Slang: A Discrete Event Simulation Language

Olivia Byer, Mauricio Castaneda, Josh Itwaru, Dina Lamdany, Tony Ling

Contents

1 Introduction 4
  1.1 Motivation ......................................................... 4
  1.2 Overview .......................................................... 4
  1.3 Goals ............................................................... 4

2 Language Tutorial 5
  2.1 Structure of a Program ............................................ 5
  2.2 Built In Functions .................................................. 5
  2.3 Hello World .......................................................... 5
  2.4 Compiling and Running a Slang Program ......................... 6
  2.5 Getting the Most out of Slang .................................... 6

3 Language Reference Manual 9
  3.1 Lexical Elements .................................................... 9
    3.1.1 Identifiers ...................................................... 9
    3.1.2 Keywords ...................................................... 9
    3.1.3 Punctuation ................................................... 9
    3.1.4 Escape Sequences .............................................. 9
    3.1.5 White Space ................................................... 10
  3.2 Data Types .......................................................... 10
    3.2.1 int ............................................................. 10
    3.2.2 float ........................................................... 10
    3.2.3 boolean ......................................................... 10
    3.2.4 String .......................................................... 10
    3.2.5 Array .......................................................... 10
    3.2.6 Declarations .................................................. 11
  3.3 Operators ............................................................ 11
    3.3.1 Arithmetic ...................................................... 11
    3.3.2 Assignment ..................................................... 11
    3.3.3 Comparison ..................................................... 11
3.3.4 Logical ......................................... 12
3.3.5 Precedence and Associativity .................. 12
3.4 Type Casting ...................................... 12
3.5 Statements ........................................ 13
3.6 Blocks ............................................ 13
3.7 The return Statement .............................. 13
3.8 The if else Statement ............................. 13
3.9 Loops ............................................. 14
  3.9.1 for loop ..................................... 14
  3.9.2 while loop .................................. 15
3.10 Terminate ........................................ 15
3.11 Functions ....................................... 15
3.12 Function Declaration ............................ 15
3.13 Pass by Value and Pass by Reference .......... 16
3.14 Threads .......................................... 16
3.15 Delays ........................................... 17
3.16 Overloading ...................................... 17
3.17 Program Structure and Scoping ................. 17
3.18 Definition of a Program ......................... 17
3.19 Program Execution ............................... 17
3.20 Scoping .......................................... 18
3.21 Grammar ........................................ 18

4 Project Planning .................................. 21
  4.1 The Planning and Development Process ........ 21
    4.1.1 Planning .................................. 21
    4.1.2 Specification .............................. 21
    4.1.3 Development .............................. 21
    4.1.4 Testing .................................. 21
  4.2 Programming Style Guide ....................... 22
  4.3 Project Timeline ................................ 22
  4.4 Team Roles and Responsibilities ............... 22
  4.5 Software Development Environment ............. 22
  4.6 Project Log .................................. 23

5 Architectural Design ............................. 38
  5.1 Scanner ........................................ 38
  5.2 Parser ......................................... 39
  5.3 Semantic Checking ............................. 39
  5.4 Intermediate Representation ................... 39
  5.5 Code Generation ............................... 39
5.6 Compiler .................................................. 40
5.7 Authorship of the Components .......................... 40

6 Test Plan .................................................. 41
6.1 Development of a Test Plan .............................. 41
6.2 Test Suites ................................................. 41
6.3 Component Testing ........................................... 42
6.4 Automation .................................................. 45
6.5 Slang to C++ .................................................. 45
   6.5.1 JK Flip Flop Simulation ................................. 45
   6.5.2 Bank Queueing .......................................... 62

7 Lessons Learned ............................................. 77
7.1 Olivia Byer .................................................. 77
7.2 Mauricio Castaneda ......................................... 77
7.3 Josh Itwaru .................................................... 77
7.4 Dina Lamdany ............................................... 78
7.5 Tony Ling ..................................................... 79

8 Appendix ..................................................... 79
8.1 scanner.mll .................................................. 79
8.2 parser.mly .................................................. 81
8.3 type.mli ..................................................... 85
8.4 ast.mli ....................................................... 85
8.5 semantic_check.ml ........................................... 87
8.6 sast.mli ...................................................... 105
8.7 pretty_c_gen.ml ............................................. 107
8.8 gen_cpp.ml .................................................. 112
8.9 compiler.ml .................................................. 123
1 Introduction

Slang is a discrete event simulation language. It allows for a programmer to schedule events at discrete times, and have those events executed based on an event queue ordered by start time. Slang compiles into C++ code, which can then be compiled into a binary using the g++ compiler.

1.1 Motivation

Slang was conceived of in order to represent simulations programatically. Simulations are a valuable tool in many disciplines, including statistics, physics, and operations research. A language which uses intuitive syntax and allows for events to be scheduled either as one-time or recurring serves to simplify this simulation process.

1.2 Overview

There are two primary components of a Slang program - those that deal with time and those that deal with actions. As far as time, there are two types of threads that place events on the queue. Blocks labeled as "init" execute once upon program start-up, scheduling once all of the events contained within the block. Blocks labeled as "always" schedule events continuously in a loop, and these events occur repeatedly until program termination. The programmer also has delay statements at their disposal to move program time forward. For the events themselves, the user has user-defined functions at their disposal as well as any code they choose to write within the init and always threads.

1.3 Goals

Slang was designed to be intuitive and broadly applicable to a variety of situations.

Intuitive As it currently stands, many languages with which people are familiar make the scheduling and simulation process extremely complicated. The programmer is tasked with keeping track of program time and creating an event queue on which to place events. Slang provides an intuitive structure with the queue and clock already set up. However, while the program structure and execution is different, we kept the syntax similar to the widely familiar C and Java syntax. This allows for a short learning curve in order to use Slang.

Broadly Applicable Slang is applicable to a wide variety of situations and disciplines. It is certainly useful for the simulation problems mentioned above in the Motivation section. Additionally, a programmer could represent many other algorithms with Slang (either as stand alone programs or for use in simulations), such as the Fibonacci example in Section 6.
2 Language Tutorial

2.1 Structure of a Program

A slang program is a single file consisting of functions, defined and written above the main function, and a mandatory main function. The main block consists of zero or more init and/or always threads. An init thread consists of statements that are to be executed sequentially from time 0. An always thread consists of code to be continuously run until program termination. These time blocks can exist only within the main function, not within any other function. The program ends when the Terminate keyword is used or, in the absence of any always threads, upon the termination of code in the init threads. While Terminate is optional, it is necessary to prevent the continuous looping of always threads.

2.2 Built In Functions

Slang comes with two built-in functions. The programmer has a print function available to them as follows:

```
print(5);
```

where the function can take a number or string argument. Additionally the print_time function can be used as follows:

```
print_time();
```

which prints the current time on the program's internal clock.

2.3 Hello World

Below is an example of the Hello World program in Slang:

```
func void helloworld()
{
    print("hello world");
}
main()
{
    init
    {
        print("Welcome to the demo");
        #2
        Terminate;
    }
    always
    {
```
This simple example demonstrates the use of init and always threads, as well as time delays. Upon start up, the program would schedule the printing “Welcome to the demo” at time 0 using Slang’s built-in print function. The init thread also states that the program will terminate at time 2. The always thread advances time to by 1 unit and then schedules a call to the helloworld() function. Therefore, the function call is scheduled at time 1. The Terminate statement prevents any future function calls past time 2. So helloworld() is called once and the program has the following output:

Welcome to the demo
hello world

2.4 Compiling and Running a Slang Program

Begin by writing a .sl file in your preferred text editor. Run the following command:

$ ./compiler < [path to your .sl file]

This will compile your code and produce a file titled output.cpp. Compile the C++ file with:

$ g++ output.cpp

and finally run your file using:

$ ./a.out

2.5 Getting the Most out of Slang

Consider the following two solutions to the question of finding a number at a specified index in the fibonacci sequence. The fibonacci sequence is the sequence of numbers in which each number is the sum of the two preceding numbers as follows: {1, 1, 2, 3, 5, 8, 13, 21, 34, 55...}. The following program uses a function to calculate the seventh number in the sequence:

```c
func int fib(int n)
{
    if (n==0)
    {
        return 0;
    }
```
if (n==1)
{
    return 1;
}
int prevPrev=0;
int prev=1;
int result=0;
int i=2;
for (i=2; i<=n; i++)
{
    result=prev+prevPrev;
    prevPrev=prev;
    prev=result;
}
return result;

main()
{
    init
    {
        #1
        int fib=fib(7);
        print(fib);
    }
}

This code produces the expected output of 13, the seventh number in the sequence. However, it does not best utilize Slang’s features. The calculation of the entire sequence is scheduled as one event at time 1, even though the process is repetitive and could be scheduled as sequential events. The following example utilizes always threads in order to streamline and simplify the calculations:

main()
{
    int prevPrev=0;
    int prev=1;
    int result=0;
    init
    {
        #7 print(result);
        Terminate;
    }
}
always
{
   #1
   result=prev+prevPrev;
   prevPrev=prev;
   prev=result;
}

This version still prints 13. However, it uses an always thread to loop the calculation of the sequence, terminating at time 7 as per the init thread. This ability to schedule a continuous action until a designated time is an example of best utilizing Slang’s functionality.
3 Language Reference Manual

3.1 Lexical Elements

3.1.1 Identifiers

An identifier is used to refer to a variable in declarations, assignments, and general statements. An identifier must start with a character letter and can contain any combination of letters, numbers, and the underscore symbol '_'. Whitespace signals the end of the identifier.

```plaintext
int decl_ident; /* declaration */
int ident_1=5; /* assignment */
ident_1+2==7; /* general statement, evaluates to true */
```

3.1.2 Keywords

The keywords in Slang are:

```plaintext
if else for while return int float bool string object void func main init always true false Terminate
```

These words have special meanings and are reserved. Therefore, the programmer may not use them as an identifier.

3.1.3 Punctuation

**Parenthesis** Expressions can include expressions inside parenthesis. Parenthesis can also indicate a function call.

**Braces** Braces indicate a block of statements.

**Semicolon** Semicolons are used at the end of every statement as a terminator. Additionally, they are used to separate the statements and expression in a for loop.

3.1.4 Escape Sequences

Certain characters within strings need to be preceded with a backslash ". These characters (and the sequences to produce them in a string) are:
3.1.5 White Space

Slang is whitespace-ambivalent, so there whitespace does not affect the program.

3.2 Data Types

The datatypes available in Slang are int, float, boolean, and string. Additionally, the aggregate data type of array is available to the programmer.

3.2.1 int

An int is a 32-bit signed integer.

3.2.2 float

A float is a 64-big signed floating-point number.

3.2.3 boolean

A boolean value is defined using the keywords true or false.

3.2.4 String

A string is a sequence of characters. String literals are placed between double quotes.

3.2.5 Array

In Slang, you can have arrays of any of the above data types. Arrays are dynamically sized and can be accessed either with an integer such as:

```plaintext
myArray[0];
```

or with a variable such as:

```plaintext
int i = 0;
myArray[i];
```
3.2.6 Declarations

Declaring a Variable Variables can be defined within the main function, within individual functions, or within an init or always thread. Variables may be declared and then defined, or declared and defined at the same time:

```plaintext
type variable_name;
variable_name = 5;
```

Declaring an Array Arrays can be declared on their own or filled with values upon declaration:

```plaintext
type name_of_array[];
name_of_array[] = [5, 6, 7];
```

3.3 Operators

3.3.1 Arithmetic

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition and string concatenation</td>
</tr>
<tr>
<td>-</td>
<td>subtraction and unary negation</td>
</tr>
<tr>
<td>++</td>
<td>unary increment by one</td>
</tr>
<tr>
<td>--</td>
<td>unary decrement by one</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
</tr>
</tbody>
</table>

3.3.2 Assignment

```
= assigns value or right hand side to the left hand side
```

Assignment has right to left precedence.

3.3.3 Comparison

```
== equal to
!= not equal to
> greater than
< less than
>= greater than or equal to
<= less than or equal to
```
3.3.4 Logical

<table>
<thead>
<tr>
<th>Operator</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>unary not</td>
</tr>
<tr>
<td>&amp;</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.3.5 Precedence and Associativity

Operators within the same row share the same precedence. Higher rows indicate higher precedence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>-, + , - (unary minus)</td>
<td>right</td>
</tr>
<tr>
<td>!</td>
<td>right</td>
</tr>
<tr>
<td>* , / , %</td>
<td>left</td>
</tr>
<tr>
<td>+ , -</td>
<td>left</td>
</tr>
<tr>
<td>&amp; , —</td>
<td>left</td>
</tr>
<tr>
<td>i ; , i= , i&lt; , i&gt;</td>
<td>left</td>
</tr>
<tr>
<td>== , !=</td>
<td>left</td>
</tr>
<tr>
<td>=</td>
<td>right</td>
</tr>
</tbody>
</table>

3.4 Type Casting

Casting to Numbers  Casting a float to an integer truncates the fraction part, while casting an integer to a float appends a decimal portion of "0". Casting a boolean to an int or a float results in a 1 (or 1.0) for true and 0 (or 0.0) for false.

```plaintext
#include <iostream>

int main()
{
    float a = 42.3;

    int b = int(a); /* b = 42 */
    a = float(b); /* a = 42.0 */

    a = float(true); /* a = 1.0 */
    b = int(false); /* b = 0 */

    return 0;
}
```

Casting to Strings and Booleans  Cast any number greater than 0 or a string to a boolean leads to a value of true while 0 leads to a value of false.

```plaintext
bool d = false;
    d = bool(20); /* d = true */
    d = bool("false"); /* d = true */
    d = bool("Mike"); /* d = true */
    d = bool(" "); /* d = false */
```
3.5 Statements

Statements are used to get the program to do something. Statements are used for variable declaration and assignment, returns, control flow, loops, function calls, and expressions. All statements end with a semicolon ";". Statements are used within blocks. All of the following are examples of statements:

```plaintext
string name="Pete"; /* variable declaration and assignment */
return 0; /* return statement */
if(5<6) /* control flow */
foo(5); /* function call */
while(name!="Pete") /* loops */
4+4; /* an expression */
```

3.6 Blocks

A block is defined inside curly braces, which can include a possibly-empty list of variable declarations and statements. A block is structured as:

```plaintext
{
    statements
}
```

3.7 The return Statement

The return keyword accepts an expression, function call, or nothing, and exists out of the smallest containing function or calling block.

3.8 The if else Statement

Slang accepts an if statement containing a boolean expression followed by a statement block:

```plaintext
if(3==3)
{
    statements
}
```

Additionally, Slang accepts an if else statement where the if statement contains a boolean expression and the else contains statements to execute if the if statement evaluates to false. The else keyword is tied to the nearest previous if:

```plaintext
if (5>10)
{
```
if(4>5)
{
    statements
}
else
{
    statements
}
}

In the above example, the else is tied to the innermost preceding if, in this case the one which checks if 4>5.

### 3.9 Loops

Slang has for and while loops.

#### 3.9.1 for loop

For loops contain two statements and an expression of the form starting assignment; boolean loop condition; assignment for advancing to the next iteration. That statement is then followed by a statement block as follows:

```plaintext
for (i=0; i<10; i++)
{
    statements
}
```

Slang does not allow empty infinite for loops such as:

```plaintext
for (;;)
{
    statements
}
```

Additionally, the int i must be declared outside the for loop starting assignment, making the following loop invalid:

```plaintext
for (int i=0;i<10;i++)
{
    statements
}
```
3.9.2 while loop

A while loop is of the form:

```cpp
while ( condition )
{
    statements
}
```

The condition of the while loop may not be empty.

3.10 Terminate

When the Terminate keyword statement is found within the program, program execution ends:

```cpp
main()
{
ininit
{
    int a=5;
}
always
{
    Terminate;
    a=2; /*program ends before this statement is executed*/
}
}
```

3.11 Functions

Slang allows for user-defined functions declared above the main function.

3.12 Function Declaration

Functions are of the form:

```cpp
func returntype func_name(type var1, type var2, ...)
{
    function body
}
```
Functions are defined only by identifying the block of code with the keyword func, giving the function a name, supplying zero or more typed parameters, and defining a function body. Function return types are of any data type, or void for no value. An example is:

```plaintext
func int foo(int count)
{
    return count + 1;
}
```

Slang does not allow for nested functions.

### 3.13 Pass by Value and Pass by Reference

Slang passes arguments by value. The argument sent to a function is in fact a copy of the original argument sent to the function. In this way the function cannot modify the argument originally sent to it. The only exception to this is arrays, which are passed by reference.

### 3.14 Threads

A slang program contains threads within the main function, specified by the init and always keywords. An init thread is:

```plaintext
init
{
    statements
}
```

And an always thread is:

```plaintext
always
{
    statements
}
```

An init thread of code is executed a single time at the beginning of the program, setting up any conditions necessary for execution. An always thread executes once per time unit, looping consistently until the program terminates. Always threads run as separate threads, and is is therefore possible to run multiple always threads concurrently. Multiple init threads are allowed, and are executed sequentially. It is possible to have empty threads containing no statements. Additionally, it is possible to have an always thread without an init thread. However, be mindful that without a Terminate statement such a program would loop continuously.
3.15 Delays

In Slang, you can delay a block of code for a designated number of simulation time units with the statement \#time, where time is an integer. This will delay only the current init or always thread, and the other blocks will execute and schedule as they would have before. Delays add up within a block as follows:

```slang
main()
{
    init
    {
        #2 print(2);
        #12 Terminate; /*program would terminate
           at simulation time 20*/
    }
}
```

Delay statements must exist within an init or always thread. Additionally, a delay may not exist within a control flow statement or a loop.

3.16 Overloading

Slang does not allow overloading of functions or of identifiers (there can be no int x at the same time as float x, for example).

3.17 Program Structure and Scoping

3.18 Definition of a Program

A program is a single file consisting of functions, defined and written above the main function, and a main function. The main statement block consists of zero or more init and/or always threads. An init thread consists of statements that are to be executed sequentially from time 0. An always thread consists of code to be continually run until program termination. init and always threads can only exist within the main function, not within any other function. The program ends with the Terminate keyword is used.

3.19 Program Execution

The simulator processes events in the event queue by removing all of the events for the current time off of the queue and processing them. During the processing, more events may be created (such as by functions) and placed in the proper place in the queue for later processing. The simulator does not advance time on its own, requiring a delay of \#1 to move time forward by 1 unit for instance. Because threads can wake and access variables at the same time, there are race condition concerns. In the case of race conditions, Slang does
not guarantee a particular behavior. The order in which threads execute events scheduled for the same time is undefined. Therefore, a Terminate statement at time 20 may occur before or after any of the other events that are scheduled for time 20.

### 3.20 Scoping

Slang uses block scoping, such that something that is defined within a block is defined within the current block and all sub-blocks following the line within which it is defined. Slang does not use global variables. Rather, variables defined within main above the init and always threads are visible within all threads in the main. These variables are not global to functions however, and must be passed in as parameters.

