(List.map (fun (_,a,_) -
    > a)func.locals));
  in
let lookup_function f =
  (* First try to find a matching constructor *)
try get_member_constr f
  (* If that fails try to find a method. 
this is guaranteed to fail in a normal function *)
with Not_found -> (try get_mem_func_name func.owner f
  (* Finally look for normal function *)
  with Not_found | Failure _ -> StringMap.find f function_decls)
  in
let rec check_block(local_decls, stmt_list) scopes =
  (* Prepend the block local variables to the scopes list *)
  (* Prepend any initializers to the statement list *)
let (stmt_list, scopes) =
  report_dup(fun n -
    > "Duplicate local variable " ^n ^ " in " ^func.fname)(List.map (fun (_,a,_) -
    > a)local_decls);
  (* returns updated type (only for incomplete arrays) *)
let update_local t = function
  Listinit il -> (match t with
      Array(t2,e) ->
        if e <> (Noexpr,Void) then t else Array(t2, (Intlit (List.len
  gth il), Int))
      UserType(_,_) -> t
      _ -> raise(Failure("Brace initializer cannot be used with "^(string_of_typ t)))))
let add_local m (t,n,i) =
  ignore(match t with
  | UserType(s,ss) -> ignore(
    try if (StringMap.find s struct_name_list).ss != ss the
    n raise Not_found with Not_found ->
    raise(Failure((string_of_stosh ss)^ "^s ^n is not defined "))
  | _ -> t
  )
| Array(_, (_, Noexpr, Void)) when i=Noinit ->
  raise(Failure("Incomplete array without initializer: "^n))
| Array(_, (_, Noexpr, Void)) when i<>Noinit -> ()
| (try if snd(const_expr e)>>Int then raise Not_found with Not_found|Failure _ ->
  raise(Failure("Array declaration requires constant integer: "^n))
| _ -> ()
); StringMap.add n (update_local t i) m
in
let locals = List.fold_left add_local StringMap.empty local_decls
in stmt_list,(locals, LocalScope)::scopes
in
(* Recursive ret_type *)
let rec _ret_type n scopes =
  let hd = List.hd scopes in
  try (match hd with
    | (globs, GlobalScope) -> StringMap.find n globs
    | (locls, LocalScope) -> (try StringMap.find n locls
      with Not_found -
      > _ret_type n (List.tl scopes )
    | (_, StructScope) -> (try member_var_type func.owner n with Not_found-
      > _ret_type n (List.tl scopes )
    )
    with Not_found ->
    | if func.owner = ""
    then raise(Failure("Undeclared variable "^n " in "^func.fname
    )
    else raise(Failure("Undeclared variable "^n " in "^func.fname
    ))
  )
in
(* Gets type for variable name s [old ret_type]*)
let ret_type n = _ret_type n scopes
in
(* Gets type of expression. [LATER ON HANDLE WITH SAST]*)
let rec expr_b = function
  (IntLit s,_) -> (IntLit s,Int)
  | (CharLit s, _) -> (CharLit s, Char)
  | (StringLit s,_) -> (StringLit s, String)
  | (FloatLit s,_) -> (FloatLit s, Float)
  | (VecLit (f1,f2),_) -> (VecLit(f1,f2), Vec)
  | (Id s,_) -> (Id s, ret_type s)
  | (Promote e, t) -> (Promote(expr_b e),t)
  | (Binop(e1,op,e2),_) ->
    let (e1',t1') = (expr_b e1) and (e2',t2')=(expr_b e2) in
    (match op with
     | Add|Sub|Mul|Div|Mod when t1'=Int & t2'=Int -
        > (Binop((e1',t1'),op,(e2',t2'))),Int)
