V8 Compilation Pipeline @ PLDI 2019

Michael Starzinger, mstarzinger@google.com

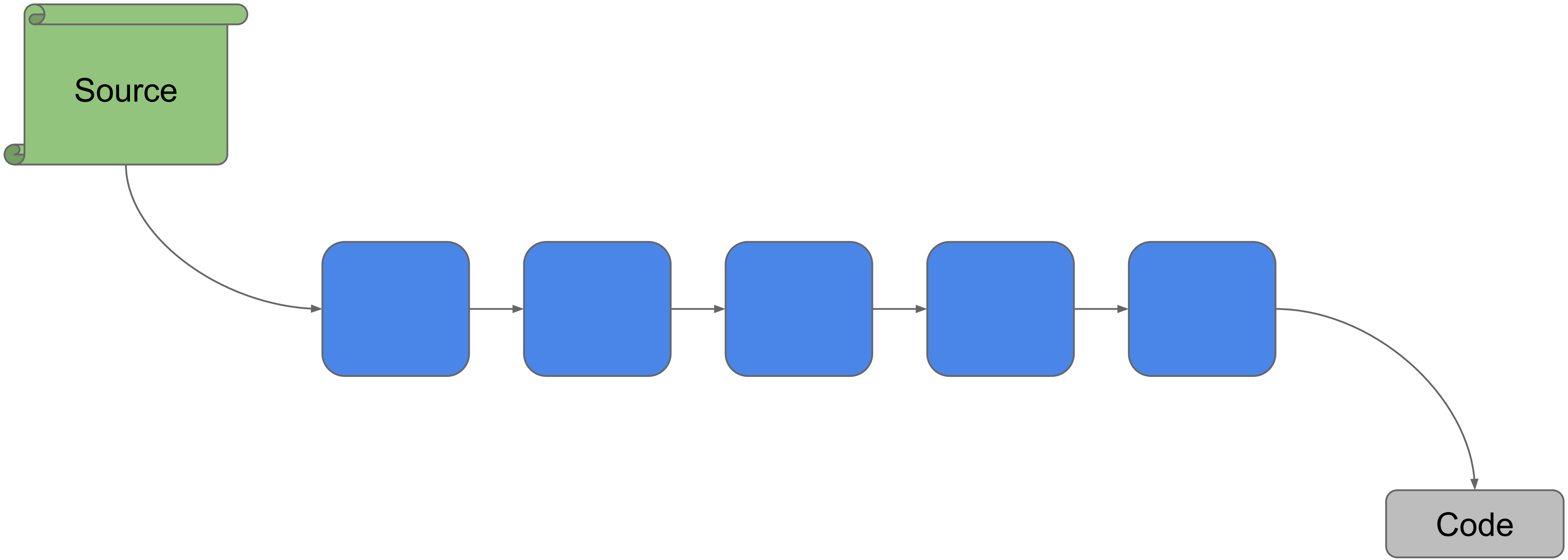
Agenda

- Optimizing Compiler Overview
  - Introduction to TurboFan IR
  - Multi-Purpose Compilation Pipeline
  - Demo IR Graph Visualizer
- JavaScript optimization in TurboFan
  - Type-Feedback Based Optimization
  - Optimization of Higher-Order Builtins
  - Open Research Problems
- WebAssembly and TurboFan
  - Tiering Strategies for WebAssembly
  - Open Research Problems

TurboFan Overview

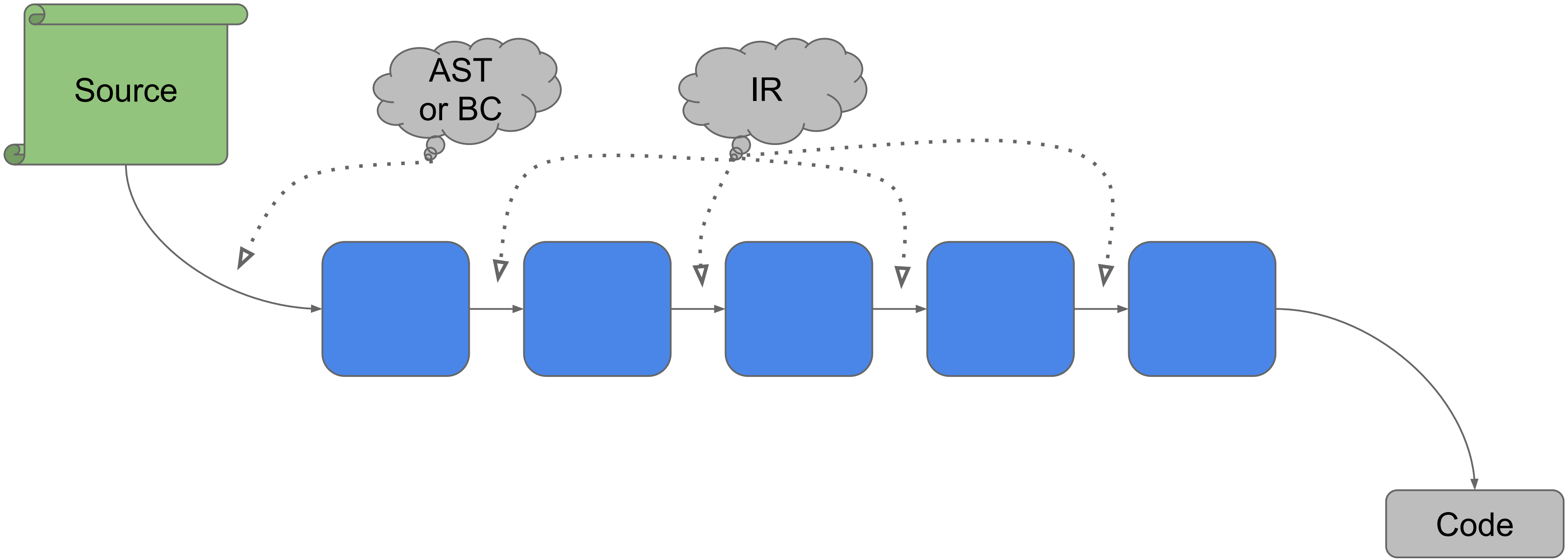
Google

Compiler Pipeline



Google

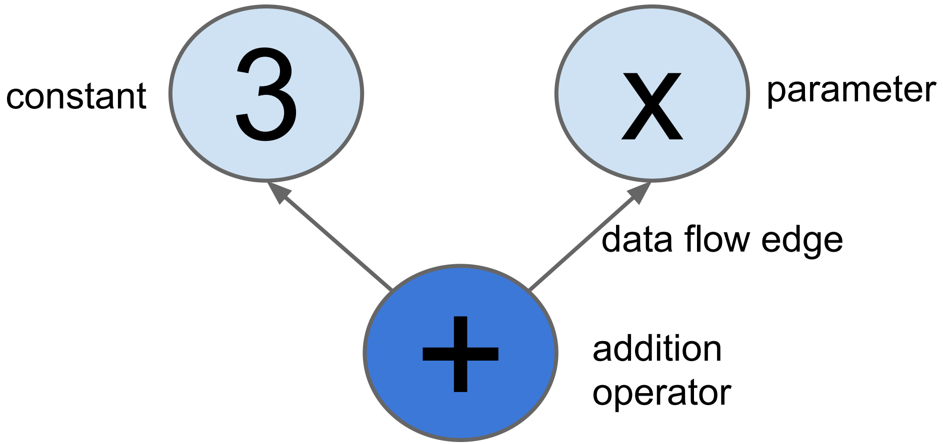
Compiler Pipeline



Intermediate Representation

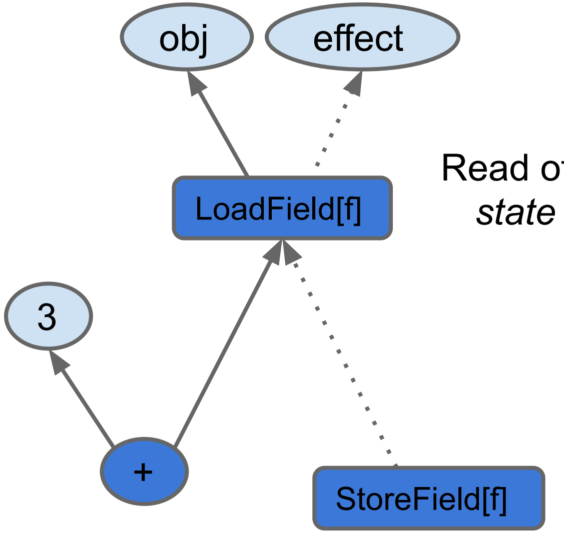
- Graph: Sea-of-Nodes style IR based on SSA
  - Allows in-place mutation of nodes throughout
- Nodes: Express computation
  - Examples: constants, parameters, arithmetic, load, store, calls
  - Source program is SSA renamed, all local variables replaced
- Edges: express data dependencies (constrain order)
  - Dataflow edges express using the value output of a computation
- More Edges: other dependencies (constrain order further)
  - Effect edges order operations reading and writing state
  - Control edges are how we express non-straight line code

Value Edges

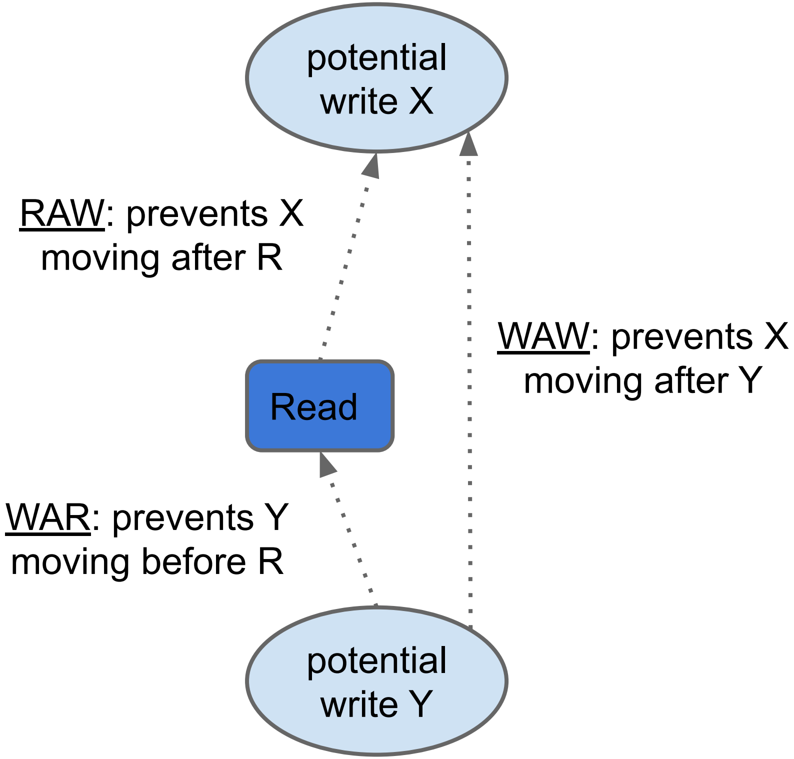


Nodes: All computations are expressed as nodes in the sea
Edges: Represent dependencies between computations

