SEAMscript

Sean Inouye (si2281)
Edmund Qiu (ejq2106)
Akira Baruah (akb2158)
Maclyn Brandwein (mgb2163

December 23, 2015

Contents

1 Introduction

1.1 Motivation

Many people who try to program computer games for the first time run into the issue of not only having to grapple with the intricacies of game development, but also the problem of juggling libraries and runtime environments. For those looking for a simple solution, perhaps for educational purposes, prototyping a concept, or hobbyist work, we offer SEAMScript, a simple programming language. We distill the ideas of object oriented programming into a simple example, in which objects represent distinct entities, a direct model which is useful for simple games.

1.2 Overview

Therefore, the high level picture of SEAMscript is a synchronous entity simulation model. Entities can be spawned, and they can be killed off. With their own fields and functions, they also possess step functions that are called at equal intervals of a predefined time step. Using this programming paradigm, entities will each be responsible for their own movement and intercommunication.

2 Quick-Start Tutorial

SEAMscript is a source-to-source language. The `seamc` compiler will convert your original SEAMscript code to SDL-compatible C source code.

2.1 Prerequisites

The following software dependencies were used for development and testing purposes. SEAMscript may be compatible with other operating systems and frameworks, but we can only recommend the following system prerequisites.

- **Ubuntu 14.04 64-bit**
  Most of the latest Debian-based GNU/Linux distributions should work.
- **Simple DirectMedia Layer (SDL)**
  On Debian-based systems, `apt-get install libsdl-dev` *as root should do the trick.*
- **SEAMscript Project Repository**
  The Git repository (repo) for this project is hosted at `https://github.com/teamSEAM/ProjectSEAM`

2.2 Getting Started

In order to build the SEAMscript compiler, `seamc`, from source, navigate to the `src/` directory found inside the root of the project repo. Once you're there, simply typing `make` will build the entire compiler. After running `make`, you should now have an executable script called `seamc` in your `src/` directory.

2.3 Basic Structure of a SEAMscript Program

```
entity World:
    string name
    int population
    func start():
        name = "My world!"
        population = 0
```

A SEAMscript program is simply a collection of `entity` definitions. Each `entity` contains variable declarations and function definitions. Function definitions contain more variable declarations (with function scoping) and a collection of statements. SEAMscript uses tab indentation to notate scoping.

2.4 Entities

An `entity` is a primitive class type from which the universe of SEAMscript is created. To declare one of these entities, simply type the keyword `entity` followed by the name of the entity class you wish to define.

2.5 Variables

To declare a variable, specify the primitive type (`int`, `float`, `string`, etc.) followed by the name of the variable. For example, `string name` declares a variable called `name` that is of type `string`.

2.6 Functions

Function are defined in a C-like syntax, with the return type followed by the function identifier and a comma-delimited list of formal arguments enclosed in parentheses. This function signature must by following by a colon, as follows: `int myfunc(string s):`.

2.7 Control Flow

There are many structures that can be used for control flow include condition jumps and looping.

2.7.1 if/else

A simple if-else statement can be written as follows:

```
if (condition):
    statement
else:
    statement
```

2.7.2 Loops

SEAMscript supports both for and while loops, which again follow a C-like syntax:

```
int i
for (i = 0; i < 4; i = i + 1):
    statements

while (true):
    statements
```

2.8 Comments

Any text enclosed by a single starting # symbol and another terminating # symbol are considered comments and are completely ignored by the compiler.

3 Language Reference Manual

3.1 Introduction

SEAMScript is a simple high-level language that focuses on entity-based applications. Applications, primarily simulations and games, benefit from a built in system for handling running events periodically, and from built in functionality to simplify the typical I/O expected from these sorts of apps. Simple games, such as Breakout! or Snake, can be prototyped much more rapidly than in other languages. Other simulations, like cars interacting at an intersection, can also be written fairly quickly. Compared to real-life, the accuracy of a SEAMScript program is low due to concerns left to developers such as buffering and interpolating events between time deltas, but the native support for 'steps' saves developers from the hassle of manually starting/stopping entities.

Throughout this document, "...[a comment]..." will be used to indicate places where code of the type described in the comment is omitted for brevity but assumed present by the compiler.

3.2 Fundamental Types

SEAMScript is statically typed and supports the following primitive types:

- `int` - Signed integers with architecture-specific size.
- `string` - ASCII-based strings of arbitrary length and enclosed by a pair of double quotations.
- `float` - 64-bit IEEE floating point numbers.
- `texture` - Stored image primitives.
- `instance` - Entity types. Entity types are described in more depth below.

3.3 Comments

Only block comments are supported. They are started with the token `#` and ended with another `#`, and may not be nested. Anything in between the comments will not be read by the parser. Comments may not be nested. For example, `# This is a comment #` is a comment. `# Malformed # comment ##` is a malformed comment (once the parser hits "comment", a syntax error is indicated).

3.4 Literals

Literals represent fixed values of ints, strings, and floats. These values are used in assignment or often calculation operations. The format and semantics of literals for each type are as follows:

- `int` - Integers are declared with either a sequence of one or more digits from 0-9, potentially prefixed with a − to indicate negative numbers. You may have integers of any length, although numeric overflow may result if you exceed the representable length of an int on your hardware.

  Examples:

```
int a_number = -35 # Valid #
int num = 24. # Invalid #
int a_positive = +5 # Invalid; \+" is assumed #
```

- `float` - Floating point values are declared with an optional prefix of – to indicate negative numbers, a sequence of zero or more digits from 0-9, a mandatory `.`, and one or more digits from 0-9. Like integers, you may write out numbers unrepresentable on your hardware, but numeric overflow will occur. Floating point values will lose a minute amount of precision once run on hardware.

  Examples:

  ```
  float a_float = -35.1 # Valid #
  float another_float = -.334 # Valid #
  float not_valid = 34. # Invalid; need a decimal portion. #
  float also_wrong = . # Invalid #
  ```

- `string` - String literals are defined by ASCII characters within quotes. To include a quotation mark within a string literal, you must first escape it with a \. String literals may be empty, and there is no limit to their length.

  Example:

  ```
  string string_beans = \String beans" # Valid #
  string a_quote = \Quoth the Raven, \"Hello\"" # Valid #
  string bad_quote = \He dictated \here is a dictate"" # Invalid #
  ```

- `entity` and `texture` - These types do not have associated literals.

3.5 Variables

3.5.1 Names

Variable names are a combination of lowercase letters, uppercase letters, and underscores. They must begin and end with a letter (either uppercase or lowercase). For example, `Hello`, `hi_there`, and `variable_` would be supported, but `_variable`, and `hello2` would not be.

3.5.2 Declaration

Variables are declared in the format:

`<type> <identifier>`

You may optionally assign the newly declared variable a value upon creation, but you must heed the standard variable assignment rules (see below). Re-declaring variables with any reused name from any scope is unsupported, except within the scope of the entity. Two entities may have member variables with same identifier (and often will, in fact), and they may reuse identifiers in the global scope. You may declare variables in the global scope, within functions, in the body of an entity, and in entity member functions.

3.5.3 Access

Variables are considered "accessed" when their identifier is used outside of their initial declaration or assignment. For example, "string catdog = convert.string_join(cat, dog)" would access the values stored at "cat" and "dog", but not "catdog" because it is being declared. Local variables – variables found in function arguments or at the top of a function – may be accessed within the function, but not elsewhere. Likewise, functions in an entity are able to access variables the member variables of an entity, although if a variable declared in the function or its arguments has the same name as a variable in an entity said variable will be accessed instead of the entity's member variable.

3.5.4 Assignment

Variables are assigned to literals, other variables, or the results of built in operators with the = token. Variables may only being assigned to values of their own type.

3.6 Operators

The supported operators are shown in the below table. Note that promotion is not supported – you may not divide a float by and int, or add a float and an int, and so on and so forth. See built-in functions for functions that deal provide conversions to get around these sorts of issues.

Operator	Meaning	Suppo
+	Add the LHS value and the RHS value and return the result	any pai
−	Subtract the RHS value from the LHS value and return the result	any pai
*	multiply the RHS and the LHS and return the result	any pai
/	divide the RHS and the LHS and return the result	any pai
==	Compare the LHS with the RHS for equality (true if equal, otherwise false)	any pai
!=	Compare the LHS with the RHS for inequality (true if not equal, otherwise false)	any pai

3.7 Statements and Blocks

Statements are terminated by a newline character. Blocks of code (e.g. what follows control flow or function declaration) are marked by increasing the level of indentation by one tab. Tabs alone are supported – tabbing done with other forms of whitespace will not be recognized and will generate syntax errors.

3.8 Control Flow

Control flow is supported with if/else statements, while loops, and for loops.

3.8.1 If/Else Statement

If statements start with an `if`, are followed with a left paren, an expression, a right paren, a colon, an indented block, optionally all followed by an `else`, a colon, and another indented block . The indented blocks must contain code other than comments. For example:

```
if(score > 100):
    score = score + 50
else:
    score = score + 100
```

would be accepted as valid. However,

```
if(score > 150):
    else:
        score = 100
```

or

```
if(score > 200):
        score = 250
    else:
    score = score + 10
```

would be considered invalid.