     | Add|Sub|Mul|Div when t1'=Vec& t2'=Vec -
        > (Binop((e1',t1'),op,(e2',t2'))),Vec)
     | Mul|Div when t1'=Vec & t2'=Int -
        > expr_b(Binop((e1',t1'),op,(Promote((e2',t2'))),Float)),Vec)
354.     |Mult|Div when t1=Vec&&t2=Float -
355.     > (Binop((e1',t1'),op,(Promote((e2',t2'))),Vec)),Vec)
356.     |Mult when t1=Int&&t2=Vec -
357.     > expr_b(Binop((Promote((e1',t1'))),Float),op,(e2',t2'))),Vec)
358.     |Mult when t1=Float&&t2=Vec -
359.     > (Binop((Promote((e1',t1'))),Vec),op,(e2',t2'))),Vec)
360.     |Add|Sub|Mult|Div when t1=Float && t2=Float -
361.     > (Binop((e1',t1'),op,(e2',t2'))),Float)
362.     |Add|Sub|Mult when t1=Int && t2=Float -
363.     > (Binop((e1',t1'),op,(Promote((e2',t2'))),Float)),Float)
364.     |Add|Sub|Mult when t1=Float && t2=Int -
365.     > (Binop((e1',t1'),op,(Promote((e2',t2'))),Float)),Float)
366.     |Equal|Neq when t1=t2-> (Binop((e1',t1'),op,(e2',t2'))),Int)
367.     |Equal|Neq when t1=Int && t2=Float -
368.     > (Binop((e1',t1'),op,(Promote((e2',t2'))),Float)),Int)
369.     |Equal|Neq when t1=Vec && t2=Vec -
370.     > (Binop((e1',t1'),op,(e2',t2'))),Int)
371.     |Less|Leq|Greater|Geq when t1=t2 -
372.     > (Binop((e1',t1'),op,(e2',t2'))),Int)
373.     |Less|Leq|Greater|Geq when t1=Int && t2=Float -
374.     > (Binop((Promote((e1',t1'))),Float),op,(e2',t2'))),Int)
375.     |Less|Leq|Greater|Geq when t1=Float && t2=Int -
376.     > (Binop((e1',t1'),op,(Promote((e2',t2'))),Float)),Int)
377.     |Less|Leq|Greater|PreInc when t1=Int &
378.     > (Binop((e1',t1'),op,(e2',t2'))),Int)
379.  
380.     > raise(Failure("Unsupported operands " ^ Ast.string_of_expr e1 ^ " and " ^Ast.string
381.     of_expr e2" for " ^string_of_op op))
382.     )
383.     (Unop(op,e1),_)) -> let (e1',t1') as f = (expr_b e1) in
384.     (match op with
385.     Neg when t1=Int -> (Unop(op,(e1',t1'))),Int)
386.     Not when t1=Int -> (Unop(op,(e1',t1'))),Int)
387.     Neg when t1=Float -> (Unop(op,(e1',t1'))),Float)
388.     Neg when t1=Vec -> (Unop(op,(e1',t1'))),Vec)
389.     Pos when t1=Int -> (Unop(op,(e1',t1'))),Int)
390.     Pos when t1=Float -> (Unop(op,(e1',t1'))),Float)
391.     Pos when t1=Vec -> (Unop(op,(e1',t1'))),Vec)
392.     PreInc when t1=Int ->(match f with
393.     (Id(_,Int) -> (Unop(op,(f)),Int)
394.     |(Index(_,Int),Int) -
395.     > (Unop(op,(f)),Int)
396.     (Member(_,Int) -
397.     > raise(Failure("PreInc or PreDec cannot be applied to " ^ Ast.string_of_expr f))
398.     )
399.     Predec when t1=Int -> (match f with
400.     (Id(_,Int) -> (Unop(op,(f)),Int)
401.     |(Index(_,Int),Int) -
402.     > (Unop(op,(f)),Int)
403.     (Member(_,Int) -
404.     > raise(Failure("PrecInc or PrecDec cannot be applied to " ^ Ast.string_of_expr f))
405.     )
406.     > raise(Failure("Unsupported unary operation for " ^ Ast.string_of_expr e1 ))
407.     )
408.     (Noexpr,_) -> (Noexpr,Void)
409.     (Asnp(e1,asn,p,e2),_) -
410.     let (e1',t1') = (lexpr_b e1) and (e2',t2') as (expr_b e2) in
411.     (match asnp with
Asn when t1'=t2' -> (Asnop((e1',t1'),asnp,(e2',t2'))),t1')

