Google

A Little on V8 and WebAssembly

Ben L. Titzer

Virtual Machines Summer School 2016

2016-06-01

Agenda

- What makes JavaScript unique and challenging?
- What makes V8 unique and challenging?
- What the heck is WebAssembly and why?

We all love JavaScript

What makes JavaScript unique and interesting?

- JavaScript is the language of the Web
- Scripting language: programs presented in source form
- “Classically slow” language
- Prototype-based object model
- Functional features with closures
- Untyped: variables and properties do not have types, values do
- A smattering of oddball features
    - Weird scoping rules
    - `eval`
    - `with` scopes
    - Proxies
    - Rest parameters
    - Default parameters
    - Generators
    - Undetectables
    - Holey arrays
    - Arguments object
    - ...

Challenge: programs presented in source form

- Parsing has to be fast
- Source code is slower for machines to parse
  - Source code parser: 1-10MB/s
  - Binary format like bytecode: 100MB/s
- New language features all the time
  - All features supported by all virtual machines

Challenge: prototype-based object model

```
var x = new SubClass(“mine”, 100);

function BaseClass(name) {
  this.name = name;
}
function SubClass(name, data) {
  BaseClass.call(this, name);
  this.data = data;
}
BaseClass.prototype.print = function() {
  print(this.name);
}

SubClass.prototype.__proto__ = BaseClass.prototype;
```

Challenge: prototype-based object model

```
var x = new SubClass(“mine”, 100);

function BaseClass(name) {
  this.name = name;
}
function SubClass(name, data) {
  BaseClass.call(this, name);
  this.data = data;
}
BaseClass.prototype.print = function() {
  print(this.name);
}

SubClass.prototype.__proto__ = BaseClass.prototype;
```

- Objects instantiated by “new Function()” syntax
- Methods installed on the “prototype” of an object
- Prototypes chain together to emulate inheritance

Challenge: prototype-based object model

```
var x = new SubClass(“mine”, 100);

function BaseClass(name) {
  this.name = name;
}
function SubClass(name, data) {
  BaseClass.call(this, name);
  this.data = data;
}
BaseClass.prototype.print = function() {
  print(this.name);
}

SubClass.prototype.__proto__ = BaseClass.prototype;
```

- Objects instantiated by “new Function()” syntax
- Methods installed on the “prototype” of an object
- Prototypes chain together to emulate inheritance

Challenge: prototype-based object model

```
var x = new SubClass(“mine”, 100);

function BaseClass(name) {
  this.name = name;
}
function SubClass(name, data) {
  BaseClass.call(this, name);
  this.data = data;
}
BaseClass.prototype.print = function() {
  print(this.name);
}

SubClass.prototype.__proto__ = BaseClass.prototype;
```

- Objects instantiated by “new Function()” syntax
- Methods installed on the “prototype” of an object
- Prototypes chain together to emulate inheritance

Challenge: functional programming with closures

```
function Counter(name) {
  var count = 0;
  return {
    inc: function() { count++; },
    get: function() { return count; },
    print: function() { print(name + “:” + count); }
  }
}

var x = new Counter();

var before = x.get();
x.inc();
x.print();
```

- Closures over local variables, even mutable locals
- Object literals allow grouping multiple closures into a “mini-object”

Challenge: functional programming with closures

```
function Counter(name) {
  var count = 0;
  return {
    inc: function() { count++; },
    get: function() { return count; },
    print: function() { print(name + “:” + count); }
  }
}

var x = new Counter();

var before = x.get();
x.inc();
x.print();
```

- Closures over locals, even mutable locals
- Object literals allow grouping multiple closures into a “mini-object”

Challenge: untyped variables and operations

```
function add(a, b) {
     return a + b;
}
add(1, 2);
add(“foo”, 1);
add(1, “foo”);
add({foo: “”}, 1);
add(“hello”, {toString: () => “me”});
add(1.01, 3.03);
```

- Variables, parameters, properties, and expressions do not have types
- Operators are overloaded for different types of values

Challenge: untyped variables and operations

```
function add(a, b) {
     return a + b;
}
add(1, 2);
add(“foo”, 1);
add(1, “foo”);
add({foo: “”}, 1);
add(“hello”, {toString: () => “me”});
add(1.01, 3.03);
```