3.8.2 While Loop

While loops start with a `while`, are followed with a left paren, an expression that evaluates to true or false, a right paren, a colon, and an indented block. The indented block must contain code other than comments. For example:

```
int i = 5
while(i < 10):
    i = i + 1
```

is considered valid. However,

```
int i = 5
while(true):
i = i + 5
```

is considered invalid.

3.9 Entities

Entities are collections of variables and methods, with special methods that are invoked by SEAMScript at various times if they exist. Conceptually, entities are very close to objects in other object-oriented languages, although entities lack certain features of objects and possess a bit of convenience functionality. Entities may contain methods, they may contain any number of variables (including other entities).

Entities are started with the keyword 'spawn' and the type of entity that is to be created, and destroyed with "kill" and the identifier (note that this must be an identifier; runtime expressions will not work here). For example:

```
entity World:
<Car> c

func start():
c = spawn Car # Calls c.start() and adds to the step/render pipeline

func stop():
kill c # Calls c.stop() and takes out of step/render pipeline
```

As soon as an entity is spawned, it is considered 'staged' to have its step and render functions called in the pipeline. Entities are stopped with the keyword `kill`. After an entity is 'killed', it may no longer be used.

Entities are declared with `entity`, an identifier that must start with a capital letter, a colon, and followed by an indented block containing (in order):

- Variable declarations for any variables accessible throughout the entity and to other portions of code with a reference to the entity. Note assignment with declaration is not allowed here.
- Any user-defined functions. The format for these is the same as other function declarations. Other code can directly call these functions.
- Functions the language uses. These functions, all of which are optional, but if used must have at least one statement of executable code, are:
  - `start` - This function is called when an entity is created. start's arguments are user-defined, but all must be provided to the language keyword `spawn` that starts the entity. `start` can be considered a sort of constructor.
  - `stop` - This function is called when an entity is destroyed with `kill`. `stop` is considered like a destructor.
  - `step` - Step is called 60 times a second on any entities that have 'start'ed.
  - `render` - Render is also called 60 times a second, but is called on each entity after every entity with a step function has had step called (i.e. in a program with 2 entities, SEAMScript will call `step` on both first, and then call `render` on both. Render is highly recommended not to modify any variable value, and should just be used for drawing/output work, but it is to use render as a general-purpose function.

Entities `step` and `render` functions are called in the order the entities are 'spawned'. When an entity is removed with `kill`, the order in which `step` and `render` functions are called is not modified, except to remove the 'dead' entity from the list. Neither step nor render should contain infinite loops; this will prevent the program from running. Some examples of entity definition and use are:

```
entity Player:
    int score
    string name

    function start():
       ...initialization code...

    function stop():
        ...stop code...

    function step():
        ...step code...

    function render():
        ...draw code...
```

3.10 Built-In Entities

To facilitate rapid development of certain types of applications, SEAMScript contains a few built-in objects that behave like entities. These built-ins are:

3.10.1 screen

- Properties
  - `width` - The width of the display screen. (int)
  - `height` - The height of the display screen. (int)
- Methods
  - `draw_sprite(texture tex, int x, int y)` - returns 0 - Draw a texture 'tex' to the screen at `x`, `y`.
  - `draw_rect(int color, int x, int y, int width, int height)` - returns 0 - Draw a filled rectangle with no border with the color `color`, width `width`, height `height`, x-position `x`, and y-position `y`.
  - `log(string to_log)` - returns 0 - Logs the string `to_log` to `stdout`.

3.10.2 keyboard

- Properties
  - `{left,right,up,down,space}.pressed` - returns boolean - Whether one of the listed keys has been pressed. Once checked, subsequent checks will return false until a complete key up/key down event has been performed again. For example,

    ```
    if(keyboard.left.pressed == true):
        screen.log(\Left pressed!")
    ```

    would be a valid use of this property.
- Methods
  - (NONE)

3.10.3 loader

- Properties
  - (NONE)
- Methods
  - `load_tex(string filename, int desired_width, int desired_height)` - returns a texture - The only way to load a texture, `load_tex` takes in a filename, width, and height, and generates a texture of those parameters. If the given file (expected to be in a directory relative to the executable) is not found, a runtime error is created and the program will crash.

3.11 Built-In Functions

Built-in functions provide conversion facilities.

- `int_to_string(int i)` - Convert `i` to its string representation.
- `int_to_float(int i)` - Convert `i` to its floating-point representation. This may result in a slight loss of precision.
- `int_to_boolean(int i)` - Convert `i` to its boolean representation. `0` will be converted to `false`, while everything else will be converted to `true`.
- `float_to_int(float f)` - Convert `f` to its integer representation. If the floating-point value exceeds what is representable in integers, or has a decimal portion, a loss of precision will result.
- `float_to_string(float f)` - Convert `f` to its string representation. Up to 4 decimals places will be printed.

- `boolean_to_int(boolean b)` - Convert `b` to its integer representation. `false` will be converted to `0`, while true will be converted to `1`.
- `boolean_to_string(boolean b)` - Convert `b` to its string representation. false will be converted to `false`, and true will be converted to `true`.

3.12 Layout

The layout of a SEAMScript program is as follows (in order):

- File includes (optional); see file structure
- Global variable declarations (optional); assignment here is not supported
- Function definitions (optional)
- Entity definitions (optional, but necessary for any real work)
- Main (required) - A function named `main` that's the entry point of the program. Main is responsible for initially staging all entities. If a program needs to restage entities regularly, developers should consider creating an entity that does staging in its step function depending on various state values stored in global variables (e.g. a couple variables called `level` and `is_done`, and an entity of type `level` would be a canonical way to do it). The main function is called once the program starts and is never called again. Although the compiler doesn't check, the main function should not contain an infinite loop. Since the functions for `step` and `render` on entities are not called on a regular basis until after main concludes, infinite loops will prevent the program from running.

An example layout would be as follows:

```
include \tilemap.seam"
...more includes...

int level
...more global variables...

int get_current_terrain(int x, int y):
    ...function definition...
...more user functions...

entity Player:
    ...player definition...
...more entity declarations...

function main():
    player = spawn(Player, 50, 50)
    ...more init code...
```

3.13 File Structure

SEAMScript does not have a robust system of library supports, but it is possible to approximate libraries by including other files in your program so long as there are no namespace conflicts. To other file in your program, whose mains will be called before the main of your program, and in the order they are included, using the following syntax:

```
include "filename.seam"
```

3.14 Function Definitions

Functions must be defined before they are called in SEAMScript. A function declaration must adhere to the following format:

```
<return type OR "function" keyword> <identifier> (<argument list>):
    ...block of statements...
```

If a function definition begins with the `function` keyword, it is implied that the function does not return a value (similar to `void` in C-like languages). The argument list consists of 0 or more identifiers separated by commas. The block of statements describing the function's behavior must be one indentation level past that of the function declaration itself. If a return type is specified (i.e. the declaration begins with a primitive type rather than `function`), the function block must contain a `return` statement, which returns control to the calling function and returns the value of the expression following the `return` keyword.

4 Project Outline

Entering COMS 4115, our group members were acquainted with one another's programming backgrounds, so we assigned jobs to everyone. Our group always met every Monday and Wednesday after class, and usually we would get dinner while discussing what we had accomplished in the previous week, and our goals for the next. We would also show each other our results by posting on the group chat that we had made a change, and we would then tell everyone else to pull the latest commit to check it out. Within Team SEAM, there was a strict timeline that had set due dates for each component of the program. Hence we were each assigned several things to get done before finals week started.

On that note, we rationed out roles. Sean was our Manager, Maclyn was our Language Guru, Akira was our System Architect, and Edmund was our Tester. We eventually found that despite the roles, we usually worked together to address some issues, and specialized according to job or module for others. Still, most of us satisfied these roles to a certain extent. Sean did make sure to remind everyone of deadlines and of problems and corner cases we had not addressed. Maclyn figured out how to wire in external functions without polluting our language's namespace. Akira was responsible for restructuring after

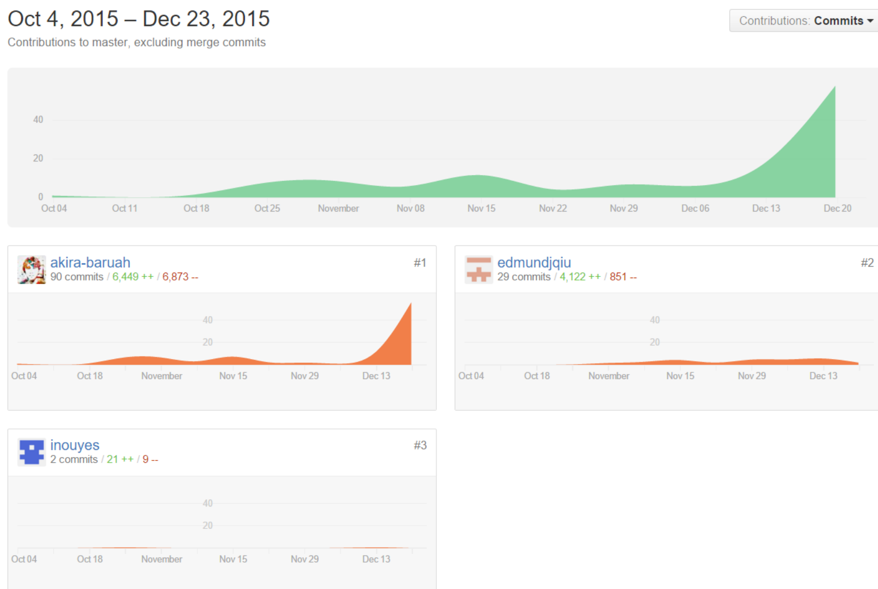

Figure 1: Commit History

it became clear that our prior, simplistic additions to MicroC were not enough. Edmund built a semantic checking module, and conducted tests on those.