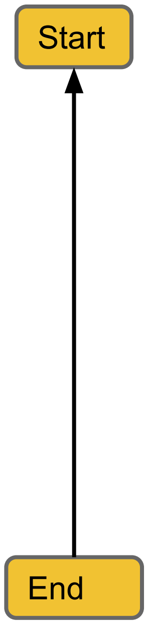
Effect Edges



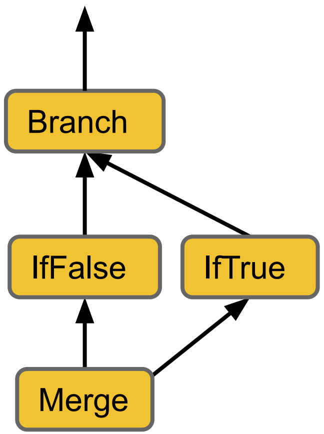
Read of *mutable state* `obj.f`



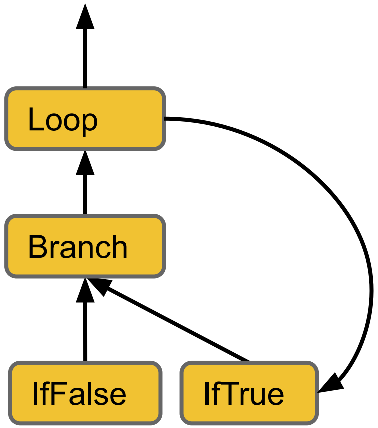
Control Edges



straight-line program



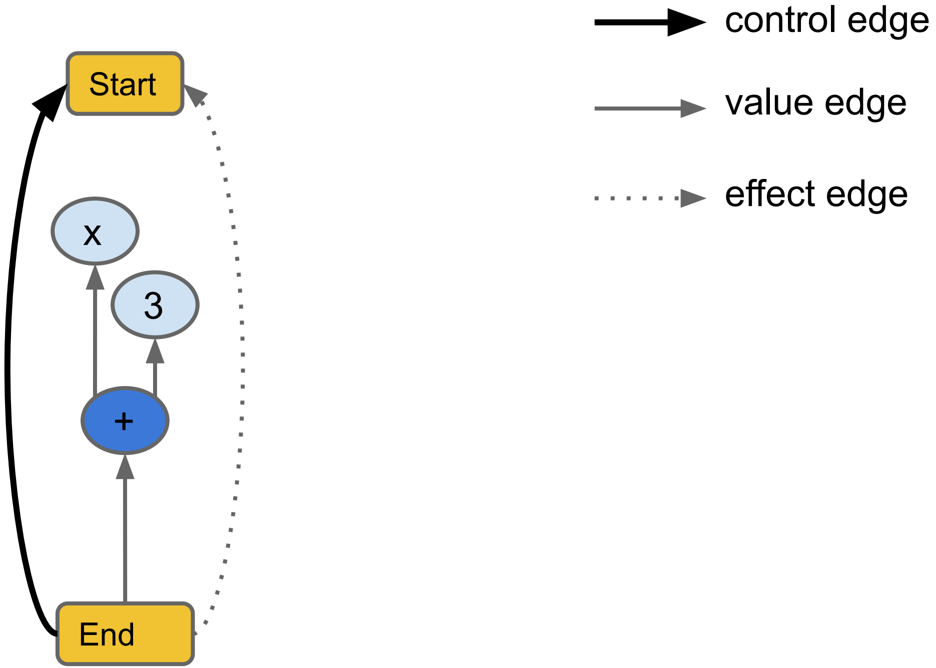
branch



while loop

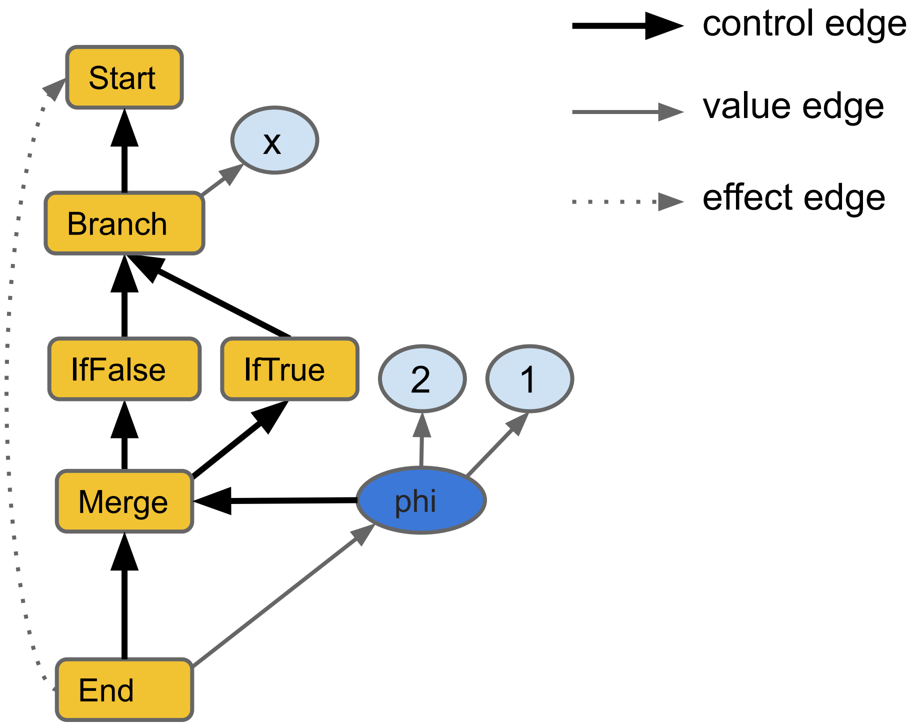
Complete Graph (1)

```
function (x) {
  return x + 3;
}
```



Complete Graph (2)

```
function (x) {
  return x ? 1 : 2;
}
```



Operator Language Levels

- JavaScript:

  JSAdd JSSubtract JSMultiply JSDivide JSModulus JSDivide
  JSBitwiseOr JSBitwiseAnd JSBitwiseXor JSShiftLeft JSShiftRight
  JSEqual JSStrictEqual JSToBoolean JSToNumber JSCall

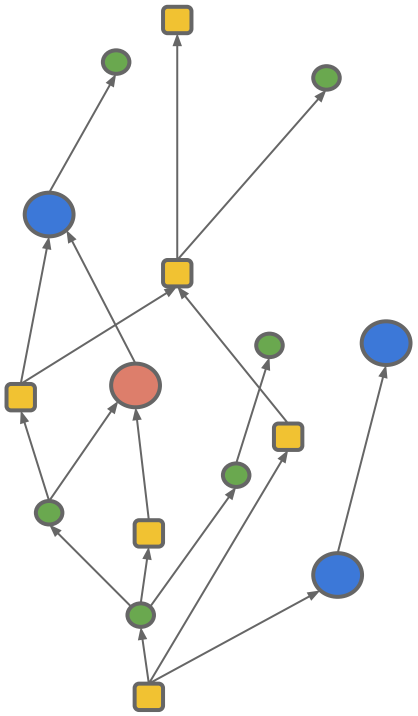
- Simplified:

  NumberAdd NumberSub NumberMul NumberDiv NumberMod
  NumberEqual NumberLessThan LoadField StoreField
  StringEqual StringAdd ChangeTaggedToInt32

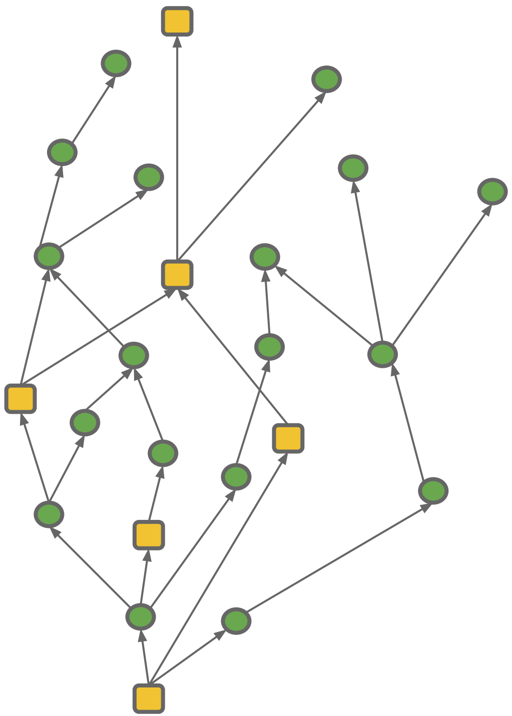
- Machine:

  Int32Add Int32Sub Int32Mul Float64Add Float64Sub Float64Mul
  Load Store Call ConvertFloat64ToInt32

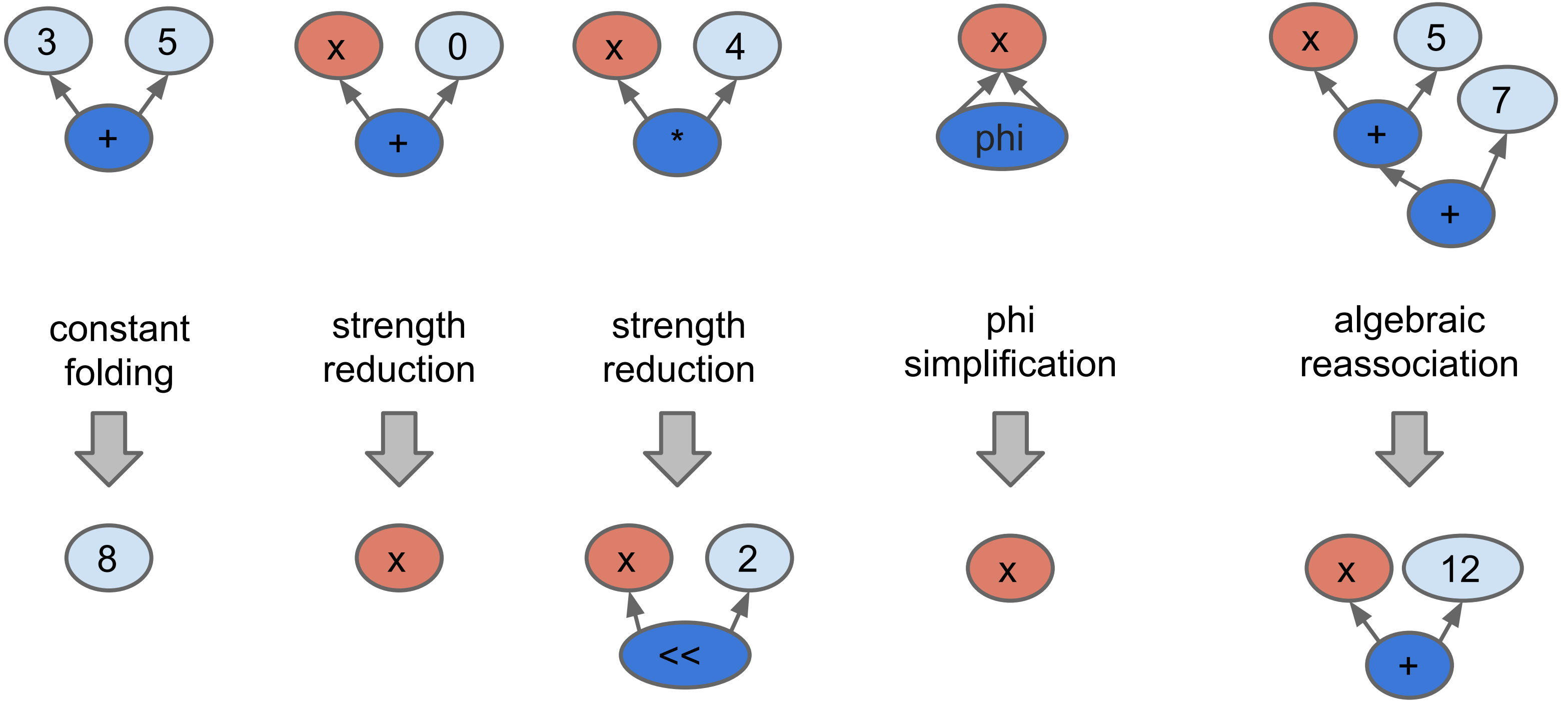
Operator Language Levels



expand and optimize
“JavaScript” and “Simplified”
nodes (aka. lowering)

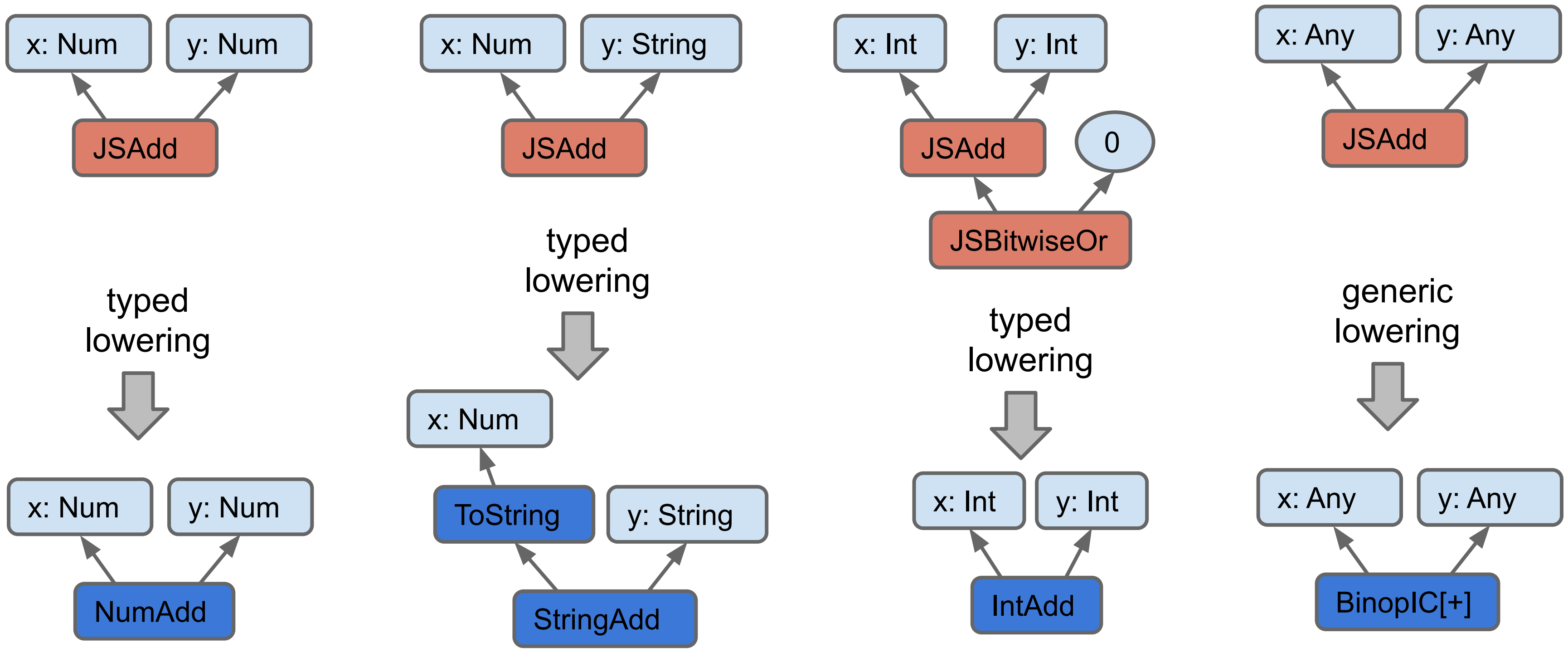


Reduction (1)