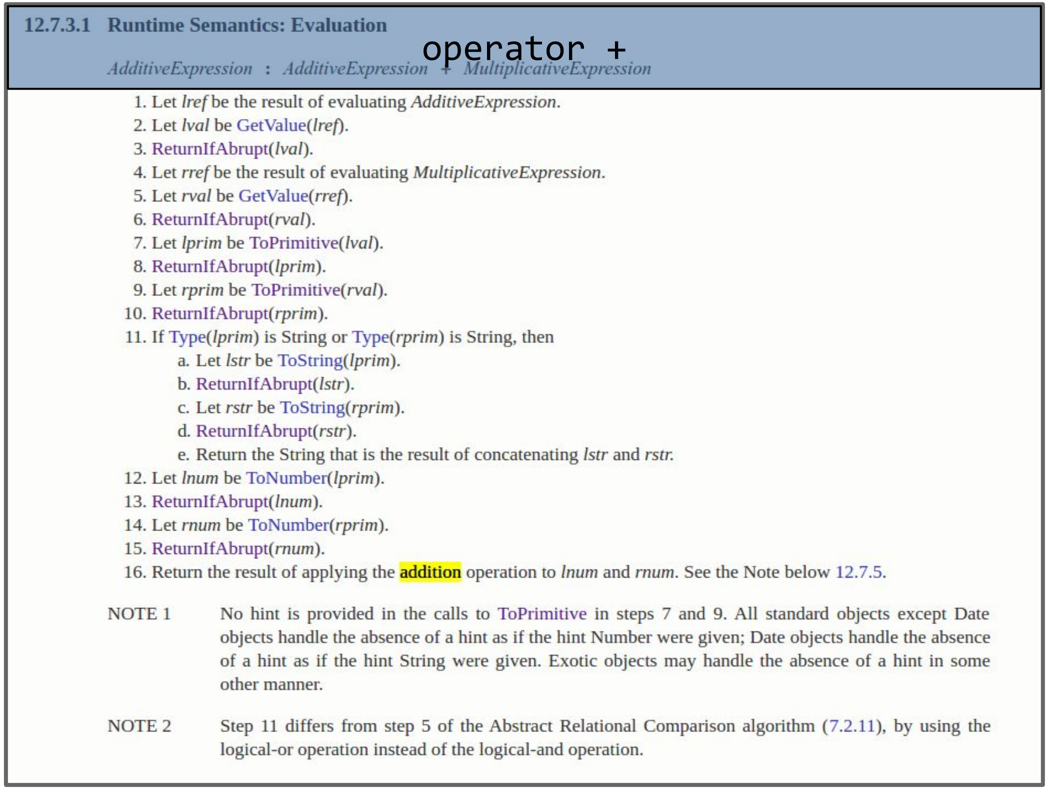
- Variables, parameters, properties, and expressions do not have types
- Operators are overloaded for different types of values

Challenge: untyped variables and operations

```
function add(a, b) {
     return a + b;
}
add(1, 2);
add(“foo”, 1);
add(1, “foo”);
add({foo: “”}, 1);
add(“hello”, {toString: () => “me”});
add(1.01, 3.03);
```

- Variables, parameters, properties, and expressions do not have types
- Operators are overloaded for different types of values

Glance at Semantics: +



operator +

12.7.3.1 Runtime Semantics: Evaluation

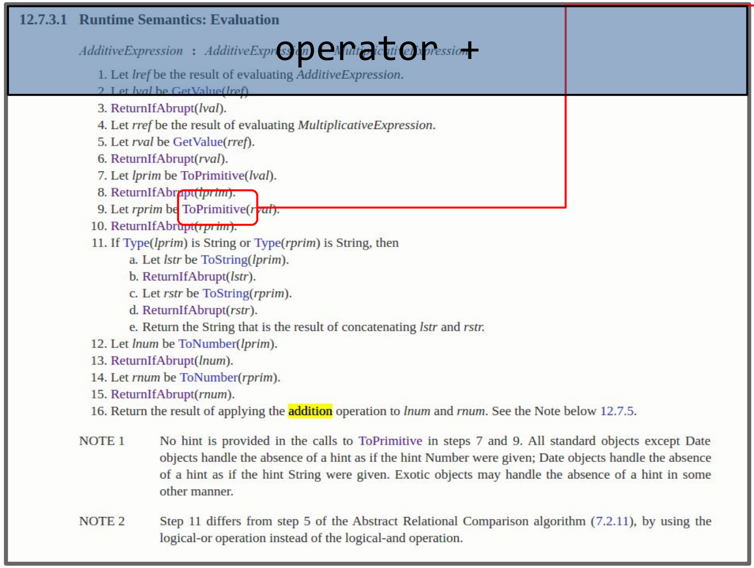
AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

```
1. Let lref be the result of evaluating AdditiveExpression.
2. Let lval be GetValue(lref).
3. ReturnIfAbrupt(lval).
4. Let rref be the result of evaluating MultiplicativeExpression.
5. Let rval be GetValue(rref).
6. ReturnIfAbrupt(rval).
7. Let lprim be ToPrimitive(lval).
8. ReturnIfAbrupt(lprim).
9. Let rprim be ToPrimitive(rval).
10. ReturnIfAbrupt(rprim).
11. If Type(lprim) is String or Type(rprim) is String, then
    a. Let lstr be ToString(lprim).
    b. ReturnIfAbrupt(lstr).
    c. Let rstr be ToString(rprim).
    d. ReturnIfAbrupt(rstr).
    e. Return the String that is the result of concatenating lstr and rstr.
12. Let lnum be ToNumber(lprim).
13. ReturnIfAbrupt(lnum).
14. Let rnum be ToNumber(rprim).
15. ReturnIfAbrupt(rnum).
16. Return the result of applying the addition operation to lnum and rnum. See the Note below 12.7.5.
```

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.