Team members were encouraged to write clear, legible code. We did not have a formal style guide per se, but we expected each other to comment appropriately, and where necessary. Not only did we work with a language which we had just learned, but we also had to specify how the language generated C code. This level of indirection meant that when OCaml code approached low level details, we had to comment and explain more clearly what it was doing. After all, part of the goal of keeping code modular was allowing others to understand our own modules.

As for our development environments, we used a varied assortment of tools individually. Akira was used to emacs, Sean used Sublime and vim, and Edmund and Maclyn used vim and gvim. Since we had to incorporate the SDL library to our project, we soon realized that we had to set up build environments on different operating systems. To help alleviate this difficulty, we also used Docker, a container engine for isolating build environments. As for version control, all members were most comfortable with git, so we used git as our version control system. We also used Github to host our project, so our repo can be explored at `https://github.com/teamSEAM/ProjectSEAM`, and seen as our sort of project log.

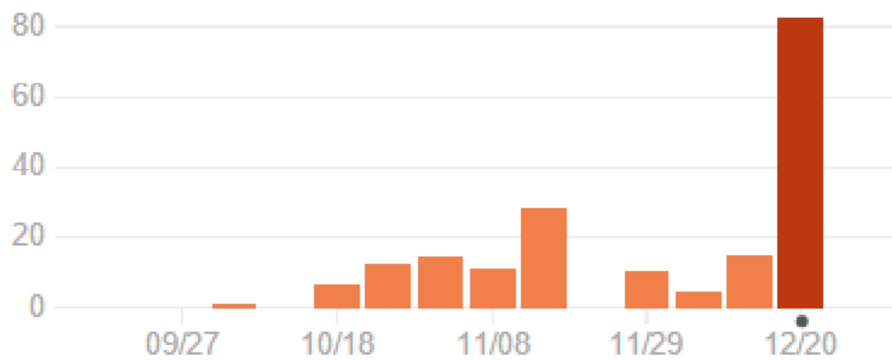

Figure 2: Commit History (total)

5 Architectural Design

6 Testing Strategy

```

Test program in SEAM:

entity World:
        func start():
                screen.init(100, 100)
                screen.out("Entities Exist")
entity Two:
        func two():
                screen.out("NOT GONNA PRINT")

The same program compiled to C:

#include "lib.h"
#include "gen.h"
typedef struct World {

} World;
void World_start(World *this) {

_screen_init(100, 100);
_screen_out("Entities Exist");
}

void World_step(void *in) {
World *this = (World *)in;
```

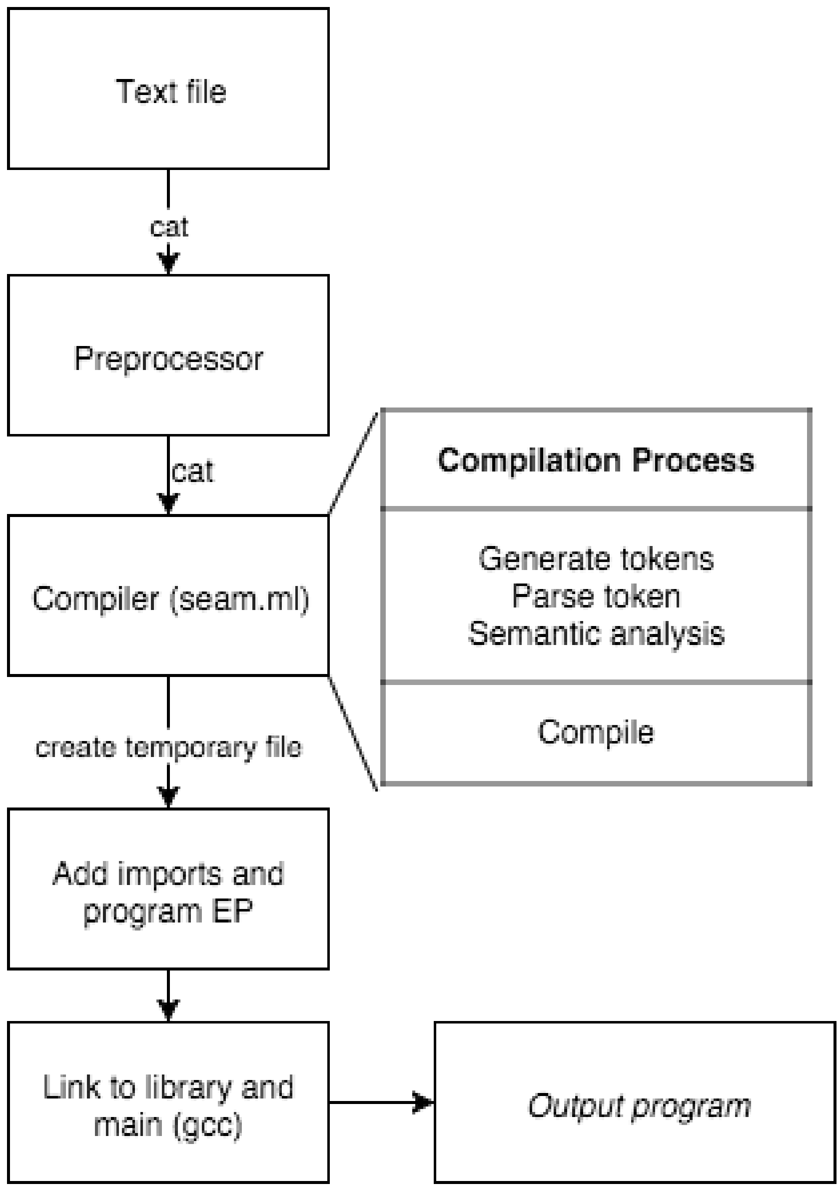

Figure 3: Architecture Block Diagram

```
}

void World_stop(World *this) {


}

void World_render(void *in) {
World *this = (World *)in;


}

World* World_spawn(){
     World *data = malloc(sizeof(World));
    entity_node *node = malloc(sizeof(entity_node));
    if(!data || !node) _seam_fatal("Allocation error!");

    node->step = &World_step;
    node->render = &World_render;
    node->data = data;
    node->next = NULL;

    entity_node *curr = ehead;
    while(curr && curr->next) curr = curr->next;

    if(curr)
        curr->next = node;
    else
        ehead = node;

    World_start(data);
    return data;
}
void World_destroy(World *this){
     World_stop(this);

    entity_node *curr = ehead;
    entity_node *prev = NULL;
    while(curr) {
        if(curr->data == this) break;
        prev = curr;
        curr = curr->next;
    }

    if(prev)
        prev->next = curr->next;
    else
    ehead = curr->next;

    free(this);
    free(curr);
}
typedef struct Two {
```

```
} Two;
void Two_two(Two *this) {

_screen_out("NOT GONNA PRINT");
}

void Two_step(void *in) {
Two *this = (Two *)in;


}

void Two_start(Two *this) {


}

void Two_stop(Two *this) {


}

void Two_render(void *in) {
Two *this = (Two *)in;


}

Two* Two_spawn(){
     Two *data = malloc(sizeof(Two));
    entity_node *node = malloc(sizeof(entity_node));
    if(!data || !node) _seam_fatal("Allocation error!");

    node->step = &Two_step;
    node->render = &Two_render;
    node->data = data;
    node->next = NULL;

    entity_node *curr = ehead;
    while(curr && curr->next) curr = curr->next;

    if(curr)
        curr->next = node;
    else
        ehead = node;

    Two_start(data);
    return data;
}
void Two_destroy(Two *this){
     Two_stop(this);

    entity_node *curr = ehead;
    entity_node *prev = NULL;
    while(curr) {
        if(curr->data == this) break;
        prev = curr;
```

```
        curr = curr->next;
    }

    if(prev)
        prev->next = curr->next;
    else
    ehead = curr->next;

    free(this);
    free(curr);
}
 void program_ep() { World_spawn(); }
```

7 Lessons Learned

7.1 Sean (si2281)

My role in the project was to be the manager. Initially reading about what the manager was responsible for, I felt that I was going to get a lot less coding responsibility compared to the others who were actually in charge of their own portion of the code; however I learned how vital and important it was to have a manager because sometimes in a group setting with people doing their own portion of the code, there needs to be someone that is communicating with all the members. There were moments in this year where one person could be way ahead of everyone else in their own section and in a way, even if it may sound like a good thing, it is also a bad thing. In situations where a feature is being cut or the program is built slightly different from what it was originally supposed to be, the person that went way ahead of everyone will have to scrap the majority of the work due to limitations in the code. As the manager, I realized that it was important to keep everyone on track and also around the same place. I also realized that as the manager, if there was a place that required my attention, then I should help the team out by doing what needed to be done to keep everyone at the same place. For future PLT members, I highly recommend having a good timeline but also create individual timelines for each role so that everyone will have a good idea what to do. It is not a good idea to have someone go really far ahead because it may not work.