<table>
<thead>
<tr>
<th>ASN when t1'=Float &amp; &amp; t2'=Int</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Asnop((e1',t1'),asnp,(Promote(e2',t2'),Float),Float),Float)</td>
</tr>
<tr>
<td>(CmpAsn b when t1'=snd(expr_b (Binop((e1',t1'), b, (e2',t2'))),</td>
</tr>
<tr>
<td>Void)) -&gt; (Asnop ((e1',t1'),asnp,(Promote f,t1'))),t1')</td>
</tr>
</tbody>
</table>

(*|CmpAsn b when e1'=Float & & e2'=Int -> Float |
|  (CmpAsn b when e1'=Int & & e2'=Float -> Float*) |

<table>
<thead>
<tr>
<th>_ -</th>
</tr>
</thead>
<tbody>
<tr>
<td>raise (Failure (&quot;Invalid assignment of &quot; ^Ast.string_of_typ t2&quot; to &quot;+Ast.string_of_typ t1&quot;)</td>
</tr>
</tbody>
</table>

| (Call(e1, actuals),_) -> let (e',fd) = (match e1 with |
|  (Id s),_) as f -> (f, (try lookup_function s |
|    with Not_found -> function_decl s)) |
|  (Member (e,s),_)-> let (_,t') as f = expr_b e in |
|    let sname= (match t' with |
|    UserType(s,_) -> s |
|    | _ - |
|    raise(Failure("Member operator (dot) can only be applied to struct or shape")) |
|   ) |
| in ((Member(f,s), Void), get_mem_func_name sna |
| me s) |

<table>
<thead>
<tr>
<th>_ -</th>
</tr>
</thead>
<tbody>
<tr>
<td>raise(Failure(&quot;Invalid function call: &quot; ^ string_of_expr e1))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>_ -</th>
</tr>
</thead>
<tbody>
<tr>
<td>let actuals' =</td>
</tr>
<tr>
<td>(</td>
</tr>
<tr>
<td>if (List.length actuals != List.length fd.params)</td>
</tr>
<tr>
<td>&quot;addshape&quot;)</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>if fd.fname=&quot;addshape&quot;&amp;&amp;((List.length actuals&gt;=0)</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>List.map(fun e -&gt; let (_,t1') as f=(expr_b e) in</td>
</tr>
<tr>
<td>(match t1' with</td>
</tr>
<tr>
<td>UserType(_,ShapeType) -&gt; f</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>raise(Failure(&quot;Arguments of addshape function must be of type shape&quot;))</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>in (</td>
</tr>
<tr>
<td>actuals</td>
</tr>
<tr>
<td>else raise (Failure (&quot;Incorrect number of arguments in &quot;+fd</td>
</tr>
<tr>
<td>.fname))</td>
</tr>
<tr>
<td>else</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>_ -</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*let actuals = List.map2 (fun (t, s, p) e -</td>
</tr>
<tr>
<td>if p = Ref then (lexpr_b e) else (expr_b e</td>
</tr>
<tr>
<td>) fd.params actuals in*)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>_ -</th>
</tr>
</thead>
<tbody>
<tr>
<td>List.map2 (fun (t,_,p) e -</td>
</tr>
<tr>
<td>let (_,et) as f = (if p = Ref then lexpr_b e else expr_b e) in</td>
</tr>
<tr>
<td>ignore (check_ass t et (p=Value)</td>
</tr>
<tr>
<td>(Failure (&quot;Illegal argument &quot;^(string_of_expr e)&quot; of t</td>
</tr>
<tr>
<td>ype &quot;+Ast.string_of_typ et&quot; in call to function &quot;+fd.fname&quot;) which expects arg</td>
</tr>
<tr>
<td>ument of type &quot;+Ast.string_of_typ t)</td>
</tr>
<tr>
<td>));</td>
</tr>
</tbody>
</table>

| (if p=Ref then f else expr_b (Promote (f),t) |
| ) |
| ) |
| fd.params actuals |

<table>
<thead>
<tr>
<th>_ -</th>
</tr>
</thead>
<tbody>
<tr>
<td>(<em>actuals</em>)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>_ -</th>
</tr>
</thead>
<tbody>
<tr>
<td>in (</td>
</tr>
<tr>
<td>Call(e',actuals'), if fd.typ &lt;&gt; Constructor then fd.rettyp</td>
</tr>
<tr>
<td>else UserType(fd.owner,(StringMap.find fd.owner str</td>
</tr>
<tr>
<td>uct_name_list).ss) )</td>
</tr>
</tbody>
</table>

