Supercharge a Language Runtime!

CASCON 2017 @ Markham

Chairs: Daryl Maier & Xiaoli Liang
Speakers: Xiaoli Liang, Robert Young, Andrew Young, Arianne Butler
November 6, 2017

IBM Runtime Technologies

Speakers

Xiaoli Liang – JIT Developer xliang6@gmail.com

Robert Young – GC Developer rwy0717@gmail.com

Arianne Butler – JIT Intern arianne.leah@usask.ca

Important disclaimers

- THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.
- WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.
- ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.
- ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.
- IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.
- IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.
- NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:
  - CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

Legal Notice

IBM and the IBM logo are trademarks or registered trademarks of IBM Corporation, in the United States, other countries or both.

Java and all Java-based marks, among others, are trademarks or registered trademarks of Oracle in the United States, other countries or both.

Other company, product and service names may be trademarks or service marks of others.

THE INFORMATION DISCUSSED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, SUCH INFORMATION. ANY INFORMATION CONCERNING IBM'S PRODUCT PLANS OR STRATEGY IS SUBJECT TO CHANGE BY IBM WITHOUT NOTICE.

Workshop Agenda

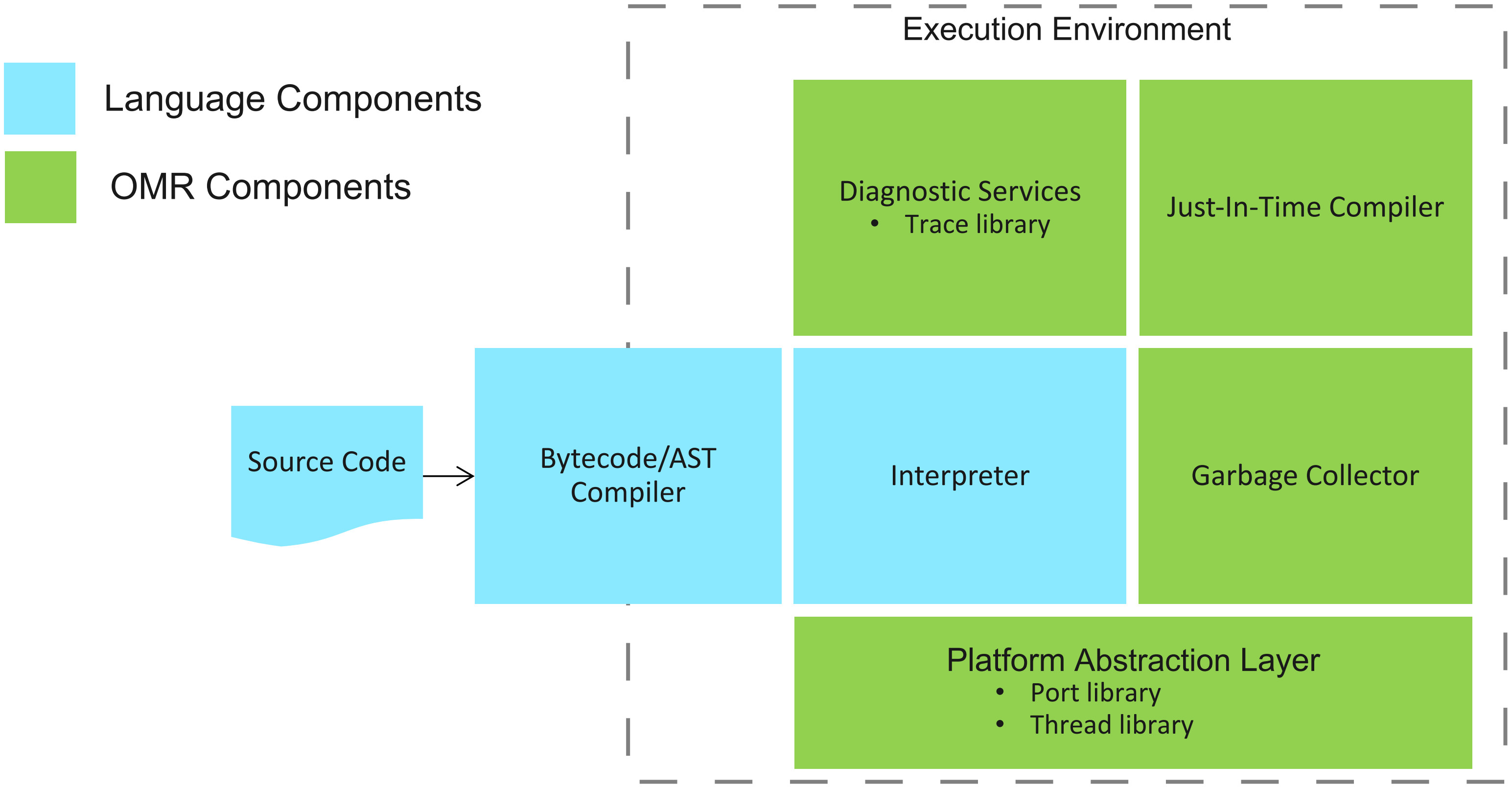
- OMR Overview
- Base9 Introduction
- Exercise to add a new opcode to Base9
  - Front-end Compiler
  - Interpreter
  - JIT Compiler
- Performance Improvements
- Future Work

OMR Overview

What is OMR?

- An open sourced and reusable C++ library for building runtimes
- A toolkit derived from the source code of IBM’s production runtimes
- Implements language-agnostic parts of a managed runtime
- Bootstraps development of new runtimes
- Allows incremental enablement of advanced functionality
- Ships as part of IBM SDK for Java 8
- Consumed by Eclipse OpenJ9
- POC integrations with Ruby, CSOM, Lua, Swift, and Rosie Pattern Language

Eclipse OMR Components



🟦 Language Components

🟩 OMR Components

Execution Environment

Diagnostic Services
- Trace library

Just-In-Time Compiler

Source Code

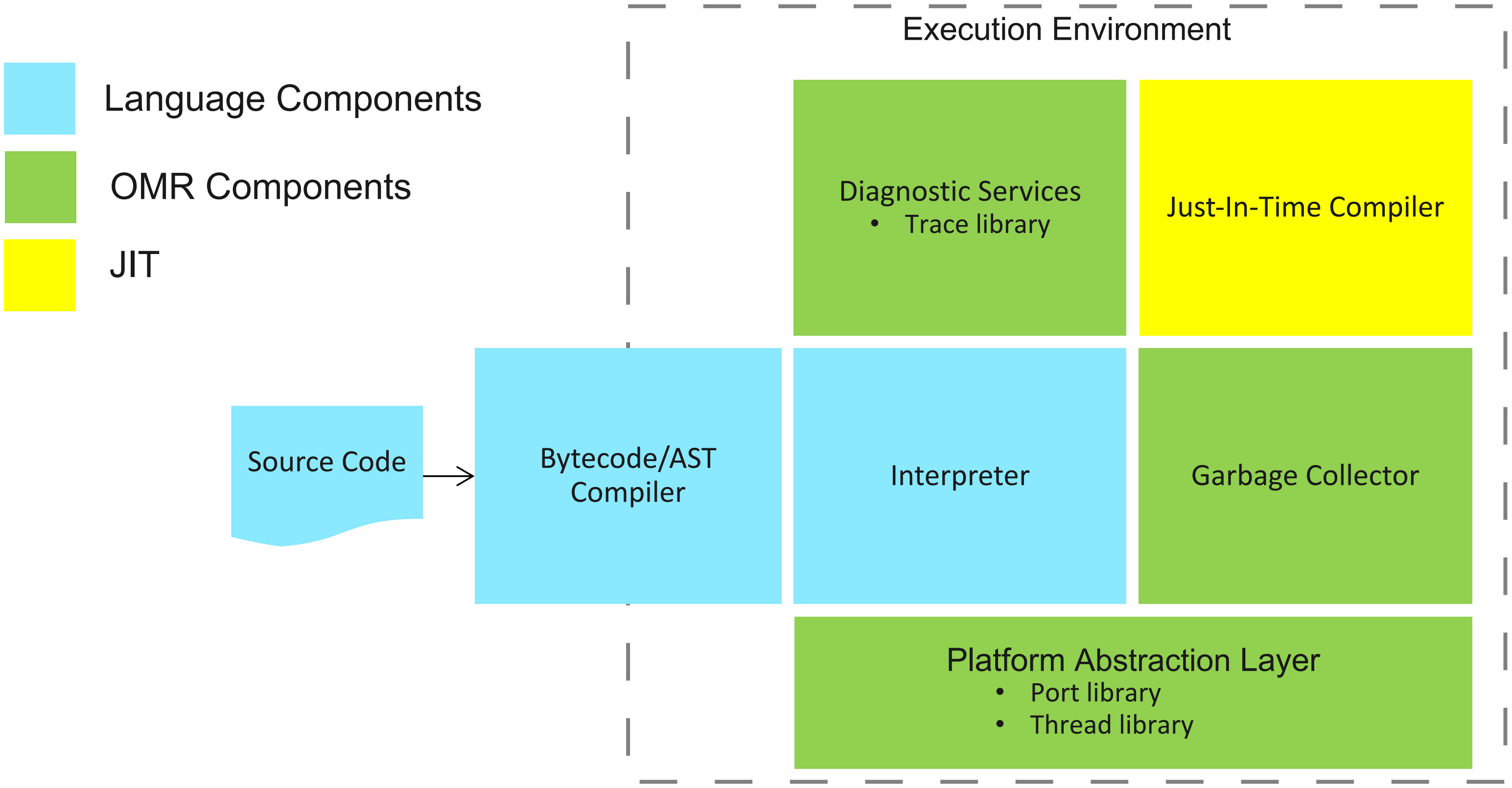
Bytecode/AST Compiler

Interpreter

Garbage Collector

Platform Abstraction Layer
- Port library
- Thread library

Eclipse OMR Components



🟦 Language Components

🟩 OMR Components

🟨 JIT

Execution Environment

Diagnostic Services
- Trace library

Just-In-Time Compiler

Source Code

Bytecode/AST Compiler

Interpreter

Garbage Collector

Platform Abstraction Layer
- Port library
- Thread library

Base9 Introduction

Base9

- Base9 is a tiny virtual machine
  - Extremely limited functionality (integers and strings)

- An educational project demonstrating how to enable the OMR JIT technology using its JitBuilder toolkit

- The Base9 language is a subset of JavaScript

What is a Virtual Machine?

