JASMINT:

Language to User-Friendly AST with Emphasis on Transpilation

By: John Bradbury

Advisor: John Clements

Computer Science Department

College of Engineering

California Polytechnic State University, San Luis Obispo

June 16, 2018

Table of Contents

ABSTRACT

The goal of this project was to create a language (JASMINT) which would be easily transformable into other languages. With this, a library could be built which provides a rich set of functions, including typechecking, interpreting, and serialization, in order to make user modules easy to write. These modules are able to translate this AST into other languages and through the translation blocks can add new functionalities to JASMINT. The final state of the project at submission includes a library which handles all features except dynamic memory, transpilers (JasmintCxxTranspiler and JasmintPythonTranspiler) which handle most features except classes and dynamic memory, and a default module (JasmintUI) which allows for basic interaction with the library.

Project Overview

Background

During the life of a long term project, the programmer often finds themselves wanting such tools and must find them for the language of their project. Unfortunately, the success of this search often depends on the popularity and/or the complexity of the language. Creators of the tools themselves must oftentimes re-parse the language, and bonuses such as a type-checked AST are practically impossible, especially if the language is dynamically typed.

This project was built with the purpose of making the programmer's life easier. With that said, JASMINT seeks to provide a high-level language through a library packed with convenience functions which allow users to easily work with the Abstract Syntax Tree (AST) generated from this new language and to create their own tools such as transpilers, code style checkers, optimizers, static code analyzers, ect.

Implementation History

The project originally began as a C++ project, and the entire first quarter was spent working on this project of ~10kLoc. However, several critical design mistakes were made due to a lack of knowledge, and C++ itself introduced more difficulty with memory management in the presence of complex circular references (closures and environments). Due to this, at the beginning of the second quarter, the entire project was rewritten in Java and all further improvements were made to the Java version.

Requirements

At the beginning of the project, the requirements were as follows: parser/interpreter for the JASMINT language including functions with overloads and classes with inheritance, serialization functionality for the AST, type-checking with inferencing, multiple source files, a unit testing framework in JASMINT for JASMINT, and a way to access target language features from JASMINT. Over the course of the project, the requirements had to be a bit adjusted due to time constraints, the overall difficulty of certain features, and the replacement of some features with more meaningful ones.

The actual current form of JASMINT can be found in its ANTLR4 grammar file (an EBNF), and all features are implemented in the interpreter, although not all are present in the transpilers due to lack of time.

System Architecture and Features

Project Organization

In the C++ version, the entire project was stuffed into a single git repository and generally regarded as a single project. Eventually, it became painfully obvious that it was actually FOUR separate projects:

1. Library (JASMINT) – Provides AST functions such as parsing, serialization/deserialization, typechecking, interpreting, and other convenience functions.
2. Basic UI – Provides a default way to interact with the JASMINT library and perform actions such as loading, executing, and serializing a file.
3. Transpiler 1 (Python) – Takes a serialized AST and outputs equivalent python code.
4. Transpiler 2 (C++) – Takes a serialized AST and outputs equivalent C++ code.

If used in the real world, a user would download the JASMINT library and then create their own project to use that lib in its own repository, and so, in the rewrite to Java, each of these four projects was put in their own repository.

Lexing/Parsing

In the C++ version, the lexer and parser were written by hand. This was done primarily for the experience and initial joy of writing a parser. However, joy quickly turned into despair. A handwritten parser is almost certainly buggier than a generated one, and a grammar file is much nicer to read and modify. In addition, a grammar file doubles as documentation since by definition it must be a correct EBNF. Along with removing type inferencing and avoiding having to manage memory correctly in the presence of circular referencing, a primary reason to switch to Java was for the excellent Antlr4 parser generator. Antlr can also be used with other languages, but regardless of the language, its versatile grammar file can really keep people sane about how the language actually works.

Pointers

Since the language is meant to be transpilable, it was designed to support a broad set of features and includes memory management (with new and delete). It also seemed to make sense to support pointers. However, while high level languages with garbage collection could just ignore the delete command or just mark the object as invalid, it turns out to be significantly trickier to transpile pointers into a language like Python. At a fundamental level, pointers are useful for addressing directly into memory, but are not essential for anything else. JASMINT cannot address directly into memory, and so would have a difficult time performing a task like zipping a file where direct control over bits is needed. It could be done through adept usage of ‘trans’ blocks, but while C++ would have no problem providing the needed functionality, Python, and many other high level languages, would not take to it naturally. Because of these reasons, pointers were removed from the language during the switch to Java.