95
448. |(Vecexpr (e1,e2),_) ->
449.   let (_,t1') as f1 = (expr_b e1) and (_,t2') as f2 = (expr_b e2)
450. in
451.   let f1 = if (t1' = Float) then f1 else if t1' = Int then (Promote f1, Float) else raise(Failure("Elements of vector must be of type double"))
452.   and f2 = if (t2' = Float) then f2 else if t2' = Int then (Promote f2, Float) else raise(Failure("Elements of vector must be of type double"))
453. in (Vecexpr(f1,f2),Vec)
454. |(Posop (_,e2),_) -> let e2' = (expr_b e2)
455. in (match e2' with
456.   (Id(_),Int) -> (Posop(_,e2'),Int)
457.   (Index(_,_),Int) -> (Posop(_,e2'),Int)
458.   (Member(_,_),Int) -> (Posop(_,e2'),Int)
459.   _ ->
460.     raise(Failure("PostInc or PostDec cannot be applied to " ^ Ast.string_of_expr e2))
461. )
462. |(Index(e1,e2),_) ->
463.   let (e1',t1') = (expr_b e1) and (e2',t2') = (expr_b e2) (* ALLOW VECTOR INDEXING *)
464. in let te1' = (match t1' with
465.   Array(_,_) -> t
466.   Vec -> Float
467.   _ _
468.     raise(Failure("Indexing only supported for arrays and vectors"))
469. in
470.   if t2' /= Int
471.      then raise(Failure("Must index with an integer"))
472. else (Index((e1',t1'),(e2',t2'))),te1'
473. |(Member(e1,s),_) --> let (e1',t1') = (expr_b e1)
474. in let te1'= (match t1' with
475.   UserType(s1,_) -> s1
476.   _ _
477.     raise(Failure("Member operator (dot) can only be applied to struct or shape"))
478. in
479.   ( try (Member((e1',t1'),s),member_var_type te1' s) with Not_found -> raise(Failure(s" is not a member of " ^ (string_of_typ t1')))))
480. (* Special handling for lvalue expressions *)
481. and lexpr_b = function
482.   (Id s,_) -> (Id s, ret_type s)
483. in
484.   (Index(e1,e2),_) ->
485.     let (e1',t1') = (expr_b e1) and (e2',t2') = (expr_b e2) (* ALLOW VECTOR INDEXING *)
486. in let te1' = (match t1' with
487.   Array(_,_) -> t
488.   Vec -> Float
489.   _ _
490.     raise(Failure("Indexing only supported for arrays and vectors"))
491. in
492.   if t2' /= Int
493.      then raise(Failure("Must index with an integer"))
494. else (Index((e1',t1'),(e2',t2'))),te1'
495. |(Member(e1,s),_) --> let (e1',t1') = (expr_b e1)
496. in let te1'= (match t1' with
497.   UserType(s1,_) -> s1
498.   _ _
499.     raise(Failure("Member operator (dot) can only be applied to struct or shape"))
500. )
let check_bool_expr e = let (_,t)` as f = expr_b e in
if t` != Int (*Could take in any number need to force check 1 or 0*)
then raise(Failure((string_of_expr` e)^" is not a boolean value")
else f in

let rec stmt = function
  | Expr f -> Expr(expr_b f)
  | Return e -
    > let (_,t1`) as f= (expr_b e) in if t1`= func.rettyp then Return(f) else
    raise(Failure("Incorrect return type in " ^ func.fname))
  | If(p,e1,e2) -> If(check_bool_expr p, stmt e1, stmt e2)
  | For(e1,e2,e3,stmt) -
    > For(expr_b e1, check_bool_expr e2,expr_b e3, stmt state)
  | While(p,s) -> While(check_bool_expr p, stmt s)
  | ForDec vdecls,e2,e3,body -
    > stmt ( Block(vdecls, [For((Noexpr,Void) , e2, e3, body)],PointContext) )
  | Timeloop(s1,e1,e2,e3,st1) ->
    (* handle timeloop variables *)
    let htlv = (match st1 with
    | Block (vl,s1,c) -
    > stmt (Block ([(lvalue Float,Exprinit(e1))::(lvalue Float,Exprinit(e2))::vl,s1,c) )
    | s -
    > stmt (Block( [(lvalue Float,Exprinit(e1)) ; (lvalue Float,Exprinit(e2))], [s], PointContext))
    ) in
(* Need to check statements also ? *)
if s1 = s2 then raise(Failure("Duplicate variable "^s1^" in timeloo p definition"))
else
  let (_,t1`) as f1 = expr_b e1
  and (_,t2`) as f2 = expr_b e2
  in
  if (t1` = Float || t1`=Int) && (t2` = Float || t2`=Int)
  then Timeloop(s1, (Promote f1,Float), s2, (Promote f2,Float),ht lv)
  else raise(Failure("Timeloop definition only accepts expression s of type double"))
| Frameloop (s1,e1,e2,st1) -
  (* handle frameloop variables *)
  let htlv = (match st1 with
  | Block (vl,s1,c) -
  > stmt (Block ([(lvalue Float,Exprinit(e1))::(lvalue Float,Exprinit(e2))::vl,s1,c) )
  | s -
  > stmt (Block( [(lvalue Float,Exprinit(e1)) ; (lvalue Float,Exprinit(e2))], [s], PointCon text))
  ) in
(* Need to check statements also ? *)
if s1 = s2 then raise(Failure("Duplicate variable "^s1^" in timeloo p definition"))
else
  let (_,t1`) as f1 = expr_b e1
  and (_,t2`) as f2 = expr_b e2
  in
  if (t1` = Float || t1`=Int) && (t2` = Float || t2`=Int)
  then Frameloop(s1, (Promote f1,Float), s2, (Promote f2,Float),ht lv)
else raise(Failure("Timeloop definition only accepts expressions of type double"))