7.2 Edmund (ejq2106)

I was mostly responsible for the working preprocessor, and for developing semantic analysis. I realized that the fact that we were translating rather than generating bytecode made the job more modular, since I did not need to produce a checked abstract syntax tree for the compiler module. Therefore, I designed my module, `semantic.ml`, to take the AST and to produce error messages if it finds anything at fault, so that the rest of the compiler will not run if there are any problems. In theory, this made the division of labor more clear, but in practice, as specifications changed and features were added and dropped, it became harder for me to keep my module on top of the latest revisions. I ended up with quite a bit to do at the end when a few issues we ran into in code gen required me to overhaul the structure of my semantic checker. I had fortunately written some functions that were generic enough to easily reuse and adapt into the final, but other functions I had to discard. Therefore, I would recommend that any future teams get very comfortable with the basics of OCaml, since it is quite likely that they'll have to adapt and change their implementation. For example, try to know a good part of the List and String module. And, of course, I would echo the prevailing tidbit of advice, which is to start early.

7.3 Akira (akb2158)

My largest contributions to this project involved designing and implementing the abstract syntax tree (AST) used to parse SEAMscript programs. I ended up writing the majority of the AST (`ast.ml`), scanner (`scanner.mll`), parser (`parser.mly`), and translator (`compile.ml`) using OCaml, drawing upon Stephen Edwards's microC example as an initial reference. I found it very helpful to model our initial compiler pipeline on a gold standard in order to adhere to best practices and avoid reinventing the wheel when possible.

Furthermore, I found it quite productive to integrate my designs with those of my team members, who were working on other components of the SEAMscript compiler. For example, I was able to reuse the linked list scoping structures used by Edmund's semantic checker. As Maclyn discusses below, we were able to tightly integrate the SEAMscript-to-C translator by formally writing the C code expected to be generated by a given SEAMscript program. Using this top-down approach, we were able to quickly converge the compiler frontend and C backend interfaces. Finally, Sean's test scripts were very useful in quickly debugging issues within the entire compiler stack.

As everyone can attest, slow and steady progress is much preferred over a mad sprint towards the end of the project. The earlier the team gets their hands dirty working on the parser and AST, the better.

7.4 Maclyn (mgb2163)

I wrote most of the boilerplate the generate code interacted with and was responsible for structuring the output program of the compiler so it could link with the boilerplate. Since we compiled into C, it was really helpful for Akira, who wound up doing most of the compilation work, for me to write a sample SEAM file and its expected conversion into C. It also helped me figure out how to work everything together. While waiting for certain compiler features to get implemented, I wrote a tester for my library, which was also quite helpful. I would caution strongly against separating code gen and the runtime of your compiled language too much, as differences in expectations of the compiler component early on led some messiness when Akira and I went to merge our work. Also, don't put it off!