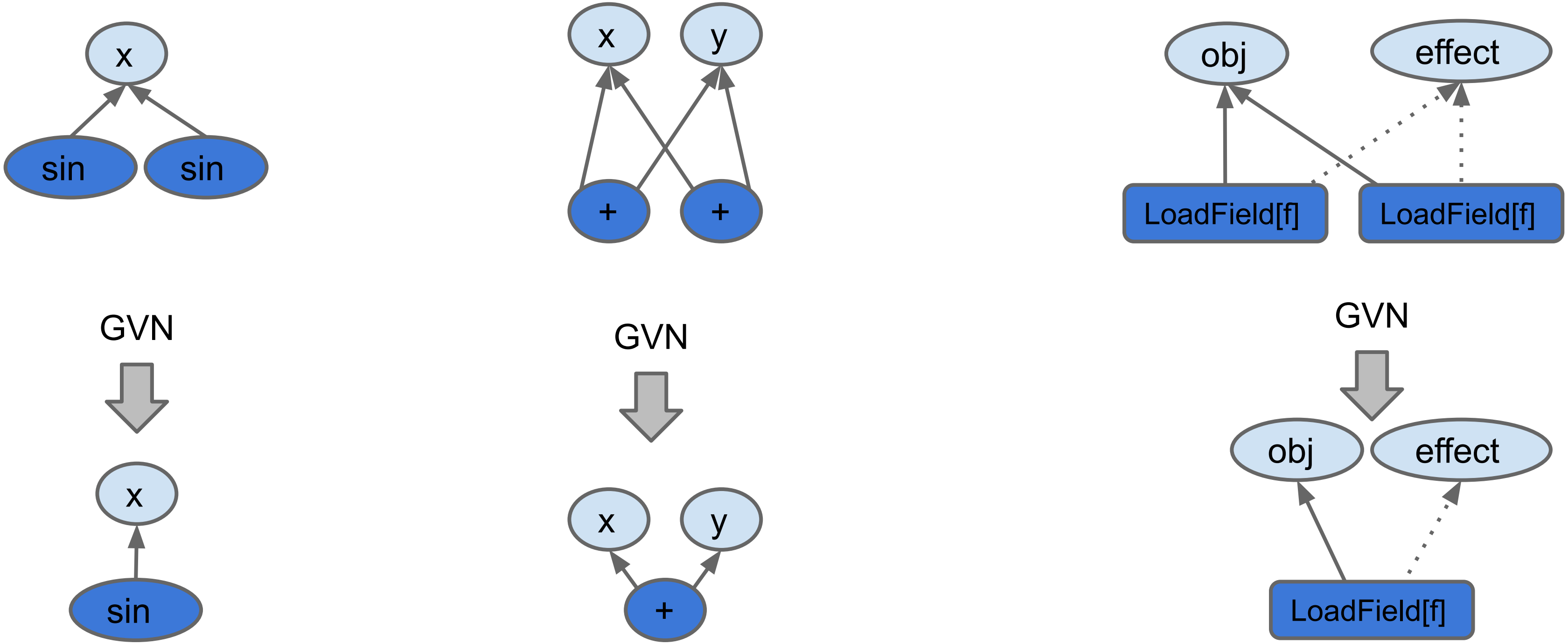
3
5
+
x
0
+
x
4
*
x
phi
x
5
7
+
+
constant folding
strength reduction
strength reduction
phi simplification
algebraic reassociation
8
x
x
2
<<
x
x
12
+

Reduction (2) - Typed Lowering



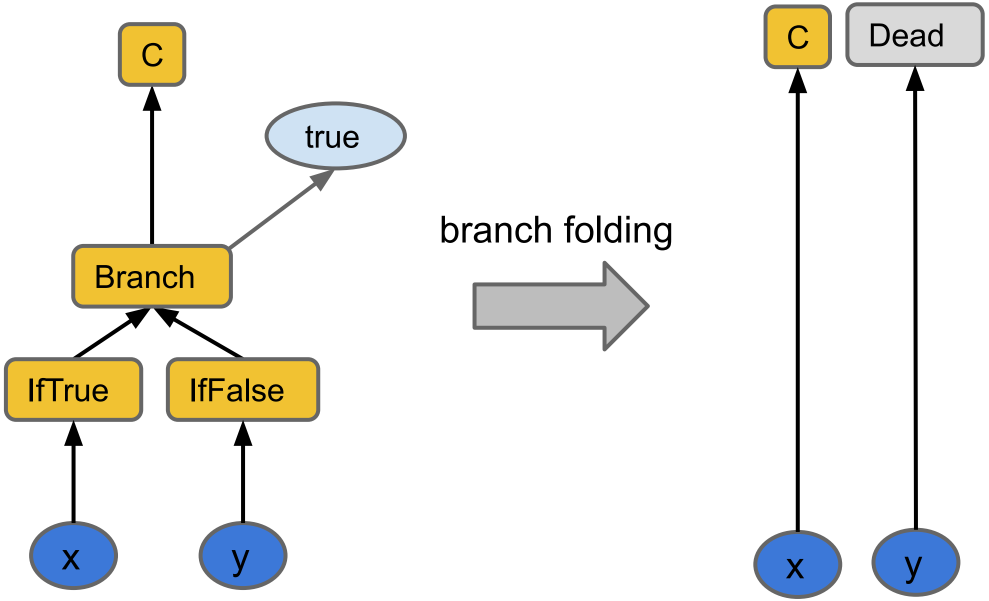
Google

Reduction (3) - Global Value Numbering

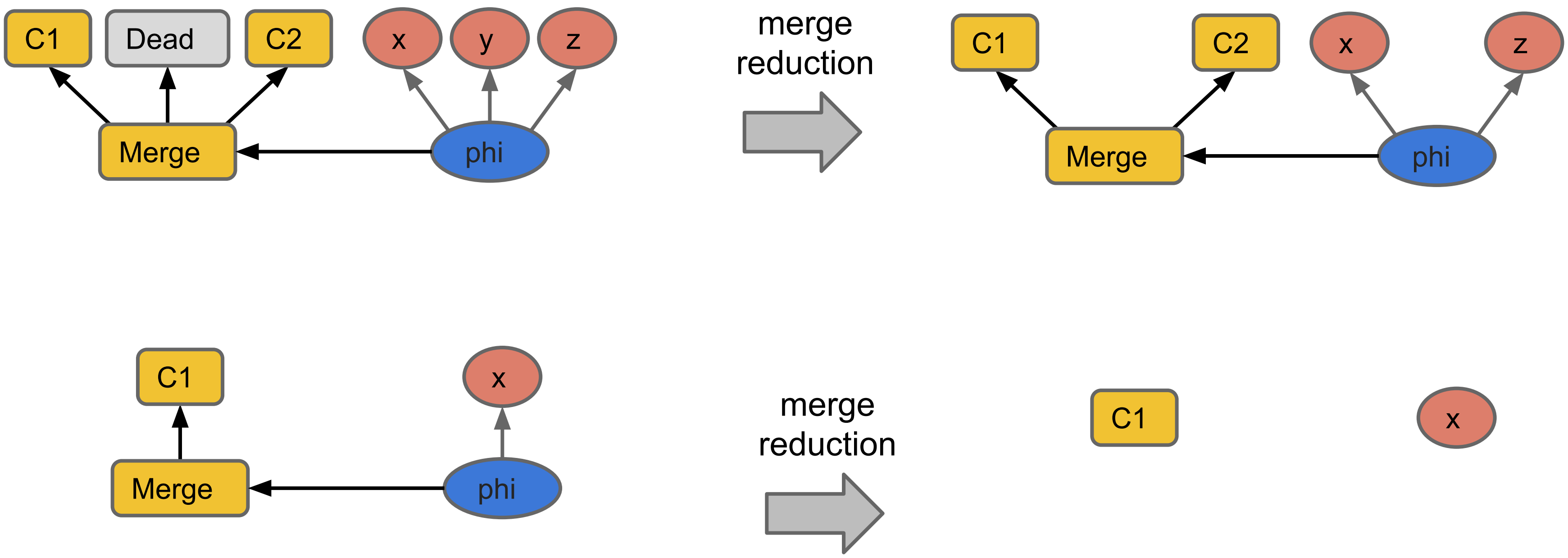


Google

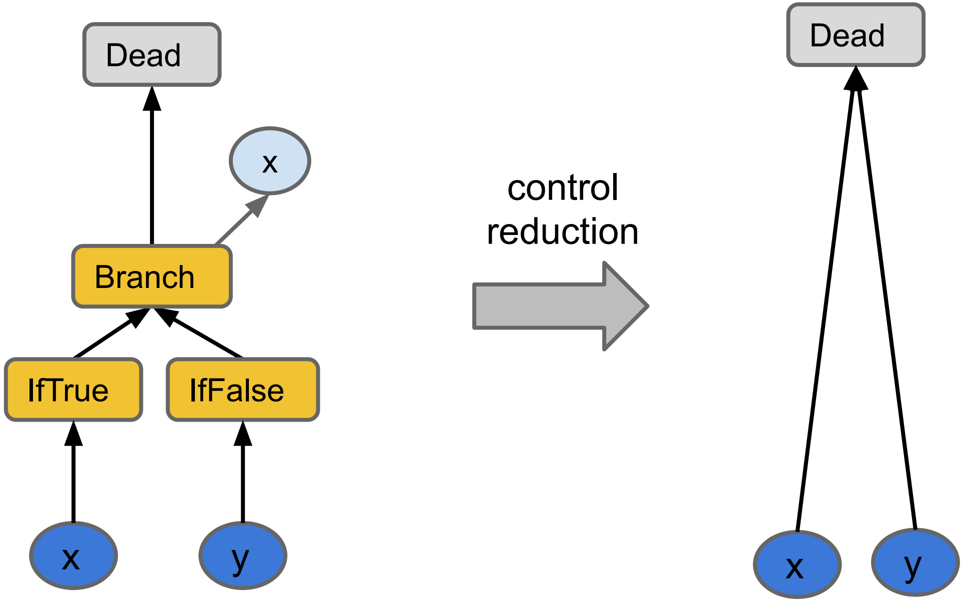
Reduction (4) - Control Optimization



Reduction (5) - Control Optimization

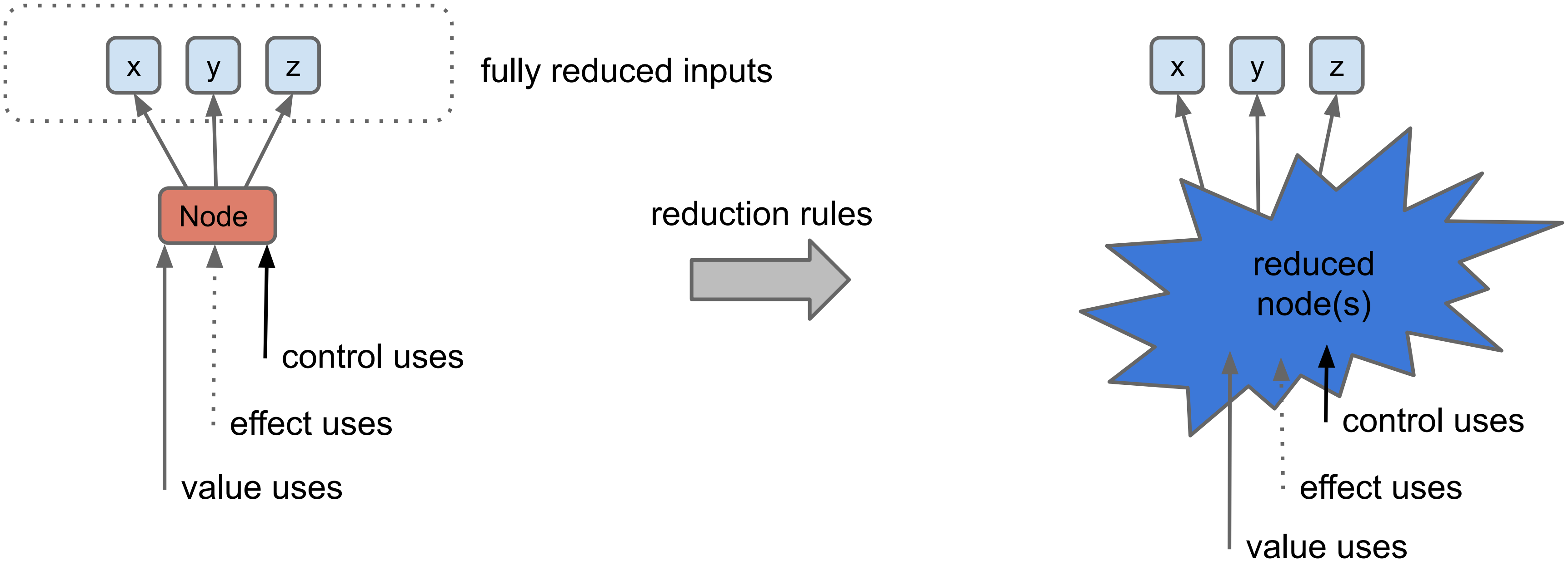


Reduction (6) - Control Optimization



Google

Reduction as Top-Down Graph Rewriting



Turbolizer

TurboFan IR graph visualizer

- Browser based (loads JSON dump)
- Interactive navigation
- End-to-end mapping

Demo Time!

