JaTesté

Build Software So Secure You May Actually Make America Great Again

Jake Weissman, Andrew Grant, Jemma Losh, Jared Weiss

Why JaTesté?

- JaTesté promotes good coding practices, allowing the programmer to easily define test cases, for any function, directly into his or her source code.
- Compiler creates two files:
  - 1) Normal executable
  - 2) An executable that runs the user-defined tests and prints results

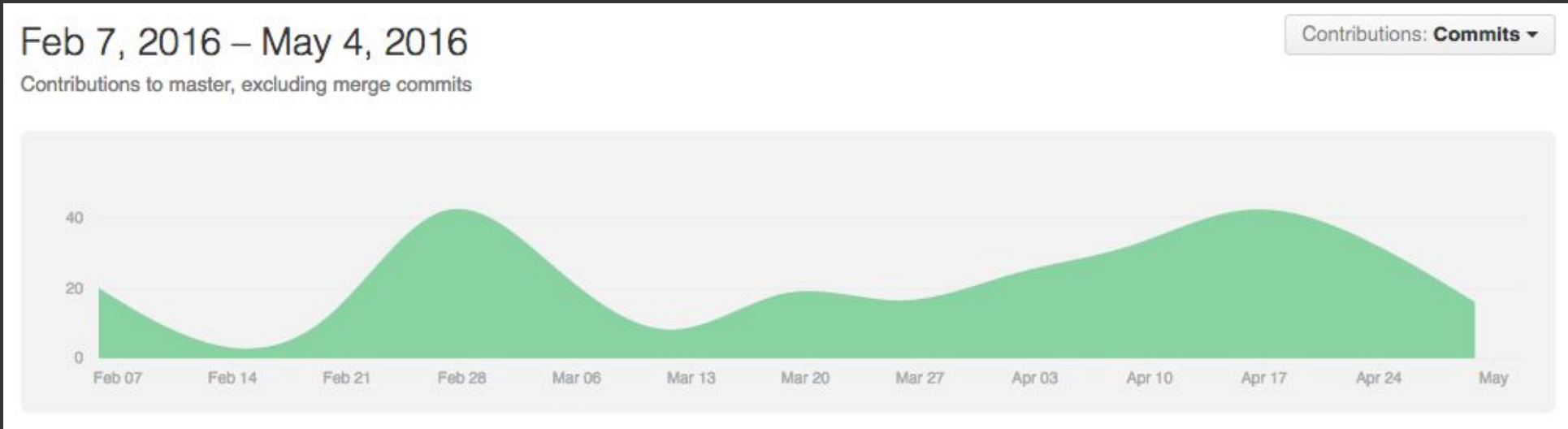
JaTesté Team

- Andrew Grant (amg2215@columbia.edu), Language Guru
- Jake Weissman (jdw2159@columbia.edu), Manager
- Jared Weiss (jbw2140@columbia.edu), Software Architect
- Jemma Losh (jal2285@columbia.edu), Tester

Software Development Environment

- Git + github
  - https://github.com/jaredweiss/JaTeste
  - 28 Issues Closed
  - 137 Pull Requests Closed
  - 530+ commits
- Vim
- Ubuntu 15.10 + VirtualBox
- OCaml
- Makefile

Teamwork Success



(Very) Quick JaTesté Overview

- Can directly embed test cases into one's source code
  - This is the main point of the language
- Imperative programming language, with light object-oriented features.
  - Syntax and paradigm similar to C, Java, etc
- Compiles into LLVM, a portable assembly-like language

(Very) Quick JaTesté Overview contd.

- Testing is at the heart of the JaTesté programming language
  - “with test” keyword appended to function to define tests
  - “using” keyword appended to “with test” block to set up environment for tests

```
func int main()
{
    Do_insightful_stuff;
    return 0;
}

func int add(int x, int y)
{
        return x + y;
} with test {
        assert(add(a,0) == 10);
} using {
        int a;
        a = 10;
}
```