in

let check_ret () = match stmt_list with
  | Return _ -> ()
  | Return _ :: _ -
    > raise(Failure("Cannot have any code after a return statement"))
  | _ -> ()

in

let var_promote e src trg = match (src,trg) with
  | (Int, Float) -> (Promote e,Float)
  | (t1, t2) when t1 = t2 -> e (* No Op promotion *)
  | _ -
    > raise(Failure("Can not convert literal of type "^\(string_of_typ src\)^" to "^\(string_of_typ trg\)"))

in

(* returns type and improved initializer *)

let thrd (_,_,x) = x

in

let rec check_local_initer (t,n,i) =
  (match i with
   Exprinit e -> let (_,t') as e' = expr_b e in
   let e'' = (try var_promote e' t' t with Failure _ -
     >raise(Failure("Invalid initializer for local var "^n")))
   in (t,n,Exprinit(e''))
   | Listinit il -> (match t with
     Array(t2,e) ->
     let il = List.map ( fun i -
       >thrd(check_local_initer (t2, n, i)) ) il in
     if e <> (Noexpr,Void) then (t, n, Listinit il) else ( Array(t2,
       (IntLit (List.length il), Int)), n, Listinit il)
     | UserType(s,_) ->
       (* type of members *)
     let dtl = List.map ( fun (t,_)-
       >t) (StringMap.find s struct_name_list).decls in
     if (List.length dtl = List.length il) then
     (t, n, Listinit(List.map2 (fun t 1 -
       >thrd(check_local_initer (t,n,i))) dtl il))
     else raise(Failure("Invalid initializer for local var "^n")
    | _ -
      >raise(Failure("Brace initializer cannot be used with "^\(string_of_typ t\)"))
     )
   )
  )

in

check_ret(); (List.map check_local_initer local_decls, List.map stmt stmt_l
ist) (* End of check_block *)

in

(* Construct the scopes list before calling check_block *)

let scopes_list = match func.typ with
  | Function f -> [(local_vars, LocalScope); (global_vars, GlobalScope) ]
  | _ -
    > [[(local_vars, LocalScope); (StringMap.empty, StructScope) ; (global_vars, GlobalScope)]

in

let (locals',body') = check_block (func.locals, func.body) scopes_list in

{ rettyp = func.rettyp; fname = func.fname; params = func.params; locals = loca
ls'; body = body'; typ = func.typ; owner= func.owner}

in

let f' = List.map function_check functions in
let s' = List.map (fun st ->
  {
    ss = st.ss ; sname = st.sname ; decls = st.decls;
    ctor = function_check st.ctor;
    methods = List.map function_check st.methods
  }
)