Glance at Semantics: +



operator +

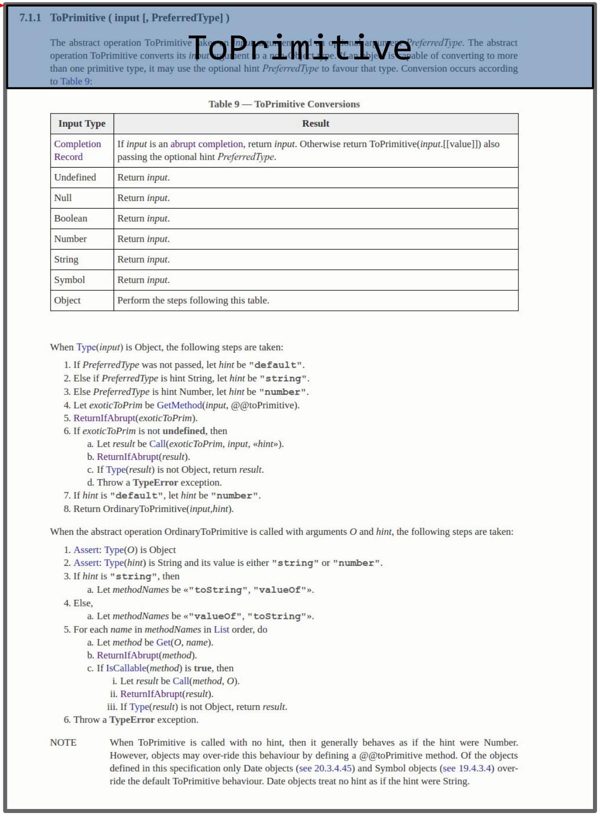
12.7.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

```
1. Let lref be the result of evaluating AdditiveExpression.
2. Let lval be GetValue(lref).
3. ReturnIfAbrupt(lval).
4. Let rref be the result of evaluating MultiplicativeExpression.
5. Let rval be GetValue(rref).
6. ReturnIfAbrupt(rval).
7. Let lprim be ToPrimitive(lval).
8. ReturnIfAbrupt(lprim).
9. Let rprim be ToPrimitive(rval).
10. ReturnIfAbrupt(rprim).
11. If Type(lprim) is String or Type(rprim) is String, then
    a. Let lstr be ToString(lprim).
    b. ReturnIfAbrupt(lstr).
    c. Let rstr be ToString(rprim).
    d. ReturnIfAbrupt(rstr).
    e. Return the String that is the result of concatenating lstr and rstr.
12. Let lnum be ToNumber(lprim).
13. ReturnIfAbrupt(lnum).
14. Let rnum be ToNumber(rprim).
15. ReturnIfAbrupt(rnum).
16. Return the result of applying the addition operation to lnum and rnum. See the Note below 12.7.5.
```

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.



ToPrimitive

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 Table 9:

Table 9 — ToPrimitive Conversions

Input Type	Result
Completion Record	If *input* is an abrupt completion, return *input*. Otherwise return ToPrimitive(*input*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *input*.
Null	Return *input*.
Boolean	Return *input*.
Number	Return *input*.
String	Return *input*.
Symbol	Return *input*.
Object	Perform the steps following this table.

When Type(*input*) is Object, the following steps are taken:

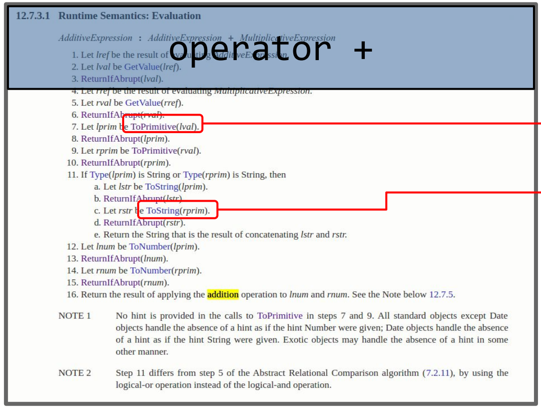
```
1. If PreferredType was not passed, let hint be "default".
2. Else if PreferredType is hint String, let hint be "string".
3. Else PreferredType is hint Number, let hint be "number".
4. Let exoticToPrim be GetMethod(input, @@toPrimitive).
5. ReturnIfAbrupt(exoticToPrim).
6. If exoticToPrim is not undefined, then
    a. Let result be Call(exoticToPrim, input, «hint»).
    b. ReturnIfAbrupt(result).
    c. If Type(result) is not Object, return result.
    d. Throw a TypeError exception.
7. If hint is "default", let hint be "number".
8. Return OrdinaryToPrimitive(input,hint).
```

When the abstract operation OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

```
1. Assert: Type(O) is Object
2. Assert: Type(hint) is String and its value is either "string" or "number".
3. If hint is "string", then
    a. Let methodNames be «"toString", "valueOf"».
4. Else,
    a. Let methodNames be «"valueOf", "toString"».
5. For each name in methodNames in List order, do
    a. Let method be Get(O, name).
    b. ReturnIfAbrupt(method).
    c. If IsCallable(method) is true, then
        i. Let result be Call(method, O).
        ii. ReturnIfAbrupt(result).
        iii. If Type(result) is not Object, return result.
6. Throw a TypeError exception.
```