Phases and IR Layering

Graph building

Typed specialization, inlining

Typing, typed lowering

Representation selection

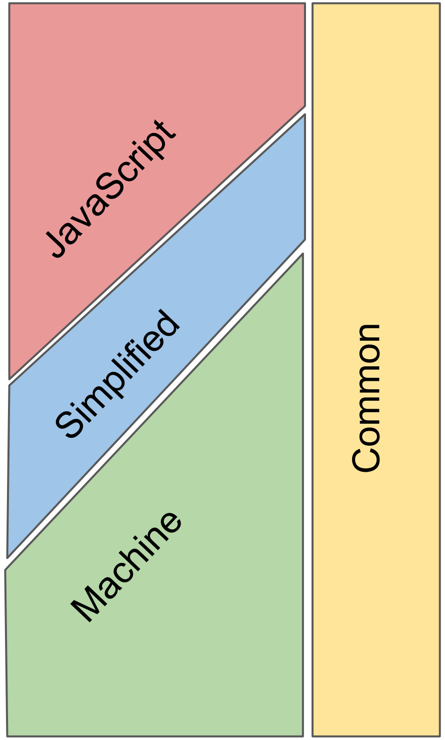
JS Generic lowering

Early optimizations

Effect-control linearization

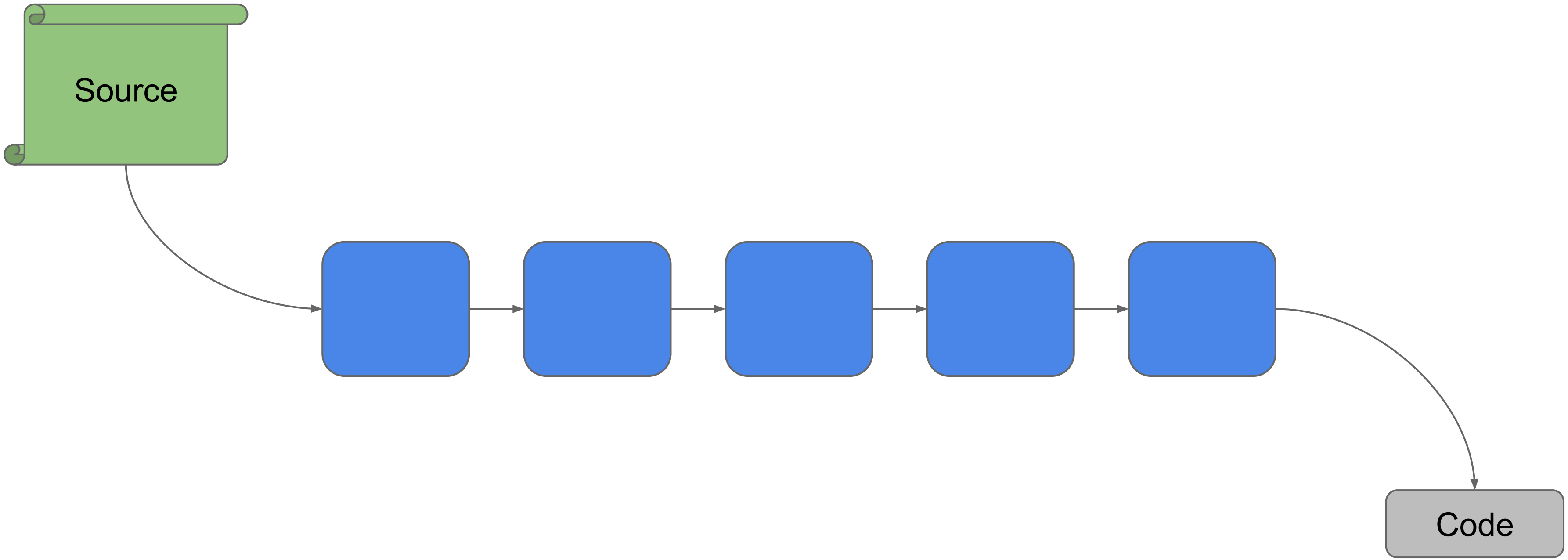
Late optimizations

Scheduling & instruction selection

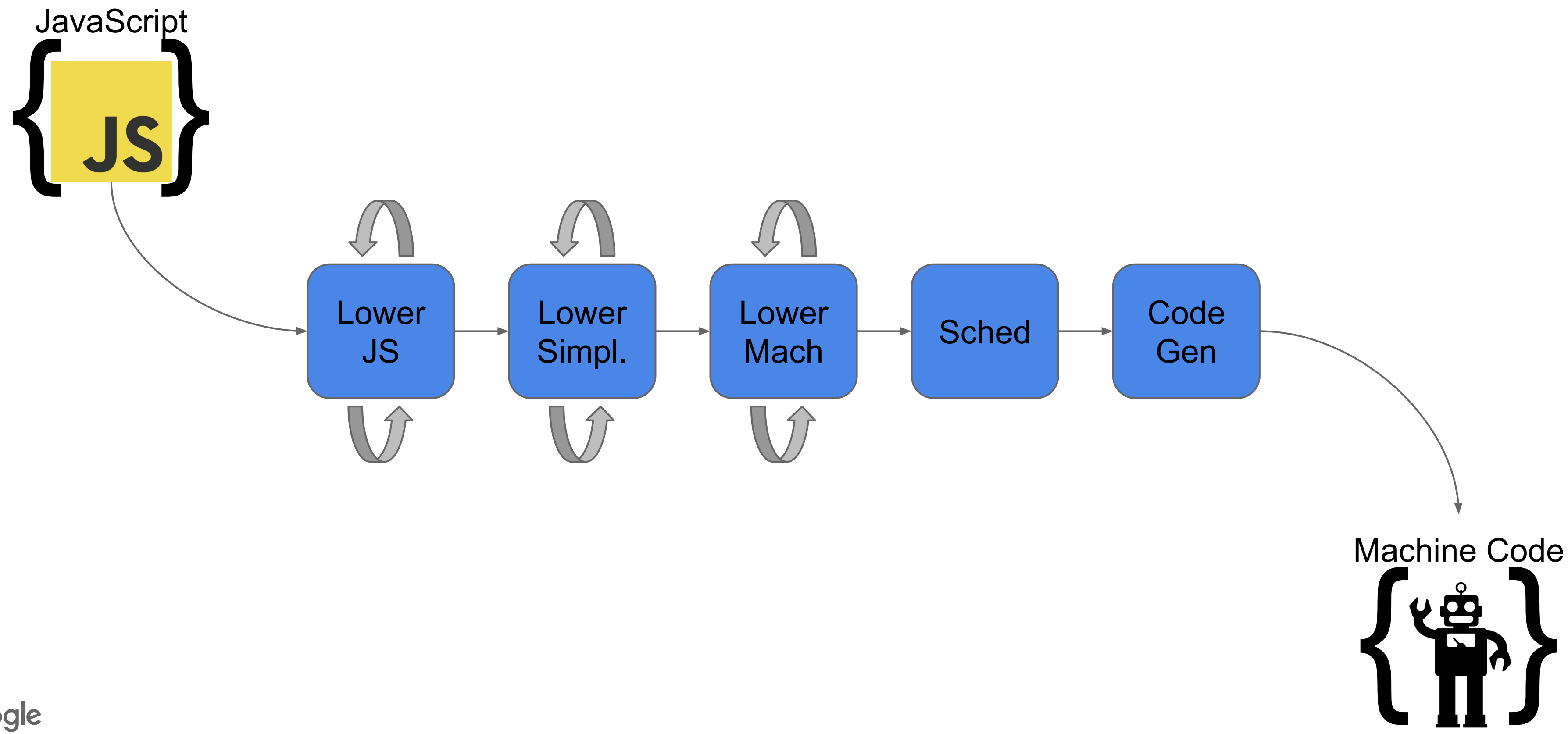


Google

Compiler Pipeline (revisited)

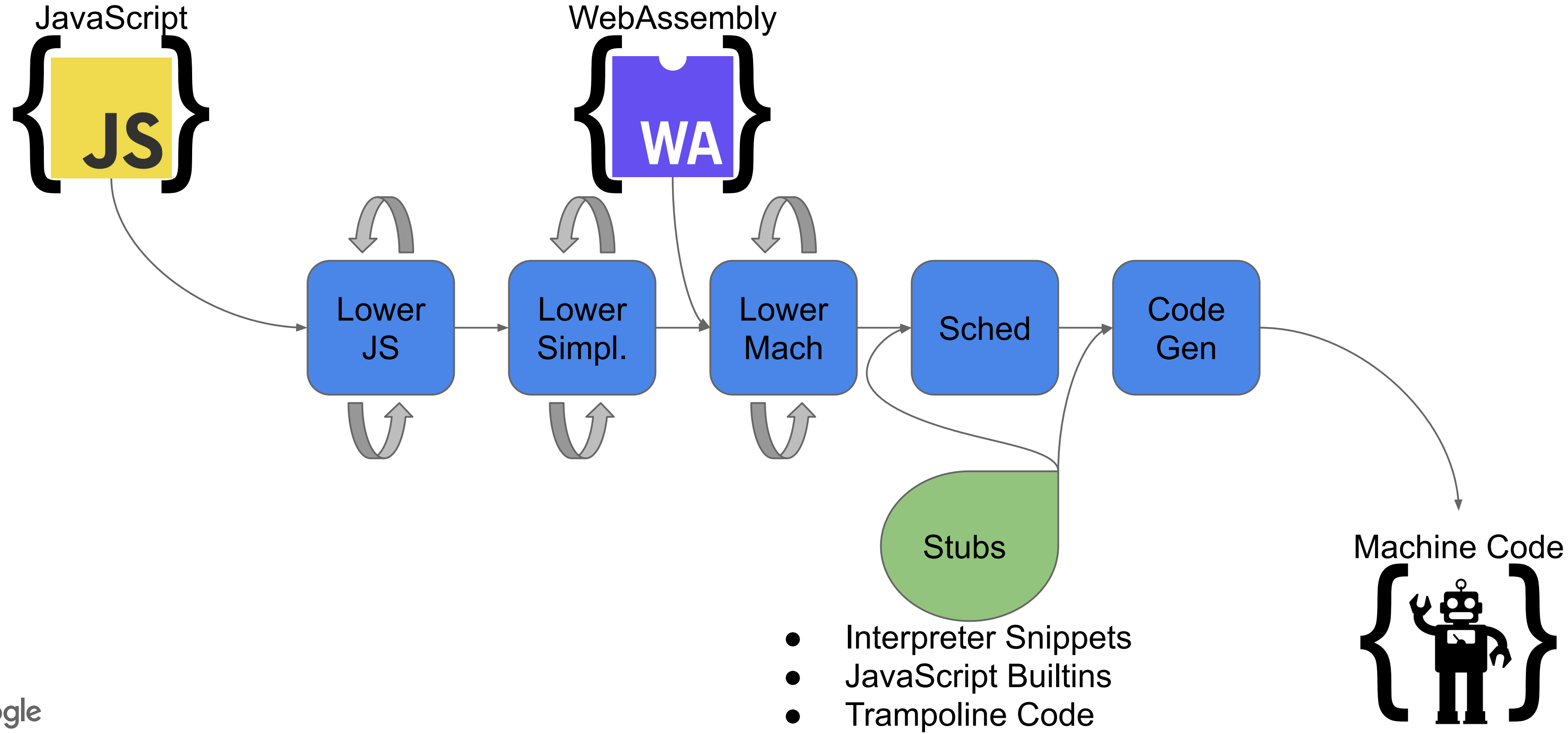


Compiler Pipeline (revisited)



Google

Compiler Pipeline (revisited)



- Interpreter Snippets
- JavaScript Builtins
- Trampoline Code

Recap & Additional Notes

- TurboFan uses sea-of-nodes IR
  - Flexible pipeline used for: JavaScript, WebAssembly & Builtins
  - Operators layered in language levels
  - Mutable graph, reduction & lowering
- Supports advanced language constructs
  - Can model exceptional control-flow as well
  - Supports deoptimization at checkpoints
- Function-Inlining done on IR graph
  - When call target(s) statically/speculatively known