### 3.21 Grammar

Below is the grammar for Slang. Words in all capital letters are tokens passed in from the lexer.

```
program :
    main
    | fdecl program
main :
    MAIN( ) { vdecl_list timeblock_list } 
var_type :
    INT
    | FLOAT
    | BOOLEAN
    | STRING
ret_type :
    var_type
    | VOID
    | var_type []
timeblock_list :
    /* nothing */
    | timeblock_list timeblock
timeblock :
    INIT { events }
    | ALWAYS { events }
fdecl :
    FUNC ret_type ID ( formals_opt ) { stmt_list }
formals_opt :
    /* nothing */
    | formal_list
```
formal_list:
  param
  [ formal_list , param ]
param:
  var_type ID
  [ var_type ID [] ]
event:
  DELAY INT_LITERAL stmt_list
events:
  stmt_list
  [ stmt_list event_list ]
event_list:
  event
  [ event event_list ]
stmt_list:
  /* nothing */
  [ stmt_list stmt ]
vdecl_list:
  /* nothing */
  [ vdecl ; vdecl_list ]
vdecl:
  var_type ID
  [ var_type ID = expr ]
  [ var_type ID [] ]
  [ var_type ID [] = { expr_list } ]
expr_list:
  /* nothing */
  [ expr , expr_list ]
  [ expr ]
stmt:
  expr
  [ TERMINATE ]
  [ RETURN expr ]
  [ { stmt_list } ]
  [ IF ( expr ) stmt ]
  [ IF ( expr ) stmt else stmt ]
  [ FOR ( expr_opt ; expr_opt ; expr_opt ) stmt ]
  [ WHILE ( expr ) stmt ]
  [ vdecl ]
  [ ID [ INT_LITERAL ] = expr ]
  [ ID [ ID ] = expr ]
ID = [ expr_list ] ;
ID = expr ;

expr_opt:
    /* nothing */
    | ID = expr
    | expr

expr:
    INT_LITERAL
    | FLOAT_LITERAL
    | STRING_LITERAL
    | BOOL_LITERAL
    | var_type(expr)
    | ID
    | expr + expr
    | expr - expr
    | expr * expr
    | expr / expr
    | expr == expr
    | expr != expr
    | expr < expr
    | expr <= expr
    | expr > expr
    | expr >= expr
    | expr % expr
    | ID [ INT_LITERAL ]
    | ID [ ID ]
    | ( expr )
    | - expr
epxr ++
expr --
    | ! expr
    | expr & expr
    | expr | expr
    | ID ( expr_list )
4 Project Planning

4.1 The Planning and Development Process

4.1.1 Planning

The initial planning of our project was done through in person meetings. Our language was conceived of, flushed out, and reworked through multiple marathon in person sessions, both as a group and with Professor Edwards and the T.A.s. We would set a group deadline for a section of the project, adjusting the deadline as inevitable snags emerged, and setting the deadline for the next section upon completion of a component of the compiler.

4.1.2 Specification

Our language and its specifications was completely overhauled between the proposal and LRM stage. As we began writing the compiler, we realized many specifications that were either unclear or needed to be changed for the sake of consistency. We would communicate these necessary changes to the entire group. Then, the individuals responsible for each section would go back and change the different components of the compiler to adjust for the new specification.

4.1.3 Development

At the beginning of development, each team member would take on a specific portion of the project, work on producing that component, and report to the team with any difficulties or errors they experienced. We met once or twice weekly to discuss bigger picture decisions about the language and redesignate tasks. At times this approach broke down with no one knowing who was delegated to do what, but our delegation and specification of tasks improved throughout the project. As the project came to a close, we adopted the practice of daily "stand up" meetings in order to keep each other appraised of progress and work out any last minute kinks.

4.1.4 Testing

We developed a comprehensive test suite early on in our project. Additionally, for each individual component a test script was written that would print the output of a specific test (ie what the AST returns for a particular file) in order to test each functionality as we added it in. Each member was responsible for running the tests against their specific part and making sure that it was working. We also had an automated test script that tested all of the tests at once, returning either OK or BAD based on if the expected outcome was produced.
4.2 Programming Style Guide

We followed the ocaml style exhibited in Professor Edward’s slides as our primary style guide. While every team member has preferred style tendencies it was agreed upon that if we were editing a file written by another team member, we would follow the style already in that file. Overall, readability and clarity of code was emphasized over brevity. Alignment during pattern matching was also emphasized for the sake of clarity.

4.3 Project Timeline

The following was the approximate timeline originally set for our project:

<table>
<thead>
<tr>
<th>Date</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>9/15/2013</td>
<td>First Team Meeting</td>
</tr>
<tr>
<td>9/25/2013</td>
<td>Language White Paper</td>
</tr>
<tr>
<td>10/15/2013</td>
<td>Scanner, Parser, and AST Complete</td>
</tr>
<tr>
<td>10/20/2013</td>
<td>Language Reference Manual Complete</td>
</tr>
<tr>
<td>11/10/2013</td>
<td>Semantic Checking Complete</td>
</tr>
<tr>
<td>11/30/2013</td>
<td>Pretty C Intermediate Representation Complete</td>
</tr>
<tr>
<td>12/8/2013</td>
<td>Code Generation Complete</td>
</tr>
<tr>
<td>12/8/2013-12/14/2013</td>
<td>Integration Testing</td>
</tr>
</tbody>
</table>

While this was our projected timeline, we quickly learned that compiler development isn’t as linear as we imagined it to be. No portion was ever truly "complete" until the end of integration testing as we discovered oversights or small errors. The timeline was helpful however, as the entirety of the project wasn’t left until the very end, allowing for reevaluation and error checking along the way.

4.4 Team Roles and Responsibilities

<table>
<thead>
<tr>
<th>Team Member</th>
<th>Primary Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Olivia Byer</td>
<td>Semantic Checking, Final Presentation and Report</td>
</tr>
<tr>
<td>Mauricio Castaneda</td>
<td>Pretty C Intermediate Representation</td>
</tr>
<tr>
<td>Josh Itwaru</td>
<td>Semantic Checking</td>
</tr>
<tr>
<td>Dina Lamdany</td>
<td>Team Leader, Scanner, Parser, AST, Test Cases</td>
</tr>
<tr>
<td>Tony Ling</td>
<td>Code Generation, Test Cases and Automation</td>
</tr>
</tbody>
</table>

4.5 Software Development Environment

This project was primarily written in OCaml version 4.0.0. We also utilized C++ as that is the language Slang compiles to. Our test scripts were written using Bash scripting. Additional tools we used were Ocamlex for the Scanner, Ocamlyacc for the Parser, Makefile to compile all of the modules, and git with Github for our version control.
4.6 Project Log

Below is our Project Log for our Git repository:

e5c914f 2013−12−18 | Merge branch 'master' of git://github.com/dinalamdany/slang (HEAD, master) [Livi Byer]
a3b1a4c 2013−12−18 | Two files which deal with multiple always blocks [Livi Byer]
237b95a 2013−12−17 | cleaned up commented code, removed 'exhaustive checking' warnings [Josh Itwaru]
79d2bd4 2013−12−17 | renamed compiler_v3 to compiler [Tony Ling]
caf8ccc 2013−12−17 | renamed compiler_v3 to compiler [Tony Ling]
cbfc554 2013−12−17 | added new func test [Tony Ling]
aba550a 2013−12−17 | checked expr inside ArrElem to make sure type is Int [Josh Itwaru]
bc1be0d 2013−12−17 | removed references to object, changed operators, casting to reflect current program [Tony Ling]
a206c7a 2013−12−17 | added test file, for loop array print [Tony Ling]
e4254ed 2013−12−17 | does not check for uninitialized variables [Tony Ling]
3f19c13 2013−12−17 | Merge branch 'master' of github.com:
dinalamdany/slang [Josh Itwaru]
9ce444f 2013−12−17 | same msg as last commit [Josh Itwaru]
b39f8a6 2013−12−16 | added J/K flip flop implementation as sample program (origin/master, origin/HEAD) [bitnami]
c1f36a9 2013−12−16 | added compiler executable to gitignore [bitnami]
28f2773 2013−12−16 | fixed global decl/func order [Tony Ling]
ac06cd3 2013−12−16 | changes to pretty_c_gen [Tony Ling]
4ed1233 2013−12−16 | changed sast so that SArrElem and SArrElemAssign take sexpr (IntLits or Variables) instead of just ints [Josh Itwaru]
2ba029f 2013−12−16 | check for value of variables existing [Dina Lamdany]
1cd9937 2013−12−16 | Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]
82947b6 2013−12−16 | Updated examples for more similarity [bitnami]
2283510 2013−12−16 | changed gen_cpp for ast changes [Tony Ling]
c8c5f2f 2013−12−16 | modified pretty_c_gen to work with recent ast/sast changes [bitnami]
can assign to variable index values of arrays, and access them too [Dina Lamdany]

fixed func body [Tony Ling]

added output.cpp to gitignore [bitnami]

Added two sample programs [bitnami]

Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]

Update README.md [Tony Ling]

Merge branch 'master' of https://github.com/dinalamdany/slang [Tony Ling]

test_all.sh edited for older version [Tony Ling]

Update README.md [Tony Ling]

not, not twice [Dina Lamdany]

not operator working [Dina Lamdany]

test folder fixed [Tony Ling]

fixed event list [Tony Ling]

added NOT to parser [Tony Ling]

fixed delay error in parser [Tony Ling]

Merge branch 'master' of github.com:

dinalamdany/slang [Josh Itwaru]

added delay tests [Josh Itwaru]

statements, events, and threads should be in correct order [Dina Lamdany]

Update README.md [Tony Ling]

really this time... [Tony Ling]

test files moved to non-testd [Tony Ling]

can return arrays [Dina Lamdany]

Merge branch 'master' of https://github.com:

dinalamdany/slang [Tony Ling]

fixed gen.cpp [Tony Ling]

changed None to env.var_scope.parent in update_variable [Josh Itwaru]

edits [Tony Ling]

test_all.sh changed for reg tests [Tony Ling]

minor editing [Tony Ling]

fixed scoping things, with mixing up locals/globals [Josh Itwaru]

comments [bitnami]
Merge branch 'master' of github.com:
  dinalamdany/slang [Josh Itwaru]
removed multiple decls error with local and
  global var of same name [Josh Itwaru]
Merge branch 'master' of https://github.com/
  dinalamdany/slang [bitnami]
Comments and format [bitnami]
functions are global [Dina Lamdany]
Merge branch 'master' of github.com:
  dinalamdany/slang [Josh Itwaru]
added Function to scope types [Josh Itwaru]
Merge branch 'master' of https://github.com/
  dinalamdany/slang [Tony Ling]
makfile edit [Tony Ling]
Adding gencpp to makfile [Livi Byer]
Merge branch 'master' of https://github.com/
  dinalamdany/slang [bitnami]
added generated files [bitnami]
Reordering the Makefile to actually work [Livi Byer]
Merge branch 'master' of https://github.com/
  dinalamdany/slang [Tony Ling]
test_all.sh modified" [Tony Ling]
removed more useless files [Dina Lamdany]
Merge branch 'master' of https://github.com/
  dinalamdany/slang [Tony Ling]
removed files we don’t even use brahs [Dina
  Lamdany]
working compilerv_3 [Tony Ling]
Merge branch 'master' of https://github.com/
  dinalamdany/slang [Tony Ling]
A makefile that only makes what they need [Livi
  Byer]
func stuff [Josh Itwaru]
Merge branch 'master' of https://github.com/
  dinalamdany/slang [Tony Ling]
gen_cpp working [Tony Ling]
Merge branch 'master' of github.com/
  dinalamdany/slang [Livi Byer]
Print should be working [Livi Byer]
5f49577 2013-12-14 | Merge branch 'master' of https://github.com/dinalamdany/slang [Tony Ling]
c7530ba 2013-12-14 | reversed all lists within main for generated pretty_c [bitnami]
ea68ceb 2013-12-14 | Merge branch 'master' of https://github.com/dinalamdany/slang [Tony Ling]
8e88c40 2013-12-14 | format editing gen.cpp [Tony Ling]
e620e3c 2013-12-14 | correct env for threads [Dina Lamdany]
234cbf9 2013-12-14 | merged the semantic check in and the new sast [Dina Lamdany]
3eeb6d7 2013-12-14 | A better built in print [Livi Byer]
025ed50 2013-12-14 | sexprs have smaller sexprs in them [Dina Lamdany]
2135181 2013-12-14 | Added built in print (hopefully) [Livi Byer]
3f411b3 2013-12-14 | changed pretty_c and gen.cpp to use sstmt [Tony Ling]
1a78a29 2013-12-14 | added constructor to sident + changed semantics check [Tony Ling]
bde8a22 2013-12-14 | rm output.cpp [Tony Ling]
68bfbcd 2013-12-14 | Merge branch 'master' of github.com:dinalamdany/slang [Josh Itwaru]
acf50ca 2013-12-14 | semantic_check returns typed functions with sstmts [Josh Itwaru]
e46737d 2013-12-13 | added v_decl within timeblock support [bitnami]
3178965 2013-12-13 | i was joking before, now scoping is actually in there [Dina Lamdany]
01b32cc 2013-12-13 | i was joking before, now it actually has scope [Dina Lamdany]
96c9d9f 2013-12-13 | idents now have scope with them [Dina Lamdany]
e6cd7b9 2013-12-13 | same as above [Dina Lamdany]
473b7df 2013-12-13 | removed noexpr because we don’t even use them [Dina Lamdany]
dd980ab 2013-12-13 | Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]
b22d387 2013-12-13 | Fixed makefile, added changes to pretty_c_gen [bitnami]
1039a8f 2013-12-13 | minor edits [Tony Ling]
a8ae5a5 2013-12-12 | Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]

df6b825 2013-12-13 | compiler_v3 added [Tony Ling]
e835269 2013-12-12 | comments cleanup [bitnami]
d53e364 2013-12-12 | fixed statement order in generated pretty_c [bitnami]
45dce28 2013-12-12 | Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]
867d992 2013-12-13 | readme because let's be real no objects [Dina Lamdany]
7c77245 2013-12-12 | added new test file [bitnami]
ab93101 2013-12-12 | fixed pretty_c_gen function to return correct type, fixed counters [bitnami]
96945a1 2013-12-12 | Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]
277f26b 2013-12-12 | added support for mutable 'global' lists [bitnami]
c59450f 2013-12-12 | formatting... [bitnami]
251afa9 2013-12-12 | comments... [bitnami]
14a36f5 2013-12-12 | Removed PropertyAssign from asat [bitnami]
1554dddb 2013-12-12 | Merge branch 'master' of https://github.com/dinalamdany/slang [Dina Lamdany]
372cccec 2013-12-12 | removed propertyassign [Dina Lamdany]
73d64ff 2013-12-12 | Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]
37efbea 2013-12-12 | returns a pretty_c, but main 'section' of pretty_c is not completely functional yet [bitnami]
83c5504 2013-12-12 | added constructors to pretty_c [bitnami]
198dec9 2013-12-12 | Fixed block scoping [Livi Byer]
6f7da0c 2013-12-11 | Update gen_cpp.ml [Tony Ling]
298ae76 2013-12-11 | Merge branch 'master' of github.com: dinalamdany/slang [Josh Itwaru]
e05e71c 2013-12-11 | initialization of globals checks for duplicate declarations [Josh Itwaru]
7bbe89b 2013-12-11 | Added pretty_c_gen, targets in makefile and tester (test_pretty_c_gen.sh) [bitnami]
3d2a1c8 2013-12-11 | Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]
78448ea 2013-12-11 | pattern matching [Dina Lamdany]
3beaa0 2013-12-11 | fREaking arrays and shit fixed it [Josh Itwaru]
b7ee336 2013-12-11 | resolved merge conflicts in semantic_check [Josh Itwaru]

27
function return types must be compatible with assignments.

this test should fail, but it doesn't.

need to have void return type.

add functions to env before globals, because can use functions to set global.

made sast_tests compatible without print, removed object tests.

in stmt checking, was originally only returning sast nodes but not updating the env; fixed.

Called initialize_functions.

fixed stupid array None bug gah.


well, arrays are still broken.

all init/always struct objs in the same list.

all init/always struct objs in the same list.

can't have same names of types in ast/sast.

the sast is generated basically with broken things.

modified some more tests.

match all var decls.

ast parses statements in the right order.

tests ast.

error from this but shouldn't be.

correct bino types.

actual correct binary operator matching.

sast_tests directory, some of which can be run as useful tests.

script to test the output of the sast.
1) Statement checking is done. 2) Modified SVarAssignDecl to sdecl and added sval 3) all pattern matching is not exhaustive 4) commented our thread matching — everything else compiles fine [Josh Itwaru]
need to have different names for objects in sast [Dina Lamdany]

need to have different names for objects in sast [Dina Lamdany]

fixed merge issues with makefile [bitnami]

Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]

need to change names of the sdecls [Dina Lamdany]

really no more objects [Dina Lamdany]

comment [Dina Lamdany]

still broken, but a lot more robust checking ? [Dina Lamdany]

need to change names of the sdecls [Dina Lamdany]

added new target for gen_pretty.c [bitnami]

really no more objects [Dina Lamdany]

Merge branch 'semanticcheck_livi' of https://github.com/dinalamdany/slang into livi [Dina Lamdany]

removed object from type [Dina Lamdany]

Merge branch 'livi' of github.com:dinalamdany/slang into sast [Josh Itwaru]

k [Josh Itwaru]

we no longer have objects. AYO [Dina Lamdany]

checking statements almost works [Livi Byer]

sast with types for the events [Dina Lamdany]

Merge branch 'master' of git://github.com/dinalamdany/slang [Livi Byer]

Changing the sstmt in sast [Livi Byer]

updated template [Tony Ling]

Added link list to main [bitnami]

Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]

added 'links' to init blocks in pretty_c [bitnami]