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@toPrimitive method. Of the objects defined in this specification only Date objects (see 20.3.4.45) and Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

Glance at Semantics: +



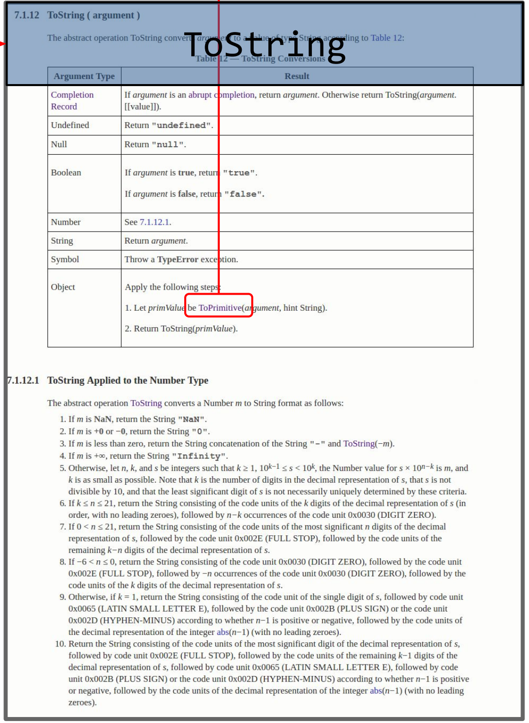
12.7.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

```
1. Let lref be the result of evaluating AdditiveExpression.
2. Let lval be GetValue(lref).
3. ReturnIfAbrupt(lval).
4. Let rref be the result of evaluating MultiplicativeExpression.
5. Let rval be GetValue(rref).
6. ReturnIfAbrupt(rval).
7. Let lprim be ToPrimitive(lval).
8. ReturnIfAbrupt(lprim).
9. Let rprim be ToPrimitive(rval).
10. ReturnIfAbrupt(rprim).
11. If Type(lprim) is String or Type(rprim) is String, then
    a. Let lstr be ToString(lprim).
    b. ReturnIfAbrupt(lstr).
    c. Let rstr be ToString(rprim).
    d. ReturnIfAbrupt(rstr).
    e. Return the String that is the result of concatenating lstr and rstr.
12. Let lnum be ToNumber(lprim).
13. ReturnIfAbrupt(lnum).
14. Let rnum be ToNumber(rprim).
15. ReturnIfAbrupt(rnum).
16. Return the result of applying the addition operation to lnum and rnum. See the Note below 12.7.5.
```

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.



7.1.12 ToString (argument)

The abstract operation ToString converts *argument* to a value of type String according to Table 12:

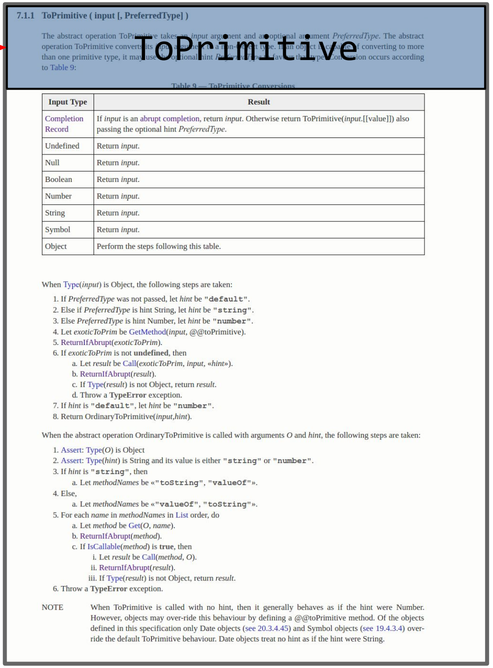
Table 12 — ToString Conversions

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToString(*argument*.[[value]]).
Undefined	Return `"undefined"`.
Null	Return `"null"`.
Boolean	If *argument* is **true**, return `"true"`. If *argument* is **false**, return `"false"`.
Number	See 7.1.12.1.
String	Return *argument*.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(*argument*, hint String). 2. Return ToString(*primValue*).

