Ignition: An Interpreter for V8

Ross McIlroy, Orion Hodson, Mythri Alle
Chrome Mobile Performance Team, London

Background

Life of a Script

Life of a Script in V8

```
function foo() { … }
function done() { … }
function unused() { … }

var Person = function() {
  this.name = name;
}
Person.prototype.doWork = function() {
  do { foo(); } while (!done());
}

var john = new Person(“John”);
john.doWork();
```

Life of a Script in V8

```
function foo() { … }
function done() { … }
function unused() { … }

var Person = function() {
  this.name = name;
}
Person.prototype.doWork = function() {
  do { foo(); } while (!done());
}

var john = new Person(“John”);
john.doWork();
```

Life of a Script in V8

```
function foo() { … }
function done() { … }
function unused() { … }

var Person = function() {
  this.name = name;
}
Person.prototype.doWork = function() {
  do { foo(); } while (!done());
}

var john = new Person(“John”);
john.doWork();
```

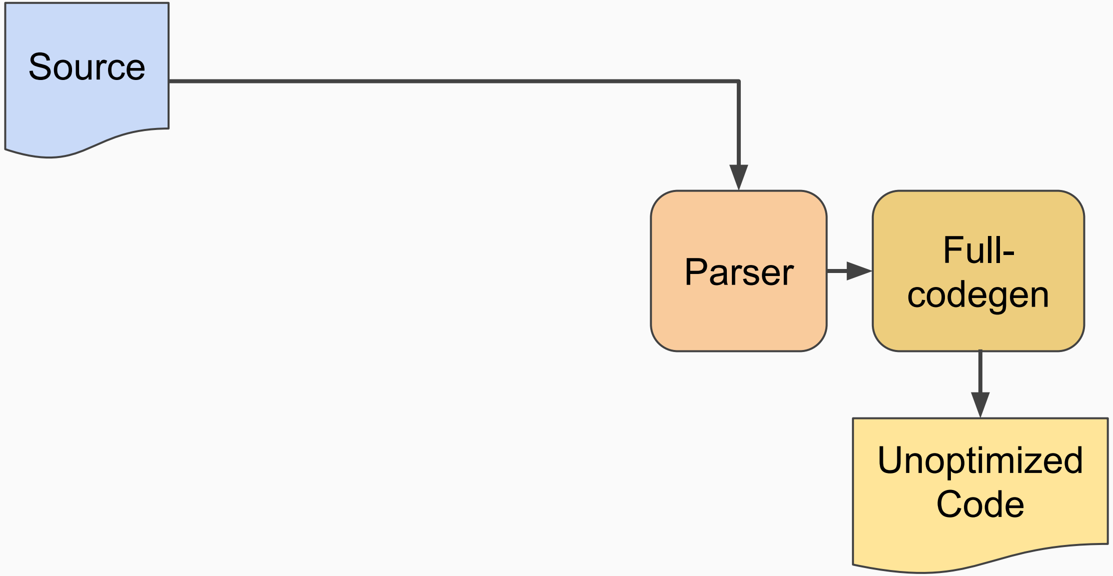
Life of a Script in V8

```
function foo() { … }
function done() { … }
function unused() { … }

var Person = function() {
  this.name = name;
}
Person.prototype.doWork = function() {
  do { foo(); } while (!done());
}

var john = new Person(“John”);
john.doWork();
```

Compiled

Top Level
00101010
10110101
...

Compiler Pipeline

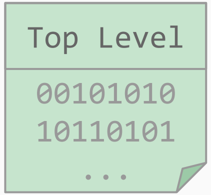


Life of a Script in V8

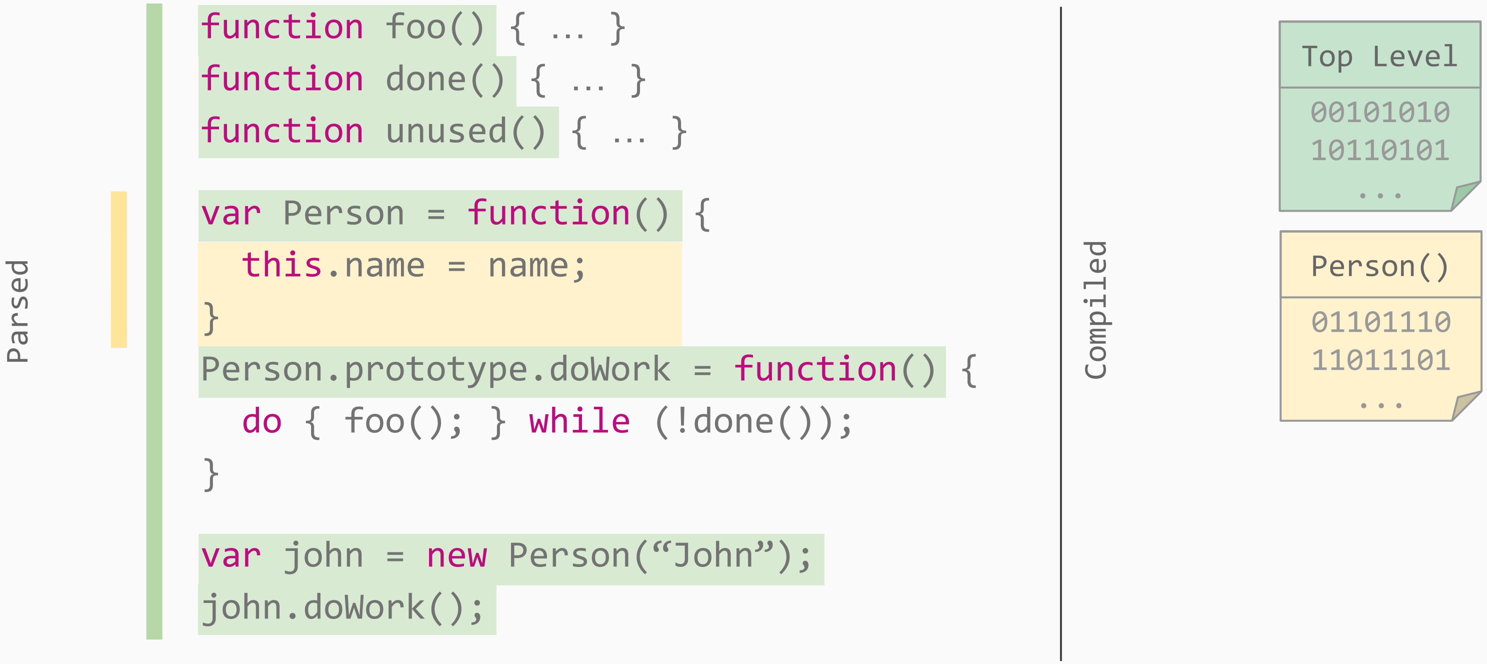
Parsed

```
function foo() { … }
function done() { … }
function unused() { … }

var Person = function() {
  this.name = name;
}
Person.prototype.doWork = function() {
  do { foo(); } while (!done());
}

var john = new Person(“John”);
john.doWork();
```

Compiled

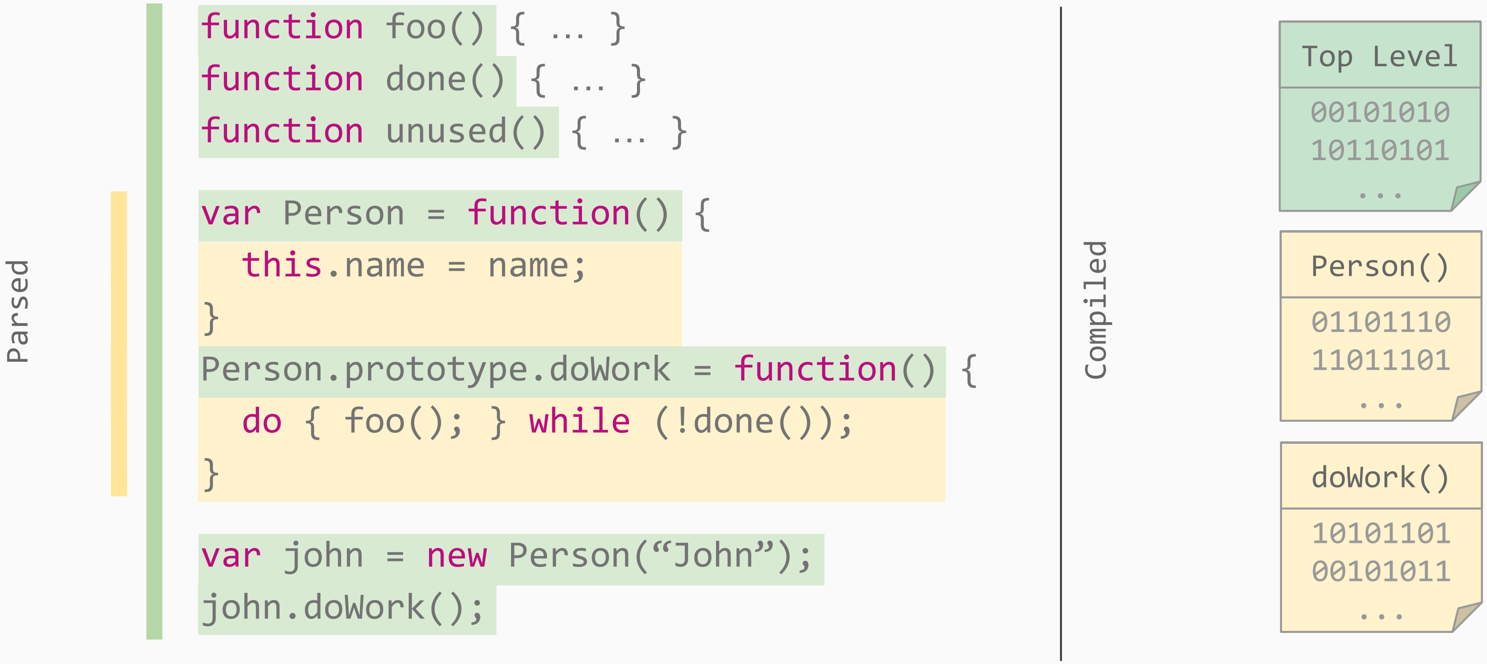


Life of a Script in V8



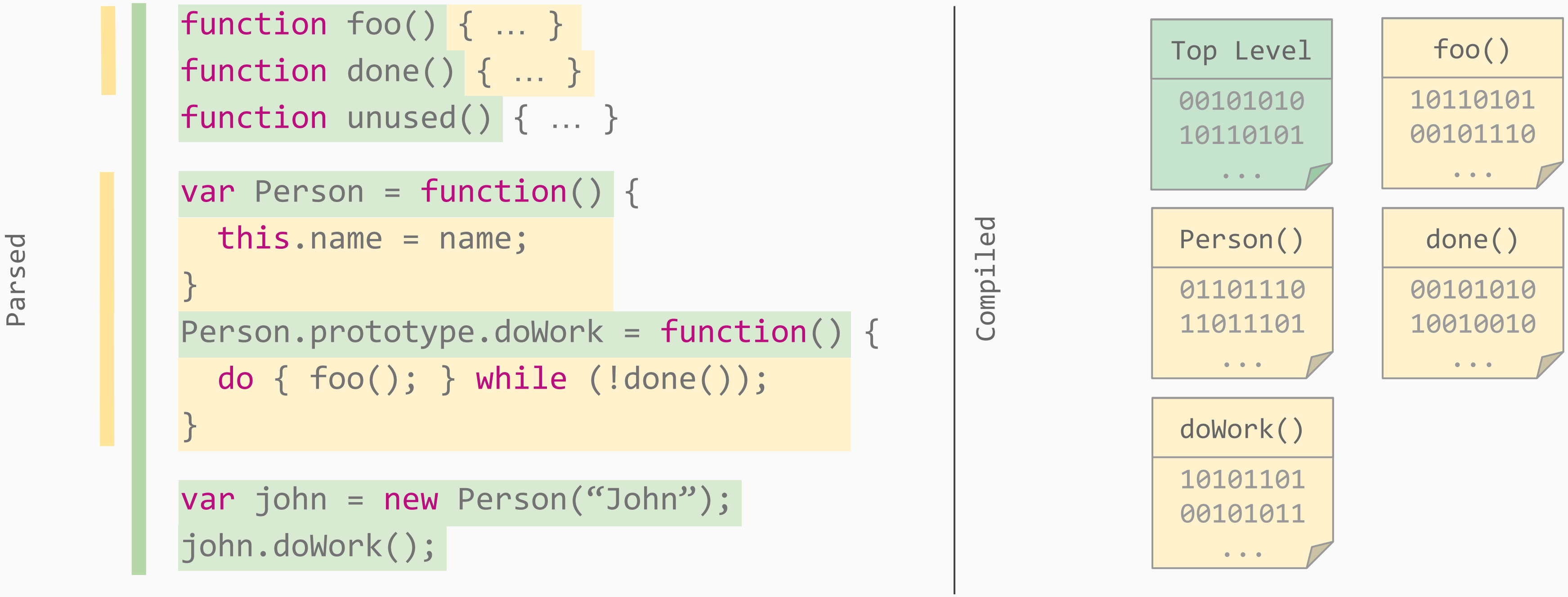
```
function foo() { … }
function done() { … }
function unused() { … }

var Person = function() {
  this.name = name;
}
Person.prototype.doWork = function() {
  do { foo(); } while (!done());
}

var john = new Person(“John”);
john.doWork();
```

Life of a Script in V8



```
function foo() { … }
function done() { … }
function unused() { … }

var Person = function() {
  this.name = name;
}
Person.prototype.doWork = function() {
  do { foo(); } while (!done());
}

var john = new Person(“John”);
john.doWork();
```

Life of a Script in V8



Parsed

```
function foo() { … }
function done() { … }
function unused() { … }

var Person = function() {
  this.name = name;
}
Person.prototype.doWork = function() {
  do { foo(); } while (!done());
}

var john = new Person(“John”);
john.doWork();
```

Compiled

Top Level

00101010
10110101
...

foo()

10110101
00101110
...

Person()

01101110
11011101
...

done()

00101010
10010010
...

doWork()

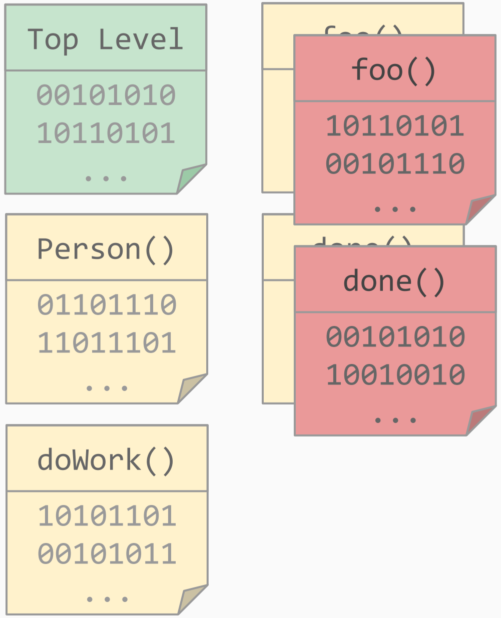
10101101
00101011
...

