# Eclipse OMR

**Reliable Shared Components for High Performance Language Runtimes**

**Focus on JitBuilder**


Mark Stoodley, Project Technical Lead

John Duimovich, IBM Distinguished Engineer & OMR Champion

Shelley Lambert, OMR Software Developer & QA Lead

Andrew Young, OMR Software Developer

Kim Briggs, OMR Software Developer

Sophia Guo, OMR Software Developer

# This Talk

1. Update on Eclipse OMR project
   - https://github.com/eclipse/omr
   - OMR published on Github Spring 2016
   - OMR JIT published Fall 2016
2. Introduction to OMR JitBuilder
   - Native code generation toolkit using OMR JIT
   - Available in a Docker image if you want to play with it!
3. Experimenting with JitBuilder and Base9
   - Tutorial language with ~10 opcodes
   - Illustrates basic JitBuilder concepts, invites you to play  with JitBuilder

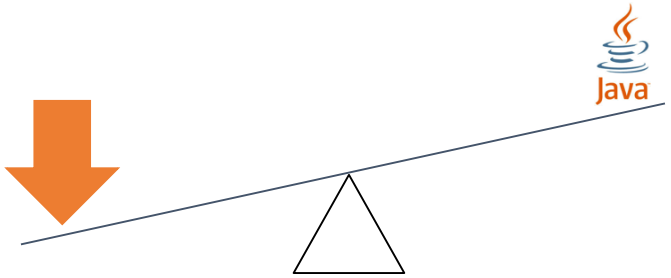
# Get ready…

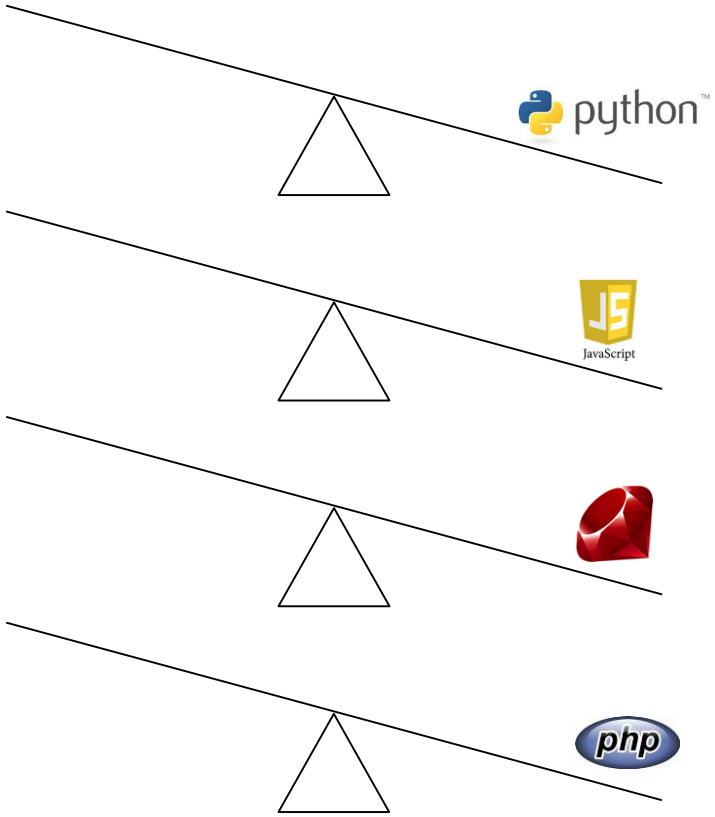
## Instructions for downloading Base9

- From docker.io:
  - docker –it jduimovich/b9 (pre-built)

- From GitHub: (need to make omr subcomponents)
  - git clone https://github.com https://github.com/aryoung/Base9

# Motivation: language runtimes similar but different technology




Investment in one runtime has close to zero carry over to other runtimes

Matters more as more workloads move into the cloud

# Eclipse OMR Mission

Build an open reusable language runtime foundation for cloud platforms

- To accelerate advancement and innovation
- In full cooperation with existing language communities
- Engaging a diverse community of people interested in language runtimes
  - Professional developers
  - Researchers
  - Students
  - Hobbyists

# Key Goals for Eclipse OMR

1. OMR has ***no*** language semantics

2. OMR is for building language runtimes
   but is ***not*** itself a language runtime

3. OMR components can be independently
   integrated into ***any*** language runtime,
   new or existing, ***without*** influencing
   language semantics

# Eclipse OMR Components

| | |
|---|---|
| port | platform abstraction (porting) library |
| thread | cross platform pthread-like threading library |
| vm | APIs to manage per-interpreter and per-thread contexts |
| gc | garbage collection framework for managed heaps |
| omrtrace | library for publishing trace events for monitoring/diagnostics |
| omrsigcompat | signal handling compatibility library |
| example | demonstration code to show how a language runtime might consume OMR components, also used for testing |
| fvtest | language independent test framework built on the example glue so that components can be tested outside of a language runtime |
| *JitBuilder, Jit Compiler* | Just-in-Time compilers made easy |

…400KLOC at this point, more components coming!