Merge branch 'master' of https://github.com/dinalamdany/slang [Dina Lamdany]

delays can only be a int [Dina Lamdany]
Adding semantic check of functions [Livi Byer]
working on check_funcs; need to write get_env_stmt function [Josh Itwaru]
Merge branch 'master' of github.com: dinalamdany/slang into sast [Josh Itwaru]
check_funcs implemented, not testes [Josh Itwaru]
Removed extra info and added keywords to sast — sorry Mauricio [Livi Byer]
More semantic check functions [Livi Byer]
added prefix tracking in pretty_c [bitnami]
Adding symbol table functions [Livi Byer]
The beginning of a semantic check file [Livi Byer]
correct output for neg and not [Dina Lamdany]
vdecls work again [Dina Lamdany]
now all things reduced, have obj declarations [Dina Lamdany]
Merge branch 'livi' [Livi Byer]
Our SAST!! Go forth and code generate :) [Livi Byer]
Added types into ast, parser, scanner [Livi Byer]
Update pretty_c.mli [Tony Ling]
pretty_c tree [Dina Lamdany]
arrays and objects and primitives are all types [Dina Lamdany]
only vdecls [Dina Lamdany]
edited generated code (origin/tonys_life) [Tony Ling]
edited generated code [Tony Ling]
edited header for generated code [Tony Ling]
example cpp generated code [Tony Ling]
edited helloword.script for compiler_v2 [Tony Ling]
added second version of compiler. Added if, for, variable decl, etc... Still missing init and always, arrays and objects. Added make configuration for this new file. [bitnami]
7b8e74b 2013−11−14 | Added scripts to script folder [Tony Ling]
8cb12f1 2013−11−14 | minor update [Tony Ling]
25209b8 2013−11−14 | file cleaning [Tony Ling]
88f9c0b 2013−11−14 | auto test for helloworld [Tony Ling]
9bc4cb0 2013−11−14 | Merge branch 'master' of https://github.com/dinalamdany/slang [Tony Ling]
06b5594 2013−11−14 | updated compiler_v1 [Tony Ling]
1066706 2013−11−14 | reverting previous commit error [bitnami]
a1272a1 2013−11−13 | tester readded [Tony Ling]
3225277 2013−11−13 | tester readded [Tony Ling]
86f90be 2013−11−13 | changed structure of delays in program [Dina Lamdany]
e906c0e 2013−11−13 | Merge branch 'master' of https://github.com/dinalamdany/slang [bitnami]
6bf017c 2013−11−13 | delays can only be ints [Dina Lamdany]
66eba0b 2013−11−13 | Changed target for scanner/parser testers [bitnami]
37a85e9 2013−11−13 | fixed parser_tester to conform to new changes on the ast and parser [bitnami]
d11516f 2013−11−13 | fixed ast types of variable assign [Dina Lamdany]
3350b3b 2013−11−13 | tests [Dina Lamdany]
1b19685 2013−11−13 | fixed pass by value test [Dina Lamdany]
a64f7a3 2013−11−13 | can declare variables inside for loop now [Dina Lamdany]
da8a38d 2013−11−05 | Added support for parser tests and scanner tests [bitnami]
8aa44bf 2013−11−05 | Added todo in ast.mli, removed token INT in parser.mly, added array type in sast.mli. Also, renamed tester.ml to slang.ml. Added functionality to the slang output file to be the main entry point [bitnami]
c9ed561 2013−11−12 | minor update [Tony Ling]
78f2107 2013−11−12 | outputs to .cpp [Tony Ling]
be7d789 2013−11−12 | compiler version 1 [Tony Ling]
d11cfbc 2013−11−12 | added more delay tests [Dina Lamdany]
f0954a0 2013−11−12 | fixed some tests [Dina Lamdany]
4f500ba 2013−11−12 | object properties should be exprs [Dina Lamdany]
c33a42e 2013−11−12 | fixed if.sl [Dina Lamdany]
87b8ba8 2013−11−12 | array test [Dina Lamdany]

32
2e55a24 2013−11−05 | Added sast.mli file (semantic abstract syntax tree), fixed indentation issues in ast.mli, added sast.mli to the Makefile [bitnami]
4a75ec9 2013−11−12 | changed assignments to statements, from exprs [Dina Lamdany]
fba45ed 2013−11−12 | fixed functions1 [Dina Lamdany]
183a890 2013−11−12 | fixed while [Dina Lamdany]
50bedb9 2013−11−11 | updated tester and script [Tony Ling]
1141b84 2013−11−11 | more tests [Dina Lamdany]
39c9e15 2013−11−11 | modified gitignore [Dina Lamdany]
46b4dd2 2013−11−11 | the makefile only makes codegenloop, not codegen, and works [Dina Lamdany]
9e99302 2013−11−11 | output of tests [Dina Lamdany]
e6bde1e 2013−11−11 | can have expressions lists of just one [Dina Lamdany]
f4a01a5 2013−11−11 | fixed call and function name types in parser and ast [Dina Lamdany]
bc63126 2013−11−11 | changed unit test code for tester [Tony Ling]
cce5d60 2013−11−11 | sh to output results [Tony Ling]
41da52a 2013−11−08 | gen code typo [Tony Ling]
61a52a9 2013−11−08 | updated for function+function formals [Tony Ling]
792fb84 2013−11−08 | String to string, codegenloop updated [Tony Ling]
33a6b06 2013−11−07 | vdecl working version [Tony Ling]
9f0f6fc 2013−11−07 | Merge branch 'master' of git://github.com/dinalamdany/slang [Livi Byer]
414fdec 2013−11−07 | Code generation for a basic function [Livi Byer]
7f54e4c 2013−11−07 | edit makefile [Tony Ling]
561163b 2013−11−07 | Merge branch 'master' of git://github.com/dinalamdany/slang
Adding codegen to make as well as Tony's test code [Livi Byer]
7f3fa95 2013−11−07 | Merge branch 'master' of git://github.com/dinalamdany/slang [Livi Byer]
3bee3e6 2013−11−07 | adding codegen.ml to makefile [Livi Byer]
411bf84 2013−11−07 | Readme update [Tony Ling]
1b66353 2013−11−07 | A code generation file that matches print function [Livi Byer]
98cd728 2013−11−07 | comments [Tony Ling]
f2463f1 2013−11−07 | Added testing files [Tony Ling]
5bb11d3 2013−11−04 | Fixed issues with typos [bitnami]
74bb8a9 2013−11−07 | Adding function call to ast [Olivia Byer]
1c8ca54 2013−11−07 | Fixing merge conflict [Olivia Byer]
6e69a4f 2013−11−07 | Adding function call to ast and parser [Olivia Byer]
850aec6 2013−11−07 | change formal_list to be made of formals instead of decls [Dina Lamdany]
7074216 2013−11−07 | string literals after [Dina Lamdany]
a4f0d53 2013−11−07 | test for types before test for ids [Dina Lamdany]
5f477af 2013−11−07 | First attempt to add printf function [Olivia Byer]
4435e75 2013−11−07 | Tony’s test files [Tony Ling]
bf10d12 2013−10−28 | Update manual.md [Tony Ling]
e3eaf83 2013−10−28 | edit comment in sample program [Josh Itwaru]
24126cd 2013−10−28 | added our names [Dina Lamdany]
b8a15ef 2013−10−28 | fixed line numbers of function calls [Dina Lamdany]
9d13039 2013−10−27 | delays are in same line as the stuff they delay [Dina Lamdany]
2d6836b 2013−10−24 | param to manual [Dina Lamdany]
b699d28 2013−10−24 | grammar code formatting [Tony Ling]
addc2ea 2013−10−24 | formal_list matches formal_list COMMA param [Tony Ling]
c0c7aee 2013−10−24 | Edited sample program [Tony Ling]
2414df5 2013−10−24 | name of array [Dina Lamdany]
09de5e2 2013−10−24 | Added semi, and added grammar to manual [Dina Lamdany]
8c87c51 2013−10−24 | block scoping [Dina Lamdany]
916c1c6 2013−10−24 | precedence for parents and bracs [Dina Lamdany]
e05de43 2013−10−24 | fixed broken formatting of LRM [Tony Ling]
09f9055 2013−10−24 | consolidated objects [Dina Lamdany]
a4385cf 2013−10−23 | race, overloading, and promotion [Dina Lamdany]
4fed872 2013−10−23 | Added test files [bitnami]
b87b240 2013−10−23 | Merge branch 'master' of github.com: dinalamdany/slang [Josh Itwaru]
bfefbb7 2013−10−23 | trivialized program structure example to arithmetic [Josh Itwaru]
variable declarations don't inherently need semis

fixed typo

added program structure, identifiers

Merge branch 'master' of https://github.com/dinalamdany/slang

tests for Comparison section of LRM

test for 'Escape Sequences' section of LRM

corrections

added escape sequences

added typecasting section

moved around things/additionally statement section

updated if-else and object declaration

tests for 'Terminate' section for LRM

corrected wrong formatting

changed formatting

added precedence order table

added precedence order table

Merge branch 'master' of https://github.com/dinalamdany/slang

tests for 'Int' section of LRM

better regex for string literals

tests for 'Semicolon' section of LRM

tests for 'Parenthesis' section of LRM

numbers or _ or characters can all be valid string literals

added in type casting

boolean example
true and false as boolean literals [Dina Lamdany]
added in bool type [Dina Lamdany]
One last try and formatting [Olivia Byer]
Trying to fix formatting [Olivia Byer]
I added to the LRM (specifically threads and Objects). I left my questions written at the bottom. [Olivia Byer]
tests for 'Program' section of LRM [Tony Ling]
test for 'Comments' section of LRM [Tony Ling]
Merge branch 'master' of https://github.com/dinalamdany/slang [Tony Ling]
tests for 'Program' section of LRM [Tony Ling]
removed todos [Dina Lamdany]
added in terminate keyword [Dina Lamdany]
dont have to set object property values when declare, just have to set property [Dina Lamdany]
tests for 'Program' section of LRM [Tony Ling]
readme for tests [Tony Ling]
illegal characters after [Dina Lamdany]
more todos [Dina Lamdany]
to-dos [Dina Lamdany]
readme [Dina Lamdany]
can set properties of objects. temporary syntax, need to decide on one [Dina Lamdany]
can create objects with a list of properties [Dina Lamdany]
added dot to scanner [Dina Lamdany]
can declare and define arrays at same time [Dina Lamdany]
can create arrays and can assign their elements values [Dina Lamdany]
Array declaration [Dina Lamdany]
made program structure require a main, and then consist of init/always blocks [Dina Lamdany]
added in strings, variable types [Dina Lamdany]
574f5bd 2013–10–18 | added in floating point numbers [Dina Lamdany]
65aa3e7 2013–10–15 | added in time to manual [Dina Lamdany]
f8bfccf 2013–10–15 | can do #int or #var for delay now [Dina Lamdany]
4678854 2013–10–15 | added in or, not, and unary minus operators [Dina Lamdany]
6040a2a 2013–10–15 | scoping in manual [Dina Lamdany]
1ffcbdc 2013–10–15 | our program is just a list of functions, not of functions and variable declarations [Dina Lamdany]
51376a2 2013–10–14 | makefile with clean and compile commands for easy working [Dina Lamdany]
8c0fc90 2013–10–14 | updated manual and parser/scanner for functions [Dina Lamdany]
5fd2de3 2013–10–14 | basic ast and parser, just for expressions [Dina Lamdany]
fd0709 2013–10–14 | better markdown [Dina Lamdany]
0bae6f 2013–10–14 | edwards’ scanner, with all parts described in manual, and a first test [Dina Lamdany]
5d8be8e 2013–10–10 | removing sandbox files [Dina Lamdany]
85b450c 2013–10–10 | moving to slang [Dina Lamdany]
89435d6 2013–10–09 | added in proposal ias html generated by gdocs [Dina Lamdany]
c7511bd 2013–10–08 | initial commit [Dina Lamdany]
5 Architectural Design

Below is a visual representation of the architecture of the Slang compiler:

![Diagram of Slang compiler architecture]

5.1 Scanner

The scanner takes in a Slang source file and outputs tokens according to the lexical rules of our language.
5.2 Parser

The parser takes in the tokens from the scanner and uses our grammatical rules to produce an abstract syntax tree.

5.3 Semantic Checking

The semantic checking file takes in the abstract syntax tree a given by the parser and performs semantic checks, outputting a typed semantic abstract syntax tree. This file uses a two symbol tables in order to keep track of global and local variables, checking that the program adheres to our scoping rules. Additionally, it checks that the arguments of operators are valid, that all variables and declared and initialized when they need to be, that assignments have matching types, that two variables are not declared with the same name, that arrays are indexed properly, and that functions return a value (and that the return type is correct). If a file does not adhere to proper semantics the compiler will raise an error and compilation will halt.

5.4 Intermediate Representation

The Pretty C intermediate representation takes in the semantic abstract syntax tree and begins to set up the structure needed for code generation. While more details are available in the following section about the structure of generated code, Pretty C gathers the individual threads and encapsulates them into structures with the time delay, making the code generation simpler.

5.5 Code Generation

Code generation takes the intermediate representation from Pretty C and simulates concurrent events in C++ by creating objects that represent portions of time threads. These objects each contain a function that when called will execute the appropriate portion of the Slang code. The code in each object is equivalent to a section of code in between two delays in the Slang file. There are base classes that the time objects inherit from that link them to a single time block. In order to simulate local variables, all variables are created as global C++ variables with a prefix of the appropriate thread, so all objects derived from the same thread will have access to its variables. Additionally, each time object holds a pointer pointing to the next object representing the next section of the Slang program. Therefore, init thread objects form a linked list, while always thread objects form a circularly linkedlist.

The generated code also implements an event queue. The event queue structure is a priority queue that takes in pointers to the time objects, and orders the objects based off of their time value at which they are supposed to occur. The event queue is on a loop that pops the first element on the queue and calls its function. Every time a time object’s
function is called, it increments its own global time counter, adding the next time object with this new time. It knows the next object due to its pointer.

5.6 Compiler

Our compiler file is the file that puts all of these modules together in succession. It brings in a file from stdin and runs it through the scanner, parser, semantic check, intermediate representation generation, and code generation to produce an output.cpp file that can be compiled using g++.

5.7 Authorship of the Components

While every team member worked on every component in some fashion (either as an author or as a debugger), the primary responsibility breakdown for each component is given above in section 4.4.
6 Test Plan

6.1 Development of a Test Plan

Our test plan consists of a test suite of comprehensive tests meant to check all of the functionalities of a Slang program, multiple scripts to test each portion of our compiler, and an automated test script that runs over all of the test cases and checks them against the expected output files. The test suites and scripts themselves were primarily authored by Mauricio Castaneda, Dina Lamdany, and Tony Ling. However, every team member authored a portion of the tests as they needed to test the components they were working on.

6.2 Test Suites

Our test suite is enumerated in the table below. These test cases were chosen because they represent the smallest building blocks of a Slang program, allowing us to test each part individually to easily identify the location of errors. There are also more complicated tests which allow for testing multiple parts of the program integrated together.
File | Functionality Tested
--- | ---
arithmetic1.sl | Binary addition
arithmetic2.sl | Other binary arithmetic
array1.sl | Declaration and definition of an array
array2.sl | Accessing array elements with a variable
comparison1.sl | Greater than operator
comparison6.sl | Less than or equal to operator
declaration1.sl | Declaration of variables
declaration2.sl | Double declaration of variables - should not allow variable to be declared twice
delay.sl | Delay statements with Terminate
delay2.sl | Delay statements without Terminate
escapeseq1.sl | Escape sequences to print reserved character \n
escapeseq2.sl | Escape sequences to print reserved character ”
escapeseq3.sl | Escape sequences to print reserved character ’
escapeseq4.sl | Escape sequences to print linefeed character
escapeseq6.sl | Escape sequences to print horizontal tabulation character
float.sl | Division of two integers to produce an integer
float2.sl | Division of two floats to produce a float
for2.sl | For loop
functions1.sl | Function declaration
functions3.sl | Calling a function with a return type
functions4.sl | Calling a function from within another function
helloworld.sl | Printing "hello world"
if.sl | If else statements
int1.sl | Use of integers
passedbyValueorReference.sl | Variables are passed by reference other than arrays which are passed by value
program1.sl | Basic program structure with just a main
terminate1.sl | Standalone Terminate statement
terminate2.sl | Terminate statement in init thread
terminate3.sl | Terminate statement in a function
while.sl | While loop

### 6.3 Component Testing

As we wrote each component, we also created test scripts that allow us to see the output of the compiler at a certain point in the compilation process. The scanner and the parser were tested with `test_ast.sh`:

```bash
#!/bin/bash
```
set -e

# This script will parse and scan STDIN into an AST, which it
# then prints.

# Usage:
# cat tests/arithmetic1.sl | ./test_sast.sh
# #
# # It contains many kludges.

#scons -c > /dev/null
#scons > /dev/null

PARSE=''
open Ast;;
open Type;;

let lexbuf = Lexing.from_channel stdin in
Parser.program Scanner.token lexbuf;;

( echo -e "$PARSE; cat -) | ocaml scanner.cmo parser.cmo
  semantic_check.cmo

The semantic checking was tested with test_sast.sh:

#!/bin/bash
set -e

# This script will parse and scan STDIN into an AST, which it
# then prints.
#
# Usage:
# cat tests/arithmetic1.sl | ./test_sast.sh
# #
# # It contains many kludges.

#scons -c > /dev/null
#scons > /dev/null

PARSE=''
open Ast;;
open Type;;
open Sast;;
The Pretty C intermediate representation was tested with test_pretty_c_gen.sh:

```bash
#!/bin/bash
set -e

# This script will parse and scan STDIN into an AST, which it then prints.
#
# Usage:
# cat tests/arithmetic1.sl | ./test_sast.sh
#
# It contains many kludges.

#scons -c > /dev/null
#scons > /dev/null

PARSE='open Ast;;
open Type;;
open Sast;;
open Semantic_check;;
open Pretty_c_gen;;

let lexbuf = Lexing.from_channel stdin in
let program = Parser.program Scanner.token lexbuf in
let sast = Semantic_check.check_program program in
gen_pretty_c sast;;'
( echo -e $PARSE; cat - ) | ocaml scanner.cmo parser.cmo
semantic_check.cmo
```

The code generation did not need a script in the same fashion, as we could read the output.cpp file to understand where a mistake was happening.
6.4 Automation

We use an automated test script for our regression testing. This script ran over each of our test cases, comparing the expected output to the actual output, and printing either OK or BAD:

```
#!/bin/bash
#script used for reg testing
COMPILER="../compiler"
COMPFILE="temp_test"

for TESTFILE in ../tests/*.sl;
do
    echo " TESTING $TESTFILE"
    LEN=$(( (${#TESTFILE} - 3) )
    OUTFILENAME="${TESTFILE:0:$LEN}.output"
    TESTFILENAME="${TESTFILE:0:$LEN}.out"
    "$COMPILER" < "$TESTFILE"
gr++ output.cpp -o "$COMPFILE".
""$COMPFILE" > "$OUTFILENAME"
if (diff "$OUTFILENAME" "$TESTFILENAME")
    then
        echo " OK"
    else
        echo " BAD!"
    fi
    rm "$OUTFILENAME" output.cpp "$COMPFILE"
done
```

6.5 Slang to C++

6.5.1 JK Flip Flop Simulation

The following example simulates a JK Flip Flop:

```
main()
{
    /* Clock */
    bool clk = false;
    /* Clock up */
    bool clk_up = false;
    bool prev_clk = false;
```
/* J/K Flip Flop Variables */
bool j = false;
bool k = false;
bool q = true;

/* Clock */
always{
    #2
    clk = !clk;
}
/* Section for generating clock up events */
always{
    #1
    if (clk & !prev_clk)
        {clk_up = true;}
    else
        {clk_up = false;}
    prev_clk = clk;
}
/* J/K Flip Flop logic */
always{
    #1
    if(clk_up){
        if(j & k){q = !q;}
        else{
            if(j){q = true;}
            if(k){q = false;}
        }
    }
}
/* Printing */
always{
    #1
    print("AbsTime");
    print_time();
    print("Clock, Clk_up");
    print(clk);
    print(clk_up);
    print("J,K");
    print(j);
}
print(k);
print("Q:");
print(q);
}

/* Changes in input */
init{
    #7 k = true;
    #7 j = true;
    #7 k = false;
    #7 j = false;
    #7 j = true; k = true;
    #7 Terminate;
}

A portion of the output of this program is:

AbsTime
Time now: 1
Clock, Clk_up
0
0
J,K
0
0
Q:
1
AbsTime
Time now: 2
Clock, Clk_up
1
1
J,K
0
0
Q:
1
AbsTime
Time now: 3
Clock, Clk_up
1
0
The C++ code that this program compiles to is:

```cpp
#include <iostream>
#include <string>
#include <deque>
#include <vector>
#include <cstdlib>

struct event {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    virtual unsigned int get_time() {}
    virtual unsigned int get_inc_time() {}
    virtual void (set_time)(unsigned int time_)
    virtual std::string get_name() {}
    virtual void foo() {};
};
```
virtual ~event_() {};
}
struct event_q_ {
    bool empty() {return event_q_.empty();}
    unsigned int get_time() {return global_time;}
    event_* pop() {
        event_* front = event_q_.front();
        global_time = front->get_time();
        event_q_.pop_front();
        return front;
    }
    void add(unsigned int time_, event_* obj_) {
        bool eol = true;
        std::deque<event_*>::iterator it;
        if (obj_ == NULL)
            return;
        for (it = event_q_.begin(); it != event_q_.end(); it++) {
            if ((*it)->get_time() > time_)
                if ((*it)->get_time() > time_)
                    event_q_.insert(it, obj_);
                    eol = false;
                    break;
        }
        if (eol)
            event_q_.push_back(obj_);
    }
private:
    unsigned int global_time;
    std::deque<event_*> event_q;
};
event_q_ event_q;

bool u_clk = false;
bool u_clk_up = false;
bool u_prev_clk = false;
bool u_j = false;
bool u_k = false;
bool u_q = true;
unsigned int always_0_time = 0;
struct always_0_link_ : public event_ {
    49
virtual void set_next(always_0_link_ *n) {};
};
std::vector<always_0_link_*> always_0_list;
struct always_0_block_0 : public always_0_link_ {
unsigned int time;
unsigned int inc_time;
std::string name;
always_0_block_0() : inc_time(0), time(0), name("always_0_block_0") {}
unsigned int get_time() {return time;}
unsigned int get_inc_time() {return inc_time;}
void set_time(unsigned int time_) {time = time_;}
std::string get_name() {return name;}
always_0_link_ *next;
void set_next(always_0_link_ *n) {next = n;};
void foo() {
if (next != NULL) {
always_0_time += next->get_inc_time();
next->set_time(always_0_time);
event_q.add(always_0_time, next);
}
}
}
struct always_0_block_1 : public always_0_link_ {
unsigned int time;
unsigned int inc_time;
std::string name;
always_0_block_1() : inc_time(2), time(2), name("always_0_block_1") {}
unsigned int get_time() {return time;}
unsigned int get_inc_time() {return inc_time;}
void set_time(unsigned int time_) {time = time_;}
std::string get_name() {return name;}
always_0_link_ *next;
void set_next(always_0_link_ *n) {next = n;};
void foo() {
u_clk = !u_clk;
if (next != NULL) {
always_0_time += next->get_inc_time();
next->set_time(always_0_time);
event_q.add(always_0_time, next);
}
}

unsigned int always_1_time = 0;
struct always_1_link_ : public event_ {
    virtual void set_next(always_1_link_*n) {};
};
std::vector<always_1_link_*> always_1_list;
struct always_1_block_0 : public always_1_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    always_1_block_0() : inc_time(0), time(0), name("always_1_block_0") {};
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    always_1_link_*next;
    void set_next(always_1_link_*n) {next = n;}
    void foo() {
        if (next != NULL) {
            always_1_time += next->get_inc_time();
            next->set_time(always_1_time);
            event_q.add(always_1_time, next);
        }
    }
};
struct always_1_block_1 : public always_1_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    always_1_block_1() : inc_time(1), time(1), name("always_1_block_1") {};
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
std::string get_name() { return name; }
always_1_link_ *next;
void set_next(always_1_link_ *n) { next = n; };
void foo() {
  if (u_clk && !(u_prev_clk))
  {
    u_clk_up = true;
  }
  else {
    u_clk_up = false;
  }
  u_prev_clk = u_clk;
  if (next != NULL) {
    always_1_time += next->get_inc_time();
    next->set_time(always_1_time);
    event_q.add(always_1_time, next);
  }
}

unsigned int always_2_time = 0;
struct always_2_link_ : public event_ {
  virtual void set_next(always_2_link_ *n) {};
};
std::vector<always_2_link_*> always_2_list;
struct always_2_block_0 : public always_2_link_ {
  unsigned int time;
  unsigned int inc_time;
  std::string name;
  always_2_block_0() : inc_time(0), time(0), name("always_2_block_0") {}
  unsigned int get_time() { return time; }
  unsigned int get_inc_time() { return inc_time; }
  void set_time(unsigned int time_) { time = time_; }
  std::string get_name() { return name; }
  always_2_link_ *next;
  void set_next(always_2_link_ *n) { next = n; };
void foo() {

    if (next != NULL) {
        always_2_time += next->get_inc_time();
        next->set_time(always_2_time);
        event_q.add(always_2_time, next);
    }
}

};

struct always_2_block_1 : public always_2_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    always_2_block_1() : inc_time(1), time(1), name("always_2_block_1") {}
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    always_2_link_ *next;
    void set_next(always_2_link_ *n) {next = n;};
    void foo() {
        if (u_clk_up)
            if (u_j&&u_k)
                u_q = !(u_q);
    
        else {
            if (u_j)
            {
                u_q = true;
            }
        
            else {

            }
        
        if (u_k)

    }

}
{ u_q = false; }
else { }
}
else { }
}
if (next != NULL) {
    always_2_time += next->get_inc_time();
    next->set_time(always_2_time);
    event_q.add(always_2_time, next);
}
}

unsigned int always_3_time = 0;
struct always_3_link_ : public event_ {
    virtual void set_next(always_3_link_ *n) {};
};
std::vector<always_3_link_*> always_3_list;
struct always_3_block_0 : public always_3_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    always_3_block_0() : inc_time(0), time(0), name("always_3_block_0") {};
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    always_3_link_ *next;
void set_next(always_3_link_ *n) {next = n;};
void foo() {

if(next != NULL) {
    always_3_time += next->get_inc_time();
    next->set_time(always_3_time);
    event_q.add(always_3_time, next);
}
}

struct always_3_block_1 : public always_3_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    always_3_block_1() : inc_time(1), time(1), name("always_3_block_1") {}
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    always_3_link_ *next;
    void set_next(always_3_link_ *n) {next = n;};
    void foo() {
        std::cout << ("AbsTime") << std::endl;
        std::cout << "Time now: " << event_q.get_time() << std::endl;
        std::cout << ("Clock, Clk_up") << std::endl;
        std::cout << (u_clk) << std::endl;
        std::cout << (u_clk_up) << std::endl;
        std::cout << ("J,K") << std::endl;
        std::cout << (u_j) << std::endl;
        std::cout << (u_k) << std::endl;
        std::cout << ("Q:") << std::endl;
        std::cout << (u_q) << std::endl;

        if(next != NULL) {
            always_3_time += next->get_inc_time();
            next->set_time(always_3_time);
            event_q.add(always_3_time, next);
        }
    }
}
unsigned int init_0_time = 0;

struct init_0_Link_ : public event {
    virtual void set_next(init_0_Link_ *n) {};
};

std::vector<init_0_Link_*> init_0_list;

struct init_0_block_0 : public init_0_Link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    init_0_block_0() : inc_time(0), time(0), name("init_0_block_0") {}
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    init_0_Link_ *next;
    void set_next(init_0_Link_ *n) {next = n;};
    void foo() {
        if(next != NULL) {
            init_0_time += next->get_inc_time();
            next->set_time(init_0_time);
            event_q.add(init_0_time, next);
        }
    }
};

struct init_0_block_1 : public init_0_Link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    init_0_block_1() : inc_time(7), time(7), name("init_0_block_1") {}}
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    init_0_Link_ *next;
    void set_next(init_0_Link_ *n) {next = n;};
    void foo() {

u_k = true;

if(next != NULL) {
    init_0_time += next->get_inc_time();
    next->set_time(init_0_time);
    event_q.add(init_0_time, next);
}

};
struct init_0_block_2 : public init_0_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    init_0_block_2() : inc_time(7), time(7), name("init_0_block_2") {}
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    init_0_link_ *next;
    void set_next(init_0_link_ *n) {next = n;};
    void foo() {
        u_j = true;

        if(next != NULL) {
            init_0_time += next->get_inc_time();
            next->set_time(init_0_time);
            event_q.add(init_0_time, next);
        }
    }
};
struct init_0_block_3 : public init_0_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    init_0_block_3() : inc_time(7), time(7), name("init_0_block_3") {}
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}

57
init_0_link_ *next;
void set_next(init_0_link_*n) {next = n;}
void foo() {
    u_k = false;

    if (next != NULL) {
        init_0_time += next->get_inc_time();
        next->set_time(init_0_time);
        event_q.add(init_0_time, next);
    }
}

struct init_0_block_4 : public init_0_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    init_0_block_4() : inc_time(7), time(7), name("init_0_block_4") {}
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    init_0_link_ *next;
    void set_next(init_0_link_ *n) {next = n;}
    void foo() {
        u_j = false;

        if (next != NULL) {
            init_0_time += next->get_inc_time();
            next->set_time(init_0_time);
            event_q.add(init_0_time, next);
        }
    }
}

struct init_0_block_5 : public init_0_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    init_0_block_5() : inc_time(7), time(7), name("init_0_block_5") {}
    unsigned int get_time() {return time;}

unsigned int get_inc_time() { return inc_time; }
void set_time(unsigned int time_) { time = time_; }
std::string get_name() { return name; }
init_0_link_ *next;
void set_next(init_0_link_ *n) { next = n; };
void foo() {
  u_j = true;
  u_k = true;
  if (next != NULL) {
    init_0_time += next->get_inc_time();
    next->set_time(init_0_time);
    event_q.add(init_0_time, next);
  }
}

struct init_0_block_6 : public init_0_link_ {
  unsigned int time;
  unsigned int inc_time;
  std::string name;
  init_0_block_6() : inc_time(7), time(7), name("init_0_block_6") {} 
  unsigned int get_time() { return time; }
  unsigned int get_inc_time() { return inc_time; }
  void set_time(unsigned int time_) { time = time_; }
  std::string get_name() { return name; }
  init_0_link_ *next;
  void set_next(init_0_link_ *n) { next = n; };
  void foo() {
    exit(0);
    if (next != NULL) {
      init_0_time += next->get_inc_time();
      next->set_time(init_0_time);
      event_q.add(init_0_time, next);
    }
  }
};

int main() {
  always_0_block_0 always_0_block_0obj;
  59
always_0_list.push_back(&always_0_block_0obj);
always_0_block_1 always_0_block_1obj;
always_0_list.push_back(&always_0_block_1obj);
always_1_block_0 always_1_block_0obj;
always_1_list.push_back(&always_1_block_0obj);
always_1_block_1 always_1_block_1obj;
always_1_list.push_back(&always_1_block_1obj);
always_2_block_0 always_2_block_0obj;
always_2_list.push_back(&always_2_block_0obj);
always_2_block_1 always_2_block_1obj;
always_2_list.push_back(&always_2_block_1obj);
always_3_block_0 always_3_block_0obj;
always_3_list.push_back(&always_3_block_0obj);
always_3_block_1 always_3_block_1obj;
always_3_list.push_back(&always_3_block_1obj);
init_0_block_0 init_0_block_0obj;
init_0_list.push_back(&init_0_block_0obj);
init_0_block_1 init_0_block_1obj;
init_0_list.push_back(&init_0_block_1obj);
init_0_block_2 init_0_block_2obj;
init_0_list.push_back(&init_0_block_2obj);
init_0_block_3 init_0_block_3obj;
init_0_list.push_back(&init_0_block_3obj);
init_0_block_4 init_0_block_4obj;
init_0_list.push_back(&init_0_block_4obj);
init_0_block_5 init_0_block_5obj;
init_0_list.push_back(&init_0_block_5obj);
init_0_block_6 init_0_block_6obj;
init_0_list.push_back(&init_0_block_6obj);
for (int i = 0; i < init_0_list.size(); i++)
{
    if (i != init_0_list.size() - 1)
        init_0_list[i]->set_next(init_0_list[i + 1]);
    else
        init_0_list[i]->set_next(NULL);
}
event_q.add(init_0_block_0obj.get_time(), &
            init_0_block_0obj);
for (int i = 0; i < always_0_list.size(); i++)
{
if (i != always_0_list.size() - 1)
        always_0_list[i]->set_next(always_0_list[i+1]);
else
        always_0_list[i]->set_next(always_0_list[0]);
}
event_q.add(always_0_block_0obj.get_time(), &always_0_block_0obj);
for (int i = 0; i < always_1_list.size(); i++)
{
    if (i != always_1_list.size() - 1)
        always_1_list[i]->set_next(always_1_list[i+1]);
    else
        always_1_list[i]->set_next(always_1_list[0]);
}
event_q.add(always_1_block_0obj.get_time(), &always_1_block_0obj);
for (int i = 0; i < always_2_list.size(); i++)
{
    if (i != always_2_list.size() - 1)
        always_2_list[i]->set_next(always_2_list[i+1]);
    else
        always_2_list[i]->set_next(always_2_list[0]);
}
event_q.add(always_2_block_0obj.get_time(), &always_2_block_0obj);
for (int i = 0; i < always_3_list.size(); i++)
{
    if (i != always_3_list.size() - 1)
        always_3_list[i]->set_next(always_3_list[i+1]);
    else
        always_3_list[i]->set_next(always_3_list[0]);
}
event_q.add(always_3_block_0obj.get_time(), &always_3_block_0obj);
while(!event_q.empty()) {
    event_q.pop()->foo();
}
return 0;
}

6.5.2 Bank Queueing
The following example simulates a line at a bank, where people enter the line, get seen by a teller, and get bored and leave the line at different rates:

func int [] popLine(int line [])
{
    int newline []={0,0,0,0,0,0,0,0,0,0};
    int place=1;
    for (place=1;place<10;place++)
    {
        int newPlace=place-1;
        newline[newPlace]=line[place];
    }
    return newline;
}

func int [] pushLine(int line [], int newEntry)
{
    int newline []={0,0,0,0,0,0,0,0,0,0};
    int place=0;
    for (place=0;place<10;place++)
    {
        newline[place]=line[place];
    }
    bool end=false;
    place=0;
    while ((place<10) && (end==false))
    {
        if (line[place]==0)
            {end=true;}
        else {place=place+1;}
    }
if (end==false) {
    print("No room at end of line");
} else if (end==true) {
    newLine[place]=newEntry;
    return newLine;
}

func int lineSize(int ourArray[], int arraySize) {
    int place=0;
    int count=0;
    while (place<arraySize) {
        if (ourArray[place]!=0) {
            count=count+1;
        }
        place=place+1;
    }
    return count;
}

main() {
    int person1=1;
    int person2=2;
    int person3=3;
    int person4=4;
    int person5=5;
    int person6=6;
    int person7=7;
    int person8=8;
    int person9=9;
    int person10=10;
    int people[] = [person1, person2, person3, person4, person5, person6, person7, person8, person9, person10];
    int peopleSize=10;
    int peopleindex=0;
    int lineindex=0;
    int seenindex=0;
    int boredindex=0;
int line[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int seen[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int bored[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

init
{
    #99
    int lineSize = lineSize(line, peopleSize);
    int seenSize = lineSize(seen, peopleSize);
    int boredSize = lineSize(bored, peopleSize);
    print("People still in line:");
    print(lineSize);
    print("People seen:");
    print(seenSize);
    print("People bored in line:");
    print(boredSize);
}

always
{
    #2
    line = pushLine(line, people[0]);
    people = popLine(people);
}
always
{
    #3
    seen = pushLine(seen, line[0]);
    line = popLine(line);
}
always
{
    #4
    bored = pushLine(bored, line[0]);
    line = popLine(line);
}
The output of this program is:
People still in line:
0
People seen:
6
people bored in line:
4

The C++ code that this program compiles to is:

```cpp
#include <iostream>
#include <string>
#include <deque>
#include <vector>
#include <cstdlib>

struct event {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    virtual unsigned int get_time() {}
    virtual unsigned int get_inc_time() {}
    virtual void (set_time)(unsigned int time) {}
    virtual std::string get_name() {}
    virtual void foo() {}
    virtual ~event() {}
};

struct event_q {
    bool empty() {return event_q.empty();}
    unsigned int get_time() {return global_time;}
    event_* pop() {
        event_* front = event_q.front();
        global_time = front->get_time();
        event_q.pop_front();
        return front;
    }
    void add(unsigned int time_, event_* obj_) {
        bool eol = true;
        std::deque<event_*>::iterator it;
        if (obj_ == NULL)
            return;
        for (it = event_q.begin(); it != event_q.end(); it++) {
```
if ((*it)->get_time() > time_) {
    event_q.insert(it, obj_);
    eol = false;
    break;
}
if (eol)
    event_q.push_back(obj_);
private:
    unsigned int global_time;
    std::deque<event_*> event_q;
};
event_q_ event_q;

std::vector<int> u_popLine(std::vector<int> u_line) {
    const int a_newLine[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    std::vector<int> u_newLine(a_newLine, a_newLine+sizeof(a_newLine)
        /sizeof(a_newLine[0]));
    int u_place = 1;
    for (u_place = 1; u_place < 10; ++(u_place))
    {
        int u_newPlace = u_place - 1;
        u_newLine[u_newPlace] = u_line[u_place];
    }
    return u_newLine;
}
std::vector<int> u_pushLine(std::vector<int> u_line, int
    u_newEntry) {
    const int a_newLine[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    std::vector<int> u_newLine(a_newLine, a_newLine+sizeof(a_newLine)
        /sizeof(a_newLine[0]));
    int u_place = 0;
    for (u_place = 0; u_place < 10; ++(u_place))
    {
u_newLine[u_place] = u_line[u_place];

}
bool u_end = false;
;
 u_place = 0;
while (u_place<10&&u_end==false)
{
    if (u_line[u_place]==0)
    {
        u_end = true;
    }else {
        u_place = u_place+1;
    }
}
;
if (u_end==false)
{
    std::cout << ("No room at end of line") << std::endl;
}
else {