Life of a Script in V8

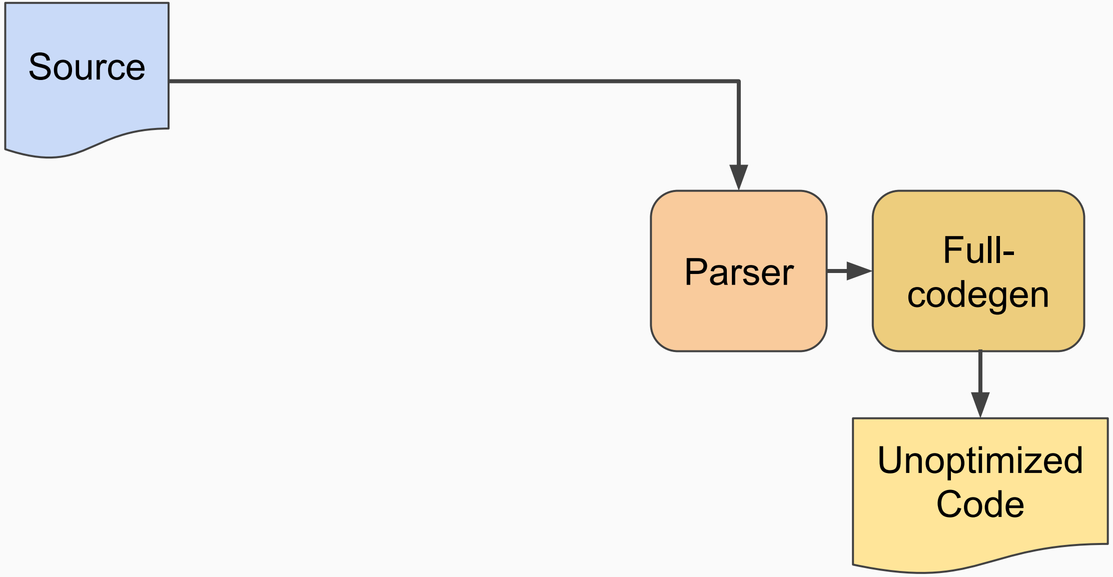
Parsed

```
function foo() { … }
function done() { … }
function unused() { … }

var Person = function() {
  this.name = name;
}
Person.prototype.doWork = function() {
  do { foo(); } while (!done());
}

var john = new Person(“John”);
john.doWork();
```

Compiled

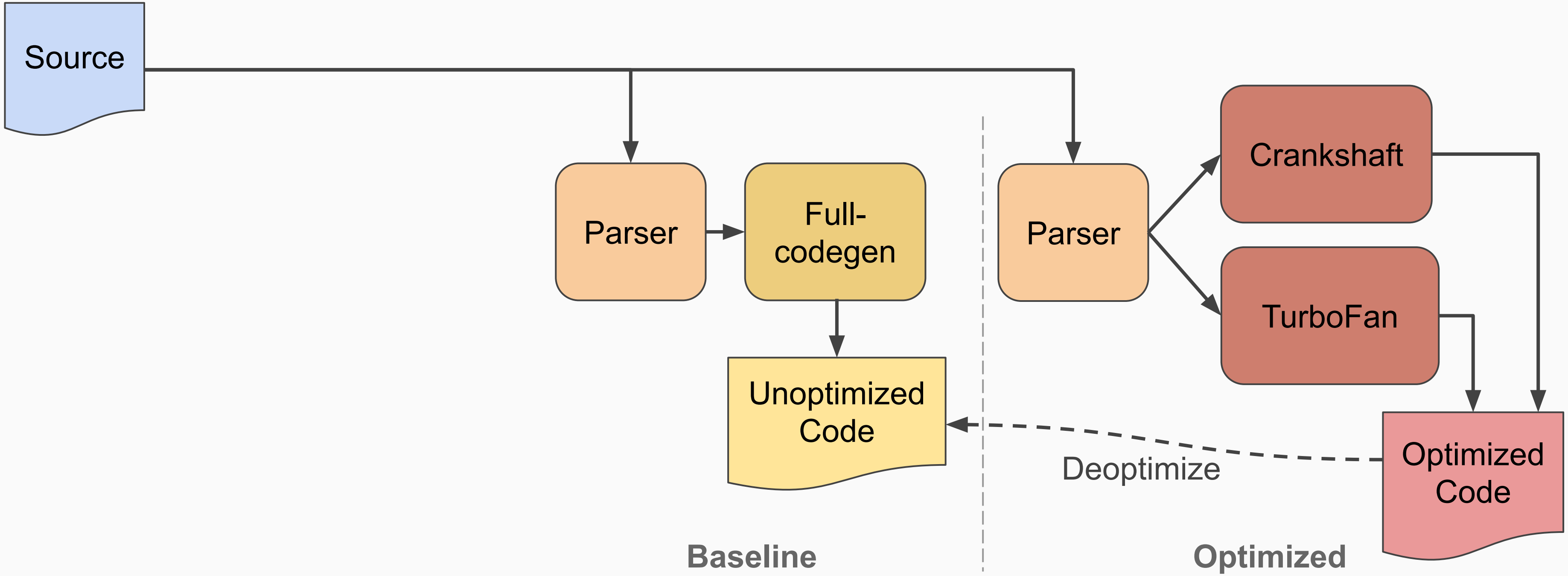


Compiler Pipeline



Baseline

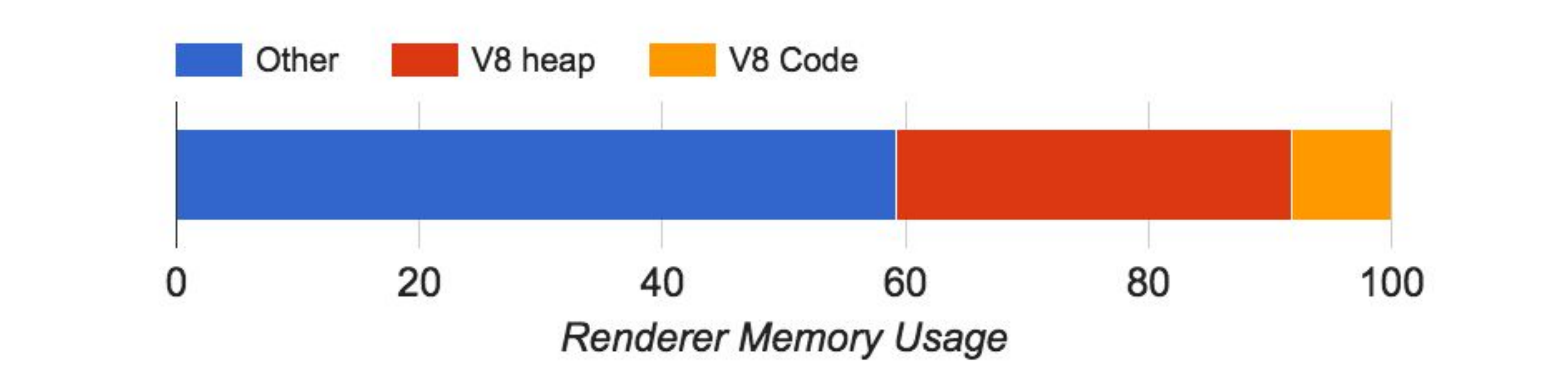
Compiler Pipeline



What’s the problem?

Memory

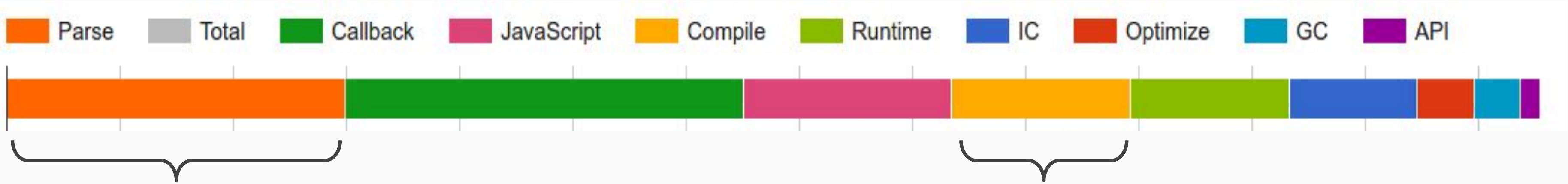
About 30% of the V8 heap is JITed unoptimized code



What’s the problem?

Startup Speed

Most functions are parsed multiple times

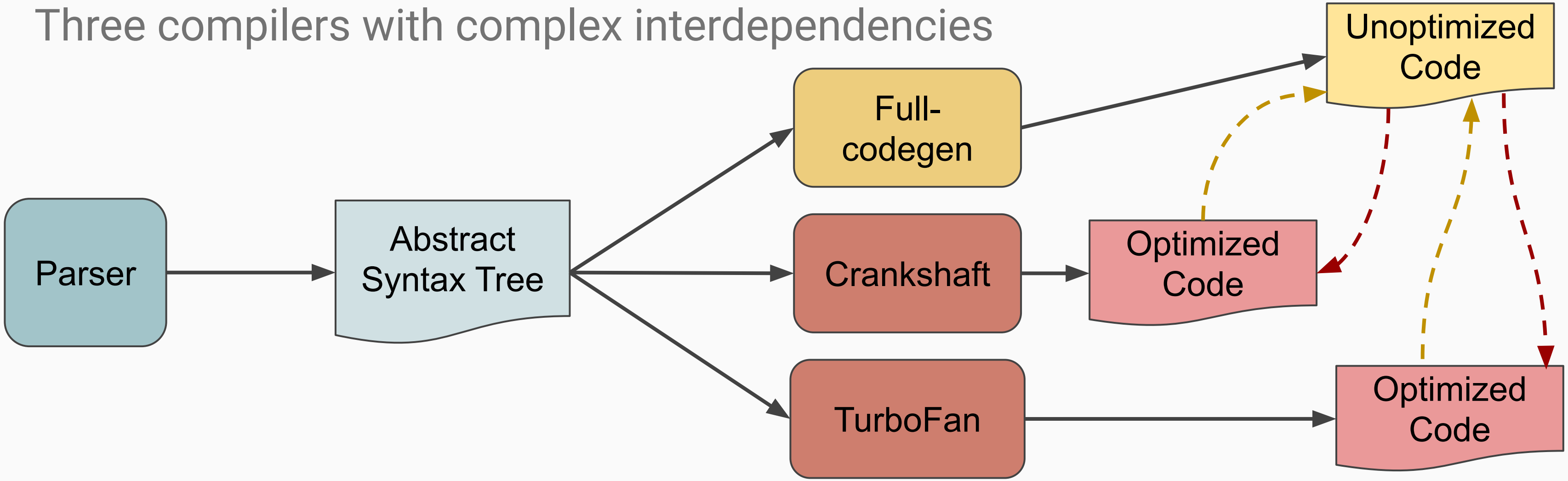


33% of time spent parsing + compiling

What’s the problem?

Complexity

Three compilers with complex interdependencies



Ignition

JavaScript Bytecode Interpreter for V8

Why Interpret?

Why Interpret?

- Reduced memory usage
    - *Compiled* to a concise bytecode, rather than machine code

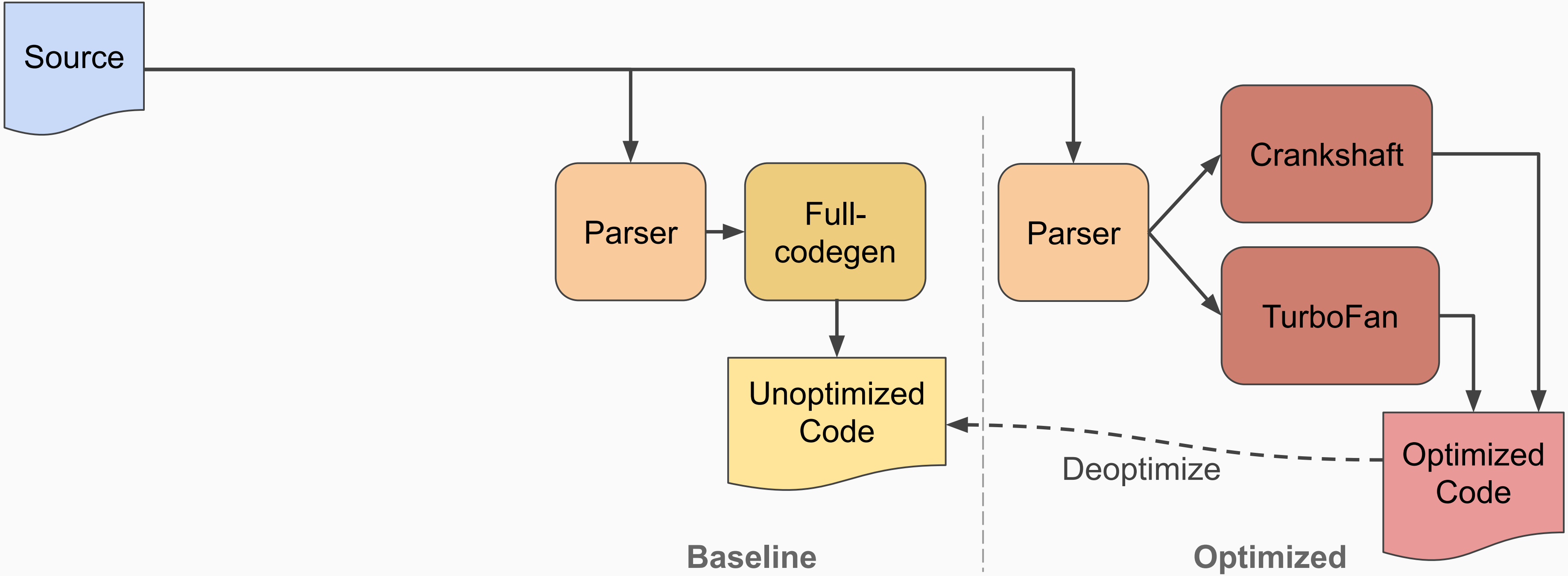
Why Interpret?

- Reduced memory usage
  - *Compiled* to a concise bytecode, rather than machine code
- Reduced parsing overhead
  - Bytecode is concise, allowing eager compilation of JS source

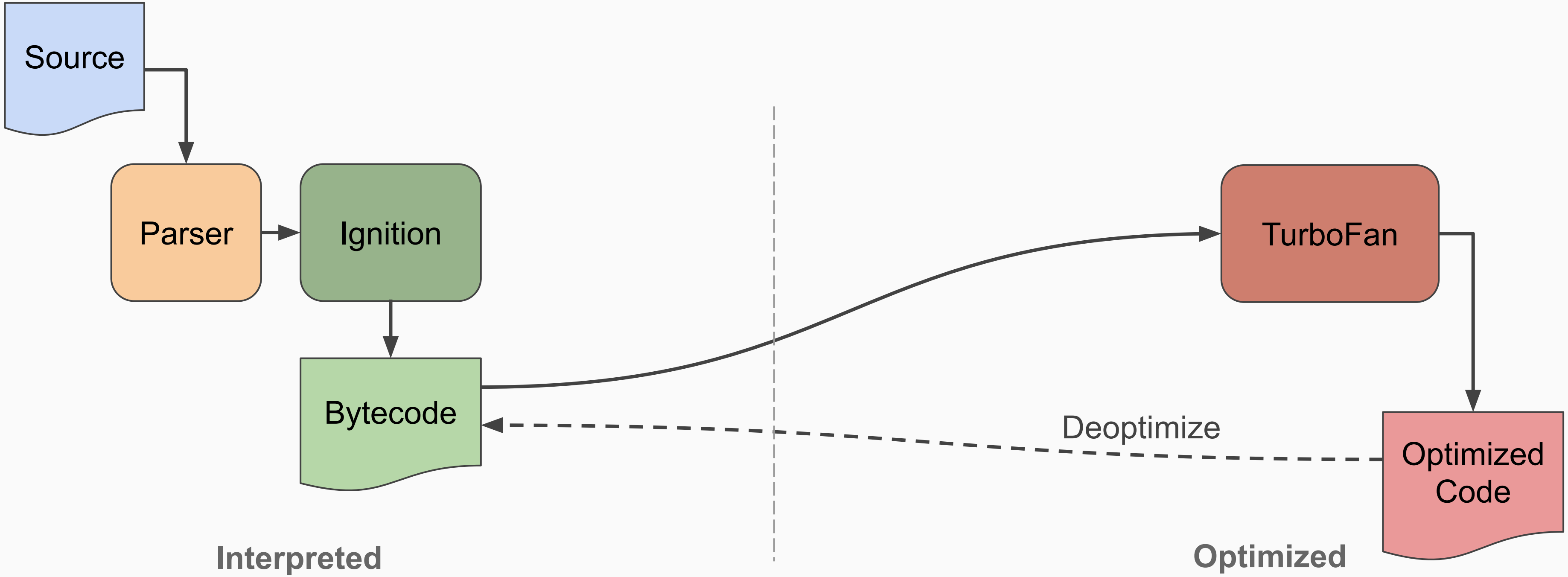
Why Interpret?