# 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 V1.0
Apache 2.0

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

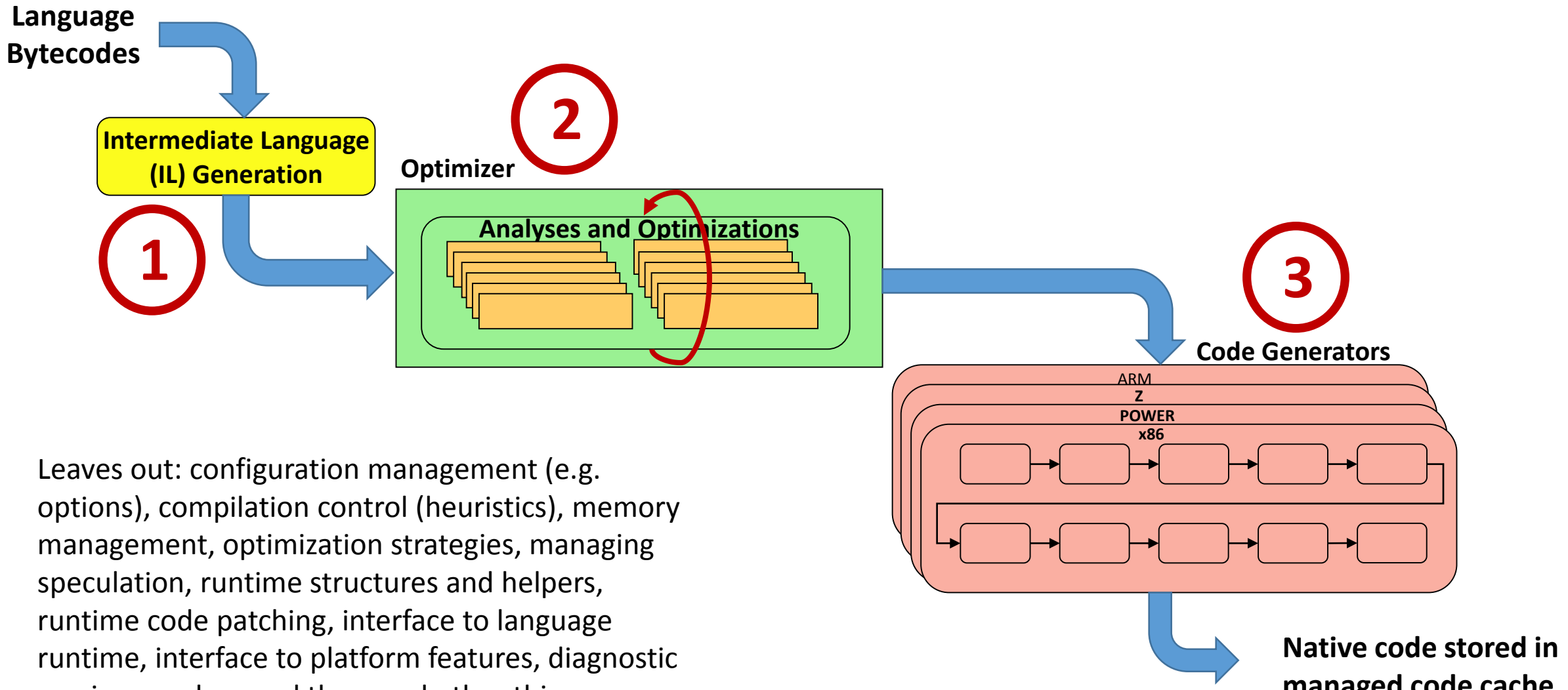
# JitBuilder Library

Just In Time (JIT) Compilers
Made Easy(er)

# What is JitBuilder?

- Interface to the OMR JIT compiler technology
  - Designed to simplify bootstrapping a JIT compiler for interpreted methods
  - Really a general native code generation toolkit
- Pre-release JitBuilder Docker image available via Box.com download:
  - https://ibm.box.com/v/JitBuilder-x86-64-image
- Ease into it with this first blog article:
  - https://developer.ibm.com/open/2016/07/19/jitbuilder-library-and-eclipse-omr-just-in-time-compilers-made-easy/
- Follow-up with recent update on DeveloperWorks
  - https://developer.ibm.com/open/events/dw-open-tech-talk-eclipse-omr-update/