7.1.12.1 ToString Applied to the Number Type

The abstract operation ToString converts a Number *m* to String format as follows:

```
1. If $m$ is NaN, return the String "NaN".
2. If $m$ is +0 or −0, return the String "0".
3. If $m$ is less than zero, return the String concatenation of the String "-" and ToString($-m$).
4. If $m$ is $+\infty$, return the String "Infinity".
5. Otherwise, let $n$, $k$, and $s$ be integers such that $k \geq 1$, $10^{k-1} \leq s < 10^k$, the Number value for $s \times 10^{n-k}$ is $m$, and $k$ is as small as possible. Note that $k$ is the number of digits in the decimal representation of $s$, that $s$ is not divisible by 10, and that the least significant digit of $s$ is not necessarily uniquely determined by these criteria.
6. If $k \leq n \leq 21$, return the String consisting of the code units of the $k$ digits of the decimal representation of $s$ (in order, with no leading zeroes), followed by $n-k$ occurrences of the code unit 0x0030 (DIGIT ZERO).
7. If $0 < n \leq 21$, return the String consisting of the code units of the most significant $n$ digits of the decimal representation of $s$, followed by the code unit 0x002E (FULL STOP), followed by the code units of the remaining $k-n$ digits of the decimal representation of $s$.
8. If $-6 < n \leq 0$, return the String consisting of the code unit 0x0030 (DIGIT ZERO), followed by the code unit 0x002E (FULL STOP), followed by $-n$ occurrences of the code unit 0x0030 (DIGIT ZERO), followed by the code units of the $k$ digits of the decimal representation of $s$.
9. Otherwise, if $k = 1$, return the String consisting of the code unit of the single digit of $s$, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether $n-1$ is positive or negative, followed by the code units of the decimal representation of the integer abs($n-1$) (with no leading zeroes).
10. Return the String consisting of the code units of the most significant digit of the decimal representation of $s$, followed by code unit 0x002E (FULL STOP), followed by the code units of the remaining $k-1$ digits of the decimal representation of $s$, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether $n-1$ is positive or negative, followed by the code units of the decimal representation of the integer abs($n-1$) (with no leading zeroes).
```



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 Table 9:

Table 9 — ToPrimitive Conversions

Input Type	Result
Completion Record	If *input* is an abrupt completion, return *input*. Otherwise return ToPrimitive(*input*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *input*.
Null	Return *input*.
Boolean	Return *input*.
Number	Return *input*.
String	Return *input*.
Symbol	Return *input*.
Object	Perform the steps following this table.

When Type(*input*) is Object, the following steps are taken:

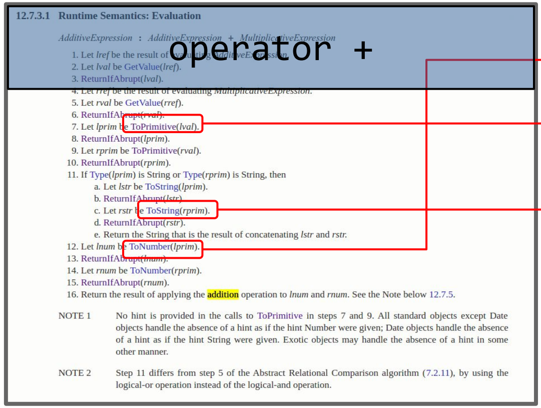
```
1. If PreferredType was not passed, let hint be "default".
2. Else if PreferredType is hint String, let hint be "string".
3. Else PreferredType is hint Number, let hint be "number".
4. Let exoticToPrim be GetMethod(input, @@toPrimitive).
5. ReturnIfAbrupt(exoticToPrim).
6. If exoticToPrim is not undefined, then
    a. Let result be Call(exoticToPrim, input, «hint»).
    b. ReturnIfAbrupt(result).
    c. If Type(result) is not Object, return result.
    d. Throw a TypeError exception.
7. If hint is "default", let hint be "number".
8. Return OrdinaryToPrimitive(input,hint).
```

When the abstract operation OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

```
1. Assert: Type(O) is Object
2. Assert: Type(hint) is String and its value is either "string" or "number".
3. If hint is "string", then
    a. Let methodNames be «"toString", "valueOf"».
4. Else,
    a. Let methodNames be «"valueOf", "toString"».
5. For each name in methodNames in List order, do
    a. Let method be Get(O, name).
    b. ReturnIfAbrupt(method).
    c. If IsCallable(method) is true, then
        i. Let result be Call(method, O).
        ii. ReturnIfAbrupt(result).
        iii. If Type(result) is not Object, return result.
6. Throw a TypeError exception.
```

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@toPrimitive method. Of the objects defined in this specification only Date objects (see 20.3.4.45) and Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

Glance at Semantics: +



operator +

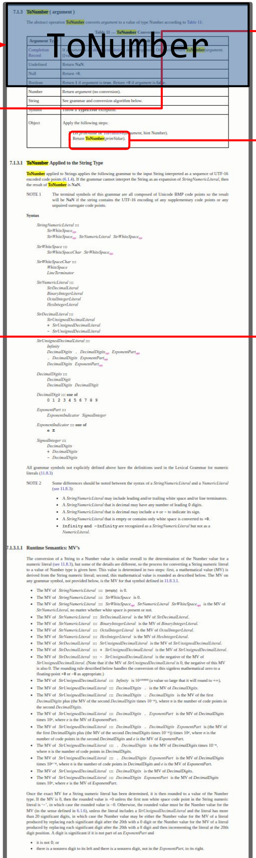
12.7.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

```
1. Let lref be the result of evaluating AdditiveExpression.
2. Let lval be GetValue(lref).
3. ReturnIfAbrupt(lval).
4. Let rref be the result of evaluating MultiplicativeExpression.
5. Let rval be GetValue(rref).
6. ReturnIfAbrupt(rval).
7. Let lprim be ToPrimitive(lval).
8. ReturnIfAbrupt(lprim).
9. Let rprim be ToPrimitive(rval).
10. ReturnIfAbrupt(rprim).
11. If Type(lprim) is String or Type(rprim) is String, then
    a. Let lstr be ToString(lprim).
    b. ReturnIfAbrupt(lstr).
    c. Let rstr be ToString(rprim).
    d. ReturnIfAbrupt(rstr).
    e. Return the String that is the result of concatenating lstr and rstr.
12. Let lnum be ToNumber(lprim).
13. ReturnIfAbrupt(lnum).
14. Let rnum be ToNumber(rprim).
15. ReturnIfAbrupt(rnum).
16. Return the result of applying the addition operation to lnum and rnum. See the Note below 12.7.5.
```

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.