- Reduced memory usage
  - *Compiled* to a concise bytecode, rather than machine code
- Reduced parsing overhead
  - Bytecode is concise, allowing eager compilation of JS source
- Reduced compiler pipeline complexity
  - Bytecode is source-of-truth for optimizing / deoptimizing

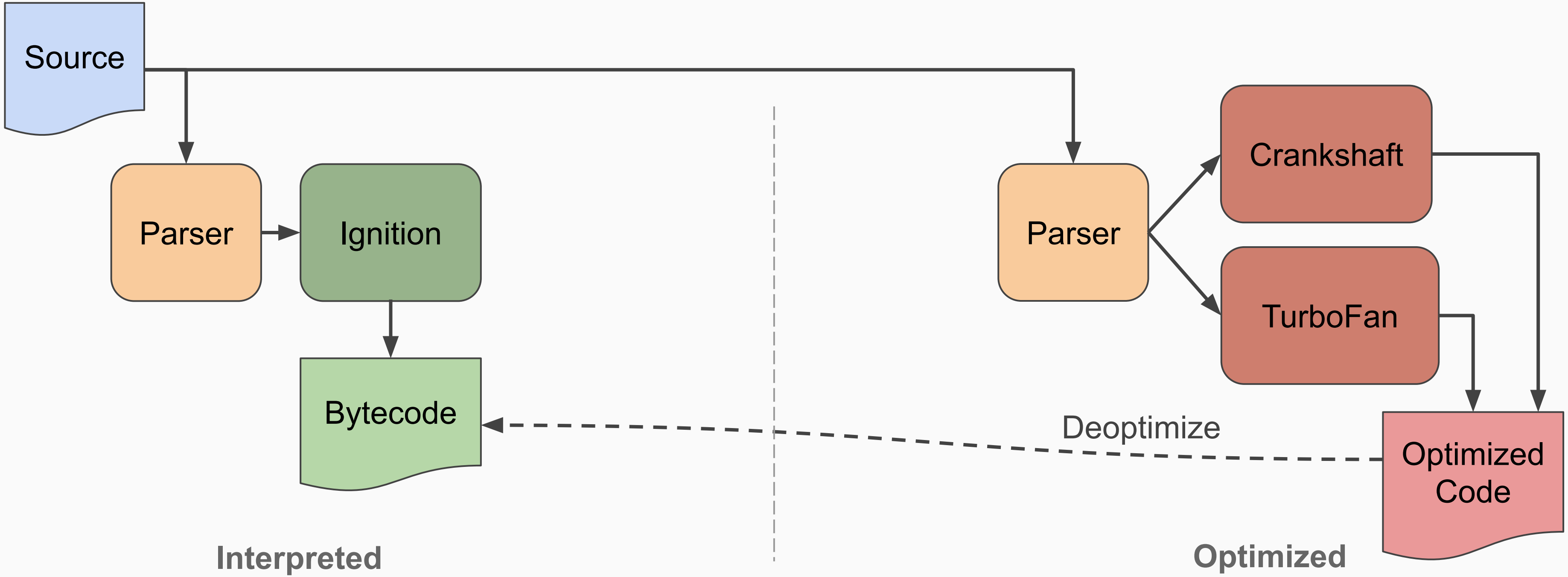
Compiler Pipeline



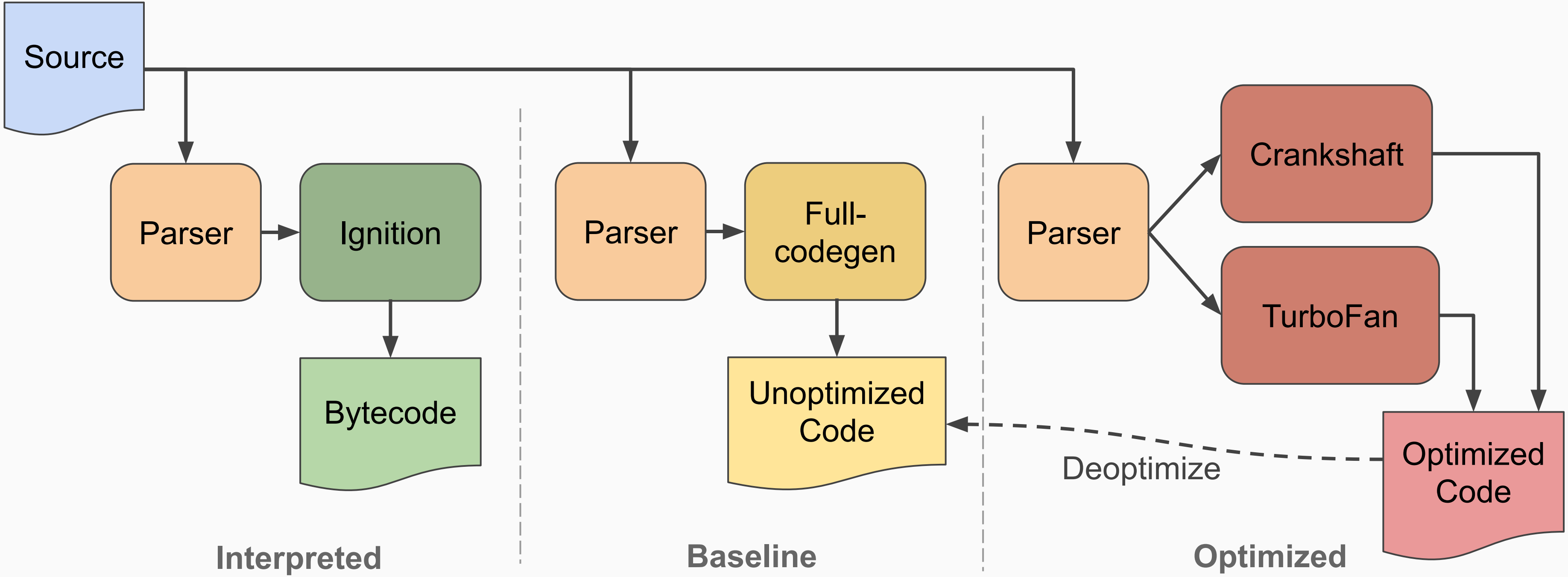
Compiler Pipeline



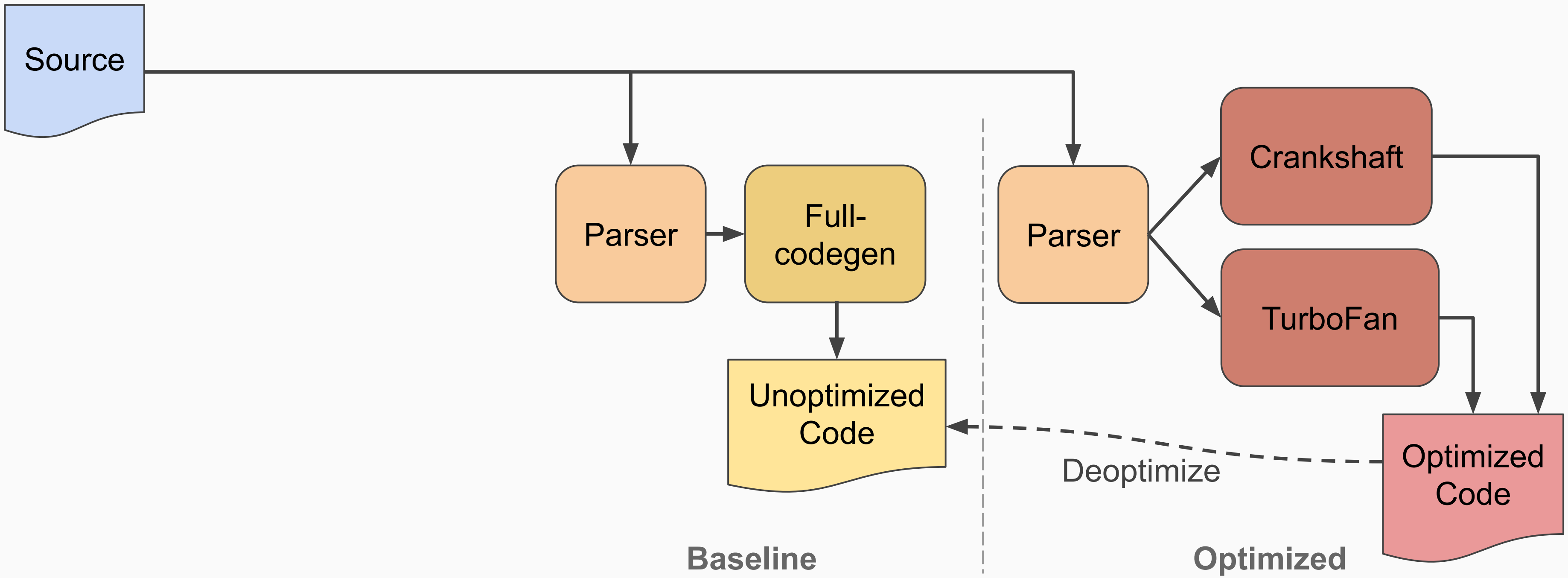
Compiler Pipeline



Compiler Pipeline



Compiler Pipeline



Deep Dive

How to build an Interpreter

Ignition Bytecode

```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

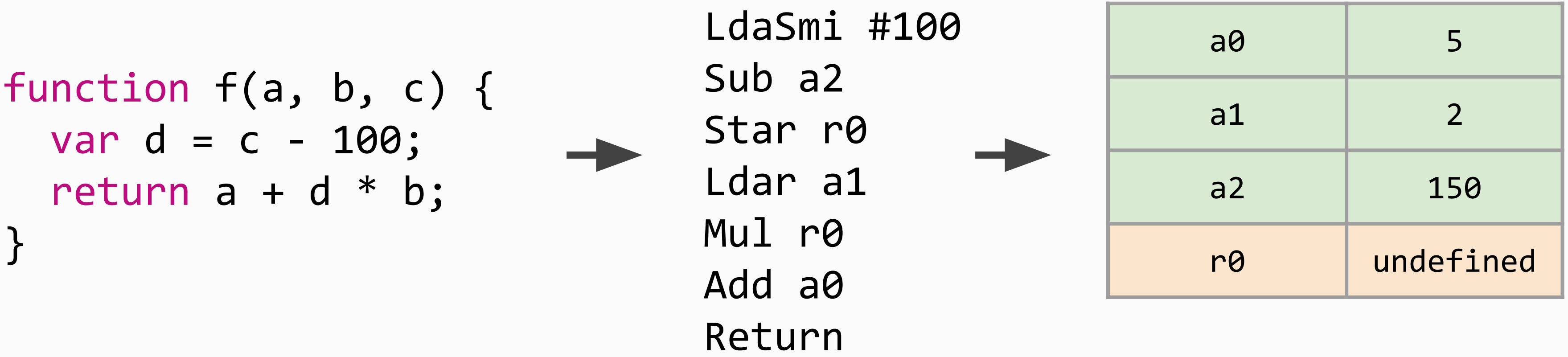
Ignition Bytecode

```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

r0	undefined

Ignition Bytecode



```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

a0	5
a1	2
a2	150
r0	undefined

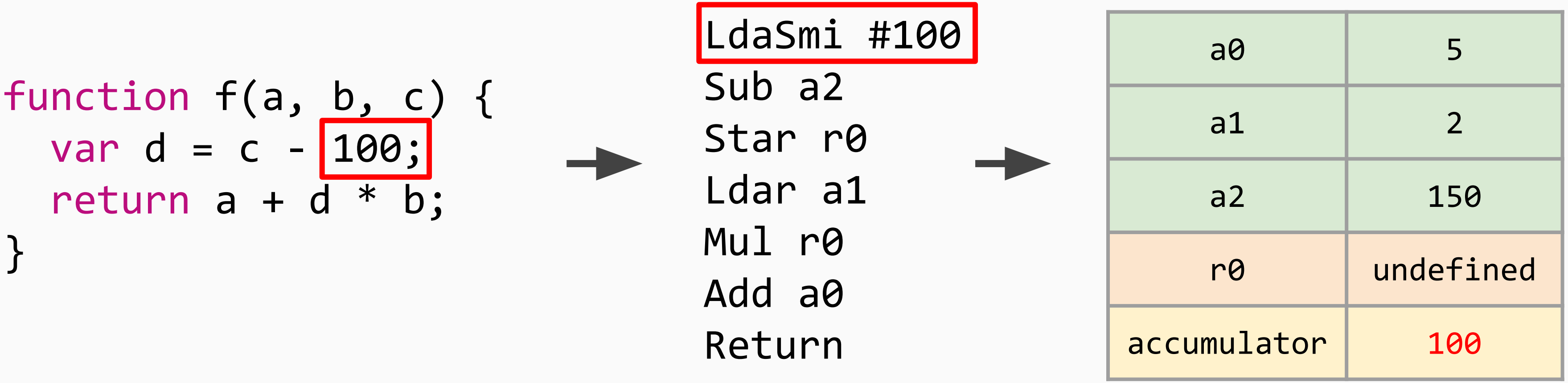
Ignition Bytecode

```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

a0	5
a1	2
a2	150
r0	undefined
accumulator	undefined

Ignition Bytecode



```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

a0	5
a1	2
a2	150
r0	undefined
accumulator	100

Ignition Bytecode

```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

a0	5
a1	2
a2	150
r0	undefined
accumulator	50

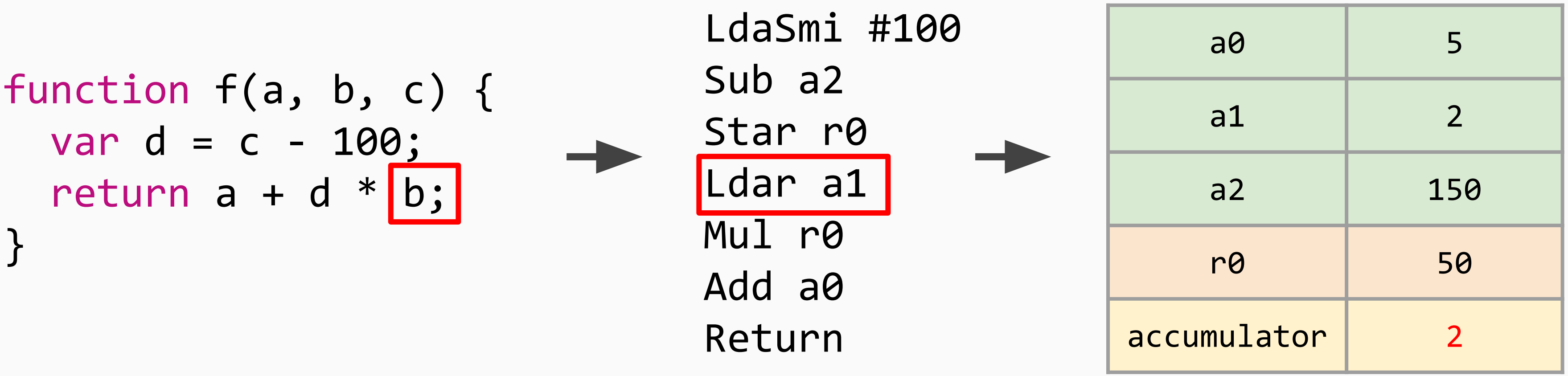
Ignition Bytecode

```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

a0	5
a1	2
a2	150
r0	50
accumulator	50

Ignition Bytecode

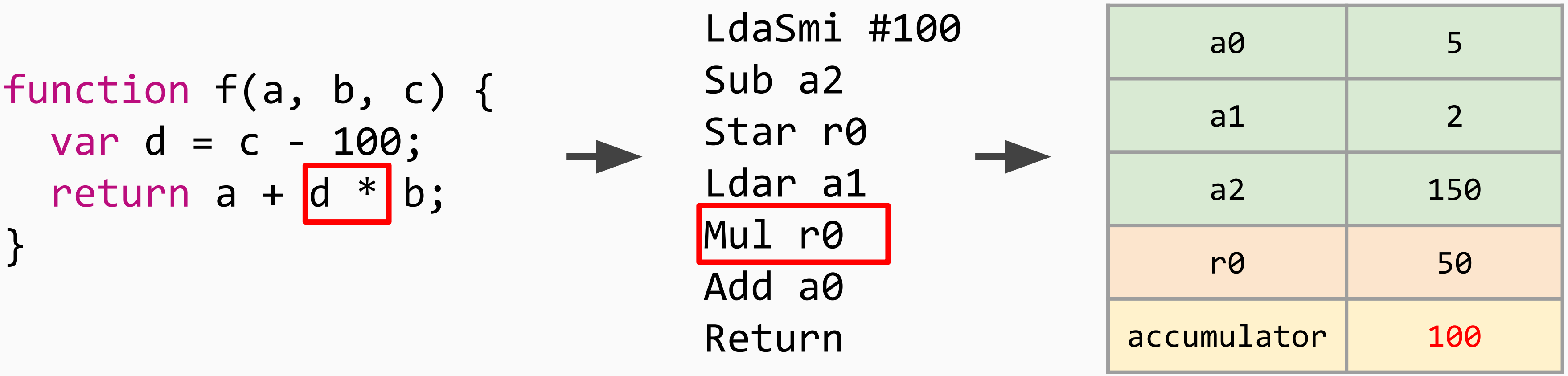