# Simplified OMR Compiler Process




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

# Simple API

- InitializeJit() gets the OMR JIT ready to go
  - Among other things, allocates a code cache into which compiled methods go
- ShutdownJit() when you're done with your native code
  - It will free the code cache, so compiled methods go away after this call
- Then there's the compiling part
  - Need to write a *MethodBuilder*

# Huh, MethodBuilder? What?

- MethodBuilder corresponds to a method callable with system linkage taking whatever parameters you want, returning a value if you want

- 2 basic parts to this C++ class:
  - Constructor describes return and parameter types
  - ::buildIL() describes the method's code

- MethodBuilder class provides services to inject code operations

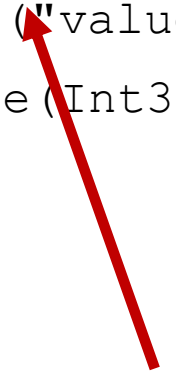
# Simple MethodBuilder Example

```
SimpleMB::SimpleMB(TR::TypeDictionary *d)
   : MethodBuilder(d)
   {
   DefineName("increment");
   DefineParameter("value", Int32);
   DefineReturnType(Int32);
   }
```

```
bool
SimpleMB::buildIL()
   {
   Return(
      Add(
         Load("value"),
         ConstInt32(1)));
   return true;
   }
```

# Simple MethodBuilder Example

```
SimpleMB::SimpleMB(TR::TypeDictionary *d)
   : MethodBuilder(d)
   {
   DefineName("increment");
   DefineParameter("value", Int32);
   DefineReturnType(Int32);
   }
```



Set the name of the function

```
bool
SimpleMB::buildIL()
   {
   Return(
      Add(
         Load("value"),
         ConstInt32(1)));
   return true;
   }
```

# Simple MethodBuilder Example

```
SimpleMB::SimpleMB(TR::TypeDictionary *d)
   : MethodBuilder(d)
   {
   DefineName("increment");
   DefineParameter("value", Int32);
   DefineReturnType(Int32);
   }
```

```
bool
SimpleMB::buildIL()
   {
   Return(
      Add(
         Load("value"),
         ConstInt32(1)));
   return true;
   }
```

Define a 32-bit integer parameter called “value”

# Simple MethodBuilder Example

```
SimpleMB::SimpleMB(TR::TypeDictionary *d)
   : MethodBuilder(d)
   {
   DefineName("increment");
   DefineParameter("value", Int32);
   DefineReturnType(Int32);
   }
```

increment will return a 32-bit integer

```
bool
SimpleMB::buildIL()
   {
   Return(
      Add(
         Load("value"),
         ConstInt32(1)));
   return true;
   }
```

# Simple MethodBuilder Example

```
SimpleMB::SimpleMB(TR::TypeDictionary *d)
    : MethodBuilder(d)
    {
    DefineName("increment");
    DefineParameter("value", Int32);
    DefineReturnType(Int32);
    }
```

Used to record and lookup types (e.g. Int32)

```
bool
SimpleMB::buildIL()
    {
    Return(
       Add(
          Load("value"),
          ConstInt32(1)));
    return true;
    }
```

# Simple MethodBuilder Example

```
SimpleMB::SimpleMB(TR::TypeDictionary *d)
   : MethodBuilder(d)
   {
   DefineName("increment");
   DefineParameter("value", Int32);
   DefineReturnType(Int32);
   }
```

```
bool
SimpleMB::buildIL()
   {
   Return(
      Add(
         Load("value"),
         ConstInt32(1)));
   return true;
   }
```

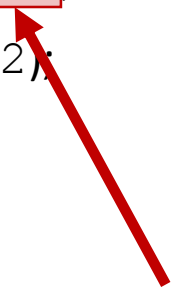
Describes what code to generate for increment

Construct expression trees from simple operations

# Simple MethodBuilder Example

```
SimpleMB::SimpleMB(TR::TypeDictionary *d)
   : MethodBuilder(d)
   {
   DefineName("increment");
   DefineParameter("value", Int32);
   DefineReturnType(Int32);
   }
```

```
bool
SimpleMB::buildIL()
   {
   Return(
      Add(
         Load("value"),
         ConstInt32(1)));
   return true;
   }
```