Google

JavaScript & TurboFan

A Tale of Two Compilers

Ignition

Bytecode Interpreter

TurboFan

Optimizing Compiler

A Tale of Two Compilers

- Fast startup speed
- Low memory use
- Handles all cases

Ignition

Bytecode Interpreter

- Requires warm-up
- Longer compilation
- Peak performance

TurboFan

Optimizing Compiler

A Tale of Two Compilers

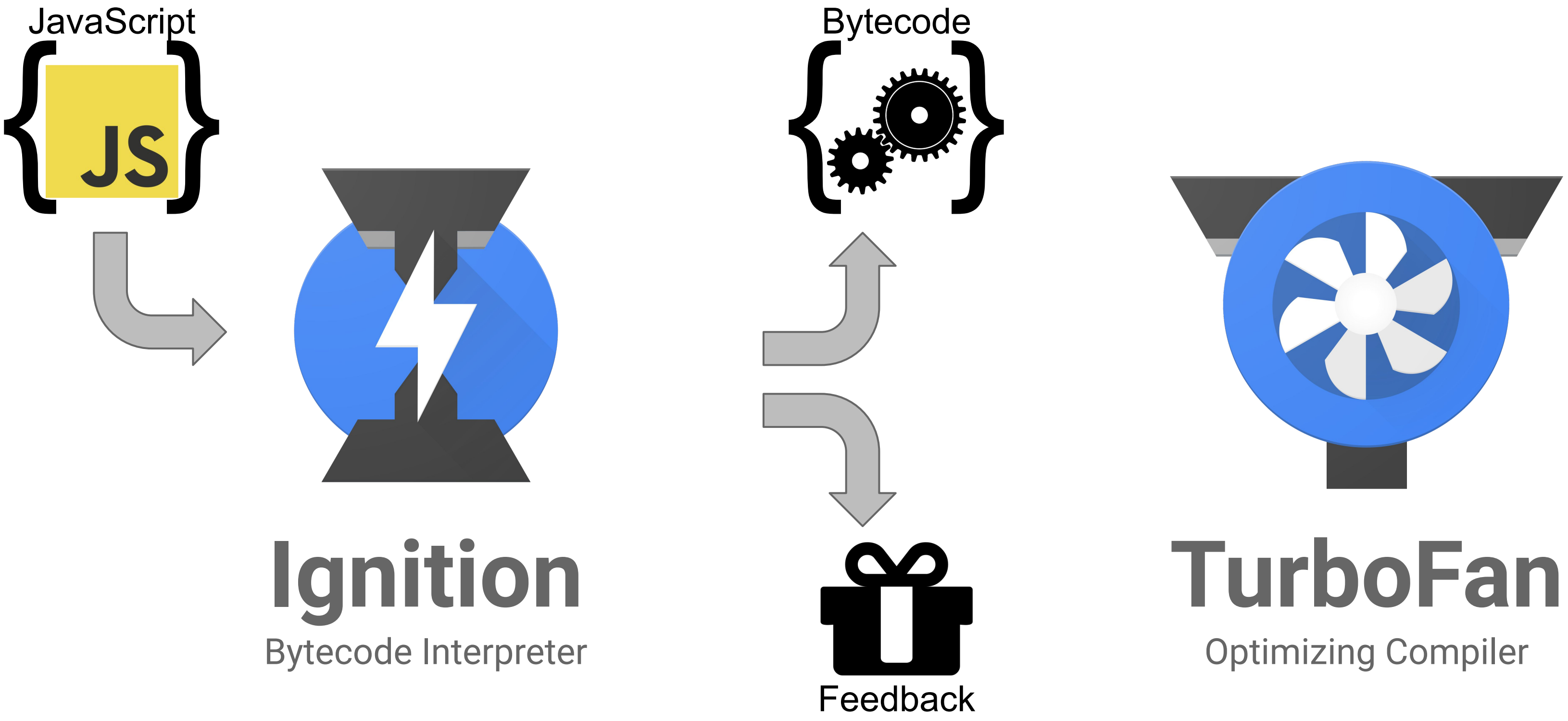
Ignition

Bytecode Interpreter

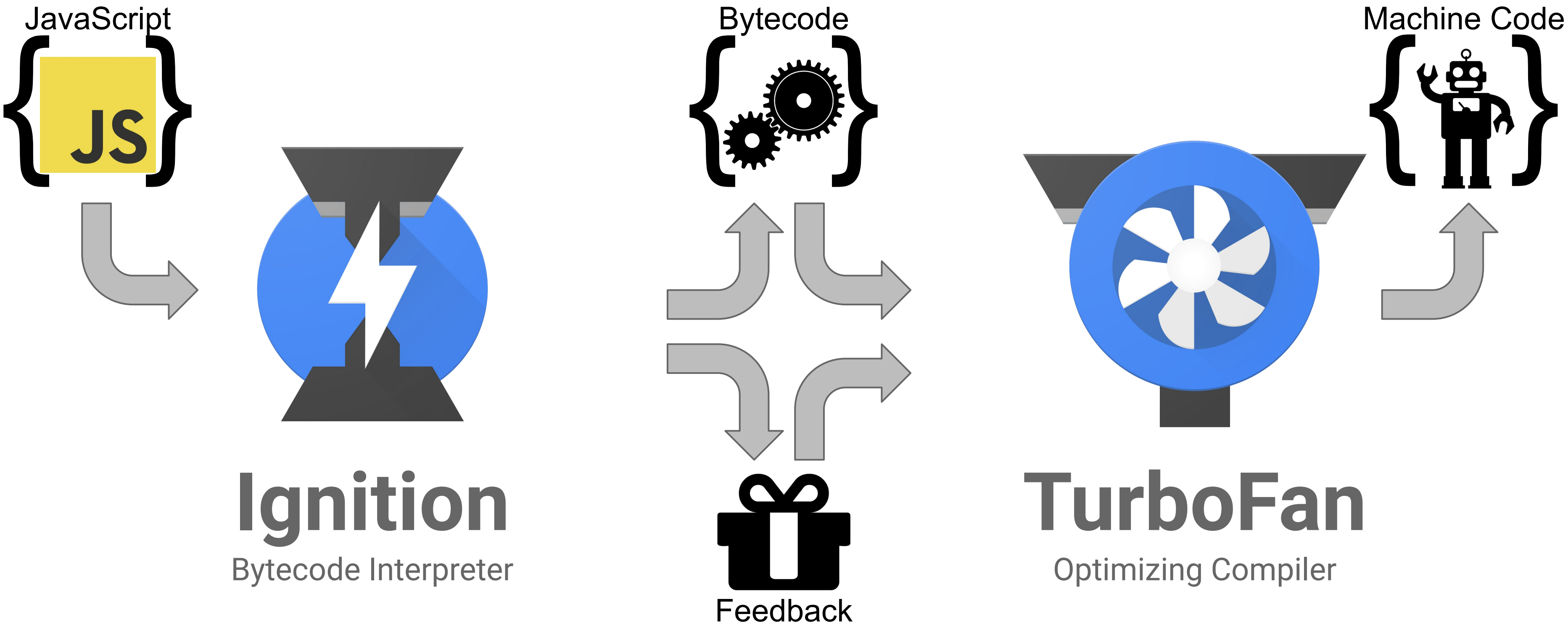
TurboFan

Optimizing Compiler

A Tale of Two Compilers



A Tale of Two Compilers



Type Feedback - Motivation

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

function f() {
  return add(2, 3);
}
```

Type Feedback - Motivation

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

function f() {
  return add(2, 3);
}
```

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **+** *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Let *lstr* be ? ToString(*lprim*).
    b. Let *rstr* be ? ToString(*rprim*).
    c. Return the string-concatenation of *lstr* and *rstr*.
8. Let *lnum* be ? ToNumber(*lprim*).
9. Let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.8.5.

Type Feedback - Motivation

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

function f() {
  return add(2, 3);
}
```

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **+** *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String
   a. Let *lstr* be ? ToStr
   b. Let *rstr* be ? ToStr
   c. Return the string-c
8. Let *lnum* be ? ToNumbe
9. Let *rnum* be ? ToNumbe
10. Return the result of appl

7.1.1 ToPrimitive (*input* [, *PreferredType*])

The abstract operation ToPrimitive takes an *input* argument and an optional argument *PreferredType*. The abstract operation ToPrimitive converts its *input* argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint *PreferredType* to favour that type. Conversion occurs according to the following algorithm:

1. Assert: *input* is an ECMAScript language value.
2. If Type(*input*) is Object, then
   a. If *PreferredType* is not present, let *hint* be `"default"`.
   b. Else if *PreferredType* is hint String, let *hint* be `"string"`.
   c. Else *PreferredType* is hint Number, let *hint* be `"number"`.
   d. Let *exoticToPrim* be ? GetMethod(*input*, @@toPrimitive).
   e. If *exoticToPrim* is not **undefined**, then
      i. Let *result* be ? Call(*exoticToPrim*, *input*, « *hint* »).
      ii. If Type(*result*) is not Object, return *result*.
      iii. Throw a **TypeError** exception.
   f. If *hint* is `"default"`, set *hint* to `"number"`.
   g. Return ? OrdinaryToPrimitive(*input*, *hint*).
3. Return *input*.

Type Feedback - Motivation

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

function f() {
  return add(2, 3);
}
```

12.8.3.1 Runtime Seman

AdditiveExpression : *Additiv*

1. Let *lref* be the result of
2. Let *lval* be ? GetValue(*l*
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String
   a. Let *lstr* be ? ToStr
   b. Let *rstr* be ? ToStr
   c. Return the string-c
8. Let *lnum* be ? ToNumbe
9. Let *rnum* be ? ToNumbe
10. Return the result of appl

7.3.9 GetMethod (*V*, *P*)

The abstract operation GetMethod is used to get the value of a specific property of an ECMAScript language value when the value of the property is expected to be a function. The operation is called with arguments *V* and *P* where *V* is the ECMAScript language value, *P* is the property key. This abstract operation performs the following steps:

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *func* be ? GetV(*V*, *P*).
3. If *func* is either **undefined** or **null**, return **undefined**.
4. If IsCallable(*func*) is **false**, throw a **TypeError** exception.
5. Return *func*.

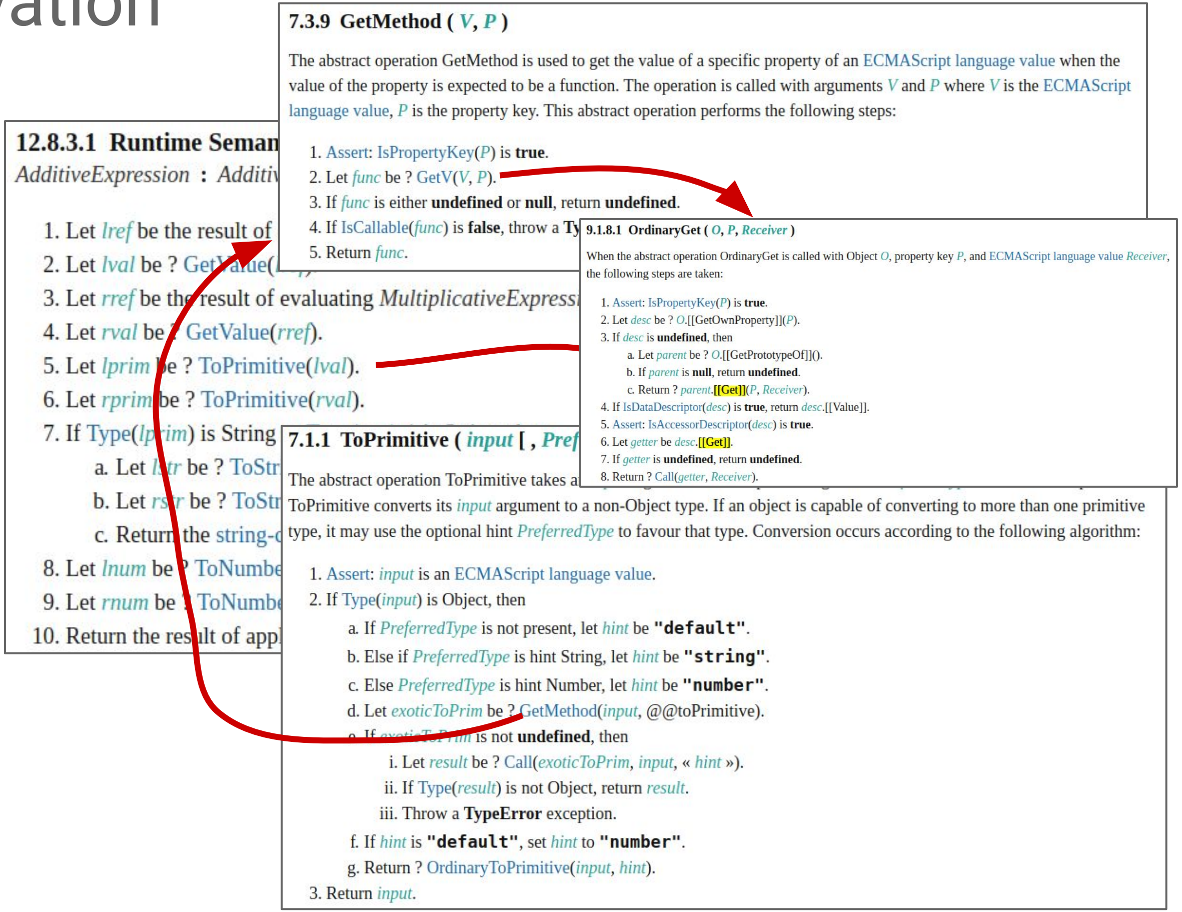
7.1.1 ToPrimitive (*input* [, *PreferredType*])

The abstract operation ToPrimitive takes an *input* argument and an optional argument *PreferredType*. The abstract operation ToPrimitive converts its *input* argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint *PreferredType* to favour that type. Conversion occurs according to the following algorithm:

1. Assert: *input* is an ECMAScript language value.
2. If Type(*input*) is Object, then
   a. If *PreferredType* is not present, let *hint* be `"default"`.
   b. Else if *PreferredType* is hint String, let *hint* be `"string"`.
   c. Else *PreferredType* is hint Number, let *hint* be `"number"`.
   d. Let *exoticToPrim* be ? GetMethod(*input*, @@toPrimitive).
   e. If *exoticToPrim* is not **undefined**, then
      i. Let *result* be ? Call(*exoticToPrim*, *input*, « *hint* »).
      ii. If Type(*result*) is not Object, return *result*.
      iii. Throw a **TypeError** exception.
   f. If *hint* is `"default"`, set *hint* to `"number"`.
   g. Return ? OrdinaryToPrimitive(*input*, *hint*).
3. Return *input*.

Type Feedback - Motivation

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

function f() {
  return add(2, 3);
}
```



Type Feedback - Motivation

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

function f() {
  return add(2, 3);
}
```

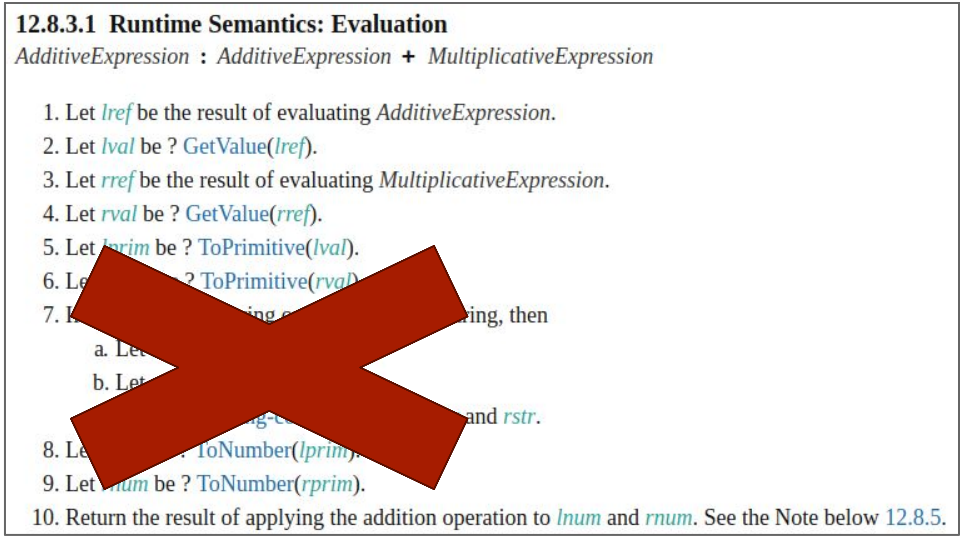
12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **+** *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
   a. Let *lstr* be ? ToString(*lprim*).
   b. Let *rstr* be ? ToString(*rprim*).
   c. Return the string-concatenation of *lstr* and *rstr*.
8. Let *lnum* be ? ToNumber(*lprim*).
9. Let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.8.5.