- A program that takes a sequence of instructions and executes them
  - Just like a computer!

- Programs are ahead of time (AOT) compiled into bytecodes and then interpreted

- The bytecodes operate on a stack of values
  - Uses a stack to store parameters, local variables, and return values

What are Bytecodes?

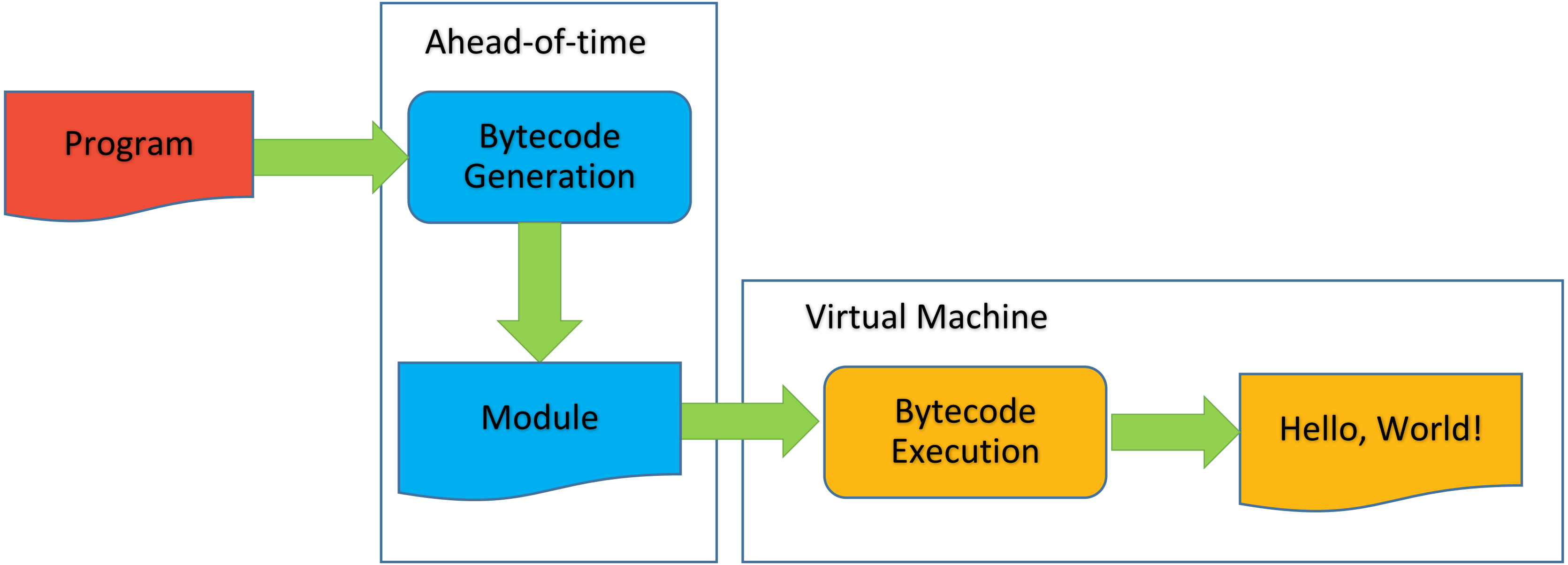
- A Bytecode is a type of instruction that can be consumed by a virtual machine
- Bytecodes are not dependent on a specific processor’s instruction set, which allows for cross-platform portability
- Bytecodes are designed to be efficiently decoded and interpreted by a computer program and are not necessarily human readable

What are Bytecodes? …

- Example Base9 bytecodes:
  - FUNCTION_CALL
  - FUNCTION_RETURN
  - INT_ADD
  - INT_SUB
  - PUSH_FROM_VAR
  - POP_INTO_VAR

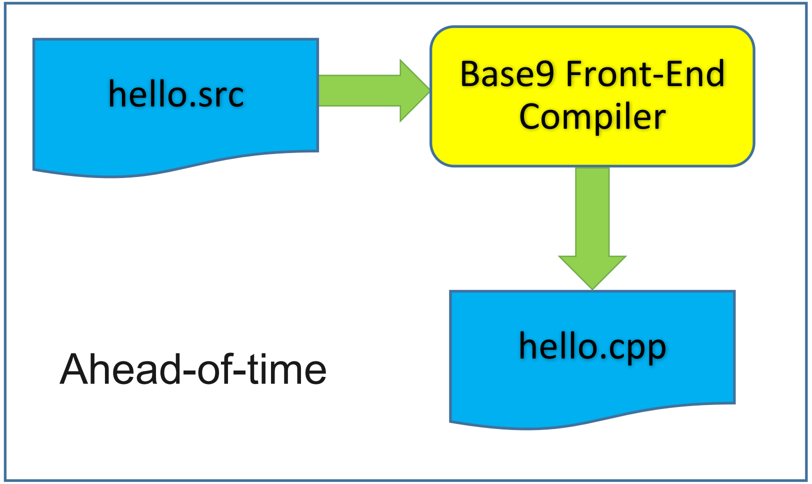
- See b9/include/b9/instructions.hpp for more…

Base9: From Program to Execution



Generating Base9 Modules

- Base9 programs are compiled into modules before execution (AOT Compilation)
- Base9 modules contain a program’s functions, laid out as static sequences of bytecodes
- The Base9 Front-End produces a file which contains a C++ representation of our bytecodes



Base9 Language to Bytecode

```
function b9main() {
  b9PrintString("Hello World!");
}
```

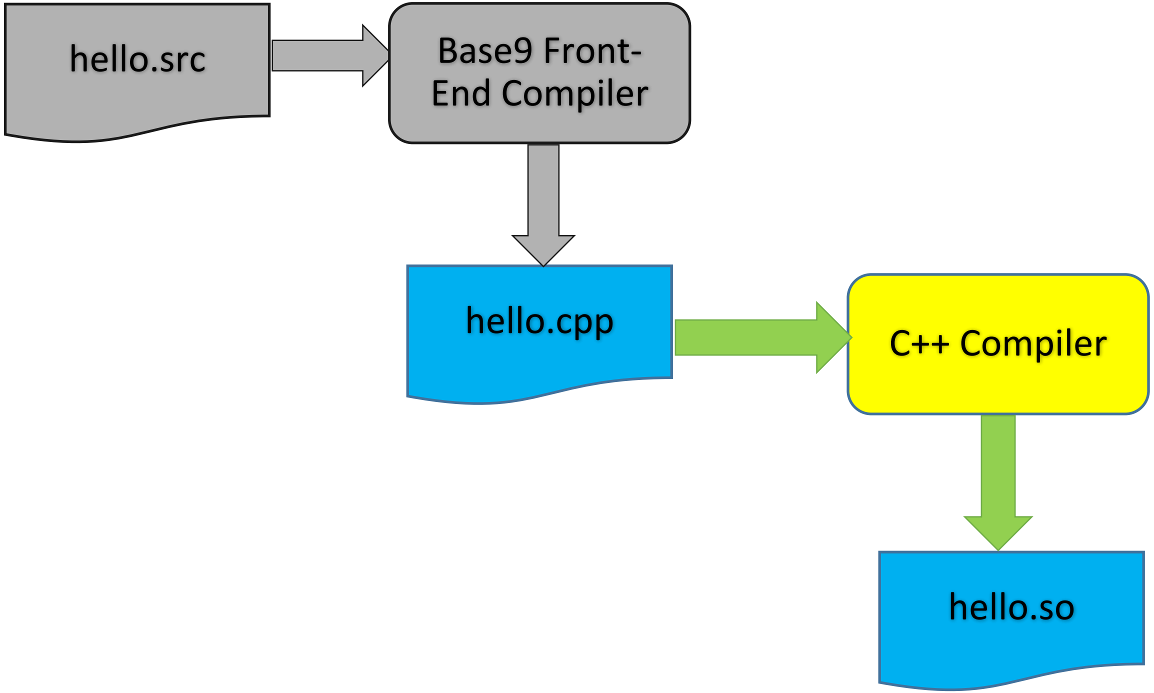
Base9 Language

Base9 Front-End Compiler

```
Instruction b9main[] = {
  {ByteCode::STR_PUSH_CONSTANT, 0}
  {ByteCode::FUNCTION_CALL, 0},
  {ByteCode::DROP},
  {ByteCode::INT_PUSH_CONSTANT, 0},
  {ByteCode::FUNCTION_RETURN},
  END_SECTION
};
```