Reference parameters by name e.g. “value”
You can also create new locals by name

# Operations are (mostly) typeless

- You say “Add” not “Add 32-bit integers”
  - Jit Builder will derive type from operands
  - Leaves of expression trees are typically Load or Const operations
- Current exceptions are things like: LoadAt, StoreAt, IndexAt

- Types of params and/or locals described in constructor
- Every Store to a new variable assigns its type (cannot change)
  - Store to a new name creates a new slot in the native stack frame

# Create IlBuilders for new code paths

```
// if-then-else construct
IlBuilder *thenPath=OrphanBuilder();
IlBuilder *elsePath=OrphanBuilder();
IfThenElse(&thenPath, &elsePath,
   LessThan(Load("a"), Load("b"));

// put then operations into thenPath builder
thenPath->Store("T",
thenPath->   ConstInt32(1));

// put else operations into elsePath builder
elsePath->Store("T",
elsePath->   ConstInt32(0));

// code after merge goes into "this" builder
Return(
   Load("T"));
```

# Create IlBuilders for new code paths

```
// if-then-else construct
IlBuilder *thenPath=OrphanBuilder();
IlBuilder *elsePath=OrphanBuilder();
IfThenElse(&thenPath, &elsePath,
   LessThan(Load("a"), Load("b"));

// put then operations into thenPath builder
thenPath->Store("T",
thenPath->   ConstInt32(1));

// put else operations into elsePath builder
elsePath->Store("T",
elsePath->   ConstInt32(0));

// code after merge goes into "this" builder
Return(
   Load("T"));
```

Creates two new code paths (thenPath, elsePath)

# Create IlBuilders for new code paths

```
// if-then-else construct
IlBuilder *thenPath=OrphanBuilder();
IlBuilder *elsePath=OrphanBuilder();
IfThenElse(&thenPath, &elsePath,
   LessThan(Load("a"), Load("b"));

// put then operations into thenPath builder
thenPath->Store("T",
thenPath->   ConstInt32(1));

// put else operations into elsePath builder
elsePath->Store("T",
elsePath->   ConstInt32(0));

// code after merge goes into "this" builder
Return(
   Load("T"));
```

IfThenElse connects the new code paths in a diamond, Decide based on (a < b)

# Create IlBuilders for new code paths

```
// if-then-else construct
IlBuilder *thenPath=OrphanBuilder();
IlBuilder *elsePath=OrphanBuilder();
IfThenElse(&thenPath, &elsePath,
   LessThan(Load("a"), Load("b"));

// put then operations into thenPath builder
thenPath->Store("T",
thenPath->   ConstInt32(1));

// put else operations into elsePath builder
elsePath->Store("T",
elsePath->   ConstInt32(0));

// code after merge goes into "this" builder
Return(
   Load("T"));
```

Inject operations directly onto the code path where they should execute

# Create IlBuilders for new code paths

```
// if-then-else construct
IlBuilder *thenPath=OrphanBuilder();
IlBuilder *elsePath=OrphanBuilder();
IfThenElse(&thenPath, &elsePath,
   LessThan(Load("a"), Load("b"));

// put then operations into thenPath builder
thenPath->Store("T",
thenPath->   ConstInt32(1));

// put else operations into elsePath builder
elsePath->Store("T",
elsePath->   ConstInt32(0));

// code after merge goes into "this" builder
Return(
   Load("T"));
```

On then path, T=1

# Create IlBuilders for new code paths

```
// if-then-else construct
IlBuilder *thenPath=OrphanBuilder();
IlBuilder *elsePath=OrphanBuilder();
IfThenElse(&thenPath, &elsePath,
   LessThan(Load("a"), Load("b"));

// put then operations into thenPath builder
thenPath->Store("T",
thenPath->   ConstInt32(1));

// put else operations into elsePath builder
elsePath->Store("T",
elsePath->   ConstInt32(0));

// code after merge goes into "this" builder
Return(
   Load("T"));
```

On else path, T=0

# Create IlBuilders for new code paths

```
// if-then-else construct
IlBuilder *thenPath=OrphanBuilder();
IlBuilder *elsePath=OrphanBuilder();
IfThenElse(&thenPath, &elsePath,
   LessThan(Load("a"), Load("b"));

// put then operations into thenPath builder
thenPath->Store("T",
thenPath->   ConstInt32(1));

// put else operations into elsePath builder
elsePath->Store("T",
elsePath->   ConstInt32(0));

// code after merge goes into "this" builder
Return(
   Load("T"));
```

Operations are appended to the code path you specify
Return(Load("T")) happens after the merge for IfThenElse