}
if (u_end==true)
{
    u_newLine[u_place] = u_newEntry;
}
else {
}
return u_newLine;
}
int u_lineSize(std::vector<int> u_ourArray, int u_arraySize) {
    int u_place = 0;
    int u_count = 0;
    while (u_place<u_arraySize)
    {
        if (u_ourArray[u_place]!=0)
        {
            u_count = u_count+1;
        }
        else {
        }
        u_place = u_place+1;
    }
    return u_count;
}
int u_person1 = 1;
int u_person2 = 2;
int u_person3 = 3;
int u_person4 = 4;
int u_person5 = 5;
int u_person6 = 6;
int u_person7 = 7;
int u_person8 = 8;
int u_person9 = 9;
int u_person10 = 10;
const int a_people[] = {u_person1, u_person2, u_person3, u_person4, u_person5, u_person6, u_person7, u_person8, u_person9, u_person10};
std::vector<int>u_people(a_people, a_people+sizeof(a_people)/sizeof(a_people[0]));
int u_peopleSize = 10;
int u_peopleindex = 0;
int u_lineindex = 0;
int u_seenindex = 0;
int u_boredindex = 0;
const int a_line[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
std::vector<int> u_line(a_line, a_line+sizeof(a_line)/sizeof(a_line[0]));
const int a_seen[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
std::vector<int> u_seen(a_seen, a_seen+sizeof(a_seen)/sizeof(a_seen[0]));
const int a_bored[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
std::vector<int> u_bored(a_bored, a_bored+sizeof(a_bored)/sizeof(a_bored[0]));

int init_0_lineSize;
int init_0_seenSize;
int init_0_boredSize;
unsigned int init_0_time = 0;
struct init_0_link_ : public event {
    virtual void set_next(init_0_link_*n) {};
};
std::vector<init_0_link_*> init_0_list;
struct init_0_block_0 : public init_0_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    init_0_block_0() : inc_time(0), time(0), name("init_0_block_0") {};
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    init_0_link_*next;
    void set_next(init_0_link_*n) {next = n;};
    void foo() {
        if(next != NULL) {
            init_0_time += next->get_inc_time();
            next->set_time(init_0_time);
            event_q.add(init_0_time, next);
        }
    }
};
struct init_0_block_1 : public init_0_link_ {
    unsigned int time;
unsigned int inc_time;
std::string name;
init_0_block_1() : inc_time(99), time(99), name(""
    init_0_block_1") {}
unsigned int get_time() {return time;}
unsigned int get_inc_time() {return inc_time;}
void set_time(unsigned int time_) {time = time_;}
std::string get_name() {return name;}
init_0_link_ *next;
void set_next(init_0_link_ *n) {next = n;};

void foo() {
    init_0_lineSize = u_lineSize(u_line, u_peopleSize);
    init_0_seenSize = u_lineSize(u_seen, u_peopleSize);
    init_0_boredSize = u_lineSize(u_bored, u_peopleSize);

    std::cout << ("People still in line:") << std::endl;
    std::cout << (init_0_lineSize) << std::endl;
    std::cout << ("People seen:") << std::endl;
    std::cout << (init_0_seenSize) << std::endl;
    std::cout << ("People bored in line:") << std::endl;
    std::cout << (init_0_boredSize) << std::endl;

    if (next != NULL) {
        init_0_time += next->get_inc_time();
        next->set_time(init_0_time);
        event_q.add(init_0_time, next);
    }
}

struct init_0_block_2 : public init_0_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    init_0_block_2() : inc_time(1), time(1), name(""
        init_0_block_2") {}
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
}
init_0_link_ *next;
void set_next(init_0_link_ *n) {next = n;};
void foo() {
exit(0);

if(next != NULL) {
    init_0_time += next->get_inc_time();
    next->set_time(init_0_time);
    event_q.add(init_0_time, next);
}
}

unsigned int always_0_time = 0;
struct always_0_link_ : public event_ {
    virtual void set_next(always_0_link_ *n){};
};
std::vector<always_0_link_*> always_0_list;
struct always_0_block_0 : public always_0_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    always_0_block_0() : inc_time(0), time(0), name("always_0_block_0") {};
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    always_0_link_ *next;
    void set_next(always_0_link_ *n) {next = n;};
    void foo() {

        if(next != NULL) {
            always_0_time += next->get_inc_time();
            next->set_time(always_0_time);
            event_q.add(always_0_time, next);
        }
    }
};
struct always_0_block_1 : public always_0_link_ {
    unsigned int time;
unsigned int inc_time;
std::string name;
always_0_block_1() : inc_time(2), time(2), name("always_0_block_1") {}
unsigned int get_time() {return time;}
unsigned int get_inc_time() {return inc_time;}
void set_time(unsigned int time_) {time = time_;}
std::string get_name() {return name;}
always_0_link_ *next;
void set_next(always_0_link_* n) {next = n;}
void foo() {
    u_line = u_pushLine(u_line, u_people[0]);
    u_people = u_popLine(u_people);
    if (next != NULL) {
        always_0_time += next->get_inc_time();
        next->set_time(always_0_time);
        event_q.add(always_0_time, next);
    }
}
}

unsigned int always_1_time = 0;
struct always_1_link_ : public event_ {
    virtual void set_next(always_1_link_* n){};
};
std::vector<always_1_link_*> always_1_list;
struct always_1_block_0 : public always_1_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    always_1_block_0() : inc_time(0), time(0), name("always_1_block_0") {} 
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time_) {time = time_;}
    std::string get_name() {return name;}
    always_1_link_ *next;
    void set_next(always_1_link_* n) {next = n;}
    void foo() {
72
if (next != NULL) {
    always_1_time += next->get_inc_time();
    next->set_time(always_1_time);
    event_q.add(always_1_time, next);
}

struct always_1_block_1 : public always_1_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
    always_1_block_1() : inc_time(3), time(3), name("always_1_block_1") {}
    unsigned int get_time() {return time;}
    unsigned int get_inc_time() {return inc_time;}
    void set_time(unsigned int time) {time = time;}
    std::string get_name() {return name;}
    always_1_link_ *next;
    void set_next(always_1_link_ *n) {next = n;};
    void foo() {
        u_seen = u_pushLine(u_seen, u_line[0]);
        u_line = u_popLine(u_line);

        if (next != NULL) {
            always_1_time += next->get_inc_time();
            next->set_time(always_1_time);
            event_q.add(always_1_time, next);
        }
    }

};

unsigned int always_2_time = 0;
struct always_2_link_ : public event_ {
    virtual void set_next(always_2_link_ *n){};
};
std::vector(always_2_link_*) always_2_list;
struct always_2_block_0 : public always_2_link_ {
    unsigned int time;
    unsigned int inc_time;
    std::string name;
always_2_block_0() : inc_time(0), time(0), name("always_2_block_0") {}
unsigned int get_time() {return time;}
unsigned int get_inc_time() {return inc_time;}
void set_time(unsigned int time_) {time = time_}
std::string get_name() {return name;}
always_2_link_*next;
void set_next(always_2_link_*n) {next = n;};

void foo() {
if (next != NULL) {
always_2_time += next->get_inc_time();
next->set_time(always_2_time);
event_q.add(always_2_time, next);
}
}

struct always_2_block_1 : public always_2_link_ {
unsigned int time;
unsigned int inc_time;
std::string name;
always_2_block_1() : inc_time(4), time(4), name("always_2_block_1") {}
unsigned int get_time() {return time;}
unsigned int get_inc_time() {return inc_time;}
void set_time(unsigned int time_) {time = time_}
std::string get_name() {return name;}
always_2_link_*next;
void set_next(always_2_link_*n) {next = n;};
void foo() {
ubored = u_pushLine(ubored, u_line[0]);
u_line = u_popLine(u_line);
if (next != NULL) {
always_2_time += next->get_inc_time();
next->set_time(always_2_time);
event_q.add(always_2_time, next);
}
}
int main() {
    init_0_block_0 init_0_block_0obj;
    init_0_list.push_back(&init_0_block_0obj);
    init_0_block_1 init_0_block_1obj;
    init_0_list.push_back(&init_0_block_1obj);
    init_0_block_2 init_0_block_2obj;
    init_0_list.push_back(&init_0_block_2obj);
    always_0_block_0 always_0_block_0obj;
    always_0_list.push_back(&always_0_block_0obj);
    always_0_block_1 always_0_block_1obj;
    always_0_list.push_back(&always_0_block_1obj);
    always_1_block_0 always_1_block_0obj;
    always_1_list.push_back(&always_1_block_0obj);
    always_1_block_1 always_1_block_1obj;
    always_1_list.push_back(&always_1_block_1obj);
    always_2_block_0 always_2_block_0obj;
    always_2_list.push_back(&always_2_block_0obj);
    always_2_block_1 always_2_block_1obj;
    always_2_list.push_back(&always_2_block_1obj);
    for (int i = 0; i < init_0_list.size(); i++)
    {
        if (i != init_0_list.size() - 1)
            init_0_list[i]->set_next(init_0_list[i + 1]);
        else
            init_0_list[i]->set_next(NULL);
    }
    event_q.add(init_0_block_0obj.get_time(), &init_0_block_0obj);
    for (int i = 0; i < always_0_list.size(); i++)
    {
        if (i != always_0_list.size() - 1)
            always_0_list[i]->set_next(always_0_list[i + 1]);
        else
            always_0_list[i]->set_next(always_0_list[0]);
    }
    event_q.add(always_0_block_0obj.get_time(), &always_0_block_0obj);
    for (int i = 0; i < always_1_list.size(); i++)
    {
if (i != always_1_list.size() - 1)
    always_1_list[i]->set_next(always_1_list[i + 1]);
else
    always_1_list[i]->set_next(always_1_list[0]);
} 

event_q.add(always_1_block_0obj.get_time(), &always_1_block_0obj); 
for (int i = 0; i < always_2_list.size(); i++) 
{
    if (i != always_2_list.size() - 1)
        always_2_list[i]->set_next(always_2_list[i + 1]);
    else
        always_2_list[i]->set_next(always_2_list[0]);
} 

event_q.add(always_2_block_0obj.get_time(), &always_2_block_0obj); 
while (!event_q.empty()) {
    event_q.pop()->foo();
}
return 0;
}
7 Lessons Learned

7.1 Olivia Byer

Throughout this project I learned the importance of clear communication in teamwork. We often had multiple group members accidentally working on the same thing, or one group member working on something without the rest of the group realizing that they needed assistance. Had we communicated with each other about what we were working on not just at our biweekly check ins but also when we started a new task I think there would have been less frustration.

I also personally learned the value of trial and error when you do not understand something. At first when I found myself confused about a portion of the project, I hoped that someone else would be the one to pick it up so that I could have more time to figure it out. However, as time went on and I took ownership of the framework of semantic check, I learned the value in sitting down and coding and debugging in order to actually learn. On a more technical side, I enjoyed learning about how functional programming really works and in what situations it is most useful. I had exposure to functional languages coming in but didn’t truly understand their structure or roots in lambda calculus, so that was extremely valuable and interesting to me.

My advice to future teams is to take your time coming up with what you really want your language to be. And more specifically, coming up with what you want your language to do. Our original proposal for a language created really interesting programs but was too broad in its application, making it too difficult of a task. We began to narrow it down to a simulation language and had a much easier time from then on as far as understanding what our language was supposed to do.

7.2 Mauricio Castaneda

It is hard working in teams. There are many different personalities and skill levels involved. It is hard to split up the work evenly and efficiently, especially because you don’t have all the theory necessary to be able to build a programming language until well into the semester. Once you have all the necessary knowledge, you might realize some of the work you did is no longer useful, and you must revise your strategy. Start early, and spread the work throughout the semester. It is also important to reach compromises with your other team members for the project to be successful.

7.3 Josh Itwaru

Task division and communication are critical for team productivity. On occasion, multiple people would be working on the same task, only to realize the futility of their work when
they see that the task that they would have been working on had already been completed and pushed to github. For me, this led to bouts of reading over completed code to keep up with the project instead of contributing useful code to the project. Only the beginning of the project is prone to this because the AST and parser aren’t difficult to write.

Make sure you fully understand the basics before trying to write up complicated project ideas. In my case, I learn almost entirely through practice, so combined with the previous point, I was pretty useless for some chunk of the beginning of the project. Not only that, but I’m pretty sure that I wasted a jack load of time trying to solidify my understanding of the basics by reading code instead of writing code. So in order to make sure I was able to contribute to the project, I practiced by expanding the sample calculator program provided on the course website. After I felt as though I gained an adequate understanding, then I moved back to the project code, and it was much easier. If you learn by coding, then write code for something, even if it’s not your task assigned by the group.

Test your code after every line. The ocaml compiler is really good at detecting errors, but sometimes the error messages don’t help - one of the most mysterious and common errors I’ve gotten is ”syntax error” on the first line of a function declaration, which actually means that there’s an error somewhere at the end of the previous function, like, you’re missing an ”in” for a ”let” or something.

If you can’t figure out an ocaml error, go to the TAs. Chances are, you’re making some rookie ocaml mistake that isn’t too complicated, but you just don’t have enough ocaml experience to figure it out.

Use the ocaml online compiler to test ocaml functions if you’re not sure what they’re going to do before writing, for example, a huge fold_left, when you don’t really remember what fold_left does. This will save you a lot of heartache and help ensure that at the very least, your ocaml code is doing what you expect it to do.

While you don’t have to do every aspect of the project, you should do a bit of everything on your own time anyway by expanding the calculator program to a fully functional, semantically checked compiler, for example. While you do learn things in class, you’ll get a far richer experience by actually writing code.

7.4 Dina Lamdany

I learned that it is important to set clear expectations among group members of who should be contributing what—and when—so as to not duplicate work, and so as to prevent one person from taking on too much. It is important to strike a careful balance between delegating and getting things done, and it’s a difficult one to achieve. I should have been
more assertive about deadlines and making things get done on time. I also learned that
people surprise you, and that sometimes the people who seem like they don’t know what’s
going on, do. Finally, I learned that picking a team is super important, and you should be
careful to go into the class with your friends if you can.

7.5 Tony Ling

My most important learning experience on this project is group work experience and working with others. I found that communication is important, especially when no one is communicating. You should ask your teammates what it is they think they are doing and then tell them what they are actually supposed to do. Working on code and implementing the compiler is easier than agreeing on standards that everyone follows. Decisions that are made that affects and changes the compiler also results in not every member getting the memo, and that leads to problems that arise later on when we are individually working on our parts.

My advice would be to start earlier than when you think you should’ve started. There are also always bugs, even when you think it a section of the code is finished, so start early.

8 Appendix

8.1 scanner.mll

```
{ open Parser }

rule token = parse
    [ ' ' 't' 'r' 'n' ] { token lexbuf } (* Whitespace *)
| "/*" { comment lexbuf } (* Comments *)
| '(' { LPAREN }
| ')' { RPAREN }
| '[' { LBRAC }
| ']' { RBRAC }
| '{' { LBRACE }
| '}' { RBRACE }
| ';' { SEMI }
| ':' { COLON }
| ',' { COMMA }
| '.' { DOT }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
```
\[
\begin{align*}
'/ ' & \quad \text{DIVIDE} \\
'= ' & \quad \text{ASSIGN} \\
'==' & \quad \text{EQ} \\
'!==' & \quad \text{NEQ} \\
',' & \quad \text{LT} \\
'<==' & \quad \text{LEQ} \\
','>=' & \quad \text{GEQ} \\
'if' & \quad \text{IF} \\
'!' & \quad \text{NOT} \\
'&' & \quad \text{AND} \\
'|' & \quad \text{OR} \\
'#' & \quad \text{DELAY} \\
'++' & \quad \text{INC} \\
'--' & \quad \text{DEC} \\
'%' & \quad \text{MOD} \\
'else' & \quad \text{ELSE} \\
'for' & \quad \text{FOR} \\
'while' & \quad \text{WHILE} \\
'return' & \quad \text{RETURN} \\
'func' & \quad \text{FUNC} \\
'main' & \quad \text{MAIN} \\
'init' & \quad \text{INIT} \\
'always' & \quad \text{ALWAYS} \\
'Terminate' & \quad \text{TERMINATE} \\
'bool' & \quad \text{BOOLEAN} \\
'\text{string}' & \quad \text{STRING} \\
'\text{int}' & \quad \text{INT} \\
'\text{float}' & \quad \text{FLOAT} \\
'\text{void}' & \quad \text{VOID} \\
'\text{true}' & \quad \text{true} \\
'false' & \quad \text{false} \\
\end{align*}
\]
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) } |

and comment = parse
   "*/" { token lexbuf } |
   _ { comment lexbuf }

8.2 parser.mly

{% open Ast %}
{% open Type %}
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA RBRAC LBRAC COLON DOT
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token NOT INC DEC
%token EQ NEQ LT LEQ GT GEQ OR AND MOD
%token RETURN IF ELSE FOR WHILE FUNC
%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <bool> BOOL_LITERAL
%token <string> STRING_LITERAL TYPE ID
%token EOF
%token DELAY MAIN INIT ALWAYS
%token TERMINATE
%token BOOLEAN STRING INT FLOAT VOID OBJECT

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left AND OR
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT
%right UMINUS DEC INC
%nonassoc LPAREN RPAREN LBRAC RBRAC

%start program
%type <Ast.program> program

81
%%

program:
  main {} $1
    | fdecl program { ($1 : : fst $2), snd $2 }  

main:
  MAIN LPAREN RPAREN LBRACE vdecl_list timeblock_list RBRACE { $5, $6 }  

var_type:
  INT { Int }
    | FLOAT { Float }
    | BOOLEAN { Boolean }
    | STRING { String }

ret_type:
  var_type { Datatype($1) }
    | VOID { Datatype(Void) }
    | var_type LBRAC RBRAC { Arraytype(Datatype($1)) }

timeblock_list:
  /* nothing */ { [] }
    | timeblock_list timeblock { $2 :: $1 }  

timeblock:
  INIT LBRACE events RBRACE { Init($3) }
    | ALWAYS LBRACE events RBRACE { Always($3) }  

fdecl:
  FUNC ret_type ID LPAREN formals_opt RPAREN RBRACE stmt_list RBRACE  
    { { return = $2;  
      fname = Ident($3);  
      formals = $5;  
      body = List.rev $8 } }  

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

82
formal_list:
  param { [$1] }
  formal_list COMMA param { $3 :: $1}

param:
  var_type ID { Formal(Datatype($1),Ident($2)) }
  var_type ID LBRAC RBRAC { Formal(Arraytype(Datatype($1)),
    Ident($2))}

event:
  DELAY INT_LITERAL stmt_list { Event($2,List.rev $3)}

events:
  stmt_list { [ Event(0,List.rev $1)])
  | stmt_list event_list { Event(0, List.rev $1) :: $2}

event_list:
  event { [$1] }
  | event event_list { $1 :: $2 }

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

vdecl_list:
  /* nothing */ {[]}
  | vdecl SEMI vdecl_list { $1 :: $3 }

vdecl:
  var_type ID { VarDecl(Datatype($1),Ident($2)) }
  | var_type ID ASSIGN expr { VarAssignDecl(Datatype($1),Ident($2),ExprVal($4)) }
  | var_type ID LBRAC RBRAC { VarDecl(Arraytype(Datatype($1)),
    Ident($2))}
  | var_type ID LBRAC RBRAC ASSIGN LBRAC expr_list RBRAC { VarAssignDecl(Arraytype(Datatype($1)),Ident($2),ArrVal($7))}

expr_list:
  /* nothing */ { [] }
  | expr COMMA expr_list { $1 :: $3 }

83
expr { [$1] }

stmt:
expr SEMI { Expr($1)}
| TERMINATE SEMI { Terminate }
| 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) }
| vdecl SEMI { Declaration($1) }
| ID LBRAC INT_LITERAL RBRAC ASSIGN expr SEMI { ArrElemAssign (Ident($1), IntLit($3), $6)}
| ID LBRAC ID RBRAC ASSIGN expr SEMI { ArrElemAssign(Ident($1), Variable(Ident($3)), $6) }
| ID ASSIGN LBRAC expr_list RBRAC SEMI { ArrAssign(Ident($1), $4) }
| ID ASSIGN expr SEMI { Assign(Ident($1), $3) }

expr_opt:
| ID ASSIGN expr { ExprAssign(Ident($1), $3) }
| expr { $1 }

eexpr:
INT_LITERAL { IntLit($1) }
| FLOAT_LITERAL { FloatLit($1) }
| STRING_LITERAL { StringLit($1) }
| BOOL_LITERAL { BoolLit($1) }
| var_type LPAREN expr RPAREN { Cast(Datatype($1),$3) }
| ID { Variable(Ident($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) }
84
8.3 type.mli

```ml
type var_type =
  | Int
  | Float
  | String
  | Boolean
  | Void
```