let v' = globals in
{ s = s' ; f=f' ; v =v' ;}
8.7 Test Script

Filename: testall.sh

```
#!/bin/sh
# 'testall.sh'
# Regression Testing script for ART
# 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

# llvm static compiler
LLC="llc"

# c compiler for linking in glut
CC="gcc"

# does the compilation
COMPILE="compile"
COMP="bash $COMPILE"

# Path to ART compiler
if [ -e "./art.native" ];
  then
  ART="./art.native"
  elif [ -e "./art" ];
  then
    ART="./art"
    else
      echo "No art compiler found. Attempting build..."
      echo ""
      make clean
      if make
        then
          ART="./art.native"
          else
            echo -e "\nBuild Failed!!!" && exit 2
            fi
      fi

# Time limit for all operations
ulimit -t 30

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

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

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

# Compare <outfile> <reffile> <difffile>
# Compare < outfile > with < reffile >
# Differences, if any, are written to <difffile>
Compare() {
  generatedfiles="$generatedfiles $3"
  echo diff -bu "$1 "$2 ""> "$3" 1>&2
  diff -bu "$1" "$2" ""> "$3" 2>&1 || {
    SignalError "$1 differs"
    echo "FAILED $1 differs from $2" 1>&2
  }
}

# Run <outfile> <reffile> <diffile>
# Compare <outfile> with <reffile>
# Differences, if any, are written to <diffile>
Compare()
{
  generatedfiles="$generatedfiles $3"
  echo diff -bu "$1 "$2 ""> "$3" 1>&2
  diff -bu "$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
# Used for tests that are supposed to run without any errors
Run() {
  echo "$*" 1>&2
  eval "$*" || {
    SignalError "$1 failed on "$*
    return 1
  }
  
}

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

# For tests that are supposed to run without any errors
Check() {
  error=0
  basename=`echo $1 | sed 's/.*\\///' $/.art//`
  reffile=`echo $1 | sed 's/.*\/$//'`
  basedir="`echo $1 | sed 's/[^/\/^/][^/\/^/]/'`/.
  resultname="results/"${basename}
  
  if [ -n "$basename..." ];
  echo $1>&2
  echo "######## Testing $basename" 1>&2
  
  generatedfiles=""
  
  generatedfiles="$generatedfiles ${resultname}.ll ${resultname}.out ${resultname}.s ${resultname}" &&
  
  Run "$COMP" "$1" "$resultname" "results/" &&
  Run "$resultname" ""> "$resultname.out" &&
  Compare "$${resultname}.out" "$reffile.out" "$${resultname}.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
    #rm -f $generatedfiles
    echo "##### FAILED" 1>&2
globalerror=$error
fi
}
#
For tests that are supposed to give an error
CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\///'
    s/.art//'`
    reffile=`echo $1 | sed 's/^[^/\[\]/]*$//'`
    resultname="results/"${basename}
    generatedfiles=
    generatedfiles="$generatedfiles ${resultname}.err ${resultname}.diff" & &
    RunFail "$ART" "<" $1 "2>" "$resultname.err ">" $globallog & &
    Compare "$resultname.err" $reffile.err "$resultname.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
}
#
Options
while getopts kdpsh c; do
    case $c in
    k) # Keep intermediate files
        keep=1
        ;;
h) # Help
        Usage
        ;;
esac
    done
    shift `expr $OPTIND - 1`
#
Error finding CC
CCFail() {
    echo "Could not find c compiler "$CC"."

echo "Check your installation and/or modify the CC variable in testall.sh"
exit 1

# Error finding LLC
LLCFail() {
  echo "Could not find the LLVM static compiler \"$LLC\"."
  echo "Check your LLVM installation and/or modify the LLC variable in testall.sh"
  exit 1
}

# Error finding COMPIL
COMPILEFail() {
  echo "Could not find compile \"$COMPILE\"."
  exit 1
}

which "$CC" >> $globallog || CCFail

which "$LLC" >> $globallog || LLCFail

ls "$COMPILE" >> $globallog || COMPILEFail

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

# Make the results directory
if [ ! -e results ]
then
  mkdir results
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
8.8 Makefile

Filename: Makefile

1. # Make sure ocamlbuild can find opam-managed packages: first run
2. #
3. # eval `opam config env`
4. # Easiest way to build: using ocamlbuild, which in turn uses ocamlfind
5.  .PHONY : art.native
6.  art.native:
7.    ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \
8.    art.native
9.  # More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM
10. OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx art.cmx
11. art : $(OBJS)
12.    ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis $(OBJS) -o art
13. scanner.ml : scanner.mll
14. ocamlllex scanner.mll
15. parser.ml parser.mli : parser.mly
16. ocamlyacc parser.mly
17. %.cmo : %.ml
18.    ocamlc -c $<
19. %.cmi : %.mli
20.    ocamlc -c $<
21. %.cmx : %.ml
22.    ocamlfind ocamlcc -c -package llvm $<
23. ### Generated by "ocamlddep *.ml *.mli" after building scanner.ml and parser.ml
24. art.cmo : semant.cmo scanner.cmo parser.cmo codegen.cmo codegen.cmo ast.cmo
25. art.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx codegen.cmx ast.cmx
26. ast.cmo :
27. ast.cmx :
28. codegen.cmo : ast.cmo
29. codegen.cmx : ast.cmx
30. parser.cmo : ast.cmo parser.cmi
31. parser.cmx : ast.cmx parser.cmi
32. scanner.cmo : parser.cmi
33. scanner.cmx : parser.cmx
34. semant.cmo : ast.cmo
35. semant.cmx : ast.cmx
36. parser.cmi : ast.cmo
37. 
38. .PHONY : clean
39. clean :
40.    ocamlbuild -clean
41.    rm -rf testall.log *.diff art scanner.ml parser.ml parser.mli
42.    rm -rf *.cmx *.cml *.cmo *.cmx *.o
43. clean-tests:
```
61.   rm -rf results
62.   rm -f *.diff *.ll *.out *.err
63.
64.   # Building the tarball
65.
66.   TESTS = *
67.
68.   FAILS = *
69.
70.   DEMOS = *
71.
72.   TESTFILES = $(TESTS:%=test-%.out) $(TESTS:%=test-%.art)
73.   $(FAILS:%=fail-%.art) $(FAILS:%=fail-%.err)
74.
75.   DEMOFILES = $(DEMOS)
76.
77.   TARFILES = ast.ml codegen.ml compile Makefile art.ml parser.mly README.md \
78.      scanner.ml semant.ml testall.sh $(TESTFILES:%=tests/%) $(DEMOFILES:%=demos/%)
79.
80.   ART.tar.gz : $(TARFILES)
81.      cd .. && tar czf ART/ART.tar.gz \
82.         $(TARFILES:%=ART/%)
```
8.9 Compile

Filename: compile