# Create IlBuilders for new code paths

```
// if-then-else construct
IlBuilder *thenPath=OrphanBuilder();
IlBuilder *elsePath=OrphanBuilder();
IfThenElse(&thenPath, &elsePath,
   LessThan(Load("a"), Load("b"));

// put then operations into thenPath builder
thenPath->Store("T",
thenPath->   ConstInt32(1));

// put else operations into elsePath builder
elsePath->Store("T",
elsePath->   ConstInt32(0));
```

```
// else path returns, but then path does not
elsePath->Return(
elsePath->   Load("T"));
```

Return(Load("T")) directly from the elsePath

# Other Control Flow

- Loops: ForLoops, WhileDo, DoWhile
  - For loop variants that count up or down
  - All loop variants can handle break and continue
- IfThen (i.e. no else)
- Switch

# TypeDictionary for managing types

- Primitive types: NoType, Int8, Int16, Int32, Int64, Float, Double
- Pointer (array) types: pInt8, pInt16, pInt32, pInt64, pFloat, pDouble
- Can define structures with fields
  - Then LoadIndirect() or StoreIndirect() to access fields offset from a base pointer

```
TR::IlType *elementType = DefineStruct("Element");
DefineField("Element", "next", PointerTo(elementType));
DefineField("Element", "key", Int32);
DefineField("Element", "value", Double);

// load ((Element *)ptr)->key
TR::IlValue *result = LoadIndirect("Element", "key", Load("ptr"));
```

# BytecodeBuilders for easy(er) JIT writing

- Write a handler for each type of bytecode to inject code to execute that type of bytecode
- Create a BytecodeBuilder object for every bytecode index
- Iterate over bytecodes in a method:
  - Call the right handler on BytecodeBuilder object for this bytecode index
  - Tell the object how control flows out from this bytecode

    bytecodeBuilder->AddFallThroughBuilder(nextBuilderObject);
    bytecodeBuilder->AddSuccessorBuilder(branchesToBuilderObject);
- MethodBuilder must call AppendBuilder(bytecodeBuilder[0]);
- More explanation in developerworksOpen talk from July 20:
  **https://developer.ibm.com/open/videos/eclipse-omr-tech-talk/**

# What all can you generate with JitBuilder?

- Data Types
  - Primitives: Int8, Int16, Int32, Int64, Float, Double
  - Arbitrary structs of primitives
  - Arrays and pointers, but aliasing isn't yet perfect
- Arithmetic
  - Add, Sub, Mul, Div, And, Xor, ShiftL, ShiftR, UnsignedShiftR
- Conditional
  - EqualTo, NotEqualTo, LessThan, GreaterThan
- Type Conversion
  - ConvertTo
- Memory
  - Load, Store, IndexAt, LoadAt, StoreAt, LoadArray, StoreArray, LoadField, StoreField
  - VectorLoad, VectorStore, VectorLoadAt, VectorStoreAt
- Call (use DefineFunction to enable arbitrary C functions to be called)
- Control flow
  - IfThen, IfThenElse, Switch, ForLoopUp, ForLoopDown, DoWhile, WhileDo, variants with break, continue, etc.
  - Return

# What is JitBuilder really about?

- Create an incremental investment story for building JIT compilers
- JitBuilder is really the first layer in a JIT implementation strategy
  - Designed to get first JIT compiler up and running quickly
  - Lower implementation layers give you more control performance
  - But at the cost of learning the OMR JIT intermediate language

- Java JitBuilder shows how the library can be used from Java
  - But bindings could be created for just about any language
  - Probably about 2 solid days work to create functional Java bindings
  - Java bindings not yet complete, but remaining work is mechanical