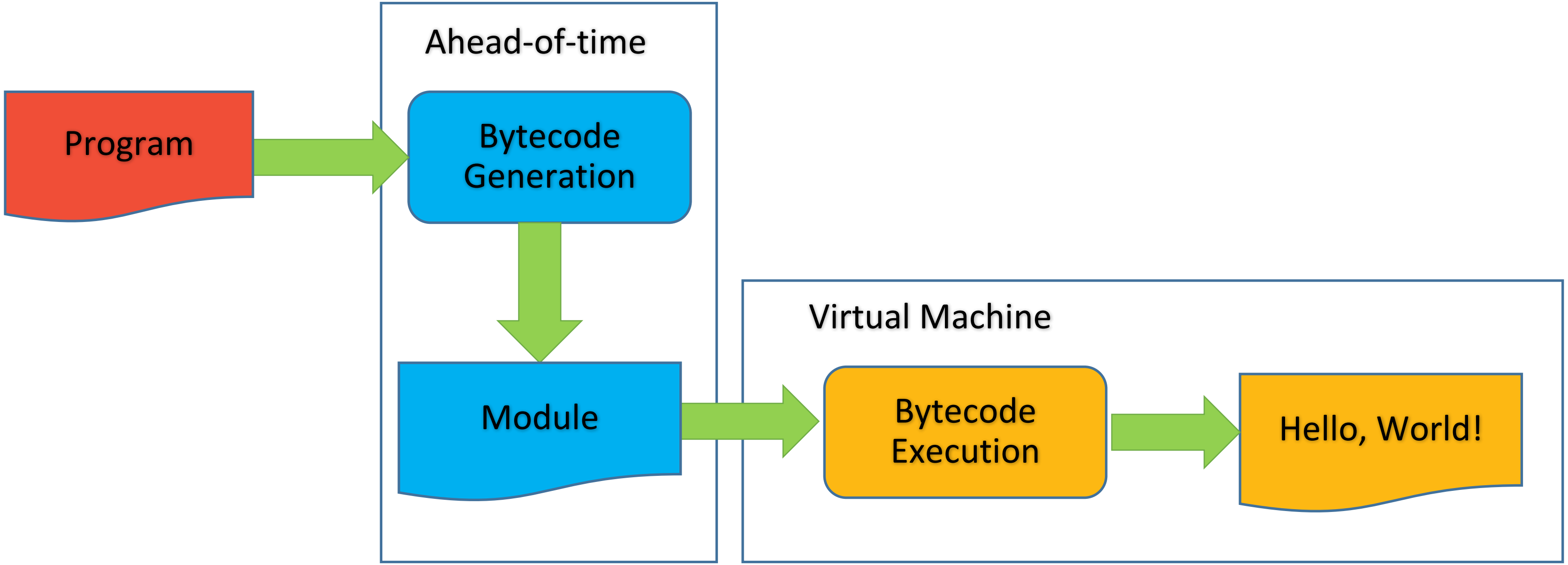
Bytecodes

Base9 Module AOT Compilation

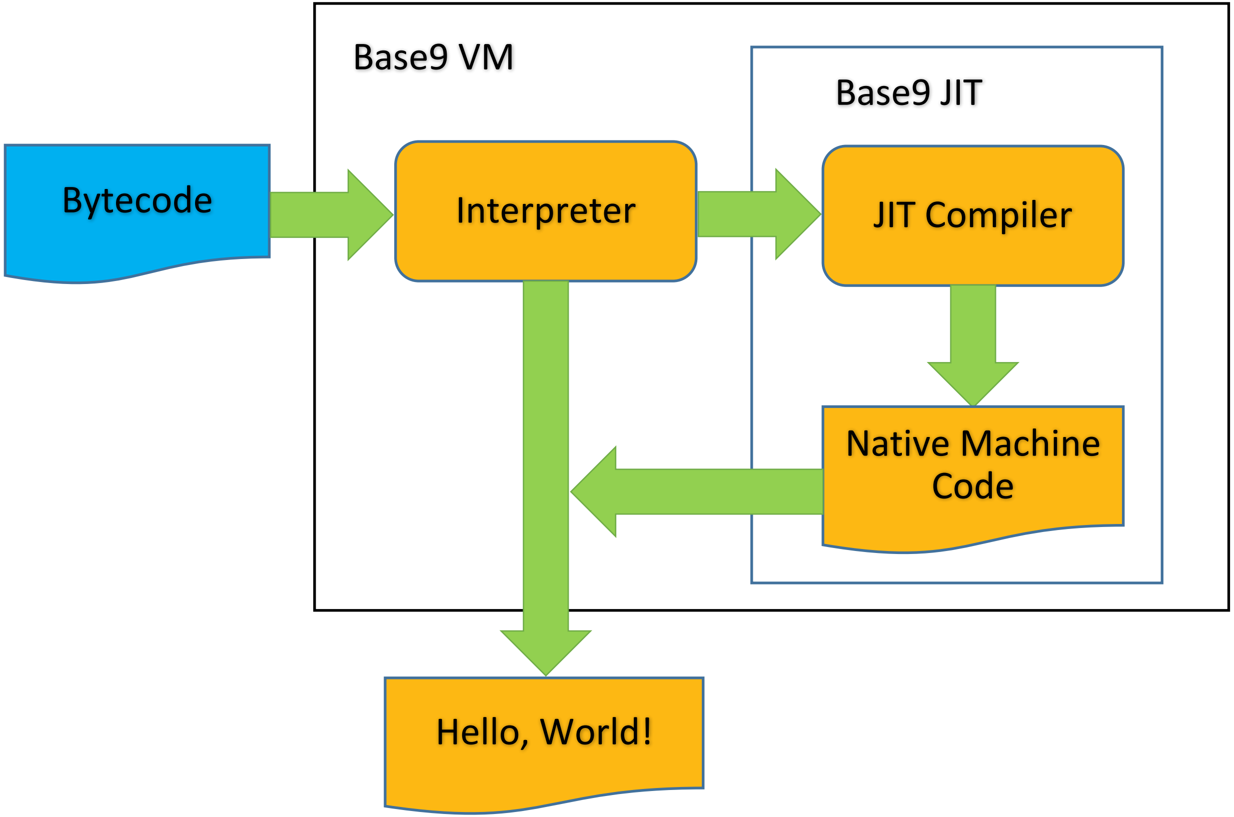


- The C++ compiler translates the static bytecode sequences into a native library

Base9: From Program to Execution

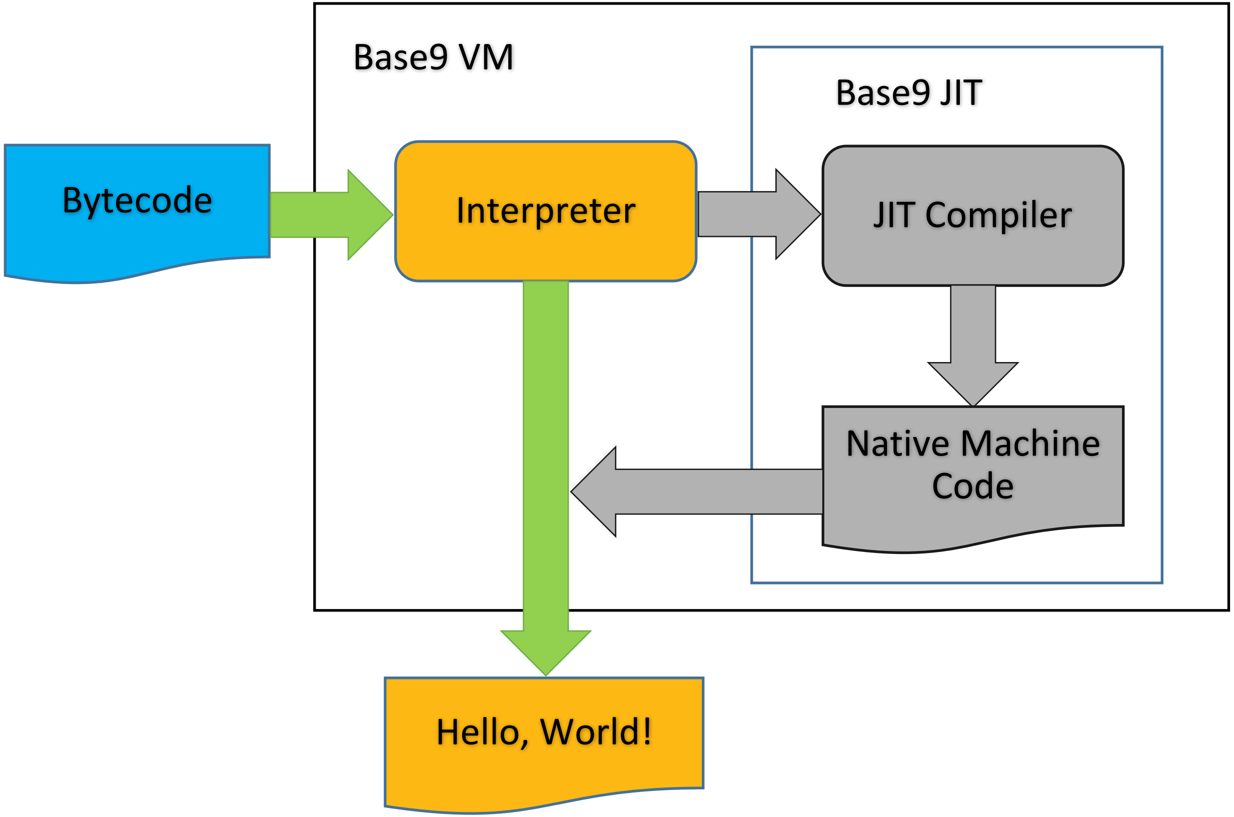


Base9 Runtime



Base9 Interpreter

Base9 Runtime



Understanding bytecode interpreters

- An interpreter “translates” bytecodes to native executable code

- The interpreter is essentially a while-loop around a giant switch statement

- For each bytecode, it executes a corresponding C++ function that implements the functionality of the bytecode

Instruction Pointer and Stack Pointer

- The Instruction Pointer (IP) is initialized to the start of the sequence of bytecodes