Type Inferencing

It has been mentioned a few times that type inferencing was removed, but no explanation has been given as to what it was. The time has come to remedy that. Initially, the idea was to create a language that would appear to be dynamically typed and would be easy to write but difficult to compile, the argument being that compilation happens relatively rarely. The C++ version still supports this feature, so any interested party could at least attempt to view the source to find out the gory details of how it works. Here is an example test in Figure 1.

```
1 var fn = fun(x, c)
2   x *= 3;
3   return (x);
4 };
5 var x = 2;
6 var y =
7   fn(x, fun(x) {
8     return x * 2;
9   });
```

Figure 1

The ‘var’ keyword is used to declare a new variable of some name but of undetermined type. This was optional, and the programmer was given the option of specifying the type if they wished. In a similar vein, function parameters did not require any type notation whatsoever, nor was any return type given. In essence, JASMINT was supposed to figure out that ‘fn’ had variant types but was used on line 6. From line 5, ‘x’ was determined to have the int type, and so the first argument of ‘fn’ must be of int type. This process would continue, deriving types from usages and bindings until all types had been determined.

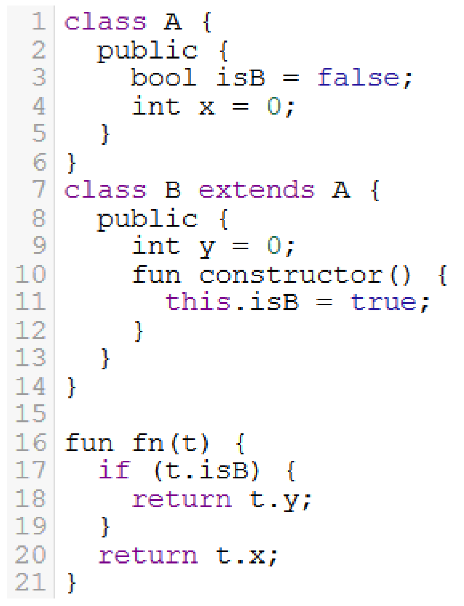
Even this example is rather complex, but the problem became even worse with function overloads and inheritance. Since type information was optional, the programmer was allowed to write a single function, almost macro-style, and the overloads would be automatically generated, as seen in Figure 2.

```
var x = [];
var y = [];
fun test(a) {
    var t;
    a.push(t);
}
test(x);
printl(x);
test(y);
x[0] = 10;
y[0] = "hi";
printl(x);
printl(y);
```

Figure 2

This code has a particular problem. Both ‘x’ and ‘y’ are given variant array types. Normally, variant identifiers would need a concrete value before being passed into a function, but in this case the empty array has no type but is still considered a value. Uh oh. The ‘test’ function is then given both of these arrays and considers them to both be arrays of an unknown inner type (of what type are the values inside the array?). After the function is used, ‘x’ and ‘y’ are determined to be arrays of ints and of strings, respectively. JASMINT must then know to generate one overload of ‘test’ for an array of ints, and another overload for an array of strings and apply those usages correctly. This is not a pleasant problem to solve for the general case.

There is also an issue with class inheritance or any instance where casting to a super-type is the desired behavior which requires that overload generation work. In Figure 3, class 'B' is a child of class 'A'. Both will be passed into 'fn'. Suppose an instance of 'B' is passed into 'fn'. Then 'fn' will determine that the type of 't' must be class B. The issue arises when another class 'C' also extends 'A' and is also passed into 'fn'. Another overload will be generated, but what the programmer really wanted was for 't' to just be of type 'A'.



```
class A {
  public {
    bool isB = false;
    int x = 0;
  }
}
class B extends A {
  public {
    int y = 0;
    fun constructor() {
      this.isB = true;
    }
  }
}

fun fn(t) {
  if (t.isB) {
    return t.y;
  }
  return t.x;
}
```

Figure 3

These problems are undoubtedly solvable, but after working with type-inferencing, it soon became apparent that this feature might not even be desirable. In large code bases it is extremely helpful to be able to look at a variable and immediately know which type it is and what data it represents. JavaScript is a perfect example of a language which started with no types, but is gradually becoming typed as programmers become frustrated with lack of type information and the inherent unpredictability. Given all of these concerns, it was decided to remove the type inferencing entirely, although it could be re-implemented in a restricted sense similar to that of C++'s 'auto' keyword.

