Module Statement and Automatic Pallene to Lua Translator

Samuel Rowe
samuelrowe1999@gmail.com
+91 8310843835
Bengaluru, India

BASICS

What is your preferred email address?
samuelrowe1999@gmail.com

What is your web page / blog / github?
GitHub: https://github.com/itssamuelrowe
Web Page: https://itssamuelrowe.github.io
Blogs: https://medium.com/@itssamuelrowe | https://itssamuelrowe.github.io/binarystars | https://itssamuelrowe.github.io/zen-website

What is your academic background?
B.Tech in Computer Science and Engineering, sixth semester, Presidency University Bangalore

What other time commitments, such as school work, another job (GSoC is a full-time activity!), or planned vacations will you have during the period of GSoC?
I have holidays. I am available during the period of GSoC.

EXPERIENCE

What programming languages are you fluent in?
C, C++, Java, Python, JavaScript, PHP

Which tools do you normally use for development?
CMake, GNU Toolchain, CPython, JDK, NodeJS, Visual Studio Code, Notepad++, Windows Terminal, WSL, MSYS2, Git, GitHub

Are you familiar with the Lua programming language? Have you developed any projects using Lua?
I skimmed through the Lua documentation and the codebase long back when I was designing the grammar for my programming language, Zen. Technically, I do not have any experience with Lua or its C API except for writing a few examples. I am currently learning Lua from the book “Programming in Lua, Fourth Edition” by Roberto Ierusalimsch.

Have you developed software in a team environment before?
Yes, I have worked on a few projects with my friend.

Any projects with actual users?
Yes. I developed a Property Management System (PMS) for a local lodge. Apart from that, I have developed a few websites as a freelancer.

What kinds of projects/software have you worked on previously? (anything larger than a class project: academic research, internships, freelance, hobby projects, etc.)
My biggest projects so far deal with compiler design, virtual machines, system programming, and web applications.

The following projects have provided me with relevant experience for contributing to Pallene.
Zen: A general purpose programming language designed to build simple, reliable and efficient programs. The project includes a compiler and a reference implementation of the Zen Virtual Machine (ZVM), which will be powered by Eclipse OMR.
(https://github.com/itssamuelrowe/zen, https://github.com/itssamuelrowe/pulsarvm)

JTK: A library designed for writing applications and libraries in C, which provides core utilities such as collections, unit testing, I/O streams, threads and much more.
(https://github.com/itssamuelrowe/jtk)

Are you (or have you been) involved with any open source development project? If so, briefly describe the project and the scope of your involvement.
Yes, I have been involved in open source projects. However, all the open source projects I have contributed to were created by me.

PROJECT

Did you select a project from our list? If yes, which project did you select?
From the list of projects suggested by my mentors, Gabriel and Hugo, I found issue 173 (https://github.com/pallene-lang/pallene/issues/173) the most interesting. The issue is titled "*It would be nice to have an automatic way to convert Pallene into Lua.*"

Why did you choose this project?
I have been learning about compiler design since 9th grade. I have worked on a code generator before (AST to bytecode), that is, for my programming language Zen. Therefore, I found this project to be the most relevant with respect to my experience and knowledge.

If you are proposing a project, give a description of your proposal, including the expected results. Please provide a schedule with dates and important milestones/deliverables, in two week increments).