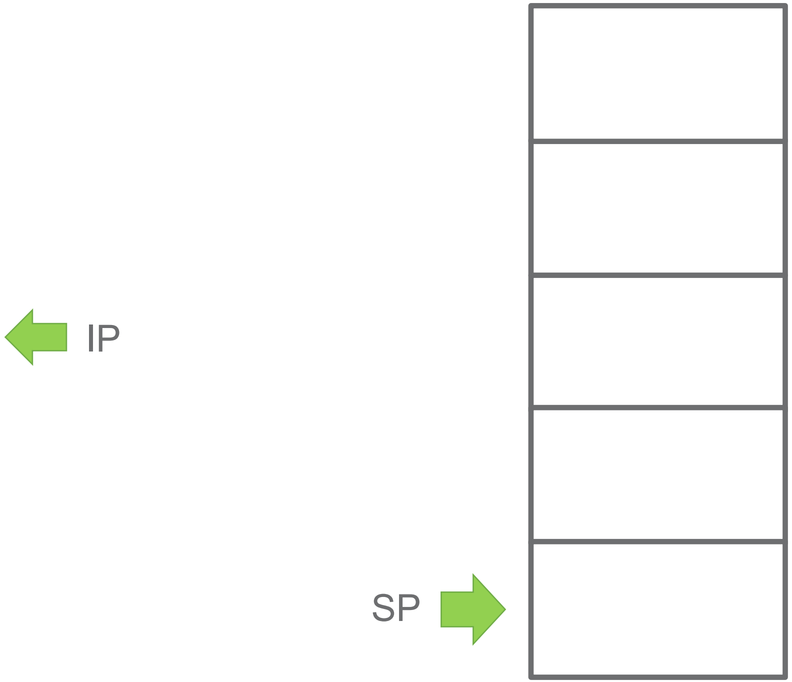
- The while-loop terminates when the IP is pointing at the END_SECTION terminal

- The Stack Pointer (SP) tracks the top of the stack

Bytecodes and the Stack

```
function simple_add() {
  return 5 + 6;
}
```

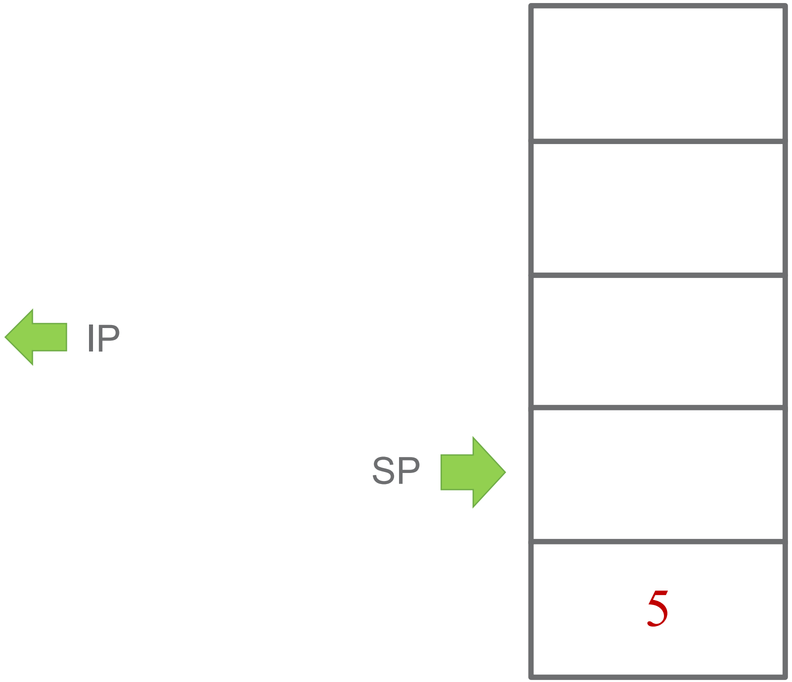
```
Instruction simple_add[] = {
  {ByteCode::INT_PUSH_CONSTANT, 5},
  {ByteCode::INT_PUSH_CONSTANT, 6},
  {ByteCode::INT_ADD},
  {ByteCode::FUNCTION_RETURN},
  END_SECTION
};
```



Bytecodes and the Stack

```
function simple_add() {
  return 5 + 6;
}
```

```
Instruction simple_add[] = {
  {ByteCode::INT_PUSH_CONSTANT, 5},
  {ByteCode::INT_PUSH_CONSTANT, 6},
  {ByteCode::INT_ADD},
  {ByteCode::FUNCTION_RETURN},
  END_SECTION
};
```

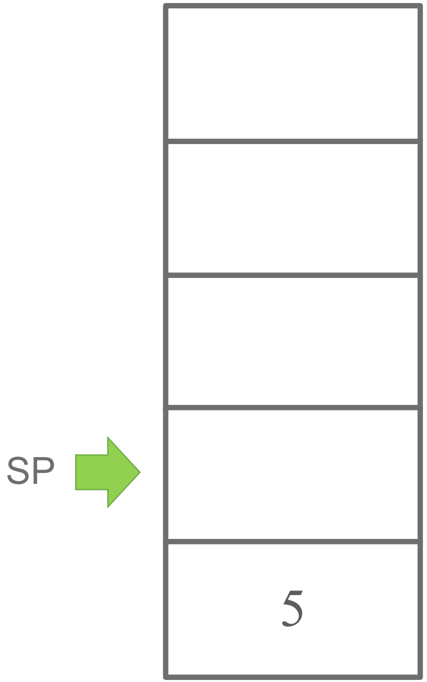


Bytecodes and the Stack

```
  function simple_add() {
    return 5 + 6;
  }
```

```
Instruction simple_add[] = {
  {ByteCode::INT_PUSH_CONSTANT, 5},
  {ByteCode::INT_PUSH_CONSTANT, 6},
  {ByteCode::INT_ADD},
  {ByteCode::FUNCTION_RETURN},
  END_SECTION
};
```

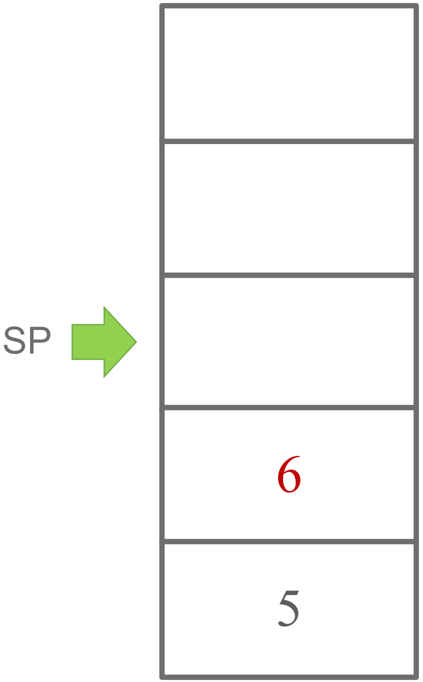
IP



Bytecodes and the Stack

```
function simple_add() {
  return 5 + 6;
}
```

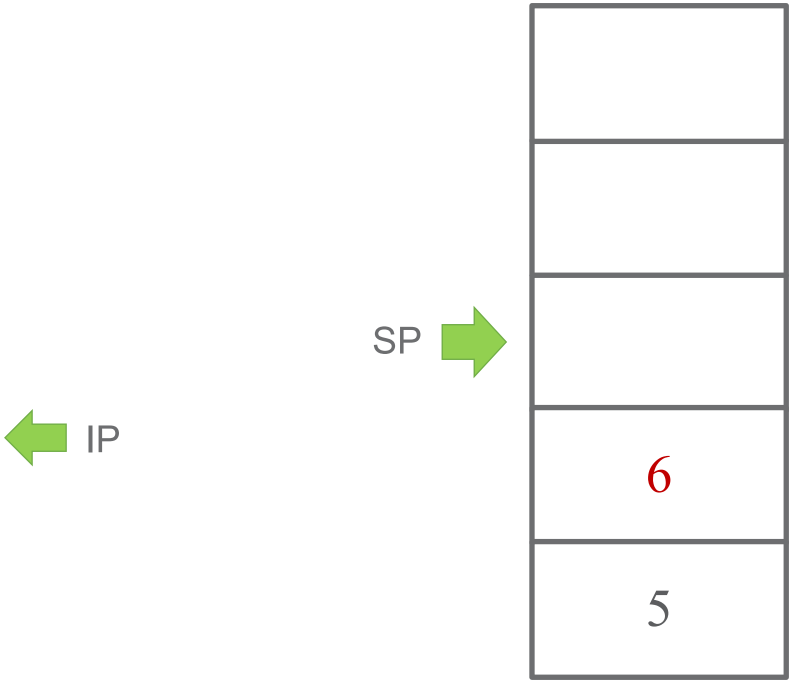
```
Instruction simple_add[] = {
  {ByteCode::INT_PUSH_CONSTANT, 5},
  {ByteCode::INT_PUSH_CONSTANT, 6},   IP
  {ByteCode::INT_ADD},
  {ByteCode::FUNCTION_RETURN},
  END_SECTION
};
```



Bytecodes and the Stack

```
  function simple_add() {
    return 5 + 6;
  }
```

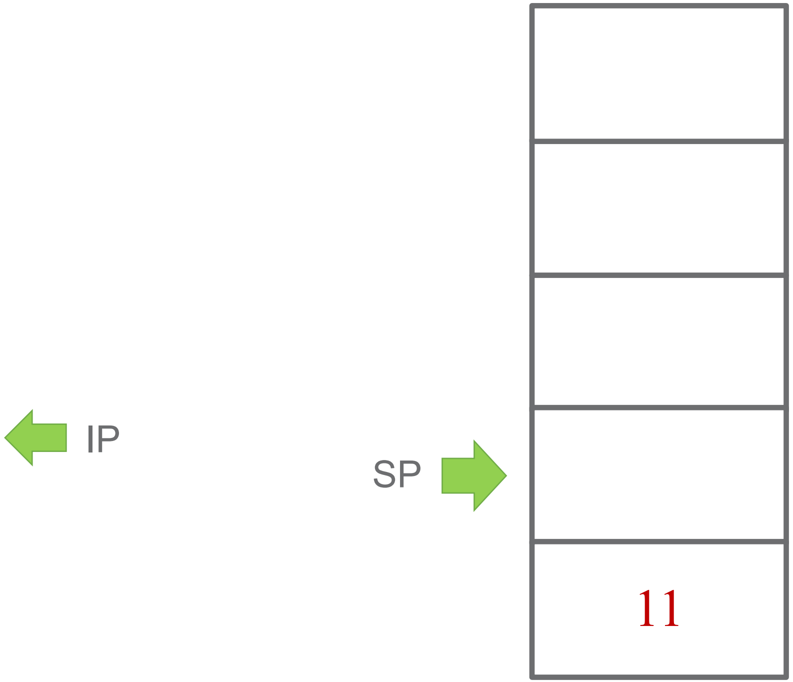
```
Instruction simple_add[] = {
  {ByteCode::INT_PUSH_CONSTANT, 5},
  {ByteCode::INT_PUSH_CONSTANT, 6},
  {ByteCode::INT_ADD},
  {ByteCode::FUNCTION_RETURN},
  END_SECTION
};
```