- Normal function
- Test cases for function
- Environment for test cases

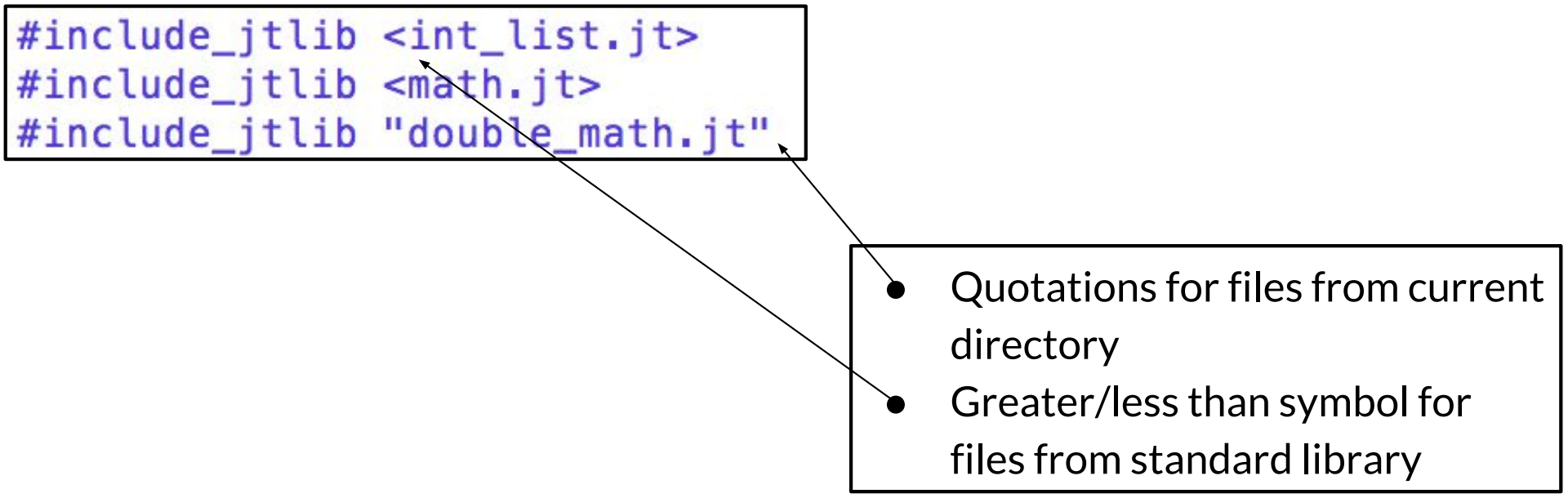
JaTesté Program Syntax

```
#include_jtlib <int_list.jt>

double global_var;

func int main(int argc)
{
                int a;
                int b;
                int c;
                a = 9;
                b = 25;
                c = add(a,b);
                print(c);

                return 0;
}

func int add(int x, int y)
{
        return x + y;
}  with test {
        assert(add(a,0) == 10);
        assert(add(b,b) == 10);
        assert(add(a,b) == 10);
} using {
        int a;
        int b;
        a = 10;
        b = 5;
}

struct square {
        int height;
        int width;
};
```

- Program made up of four segments:
  - 1) Header files
  - 2) Global variables
  - 3) Function definitions
  - 4) Struct definitions
- Must be in this order

JaTesté Header Files Syntax

```
#include_jtlib <int_list.jt>
#include_jtlib <math.jt>
#include_jtlib "double_math.jt"
```



JaTesté Statements Syntax

```
for (i = 0; i < 100; i = i + 1) {

}

if (my_bool == true)  {
        print("true");
} else {
        print("falsed");
}

return 0;
```

- Standard control flow constructs
  - For loops
  - While loops
  - If-else statments
  - Return statements
- Can have side effects

JaTesté Struct Syntax

```
struct square {
        int height;
        int width;

        method int get_area()
        {
                int temp_area;
                temp_area = height * width;
                return temp_area;
        }

        method void set_height(int h) {
                height = h;
        }

        method void set_width(int w) {
                width = w;
        }

};
```

- Structs can contain fields and methods
- Like objects in Java
  - But significantly worse