8 Appendix

Listing 1: ast.ml

```
(* signed off: Akira, Edmund, Maclyn, Sean *)
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq |
    Greater | Geq
type dtype = Bool | Int | String | Float | Instance of string |
    Array of dtype * int | Texture
type rtype = Void | ActingType of dtype

type literal =
| LitBool of bool
| LitInt of int
| LitFloat of float
| LitString of string
| LitArray of literal * int

type identifier =
| Name of string
| Member of string * string  (* entity id, member id *)

type expr =
| Literal of literal
| Id of identifier               (* variables and fields *)
| Call of identifier * expr list (* functions and methods *)
| Binop of expr * op * expr
| Spawn of string
| Assign of identifier * expr
| Access of identifier * expr    (* array access *)
| Noexpr

type stmt =
| Block of stmt list
| Expr of expr
| Return of expr
| If of expr * stmt * stmt
| For of expr * expr * expr * stmt
| While of expr * stmt
| Kill of identifier

type vdecl = dtype * string

type fdecl = {
  rtype : rtype;
  fname : string;
  formals : vdecl list;
  locals : vdecl list;
  body : stmt list;
}

type edecl = {
  ename : string;
  fields : vdecl list;
  methods : fdecl list;
}
```

```
type program = edecl list

let string_of_op = function
  | Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
  | Equal -> "==" | Neq -> "!="
  | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="

let rec string_of_dtype = function
  | Bool -> "bool"
  | Int -> "int"
  | String -> "string"
  | Float -> "float"
  | Array(t, size) ->
    string_of_dtype t ^ "[" ^ string_of_int size ^ "]"
  | Instance(name) -> name
  | Texture -> "texture *"

let string_of_rtype = function
  | Void -> "void"
  | ActingType(at) -> string_of_dtype at

let rec string_of_literal = function
  | LitBool(b) -> string_of_bool b
  | LitInt(b) -> string_of_int b
  | LitString(s) -> s
  | LitFloat(f) -> string_of_float f
  | LitArray(l, size) ->
    string_of_literal l ^ "[" ^ string_of_int size ^ "]"

let rec string_of_identifier = function
  | Name(name) -> name
  | Member(parent, name) -> parent ^ "." ^ name

let name_of_identifier = function
  | Name(name) -> name
  | Member(parent, name) -> name

let parent_of_identifier = function
  | Name(name) -> ""
  | Member(parent, name) -> parent

let rec string_of_expr = function
  | Literal(lit) -> string_of_literal lit
  | Id(id) -> string_of_identifier id
  | Binop(e1, o, e2) ->
    string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Assign(id, e) -> string_of_identifier id ^ " = " ^ string_of_expr e
  | Access(id, e) -> string_of_identifier id ^ "[" ^ string_of_expr e ^ "]"
  | Spawn(ent) -> "spawn " ^ ent
  | Call(id, args) ->
    string_of_identifier id ^
      "(" ^ String.concat ", " (List.map string_of_expr args) ^ ")"
  | Noexpr -> ""
```

```
let rec string_of_stmt = function
  | Block(stmts) ->
    "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Kill(id) -> "kill " ^ string_of_identifier id ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) ->
    "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | 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) ->
    "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_vdecl (t, id) = string_of_dtype t ^ " " ^ id ^ ";\n"

let string_of_formal (t, id) = string_of_dtype t ^ " " ^ id

let string_of_fdecl fdecl =
  string_of_rtype fdecl.rtype ^ " " ^ fdecl.fname ^ "(" ^
    String.concat ", " (List.map string_of_formal fdecl.formals) ^ ")\n{\n" ^
    String.concat "" (List.map string_of_vdecl fdecl.locals) ^
    String.concat "" (List.map string_of_stmt fdecl.body) ^
    "}\n"

let string_of_edecl edecl =
  "entity " ^ edecl.ename ^ "\n{\n" ^
    String.concat "" (List.map string_of_vdecl edecl.fields) ^ "\n" ^
    String.concat "" (List.map string_of_fdecl edecl.methods) ^
    "}\n"

let string_of_program entities =
  String.concat "\n" (List.map string_of_edecl entities)
```

```
{ open Parser }
(* signed off: Akira, Maclyn, Edmund, Sean *)
(* Generally useful regexes *)
let digit  = ['0'-'9']
let lower  = ['a'-'z']
let upper  = ['A'-'Z']
let letter = (upper | lower)
let minus  = ['-']
let plus   = ['+']
let sign   = (plus | minus)
let exp    = ['e' 'E'] sign? (digit+)

(* Literals *)
let lit_bool   = "true" | "false"
let lit_int    = minus? (digit+)
let lit_string = '"' [^'"']* '"'
let lit_float  = minus? (digit*) ['.']? (digit+) (exp)?
let regex_lit = (lit_bool | lit_int | lit_string | lit_float)
```

```

(* Identifiers *)
let regex_id = (letter | ’_’) ((letter | digit | ’_’)*)

(* Primitives *)
let type_bool     = "bool"
let type_int      = "int"
let type_string   = "string"
let type_float    = "float"
let type_instance = "instance " regex_id
let type_texture  = "texture"
let regex_type =
  (type_bool | type_int | type_string | type_float |
      type_instance | type_texture)

rule token = parse
  [’ ’ ’\t’ ’\r’ ’\n’] { token lexbuf } (* Whitespace *)
| ’#’      { comment lexbuf }           (* Comments *)
| ’(’      { LPAREN }
| ’)’      { RPAREN }
| ’{’      { LBRACE }
| ’}’      { RBRACE }
| ’[’      { LBRACKET }
| ’]’      { RBRACKET }
| ’;’      { SEMI }
| ’,’      { COMMA }
| ’.’      { DOT }
| ’+’      { PLUS }
| ’-’      { MINUS }
| ’*’      { TIMES }
| ’/’      { DIVIDE }
| ’=’      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| ’<’      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "bool"   { BOOL }
| "int"    { INT }
| "float"  { FLOAT }
| "string" { STRING }
| "entity" { ENTITY }
| "func"   { FUNC }
| "texture"{ TEXTURE }
| "spawn"  { SPAWN }
| "kill"   { KILL }
| lit_bool as b   { LIT_BOOL(bool_of_string b) }
| lit_int as i    { LIT_INT(int_of_string i) }
| lit_float as f  { LIT_FLOAT(float_of_string f) }
| lit_string as s { LIT_STRING(s) }
| regex_id as id  { ID(id) }
```

```
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
    char)) }

and comment = parse
  '#' { token lexbuf }
| _    { comment lexbuf }
```

```
%{
  (* signed off: Akira, Macyln, Edmund, Sean *)
  open Ast
%}

%token BOOL INT FLOAT STRING
%token ENTITY FUNC TEXTURE
%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET
%token SEMI COMMA DOT
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE
%token SPAWN KILL
%token <string> ID
%token <bool>   LIT_BOOL
%token <int>    LIT_INT
%token <float>  LIT_FLOAT
%token <string> LIT_STRING
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right SPAWN
%right KILL
%left DOT

%start program
%type <Ast.program> program

%%

program:
 | edecls EOF { List.rev $1 }

edecls:
 | /* nothing */ { [] }
 | edecls edecl { $2 :: $1 }

edecl:
 | ENTITY ID LBRACE vdecl_list fdecl_list RBRACE
     { { ename = $2;
         fields = List.rev $4;
         methods = $5; } }

fdecl_list:
```

```
 | /* nothing */    { [] }
 | fdecl fdecl_list { $1 :: $2 }

fdecl:
 | dtype ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { rtype = ActingType($1);
         fname = $2;
         formals = $4;
         locals = List.rev $7;
         body = List.rev $8; } }
 | FUNC ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { rtype = Void;
         fname = $2;
         formals = $4;
         locals = List.rev $7;
         body = List.rev $8; } }

formals_opt:
 | /* nothing */ { [] }
 | formal_list   { List.rev $1 }

formal_list:
 | dtype ID                   { [ ($1, $2) ] }
 | formal_list COMMA dtype ID { ($3, $4) :: $1 }

vdecl_list:
 | /* nothing */    { [] }
 | vdecl_list vdecl { $2 :: $1 }

vdecl:
 | dtype ID SEMI { $1, $2 }

dtype:
 | BOOL   { Bool }
 | INT    { Int }
 | FLOAT  { Float }
 | STRING { String }
 | LT ID GT { Instance($2) }
 | dtype LBRACKET LIT_INT RBRACKET { Array($1, $3) }
 | TEXTURE { Texture }

stmt_list:
 | /* nothing */  { [] }
 | stmt_list stmt { $2 :: $1 }

stmt:
 | expr SEMI { Expr($1) }
 | RETURN expr SEMI { Return($2) }
 | LBRACE stmt_list RBRACE { Block(List.rev $2) }
 | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
 | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
 | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
 | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
 | KILL ID SEMI { Kill(Name($2)) }
```

```

expr_opt:
 | /* nothing */ { Noexpr }
 | expr          { $1 }

expr:
 | literal          { Literal($1) }
 | id               { Id($1) }
 | expr PLUS   expr { Binop($1, Add,   $3) }
 | expr MINUS  expr { Binop($1, Sub,   $3) }
 | expr TIMES  expr { Binop($1, Mult,  $3) }
 | expr DIVIDE expr { Binop($1, Div,   $3) }
 | expr EQ     expr { Binop($1, Equal, $3) }
 | expr NEQ    expr { Binop($1, Neq,   $3) }
 | expr LT     expr { Binop($1, Less,  $3) }
 | expr LEQ    expr { Binop($1, Leq,   $3) }
 | expr GT     expr { Binop($1, Greater,  $3) }
 | expr GEQ    expr { Binop($1, Geq,   $3) }
 | SPAWN ID         { Spawn($2) }
 | id ASSIGN expr   { Assign($1, $3) }
 | id LBRACKET expr RBRACKET    { Access($1, $3) }
 | id LPAREN actuals_opt RPAREN { Call($1, $3) }
 | LPAREN expr RPAREN { $2 }

literal:
 | LIT_BOOL { LitBool($1) }
 | LIT_INT { LitInt($1) }
 | LIT_FLOAT { LitFloat($1) }
 | LIT_STRING { LitString($1) }

id:
 | ID          { Name($1) }
 | expr DOT ID { Member(string_of_expr $1, $3) }

actuals_opt:
 | /* nothing */ { [] }
 | actuals_list  { List.rev $1 }

actuals_list:
 | expr                     { [$1] }
 | actuals_list COMMA expr { $3 :: $1 }
```

```
(* signed off: Akira *)
open Ast
open Boilerplate

exception UndeclaredEntity of string
exception UndeclaredIdentifier of string

type symbol_table = {
  parent : symbol_table option;
  current_entity : edecl;
  variables : vdecl list;
}

type environment = {
  entities : edecl list;
  scope : symbol_table;
```

```
}

let rec string_of_scope s =
  "parent: " ^ (match s.parent with
  | None -> ""
  | Some(p) -> string_of_scope p) ^ ")\ncurrent_entity: " ^
    string_of_edecl s.current_entity ^ "\nvariables: " ^
    String.concat "; " (List.map string_of_vdecl s.variables)

let string_of_env env =
  "entities: " ^ String.concat ", " (List.map string_of_edecl env.entities) ^
    "\nscope: " ^ string_of_scope env.scope ^ "\n"

let find_entity (env : environment) name =
  try List.find (fun e -> e.ename = name) env.entities
  with Not_found -> raise (UndeclaredEntity name)

let rec find_variable (scope : symbol_table) name =
  try List.find (fun (_, n) -> n = name) scope.variables
  with Not_found ->
    match scope.parent with
      Some(parent) -> find_variable parent name
    | _ -> raise (UndeclaredIdentifier name)

let find_function (scope : symbol_table) name =
  try List.find (fun f -> f.fname = name) scope.current_entity.methods
  with Not_found -> raise (UndeclaredIdentifier name)

let add_edecl env edecl = {
  entities = edecl :: env.entities;
  scope = {
    parent = None;
    current_entity = edecl;
    variables = edecl.fields;
  };
}

let add_scope env vdecls = {
    entities = env.entities;
    scope = {
      parent = Some(env.scope);
      current_entity = env.scope.current_entity;
      variables = vdecls;
    };
}

let in_scope scope name =
  try
    let _ = (List.find (fun (_, n) -> n = name) scope.variables) in
    true
  with Not_found -> false

let rec is_field scope name =
  match scope.parent with
  | None ->
```

```
    if (in_scope scope name) then true
    else true (* raise (UndeclaredIdentifier name) *)
  | Some(parent) ->
    if (in_scope scope name) then false
    else is_field parent name

let pop_scope env =
  match env.scope.parent with
  | Some(new_scope) ->
    {
      entities = env.entities;
      scope = new_scope;
    }
  | None -> raise (Failure "Attempting to pop from empty environment")

let tr_identifier env id =
  (if (is_field env.scope (name_of_identifier id)) then
      "(this->" else "(") ^
    (match parent_of_identifier id with
        | "" ->  name_of_identifier id ^ ")"
        | _ -> parent_of_identifier id ^ ")->" ^ name_of_identifier id)

let is_builtin name =
  try let _ = List.find (fun s -> s = name) Lib.modules in true
  with Not_found -> false

let rec tr_expr env = function
  | Literal(lit) -> string_of_literal lit
  | Id(id) -> tr_identifier env id
  | Binop(e1, o, e2) ->
    (tr_expr env) e1 ^ " " ^ string_of_op o ^ " " ^ (tr_expr env) e2
  | Assign(id, e) -> tr_identifier env id ^ " = " ^ (tr_expr env) e
  | Access(id, e) -> tr_identifier env id ^ "[" ^ (tr_expr env) e ^ "]"
  | Spawn(ent) -> ent ^ "_spawn()"
  | Call(id, args) ->
    (match id with
    | Name(n) -> if (n = "load") || (n = "unload")
      then "_" ^ n ^ "_tex(" ^ String.concat ", " (List.map (tr_expr env) args) ^ ")"
      else tr_identifier env id ^ "(" ^
        String.concat ", " (List.map (tr_expr env) args) ^ ")"
    | Member(p, n) ->
      if is_builtin p then "_" ^ p ^ "_" ^ n ^
        "(" ^ String.concat ", " (List.map (tr_expr env) args) ^ ")"
      else tr_identifier env id ^
        "(" ^ String.concat ", " (List.map (tr_expr env) args) ^ ")")
  | Noexpr -> ""

let rec tr_stmt env = function
  | Block(stmts) ->
    "{\n" ^ String.concat "\n" (List.map (tr_stmt env) stmts) ^ "\n}"
  | Expr(expr) -> (tr_expr env) expr ^ ";";
```

```
  | Return(expr) -> "return " ^ (tr_expr env) expr ^ ";";
  | If(e, s, Block([])) ->
    "if (" ^ (tr_expr env) e ^ ") " ^ (tr_stmt env) s
  | If(e, s1, s2) ->
    "if (" ^ (tr_expr env) e ^ ") " ^ (tr_stmt env) s1 ^
      " else " ^ (tr_stmt env) s2
  | For(e1, e2, e3, s) ->
    "for (" ^ (tr_expr env) e1  ^ " ; " ^ (tr_expr env) e2 ^ " ; " ^
      (tr_expr env) e3  ^ ") " ^ (tr_stmt env) s
  | While(e, s) -> "while (" ^ (tr_expr env) e ^ ") " ^ (tr_stmt env) s