Bytecodes and the Stack

```
function simple_add() {
  return 5 + 6;
}
```

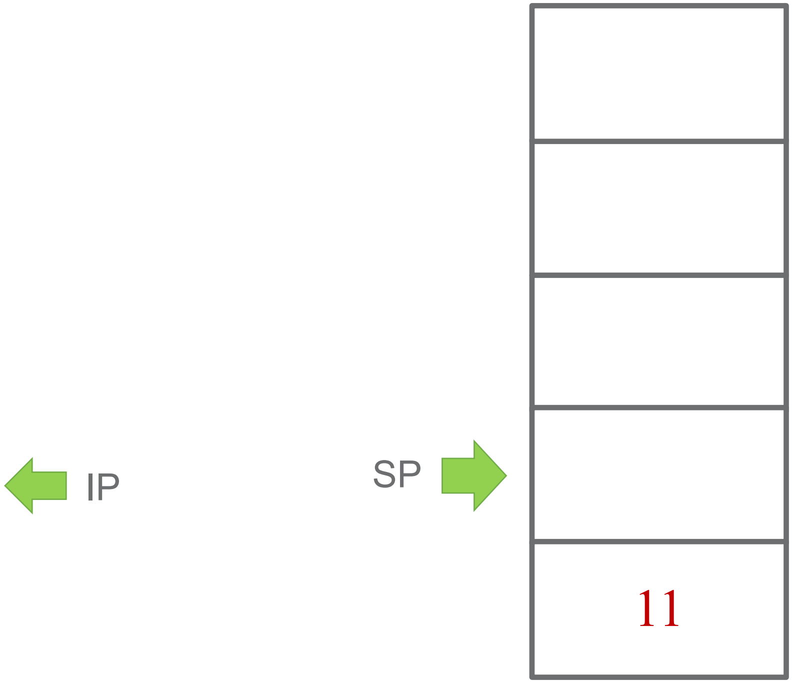
```
Instruction simple_add[] = {
  {ByteCode::INT_PUSH_CONSTANT, 5},
  {ByteCode::INT_PUSH_CONSTANT, 6},
  {ByteCode::INT_ADD},
  {ByteCode::FUNCTION_RETURN},
  END_SECTION
};
```



Bytecodes and the Stack

```
function simple_add() {
  return 5 + 6;
}
```

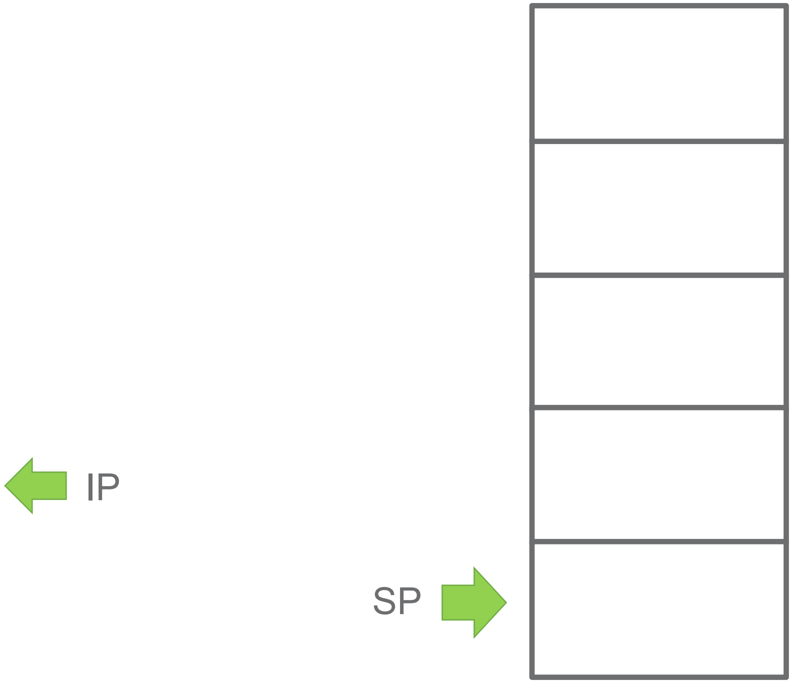
```
Instruction simple_add[] = {
  {ByteCode::INT_PUSH_CONSTANT, 5},
  {ByteCode::INT_PUSH_CONSTANT, 6},
  {ByteCode::INT_ADD},
  {ByteCode::FUNCTION_RETURN},
  END_SECTION
};
```



Bytecodes and the Stack

```
function simple_add() {
  return 5 + 6;
}
```

```
Instruction simple_add[] = {
  {ByteCode::INT_PUSH_CONSTANT, 5},
  {ByteCode::INT_PUSH_CONSTANT, 6},
  {ByteCode::INT_ADD},
  {ByteCode::FUNCTION_RETURN},
  END_SECTION
};
```



Returns 11

Interpreter Loop

```
while (*instructionPointer != END_SECTION) {
  switch (instructionPointer->byteCode()) {

    case ByteCode::INT_PUSH_CONSTANT:
      intPushConstant(instructionPointer->parameter());
      break;

    ...


    case ByteCode::INT_ADD:
      intAdd();
      break;

    ...
  }
  instructionPointer++;
}
```

Corresponding C++ function

```
void ExecutionContext::intAdd() {
  StackElement right = pop();
  StackElement left = pop();
  StackElement result = left + right;
  push(result);
}
```

Base9 Tutorial Set-up

What We're Using

Esprima

CMake

Clang

GNU

Set up: Ubuntu

```
sudo apt-get install <item>
```

- items:
  - `git`
  - `build-essential`
  - `npm`
  - `nodejs`
  - `cmake`
  - `ninja-build`

Set up: macOS

Get homebrew: https://brew.sh/

```
sudo xcode-select --install
```

```
brew install <item>
```

- items:
  - git
  - node
  - npm
  - cmake
  - ninja

Clone the Base9 Repository

```
git clone --recursive https://github.com/youngar/Base9.git
```

Get things Compiling

```
git clone --recursive https://github.com/youngar/Base9.git


cd Base9

git submodule update --init

mkdir build && cd build

npm install esprima

cmake –GNinja –DCMAKE_BUILD_TYPE=Debug ..

ninja
```

Running Hello World!

- `b9run:` the main executable that runs the Base9 modules
- `Usage: b9run [-function <function>] <module> [<arg>...]`

Go ahead and try it!

```
$ ./b9run/b9run ./test/libhellod.so
```

```
Hello World!
=> 0
```

Note: on OSX use `libhellod.dylib`

Look at the bytecodes:

```
$ vim ./test/hello.cpp
```

Running Hello World!

Make a change to hello.src and recompile:

```
cd ..                 # back to the source directory

vim test/hello.src

cd build/

ninja

./b9run/b9run ./test/libhellod.so
```

The Layout of Base9

- `b9/` – the base9 core library

- `b9run/` – the b9run program

- `js_compiler/` – the frontend compiler (JS to C++)

- `test/` – tests and example programs

- `build/` – build artifacts go here

Looking inside b9/

- `core.cpp` – the VM interpreter source
- `jit.cpp` – the JIT compiler source
- `include/b9/instructions.hpp` – the header defining our ByteCodes
  - All of our headers are inside of include/b9

How to Debug Base9

- Build in Debug mode:

```
cmake -DCMAKE_BUILD_TYPE=Debug -GNinja $Base9_dir
ninja
```

- Enable JIT tracing using `TR_Options`:

```
export TR_Options="{*b9*}(tracefull,traceILgen,log=my.trace)"
./b9run/b9run -jit test/libhellod.so
```

- Enable b9run Debug:

```
./b9run/b9run -debug ./test/libhellod.so
```

Adding a new Bytecode to Base9

Let’s add a new bytecode!

- INT_MUL (integer multiply)
  - pops two elements off the stack and pushes the product back on to the stack

TODO

1. Update the Base9 Front-End Compiler to handle the * operator and emit the INT_MUL bytecode
2. Update the interpreter to understand INT_MUL
3. Update the JIT to compile INT_MUL

HINT: Search for the “CASCON2017” eye-catcher for places to add code

INT_MUL Front-end Compiler

Add the Bytecode

- Open b9/include/b9/instructions.hpp

- Within the ByteCode enumeration, find the “CASCON2017” eye catcher and add:

```
// Multiply two integers
INT_MUL = 0xb,
```

- Find the second “CASCON2017” eye catcher and add:

```
case ByteCode::INT_MUL:
  return "INT_MUL";
```

Update the Base9 Front-End compiler

- Open up js_compiler/compile.js

- The bytecode handlers are in “`function CodeGen(f)`”

- Search “`CASCON2017`” and add:

```
if (decl.operator == "*") {
  this.outputInstruction("ByteCode::INT_MUL", 0, "");
  this.currentFunction.pushN(-1); // pop 2, push 1
  return decl.operator;
}
```

Test the Base9 Front-end

- Change factorial test to use multiply instead add:

```
vim ../test/factorial.src
```

- Rebuild the C++ file:

```
nodejs ../js_compiler/compile.js ../test/factorial.src > test/factorial.cpp

Note: on OSX use node instead of nodejs
```

- Look at the bytecodes:

```
vim test/factorial.cpp
```

INT_MUL Interpreter

Update the interpreter

- Open b9/include/b9/interpreter.hpp

- Search “`CASCON2017`” and add:

```
void intMul();
```

Update the interpreter

- Open b9/core.cpp

- Search “CASCON2017” and add:

```
void ExecutionContext::intMul() {
  StackElement right = pop();
  StackElement left = pop();
  StackElement result = left * right;
  push(result);
}
```