8.4 ast.mli

```ml
open Type

type binop = Add | Sub | Mult | Div | Equal | Neq | Less | Leq |
  | Greater | Geq | Mod | And | Or

type unop = Neg | Inc | Dec | Not

type ident =
  | Ident of string

type datatype =
  | Datatype of Type.var_type |
  | Arraytype of datatype

type expr =
  | IntLit of int
  | BoolLit of bool
```
| FloatLit of float |
| StringLit of string |
| Variable of ident |
| Unop of unop * expr |
| Binop of expr * binop * expr |
| ArrElem of ident * expr |
| ExprAssign of ident * expr |
| Cast of datatype * expr |
| Call of ident * expr list |

```plaintext
type value =
  ExprVal of expr |
  ArrVal of expr list
```

```plaintext
and decl =
  VarDecl of datatype * ident |
  VarAssignDecl of datatype * ident * value
```

```plaintext
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 |
  Declaration of decl |
  Assign of ident * expr |
  ArrAssign of ident * expr list |
  ArrElemAssign of ident * expr * expr |
  Terminate
```

```plaintext
type event =
  Event of int * stmt list
```

```plaintext
type formal =
  Formal of datatype * ident
```

```plaintext
type func_decl = {
  return: datatype;
  fname : ident;
  formals : formal list;
}
```

86
body : stmt list;
}

type thread =
  Init of event list
  | Always of event list

type program =
  func_decl list * (decl list * thread list)

**8.5 semantic_check.ml**

open Ast
open Sast
open Type

exception Error of string

(*a symbol table consisting of the parent as the variables*)
type symbol_table = {
  parent: symbol_table option;
  (* Added value so that we can check "out of bounds" error on arrays *)
  variables: (ident * datatype * value option) list;
  (* Arrays are explicitly added here with their *)
  (* arrays: (ident * datatype * sexpr list) list *)
}

(*a function table containing function definitions*)
type function_table = {
  functions: (ident * var_type * formal list * stmt list) list
}

(*our environment*)
type translation_environment = {
  return_type: datatype; (*function's return type*)
  return_seen: bool; (*does the function have
  a return statement*)
  location: string; (*init, always, main, or
  function name, used for global or local checking*)

87
global_scope: symbol_table; (*symbol table for global variables*)
var_scope: symbol_table; (*symbol table for local variables*)
fun_scope: function_table; (*symbol table for functions*)

(* search for a function in our function table*)
let rec find_function (fun_scope: function_table) name =
  List.find (fun (s,_,_,_) -> s=name) fun_scope.functions

let basic_math t1 t2 = match (t1, t2) with
  (Float, Int) -> (Float, true)
| (Int, Float) -> (Float, true)
| (Int, Int) -> (Int, true)
| (Float, Float) -> (Int, true)
| (_,_) -> (Int, false)

let relational_logic t1 t2 = match (t1, t2) with
  (Int,Int) -> (Boolean,true)
| (Float,Float) -> (Boolean,true)
| (Int,Float) -> (Boolean,true)
| (Float,Int) -> (Boolean,true)
| (_,_) -> (Boolean,false)

let basic_logic t1 t2 = match(t1,t2) with
  (Boolean,Boolean) -> (Boolean,true)
| (_,_) -> (Int,false)

let equal_logic t1 t2 = match(t1,t2) with
  (Boolean,Boolean) -> (Boolean,true)
| (Int,Int) -> (Boolean,true)
| (Float,Float) -> (Boolean,true)
| (Int,Float) -> (Boolean,true)
| (Float,Int) -> (Boolean,true)
| (String,String) -> (Boolean,true)
| (_,_) -> (Int,false)

(*extracts the type from a datatype declaration*)
let rec get_type_from_datatype = function
Datatype (t) -> t
| Arraytype (ty) -> get_type_from_datatype ty

let get_binop_return_value op typ1 typ2 =
  let t1 = get_type_from_datatype typ1 and t2 =
    get_type_from_datatype typ2 in
  let (t, valid) =
    match op with
    | Add -> basic_math t1 t2
    | Sub -> basic_math t1 t2
    | Mult -> basic_math t1 t2
    | Div -> basic_math t1 t2
    | Mod -> equal_logic t1 t2
    | Equal -> equal_logic t1 t2
    | Neq -> equal_logic t1 t2
    | Less -> relational_logic t1 t2
    | Leq -> relational_logic t1 t2
    | Greater -> relational_logic t1 t2
    | Geq -> relational_logic t1 t2
    | Or -> basic_logic t1 t2
  in (Datatype (t), valid)

(* extracts the type and name from a Formal declaration *)

let get_name_type_from_formal env = function
  Formal (datatype, ident) -> (ident, datatype, None)

(* Find the variable. If you find the variable:
  Create a new list with the updated variable *)

let update_variable env (name, datatype, value) =
  let ((_,_,_), location) =
    try (fun var_scope -> ((List.find (fun (s,_,_) -> s=name)
      var_scope),1)) env.var_scope.variables
    with Not_found -> try (fun var_scope -> ((List.
      find (fun (s,_,_) -> s=name) var_scope),2))
      env.global_scope.variables
    with Not_found -> raise Not_found in
  let new_envf =
    match location with
    | 1 ->
      (* update local vars *)
let new_vars = List.map (fun (n, t, v) ->
    if(n=name) then (name, datatype, value) else (n, t, v)) env.var_scope.variables in
let new_sym_table = {parent = env.var_scope.parent; variables = new_vars ;} in
let new_env = {env with var_scope = new_sym_table} in
new_env

| 2 ->
(* update global vars *)
let new_vars = List.map (fun (n, t, v) ->
    if(n=name) then (name, datatype, value) else (n, t, v)) env.global_scope.variables in
let new_sym_table = {parent = env.var_scope.parent; variables = new_vars ;} in
let new_env = {env with global_scope = new_sym_table} in
new_env

| _ -> raise (Error("Undefined scope"))
in new_envf

let update_list expr_list index expr =
    let xarr = Array.of_list expr_list in
    let _ = Array.set xarr index expr in
    let xlist = Array.to_list xarr in
    xlist

(*search for variable in global and local symbol tables*)
let find_variable env name =
    try List.find (fun (s,_,_) -> s=name) env.var_scope.variables
    with Not_found -> try List.find(fun (s,_,_) -> s=name)
    env.global_scope.variables
    with Not_found -> raise Not_found

let get_int_from_var env v =
let ( _, ty, value ) = try find_variable env v with Not_found -> raise ( Error("Cannot index a non-initialized variable")) in
match value with
  Some(ExprVal(IntLit(x))) -> x
| _ -> raise (Error("Non-integer variable value"))

(*Semantic checking on expressions*)
let rec check_expr env e = match e with
  IntLit(i) -> Datatype(Int)
| BoolLit(b) -> Datatype(Bool)
| FloatLit(f) -> Datatype(Float)
| StringLit(s) -> Datatype(String)
| Variable(v) -> let ( _, s_type, _ ) = try find_variable env v with
  Not_found -> raise (Error("Undeclared Identifier ")) in
  s_type
| Unop(u, e) ->
  let t = check_expr env e in
  match u with
    Not -> if t = Datatype(Bool) then t else raise (Error("Cannot negate a non-boolean value"))
  | _  -> if t = Datatype(Int) then t else if t = Datatype(Float) then t
  else raise (Error("Cannot perform operation on "))
| Binop(e1, b, e2) ->
  let t1 = check_expr env e1 and t2 = check_expr env e2 in
  let (t, valid) = get_binop_return_value b t1 t2 in
  if valid then t else raise (Error("Incompatible types with binary operator"));
| ArrElem(id, expr) ->
  (* return SArrElem(id, expr, datatype) where: 
   * id is name of array variable
   * expr has datatype Int
   * datatype is type of array *)
  let ( _, ty, v ) = try find_variable env id with
Not_found \rightarrow \text{raise}(\text{Error}("\text{Uninitialized array}")) \text{ in }
\text{let el\_type = (match ty with } \\
\text{Arraytype(Datatype(x)) } \rightarrow \text{Datatype(x)} \\
| \_ \rightarrow \text{raise}(\text{Error}("\text{Cannot index a non-array expression}")) \text{ in } \\
\text{let expr\_type = check\_expr env expr in } \\
\text{let sty = match expr\_type with } \\
\text{Datatype(ty) } \rightarrow \text{Some(ty)} \\
| \text{Arraytype(dt) } \rightarrow \text{None in } \\
\text{let ty = match sty with } \\
\text{Some(ty) } \rightarrow \text{ty} \\
| \text{None } \rightarrow \text{raise}(\text{Error}("\text{Can’t invoke array element as index}")) \text{ in } \\
\text{let }\_ = \text{if not(ty=Int) then raise}(\text{Error}("\text{index must be an integer}")) \text{ in } \\
(\text{el\_type}) \\
| \text{ExprAssign}(id, e) \rightarrow \text{let } (\_, t1, \_) = (\text{find\_variable env id}) \text{ and } t2 = \\
\text{check\_expr env e} \\
\text{in (if not(t1 = t2) then (raise}(\text{Error}("\text{Mismatch in types for assignment}")))) ; \text{check\_expr env e} \\
| \text{Cast}(ty, e) \rightarrow ty \\
\text{| Call(Ident("print"),e) } \rightarrow \text{let }\_ = \text{List.map}(\text{fun exp } \rightarrow \\
\text{check\_expr env exp) e in } \\
\text{Datatype(Void)} \\
\text{| Call(Ident("print\_time"),e) } \rightarrow \text{let }\_ = \text{List.map}(\text{fun exp } \rightarrow \\
\text{check\_expr env exp) e in } \\
\text{Datatype(Void)} \\
\text{| Call(id, e) } \rightarrow \text{try (let } (\text{fname, fret, fargs, fbody}) = \\
\text{\text{find\_function env.fun\_scope id in } } \\
\text{let el\_tys = List.map (fun exp } \rightarrow \text{check\_expr env exp) e in } \\
\text{let fn\_tys = List.map (fun farg } \rightarrow \text{let } (\_, ty, \_) = \\
\text{get\_name\_type\_from\_formal env farg in ty) \\
\text{fargs in } \\
\text{if not(el\_tys = fn\_tys) then } \\
\text{raise}(\text{Error}("\text{Mismatching types in function call"}) \text{ else } \\
\text{Datatype(fret))} \\
\text{with Not\_found } \rightarrow \\
92
raise (Error("Undeclared Function "))

let get_val_type env = function
    ExprVal(expr) -> check_expr env expr
  | ArrVal(expr_list) -> check_expr env (List.hd expr_list)

let get_var_scope env name =
    try (let (_, _, _) = List.find (fun (s, _, _) -> s = name) env.
        var_scope.variables in Local)
    with Not_found -> try (let (_, _, _) = List.find (fun (s, _, _) -> s = name) env.
        env.global_scope.variables in Global)
    with Not_found -> raise (Error("get_var_scope is failing"))

(*converts expr to sexpr*)
let rec get_sexpr env e = match e with
    IntLit(i) -> SIntLit(i, Datatype(Int))
  | BoolLit(b) -> SBoolLit(b, Datatype(Boolean))
  | FloatLit(f) -> SFloatLit(f, Datatype(Float))
  | StringLit(s) -> SStringLit(s, Datatype(String))
  | Variable(id) -> SVariable(SIdent(id, get_var_scope env id), check_expr env e)
  | Unop(u,ex) -> SUnop(u, get_sexpr env ex, check_expr env e)
  | Binop(e1,b,e2) -> SBinop(get_sexpr env e1, b, get_sexpr env e2, check_expr env e)
  | ArrElem(id, expr) -> SArrElem(SIdent(id, get_var_scope env id), get_sexpr env expr, check_expr env expr)
  | ExprAssign(id,ex) -> SExprAssign(SIdent(id, get_var_scope env id), get_sexpr env ex, check_expr env e)
  | Cast(ty,ex) -> SCast(ty, get_sexpr env ex, ty)
    | Call(Ident("print"),ex_list) -> let s_ex_list = List.
      map(fun exp -> get_sexpr env exp) ex_list
    in SCall(SIdent(Ident("print"),Global), s_ex_list, check_expr env e)
  | Call(Ident("print_time"),ex_list) -> let s_ex_list = List.
      map(fun exp -> get_sexpr env exp) ex_list
    in SCall(SIdent(Ident("print_time"),Global), s_ex_list, check_expr env e)
\[
\begin{align*}
| \text{Call}(\text{id}, \text{ex}\_\text{list}) & \rightarrow \text{let } s\_\text{ex}\_\text{list} = \text{List.map}(\text{fun } \text{exp} \rightarrow \text{get_sexpr env} \\
& \quad \text{exp}) \text{ex}\_\text{list} \text{ in SCall(SIdent(id,Global),s\_ex\_list,} \\
& \quad \text{check_expr env e}) \\
\end{align*}
\]

(* Make sure a list contains all items of only a single type; 
returns (sexpr list, type in list) *)

let get_sexpr_list env expr_list = 
  let sexpr_list = 
    \[
    \text{List.map}(\text{fun } \text{expr} \rightarrow \text{let t1 = get_type_from_datatype(} \\
    \quad \text{check_expr env (List.hd} \\
    \quad \text{expr_list))} \\
    \quad \text{and t2 = get_type_from_datatype (} \\
    \quad \text{check_expr env expr} \text{ in} \\
    \quad \text{if(t1=t2) then get_sexpr env expr} \\
    \quad \text{else raise (Error("Type Mismatch"))} \) \\
    \text{expr_list in sexpr_list} \\
    \]

(* replacement for get_typed_value *)

let get_sval env = function 
  ExprVal(expr) \rightarrow \text{SExprVal(get_sexpr env expr)} \\
  | ArrVal(expr_list) \rightarrow \text{SArrVal(get_sexpr_list env expr_list)}

let get_datatype_of_list env expr_list = 
  let ty = \text{List.fold_left}(\text{fun } \text{dt1 expr2} \rightarrow \\
  \text{let dt2 = check_expr env expr2 in} \\
  \text{if(dt1 = dt2) then dt1 else raise} \\
  \text{(Error("Inconsistent array types "))} \) \\
  \text{check_expr env (List.hd expr_list)) expr_list in ty}

let get_datatype_from_val env = function 
  ExprVal(expr) \rightarrow \text{check_expr env expr} \\
  | ArrVal(expr_list) \rightarrow \text{get_datatype_of_list env expr_list}
(* if variable is not found, then add it to table and return SVarDecl *)
(* if variable is found, throw an error: multiple declarations *)
let get_sdecl env decl = match decl with
  (* if ident is in env, return typed sdecl *)
  VarDecl(datatype, ident) -> (SVarDecl(datatype, SIdent(ident, Local)), env)
  VarAssignDecl(datatype, ident, value) ->
    let sv = get_sval env value in
    (SVarAssignDecl(datatype, SIdent(ident, Local), sv), env)

let get_name_type_from_decl decl = match decl with
  VarDecl(datatype, ident) -> (ident, datatype)
  VarAssignDecl(datatype, ident, value) -> (ident, datatype)

let get_name_type_val_from_decl decl = match decl with
  VarDecl(datatype, ident) -> (ident, datatype, None)
  VarAssignDecl(datatype, ident, value) -> (ident, datatype, Some(value))

(* returns tuple (left hand id, left hand id type, right hand value type) *)
let get_name_type_from_var env = function
  VarDecl(datatype,ident) -> (ident,datatype,None)
  VarAssignDecl(datatype,ident,value) -> (ident,datatype,Some(value))

(*extracts the stmt list from the event declaration*)
let rec get_time_stmts_from_event = function
  Event(i,stmts) -> (i,stmts)

(*extracts event list from thread declaration*)
let rec get_events_from_thread = function
  Init(event) -> event
  | Always(event2) -> event2

(*function that adds variables to environment’s var_scope for use in functions*)
let add_to_var_table env name t v =
  let new_vars = (name,t,v)::env.var_scope.variables in
let new_sym_table = {parent = env.var_scope.parent; variables = new_vars;} in
let new_env = {env with var_scope = new_sym_table} in new_env

(*function that adds variables to environment’s global_scope for use with main*)
let add_to_global_table env name t v =
  let new_vars = (name,t,v)::env.global_scope.variables in
  let new_sym_table = {parent=env.global_scope.parent;
    variables = new_vars;} in
  let new_env = {env with global_scope = new_sym_table} in
  new_env

(* check both sides of an assignment are compatible*)
let check_assignments type1 type2 = match (type1, type2) with
  (Int, Int) -> true
  (Float, Float) -> true
  (Int, Float) -> true
  (Float, Int) -> true
  (Boolean, Boolean) -> true
  (String, String) -> true
  (_,_) -> false

(* checks the type of a variable in the symbol table*)
(* Changed from "check_var_type" *)
let match_var_type env v t =
  let (name,ty,value) = find_variable env v in
  if (t<>ty) then false else true

(* Checks that a function returned if it was supposed to*)
let check_final_env env =
  (if (false = env.return_seen && env.return_type <> Datatype(Void)) then
    raise (Error("Missing Return Statement"));
  true

(* Default Table and Environment Initializations *)
let empty_table_initialization = {parent=None; variables =[];}
let empty_function_table_initialization = {functions=[(Ident("print_string"), Void, [Formal(Datatype(String), Ident("s")))]}
let empty_environment = {return_type = Datatype(Void);
  return_seen = false; location="main"; global_scope =
  empty_table_initialization; var_scope =
  empty_table_initialization; fun_scope =
  empty_function_table_initialization}

let find_global_variable env name =
  try List.find (fun (s,_,_) -> s=name) env.global_scope.
variables
with Not_found -> raise Not_found

let initialize_globals (globals, env) decl =
  let (name, ty) = get_name_type_from_decl decl in
  let ((_,dt,_) , found) = try (fun f -> ((f env name
           ),true)) find_global_variable with
    Not_found ->
      ((name,ty,None),false) in
  let ret = if(found=false) then
    match decl with
    VarDecl(datatype,ident) ->
      let (name,ty,_) =
        get_name_type_from_var env
      decl in
      let new_env = add_to_global_table env
      name ty None in
      (SVarDecl(datatype,SIdent(ident, Global))::globals, new_env)
    | VarAssignDecl(dt,id,value) ->
      let t1 =
        get_type_from_datatype
      (dt) and t2 =
        get_type_from_datatype
      (get_datatype_from_val env value) in
      if(t1=t2) then
        let (n, t, v) =
          get_name_type_val_from_decl
      decl in

97
let new_env =
    add_to_global_table
    env n t v in
    (SVarAssignDecl(
        dt , SIdent (id ,
            Global),
        get_sval env
        value)::
            globals ,
            new_env)
else raise (Error("Type
    mismatch"))
else raise (Error("Multiple
    declarations")) in ret

let find_local_variable env name =
    try List.find (fun (s , _, _) -> s=name) env.var_scope.
    variables
    with Not_found -> raise Not_found

(*Semantic checking on a stmt*)
let rec check_stmt env stmt = match stmt with
  | Block (stmt_list) ->
    let new_env=env in
    let getter (env , acc) s =
        let (st , ne) = check_stmt env s in
        (ne , st::acc) in
    let (ls , st) = List.fold_left (fun e s -> getter e
        s) (new_env , []) stmt_list in
    let revst = List.rev st in
    (SBlock (revst) , ls)
  | Expr (e) ->
    let _ = check_expr env e in
    (SSExpr (get_sexpr env e) , env)
  | Return (e) ->
    let type1 = check_expr env e in
    (if not ((type1=env.return_type)) then
        raise (Error("Incompatible Return Type")) )
    ;
    let new_env = {env with return_seen=true} in