Summary

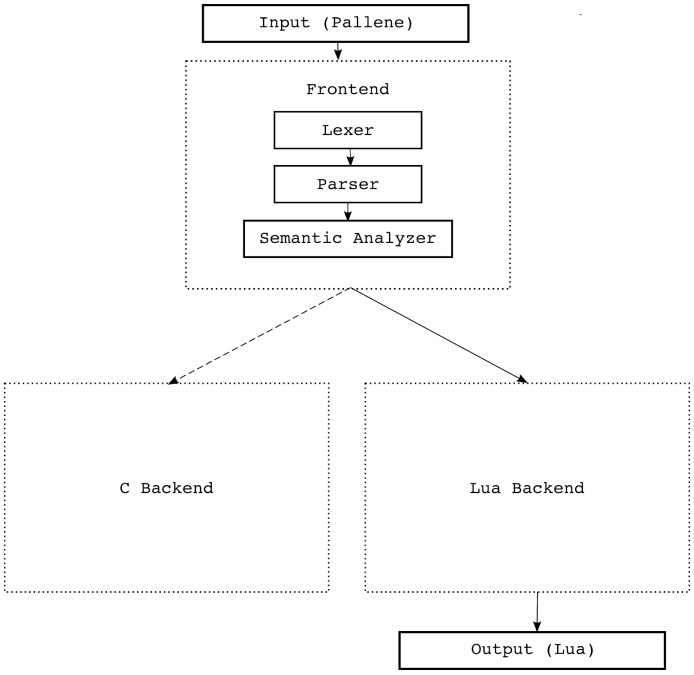
Pallene to Lua translator
Pallene is a statically typed, ahead-of-time-compiled sister language to Lua, with a focus on performance. Programmers can use Pallene instead of C modules and LuaJIT to improve and gain a predictable run-time performance. Pallene achieves this by translating a statically typed compilation unit into a C source file, which is implicitly compiled in the background.

However, there are situations where removal of type annotations are useful.

- The main premise of Pallene is that it is compatible with Lua. This compatibility is defined as removing type annotations from a Pallene program results in transforming it into a Lua program. A Pallene to Lua translator will allow us to check whether this property is still valid.
- Provides greater portability, interoperability and integration with existing Lua codebase and tools.
- The Pallene developers could check if the unit tests are obeying the "gradual guarantee".
- In the benchmarks that do not use LuaJIT features, we could generate the Lua version of the code from the Pallene version.

With the help of the Pallene to Lua translator, users can remove Pallene type annotations to generate plain Lua.

Pallene type annotations can be removed by extending the backend of the Pallene compiler to generate Lua.



The architecture of the Pallene compiler with decoupled frontend and backend.

Consider the following example written in Pallene.

```
function sum(values : { float }): float
    local s : float = 0.0
    for i = 1, #values do
        s = s + values[i]
    end
    return s
end
```

The Pallene to Lua translator should produce the following Lua program.

```
function sum(values)
    local s = 0.0
    for i = 1, #values do
        s = s + values[i]
    end
    return s
end
```

Module statement
A module is some part of your source code that can be imported from another source file. Internally, the runtime creates and returns a table. Everything that the module exports, such as functions and constants, is defined inside this table. This allows each module to have its own namespace.

Consider the following example written in Pallene:

```
module m
function m.add(a : integer, b : integer) : integer
    return a + b
end
return m
```

The Lua translator would produce the following output:

```
local m = {}
function m.add(a, b)
    return a + b
end
return m
```

The programmer can later import the module into his/her own source file. The functions and other components exported by the module can be accessed using the dot notation.

Implementation

This project consists of two components: **module statement and Pallene AST to Lua backend.**

Pallene AST to Lua backend
The Pallene compiler is divided into two logical ends:

- The frontend which parses Pallene source code to generate AST and performs semantic analysis.
- The backend which generates C source code.

Both these ends are decoupled, this provides us with the flexibility to integrate another backend that generates Lua source code. The users can then run the compiler with the flag `--lua`, or `--target=lua` to cause the compiler to execute the frontend and then trigger the backend that generates plain Lua instead of C.

The generation of Lua source code is performed by a new backend. It accepts input in the form of an AST generated by the parser. The generator then walks over the AST to generate output with print statements, or something equivalent. The advantage of creating an AST first is that semantic analysis can be performed before generating the output.

Module statement
A compilation unit can be included in a module with the special module declaration statement at the top of the source file. This statement would be compiled to Lua as `local m = {}`. This allows the Pallene compiler to optimize the calls to local module functions like it does now.

Programmers can call functions declared within a module using the dot notation which will be converted with the proper module prefix. This could be the first step to implementing the module statement because we already have the built-in functions loaded in the symbol table to test with. For instance this would allow us to write `io.write` instead of `io_write`. The translation from `io.write` to the actual write function would happen inside the checker pass (the checker.lua file).