Update the interpreter

- Again within b9/core.cpp

- Locate the main interpreter loop and note the giant switch statement

- Again find the "CASCON2017" eye catcher and add:

```
case ByteCode::INT_MUL:
  intMul();
  break;
```

Test interpreter changes

- If you’ve fallen behind, you can fetch the full solution from GitHub:

```
git remote add upstream https://github.com/youngar/Base9.git
git checkout upstream/CASCON2017_Solution
git checkout -b solution
```

- Compile your changes:

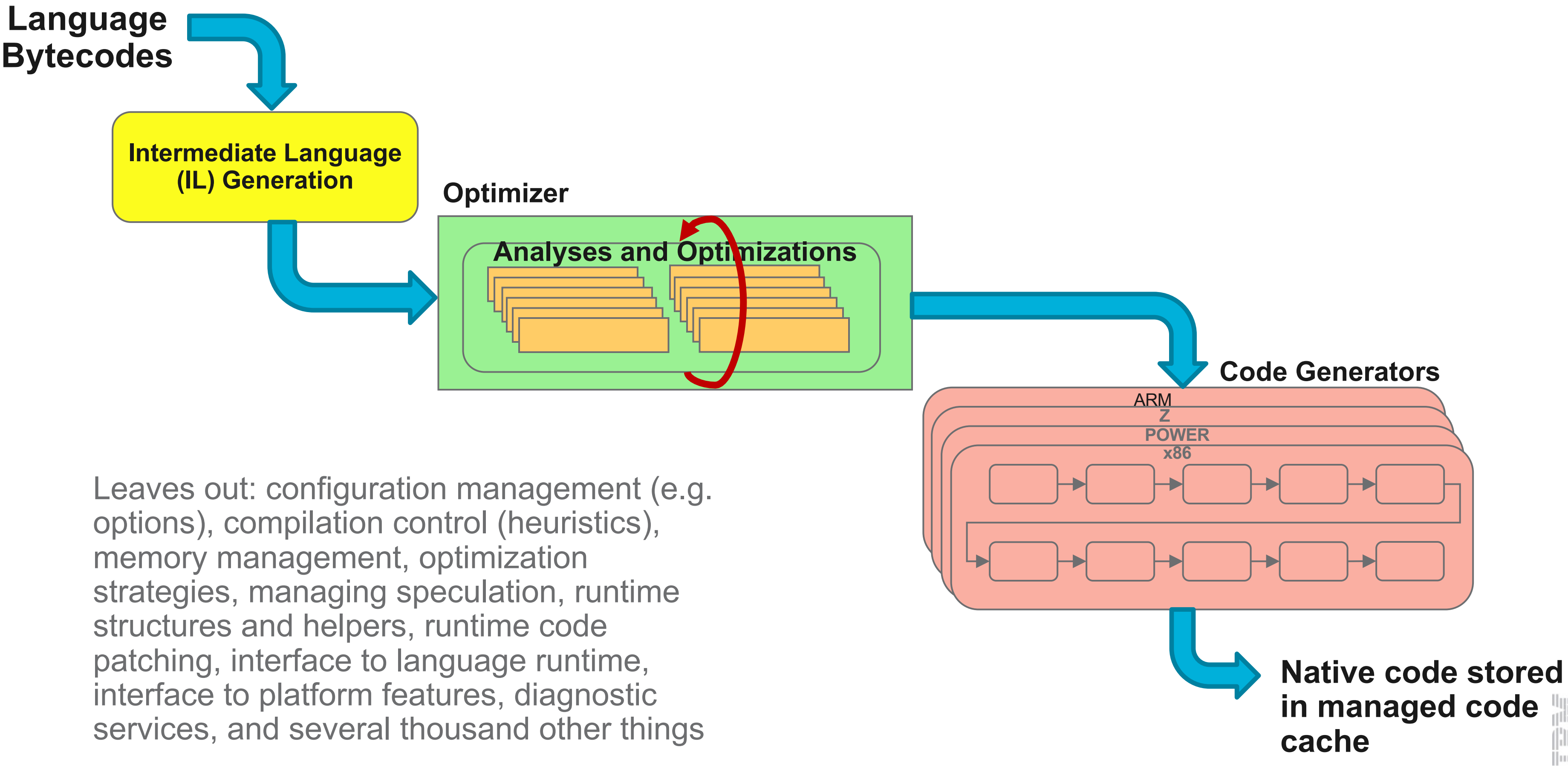
```
cd build/
ninja
```

- Run the program with your changes:

```
./b9run/b9run -function factorial test/libfactoriald.so 20
```

Base9 JIT Compiler

Simplified OMR JIT Compiler Process


Language Bytecodes
Intermediate Language (IL) Generation
Optimizer
Analyses and Optimizations
Code Generators
ARM
Z
POWER
x86
Native code stored in managed code cache

Leaves out: configuration management (e.g. options), compilation control (heuristics), memory management, optimization strategies, managing speculation, runtime structures and helpers, runtime code patching, interface to language runtime, interface to platform features, diagnostic services, and several thousand other things

Base9 JIT Compilation

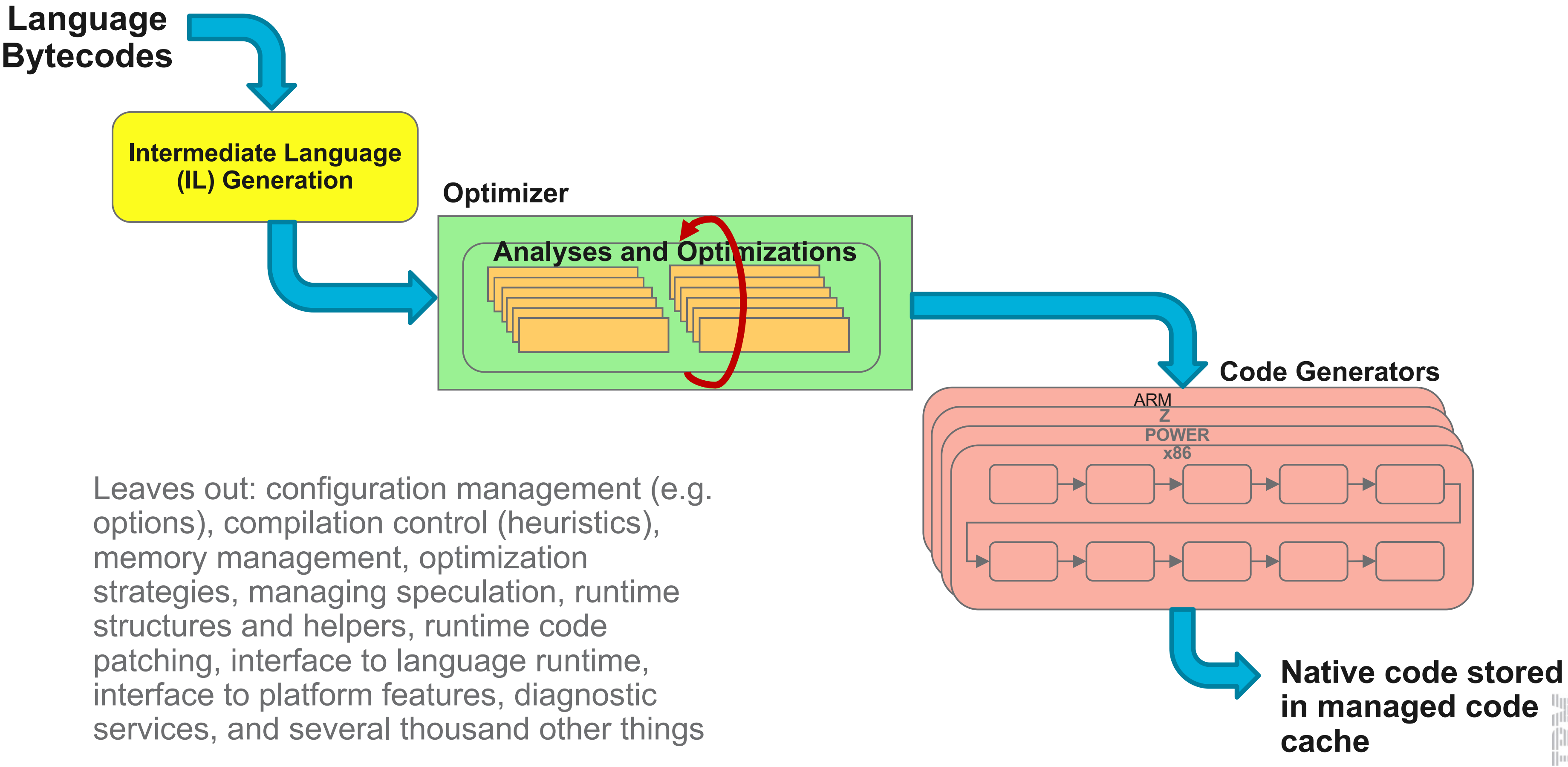
- The Base9 JIT uses a loop and switch statement, just like the interpreter

- Instead of simply interpreting Base9 bytecodes, the Base9 JIT uses OMR JitBuilder to compile the bytecodes to OMR Intermediate Language (IL)

- Various optimizations are performed on the generated IL before native code is generated for our Base9 program

- The Base9 JIT returns a pointer to our natively compiled function

Simplified OMR JIT Compiler Process


Language Bytecodes
Intermediate Language (IL) Generation
Optimizer
Analyses and Optimizations
Code Generators
ARM
Z
POWER
x86
Native code stored in managed code cache

Leaves out: configuration management (e.g. options), compilation control (heuristics), memory management, optimization strategies, managing speculation, runtime structures and helpers, runtime code patching, interface to language runtime, interface to platform features, diagnostic services, and several thousand other things

JitBuilder

- JitBuilder allows you to programmatically describe the Intermediate Language (IL) that implements the semantics of each bytecode in your language runtime