98
(SReturn(get_expr env e),new_env)
  | If(e,s1,s2) ->
    let t=get_type_from_datatype(check_expr env e) in
    (if not (t=Boolean) then
      raise (Error("If predicate must be a boolean"));
    let (st1,new_env1)=check_stmt env s1
    and (st2,new_env2)=check_stmt env s2 in
    let ret_seen=(new_env1.return_seen&&new_env2.return_seen) in
    let new_env = {env with return_seen=ret_seen} in
    (SIf((get_expr env e),st1,st2),new_env)
  | For(e1,e2,e3,s) ->
    let t1=get_type_from_datatype(check_expr env e1)
    and t2=get_type_from_datatype(check_expr env e2)
    and t3=get_type_from_datatype(check_expr env e3) in
    (if not (t1=Int && t3=Int && t2=Boolean) then
      raise (Error("Improper For loop format"));
    let (st,new_env)=check_stmt env s in
    (SFor((get_expr env e1),(get_expr env e2),
      (get_expr env e3),st),new_env)
  | While(e,s) ->
    let t=get_type_from_datatype(check_expr env e) in
    (if not (t=Boolean) then
      raise (Error("Improper While loop format"));
    let (st,new_env)=check_stmt env s in
    (SWhile((get_expr env e),st),new_env)
  | Ast.Declaration (decl) ->
    (* If variable is found, multiple decls error
      If variable is not found and var is
      assigndecl, check for type compat *)
    let (name,ty)=get_name_type_from_decl decl in
    let ((_,dt,_,found)=try (fun f -> ((f env name),true)) find_local_variable with
      Not_found ->
        ((name,ty,None),false) in
    let ret = if (found=false) then
      match decl with

VarDecl(…)

let (sdecl,_) = get_sdecl env decl in
let (n, t, v) =
    get_name_type_val_from_decl decl in
let new_env =
    add_to_var_table env n t v in
    (SDeclaration(sdecl), new_env)

| VarAssignDecl(dt, id, value) ->
  let t1 =
    get_type_from_datatype (dt) and t2 =
    get_type_from_datatype (get_datatype_from_val env value) in
  if t1=t2 then
    let (sdecl,_) = get_sdecl env decl in
    let (n, t, v) =
        get_name_type_val_from_decl decl in
    let new_env =
        add_to_var_table env n t v in
        (SDeclaration(sdecl),
        new_env)
  else raise (Error("Type mismatch"))
  else raise (Error("Multiple declarations")) in ret

| Ast.Assign(ident, expr) ->
  (* make sure 1) variable exists, 2) variable and expr have same types *)
  let (_, dt, _) = try find_variable env ident with
      Not_found -> raise (Error("Uninitialized"))
variable") in
let t1 = get_type_from_datatype dt
and t2 = get_type_from_datatype(check_expr env expr) in
if( not(t1=t2) ) then
    raise (Error("Mismatched type assignments "));
let sexpr = get_sexp env expr in
let new_env = update_variable env (ident,dt,Some((ExprVal(expr)))) in
(SAssign(SIdent(ident, get_var_scope env ident), sexpr), new_env)
| Ast.ArrAssign(ident, expr_list) ->
    (* make sure 1) array exists and 2) all types in
      expr list are equal *)
let (n,dt,v) = try find_variable env ident with
    Not_found -> raise (Error("Undeclared array")) in
let sexpr_list = List.map (fun expr2 ->
    let expr1 = List.hd expr_list in
    let t1 = get_type_from_datatype(check_expr env expr1) and t2 =
        get_type_from_datatype(check_expr env expr2) in
        if(t1=t2) then
            let sexpr2 = get_sexp env expr2 in sexpr2
        else raise (Error("Array has inconsistent types"))
    expr_list in
let _ =
    let t1=get_type_from_datatype(check_expr env (List.hd expr_list)) and t2=
        get_type_from_datatype(dt) in
    if(t1!=t2) then raise (Error("Type Mismatch")) in
let new_env = update_variable env (n,dt,(Some(ArrVal(expr_list)))) in
(SArrAssign(SIdent(ident, get_var_scope env ident)
    , sexpr_list), new_env)
| Ast.ArrElemAssign(ident, expr1, expr2) ->
(* Make sure
1) array exists (if it exists, then it was already declared and semantically checked)
2) expr matches type of array
3) index is not out of bounds *)
let (id, dt, v) = try find_variable env ident
   with Not_found -> raise (Error("Undeclared array")) in
let t1 = get_type_from_datatype(dt) and t2 =
   get_type_from_datatype(check_expr env expr2)
in
let _ = if(t1=t2) then true else raise (Error("Type Mismatch")) in
let _ = (match v with
   Some(ArrVal(el)) -> (*
      get_expr_list env *)el
   | None -> raise (Error("No expression on right hand side "))
   | _ -> raise (Error("???"))
   ) in
let t = get_type_from_datatype(check_expr env expr1)
in
let _ = if not(t=Int) then raise (Error("Array index must be an integer")) in
   (SArrElemAssign(SIdent(ident, get_varscope env ident), get_expr env expr1, get_expr env expr2), env)
| Terminate -> (STerminate, env)

let get_stmt_list env stmt_list =
   List.fold_left (fun (stmt_list, env) stmt ->
      let (stmt, new_env) = check_stmt env stmt in
      (stmt::stmt_list, new_env)) ([], env) stmt_list

(* add a function to the environment*)
let add_function env sfunc_decl =
   let f_table = env.fun_scope in
   let old_functions = f_table.functions in
   match sfunc_decl with
      SFunc_Decl(sfuncstr, datatype) ->
         let func_name = sfuncstr.sfname in
let func_type = get_type_from_datatype sfuncstr.sreturn in
let func_formals = sfuncstr.sformals in
let func_body = sfuncstr.sbody in
let new_functions = (func_name, func_type, func_formals, func_body): old_functions in
let new_fun_scope = {functions = new_functions} in
let final_env = {env with fun_scope = new_fun_scope} in
final_env

(* Semantic checking on a function *)
let check_func env func_declaration =
  let new_locals = List.fold_left(fun a vs -> (get_name_type_from_formal env vs)::a) [] func_declaration.formals in
  let new_var_scope = {parent=Some(env.var_scope); variables = new_locals;} in
  let new_env = {return_type = func_declaration.return;
                 return_seen=false; location="in_func";
                 global_scope = env.global_scope; var_scope = new_var_scope; fun_scope = env.fun_scope} in

(* let final_env =List.fold_left(fun env stmt -> snd (check_stmt env stmt)) new_env func_declaration.body in *)
let (typed_statements, final_env) = get_sstmt_list new_env func_declaration.body in
let _=check_final_env final_env in
let sfuncdecl = ({sreturn = func_declaration.return;
                  sfname = func_declaration.fname; sformals = func_declaration.formals;
                  sbody = typed_statements}) in
(SPfunc_Decl(sfuncdecl, func_declaration.return), env)

let initialize_functions env function_list =
let (typed_functions, last_env) = List.fold_left
(fun (sfuncrecl_list, env) func-> let (sfuncrecl, _) = check_func env func in

(*Semantic checking on events*)
let check_event (typed_events, env) event =
let (time, statements) = get_time_stmts_from_event event in
let (typed_statements, final_env) = List.fold_left (fun (sstmt_list, env) stmt -> let (sstmt, new_env) =
  check_stmt env stmt in
  (sstmt::sstmt_list,new_env)) ([],env) statements
in (SEvent(time, typed_statements)::typed_events, final_env)

(* Semantic checking on threads*)
let check_thread env thread_declaration = match
  thread_declaration with
  Init(events) -> let (typed_events,_) = List.fold_left
    check_event ([],env) events
    in SInit(typed_events)
| Always(events) -> let (typed_events,_) = List.fold_left
    check_event ([],env) events
    in SAlways(typed_events)

(*Semantic checking on a program*)
let check_program program =
  let (functions,(globals,threads)) = program in
  let env = empty_environment in
  let (typed_functions, new_env) = initialize_functions env functions in
  let (typed_globals, new_env2) = List.fold_left(fun (new_globals,env)
    globals -> initialize_globals (new_globals, env)
    globals) ([],new_env) globals in
  let typed_threads = List.map(fun thread -> check_thread
    new_env2 thread) threads in

  Prog(typed_functions, (typed_globals, typed_threads))

8.6 sast.mli

open Ast
open Type

(* added to work with arrays *)

type scope =
Global
| Local

type sident =
   SIdent of ident * scope

type sval =
   SExprVal of sexpr
| SArrVal of sexpr list

and sexpr =
   SIntLit of int * datatype
| SBoolLit of bool * datatype
| SFloatLit of float * datatype
| SStringLit of string * datatype
| SVariable of sident * datatype
| SUnop of unop * sexpr * datatype
| SBinop of sexpr * binop * sexpr * datatype
   (* changed int to sexpr in SArrElem *)
| SArrElem of sident * sexpr * datatype
| SExprAssign of sident * sexpr * datatype
| SCast of datatype * sexpr * datatype
| SCall of sident * sexpr list * datatype

type sdecl =
   SVarDecl of datatype * sident (* put these inside decl_list for each timeblock *)
   (* changed sexpr to svalue *)
| SVarAssignDecl of datatype * sident * sval (* v_assignment and put v.decl in timeblock decl_list *)

type sstmt =
   SBlock of sstmt list
| SExpr of sexpr
| SReturn of sexpr
| SIf of sexpr * sstmt * sstmt
| SFor of sexpr * sexpr * sexpr * sstmt
| SWhile of sexpr * sstmt
| SDeclaration of sdecl
| SAssign of sident * sexpr
| SArrAssign of sident * sexpr list
(* changed int to sexpr *)
| SArrElemAssign of sident * sexpr * sexpr
| STerminate

type sfuncstr = {
    sreturn: datatype;
    sfname : ident;
    sformals : formal list;
    sbody : sstmt list;
}
type sfunc_decl = 
    SFunc_Decl of sfuncstr * datatype

type sevent = 
    SEvent of int * sstmt list

type sthread = 
    SInit of sevent list
    | SAlways of sevent list

type sprogram = 
    Prog of sfunc_decl list * (sdecl list * sthread list)

8.7 pretty_c_gen.ml

open Ast
open Sast
open Pretty_c

(* ------------ Helpers ------------ *)

(* Counters *)
let incl prefix =
    let count = ref (-1) in
    fun () ->
        incr count;
        prefix ^ string_of_int !count

(* Global counter functions *)
let init_count = incl "init_"
let always_count = incl "always."

(* Main_lists type *)
type main_lists = {
  mutable so: struct_obj list;
  mutable l1: link list;
  mutable l2: link list;
  mutable v.decls: decl list;
}

(* 'Global variable' for lists in main *)
let m_lists = {so=[]; l1=[]; l2=[]; v.decls=[]};

(* ———— Begin Pretty_c_gen functions ———— *)

(* Statement declaration *)
let gen_ident = function
  SIdent(ident, scope) -> ident

let rec gen_expr = function
  SIntLit(i, datatype) -> IntLit(i)
| SBoolLit(b, datatype) -> BoolLit(b)
| SFloatLit(f, datatype) -> FloatLit(f)
| SStringLit(s, datatype) -> StringLit(s)
| SVariable(sident, datatype) -> Variable(gen_ident sident)
| SUnop(unop, sexpr, datatype) -> Unop(unop, gen_expr sexpr)
| SBinop(sexpr, binop, sexpr2, datatype) -> Binop(gen_expr sexpr, binop, gen_expr sexpr2)
| SArrElem(sident, i, datatype) -> ArrElem(gen_ident sident, gen_expr i)
| SExprAssign(sident, sexpr, datatype) -> ExprAssign(gen_ident sident, gen_expr sexpr)
| SCast(datatype, sexpr, datatype2) -> Cast(datatype, gen_expr sexpr)
| SCall(sident, sexpr_list, datatype) -> Call(gen_ident sident, List.map gen_expr sexpr_list)

let gen_val = function
  SExprVal(sexpr) -> ExprVal(gen_expr sexpr)
| SArrVal(sexpr_list) -> ArrVal(List.map gen_expr sexpr_list)

108
let gen_decl = function
    SVarDecl(datatype, sident) → VarDecl(datatype, gen_ident sident)
    | SVarAssignDecl(datatype, sident, sval) → VarAssignDecl(
        datatype, gen_ident sident, (gen_val sval))

let rec gen_stmt = function
    SBlock(stmt_list) → Block(List.map gen_stmt stmt_list)
    | SSExp(sexpr) → Expr(gen_expr sexpr)
    | SReturn(sexpr) → Return(gen_expr sexpr)
    | SIf(sexpr, stmt1, stmt2) → If(gen_expr sexpr, gen_stmt stmt1, gen_stmt stmt2)
    | SFor(sexpr1, sexpr2, sexpr3, stmt) → For(gen_expr sexpr1, gen_expr sexpr2, gen_expr sexpr3, gen_stmt stmt)
    | SWhile(sexpr1, stmt) → While(gen_expr sexpr1, gen_stmt stmt)
    | SDeclaration(sdecl) → Declaration(gen_decl sdecl)
    | SAssign(sident, sexpr) → Assign(gen_ident sident, gen_expr sexpr)
    | SArrAssign(sident, sexpr_list) → ArrAssign(gen_ident sident, List.map gen_expr sexpr_list)
    | SArrElemAssign(sident, i, sexpr) → ArrElemAssign(gen_ident sident, gen_expr i, gen_expr sexpr)
    | STerminate → Terminate

(* Section for special v_decl filtering *)

let rec gen_tb_expr = function
    SIntLit(i, datatype) → IntLit(i)
    | SBoolLit(b, datatype) → BoolLit(b)
    | SFloatLit(f, datatype) → FloatLit(f)
    | SStringLit(s, datatype) → StringLit(s)
    | SVariable(sident, datatype) → Variable(gen_ident sident)
    | SUnop(unop, sexpr, datatype) → Unop(unop, gen_expr sexpr)
    | SBinop(sexpr, binop, sexpr2, datatype) → Binop(gen_tb_expr sexpr, binop, gen_tb_expr sexpr2)
    | SArrElem(sident, i, datatype) → ArrElem(gen_ident sident, gen_expr i)
    | SExprAssign(sident, sexpr, datatype) → ExprAssign(gen_ident sident, gen_tb_expr sexpr)
| SCast (datatype, sexpr, datatype2) -> Cast (datatype, gen_tb_expr sexpr) |
| SCall (sident, sexpr_list, datatype) -> Call (gen_ident sident, List.map gen_tb_expr sexpr_list) |

let gen_tb_decl = function
  SVarDecl (datatype, sident) -> m_lists.v_decls <- VarDecl (datatype, gen_ident sident) :: m_lists.v_decls;
  VarDecl (datatype, gen_ident sident)
| SVarAssignDecl (datatype, sident, sval) -> m_lists.v_decls <-
  VarDecl (datatype, gen_ident sident) :: m_lists.v_decls;
  VarAssignDecl (datatype, gen_ident sident, (gen_val sval)) |

let rec gen_tb_stmt = function
  SBlock (sstmt_list) -> Block (List.map gen_tb_stmt sstmt_list)
| SSExpr (sexpr) -> Expr (gen_tb_expr sexpr)
| SReturn (sexpr-> Return (gen_tb_expr sexpr)
| SIf (sexpr, sstmt1, sstmt2) -> If (gen_tb_expr sexpr, gen_tb_stmt sstmt1, gen_tb_stmt sstmt2)
| SFor (sexpr1, sexpr2, sexpr3, sstmt) -> For (gen_tb_expr sexpr1, gen_tb_expr sexpr2, gen_tb_expr sexpr3, gen_tb_stmt sstmt)
| SWhile (sexpr1, sstmt) -> While (gen_tb_expr sexpr1, gen_tb_stmt sstmt)
| SDeclaration (sdecl) -> Declaration (gen_tb_decl sdecl)
| SAssign (sident, sexpr) -> Assign (gen_ident sident, gen_tb_expr sexpr)
| SArrAssign (sident, sexpr_list) -> ArrAssign (gen_ident sident, List.map gen_tb_expr sexpr_list)
| SArrElemAssign (sident, i, sexpr) -> ArrElemAssign (gen_ident sident, gen_expr i, gen_tb_expr sexpr)
| STerminate -> Terminate |

let gen_tb_vdecls = function
  SEvent (delay, sstmt_list) -> List.map gen_tb_stmt sstmt_list

(* Receives the link, creates a current name (i.e. init_0_block_1). *)

110
Then creates the time struct with the name, the delay from event *)
let gen_structure curr_name_f link = function
SEvent(delay, stmt_list) -> let name = Time_struct_name(curr_name_f ()) in (* Counter *)
    m_lists.so <- Time_struct_obj(name, link) :: m_lists.so; (* adding to main list *)
    Time_struct(name, delay, link, List.rev stmt_list)

(* Receives sthread, creates Link for init or always, sends this link as parameter for gen_structure function *)
let gen_time_block = function
SInit(sthread) -> m_lists.v_decls <- []; (* Reset v_decl list *)
    let curr_link_str = init_count () in let curr_name_f = incl (curr_link_str ^ "_block") in (*Counters*)
    let link = Link(curr_link_str) in m_lists.l1 <- link :: m_lists.l1; (*adding to main list *)
    let partial_gen_struct = gen_structure curr_name_f link in
    let _ = List.map gen_tb_vdecls sthread in
    Time_block(link, m_lists.v_decls, List.map partial_gen_struct (List.rev sthread)) | SAlways(sthread) -> m_lists.v_decls <- []; (* Reset v_decl list *)
    let curr_link_str = always_count () in let curr_name_f = incl (curr_link_str ^ "_block") in (*Counters*)
    let link = Link(curr_link_str) in m_lists.l2 <- link :: m_lists.l2; (*adding to main list *)
    let partial_gen_struct = gen_structure curr_name_f link in
    let _ = List.map gen_tb_vdecls sthread in
    Time_block(link, m_lists.v_decls, List.map partial_gen_struct (List.rev sthread))

let gen_func = function
SFuncDecl(sfuncstr, datatype) -> {return=sfuncstr.sreturn; fname=sfuncstr.fname;
formals=sfuncstr.sformals;
    body= List.map gen_stmt ( List.rev sfuncstr.sbody))

(* Main function to generate a pretty_c *)
let gen_pretty_c = function
    Prog(sfunc_decl_list, (sdecl_list, sthread_list)) ->
    let func_list = List.map gen_func sfunc_decl_list in (*
        Functions *)
    let decl_list = List.map gen_decl sdecl_list in (*
        Declarations *)
    let time_block_list = List.map gen_time_block (List.rev
        sthread_list) in (* Time Blocks *)
    let main = Main(List.rev m_lists.so, List.rev m_lists.ll,
        List.rev m_lists.l12) in (* Main *)
    Pretty_c(List.rev decl_list, List.rev func_list,
        time_block_list, main) (* Pretty_c *)