  | Kill(id) ->
    let iname = name_of_identifier id in
    let (dtype, _) = find_variable env.scope iname in
    let ename = string_of_dtype dtype in
    ename ^ "_destroy(" ^ (tr_identifier env id) ^ ");"

let rec tr_formal (typ, name) =
  match typ with
  | Bool -> "int " ^ name
  | Int -> "int " ^ name
  | String -> "char *" ^ name
  | Float -> "float " ^ name
  | Instance(s) -> s ^ " *" ^ name
  | Array(t, size) -> tr_formal(t, name) ^ "[" ^ string_of_int size ^ "]"
  | Texture -> "texture *" ^ name

let tr_vdecl vdecl = (tr_formal vdecl) ^ ";"

let is_stub fname =
  try let _ = List.find (fun stub -> fname = stub)
        Boilerplate.stubs_action in true
  with Not_found -> false

let tr_fdecl env fdecl =
  let env = add_scope env (fdecl.formals @ fdecl.locals) in
  let ename = env.scope.current_entity.ename in
  let mangled_fname = ename ^ "_" ^ fdecl.fname in
  let first_arg = if (is_stub fdecl.fname) then "void *in" else ename ^ " *this" in
  let rtype = fdecl.rtype in
  string_of_rtype rtype ^ " " ^ mangled_fname ^
    "(" ^ String.concat ", " (first_arg :: List.map string_of_formal fdecl.formals) ^
    ") {\n" ^
    (if (is_stub fdecl.fname)
     then ename ^ " *this = (" ^ ename ^ " *)in;\n" else "") ^
    String.concat "\n" (List.map tr_vdecl fdecl.locals) ^ "\n" ^
    String.concat "\n" (List.map (tr_stmt env) fdecl.body) ^ "\n}\n"

let update_stub edecl fdecl =
  try let _ = List.find (fun f -> f.fname = fdecl.fname)
        edecl.methods
      in edecl
  with Not_found -> {
    ename = edecl.ename;
```

```
    fields = edecl.fields;
    methods = List.rev (fdecl :: (List.rev edecl.methods));
  }

let tr_edecl (env, output) edecl =
  let stubs = [ {rtype = Void;
                 fname = "step";
                 formals = [];
                 locals = [];
                 body = [];
                };
                {rtype = Void;
                 fname = "start";
                 formals = [];
                 locals = [];
                 body = [];
                };
                {rtype = Void;
                 fname = "stop";
                 formals = [];
                 locals = [];
                 body = [];
                };
                {rtype = Void;
                 fname = "render";
                 formals = [];
                 locals = [];
                 body = [];
                }
              ]
  in
  let edecl = List.fold_left update_stub edecl stubs in
  let env = add_edecl env edecl in
  let ename = edecl.ename in
  let fields = List.map tr_vdecl edecl.fields in
  let methods = List.map (tr_fdecl env) edecl.methods in
  let translated = "typedef struct " ^ ename ^ " {\n" ^
    String.concat "\n" fields ^ "\n} " ^ ename ^";\n" ^
    String.concat "\n" methods ^ "\n" ^
    (gen_spawn ename) ^ "\n" ^
    (gen_destroy ename) in
  (env, translated :: output)

let translate entities =
  let empty_edecl = { ename = ""; fields = []; methods = [] } in
  let empty_env = {
    entities = [];
    scope = { parent = None; current_entity = empty_edecl; variables
         = [] };
  } in
  let (env, translated) = (List.fold_left tr_edecl (empty_env, [])
      entities) in
  String.concat "\n" (List.rev translated)
```

```
(* signed off: Maclyn *)
open Printf

let _ =
```

```
    try
        let lexbuf = Lexing.from_channel stdin in
        let program = Parser.program Scanner.token lexbuf in
        let verified = Semantic.semantic_check program in
        let result =
            if (String.compare verified "") == 0 then
                Compile.translate program
            else
                (
                    output_string stderr verified;
                    output_string stderr "Continuing anyways...\n";
                    Compile.translate program
                )
            in
        print_endline result;
    with
        Parsing.Parse_error ->
            (
                print_endline "Parsing error!";
                exit 1;
            )
        | _ -> exit 1
```

```
#!/bin/bash
# Called with [input program] [output program]

if [ $# -ne 2 ]
then
    echo "usage: $0 <input file> <output program>"
    exit 1
fi

# Check if libsdl2-dev is installed
dpkg-query -l libsdl2-dev > /dev/null
if [ "$?" -ne "0" ]
then
    echo "Warning: dpkg/libsdl2-dev not installed! Compilation may fail!"
fi

cat $1 | ./preprocessor > temp.seami
cat temp.seami | ./seam > gen.c
if [ "$?" -ne "0" ]
then
    echo "Error encountered while compiling: "
    cat gen.c

    rm temp.seami
    rm gen.c
    exit 1
else
    echo "Input program translated succesfully; compiling..."
fi

# See Google: http://superuser.com/questions/246837/how-do-i-add-text-to-the-beginning-of-a-file-in-bash
echo "#include \"gen.h\"" | cat - gen.c > temp && mv temp gen.c
echo "#include \"lib.h\"" | cat - gen.c > temp && mv temp gen.c
```

```

echo " void program_ep() { World_spawn(); }" >> gen.c

gcc -g -c lib.c -o lib.o
gcc -g -c gen.c -o gen.o
gcc -g -c main.c -o main.o
gcc -g main.o lib.o gen.o -lSDL2  -o $2

if [ "$?" -ne "0" ]
then
        echo "Compilation error! Checkout temp.seami and gen.c."
else
        rm temp.seami
#       rm gen.c
        echo "$2 created."
fi
```

```
#!/bin/bash
# signed off: Maclyn

# Called with [input program] [output program]

if [ $# -ne 2 ]
then
    echo "usage: $0 <input file> <output program>"
    exit 1
fi

# Check if libsdl2-dev is installed
dpkg-query -l libsdl2-dev > /dev/null
if [ "$?" -ne "0" ]
then
    echo "Warning: dpkg/libsdl2-dev not installed! Compilation may fail!"
fi

cat $1 | ./preprocessor > temp.seami
cat temp.seami | ./seam > gen.c
if [ "$?" -ne "0" ]
then
    echo "Error encountered while compiling: "
    cat gen.c

    rm temp.seami
    rm gen.c
    exit 1
else
    echo "Input program translated succesfully; compiling..."
fi

# See Google: http://superuser.com/questions/246837/how-do-i-add-text-to-the-beginning-of-a-file-in-bash
echo "#include \"gen.h\"" | cat - gen.c > temp && mv temp gen.c
echo "#include \"lib.h\"" | cat - gen.c > temp && mv temp gen.c

echo " void program_ep() { World_spawn(); }" >> gen.c

gcc -g -c lib.c -o lib.o
```

```
gcc -g -c gen.c -o gen.o
gcc -g -c main.c -o main.o
gcc -g main.o lib.o gen.o -lSDL2  -o $2

if [ "$?" -ne "0" ]
then
        echo "Compilation error! Checkout temp.seami and gen.c."
else
        rm temp.seami
#       rm gen.c
        echo "$2 created."
fi
```

```
(* signed off: Edmund *)
(* Working preprocessor. Still needs to be integrated into
   the project appropriately, but here it is. See
   src/tests/preprocessor_example.txt for an example of a
   file that would be handled by this *)


(* open the file, which I should figure out how to close *)
let myfile = stdin in

(* read in the lines one by one into a list *)
let rec input_lines file =
  match try [input_line file] with End_of_file -> [] with
    [] -> []
  | line -> line @ input_lines file

in

(* Function for removing comments now *)
let remove_comments lines =

  let rec eachlinehandler state_tuple current_string =

                (* grab stuff from tuples *)
    let comment_state = fst state_tuple in
    let current_list = snd state_tuple in

                (* first check if length of string is 0 *)
    if String.length current_string == 0 then
      ( comment_state, []  )
    else
      try
        let pound_index = String.index current_string '#' in
        let end_diff = (String.length current_string) - (pound_index + 1) in
        let ahalf = [String.sub current_string 0 pound_index;] in
        let bhalf = String.sub current_string (pound_index + 1) end_diff in

        if comment_state then
          let choice_tuple = (false, []) in
          let result_tuple = eachlinehandler choice_tuple bhalf in
          (fst result_tuple, current_list @ (snd result_tuple))
        else
          let choice_tuple = (true, []) in
```

```
          let result_tuple = eachlinehandler choice_tuple bhalf in
          (fst result_tuple, (current_list @ (ahalf @ (snd result_tuple))))
      with
        Not_found ->
          if comment_state then
            (true, [])
          else
            (false, current_string :: [])
  in


        (* now use the recursive line handler to do things *)

  let remove_comment_aux aux_tuple next_line =
              (* cumulative list and whether we're starting with a comment *)
    let start_with_comment = fst aux_tuple in
    let list_so_far = snd aux_tuple in

              (* eachlinehandler spits out (still comment?, [list, of, strings] *)
    let result_tuple = eachlinehandler (start_with_comment, []) next_line in
    let new_string_tokens = snd result_tuple in

              (* put the small strings together into one line again, backwards *)
    (fst result_tuple, String.concat "" new_string_tokens :: list_so_far)
  in

        (* call auxiliary function with the lines, then reverse the output *)
  let results = List.fold_left remove_comment_aux (false, []) lines in
  List.rev (snd results)
in


(* read in all lines, then remove the comments *)
let lineList = remove_comments (input_lines myfile) in


(* this is where the indent-removal magic happens*)

let rec process_indents current_list current_indent_level =

        (* returns whether string is only whitespaces *)
  let only_whitespace my_string =
    let length = String.length my_string in
    let rec check_whitespace pos =
      if pos == length then true
      else
        let item = String.get my_string pos in
        if (item == '\t' || item == ' ') then
          true && check_whitespace (pos + 1)
```

```
      else false
    in check_whitespace 0
  in

      (* counts the number of tabs in the left side *)
  let count_tabs my_string =
    let length = String.length my_string in
    let rec count_tabs_rec pos =
      if String.get my_string pos == '\t' then
        1 + count_tabs_rec (pos + 1)
      else 0 in
    if length == 0 then 0 else count_tabs_rec 0
  in

      (* make new line *)
  let make_new_line my_string =
    try
      let colon_index = String.rindex my_string ':' in
      String.concat "" [(String.sub my_string 0 colon_index); " {";]
    with
      Not_found -> String.concat ""  [my_string; "; ";]
  in

      (* generates a string of n number of tabs together *)
  let generate_n_tabs n =
    let rec tab_list tabs =
      if tabs <= 0 then []
      else
        "\t" :: (tab_list ( tabs - 1)) in
    String.concat "" (tab_list n)
  in

      (* n is the number of brackets we need,
         old_level is the indentation level we left, so we can
         properly tab and indent, and make everyting look nice*)
  let generate_n_close_brackets n old_level=
    let rec bracket_list brackets level=
      if brackets <= 0 then []
      else
        let rest_of_list = bracket_list (brackets - 1) (level - 1) in
        generate_n_tabs (level - 1) :: "} \n" :: rest_of_list in
    String.concat "" (bracket_list n old_level)
  in

  match current_list with
              (* if we have *)
  | [] -> String.concat "" [ generate_n_close_brackets current_indent_level
                             current_indent_level;]
  | head :: tail ->
    if only_whitespace head then
                               (* okay just do the next line *)
      process_indents tail current_indent_level
    else
```

```
                                (* Finds the closing brackets necessary based
                                   on the indentation level *)
      let new_indent_level = count_tabs head in
      let close_needed = current_indent_level - new_indent_level in
      let new_close_brackets = generate_n_close_brackets close_needed
        current_indent_level in
      if (String.length new_close_brackets) > 0 then
        String.concat "" [      (* We have to close some brackets*)
          new_close_brackets;
          (make_new_line head);"\n";
          process_indents tail new_indent_level;]

      else
        String.concat "" [
                                                (* stick on same indent level *)
          make_new_line head; "\n";
          process_indents tail new_indent_level;]
in

print_endline (process_indents lineList 0) ;;

semantic.ml - Edmund

include Errors (* note how if we need Ast, Errors includes Ast *)

module IntMap = Map.Make(struct type t = int let compare = compare end) (* for int map support *)
module StringMap = Map.Make(String)


(* The following is my procedure:

    Perform repeat entity declaration checks
    Perform repeat function declaration checks
    Perform repeat variable declaration checks
    Iterate through the functions to check everything

 *)

type translation_env = {
    current_scope: int;
    (* a map from scopes to the map of things in each scope,
    which maps the variable name to a vdecl *)
    variables: vdecl StringMap.t IntMap.t;
    entities: edecl StringMap.t;
    functions: fdecl StringMap.t;

    (* errors *)
    errors: error list;
}


(* ////////////////////////////////////////////////////////////////
        auxiliary functions for variables and scoping *)
```

```
(* first let's introduce this auxiliary function for the add_var_decl
    and also useful in expression checking *)
let find_variable_scope env var =
    let current_scope = env.current_scope in
    let rec search_scope scope_number =
        (* -1, didn't find *)
        if scope_number < 0 then scope_number
        else
            (* get the map corresponding to this scope *)
            let var_map = IntMap.find scope_number env.variables in