- Designed to bootstrap a native code JIT compiler for interpreted methods

- Provides sensible defaults for compiler configuration and setup

- A general cross platform native code generation toolkit
  - Works on OSX as well as Linux for X86, POWER, and Z

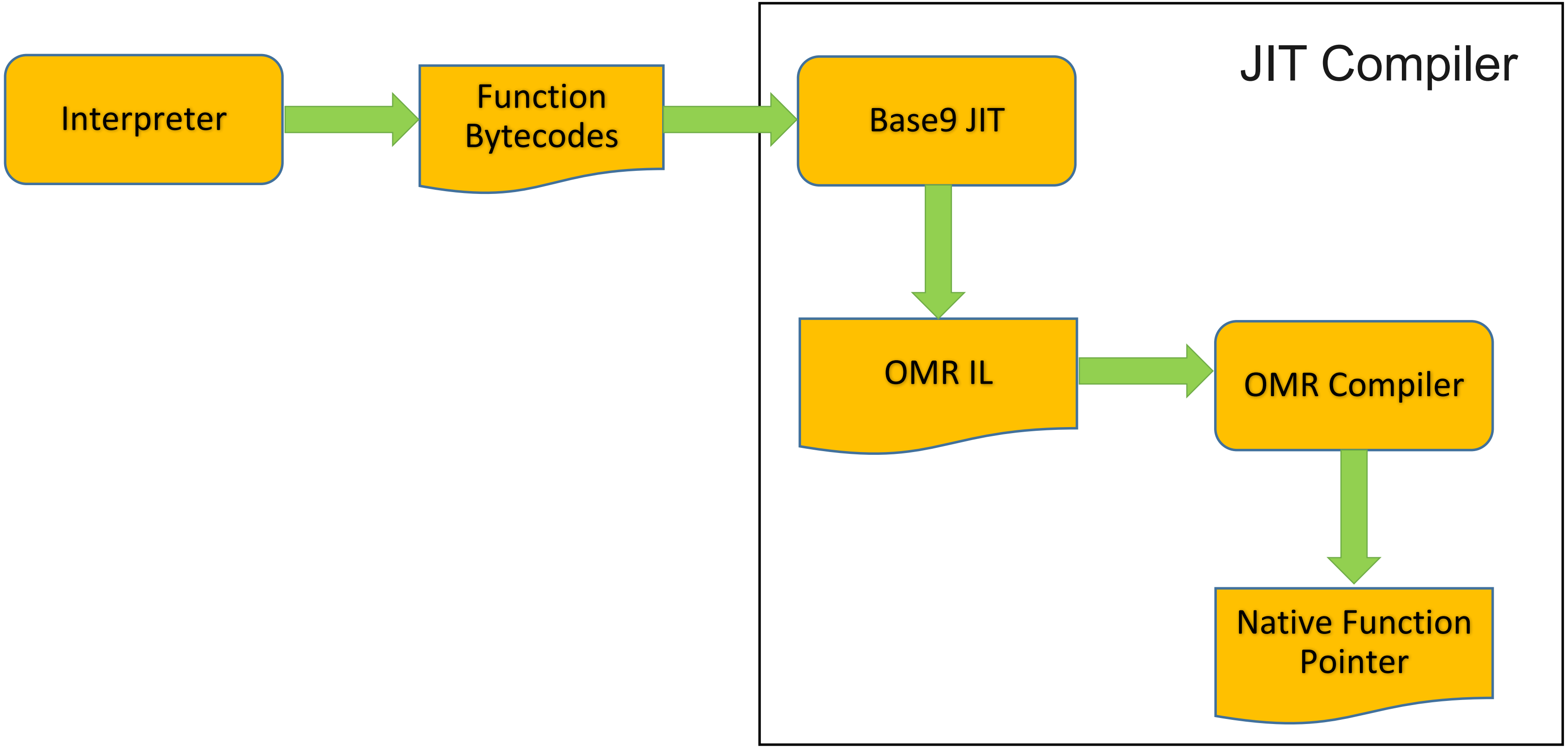
JitBuilder High-level API

- InitializeJit() sets up the OMR JIT by allocating a code cache for compiled methods

- ShutdownJit() frees the code cache of compiled methods when you are done with your native code

- MethodBuilder corresponds to a method callable with native C calling conventions taking parameters and returning a value if desired
  - Two main parts of this C++ class:
    - Constructor describes parameters, locals and return type of the method being compiled
    - buildIL() describes the code of the method being compiled in IL trees via BytecodeBuilder

Stages of Base9 JIT Compilation



JIT Compile INT_MUL

Update the Base9 JIT to compile INT_MUL

- In b9/jit.cpp

- Search “CASCON2017” and add:

```
case ByteCode::INT_MUL:
  handle_bc_mul(builder, nextBytecodeBuilder);
  break;
```

Update the Base9 JIT to compile INT_MUL

- Again in b9/jit.cpp

- Search “CASCON2017” and add:

```
void MethodBuilder::handle_bc_mul(TR::BytecodeBuilder *builder,
                                  TR::BytecodeBuilder *nextBuilder) {
  TR::IlValue *right = pop(builder);
  TR::IlValue *left = pop(builder);

  push(builder, builder->Mul(left, right));
  builder->AddFallThroughBuilder(nextBuilder);
}
```

Update the Base9 JIT to compile INT_MUL …

- Open b9/include/b9/jit.hpp

- Search “CASCON2017” and add:

```
void handle_bc_mul(TR::BytecodeBuilder *builder,
                   TR::BytecodeBuilder *nextBuilder);
```

Test JIT changes

- If you’ve fallen behind, you can fetch the full solution from GitHub:

```
git checkout upstream/CASCON2017_Solution
git checkout -b solution
```

- Compile your changes:

```
cd build/
ninja
```

- Run the program with your changes:

```
./b9run/b9run -jit -function factorial test/libfactoriald.so 20
```

Unit Test

Add a Unit Test

- Open test/interpreter_test.src

- Search “CASCON2017” and add:

```
function test_mul() {
  var a = 4;
  var b = 6;
  if ((a * b) == 24) {
    return 1;
  }
  return 0;
}
```

Add test_mul() to Test List

- Open test/b9test.cpp

- Search “`CASCON2017`” and add:

```
"test_mul",
```

Run Unit Test

- Compile our changes:

```
cd build
ninja
```

- Run Base9 test suite:

```
ctest -V -R run_b9test
```

Success! You are DONE!

Performance Improvements

Running the Interpreter and the JIT

- Run with only the interpreter:

  ```
  time ./b9run/b9run -loop 1000 -function fib test/libfib.so 20 > log
  ```

- Add **-jit**:

  ```
  time ./b9run/b9run -loop 1000 -jit -function fib test/libfib.so 20 > log
  ```

JIT Features

- Direct Call
  - Allows us to check whether or not the function we are calling has been JITed and then jump directly to the JITed function, bypassing the interpreter
- Pass Param
  - Pass Param allows JIT compiled methods calling other JIT compiled methods to pass their parameters using C native calling conventions
- Lazy VM State
  - Simulates the interpreter stack while running in a compiled method and restores the interpreter stack when returning into the interpreter

Running with JIT Features

- Add **-directcall**:

  ```
  time ./b9run/b9run -loop 1000 -jit -directcall -function fib test/libfib.so 20 > log
  ```

- Add **-passparam**:

  ```
  time ./b9run/b9run -loop 1000 -jit -directcall -passparam -function fib test/libfib.so 20 > log
  ```

- Add **-lazyvmstate**:

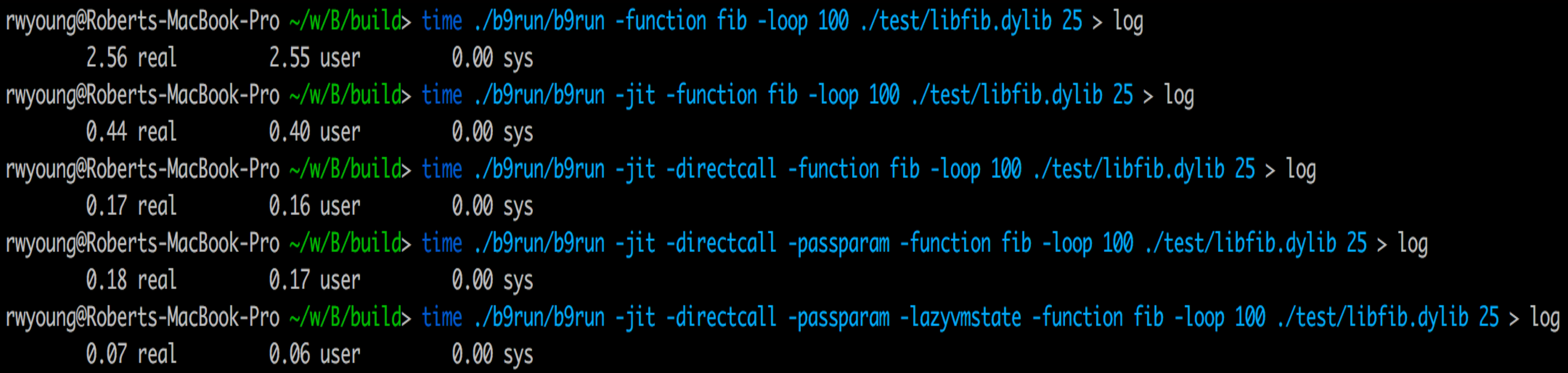
  ```
  time ./b9run/b9run -loop 1000 -jit -directcall -passparam -lazyvmstate -function fib test/libfib.so 20 > log
  ```

Performance Comparison

- Are the interpreter and the JIT getting the same results?

- What is the speedup?