ToNumber

7.1.3 ToNumber (argument)

The abstract operation ToNumber converts *argument* to a value of type Number according to Table 11:

Table 11 — ToNumber Conversions

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToNumber(*argument*.[[value]]).
Undefined	Return NaN.
Null	Return +0.
Boolean	Return 1 if *argument* is **true**. Return +0 if *argument* is **false**.
Number	Return *argument* (no conversion).
String	See grammar and conversion algorithm below.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(*argument*, hint Number). 2. Return ToNumber(*primValue*).

7.1.3.1 ToNumber Applied to the String Type

ToNumber applied to Strings applies the following grammar to the input String interpreted as a sequence of UTF-16 encoded code points (6.1.4). If the grammar cannot interpret the String as an expansion of *StringNumericLiteral*, then the result of ToNumber is NaN.

NOTE 1 The terminal symbols of this grammar are all composed of Unicode BMP code points so the result will be NaN if the string contains the UTF-16 encoding of any supplementary code points or any unpaired surrogate code points.

Syntax

```
StringNumericLiteral :::
    StrWhiteSpace_opt
    StrWhiteSpace_opt StrNumericLiteral StrWhiteSpace_opt

StrWhiteSpace :::
    StrWhiteSpaceChar StrWhiteSpace_opt

StrWhiteSpaceChar :::
    WhiteSpace
    LineTerminator

StrNumericLiteral :::
    StrDecimalLiteral
    BinaryIntegerLiteral
    OctalIntegerLiteral
    HexIntegerLiteral

StrDecimalLiteral :::
    StrUnsignedDecimalLiteral
    + StrUnsignedDecimalLiteral
    - StrUnsignedDecimalLiteral

StrUnsignedDecimalLiteral :::
    Infinity
    DecimalDigits . DecimalDigits_opt ExponentPart_opt
    . DecimalDigits ExponentPart_opt
    DecimalDigits ExponentPart_opt

DecimalDigits :::
    DecimalDigit
    DecimalDigits DecimalDigit

DecimalDigit ::: one of
    0 1 2 3 4 5 6 7 8 9

ExponentPart :::
    ExponentIndicator SignedInteger

ExponentIndicator ::: one of
    e E

SignedInteger :::
    DecimalDigits
    + DecimalDigits
    - DecimalDigits
```

All grammar symbols not explicitly defined above have the definitions used in the Lexical Grammar for numeric literals (11.8.3)

NOTE 2 Some differences should be noted between the syntax of a *StringNumericLiteral* and a *NumericLiteral* (see 11.8.3):

- A *StringNumericLiteral* may include leading and/or trailing white space and/or line terminators.
- A *StringNumericLiteral* that is decimal may have any number of leading 0 digits.
- A *StringNumericLiteral* that is decimal may include a + or - to indicate its sign.
- A *StringNumericLiteral* that is empty or contains only white space is converted to +0.
- `Infinity` and `-Infinity` are recognized as a *StringNumericLiteral* but not as a *NumericLiteral*.

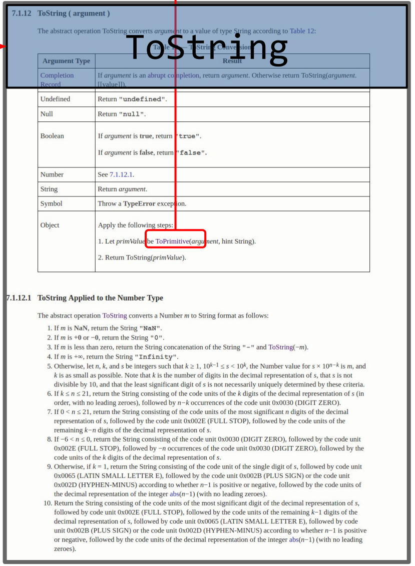
7.1.3.1.1 Runtime Semantics: MV's

The conversion of a String to a Number value is similar overall to the determination of the Number value for a numeric literal (see 11.8.3), but some of the details are different, so the process for converting a String numeric literal to a value of Number type is given here. This value is determined in two steps: first, a mathematical value (MV) is derived from the String numeric literal; second, this mathematical value is rounded as described below. The MV on any grammar symbol, not provided below, is the MV for that symbol defined in 11.8.3.1.