            (* see whether the variable is present *)
            let result = StringMap.mem var var_map in
            if result then
                scope_number
            else
                search_scope (scope_number - 1)
        in
    search_scope current_scope


let add_var_decl env possible_error_locus var_decl =

    (* use find_variable_scope *)
    let var_name = snd var_decl in
    let scope_number = find_variable_scope env var_name in

    (* react accordingly *)
    if scope_number == env.current_scope then
        (* error, we have a duplicate variable declaration
                inside the same scope... *)
        let new_error = (
                possible_error_locus,
                VariableRepeatDecl(var_decl))
            in
        { env with errors = new_error :: env.errors }

    else
        (* whether NOT FOUND or declared in an earlier scope
            it's okay, we're adding it to the current scope now *)
        let current_stringmap = IntMap.find env.current_scope env.variables in
        let updated_stringmap = StringMap.add var_name var_decl current_stringmap in
        let updated_mapping = IntMap.add env.current_scope updated_stringmap env.variables in
        { env with variables = updated_mapping; }


(* Auxiliary function to set a given scope's variables to zero *)
let clear_variable_scope env scope_number =
    let revised_variables =
        let empty_stringmap = StringMap.empty in
        IntMap.add scope_number empty_stringmap env.variables
        in
    let fixed_env = { env with variables = revised_variables;}
```

```
    in fixed_env

let make_basic_env =
    let empty_intmap = IntMap.empty in
    let basic_environment =
    {
        current_scope = 0;
        variables = empty_intmap;
        entities = StringMap.empty;
        functions = StringMap.empty;
        errors = [];
    } in
    clear_variable_scope basic_environment 0


(* In fact searching for the ID should be generalized *)
let check_id_usage env expr error_locus identifier = match identifier with
    | Member(entity, id_name) ->

        (env, Void )
        (* use our searcher *)
    | Name(id_name) ->
        let scope = find_variable_scope env id_name in
        if scope < 0 then
            (* We didn't even find it gg *)
            (* Error message in environment, then spit out a Void result *)
            let new_error = (error_locus, UndeclaredVariable(id_name, expr)) in
            let updated_env = { env with errors = new_error :: env.errors } in
            ( updated_env, Void)
        else
            (* this is a Stringmap *)
            let var_map = IntMap.find scope env.variables in
            let dtype = fst (StringMap.find id_name var_map) in
            let wrapped_dtype = ActingType(dtype) in
            ( env, wrapped_dtype)




(* /////////////////////////////////////////////////////////////
    the meat of the checking is here  *)

(* We will return a type of rtype, with the possibility of Void,
    the absense of return *)
let rec check_expression env func error_locus expr = match expr with
| Noexpr -> (env, Void)
| Literal (lit) ->
    let lit_dtype_lookup = function
        | LitBool(b) -> Bool
        | LitInt(i) -> Int
        | LitFloat(f) -> Float
        | LitString(s) -> String
        | LitArray(_, _) -> Int in
```

```
    let equiv_dtype = match lit with
    | LitArray(inner_lit, i) -> ActingType(
        Array( lit_dtype_lookup inner_lit, i) )
    | LitBool(b) -> ActingType(Bool)
    | LitInt(i) ->ActingType(Int)
    | LitFloat(f) ->ActingType(Float)
    | LitString(s) ->ActingType(String)         in (env, equiv_dtype
        )

| Call(id, []) -> (env, Void ) (*of identifier * expr list (*
    functions and methods *) *)
| Call(id, hd::tl) -> (env, Void ) (*of identifier * expr list (*
    functions and methods *) *)
| Binop(e1, o, e2) ->
    (* First, check e1 and e2 *)
    let tuple1 = check_expression env func error_locus e1 in
    let tuple2 = check_expression (fst tuple1) func error_locus e2
        in

    (* Next, compare their types *)
    let type1 = snd tuple1 in
    let type2 = snd tuple2 in


    let resulttype = match o with
    | Add | Sub | Mult | Div -> type1
    | Equal| Neq | Less | Leq | Greater| Geq ->ActingType(Bool) in
    let str1 = rtype_to_str type1 in
    let str2 = rtype_to_str type2 in
    let env = fst tuple2 in

    if String.compare str1 str2 != 0  then
        let error_type = BinopTypeMismatch (type1, o, type2) in
        let new_error = ( error_locus, error_type) in
        let updated_env = { env with errors = new_error :: env.
            errors } in
        (updated_env, type1)
    else
        (env, resulttype)

| Assign(id, val_expr)
    ->
    (* check expr, then get its type *)
    let tuple1 = check_expression env func error_locus val_expr in
    let updated_env = fst tuple1 in
    (* check id, then get its type *)
    let tuple2 = check_id_usage updated_env expr error_locus id in
    (* check that the types are the same *)
    let type1 = snd tuple1 in
    let type2 = snd tuple2 in

    let str1 = rtype_to_str type1 in
    let str2 = rtype_to_str type2 in

    if String.compare str1 str2 == 0  then
        (fst tuple2, type1)
    else
```

```
        (* it's this order because type TWO comes from the id *)
        let error_type = AssignmentError(type2, type1) in
        let new_error = ( error_locus, error_type) in
        let updated_env = { env with errors = new_error :: env.errors } in
        (updated_env, type1)





| Access(id, expr) -> (env, Void ) (*of identifier * expr    (* array access *) *)
| Id(id) ->
    check_id_usage env expr error_locus id
| _ -> (env, Void)


(* checks a given statement. returns env with possible errors *)
let rec check_statement env func error_locus statement = match statement with
    (* Nothing happens if it's an empty block *)
    | Block ([]) -> env

    (* Handle head, then handle the tail *)
    | Block (hd :: tl) ->
        let head_env = check_statement env func error_locus hd in
        let the_rest = Block(tl) in
        check_statement head_env func error_locus the_rest

    (* we do not care about the type *)
    | Expr (e) ->
        let out_tuple = check_expression env func error_locus e in
        fst out_tuple

    (* We care that return matches up with the func declaration *)
    | Return (e) ->

    env

    (* We care that e is a boolean, and then check statements *)
    | If (e, stmt1, stmt2) ->
        let tuple = check_expression env func error_locus e in
        let environment = match (snd tuple) with
                | Void ->
                    let new_error = ( error_locus,
                        StatementTypeMismatch(ActingType(Bool),
                        Void, "a if statement") ) in
                    { env with errors = new_error :: env.errors }
                | ActingType t -> match t with
                    | Bool-> env
                    | _ ->
                        let actualtype = ActingType(t) in
                        let new_error = ( error_locus,
                            StatementTypeMismatch(ActingType(Bool),
                                actualtype, "a if statement") ) in
```

```
                            { env with errors = new_error :: env.errors } in
        let env2 = check_statement environment func error_locus stmt1  in
        check_statement env2 func error_locus stmt2


    | For (exp1, exp2, exp3, s) ->
        (* For for loops, we honestly couldn't care about the
        expression types, they can do stupid things in it like C permits you to *)
        let e1 = fst (check_expression env func error_locus exp1) in
        let e2 = fst (check_expression e1 func error_locus exp2) in
        let e3 = fst (check_expression e2 func error_locus exp3) in
        check_statement e3 func error_locus s
    | While (e, s) ->
        (* again, caring that our expression is a boolean *)
        let tuple = check_expression env func error_locus e in
        let environment = match (snd tuple) with
                | Void ->
                    let new_error = ( error_locus,
                        StatementTypeMismatch(ActingType(Bool),
                        Void, "a while statement") ) in
                    { env with errors = new_error :: env.errors }
                | ActingType t -> match t with
                    | Bool-> env
                    | _ ->
                        let actualtype = ActingType(t) in
                        let new_error = ( error_locus,
                            StatementTypeMismatch(ActingType(Bool),
                                actualtype, "a while statement") ) in
                            { env with errors = new_error :: env.errors }
        in check_statement environment func error_locus s
    | _ -> env

(* checks a function, updates environment *)
let check_function env possible_error_locus func =

    (* A variable adde and error-maker *)
    let f env current_vdecl =
        add_var_decl env possible_error_locus current_vdecl in

    (* 0: add formals BEFORE the variables, so that variables come into
        conflict with these formals already declared! *)
    (* note: sweet, I could completely reuse the above function *)
    let env = List.fold_left f env func.formals in

    (* 1. add variables *)
    let env = List.fold_left f env func.locals in

    (* 2. go through each statement, checking the types *)
    let f env current_statement =
        check_statement env func possible_error_locus current_statement in
```

```

    List.fold_left f env func.body







let main_checker ast_head =


    let basic_env = make_basic_env in

    (*
        ///////////////////////////////////////////////////////////////

        first, verify that no entities have been duplicated *)
    let verified_duplicate_entities =
        let f env e =
           (* Add entity to our environment, check for duplicates *)
            let name = e.ename in
            let entities = env.entities in
            let found = StringMap.mem name entities in
            if found then
                (* error message, because there shouldn't be another
                     with same name *)
                let new_error = (
                        Global,
                        EntityRepeatDecl(e))
                    in
                { env with errors = new_error :: env.errors }
            else
                let updated_entities = StringMap.add name e entities
                     in
                    { env with entities = updated_entities; } in
        List.fold_left f basic_env ast_head
    in
    (*
        ///////////////////////////////////////////////////////////////

        next go entity by entity to 1. check repeat function decls
                                and 2. handle each function *)

    let do_each_entity env entity =

        (* now for each entity... *)

        (* The part that sees if we have duplicate functions *)
        let verify_entity_functions env function_list =
            let map = StringMap.empty in
            let aux result f_decl =
                (* we're passing a tuple around with both the
                    updated environment
                    and a map that acts as a set for whether we have
                         a function already *)
                let e = fst result and m = snd result in
```

```
                let search =
                    try  (function a -> true) (StringMap.find f_decl.fname m )
                    with Not_found -> false in
                if search then
                    (* error message, because there shouldn't be another with same name *)
                    let new_error = (
                            Entity(entity.ename),
                            FunctionRepeatDecl(f_decl))
                    in
                    ( { e with errors = new_error :: e.errors }, m)
                else
                    ( e, (StringMap.add f_decl.fname f_decl m)) in
                let out = List.fold_left aux (env, map) function_list in
                fst out in

        let env_after_verifying_functions = verify_entity_functions env entity.methods in


        (* The part that sees if we have duplicate variables *)
        let verify_entity_variables env locals =
            let error_locus = Entity(entity.ename) in
            let cleaned_env = clear_variable_scope env 0 in
            let f env current_vdecl =
                (* let add_var_decl env possible_error_locus var_decl *)
                add_var_decl env error_locus current_vdecl in
            List.fold_left f env locals in

        (* NOTE: at this point, we also have the variables registered
            in the scope 0 of the environment!! *)
        let env_verified_vars = verify_entity_variables env_after_verifying_functions entity.fields in

        (* Finally, delve into each function and check things over *)

        let check_function_aux curr_env curr_fdecl =

            (* Do not forget - the contents are all SCOPE #1 *)
            let revised_env =
                (* use our aux, but set current_scope manually!! *)
                let cleared = clear_variable_scope curr_env 1 in
                { cleared with current_scope = 1; } in

            (* Locus depends on entity and function so... *)
            let possible_error_locus = EntitysFunction(entity.ename, curr_fdecl.fname) in

            (* Now we check functions *)
            check_function revised_env possible_error_locus curr_fdecl in