```bash
#!/bin/bash

# Compiles an art program to an executable

# Path to ART compiler
if [-e "./art.native"] then
    ART=./art.native
elif [-e "./art"] then
    ART=./art
else
    echo "No art compiler found. Attempting build..."
    echo ""
    make clean
    if make
        ART=./art.native
    else
        echo -e "\nBuild Failed!!!" && exit 2
    fi
fi

base=`echo "$1" | sed 's/.*\///g' s/.art//'`

if [-z "$2"] then
    out=./$base
else
    out="$2"
fi

if [-z "$3"] then
    tmp="/tmp"
else
    tmp="$3"
fi

${ART} < $1 > ${tmp}/${base}.ll
llc ${tmp}/${base}.ll
gcc ${tmp}/${base}.s -g -lm -lglut -lGLU -lGL -o $out
```

8.10 Art.ml

Filename: art.ml

```ml
1. type action = Ast | LLVM_IR | Compile
2.
3. let _ =
4. let action = if Array.length Sys.argv > 1 then
5.   List.assoc Sys.argv.(1) [ ("-a", Ast); (* Print the AST *)
6.     ("-l", LLVM_IR); (* Generate LLVM, don't check *)
7.     ("-c", Compile) ] (* Generate, check LLVM IR *)
8.   else Compile in
9.   let lexbuf = Lexing.from_channel stdin in
10. let ast =
11. try
12.   Parser.program Scanner.token lexbuf
13. with Parsing.Parse_error ->
14.   let curr = lexbuf.Lexing.lex_curr_p in
15.   let line = curr.Lexing.pos_lnum in
16.   let cnum = curr.Lexing.pos_cnum - curr.Lexing.pos_bol in
17.   prerr_endline("Syntax Error: Line " ^ string_of_int line ^ " Column " ^ string_of_int cnum);
18.   exit 1
19. in
20. let ast = Semant.check (Semant.struct_build ast) in
21. match action with
22. Ast -> print_string (Ast.string_of_program ast)
23. | LLVM_IR -> print_string (Llvm.string_of_llmodule (CodeGen.translate ast))
24. | Compile -> let m = Codegen.translate ast in
25. Llvm_analysis.assert_valid_module m;
26. print_string (Llvm.string_of_llmodule m)
```
8.11 Readme

Filename: README.md