# OMR Exploratory Project

Rosie Pattern Matching Language
with OMR JitBuilder

# Rosie Pattern Language v0.99

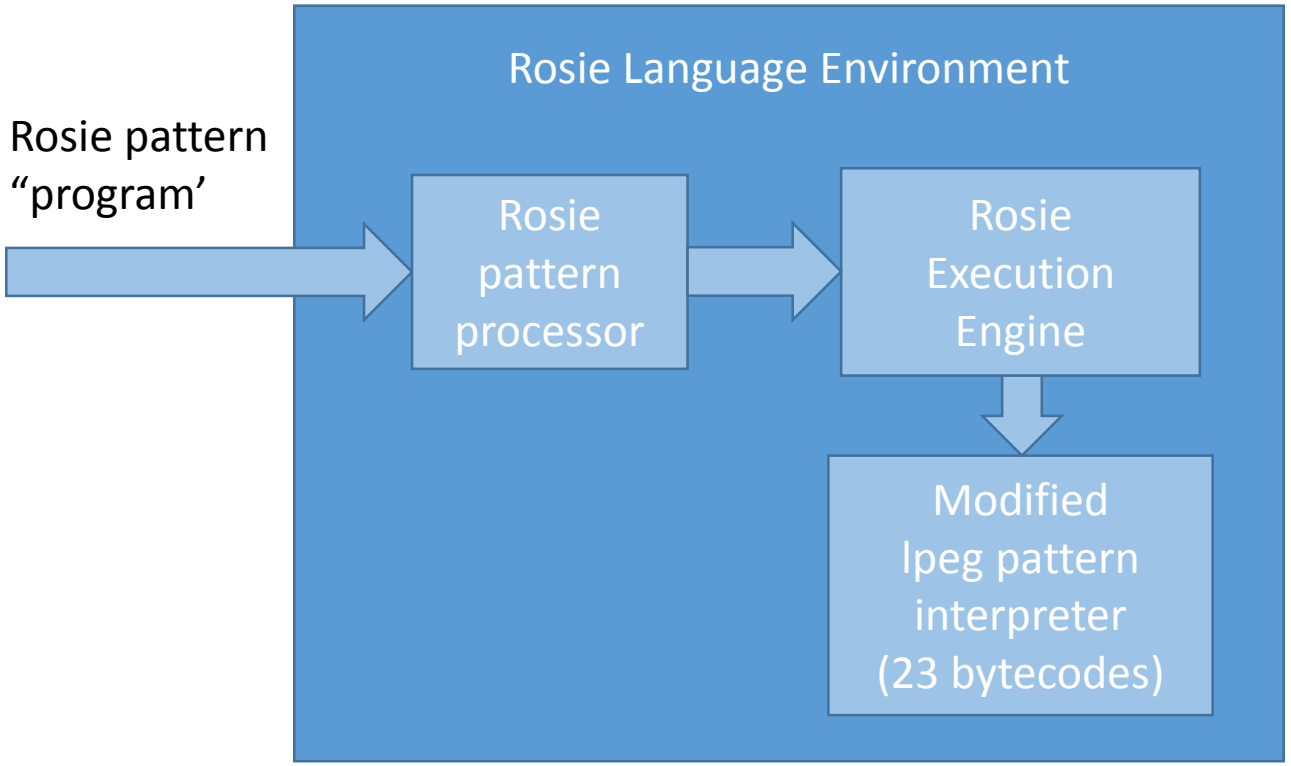
Github:
https://github.com/jamiejennings/rosie-pattern-language/

IBM developerWorks Open tutorials, blog:
https://developer.ibm.com/open/rosie-pattern-language/