General Pitfalls

Proper definition of data structures turned out to be a huge issue and was another driving force behind the switch to Java. In the C++ version, every form was defined as an AST, which was not necessarily wrong, but did lead to the assumption that every form would evaluate to a concrete value. This is obviously incorrect, as statements do not normally evaluate to values. Further, assignment and the dot accessor were modeled as a simple binops. This lead to hacky workarounds and the problems just accumulated.

Future Improvements

While the language could use more features, it is quite functional, and almost any feature could be added through 'trans' blocks, such as reading from a file, or using network functions. Some significant improvements to the library would be to add a memory leak tracker and reporter as well as debugging functionality with stepping. Some general improvements could be made, such as providing a base visitor class for the AST in order to make writing user modules easier.

The main difficulty transpiling besides writing the modules is filling out the trans blocks, so it would be quite useful to be able to automatically generate trans blocks for some set of libraries in the target languages.

Conclusion

The end result of this project was two working transpilers with the ability to take the JASMINT annotated AST and produce two equivalent programs in two very different languages: Python and C++. It is in nature very much still a prototype, but it is a functional proof of concept and can be further improved to become a usable and useful tool. One of the primary reasons for writing the library in C++ in the first place for performance, but performance can often be sacrificed to an extent in exchange for easier maintenance, debugging, and the writing of new features, and this balance between speed and programmer sanity is really what JASMINT seeks to achieve.

CITATIONS

Bazon, Mihai. "How to Implement a Programming Language in JavaScript." Lisperator, 2017, www.lisperator.net/pltut/.

Cameros, Emilio. "C++ BSON Implementation." Github, 22 Feb. 2017, www.github.com/cyberguijarro/minibson.

MongoDB. "Java BSON Implementation." MongoDB, 14 Sept. 2015, www.mongodb.github.io/mongo-java-driver/3.0/bson/.

Parr, Terrence. "Lexer/Parser for Java Version." ANTLR, 9 Dec. 2017, www.antlr.org/.

APPENDIX

Trans Block – Example with transpiled C++ and Python

```
trans StringLib {
    fun string substr(string data, int start_idx, int end_idx): {
        Python: |[ data[start_idx:end_idx] ]|,
        CXX: |[ data.substr(start_idx, end_idx - start_idx) ]|
    }
}

string x = StringLib.substr("Hello World!", 3, 5);
print("|" + x + "|\n");

string result = "";
if (x == "lo") {
    result = "PASS";
} else {
    result = "FAIL";
}
print(result);
```

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <functional>

#include "jsmnt_cxx_builtin/builtin.h"

std::string StringLib_substr_fun_string_string_int_int(std::string data, int start_idx, int end_idx) {
  return  data.substr(start_idx, end_idx - start_idx);
}

std::string u_x_20 = StringLib_substr_fun_string_string_int_int("Hello World!", 3, 5);
std::string u_result_30 = "";


int main(int argc, char** argv) {
  print_fun_void_string((("|" + u_x_20) + "|\n"));
  if ((u_x_20 == "lo")) {
    u_result_30 = "PASS";
  } else {
    u_result_30 = "FAIL";
  }
  print_fun_void_string(u_result_30);
  return EXIT_SUCCESS;
}
```

```python
import sys
import os

def print_fun_void_string(msg):
    print(msg, end='', flush=True)

def StringLib_substr_fun_string_string_int_int(data, start_idx, end_idx):
    return  data[start_idx:end_idx]
u_x_20 = StringLib_substr_fun_string_string_int_int("Hello World!", 3, 5)
u_result_30 = ""
def main():
    global u_x_20
    global u_result_30
    print_fun_void_string((("|" + u_x_20) + "|\n"))
    if ((u_x_20 == "lo")):
        u_result_30 = "PASS"

    else:
        u_result_30 = "FAIL"


    print_fun_void_string(u_result_30)

if __name__ == "__main__":
    main()
```