PLT 4115 Proposal: **JaTesté**

Andrew Grant
amg2215@columbia.edu

Jemma Losh
jal2285@columbia.edu

Jared Weiss
jbw2140@columbia.edu

Jake Weissman
jdw2159@columbia.edu

February 9, 2016

1 Description

As necessitated by a language that assures a programmer of the effectivity of his or her code, our language is designed to be fairly rigid and explicit in structure, without much behind-the-scenes magic that can make some existing programming languages difficult to read and understand. While at first this may seem like a limitation to the programmer who is well versed in non-statically typed languages (i.e Python) or object-oriented languages (i.e. C++ or Java), our language is just as capable, while also ensuring a robust compiled program that should always work as expected.
The syntax for our language will explicitly define the domain and range (including errors that may be raised) of functions for integration tests, as well as the expected input and output for various cases for unit tests. At compilation time, all of these tests will be run to make sure that the produced program works the way it's designers intended. In addition to user defined unit tests, we will analyze the input code and generate some of our own tests. These tests will be generated in order to cover important test cases that the user might have missed and cases that ensure full code coverage (i.e. all lines of the input code has been tested). We plan on compiling our code down to LLVM.

1.1 Compiler Proposals

We are considering two alternatives for the workflow of our compiler. The first option is an interpreter-compiler hybrid that runs unit tests and coverage tests at compile time. This method ensures that a program that does not successfully pass all tests will not have the ability to be compiled, and therefore will not have the chance for execution by the programmer. This would ensure that errand code does not make it through the compiler and into production. On failure, the compiler will inform the user as to what tests failed and what can be done to improve the quality and testability of the program.
The second consideration is to generate a test file based on the input program. At compile time, two files will be generated: (1) an executable file, and (2) an executable test file with all the relevant test cases. This method would allow the user to continue with his or her normal work flow and minimize interference from the compiler, while at the same time providing a robust test file to fully test one's program. As compared to the first option, this allows for quicker up times for programmers but would lower the level of rigidity the first option provides.

1.2 Types

- int
- char
- float

- struct
- arrays[] *(of any type)*

In our language, typing will be strongly enforced throughout every possible step of compilation. All variables in our language will need to be typed before they can be referenced (and naturally before assignment). This is similar to the strong typing of C, albeit without the complexities that are associated with passing pointers.

2 Syntax

While our syntax is inspired by C, it is very much functional. Each function can only rely on variables passed to it explicitly; functions cannot reference global variables or variables outside the functions scope. This ensures functions always return the same value when passed the same variables. This guarantees that tests can easily be written without having to worry about global variables.

2.1 Comments

/* No single line comments, only nested comments */

2.2 Key Words

- with test:
- using:
- func

2.3 Whitespace

Our language is not whitespace sensitive. During tokenization, whitespace will be discarded.

3 Example code

3.1

```
/*
  We are using multi-line comments.
  There are no single line comments.
*/
```

3.2

```
/* the func keyword is used for function declaration */
func int abv(int a)
{
  if (a < 0) {
    a = -a;
  } else {
    a = a;
  }
  return a;
}
with test: abv(-2) == 2;
```

```
/*
  This is how you use the keyword "with test:". Tests are attached to the end of
    function definitions with some sort of boolean condition. In the test below,
    the programmer makes sure the function abv returns a positive number (2) when
    passed the value -2.
*/
```

3.3

```
func int sum(int[] arr, int len)
{
  int sum;
  int i;

  i = 0;
  while ( i < len; i++) {
    sum = sum + arr[i];
  }
  return sum;
}
with test: sum(arr, 5) == 15 using int[] arr = {1, 2, 3, 4, 5};
/*
  The "using" keyword above is used to create variables that are used in the test
    condition.  Here, the programmar created an array using the keyword and makes
    sure the sum function computes the right value.
*/
```

3.4

```
func int add(int a, int b)
{
  return x + y;
}
with test: add(1,  2) ==  3;
with test: add(-1, 2) ==  1;
with test: add(-1,-2) == -3;
with test: 0 < add(1,2) < 100;
```

3.5

```
func void changeName(personStruct person, string newName)
{
  person.name = newName;
}
with test: {
  /* Here's an example of a multi-line test case wrapped in {}. */
  changeName(person, "newName");
  person.name == "newName";
} using: { personStruct person = personStruct("name", 18)};
/* Note: personStruct is just an example struct with the fields age and name; in
    the context of an actual program it would need to be defined somewhere before
    it is actually used here */
```

3.6

```
func void checkSameAge(personStruct p1, personStruct p2)
{
  return p1 == p2;
}
with test: checkSameAge{person1, person2} == True
    using: { personStruct person1 = personStruct("person", 20),
             personStruct person2 = personStruct("other",  20) };
/* Note: personStruct is just an example struct with the fields age and name; in
   the context of an actual program it would need to be defined somewhere before
   it is actually used here */
```

3.7

```
/*
  Here is an actual program that computes the gcd of two numbers.
*/

/* Function declarations */
func int gcd(int a, int b)
{

if (a == b) {
return a;
} else if (a > b) {
gcd(a-b, b);
} else {
gcd(a, b-a);
}

}
with test: gcd(10, 5) == 5;

/* Main body of program starts here */

int num1 = 25;
int num2 = 30;
int num3;

num3 = gcd(num1, num2);

print("Result %d\n" , num3);
```