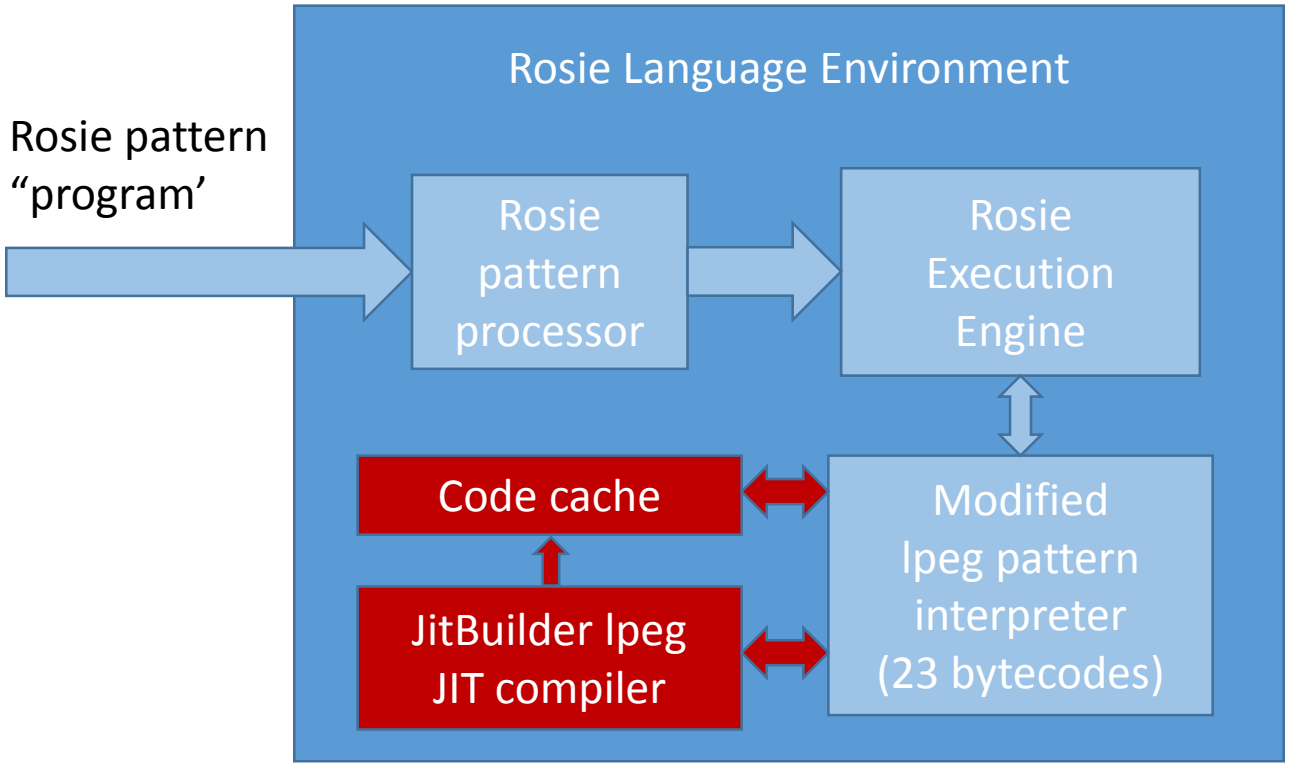
- **Problem: data mining using regex does not scale**
  - Regex not maintainable → large collections of expressions not practical
  - Regex not standard; have language dependencies → expressions not shareable
  - Most regex engines exponentially backtrack → stalls big data pipelines
- **Rosie Pattern Language overcomes regex scaling problems**
  - Designed like a programming language: understandable & maintainable
  - Packages of expressions easily shared across teams & programming languages
  - Linear time guarantee, plus access to (possibly superlinear) recursive patterns
- **Rosie Pattern Engine**
  - Small (~ 350 KB executable) and relatively fast (around 4x competition like Grok)
  - Ships with hundreds of common patterns, and built-in pattern development tools (REPL, pattern debugger)

```
jjennings$ head -4 /var/log/system.log >/tmp/syslog-snippet
jjennings$ rosie basic.matchall /tmp/syslog-snippet
Apr  8  09:42:24  Js-MacBook-Pro com.apple.xpc.launchd 1 ( homebrew.mxcl.kafka 68878 ) : Service exited with abnormal code : 1
Apr  8  09:42:24  Js-MacBook-Pro com.apple.xpc.launchd 1 ( homebrew.mxcl.kafka ) : Service only ran for 8 seconds . Pushing respawn out by 2 seconds .
Apr  8  10:10:18  Js-MacBook-Pro.local MUpdate 69707 : Endpoint at ' /Applications/Meeting.app ' is latest version ( 4732 ) , skipping .
Apr  8  10:10:18  Js-MacBook-Pro.local MUpdate 69707 : Next Update Check at 2016-04-09 02:22:03 +0000
jjennings$
jjennings$ rosie -grep 'common.word basic.network_patterns' /etc/resolv.conf
domain nc.rr.com
nameserver 10.0.1.1
jjennings$ rosie -grep -encode json 'common.word network.ip_address' /etc/resolv.conf
{"*":{"subs":[{"common.word":{"text":"nameserver","pos":1}},{"network.ip_address":{"text":"10.0.1.1","pos":12}}],"text":"nameserver 10.0.1.1","pos":1}}
```