```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

a0	5
a1	2
a2	150
r0	50
accumulator	2

Ignition Bytecode

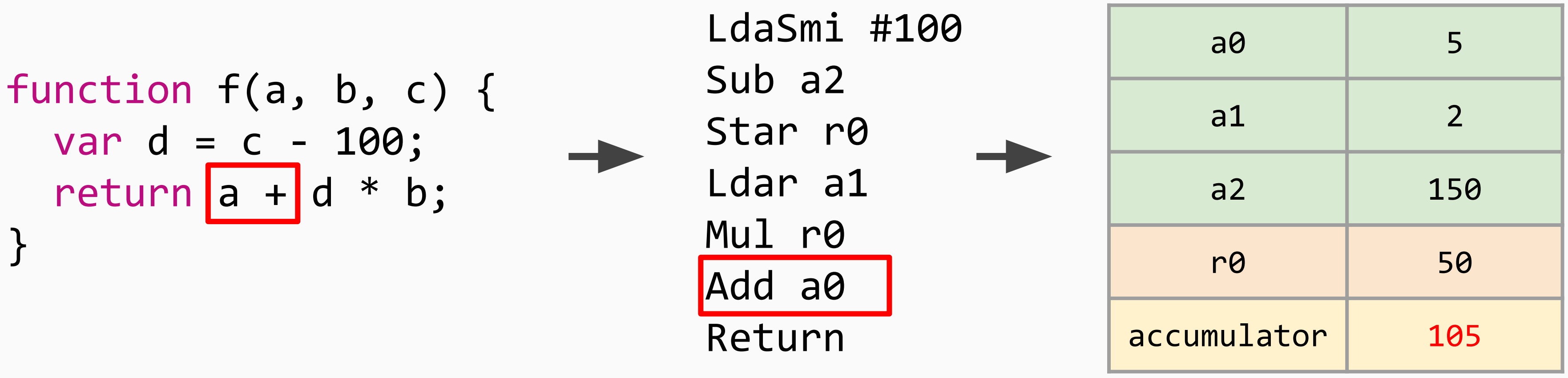


```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

a0	5
a1	2
a2	150
r0	50
accumulator	100

Ignition Bytecode



```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

a0	5
a1	2
a2	150
r0	50
accumulator	105

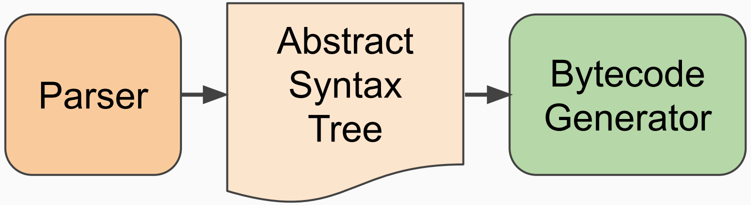
Ignition Bytecode

```
function f(a, b, c) {
  var d = c - 100;
  return a + d * b;
}
```

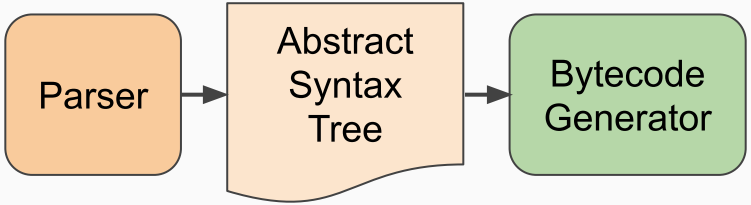
```
LdaSmi #100
Sub a2
Star r0
Ldar a1
Mul r0
Add a0
Return
```

a0	5
a1	2
a2	150
r0	50
accumulator	105

Ignition Bytecode Pipeline



Ignition Bytecode Pipeline



```
void BytecodeGenerator::VisitAddExpression(
    BinaryOperation* expr) {
  Register lhs =
      VisitForRegisterValue(expr->left());
  VisitForAccumulatorValue(expr->right());
  builder()->AddOperation(lhs);
  execution_result()->SetResultInAccumulator();
}
```

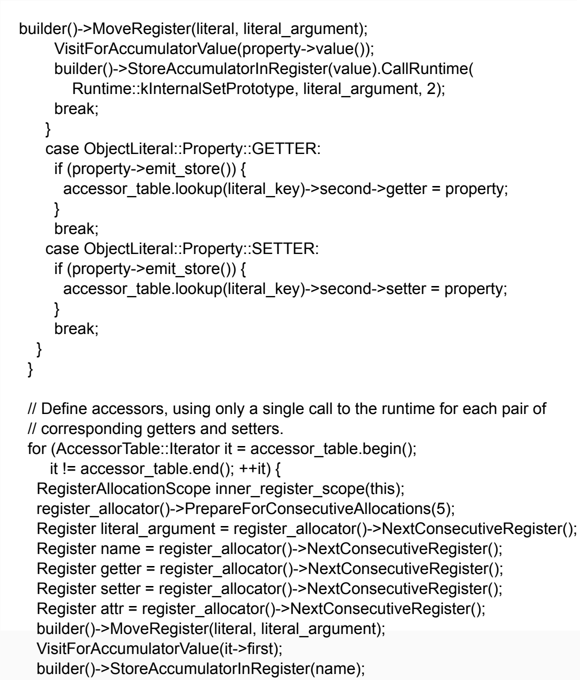
Ignition Bytecode Pipeline

```
void BytecodeGenerator::VisitObjectLiteral(ObjectLiteral* expr) {
  // Copy the literal boilerplate.
  int fast_clone_properties_count = 0;
  if (FastCloneShallowObjectStub::IsSupported(expr)) {
    STATIC_ASSERT(
        FastCloneShallowObjectStub::kMaximumClonedProperties <=
        1 << CreateObjectLiteralFlags::FastClonePropertiesCountBits::kShift);
    fast_clone_properties_count =
        FastCloneShallowObjectStub::PropertiesCount(expr->properties_count());
  }
  uint8_t flags =
      CreateObjectLiteralFlags::FlagsBits::encode(expr->ComputeFlags()) |
      CreateObjectLiteralFlags::FastClonePropertiesCountBits::encode(
          fast_clone_properties_count);
  builder()->CreateObjectLiteral(expr->constant_properties(),
                                  expr->literal_index(), flags);

  // Allocate in the outer scope since this register is used to return the
  // expression's results to the caller.
  Register literal = register_allocator()->outer()->NewRegister();
  builder()->StoreAccumulatorInRegister(literal);

  // Store computed values into the literal.
  int property_index = 0;
  AccessorTable accessor_table(zone());
  for (; property_index < expr->properties()->length(); property_index++) {
    ObjectLiteral::Property* property = expr->properties()->at(property_index);
    if (property->is_computed_name()) break;
    if (property->IsCompileTimeValue()) continue;
```

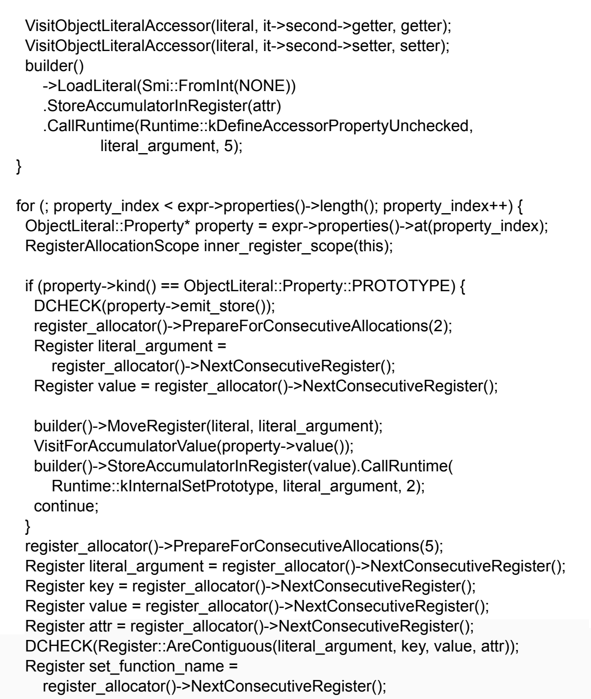
```
    RegisterAllocationScope inner_register_scope(this);
    Literal* literal_key = property->key()->AsLiteral();
    switch (property->kind()) {
      case ObjectLiteral::Property::CONSTANT:
        UNREACHABLE();
      case ObjectLiteral::Property::MATERIALIZED_LITERAL:
        DCHECK(!CompileTimeValue::IsCompileTimeValue(property->value()));
      // Fall through.
      case ObjectLiteral::Property::COMPUTED: {
        // It is safe to use [[Put]] here because the boilerplate already
        // contains computed properties with an uninitialized value.
        if (literal_key->value()->IsInternalizedString()) {
          if (property->emit_store()) {
            VisitForAccumulatorValue(property->value());
            if (FunctionLiteral::NeedsHomeObject(property->value())) {
              RegisterAllocationScope register_scope(this);
              Register value = register_allocator()->NewRegister();
              builder()->StoreAccumulatorInRegister(value);
              builder()->StoreNamedProperty(
                  literal, literal_key->AsPropertyName(),
                  feedback_index(property->GetSlot(0)), language_mode());
              VisitSetHomeObject(value, literal, property, 1);
            } else {
              builder()->StoreNamedProperty(
                  literal, literal_key->AsPropertyName(),
                  feedback_index(property->GetSlot(0)), language_mode());
            }
          } else {
            VisitForEffect(property->value());
          }
```

```
          register_allocator()->PrepareForConsecutiveAllocations(4);
          Register literal_argument =
              register_allocator()->NextConsecutiveRegister();
          Register key = register_allocator()->NextConsecutiveRegister();
          Register value = register_allocator()->NextConsecutiveRegister();
          Register language = register_allocator()->NextConsecutiveRegister();

          builder()->MoveRegister(literal, literal_argument);
          VisitForAccumulatorValue(property->key());
          builder()->StoreAccumulatorInRegister(key);
          VisitForAccumulatorValue(property->value());
          builder()->StoreAccumulatorInRegister(value);
          if (property->emit_store()) {
            builder()
                ->LoadLiteral(Smi::FromInt(SLOPPY))
                .StoreAccumulatorInRegister(language)
                .CallRuntime(Runtime::kSetProperty, literal_argument, 4);
            VisitSetHomeObject(value, literal, property);
          }
        }
        break;
      }
      case ObjectLiteral::Property::PROTOTYPE: {
        DCHECK(property->emit_store());
        register_allocator()->PrepareForConsecutiveAllocations(2);
        Register literal_argument =
            register_allocator()->NextConsecutiveRegister();
        Register value = register_allocator()->NextConsecutiveRegister();
```

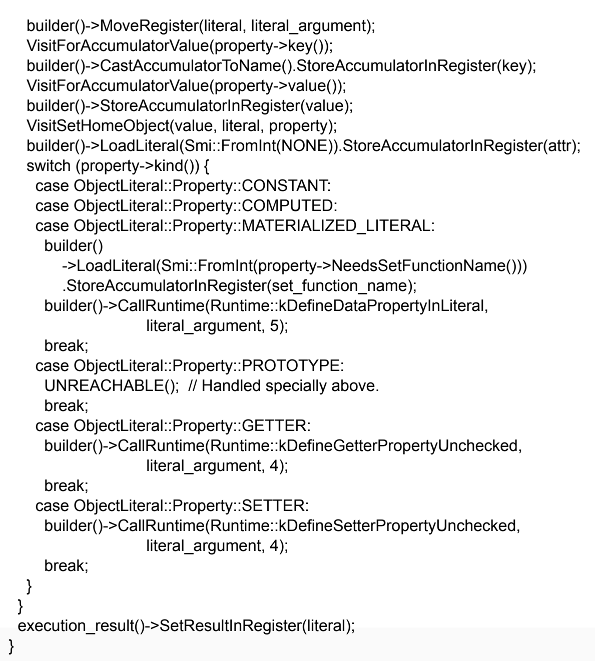
Ignition Bytecode Pipeline



```
 builder()->MoveRegister(literal, literal_argument);
     VisitForAccumulatorValue(property->value());
     builder()->StoreAccumulatorInRegister(value).CallRuntime(
       Runtime::kInternalSetPrototype, literal_argument, 2);
     break;
    }
    case ObjectLiteral::Property::GETTER:
     if (property->emit_store()) {
      accessor_table.lookup(literal_key)->second->getter = property;
     }
     break;
    case ObjectLiteral::Property::SETTER:
     if (property->emit_store()) {
      accessor_table.lookup(literal_key)->second->setter = property;
     }
     break;
   }
  }

  // Define accessors, using only a single call to the runtime for each pair of
  // corresponding getters and setters.
  for (AccessorTable::Iterator it = accessor_table.begin();
      it != accessor_table.end(); ++it) {
   RegisterAllocationScope inner_register_scope(this);
   register_allocator()->PrepareForConsecutiveAllocations(5);
   Register literal_argument = register_allocator()->NextConsecutiveRegister();
   Register name = register_allocator()->NextConsecutiveRegister();
   Register getter = register_allocator()->NextConsecutiveRegister();
   Register setter = register_allocator()->NextConsecutiveRegister();
   Register attr = register_allocator()->NextConsecutiveRegister();
   builder()->MoveRegister(literal, literal_argument);
   VisitForAccumulatorValue(it->first);
   builder()->StoreAccumulatorInRegister(name);
```