- The MV of *StringNumericLiteral* ::: [empty] is 0.
- The MV of *StringNumericLiteral* ::: *StrWhiteSpace* is 0.
- The MV of *StringNumericLiteral* ::: *StrWhiteSpace*$_{opt}$ *StrNumericLiteral* *StrWhiteSpace*$_{opt}$ is the MV of *StrNumericLiteral*, no matter whether white space is present or not.
- The MV of *StrNumericLiteral* ::: *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* ::: *BinaryIntegerLiteral* is the MV of *BinaryIntegerLiteral*.
- The MV of *StrNumericLiteral* ::: *OctalIntegerLiteral* is the MV of *OctalIntegerLiteral*.
- The MV of *StrNumericLiteral* ::: *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *StrDecimalLiteral* ::: *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StrDecimalLiteral* ::: + *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StrDecimalLiteral* ::: - *StrUnsignedDecimalLiteral* is the negative of the MV of *StrUnsignedDecimalLiteral*. (Note that if the MV of *StrUnsignedDecimalLiteral* is 0, the negative of this MV is also 0. The rounding rule described below handles the conversion of this signless mathematical zero to a floating-point +0 or −0 as appropriate.)
- The MV of *StrUnsignedDecimalLiteral* ::: *Infinity* is 10^{10000} (a value so large that it will round to +∞).
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* . is the MV of *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* . *DecimalDigits* is the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times 10^{-n}), where *n* is the number of code points in the second *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* . *ExponentPart* is the MV of *DecimalDigits* times 10^{e}, where *e* is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* . *DecimalDigits* *ExponentPart* is (the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times 10^{-n})) times 10^{e}, where *n* is the number of code points in the second *DecimalDigits* and *e* is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral* ::: . *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n}, where *n* is the number of code points in *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* ::: . *DecimalDigits* *ExponentPart* is the MV of *DecimalDigits* times 10^{e-n}, where *n* is the number of code points in *DecimalDigits* and *e* is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* *ExponentPart* is the MV of *DecimalDigits* times 10^{e}, where *e* is the MV of *ExponentPart*.

Once the exact MV for a String numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0 unless the first non white space code point in the String numeric literal is '-', in which case the rounded value is −0. Otherwise, the rounded value must be the Number value for the MV (in the sense defined in 6.1.6), unless the literal includes a *StrUnsignedDecimalLiteral* and the literal has more than 20 significant digits, in which case the Number value may be either the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit or the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit and then incrementing the literal at the 20th digit position. A digit is significant if it is not part of an *ExponentPart* and

- it is not 0; or
- there is a nonzero digit to its left and there is a nonzero digit, not in the *ExponentPart*, to its right.



ToString

7.1.12 ToString (argument)

The abstract operation ToString converts *argument* to a value of type String according to Table 12:

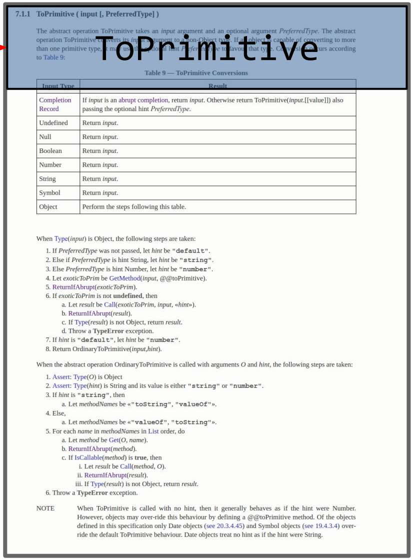
Table 12 — ToString Conversions

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToString(*argument*.[[value]]).
Undefined	Return `"undefined"`.
Null	Return `"null"`.
Boolean	If *argument* is **true**, return `"true"`. If *argument* is **false**, return `"false"`.
Number	See 7.1.12.1.
String	Return *argument*.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(*argument*, hint String). 2. Return ToString(*primValue*).

7.1.12.1 ToString Applied to the Number Type

The abstract operation ToString converts a Number *m* to String format as follows:

```
1. If m is NaN, return the String "NaN".
2. If m is +0 or −0, return the String "0".
3. If m is less than zero, return the String concatenation of the String "-" and ToString(−m).
4. If m is +∞, return the String "Infinity".
5. Otherwise, let n, k, and s be integers such that $k ≥ 1$, $10^{k-1} ≤ s < 10^k$, the Number value for $s × 10^{n-k}$ is m, and k is as small as possible. Note that k is the number of digits in the decimal representation of s, that s is not divisible by 10, and that the least significant digit of s is not necessarily uniquely determined by these criteria.
6. If $k ≤ n ≤ 21$, return the String consisting of the code units of the k digits of the decimal representation of s (in order, with no leading zeroes), followed by n−k occurrences of the code unit 0x0030 (DIGIT ZERO).
7. If $0 < n ≤ 21$, return the String consisting of the code units of the most significant n digits of the decimal representation of s, followed by the code unit 0x002E (FULL STOP), followed by the code units of the remaining k−n digits of the decimal representation of s.
8. If $−6 < n ≤ 0$, return the String consisting of the code unit 0x0030 (DIGIT ZERO), followed by the code unit 0x002E (FULL STOP), followed by −n occurrences of the code unit 0x0030 (DIGIT ZERO), followed by the code units of the k digits of the decimal representation of s.
9. Otherwise, if k = 1, return the String consisting of the code unit of the single digit of s, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether n−1 is positive or negative, followed by the code units of the decimal representation of the integer abs(n−1) (with no leading zeroes).
10. Return the String consisting of the code units of the most significant digit of the decimal representation of s, followed by code unit 0x002E (FULL STOP), followed by the code units of the remaining k−1 digits of the decimal representation of s, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether n−1 is positive or negative, followed by the code units of the decimal representation of the integer abs(n−1) (with no leading zeroes).
```