```

```
        List.fold_left check_function_aux env_verified_vars entity.methods in

    (* Right this is where we apply that massive aux function to
    every entity there is *)
    List.fold_left do_each_entity verified_duplicate_entities ast_head


let semantic_check unchecked_program =
    (* check if checking_environment says there are any errors *)
    let checked_environment = main_checker unchecked_program in


    (* Spits out all the errors *)
    let handler list_so_far next_error =
        let error_string = String.concat " " (describe_error next_error) in
        let with_nl = String.concat "" [error_string; "\n";] in
        list_so_far @ [ with_nl; ]
        in

    (* we list.rev the errors because errors are always appended left,
        thus they are backwards compared to the order in which they came *)
    let my_errors = List.fold_left handler [] (List.rev checked_environment.errors) in
    let result = String.concat "" my_errors in

    if List.length checked_environment.errors == 0 then
        ""
    else
        result (* We return a string from semantic; if empty, no errors *)
```

```
(* Signed off: Akira *)
let stubs_ctor = ["start"; "stop"]
let stubs_action = ["step"; "render"]
let stubs_helper = ["spawn"; "destroy"]

let gen_spawn ename =
  ename ^ "* " ^ ename ^ "_spawn(){\n     " ^
    ename ^ " *data = malloc(sizeof(" ^ ename ^ "));
    entity_node *node = malloc(sizeof(entity_node));
    if(!data || !node) _seam_fatal(\"Allocation error!\");

    node->step = &" ^ ename ^ "_step;
    node->render = &" ^ ename ^ "_render;
    node->data = data;
    node->next = NULL;

    entity_node *curr = ehead;
    while(curr && curr->next) curr = curr->next;

    if(curr)
        curr->next = node;
    else
```

```
        ehead = node;

    " ^ ename ^ "_start(data);
    return data;
}"

let gen_destroy ename =
  "void " ^ ename ^ "_destroy(" ^ ename ^ " *this){\n     " ^
    ename ^ "_stop(this);

    entity_node *curr = ehead;
    entity_node *prev = NULL;
    while(curr) {
        if(curr->data == this) break;
        prev = curr;
        curr = curr->next;
    }

    if(prev)
        prev->next = curr->next;
    else
    ehead = curr->next;

    free(this);
    free(curr);
}"
```