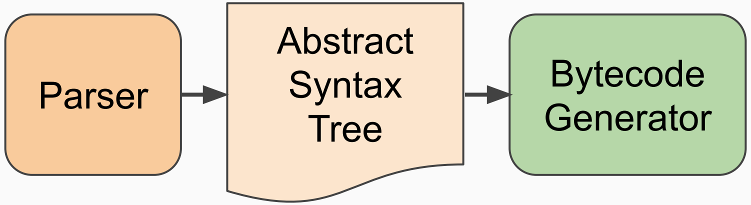


```
  VisitObjectLiteralAccessor(literal, it->second->getter, getter);
  VisitObjectLiteralAccessor(literal, it->second->setter, setter);
  builder()
    ->LoadLiteral(Smi::FromInt(NONE))
    .StoreAccumulatorInRegister(attr)
    .CallRuntime(Runtime::kDefineAccessorPropertyUnchecked,
          literal_argument, 5);
 }

 for (; property_index < expr->properties()->length(); property_index++) {
  ObjectLiteral::Property* property = expr->properties()->at(property_index);
  RegisterAllocationScope inner_register_scope(this);

  if (property->kind() == ObjectLiteral::Property::PROTOTYPE) {
   DCHECK(property->emit_store());
   register_allocator()->PrepareForConsecutiveAllocations(2);
   Register literal_argument =
     register_allocator()->NextConsecutiveRegister();
   Register value = register_allocator()->NextConsecutiveRegister();

   builder()->MoveRegister(literal, literal_argument);
   VisitForAccumulatorValue(property->value());
   builder()->StoreAccumulatorInRegister(value).CallRuntime(
     Runtime::kInternalSetPrototype, literal_argument, 2);
   continue;
  }
  register_allocator()->PrepareForConsecutiveAllocations(5);
  Register literal_argument = register_allocator()->NextConsecutiveRegister();
  Register key = register_allocator()->NextConsecutiveRegister();
  Register value = register_allocator()->NextConsecutiveRegister();
  Register attr = register_allocator()->NextConsecutiveRegister();
  DCHECK(Register::AreContiguous(literal_argument, key, value, attr));
  Register set_function_name =
    register_allocator()->NextConsecutiveRegister();
```

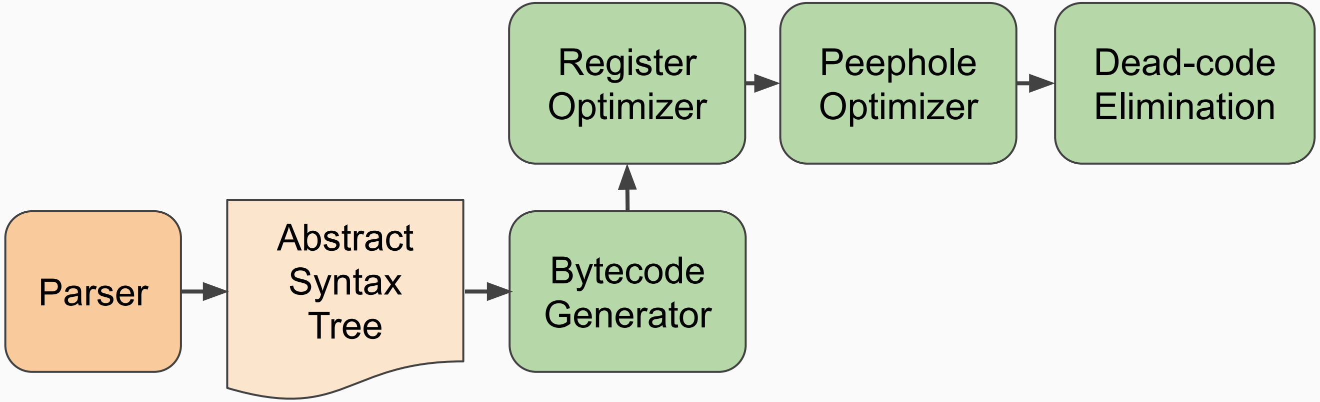


```
   builder()->MoveRegister(literal, literal_argument);
   VisitForAccumulatorValue(property->key());
   builder()->CastAccumulatorToName().StoreAccumulatorInRegister(key);
   VisitForAccumulatorValue(property->value());
   builder()->StoreAccumulatorInRegister(value);
   VisitSetHomeObject(value, literal, property);
   builder()->LoadLiteral(Smi::FromInt(NONE)).StoreAccumulatorInRegister(attr);
   switch (property->kind()) {
    case ObjectLiteral::Property::CONSTANT:
    case ObjectLiteral::Property::COMPUTED:
    case ObjectLiteral::Property::MATERIALIZED_LITERAL:
     builder()
       ->LoadLiteral(Smi::FromInt(property->NeedsSetFunctionName()))
       .StoreAccumulatorInRegister(set_function_name);
     builder()->CallRuntime(Runtime::kDefineDataPropertyInLiteral,
                literal_argument, 5);
     break;
    case ObjectLiteral::Property::PROTOTYPE:
     UNREACHABLE();  // Handled specially above.
     break;
    case ObjectLiteral::Property::GETTER:
     builder()->CallRuntime(Runtime::kDefineGetterPropertyUnchecked,
                literal_argument, 4);
     break;
    case ObjectLiteral::Property::SETTER:
     builder()->CallRuntime(Runtime::kDefineSetterPropertyUnchecked,
                literal_argument, 4);
     break;
   }
  }
  execution_result()->SetResultInRegister(literal);
 }
```

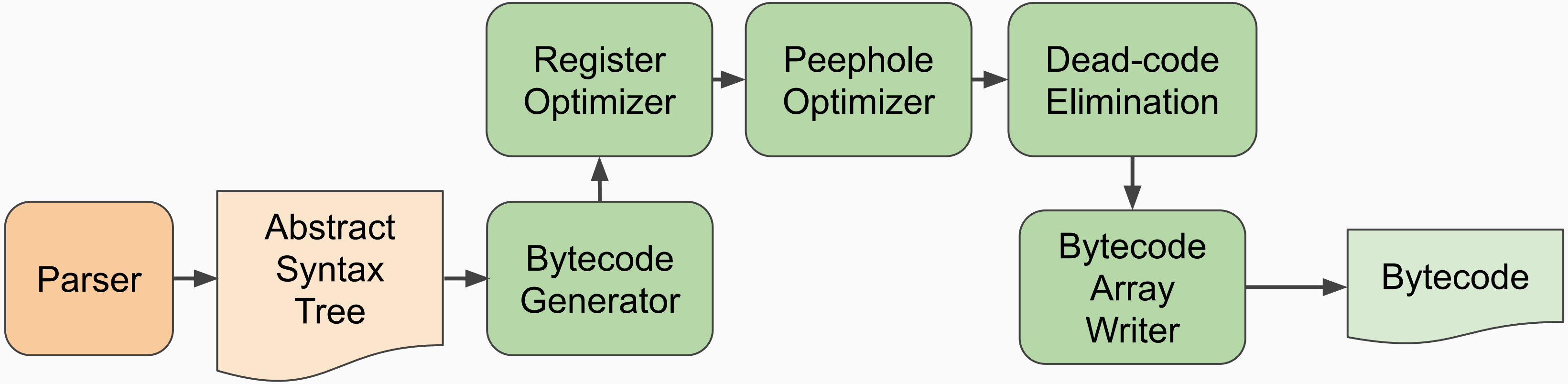
Ignition Bytecode Pipeline



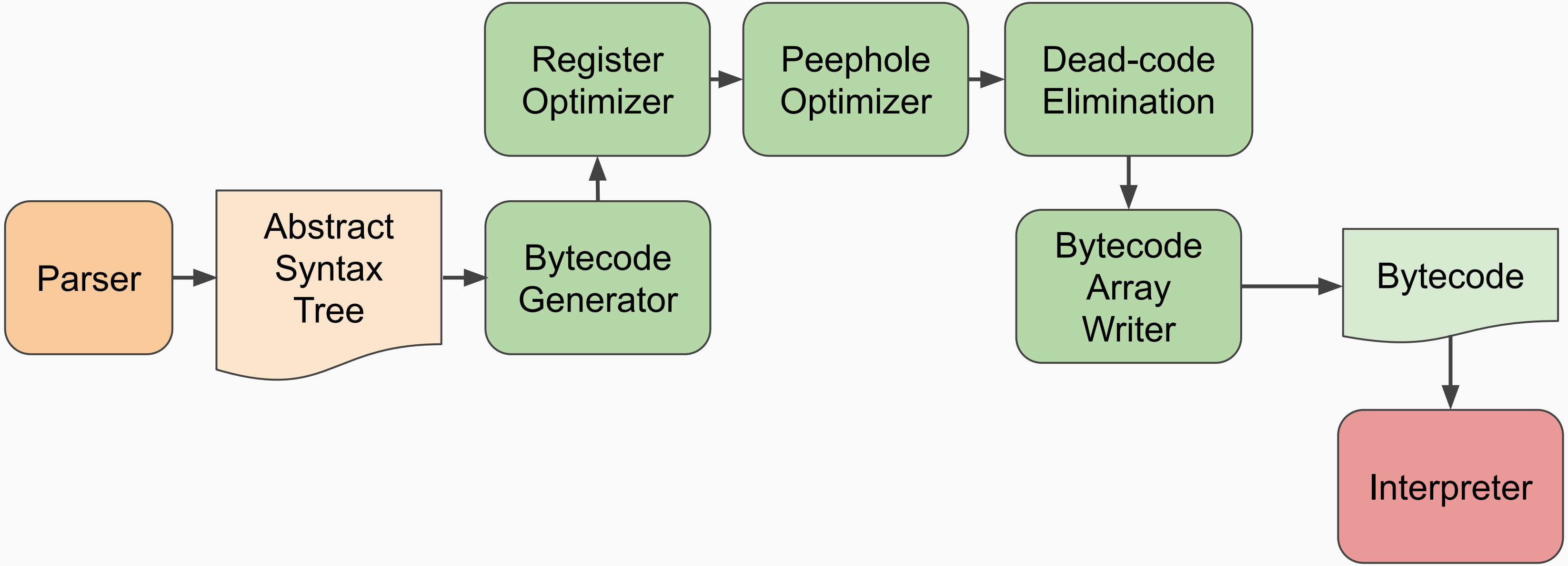
Ignition Bytecode Pipeline



Ignition Bytecode Pipeline



Ignition Bytecode Pipeline



Building the Ignition Interpreter

- Write in C++

Building the Ignition Interpreter

❌ Write in C++

- ○ Need trampolines between Interpreted and JITed functions
- ○ Can’t interoperate with fast code-stubs

Building the Ignition Interpreter

❌ Write in C++

  ○ Need trampolines between Interpreted and JITed functions

  ○ Can’t interoperate with fast code-stubs

– Hand-crafted assembly code

Building the Ignition Interpreter

- ❌ Write in C++
  - ○ Need trampolines between Interpreted and JITed functions
  - ○ Can’t interoperate with fast code-stubs
- ❌ Hand-crafted assembly code
  - ○ Would need to be ported to 9 architectures

Building the Ignition Interpreter

❌ Write in C++

- ○ Need trampolines between Interpreted and JITed functions
- ○ Can’t interoperate with fast code-stubs

❌ Hand-crafted assembly code

- ○ Would need to be ported to 9 architectures

– Backend of the TurboFan Compiler

Building the Ignition Interpreter

- ❌ Write in C++
  - ○ Need trampolines between Interpreted and JITed functions
  - ○ Can’t interoperate with fast code-stubs
- ❌ Hand-crafted assembly code
  - ○ Would need to be ported to 9 architectures
- ✔ Backend of the TurboFan Compiler
  - ○ Write-once in macro-assembly
  - ○ Architecture specific instruction selection optimizations for free
  - ○ Relatively painless interoperability with existing code-stubs

Building the Ignition Interpreter

```
void Interpreter::DoAdd(InterpreterAssembler* assembler) {
  Node* reg_index = assembler->BytecodeOperandReg(0);
  Node* lhs = assembler->LoadRegister(reg_index);
  Node* rhs = assembler->GetAccumulator();
  Node* result = AddStub::Generate(assembler, lhs, rhs);
  assembler->SetAccumulator(result);
  assembler->Dispatch();
}
```

Building the Ignition Interpreter

```
void Interpreter::DoAdd(InterpreterAssembler* assembler) {
  Node* reg_index = assembler->BytecodeOperandReg(0);
  Node* lhs = assembler->LoadRegister(reg_index);
  Node* rhs = assembler->GetAccumulator();
  Node* result = AddStub::Generate(assembler, lhs, rhs);
  assembler->SetAccumulator(result);
  assembler->Dispatch();
}
```

~375 LOC for number addition

Building the Ignition Interpreter

```
void Interpreter::DoAdd(InterpreterAssembler* assembler) {
  Node* reg_index = assembler->BytecodeOperandReg(0);
  Node* lhs = assembler->LoadRegister(reg_index);
  Node* rhs = assembler->GetAccumulator();
  Node* result = AddStub::Generate(assembler, lhs, rhs);
  assembler->SetAccumulator(result);
  assembler->Dispatch();
}
```

~375 LOC for number addition
~250 LOC for string addition

Building the Ignition Interpreter

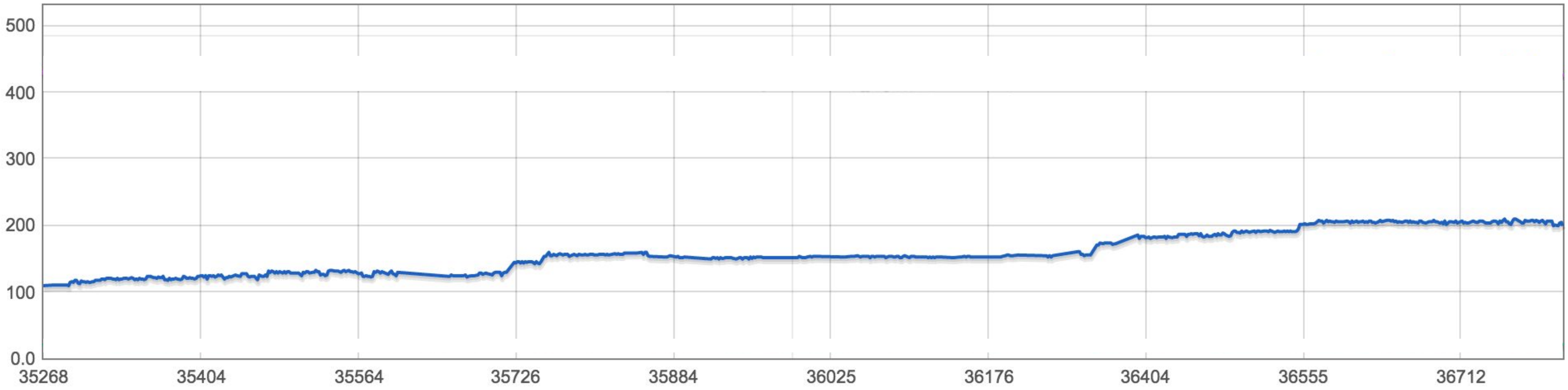
```
void Interpreter::DoAdd(InterpreterAssembler* assembler) {
  Node* reg_index = assembler->BytecodeOperandReg(0);
  Node* lhs = assembler->LoadRegister(reg_index);
  Node* rhs = assembler->GetAccumulator();
  Node* result = AddStub::Generate(assembler, lhs, rhs);
  assembler->SetAccumulator(result);
  assembler->Dispatch();
}
```

~375 LOC for number addition
~250 LOC for string addition
… for type conversions

Ignition

Results

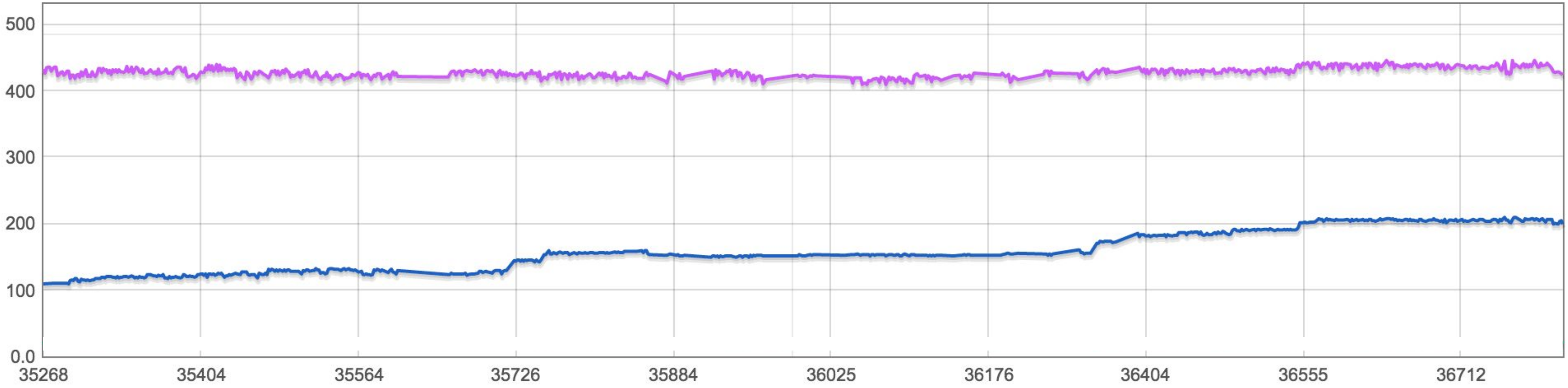
Ignition Performance (Octane)