Type Feedback - Motivation

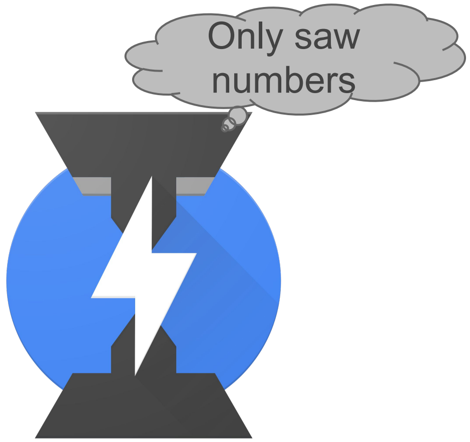
```
function add(x, y) {
  return x + y;
}

function f() {
  return add(2, 3);
}
```


12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **+** *MultiplicativeExpression*

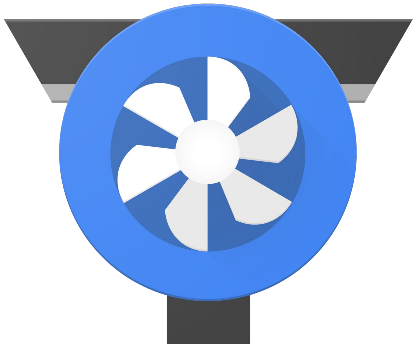
1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Le ? ToPrimitive(*rval*).
7. I ing ring, then
   a. Le
   b. Let
   and *rstr*.
8. Le ToNumber(*lprim*).
9. Let num be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.8.5.

Type Feedback - Illustration



Ignition

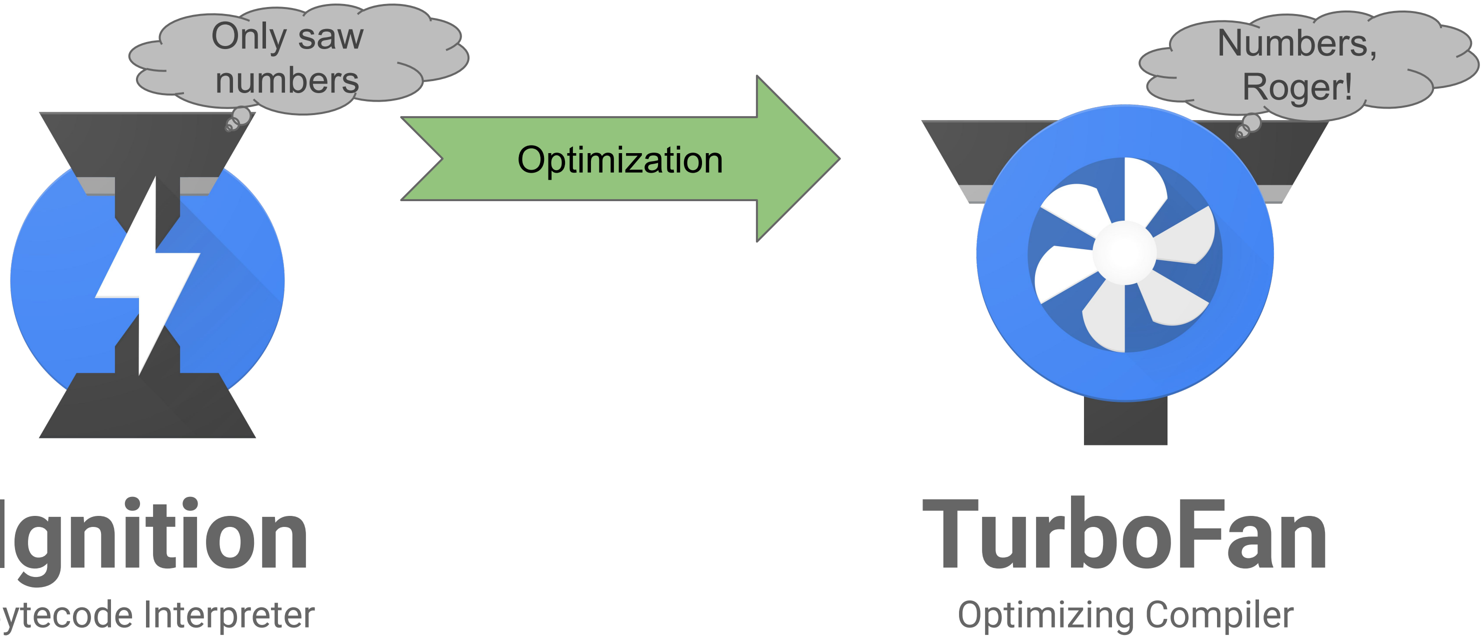
Bytecode Interpreter



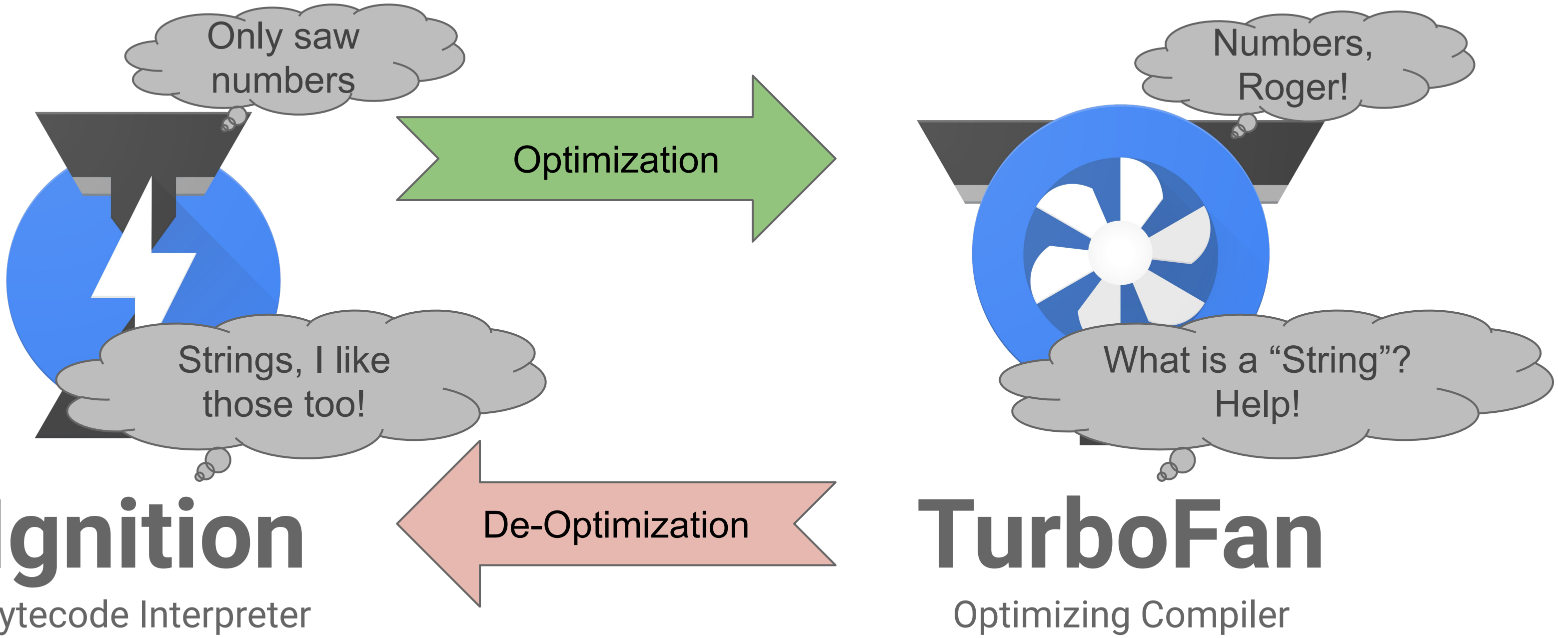
TurboFan

Optimizing Compiler

Type Feedback - Illustration



Type Feedback - Illustration



Optimization via Type-Feedback

- JavaScript is a Dynamically Typed Language
- Type-Feedback collected per Function
  - Functions with multiple call-site prone to turn polymorphic
- Speculative Optimization based on Feedback
  - Support “deoptimization” when speculative assumptions break
  - Ideal for functions used in stable & monomorphic fashion
  - Less ideal for functions that are highly polymorphic
- Let’s look at an example ...

Higher-Order Builtins

```
function foo(A) {
  return A.reduce(
    (b, x) => b + x, 0);
}

foo([1,2,3]);
// 6
```

Higher-Order Builtins

```
function foo(A) {
  return A.reduce(
    (b, x) => b + x, 0);
}

foo([1,2,3]);
// 6
```

```
function foo_Manual(A) {
  let sum = 0;
  const l = A.length;
  for (let i = 0; i < l; ++i) {
    sum += A[i];
  }
  return sum;
}
```

Higher-Order Builtins

```
function foo(A) {
  return A.reduce(
    (b, x) => b + x, 0);
}

foo([1,2,3]);
// 6
```

22.1.3.19 Array.prototype.reduce (*callbackfn* [, *initialValue*])

When the `reduce` method is called with one or two arguments, the following steps are taken:

1. Let *O* be ? ToObject(**this** value).
2. Let *len* be ? ToLength(? Get(*O*, `"length"`)).
3. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
4. If *len* is 0 and *initialValue* is not present, throw a **TypeError** exception.
5. Let *k* be 0.
6. Let *accumulator* be **undefined**.
7. If *initialValue* is present, then
   a. Set *accumulator* to *initialValue*.
8. Else *initialValue* is not present,
   a. Let *kPresent* be **false**.
   b. Repeat, while *kPresent* is **false** and *k* < *len*
      i. Let *Pk* be ! ToString(*k*).
      ii. Let *kPresent* be ? HasProperty(*O*, *Pk*).
      iii. If *kPresent* is **true**, then
         1. Set *accumulator* to ? Get(*O*, *Pk*).
      iv. Increase *k* by 1.
   c. If *kPresent* is **false**, throw a **TypeError** exception.
9. Repeat, while *k* < *len*
   a. Let *Pk* be ! ToString(*k*).
   b. Let *kPresent* be ? HasProperty(*O*, *Pk*).
   c. If *kPresent* is **true**, then
      i. Let *kValue* be ? Get(*O*, *Pk*).
      ii. Set *accumulator* to ? Call(*callbackfn*, **undefined**, « *accumulator*, *kValue*, *k*, *O* »).
   d. Increase *k* by 1.
10. Return *accumulator*.

Higher-Order Builtins

```
function foo(A) {
  return A.reduce(
    (b, x) => b + x, 0);
}

foo([1,2,3]);
// 6
```

22.1.3.19 Array.prototype.reduce (*callbackfn* [, *initialValue*])

When the `reduce` method is called with one or two arguments, the following steps are taken:

1. Let *O* be ? ToObject(**this** value).
2. Let *len* be ? ToLength(? Get(*O*, `"length"`)).
3. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
4. If *len* is 0 and *initialValue* is not present, throw a **TypeError** exception.
5. Let *k* be 0.
6. Let *accumulator* be **undefined**.
7. If *initialValue* is present, then
    a. Set *accumulator* to *initialValue*.
8. Else *initialValue* is not present,
    a. Let *kPresent* be **false**.
    b. Repeat, while *kPresent* is **false** and *k* < *len*
        i. Let *Pk* be ! ToString(*k*).
        ii. Let *kPresent* be ? HasProperty(*O*, *Pk*).
        iii. If *kPresent* is **true**, then
            1. Set *accumulator* to ? Get(*O*, *Pk*).
        iv. Increase *k* by 1.
    c. If *kPresent* is **false**, throw a **TypeError** exception.
9. Repeat, while *k* < *len*
    a. Let *Pk* be ! ToString(*k*).
    b. Let *kPresent* be ? HasProperty(*O*, *Pk*).
    c. If *kPresent* is **true**, then

NOTE 2 The `reduce` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

10. Return *accumulator*.

Higher-Order Builtins

```
function foo_Optimized(A) {
  const f = (b, x) => b + x;
  const len = A.length;
  if (typeof f !== "function")
    throw new TypeError();
  let b = 0;
  for (let i = 0; i < len; ++i) {
    if (i in A) { b = f(b, A[i]); }
  }
  return b;
}
```

Optimization Steps:

- Inlining of `Array#reduce`

Higher-Order Builtins

```
function foo_Optimized(A) {
  const f = (b, x) => b + x;
  const len = A.length;
  if (typeof f !== "function")
    throw new TypeError();
  let b = 0;
  for (let i = 0; i < len; ++i) {
    if (i in A) { b = f(b, A[i]); }
  }
  return b;
}
```

Optimization Steps:

- Inlining of `Array#reduce`
- Remove callability check

Higher-Order Builtins

```
function foo_Optimized(A) {
  const f = (b, x) => b + x;
  const len = A.length;


  let b = 0;
  for (let i = 0; i < len; ++i) {
    if (i in A) { b = b + A[i]; }
  }
  return b;
}
```

Optimization Steps:

- Inlining of `Array#reduce`
- Remove callability check
- Inlining of `f` into `foo`

Higher-Order Builtins

```
function foo_Optimized(A) {
  if (!IS_PACKED_SMI(A)) DEOPTIMIZE
  const len = A.length;
  let b = 0;
  for (let i = 0; i < len; ++i) {
    if (!IS_PACKED_SMI(A)) DEOPTIMIZE
    if (i in A) { b = b + A[i]; }
  }
  return b;
}
```

Optimization Steps:

- Inlining of `Array#reduce`
- Remove callability check
- Inlining of `f` into `foo`
- Assume array shape

Higher-Order Builtins

```
function foo_Optimized(A) {
  if (!IS_PACKED_SMI(A)) DEOPTIMIZE
  const len = A.length;
  let b = 0;
  for (let i = 0; i < len; ++i) {
    if (!IS_PACKED_SMI(A)) DEOPTIMIZE
    if (i in A) { b = b + A[i]; }
  }
  return b;
}
```

Optimization Steps:

- Inlining of `Array#reduce`
- Remove callability check
- Inlining of `f` into `foo`
- Assume array shape
- Remove “hole” check

Google

Higher-Order Builtins

```
function foo_Optimized(A) {
  if (!IS_PACKED_SMI(A)) DEOPTIMIZE
  const len = A.length;
  let b = 0;
  for (let i = 0; i < len; ++i) {
    if (!IS_PACKED_SMI(A)) DEOPTIMIZE
    b = b +SMI A[i];
  }
  return b;
}
```

Optimization Steps:

- Inlining of `Array#reduce`
- Remove callability check
- Inlining of `f` into `foo`
- Assume array shape
- Remove “hole” check
- Specialize arithmetic