JaTesté Test Syntax

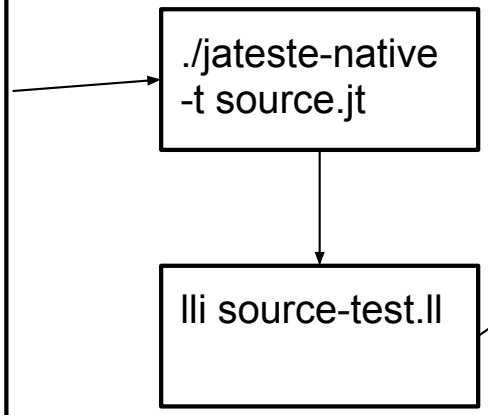
./jatest-native -t source.jt → lli source-test.ll

```
func int add(int x, int y)
{
        return x + y;
} with test {
        assert(add(a,0)   == 10);
        assert(add(b,b)   == 10);
        assert(add(a,b)   == 15);
} using {
        int a;
        int b;
        a = 10;
        b = 5;
}
```

```
TEST RESULTS!
**************
add results:
add(a,0) == 10 passed!
add(b,b) == 10 passed!
add(a,b) == 10 failed!
LHS evaluated to:
15
RHS evaluated to:
10
**************
```

JaTesté Test Syntax contd.

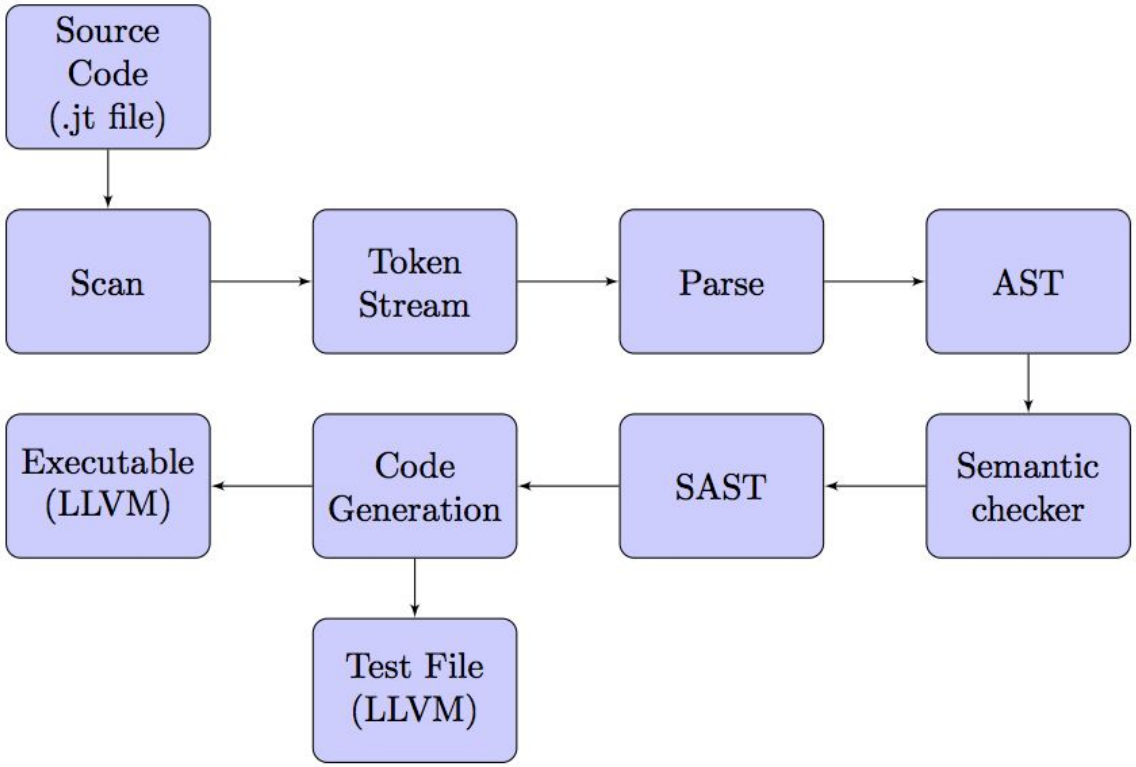
```
func int my_gcd(int a, int b)
{
        while (a != b) {
                if (a > b) {
                        a = a - b;
                } else {
                        b = b - a;
                }
        }
        return a + 1;
} with test {
        assert(my_gcd(a,b) == 3);
        assert(my_gcd(78,9) == 3);
        assert(my_gcd(d,c) == 9);
} using {
        int a;
        int b;
        int c;
        int d;
        a = 15;
        b = 9;
        c = 54;
        d = 9;
}
```




```
**************
my_gcd results:
my_gcd(a,b) == 3 failed!
LHS evaluated to:
4
RHS evaluated to:
3
my_gcd(78,9) == 3 failed!
LHS evaluated to:
4
RHS evaluated to:
3
my_gcd(d,c) == 9 failed!
LHS evaluated to:
10
RHS evaluated to:
9
**************
```