Octane Score (Nexus 5)

Crankshaft and TurboFan disabled

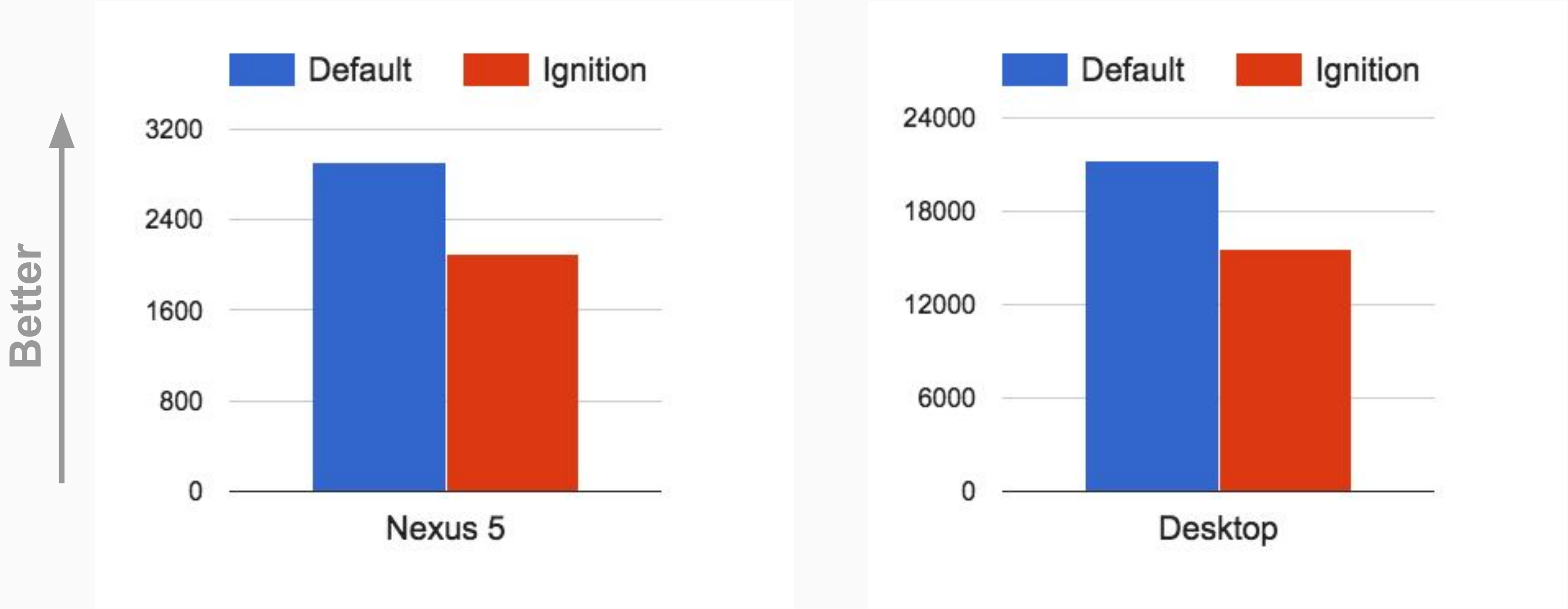
Ignition Performance (Octane)



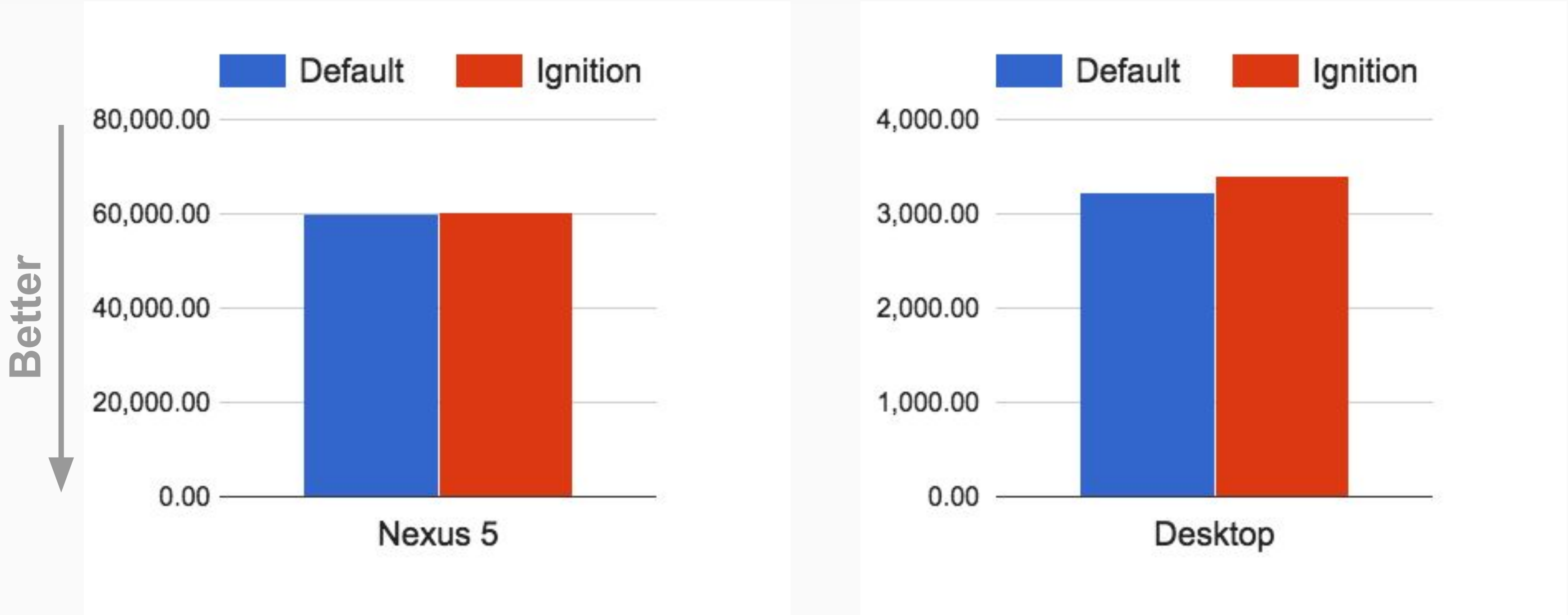
Octane Score (Nexus 5)

Crankshaft and TurboFan disabled

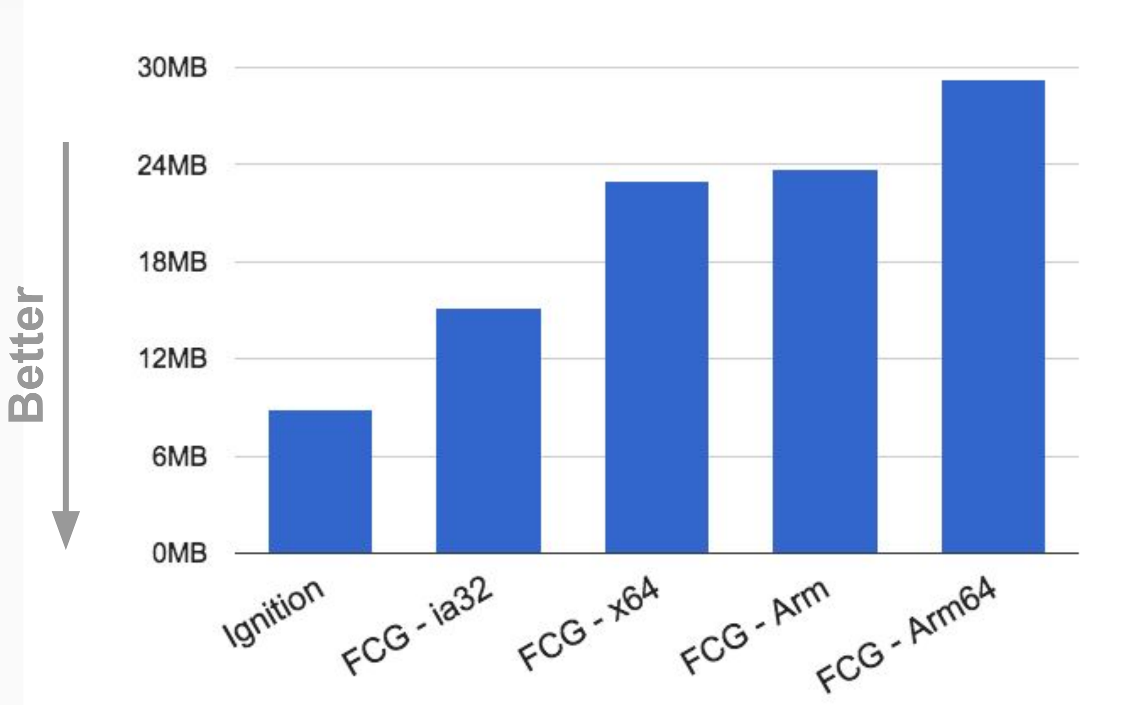
Ignition Performance (Octane)



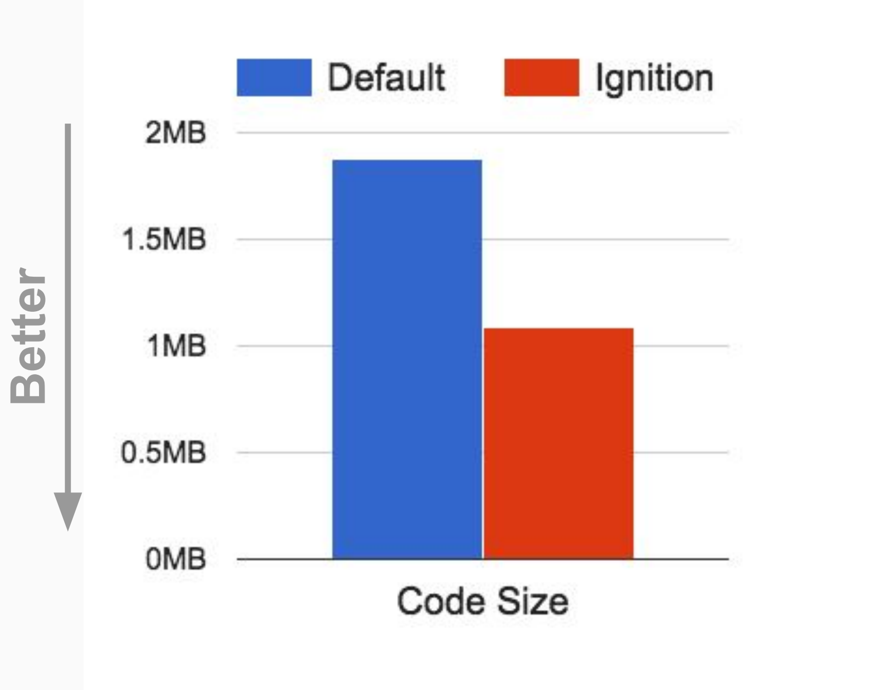
Ignition Performance (Speedometer)

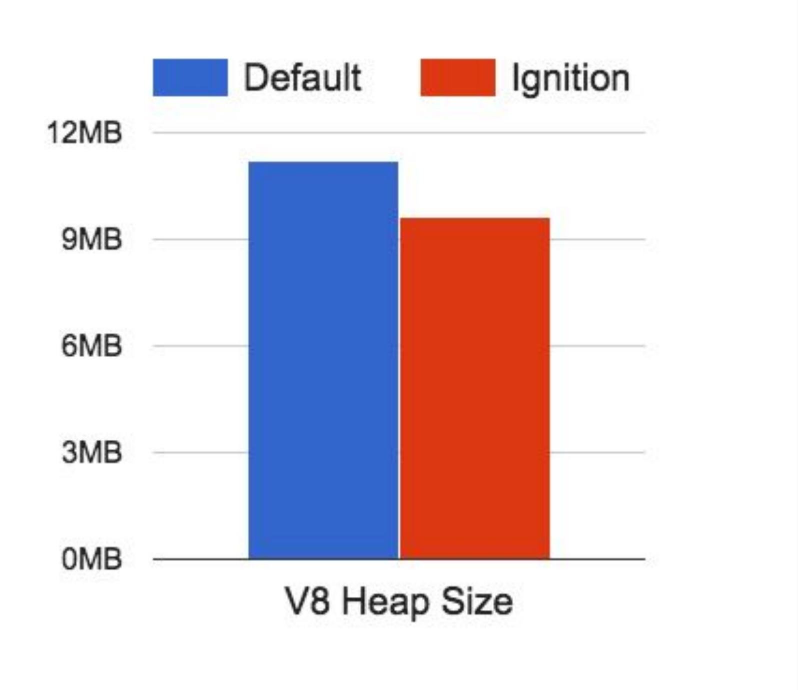


Ignition Memory Usage (Octane)



Ignition Memory Usage (Mobile Top 10)





Status and Future Plans

Shipping Plan

Shipping plans

- **m53:** Launch for Low-End (Svelte) devices

Shipping plans

- **m53:** Launch for Low-End (Svelte) devices

↓ **16%** **smaller V8 heap**

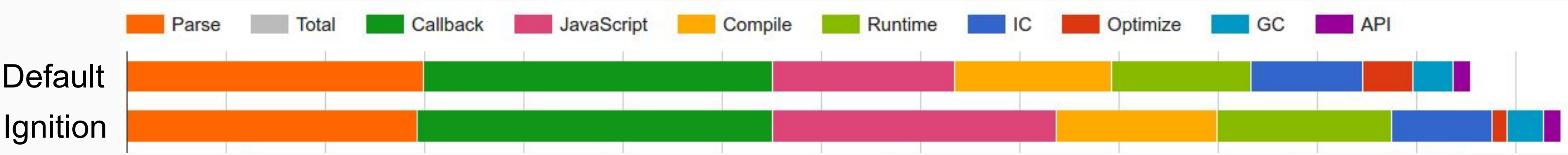
↓ **8.5%** **less renderer memory use**

↑ **14%** **more time in V8**

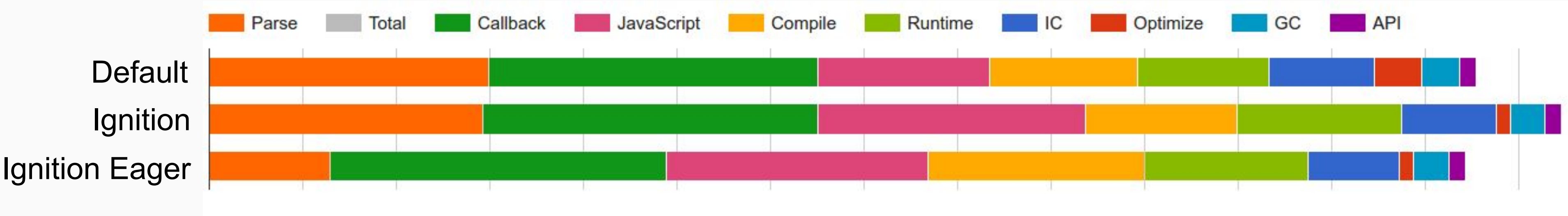
Shipping plans

- **m53:** Launch for Low-End (Svelte) devices
- **End of Q3:** Launch on all platforms focusing on startup