Higher-Order Builtins

```
function foo_Optimized(A) {
  if (!IS_PACKED_SMI(A)) DEOPTIMIZE
  const len = A.length;
  let b = 0;
  for (let i = 0; i < len; ++i) {
    if (!IS_PACKED_SMI(A)) DEOPTIMIZE
    b = b +SMI A[i];
  }
  return b;
}
```

Optimization Steps:

- Inlining of `Array#reduce`
- Remove callability check
- Inlining of `f` into `foo`
- Assume array shape
- Remove “hole” check
- Specialize arithmetic
- Remove “shape” check

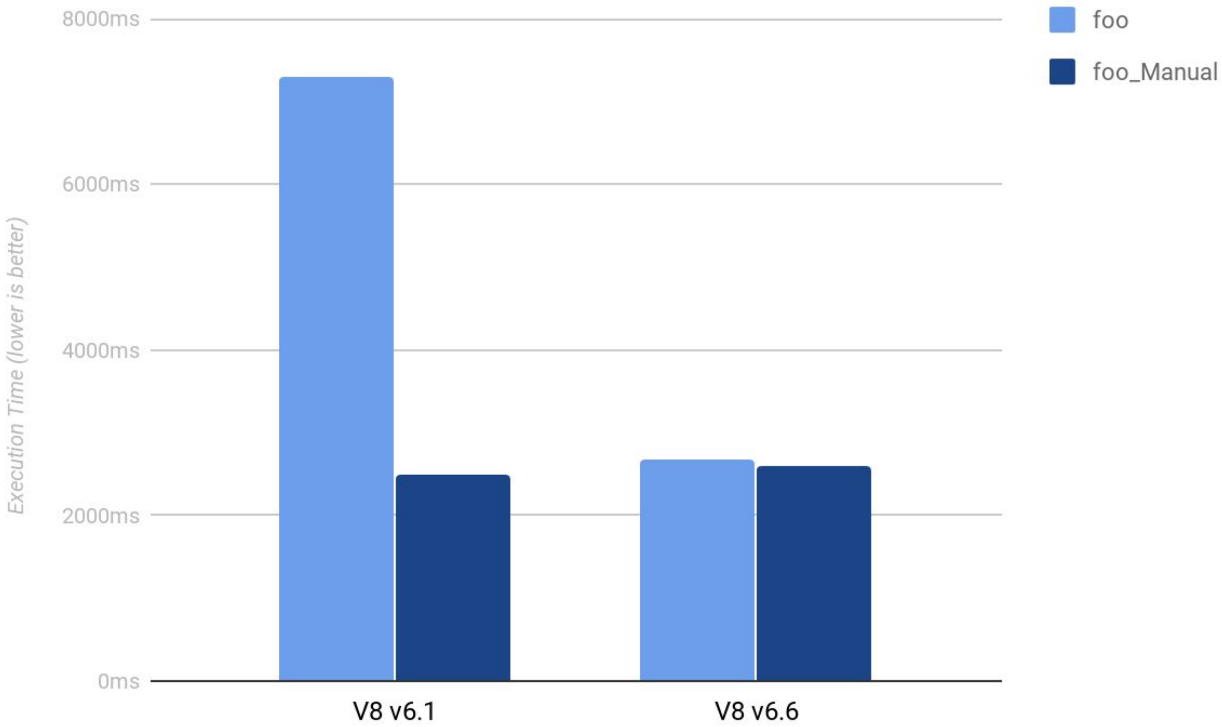
Higher-Order Builtins

```
function foo_Optimized(A) {
  if (!IS_PACKED_SMI(A)) DEOPTIMIZE
  const len = A.length;
  let b = 0;
  for (let i = 0; i < len; ++i) {
    b = b +SMI A[i];
  }
  return b;
}
```

Optimization Steps:

- Inlining of `Array#reduce`
- Remove callability check
- Inlining of `f` into `foo`
- Assume array shape
- Remove “hole” check
- Specialize arithmetic
- Remove “shape” check
- Profit!

Higher-Order Builtins - Measurement



Google

Higher-Order Builtins - Recap

Applicable higher-order builtins:

- Most Array functions:
  - Array#map, Array#filter, Array#every, Array#some, Array#reduce, …
- Some reflection methods:
  - Function#apply, Function#call

Optimistic assumptions:

- Based on feedback from warm-up
- Inlines known call targets
- Assumes “shape” unchanged
- Specializes arithmetic

Google

Open Problems/Questions

- Type-Feedback per Function vs. per Call-Site
  - Library functions often become polymorphic
  - Builtins optimized individually, other libraries don't benefit
  - Should feedback be split by call-site? Under what circumstances?
- Propagation of Call-Site Specific Feedback
  - Feedback of function argument types only?
  - Propagate into function body & override polymorphic feedback?
- Inlining Heuristics unaware of Feedback Propagation
  - Heuristics for inlining based on call frequence & function size
- Compile multiple Function Realizations
  - Dispatch according to argument types?
  - Polymorphic functions vs. generics/templates?

WebAssembly & TurboFan

Google

WebAssembly in a Nutshell

- Low-level bytecode designed to be fast to verify and compile
  - Explicit non-goal: fast to interpret
- Static types, argument counts, direct/indirect calls
  - No overloaded operations
- Unit of code is a module
  - Describes: globals, data, functions, imports, exports
  - Instantiation: New memory, New global mutable state

WebAssembly in a Nutshell - Module

```
header: 8 magic bytes
types: TypeDecl[]
imports: ImportDecl[]
funcdecl: FuncDecl[]
tables: TableDecl[]
memories: MemoryDecl[]
globals: GlobalVar[]
exports: ExportDecl[]
code: FunctionBody[]
data: Data[]
```

- Binary format
- Type declarations
- Imports:
  - Types
  - Functions
  - Globals
  - Memory
  - Tables
- Tables, memories
- Global variables
- Exports
- Function bodies (bytecode)

WebAssembly in a Nutshell - Bytecode

```
func: (i32, i32)->i32
  get_local[0]
  if[i32]
    get_local[0]
    i32.load_mem[8]
  else
    get_local[1]
    i32.load_mem[12]
  end
  i32.const[42]
  i32.add
end
```

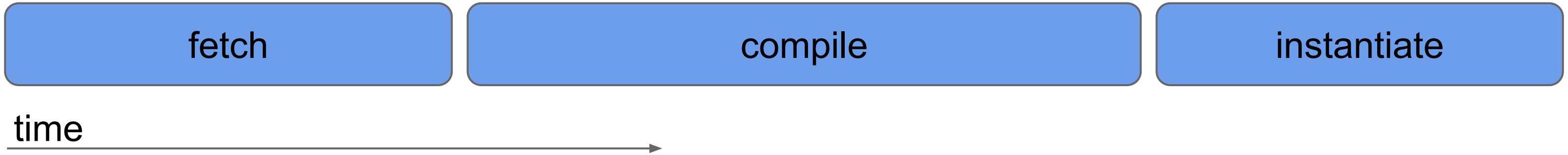
- Typed
- Stack machine
- Structured control flow
- One large flat memory
- Low-level memory operations
- Low-level arithmetic

WebAssembly Performance Goals

- WebAssembly performance goals:
  - Predictable: no lengthy warmup phase, no performance cliffs
  - Peak performance approaching native code (within ~20%)
- All major engine implementations reuse their respective JITs
  - V8: Liftoff AOT baseline full module + background with TurboFan
  - SpiderMonkey: Ion AOT baseline compiler full module + background Ion JIT
  - JSC: B3 compile on instantiate, full module
  - Edge: lazy compile to internal bytecode and dynamic tier-up with Chakra

WebAssembly Compilation

```
async function instantiateWasm(imports) {
  const response = await fetch('module.wasm');
  const buffer = await response.arrayBuffer();
  const module = await WebAssembly.compile(buffer);
  const instance = await WebAssembly.instantiate(module, imports);
  return instance;
}
```



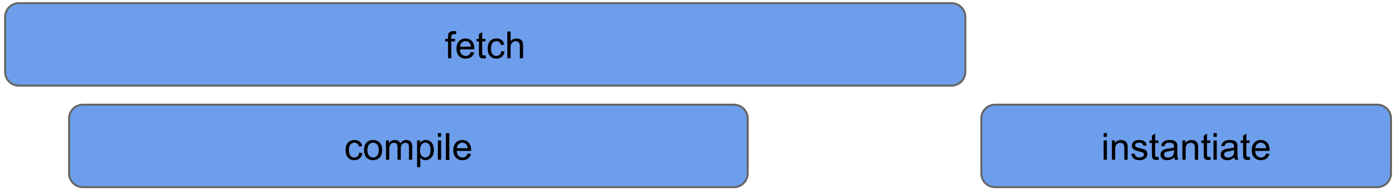
WebAssembly Compilation

```
async function instantiateWasm(imports) {
  const fetchPromise = fetch('module.wasm');
  const module = await WebAssembly.compileStreaming(fetchPromise);
  const instance = await WebAssembly.instantiate(module, imports);
  return instance;
}
```

https://developers.google.com/web/updates/2018/04/loading-wasm

WebAssembly Compilation

```
async function instantiateWasm(imports) {
  const fetchPromise = fetch('module.wasm');
  const module = await WebAssembly.compileStreaming(fetchPromise);
  const instance = await WebAssembly.instantiate(module, imports);
  return instance;
}
```



https://developers.google.com/web/updates/2018/04/loading-wasm

WebAssembly Compilation

```
async function instantiateWasm(imports) {
  const fetchPromise = fetch('module.wasm');
  const { instance } = await WebAssembly.instantiateStreaming(fetchPromise, imports);
  return instance;
}
```

https://developers.google.com/web/updates/2018/04/loading-wasm

A Tale of Two Compilers ... again

Liftoff
Baseline Compiler

TurboFan
Optimizing Compiler

A Tale of Two Compilers … again

- Fast startup speed
- Not all features
- Code not optimal

Liftoff

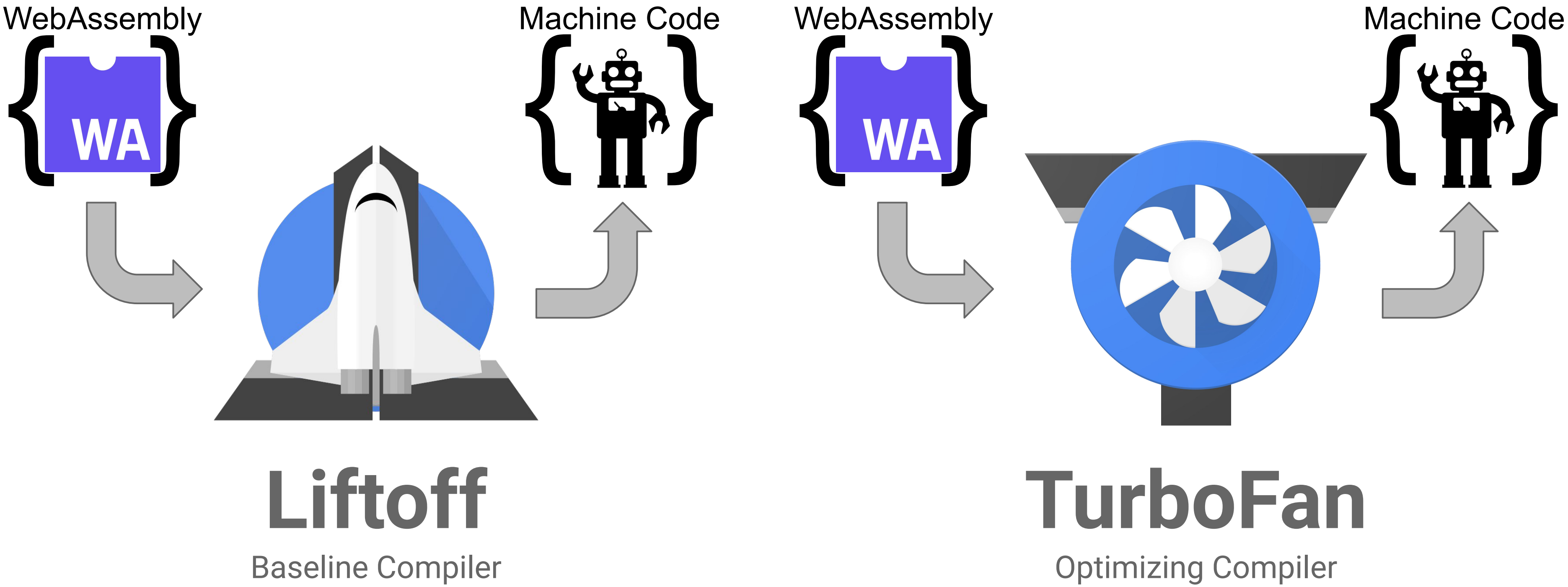
Baseline Compiler

- Supports all features
- Longer compilation
- Peak performance

TurboFan

Optimizing Compiler

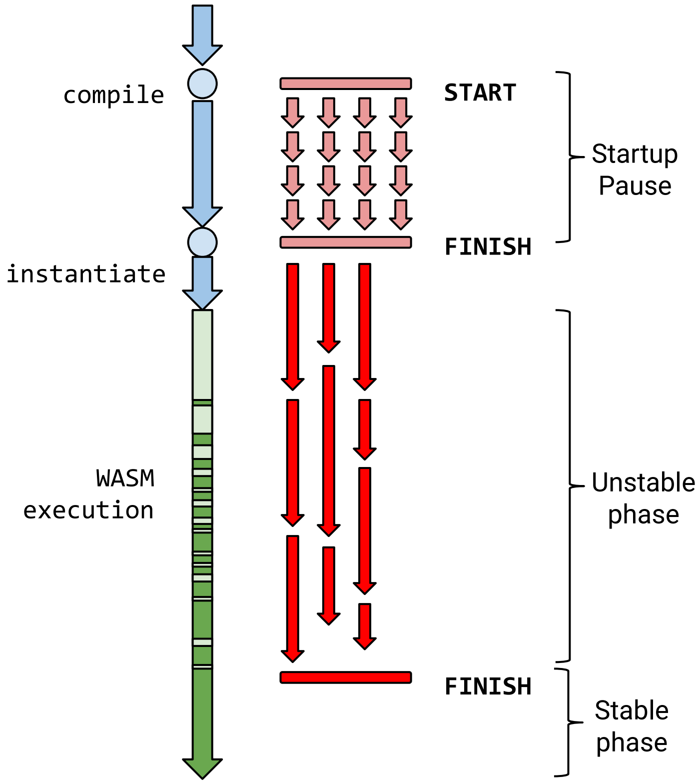
A Tale of Two Compilers ... again

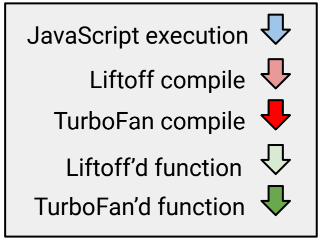


Tiering Strategies

- Tiering: balance compilation speed versus throughput
  - Liftoff is ~5x faster to compile, 1.5x slower to execute
  - Best startup requires Liftoff, peak performance requires TurboFan
  - (C++ interpreter is non-production, debugging only)
- Identified 4 different tiering strategies
  - Liftoff AOT, TurboFan background full compile
  - Liftoff AOT, dynamic tier-up
  - Liftoff lazy compile, dynamic tier-up
  - Liftoff background compile, dynamic tier-up