8.8 gen_cpp.ml

open Ast
open Sast
open Printf
open Pretty_c
open Type

let print = "print"
let get_time = "print_time"
let prefix_array = "a_"
let prefix_global_var = "u_"
let prefix_event = "event"
let prefix_event_list = "event_q"
let code_event_base = "struct " prefix_event^ " {
" \n\t unsboxed int time; \n\t unsboxed int inc_time; \n\t std:: \\
\t string name; \n\t }" ^ " virtual unsigned int get_time() \n\t virtual unsigned int get_inc_time() \n\t virtual void (set_time)(unsigned int time_) \n\t virtual std::string get_name() \n\t virtual void foo() \n\t virtual ^" prefix_event " ^"() \n\t };
\n"
let code_event_list = "struct "^ prefix_event_list ^
    {\n    bool empty() {return event_q.empty();}\n    "unsigned int get_time() {return global_time;}\n
prefix_event "^ "* pop() {\n    "^ "*front = event_q.front();\n
    "global_time = front->get_time();\n    "event_q.pop_front();\n    "^ "return front;\n
    "^ "void add(unsigned int time_"^ "prefix_event ^
    "*obj_) {\n        eol = true;\n        std::deque<"^ prefix_event ^
        prefix_event "^ "*>::iterator it;\n        if (obj_ == NULL) {\n            return ;\n        }\n
        for (it = event_q.begin(); it != event_q.end(); it++) {\n            if ((\n                "^ "^ "event_q.insert(it, obj_);\n                "^ "^ "teol = false;\n                "^ "^ "break ;\n            ) ;\n
            "^ "event_q.push_back(obj_);\n
            "^ "event_q.push_back(obj_);\n
            "^ "private:;\n            "^ "unsigned int global_time;\n    } \n
    "^ "let header = code_directives^code_event_base^code_event_list ^
    "^ "let gen_id = function
        Ident(id) -> id

    let gen_sid_prefix scope lcl_prefix = match scope with
        Global -> prefix_global_var
    | Local  -> lcl_prefix

    let gen_sid sident lcl_prefix = match sident with
        SIdent(sid, scope) -> gen_sid_prefix scope lcl_prefix ^ gen_id sid

    let gen_plain_sid sident = match sident with
        SIdent(sid, scope) -> gen_id sid

113
let gen_name = function
    Time_struct_name(s) -> s

let gen_link = function
    Link(s) -> s

let gen_unop = function
    Neg -> "-"
    Inc -> "++"
    Dec -> "--"
    Not -> "!"

let gen_binop = function
    Add -> "+
    Sub -> "-
    Mult -> "*
    Div -> "/
    Equal -> "=="
    Neq -> "!="
    Less -> "<"
    Leq -> "<="
    Greater -> ">"
    Geq -> ">="
    Mod -> "%
    And -> ">
    Or -> "||"

let gen_var_type = function
    Int -> "int"
    Float -> "float"
    String -> "std::string"
    Boolean -> "bool"
    Void -> "void"

let gen_plain_var_type = function
    Int -> "int"
    Float -> "float"
    String -> "char *"
    Boolean -> "bool"
    Void -> "void"
let rec gen_datatype = function
  Datatype(var_type) -> gen_var_type var_type
| Arraytype(datatype) -> "std::vector<" ~ gen_datatype datatype ~ ">

let rec gen_plain_datatype = function
  Datatype(var_type) -> gen_plain_var_type var_type
| Arraytype(datatype) -> gen_plain_datatype datatype

let rec gen_formal formal prefix = match formal with
  Formal(datatype, id) -> gen_datatype datatype ~ " " ~ prefix ~ gen_id id

let rec gen_sexp sexpr lcl_prefix = match sexpr with
  SIntLit(i, d) -> string_of_int i
| SBoolLit(b, d) -> string_of_bool b
| SFloatLit(f, d) -> string_of_float f
| SStringLit(s, d) -> "" ~ s ~ ""
| SVariable(sident, d) -> gen_sid sident lcl_prefix
| SUnop(unop, sexpr, d) -> gen_unop unop ~ "(" ~ gen_sexp sexpr ~ lcl_prefix ~ ")"
| SBinop(sexpr1, binop, sexpr2, d) -> gen_sexp sexpr1 lcl_prefix ~ genbinop binop ~
  gen_sexp sexpr2 lcl_prefix
| SArrElem(sident, i, d) -> gen_sid sident lcl_prefix ~ "[" ~ gen_sexp i lcl_prefix ~ "]"
| SExprAssign(sident, sexpr, d) -> gen_sid sident lcl_prefix ~ " =
  ~ gen_sexp sexpr lcl_prefix
| SCast(datatype, sexpr, d) -> ")" ~ gen_datatype datatype ~ ")"
  ~ gen_sexp sexpr lcl_prefix
| SCall(sident, sexpr_list, d) -> if ((gen_plain_sid sident) = print)
  then "std::cout << " ~ gen_sexp_list sexpr_list lcl_prefix ~ ") << std::endl"
  else begin
  if ((gen_plain_sid sident) = get_time) then
  "std::cout << "Time now: \" ~ event_q.get_time() ~ std :: endl"
else
gensid sident lcl_prefix ^ "(" ^ gen_expr_list
expr_list lcl_prefix ^ ")""
end

and gen_expr expr prefix = match expr with
  IntLit(i) -> string_of_int i
| BoolLit(b) -> string_of_bool b
| FloatLit(f) -> string_of_float f
| StringLit(s) -> \\
  "\"" ^ s ^ "\"
| Variable(ident) -> prefix ^ gen_id ident
| Unop(unop, expr) -> gen_unop unop ^ "(" ^ gen_expr expr prefix ^ "")"
| Binop(expr1, binop, expr2) -> gen_expr expr1 prefix ^ gen_binop
  binop ^
  gen_expr expr2 prefix
| ArrElem(ident, i) -> prefix ^ gen_id ident ^ "[" ^ gen_expr i
  prefix ^ "]"
| ExprAssign(ident, expr) -> prefix ^ gen_id ident ^ "= " ^
  gen_expr expr prefix
| Cast(datatype, expr) -> "(" ^ gen_datatype datatype ^ ")" ^
  gen_expr expr prefix
| Call(ident, expr_list) -> if ((gen_id ident) = print)
  then "std::cout << " ^ gen_expr_list expr_list prefix ^ " <<
    std::endl"
  else prefix ^ gen_id ident ^ "(" ^ gen_expr_list expr_list
  prefix ^ ")"

and gen_sstmt sstmt lcl_prefix = match sstmt with
  SBlock(sstmt_list) -> \\
  "{" ^ gen_sstmt_list sstmt_list
  lcl_prefix ^ "}\n\t"
| SSEExpr(sexpr) -> gen_expr sexpr lcl_prefix ^ " sixteen
  \n\t"
| SReturn(sexpr) -> "return " ^ gen_expr sexpr lcl_prefix ^ "\n\t"
| SIf(sexpr, sstmt1, sstmt2) -> "if (" ^ gen_expr sexpr
  lcl_prefix ^ "\t")\n\t" ^ gen_sstmt sstmt1 lcl_prefix ^ " sixteen\n\telse " ^
  gen_sstmt sstmt2 lcl_prefix
| SFor(sexpr1, sexpr2, sexpr3, sstmt) -> "for (" ^ gen_expr
  sexpr1 lcl_prefix ^ " sixteen; " ^ gen_expr sexpr2
  lcl_prefix ^ " sixteen; " ^ gen_expr sexpr3
  lcl_prefix ^ " sixteen)\n\t"
gen_sstmt sstmt lcl_prefix
| SWhile(sexpr, sstmt) -> "while (" \ gen_sexpr sexpr lcl_prefix ")\n"  \
|   "\n" \ gen_sstmt sstmt lcl_prefix ");\n\t"
| SDeclaration(sdecl) -> gen_sdecl sdecl lcl_prefix ");\n\t"
| SAssign(sident, sexpr) -> gen_sid sident lcl_prefix ");\n\t"
| gen_sexpr sexpr lcl_prefix ");\n\t"
| SArrAssign(sident, sexpr_list) -> gen_sid sident lcl_prefix ");\n\t"
| gen_array_sexpr_list sexpr_list sident lcl_prefix ");\n\t"
| SArrElemAssign(sident, i, sexpr) -> gen_sid sident lcl_prefix  \
|   ";\n\t"
| gen_array_sexpr_list sexpr_list i lcl_prefix  ";\n\t"
| STerminate -> "exit(0);\n\t"

(*semicolon and newline handled in genStmt because blocks dont  
 have semicolon*)

and gen_sstmt stmt prefix = match stmt with
| Block(stmt_list) -> "{\n\t" \ genStmt_list stmt_list prefix ");\n\t"
| Expr(expr) -> gen_expr expr prefix ");\n\t"
| Return(expr) -> "return " \ gen_expr expr prefix ");\n\t"
| If(expr, stmt1, stmt2) -> "if (" \ gen_expr expr prefix ");\n\t"
| genStmt stmt1 prefix ");\n\telse " \ genStmt stmt2 prefix  \
| For(expr1, expr2, expr3, stmt) -> "for (" \ gen_expr expr1  \
| prefix ");\n\t"
| gen_expr expr2 prefix ");\n\t" \ gen_expr expr3 prefix ");\n\t"

genStmt stmt prefix  
| While(expr, stmt) -> "while (" \ gen_expr expr prefix ");\n\t"
| Declaration(decl) -> gen_decl decl prefix ");\n\t"
| Assign(ident, expr) -> prefix \ gen_id ident ");\n\t"
| gen_expr expr prefix ");\n\t"
| ArrAssign(ident, expr_list) -> prefix \ gen_id ident  \
| \ clear ();\n\t"
| gen_array_expr_list expr_list ident prefix ");\n\t"
| ArrElemAssign(ident, i, expr) -> prefix \ gen_id ident  \

117
"[" ˆ gen_expr i prefix ^ "] = " ˆ gen_expr expr prefix ^ ";\n\nt"
| Terminate -> "exit(0);\n\n(*gen_sdecl only appears within time blocks, VarDecls are ignored *)
and gen_sdecl sdecl lcl_prefix = match sdecl with
  SVarDecl(datatype, sid) -> ""
| SVarAssignDecl(datatype, sident, svalue) -> gen_svalue datatype svalue sident lcl_prefix

(*gen_svalue only appears within time blocks declarations, assume all local*)
and gen_svalue datatype svalue sident lcl_prefix = match svalue with
  SExprVal(sexpr) -> lcl_prefix ^ gen_plain_sid sident ^ 
                  " = " ˆ gen_sexpr sexpr lcl_prefix ^ ";\n\n  | SArrVal(sexpr_list) -> gen_sid sident lcl_prefix ^ ".clear();\n\n                  (gen_array_sexpr_list sexpr_list sident lcl_prefix) ^ ";\n\n(*semicolon and newline handled in gen_decl since array decl assignment is actually vector push_back*)
and gen_decl decl prefix = match decl with
  VarDecl(datatype, id) -> gen_datatype datatype ^ 
                           " " ^ prefix ^ 
                          gen_id id ^ "”;\n\n  | VarAssignDecl(datatype, ident, value) -> gen_value datatype value ident prefix

and gen_value datatype value ident prefix = match value with
  ExprVal(expr) -> gen_datatype datatype ^ 
                  " " ^ prefix ^ gen_id ident ^ "" = " ˆ gen_expr expr prefix ^ ";\n\n  | ArrVal(sexpr_list) -> "const " ˆ gen_plain_datatype datatype ^ 
                        " "] = {" ˆ gen_expr_list 
                       expr_list prefix ^ 
                       "};\n\n         prefix_array ^ gen_id ident ^ "[0] = 
                  gen_datatype datatype ^ 
                  prefix ^ gen_id ident ^ "( " ^ 
                  prefix_array ^ gen_id ident ^ "", " ^ 
                  prefix_array ^ gen_id ident ^ "+sizeof(" ^ 
                  prefix_array ^ gen_id ident ^ 
                  ")/sizeof(" ^ 
                  prefix_array ^ gen_id ident ^ "[0]) });\n\n118
and gen_array_expr_list sexpr_list sident lcl_prefix = match sexpr_list with
[] -> ""
| h::[] -> gen_sid sident lcl_prefix ^ ".push_back(" ^ gen_expr h lcl_prefix ")\n"
| h::t -> gen_sid sident lcl_prefix ^ ".push_back(" ^ gen_expr h lcl_prefix ")\n" ^ (gen_array_expr_list t sident lcl_prefix)

and gen_array_expr_list expr_list ident prefix = match expr_list with
[] -> ""
| h::[] -> prefix ^ gen_id ident ^ ".push_back(" ^ gen_expr h prefix ")\n"
| h::t -> prefix ^ gen_id ident ^ ".push_back(" ^ gen_expr h prefix ")\n" ^ (gen_array_expr_list t ident prefix)

and gen_func func prefix =
  gen_datatype func.return ^ " " ^ prefix ^ gen_id func.fname ^ "(" ^ gen_formal_list func.formals prefix ^ ")\n" {\n" ^ gen_stmt_list func.body prefix ^ "}\n"

and gen_decl_list decl_list prefix = match decl_list with
[] -> ""
| h::[] -> gen_decl h prefix
| h::t -> gen_decl h prefix ^ gen_decl_list t prefix

and gen_func_list func_list prefix = match func_list with
[] -> ""
| h::[] -> gen_func h prefix
| h::t -> gen_func h prefix ^ gen_func_list t prefix

and gen_formal_list formal_list prefix = match formal_list with
[] -> ""
| h::[] -> gen_formal h prefix
| h::t -> gen_formal h prefix ^ ", " ^ gen_formal_list t prefix

and gen_stmt_list stmt_list lcl_prefix = match stmt_list with
[] -> ""
| h::[] → gen_sstmt h lcl_prefix  
| h::t → gen_sstmt h lcl_prefix ^ gen_sstmt_list t lcl_prefix  

and gen_sstmt_list stmt_list prefix = match stmt_list with  
[[]] → ""
| h::[] → gen_stmt h prefix
| h::t → gen_stmt h prefix ^ gen_sstmt_list t prefix

and gen_sexpr_list sexpr_list lcl_prefix = match sexpr_list with  
[[]] → ""
| h::[] → gen_sexpr h lcl_prefix
| h::t → gen_sexpr h lcl_prefix ^ "," ^ gen_sexpr_list t lcl_prefix

and gen_expr_list expr_list prefix = match expr_list with  
[[]] → ""
| h::[] → gen_expr h prefix
| h::t → gen_expr h prefix ^ "," ^ gen_expr_list t prefix

let gen_time_block_header link =  
"unsigned int " ^ link ^ ", time = 0;\nstruct " ^ link ^ 
"_link_ : public event_ {\n\tvirtual void set_next(" ^ link ^ 
"_link_ *n)\n};\n};\n\nstd::vector<" ^ link ^ 
"_link_*> " ^ link ^ 
"_list ;\n"

let rec gen_struct = function  
      Time_struct(name, i, link, stmt_list) → "struct " ^ gen_name name ^ 
        " : public " ^ gen_link link ^ 
        "_link_ {\n\tunsigned int time \\
\tn};\n"  
    "unsigned int inc_time;\n\tstd::string name;\n\tn" ^ 
    gen_name name ^ "() : inc_time(" ^ 
    string_of_int i ^ " ) " ^ 
    ", time(" ^ string_of_int i ^ 
    ") , name(" ^ 
    "gen_name name ^ 
    "\") \n\tunsigned int get_time() {return time;}\n\tn" ^ 
    "unsigned int get_inc_time() {return inc_time;}\n\tn" ^ 
    "void set_time(unsigned int time_){time = time_;}\n\tn" ^ 
    "std::string get_name() {return name;}\n\tn" ^ 
    "_link_ *next;\n\tvoid set_next(" ^ gen_link link ^ 
    "_link_ *n) {next = n;}\n\tn" ^ "void foo() \n"
gen_stmt_list stmt_list (gen_link link) ^ "\n"tif(next != NULL) {\n\n  time += next->get_inc_time();\n\n  next->set_time(" ^ gen_link link ^ ", time);\n  "event_q.add(" ^ gen_link link ^ ", time, next);\n\n  };

and gen_struct_list struct_list = match struct_list with
  [ ] -> ""
  | h::[] -> gen_struct h ^ "\n"
  | h::t -> gen_struct h ^ "\n" ^ gen_struct_list t

let rec gen_time_block = function
  Time_block(link, decl_list, struct_list) ->
  gen_decl_list decl_list (gen_link link) ^
  gen_time_block_header (gen_link link) ^
  gen_struct_list struct_list

and gen_time_block_list = function
  [ ] -> ""
  | h::[] -> gen_time_block h
  | h::t -> gen_time_block h ^ "\n" ^ gen_time_block_list t

let gen_struct_obj = function
  Time_struct_obj(name, link) -> gen_name name ^ " " ^ gen_name
  name ^ "obj;\n\n  gen_link link ^ "_list.push_back(&" ^ gen_name name ^ "obj);\n  \n\nlet gen_init_linker = function
  Link(s) -> "for (int i = 0; i < " ^ s ^ "_list.size(); i++)\n  "^ 
    "{\n\n    tif (i != " ^ s ^ "_list.size()-1)\n\n    s ^ "_list[i]->set_next(" ^ s ^ "_list[i+1]);\n\n    telse\n\n    s ^ "_list[i]->set_next(NULL);\n\n    "event_q.add(" ^ s ^ "_block_0obj.get_time(), &" ^ s ^ " _block_0obj);\n    "\n
let gen_always_linker = function
Link(s) -> "for (int i = 0; i < " ^ s ^ " .list.size(); i++)\n  " ^ s ^ " .list.size()-1)\n  s ^ " .list[i]-->set_next(" ^ s ^ " .list[i+1]);\n  t" ^
  s ^ " .list[i]-->set_next(" ^ s ^ " .list[0]);\n  t" ^
"event_q.add(" ^ s ^ " _block_0obj.get_time(), &" ^ s ^ "
  _block_0obj);\n"

let rec gen_init_linker_list = function
  [] -> ""
  | h::[] -> gen_init_linker h
  | h::t -> gen_init_linker h ^ gen_init_linker_list t

let rec gen_always_linker_list = function
  [] -> ""
  | h::[] -> gen_always_linker h
  | h::t -> gen_always_linker h ^ gen_always_linker_list t

let rec gen_struct_obj_list = function
  [] -> ""
  | h::[] -> gen_struct_obj h
  | h::t -> gen_struct_obj h ^ gen_struct_obj_list t

let rec gen_event_q_add_list = function
  [] -> ""
  | h::[] -> gen_event_q_add h
  | h::t -> gen_event_q_add h ^ gen_event_q_add_list t

and gen_event_q_add = function
  Time_struct_obj(name, link) -> "event_q.add(" ^ gen_name name ^
  "obj.get_time(), &" ^ gen_name name ^ "obj);\n"

(*all arguments are lists*)
let gen_main = function
  Main(time_block_obj_l, init_link_l, always_link_l) ->
  gen_struct_obj_list time_block_obj_l ^
  gen_init_linker_list init_link_l ^
  gen_always_linker_list always_link_l

let gen_program = function
Pretty_c (global_decl_list, global_func_list, time_block_list, main) →
header "
gen_func_list global_func_list prefix_global_var "
gen_decl_list global_decl_list prefix_global_var "
gen_time_block_list time_block_list "\nint main() {\n  "
gen_main main "
code_event_list_do "return 0;\n}"

8.9 compiler.ml

open Semantic_check
open Pretty_c_gen
open Gen_cpp

let _ =
  letlexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  let sast = Semantic_check.check_program ast in
  let c_sast = gen_pretty_c sast in
  let code = gen_program c_sast in
  let output = open_out "output.cpp" in
  output_string output code