Compiler Overview

- Compiler files:
  - jateste.ml: entry point for soure code
  - Scanner.mll: reads characters, and outputs tokens
  - parser.mly: generates AST from tokens
  - ast.ml: defines AST
  - semant.ml: checks semantics of the AST, generates SAST
  - sast.ml: defines SAST
  - codegen.ml: turns SAST into LLVM code
  - exceptions.ml: defines error messages
- 1830 lines of Compiler source code
- Standard library in lib/ folder
- Test files in test/



Compiler Architecture

Compiler Overview contd.

- Key idea: if “-t” command line argument is supplied, the compiler generates two executables
  - Normal file
  - Test file
- ./jateste-native -t source.jt -> source.ll, source-test.ll
  - lli source.ll
  - lli source-test.ll
  - (lli is an LLVM interpreter)

Compiler Overview contd.

source.jt (pseudo-code)

```
func int main()
{
    Do_insightful_stuff;
    return 0;
}

func int add(int x, int y)
{
        return x + y;
} with test {
        assert(add(a,0) == 10);
} using {
        int a;
        a = 10;
}
```

source-test.ll (pseudo-code)

```
func int main()
{
    printResultOf: addtest();
    return 0;
}

func int add(int x, int y)
{
        return x + y;
}

func void addtest()
{
        int a;
        a = 10;
        assert(add(a,0) == 10);
}
```

source.ll (pseudo-code)

```
func int main()
{
    Do_insightful_stuff;
    return 0;
}

func int add(int x, int y)
{
        return x + y;
}
```

Compiler Overview contd.

- cd src/
- make all -> outputs jateste-native binary
- ./jateste-native -t source.jt -> source.ll, source-test.ll
  - lli source.ll
  - lli source-test.ll
- JaTeste standard library in lib/

Testing

- Testing done via Makefile
  - diff test-var1.jt test-var.1out
  - diff test-class1.jt test-class1.out
  - etc....
- 126 test files
  - All passed
- Two Makefiles
  - Primary Makefile in src/ -> where source code is compiled
  - Test Makefile in test/ -> where tests are defined and added

Testing contd.

```
========== Running All Tests! ==========
make[1]: Entering directory '/home/plt/JaTeste/test'
Testing 'global-scope.jt'
  ----->  Test passed!
Testing 'global-scope.jt'
  ----->  Test passed!
Testing 'test-func1.jt'
  ----->  Test passed!
Testing 'test-func2.jt'
  ----->  Test passed!
Testing 'test-func3.jt'
  ----->  Test passed!
Testing 'test-pointer1.jt'
  ----->  Test passed!
Testing 'test-while1.jt'
  ----->  Test passed!
Testing 'test-for1.jt'
  ----->  Test passed!
Testing 'test-malloc1.jt'
  ----->  Test passed!
```

Testing contd.

```
========== Runtime Tests Passed! ==========
Testing 'local-var-fail.jt', should fail to compile...
  ----->  Test passed!
Testing 'no-main-fail.jt', should fail to compile...
  ----->  Test passed!
Testing 'return-fail1.jt', should fail to compile...
  ----->  Test passed!
Testing 'return-fail2.jt', should fail to compile...
  ----->  Test passed!
Testing 'return-fail3.jt', should fail to compile...
  ----->  Test passed!
Testing 'return-fail4.jt', should fail to compile...
  ----->  Test passed!
Testing 'struct-access-fail1.jt', should fail to compile...
  ----->  Test passed!
Testing 'invalid-assignment-fail1.jt', should fail to compile...
  ----->  Test passed!
Testing 'class1-var-fail1.jt', should fail to compile...
  ----->  Test passed!
```

Demo Time!