# Simplified High Level Rosie operation

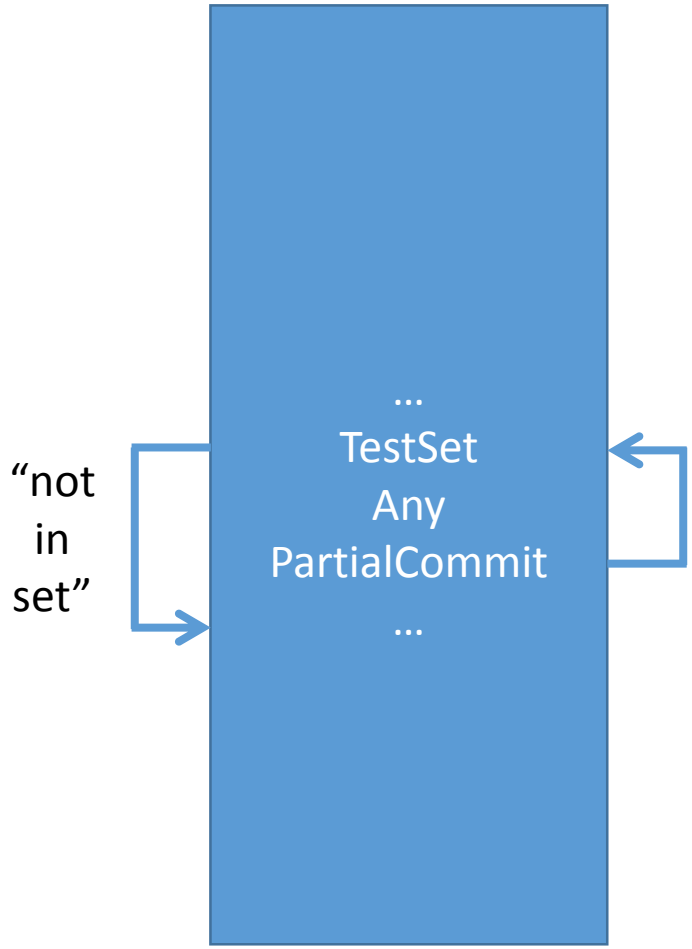


# Experiment with OMR JitBuilder: JIT compiler for lpeg bytecodes



# ”syslog” example pattern:

message = .*
syslog = datetime.datetime_RFC3339 network.ip_address common.word "["common.int"]" message




Interpreter mostly (99%) loops over 3 bytecodes:

TestSet: Is current character one of a set of characters?

Any: move current character forward

PartialCommit: advance extent of current pattern match

Full pattern:1385 bytecodes applied to each line of the log file

# Generated code for these 3 bytecodes

```
0x1082712eb                Label L0154:                          ; LABEL
```

Test Set

```
0x1082712eb    mov rdx, qword ptr [rbp-0x18]      ; L8RegMem, SymRef  s<parm 2 LInt8;>
0x1082712ef    mov r9b, byte ptr [rdx]            ; L1RegMem, SymRef  <array-shadow>
0x1082712f2    mov ecx, r9d                       ; MOV4RegReg
0x1082712f5    and ecx, 0x00000007                ; AND4RegImms
0x1082712f8    mov esi, 0x00000001                ; MOV4RegImm4
0x1082712fd    shl esi, cl                        ; SHL4RegCL
0x1082712ff    movzx r9d, r9b                     ; MOVZXReg4Reg1
0x108271303    shr r9d, 0x03                      ; SHR4RegImm1
0x108271307    mov rcx, 0x00007fb93c15c048        ; MOV8RegImm64
0x108271311    movsx r9d, byte ptr [r9+1*rcx]     ; MOVSXReg4Mem1
0x108271316    test esi, r9d                      ; TEST4RegReg
0x108271319    je  Label L0153                    ; JE4
```

Any

```
0x10827131f    cmp rdx, qword ptr [rbp-0x20]      ; CMP8RegMem, SymRef  e<parm 3 LInt8;>
0x108271323    jge Label L0153                    ; JGE4
0x108271329    mov r10, rdx                       ; MOV8RegReg
0x10827132c    add r10, 0x00000001                ; ADD8RegImms
0x108271330    mov qword ptr [rbp-0x18], r10      ; S8MemReg, SymRef  s<parm 2 LInt8;>
```

??

```
0x108271334    mov eax, dword ptr [rbp-0x8]       ; L4RegMem, SymRef  captop<auto slot 11>
0x108271337    jmp Label L0154                    ; JMP1
```

# Wrap Up

- Eclipse OMR mission to create an open reusable language runtime foundation
- Building an open community for everyone to share and discuss ideas, technology, and best practices to build language runtimes

- JitBuilder is available now for you to play with
  - Write your own JIT compiler
  - Part of a layered strategy to incrementally improve performance via JIT technology

- Everyone is welcome to join us at Eclipse OMR and we hope you will
- OMR JIT and JitBuilder are moving fast – join us on GitHub and DeveloperWorks!

# Where to contact us

- Mailing List omr-dev@eclipse.org
  Subscribe at https://dev.eclipse.org/mailman/listinfo/omr-dev

- Eclipse OMR Web Site https://www.eclipse.org/omr
- Eclipse OMR pages on DeveloperWorks Open https://developer.ibm.com/open/omr/
- Eclipse OMR Github project https://github.com/eclipse/omr
- JitBuilder Docker image https://ibm.box.com/v/JitBuilder-x86-64-image
  (Base9) docker –it jduimovich/b9

- Ruby+OMR Technology Preview Github project with Docker images for Linux on LinuxONE, OpenPOWER, and X86: https://github.com/rubyomr-preview/rubyomr-preview