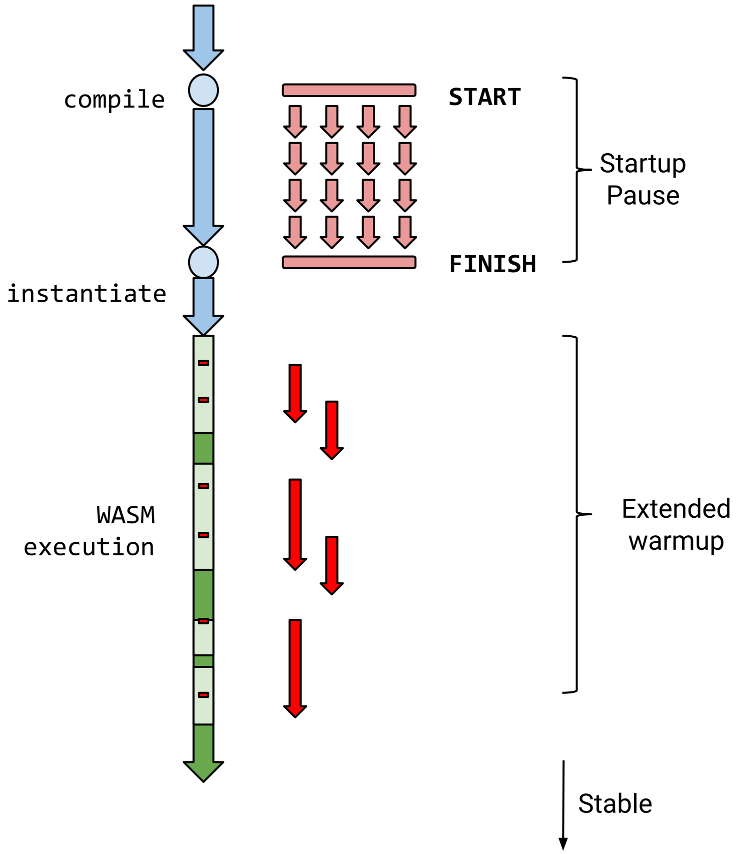
Strategy - Liftoff AOT + TurboFan Background

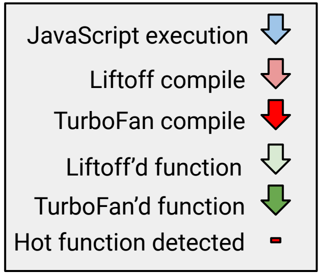




- **Advantages:**
  - Short startup pause
  - Smooth warmup: no jank
- **Disadvantages:**
  - Memory consumption
  - Double compile of everything

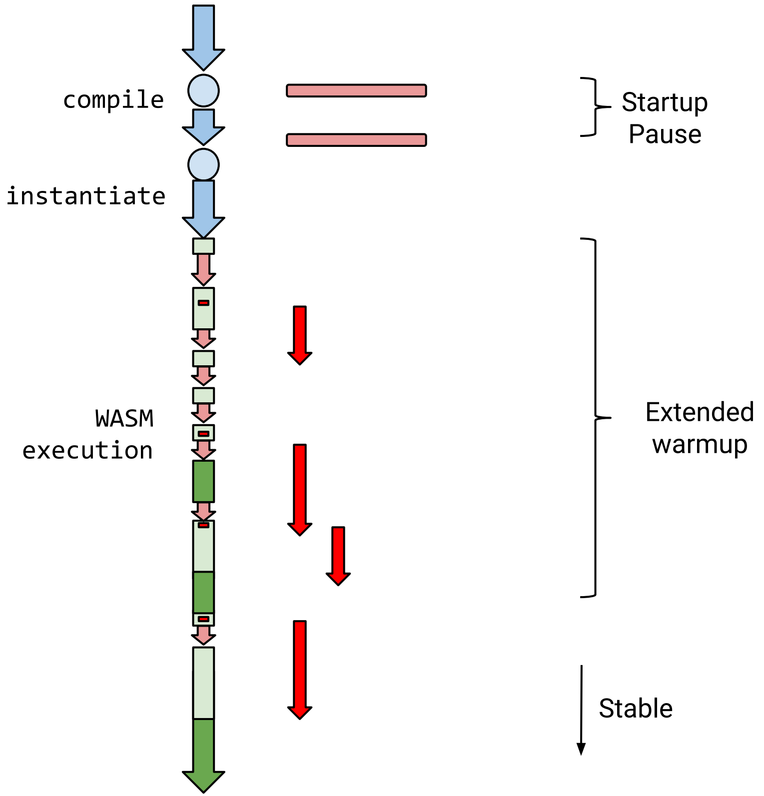
Strategy - Liftoff AOT + Dynamic Tier-Up

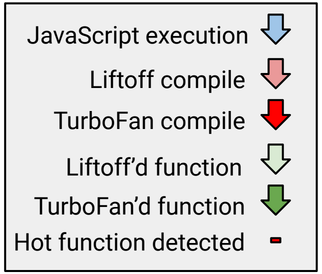




- **Advantages:**
  - Short startup pause
  - Smooth warmup: no jank
  - Less overall compile work
- **Disadvantages:**
  - Longer warmup

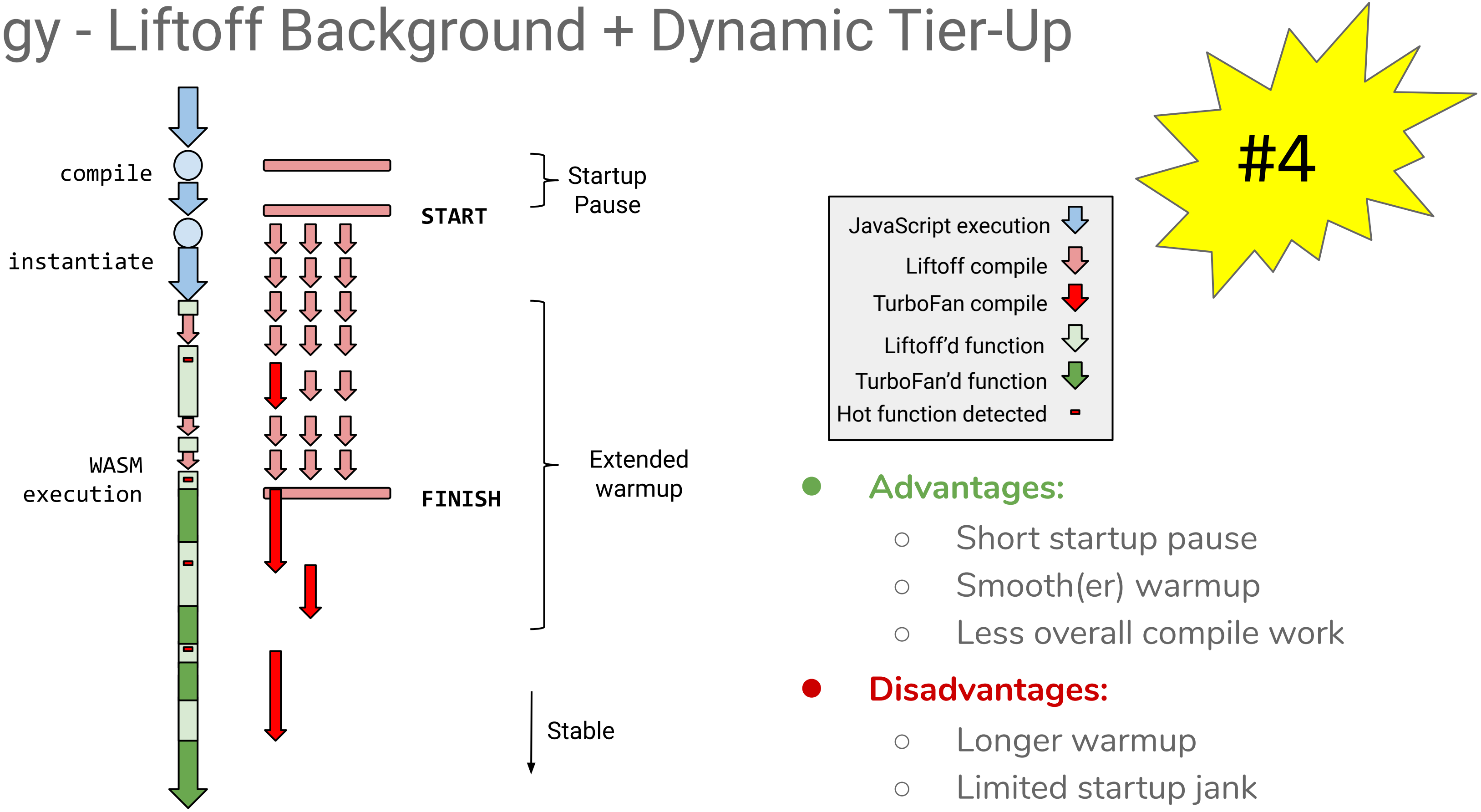
Strategy - Liftoff Lazy + Dynamic Tier-Up





- **Advantages:**
  - Shortest startup pause
  - Minimal overall compile work
- **Disadvantages:**
  - Janky startup
  - Longer warmup

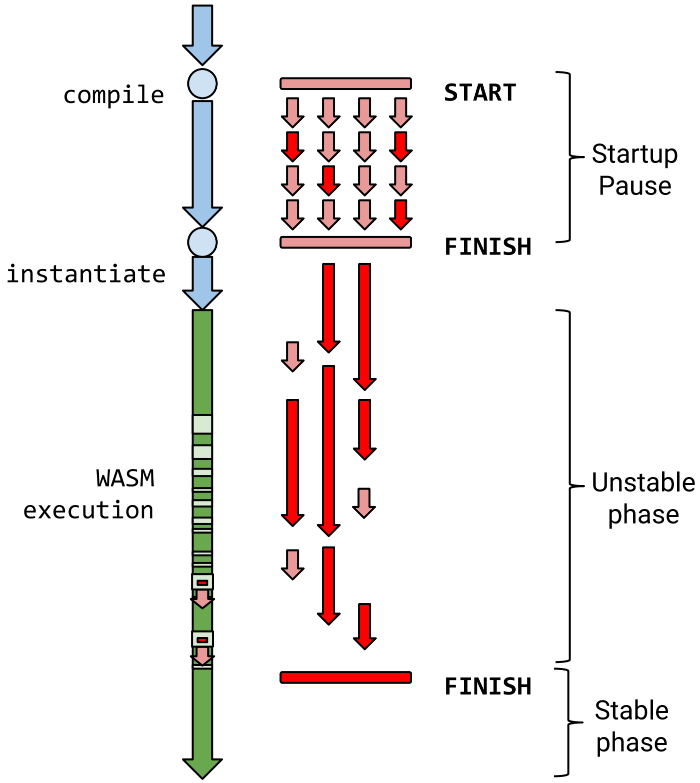
Strategy - Liftoff Background + Dynamic Tier-Up

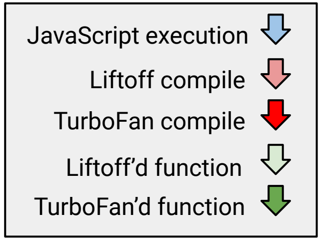


- **Advantages:**
  - Short startup pause
  - Smooth(er) warmup
  - Less overall compile work
- **Disadvantages:**
  - Longer warmup
  - Limited startup jank

Google

Strategy - Hinted AOT + Background + Lazy





- **Advantages:**
  - Shorter startup pause
  - Smooth warmup: no jank
  - Reach peak perf fast
- **Disadvantages:**
  - Imprecise static heuristic

Open Problems/Questions

- Choosing an Optimal Tiering Strategy
  - Optimal strategy might be different per device (mobile vs. desktop)
- Guide Strategy by Hints in the Module
  - Prototype: Section in the module header with per-function tiering hints
  - How to generalize to different devices (mobile vs. desktop)?
- Unit of Compilation: Function
  - Inlining in the WebAssembly Generator vs. Engine
  - Many different producers of WebAssembly expected
  - Can the engine benefit from inlining?

Google

Michael Starzinger
mstarzinger@google.com

Google

Thanks!

Image Attributions

- “Robot”: https://commons.wikimedia.org/wiki/File:Noun_project_1248.svg
- “Gears”: https://commons.wikimedia.org/wiki/File:Gears.png
- “Gift”: https://commons.wikimedia.org/wiki/File:Gift_font_awesome.svg
- “JavaScript”: https://commons.wikimedia.org/wiki/File:JavaScript-logo.png
- “WebAssembly”: https://commons.wikimedia.org/wiki/File:Web_Assembly_Logo.svg