Deliverables

- Automatic test scripts
- A backend that generates plain Lua source code for a given Pallene AST.
- Detailed documentation and tests for the above mentioned component.
- Updated benchmarks makefile that generates the Lua code automatically based on Pallene code.
- The compiler extended to process both the semantics and the syntax of the module statement.
- Examples demonstrating the module statement.

Timeline

The following timeline with one-week increments provides a rough guideline of how the project will be executed.

NOTE: As suggested by my mentors, a test driven approach will be taken in implementing the requirements of this project. Unit tests will be written to verify each feature in its corresponding week as given in the timeline.

May 04 – May 31 (before the coding period begins)
Familiarize myself with Lua and Pallene programming languages, Pallene codebase, coding standards followed by the community, read the documentation and understand the unit tests. Further, I will be reading the book Programming in Lua (4th edition) by Roberto Ierusalimschy and the dissertation by Gabriel de Quadros Ligneul (https://github.com/gligneul/Publications/blob/master/the_implementation_of_records_in_pallene.pdf).

Module Statement

June 01 – June 07, June 08 – June 14

- **Add the dot notation to the builtin functions.** For this task, an abstraction representing the builtin module will be created in the checker phase. The builtin functions will be inserted to this module abstraction. After which, the module abstraction will be registered in the symbol table. During translation the module function calls will be linked to the actual builtin functions.
  Currently, Pallene supports `io_write`, `math_sqrt`, `string_char`, `string_sub`, `tofloat`, and `type` builtin functions. After the completion of this task, users will be able to write `io.write`, `math.sqrt`, `string.char`, and `string.sub`.

June 15 – June 21

- **Document the module statement.**
- **Extend Pallene grammar to include the module declaration statement.**

June 22 – June 28

- **Extend the Pallene parser and the AST to include the module declaration statement.**
- **Extend the checker phase to perform semantic analysis.** When a module is declared, a module abstraction is created and inserted into the symbol table by the checker. Any function which is declared within a module, will be inserted into the module abstraction. During translation special care is taken to link function calls appropriately.

Pallene to Lua backend

June 29 – July 05

- Document the new architecture of the Pallene compiler, the Lua code generator, and the internals.
- Extend the command line parser of Pallene to include the `--target=lua` or `--lua` flag.

July 06 – July 12

- **Implement a pretty printer which reformats generated Lua constructs.** This allows us to produce readable Lua output without having to worry about indentation while we are generating it. Please note that the pretty printer is intended for internal use by the Lua backend.
  Pallene AST -> [Pallene to Lua Translator -> Pretty Printer] -> Output

  The pretty printer is invoked for each construct, where a construct is one of: simple statement, compound statement, primary expression (for formatting tables and other

such expressions), or function. Internal states are maintained to keep track of indentations.

July 13 – July 19

- **Integrate another backend (dummy)** which should be triggered by the `--target=lua` or `--lua` flag. The dummy backend receives the AST and outputs any empty file. The missing parts will be incrementally implemented in the following weeks.
- Implement the translator to generate Lua for top level records and variables.

July 20 – July 26

- Implement the translator to generate Lua for top level functions, blocks, and variables.

July 27 – August 02

- Implement the translator to generate Lua for do, while, and repeat statements.

August 03 – August 09

- Implement the translator to generate Lua for if, for, and local variable declaration statements.

August 10 – August 16

- Implement the translator to generate Lua for return statements and expressions.

August 17 – August 23

- Extend the Lua backend to generate corresponding Lua statements for the module system.
- Modify the benchmarks makefile to generate the Lua source files based on the Pallene source files.

GSOC

Have you participated as a student in GSoC before? If so, How many times, which year, which project?
No, I have not participated in GSoC before.

Have you applied but were not selected? When?
No, I have not applied for GSoC before.

Did you apply this year to any other organizations?
No, I am not applying to any other organizations this year.