Eager Compilation



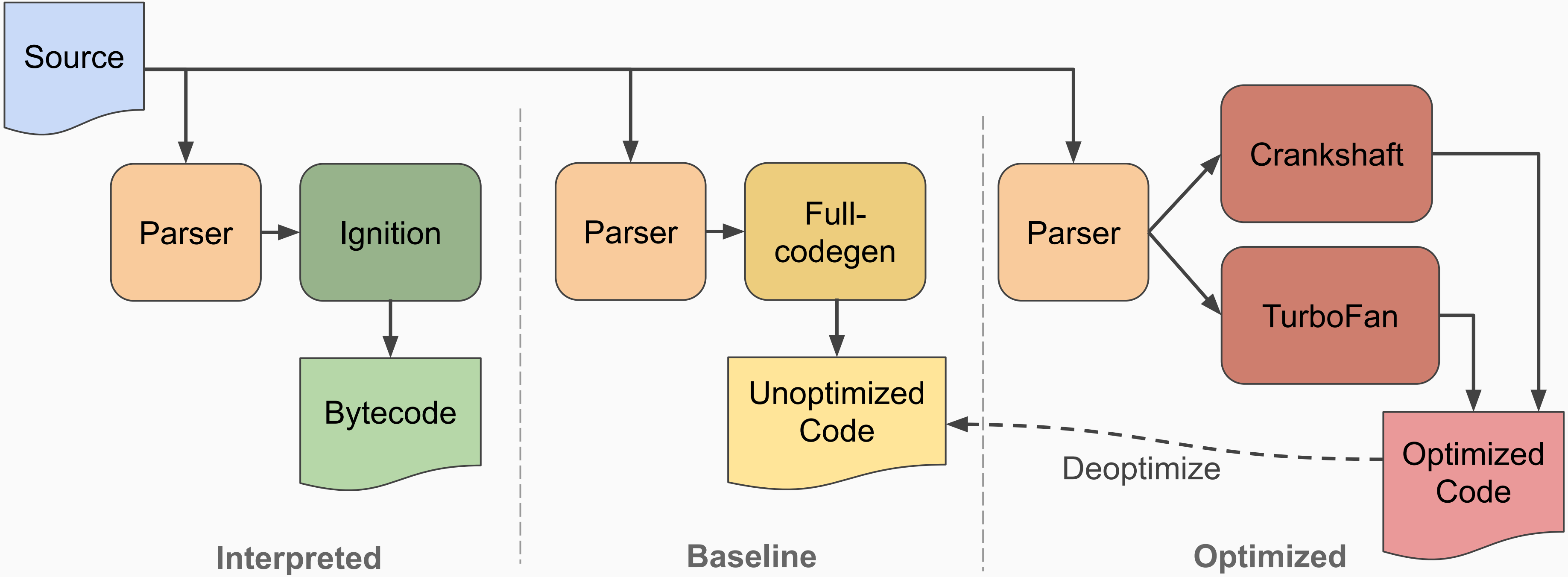
Eager Compilation



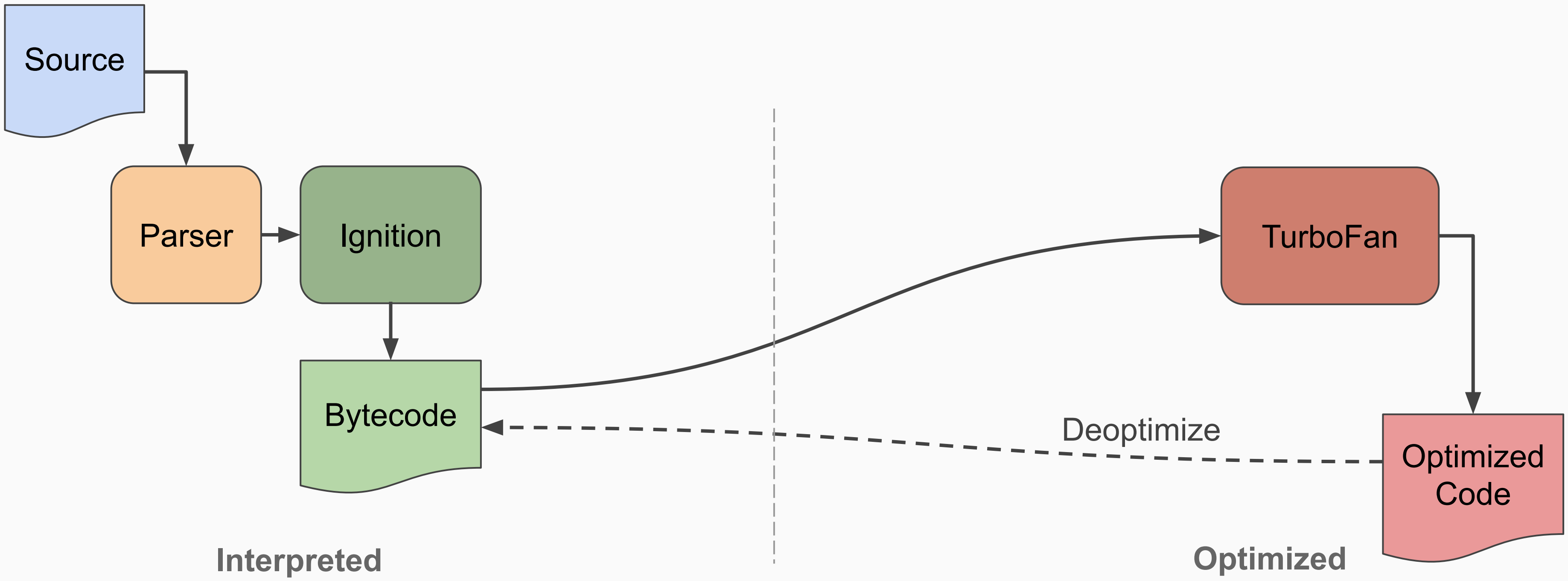
Shipping plans

- **m53:** Launch for Low-End (Svelte) devices
- **End of Q3:** Launch on all platforms focusing on startup
- **2017:** Deprecate Crankshaft / Full-codegen

Compiler Pipeline



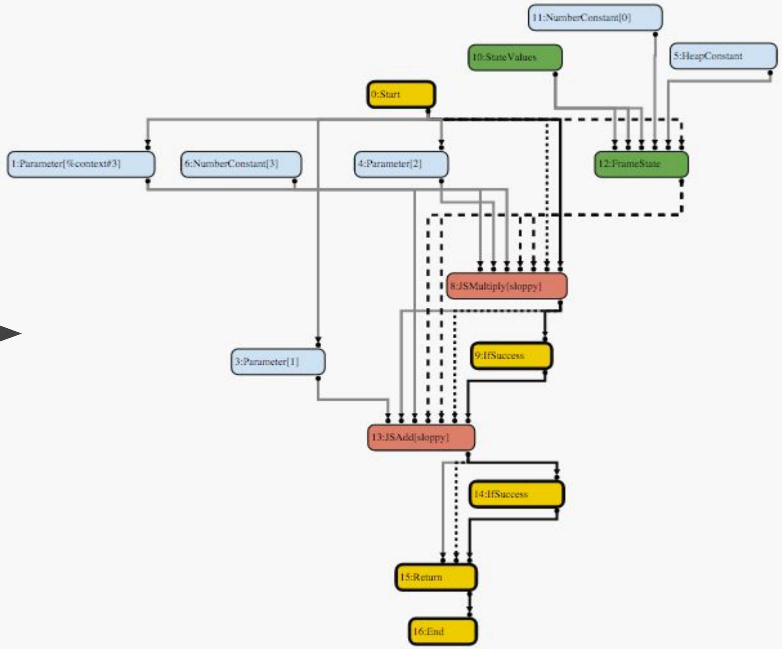
Compiler Pipeline



TurboFan Bytecode Graph Builder

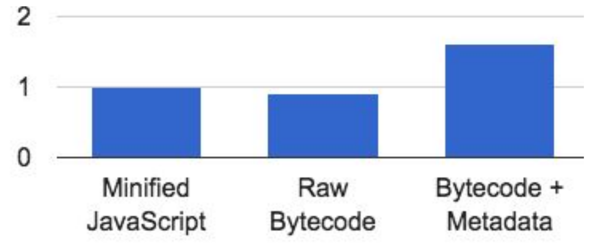
```
function f(x, y) {
  return x + 3 * y;
}
```

```
LdaSmi8 #3
Star r0
Ldar a1
Mul r0
Add a0
Return
```



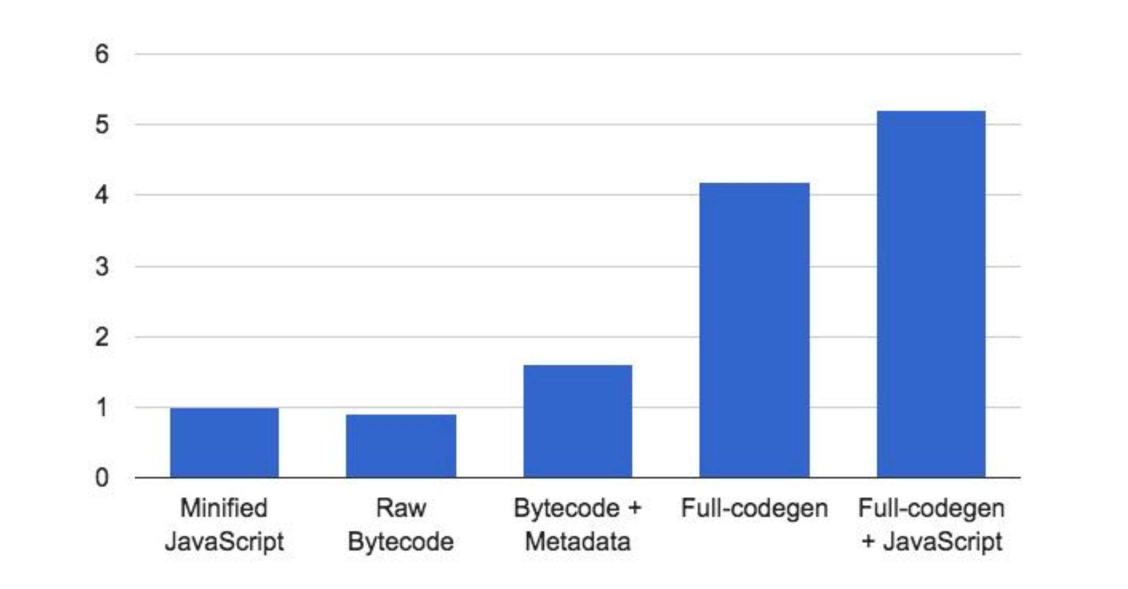
Bytecode as the source of truth

JavaScript source no longer needed after bytecode compilation



Bytecode as the source of truth

JavaScript source no longer needed after bytecode compilation



Summary

- **Ignition** - a fast JavaScript interpreter for V8
- Immediate **memory reductions** on low-end devices (m53)
- Promises to **improve startup** on high-end devices
- Basis for a **simpler** compiler pipeline with **new opportunities**

Questions?

Ignition Bytecodes

Loading the accumulator
LdaZero
LdaSmi8
LdaUndefined
LdrUndefined
LdaNull
LdaTheHole
LdaTrue
LdaFalse
LdaConstant

Binary Operators
Add
Sub
Mul
Div
Mod
BitwiseOr
BitwiseXor
BitwiseAnd
ShiftLeft
ShiftRight
ShiftRightLogical

Closure Allocation
CreateClosure

Globals
LdaGlobal
LdrGlobal
LdaGlobalInsideTypeof
StaGlobalSloppy
StaGlobalStrict

Unary Operators
Inc
Dec
LogicalNot
TypeOf
DeletePropertyStrict
DeletePropertySloppy

Call Operations
Call
TailCall
CallRuntime
CallRuntimeForPair
CallJsRuntime
InvokeIntrinsic

New Operator
New

Test Operators
TestEqual
TestNotEqual
TestEqualStrict
TestLessThan
TestGreaterThan
TestLessThanOrEqual
TestGreaterThanOrEqual
TestInstanceOf
TestIn

Context Operations
PushContext
PopContext
LdaContextSlot
LdrContextSlot
StaContextSlot

Cast Operators
ToName
ToNumber
ToObject

Arguments Allocation
CreateMappedArguments
CreateUnmappedArguments
CreateRestParameter

Register Transfers
Ldar
Star
Mov

Control Flow
Jump
JumpConstant
JumpIfTrue
JumpIfTrueConstant
JumpIfFalse
JumpIfFalseConstant
JumpIfToBooleanTrue
JumpIfToBooleanTrueConstant
JumpIfToBooleanFalse
JumpIfToBooleanFalseConstant
JumpIfNull
JumpIfNullConstant
JumpIfUndefined
JumpIfUndefinedConstant
JumpIfNotHole
JumpIfNotHoleConstant

Non-Local Flow Control
Throw
ReThrow
Return

Literals
CreateRegExpLiteral
CreateArrayLiteral
CreateObjectLiteral

Load Property Operations
LdaNamedProperty
LdaKeyedProperty
KeyedLoadICStrict

Store Property Operations
StoreICSloppy
StoreICStrict
KeyedStoreICSloppy
KeyedStoreICStrict

Complex Flow Control
ForInPrepare
ForInNext
ForInDone
ForInStep

Generators
SuspendGenerator
ResumeGenerator

Contemporary JavaScript Engines

JavaScriptCore (Apple)

- Direct threaded (== bigger code and data, but fast).
- Register Machine.
- Custom assembler generating bytecode handlers in dispatch loop.

SpiderMonkey (Mozilla)

- Indirect threaded.
- Stack Machine.
- Interpreter implemented in C++ as either switch statement or goto table (depending on compiler).

Chakra (Microsoft)

- Register based bytecode and C++ based interpreter.
- Optimizing compiler can run concurrently with bytecode generation.

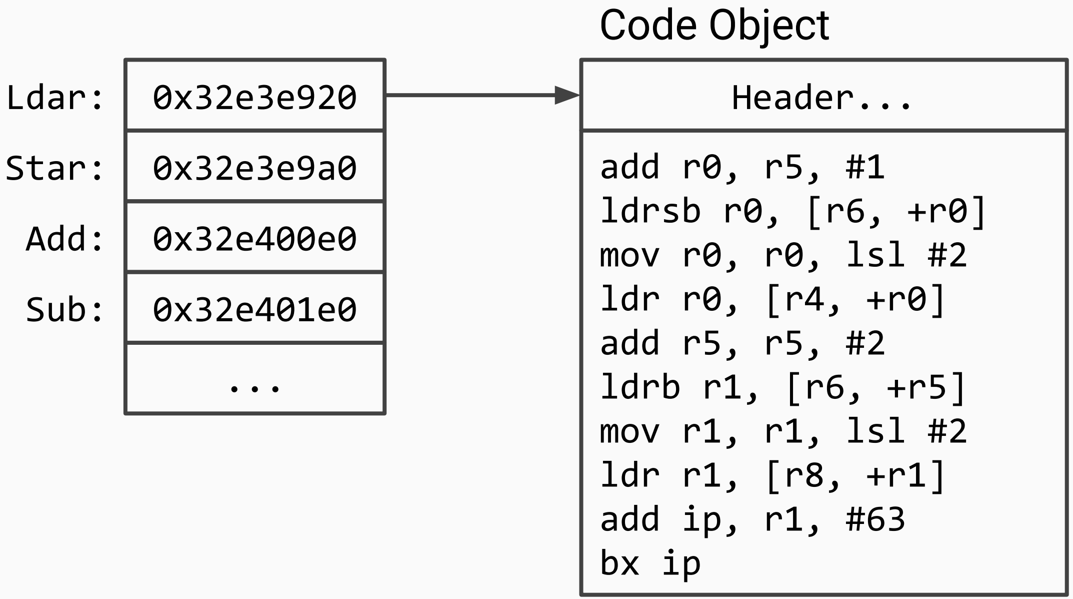
Portability

Component	Shared SLOC	Per Platform SLOC	Total SLOC
Full Code	2,000	3,700	31,600
Crankshaft	32,000	9,300	108,000
		Overall	139,600
Ignition	10,000	250	12,000
Turbofan	58,000	3,800	88,400
		Overall	100,400

Indirect Threaded Bytecode Dispatch

Ldar:	0x32e3e920
Star:	0x32e3e9a0
Add:	0x32e400e0
Sub:	0x32e401e0
	...