JIT vs. Interpreter Performance



```
rwyoung@Roberts-MacBook-Pro ~/w/B/build> time ./b9run/b9run -function fib -loop 100 ./test/libfib.dylib 25 > log
        2.56 real         2.55 user         0.00 sys
rwyoung@Roberts-MacBook-Pro ~/w/B/build> time ./b9run/b9run -jit -function fib -loop 100 ./test/libfib.dylib 25 > log
        0.44 real         0.40 user         0.00 sys
rwyoung@Roberts-MacBook-Pro ~/w/B/build> time ./b9run/b9run -jit -directcall -function fib -loop 100 ./test/libfib.dylib 25 > log
        0.17 real         0.16 user         0.00 sys
rwyoung@Roberts-MacBook-Pro ~/w/B/build> time ./b9run/b9run -jit -directcall -passparam -function fib -loop 100 ./test/libfib.dylib 25 > log
        0.18 real         0.17 user         0.00 sys
rwyoung@Roberts-MacBook-Pro ~/w/B/build> time ./b9run/b9run -jit -directcall -passparam -lazyvmstate -function fib -loop 100 ./test/libfib.dylib 25 > log
        0.07 real         0.06 user         0.00 sys
```

6x → 15x → 36x

Future Work

Future Work

- Add object model and garbage collector
- Support more of Javascript language
- Add threading
- Provide better optimizations tailored for Base9

Please get involved!

Acknowledgements

- John Duimovich
- Andrew Young
- Mark Stoodley
- Daryl Maier

Thank you!

Base9 repository: https://github.com/youngar/Base9/

OMR repository: https://github.com/eclipse/omr

Back-up Slides

Base9 Language to Bytecode

```
function test_add() {
  var a = 1;
  var b = 2;
  if ((a + b) == 3) {
    return 1;
  }
  return 0;
}
```

Base9 Language

Base9 Front-End Compiler Step

```
Instruction test_add[] = {
  {ByteCode::INT_PUSH_CONSTANT, 1},
  {ByteCode::POP_INTO_VAR, 0},
  {ByteCode::INT_PUSH_CONSTANT, 2},
  {ByteCode::POP_INTO_VAR, 1},
  {ByteCode::PUSH_FROM_VAR, 0},
  {ByteCode::PUSH_FROM_VAR, 1},
  {ByteCode::INT_ADD, 0},
  {ByteCode::INT_PUSH_CONSTANT, 3},
  {ByteCode::INT_JMP_NEQ, 2},
  {ByteCode::INT_PUSH_CONSTANT, 1},
  {ByteCode::FUNCTION_RETURN, 0},
  {ByteCode::INT_PUSH_CONSTANT, 0},
  {ByteCode::FUNCTION_RETURN, 0},
  END_SECTION
};
```

Bytecodes

Direct Call

- When we generate code for a function call, we use the interpreter to find out whether or not the method has been JIT compiled

- Direct Call allows us to check whether or not the function we are calling has been JITed and then jump directly to the JITed function, bypassing the interpreter

Pseudo Code:

```
if(functionToCall == currentFunction ||
  functionToCall.hasBeenJitted) {
  generateCallTo(functionToCall);
} else
  generateCallTo(interpreter);
}
```

Pass Param

- Base9 methods pass parameters on the stack
- The interpreter calls compiled code using a C calling convention (i.e. pop parameters off the stack, and pass them through a C call)
- Pass Param allows JIT compiled methods calling other JIT compiled methods to pass their parameters using C native calling conventions, bypassing the interpreter

Pass Param

- Old calling convention:

```
push(1);
push(2);
result = *compiledFunction(stack, stackPointer);
```

- Calling convention with Pass Param:

```
result = *compiledFunction(1, 2);
```

- Pseudo Code:

```
if (functionToCall.isCompiled()) {
  // pop from the simulated stack
  TR::IlValue * p1 = pop(builder);
  // Return values still on the stack, commit it to to the simulated stack
  result = builder->Call(functionToCall, argC, p1);
}
```

VirtualMachineState

- The generated bytecode must always push on and pop from the interpreter’s stack

- What if we could keep the values in registers, and omit the pushes and pops to the stack?

- Problem:
  - When we call another function, we re-enter the interpreter
  - The interpreter’s stack must be correct and up to date, so how can we omit stack pushes and pops?

- Solution:
  - Simulate the interpreter’s stack while running in a compiled method and restore the correct stack when returning into the interpreter

VirtualMachineState

- Call Commit() before returning to the interpreter from a compiled method
  - Store values from the simulated stack into the interpreter’s stack

- Call Reload() before leaving the interpreter to go to a compiled method
  - Reloads values from the interpreter’s stack into the simulated stack

- VirtualMachineState can be enabled by a flag

INT_DIV Full Solution

Now Implement Divide

- See if you can follow the multiply example to implement divide
- Remember to search for the CASCON2017 eye catchers
- Good luck!

Divide Solution – interpreter changes

- Update the JavaScript compiler in js_compiler/compile.js:

```
if (decl.operator == "/") {
  this.outputInstruction("ByteCode::INT_DIV", 0, "");
  this.currentFunction.pushN(-1); // pop 2, push 1
  return decl.operator;
}
```

- Update the interpreter in b9/include/b9/bytecodes.hpp:

```
// Dividetwo integers
INT_DIV = 0xc,
```

and ...

```
case ByteCode::INT_DIV:
  return "INT_DIV";
```

Divide Solution – interpreter changes

- Update the interpreter in b9/include/b9.hpp:

```
void intDiv();
```

Divide Solution – interpreter changes

- Update the interpreter in b9/core.cpp

```
void ExecutionContext::intDiv() {
  StackElement right = pop();
  StackElement left = pop();
  StackElement result = left / right;
  push(result);
}
```

and...

```
case ByteCode::INT_DIV:
  intDiv();
  break;
```

Divide Solution – interpreter changes

- Add a test case in test/interpreter_test.src

```
function test_div() {
  var a = 24;
  var b = 6;
  if ((a / b) == 4) {
    return 1;
  }
  return 0;
}
```

Divide Solution – test interpreter changes

- Open test/b9test/cpp

- Search “`CASCON2017`” and add:

```
"test_div",
```

- Compile our changes:
  - `cd build`
  - `ninja`

- Run Base9 test suite:
  - Run `ctest -V -R run_b9test`

Divide Solution – JIT changes

- In b9/include/b9/jit.hpp, add:

```
void handle_bc_div(TR::BytecodeBuilder *builder,
                   TR::BytecodeBuilder *nextBuilder);
```

Divide Solution – JIT changes

- Update the JIT in b9/jit.cpp

```
void MethodBuilder::handle_bc_div(TR::BytecodeBuilder *builder,
                                  TR::BytecodeBuilder *nextBuilder) {
  TR::IlValue *right = pop(builder);
  TR::IlValue *left = pop(builder);

  push(builder, builder->Div(left, right));
  builder->AddFallThroughBuilder(nextBuilder);
}
```

Divide Solution – JIT changes

- Again in b9/jit.cpp
- Search “CASCON2017” and add:

```
case ByteCode::INT_DIV:
  handle_bc_add(builder, nextBytecodeBuilder);
  break;
```

Divide Solution – test JIT changes

- Compile your changes:

```
cd build/
ninja
```

- Run the program with your changes:

```
./b9run/b9run -jit –function test/libfactoriald.so
```

- Run Base9 test suite:

```
ctest -V -R run_b9test
```

Eclipse OMR
Created March 2016

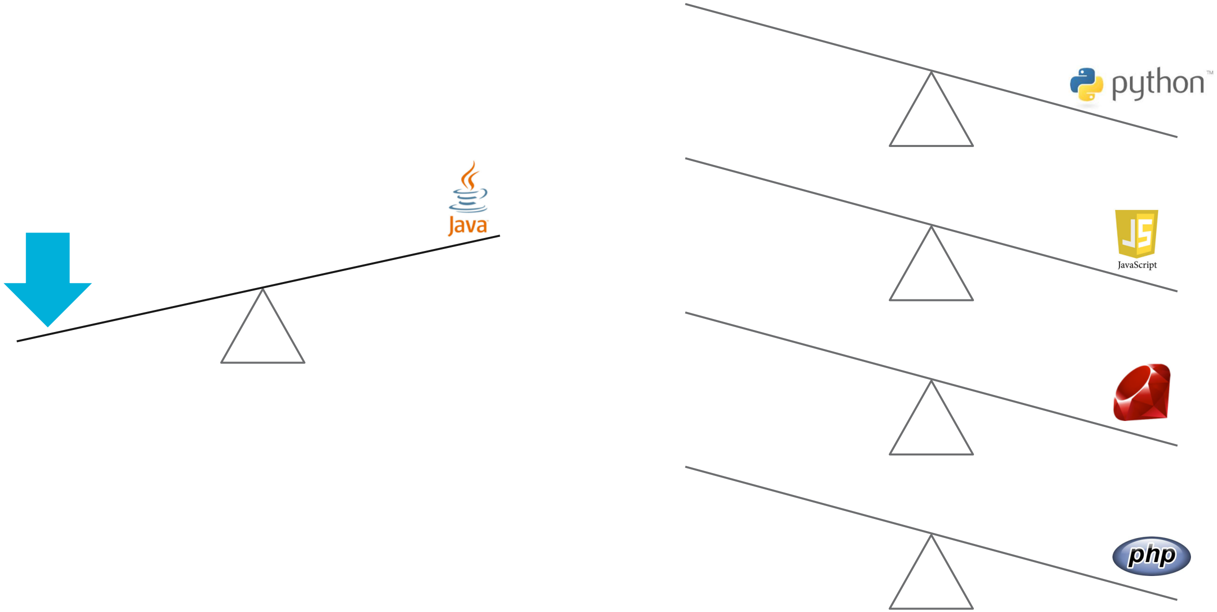
http://www.eclipse.org/omr
https://github.com/eclipse/omr
https://developer.ibm.com/open/omr/

Dual License:
Eclipse Public License v2.0
Apache 2.0

Users and contributors very welcome

https://github.com/eclipse/omr/blob/master/CONTRIBUTING.md

Motivation



- Generally, investment in one runtime has almost to zero carry over to other runtimes
- Making reusable tools is increasingly important as workloads move into the cloud