Indirect Threaded Bytecode Dispatch



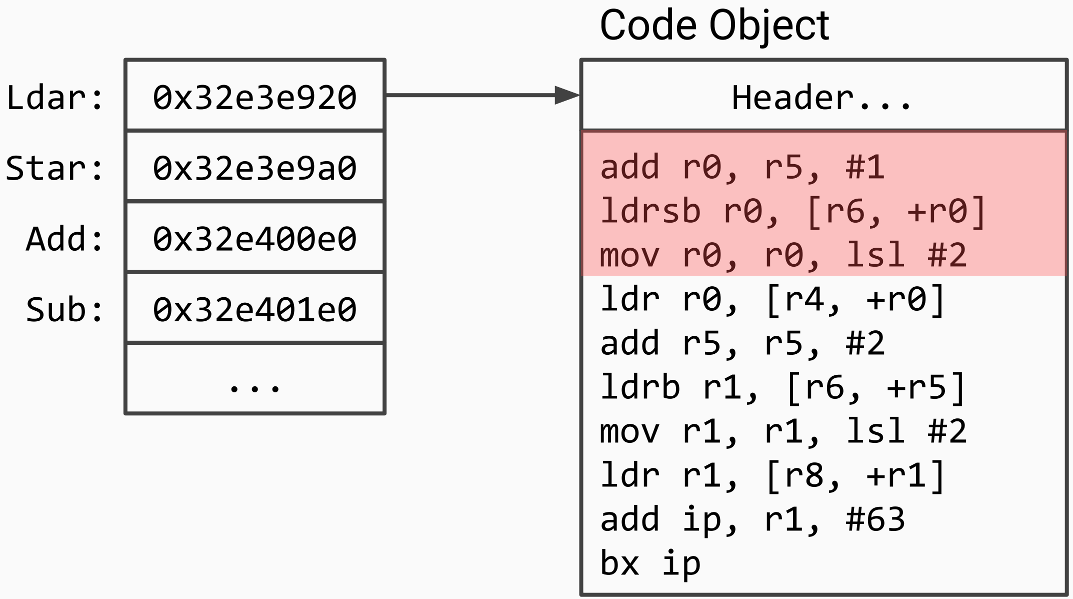
Ldar:	0x32e3e920
Star:	0x32e3e9a0
Add:	0x32e400e0
Sub:	0x32e401e0
	...

Code Object

Header...

```
add r0, r5, #1
ldrsb r0, [r6, +r0]
mov r0, r0, lsl #2
ldr r0, [r4, +r0]
add r5, r5, #2
ldrb r1, [r6, +r5]
mov r1, r1, lsl #2
ldr r1, [r8, +r1]
add ip, r1, #63
bx ip
```

Indirect Threaded Bytecode Dispatch



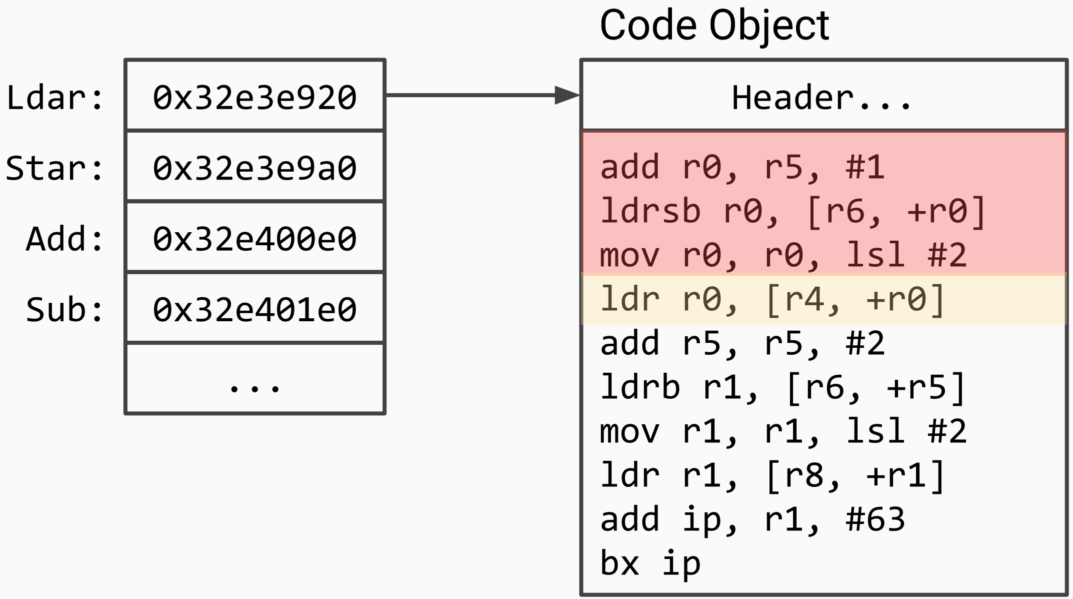
Ldar:	0x32e3e920
Star:	0x32e3e9a0
Add:	0x32e400e0
Sub:	0x32e401e0
	...

Code Object

```
Header...
add r0, r5, #1
ldrsb r0, [r6, +r0]
mov r0, r0, lsl #2
ldr r0, [r4, +r0]
add r5, r5, #2
ldrb r1, [r6, +r5]
mov r1, r1, lsl #2
ldr r1, [r8, +r1]
add ip, r1, #63
bx ip
```

```
BytecodeOperandReg(0);
```

Indirect Threaded Bytecode Dispatch



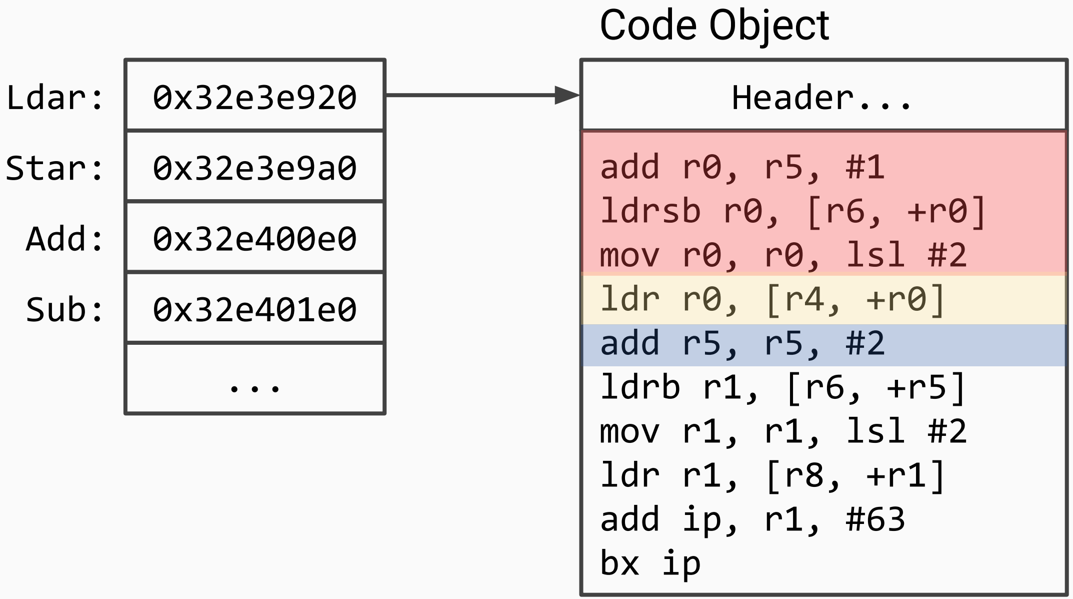
Ldar:	0x32e3e920
Star:	0x32e3e9a0
Add:	0x32e400e0
Sub:	0x32e401e0
	...

Code Object

```
Header...
add r0, r5, #1
ldrsb r0, [r6, +r0]
mov r0, r0, lsl #2
ldr r0, [r4, +r0]
add r5, r5, #2
ldrb r1, [r6, +r5]
mov r1, r1, lsl #2
ldr r1, [r8, +r1]
add ip, r1, #63
bx ip
```

```
BytecodeOperandReg(0);
LoadRegister(reg_idx);
SetAccumulator(value);
```

Indirect Threaded Bytecode Dispatch



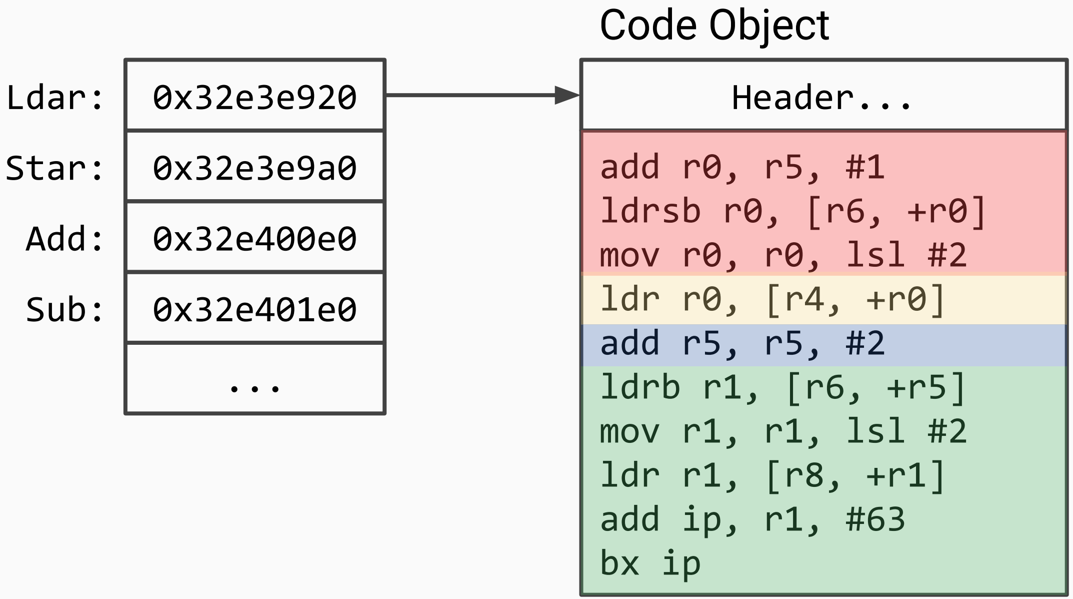
Ldar:	0x32e3e920
Star:	0x32e3e9a0
Add:	0x32e400e0
Sub:	0x32e401e0
	...

Code Object

```
Header...
add r0, r5, #1
ldrsb r0, [r6, +r0]
mov r0, r0, lsl #2
ldr r0, [r4, +r0]
add r5, r5, #2
ldrb r1, [r6, +r5]
mov r1, r1, lsl #2
ldr r1, [r8, +r1]
add ip, r1, #63
bx ip
```

```
BytecodeOperandReg(0);
LoadRegister(reg_idx);
SetAccumulator(value);
Advance();
```

Indirect Threaded Bytecode Dispatch



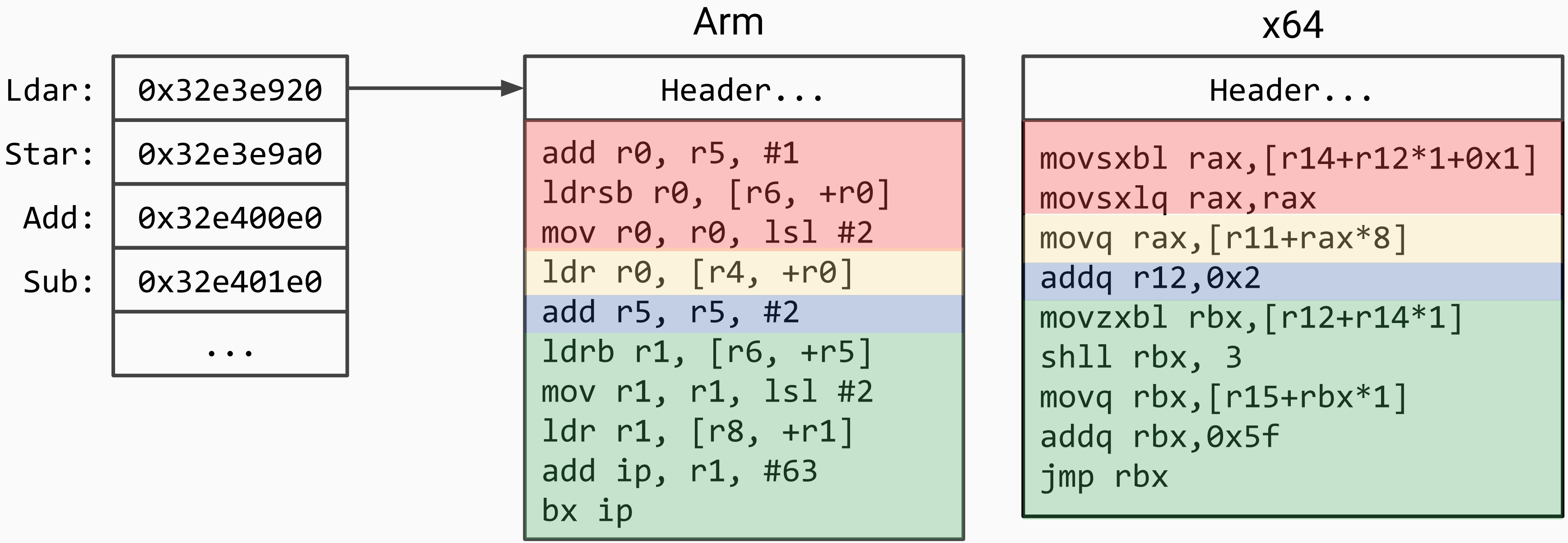
Ldar:	0x32e3e920
Star:	0x32e3e9a0
Add:	0x32e400e0
Sub:	0x32e401e0
	...

Code Object

```
Header...
add r0, r5, #1
ldrsb r0, [r6, +r0]
mov r0, r0, lsl #2
ldr r0, [r4, +r0]
add r5, r5, #2
ldrb r1, [r6, +r5]
mov r1, r1, lsl #2
ldr r1, [r8, +r1]
add ip, r1, #63
bx ip
```

```
BytecodeOperandReg(0);
LoadRegister(reg_idx);
SetAccumulator(value);
Advance();
Dispatch();
```

Indirect Threaded Bytecode Dispatch



Ldar:	0x32e3e920
Star:	0x32e3e9a0
Add:	0x32e400e0
Sub:	0x32e401e0
	...

Arm

```
Header...
add r0, r5, #1
ldrsb r0, [r6, +r0]
mov r0, r0, lsl #2
ldr r0, [r4, +r0]
add r5, r5, #2
ldrb r1, [r6, +r5]
mov r1, r1, lsl #2
ldr r1, [r8, +r1]
add ip, r1, #63
bx ip
```

x64

```
Header...
movsxbl rax,[r14+r12*1+0x1]
movsxlq rax,rax
movq rax,[r11+rax*8]
addq r12,0x2
movzxbl rbx,[r12+r14*1]
shll rbx, 3
movq rbx,[r15+rbx*1]
addq rbx,0x5f
jmp rbx
```

TurboFan Language Levels

- **JavaScript**: (“JS”) operators (`JSAdd`, `JSSubtract`, `JSCall`)
    - Express semantics of JavaScript’s overloaded operators
    - Produce and consume effects in the graph
- **Intermediate**: (“Simplified”) operators (`NumberAdd`, `NumberSubtract`)
    - Express VM-level operations, such as allocation, bounds checks
    - Arithmetic independent of number representation
- **Machine**: (“Machine”) operators (`Int32Add`, `Int64Add`)
    - Correspond closely to single machine instructions
    - Most have no side effects
    - Must be supported by backend for each platform