ToPrimitive

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 Table 9:

Table 9 — ToPrimitive Conversions

Input Type	Result
Completion Record	If *input* is an abrupt completion, return *input*. Otherwise return ToPrimitive(*input*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *input*.
Null	Return *input*.
Boolean	Return *input*.
Number	Return *input*.
String	Return *input*.
Symbol	Return *input*.
Object	Perform the steps following this table.

When Type(*input*) is Object, the following steps are taken:

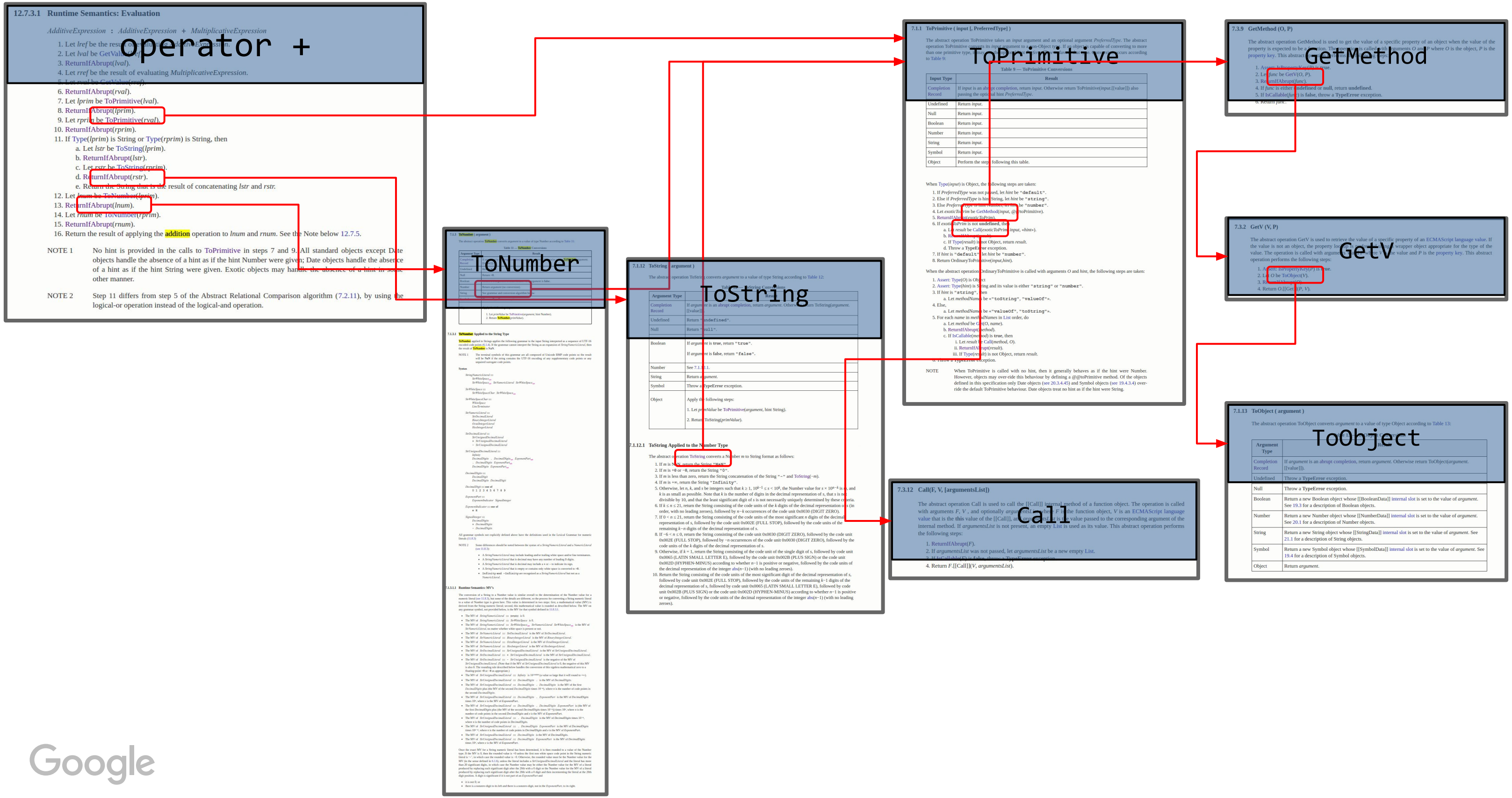
```
1. If PreferredType was not passed, let hint be "default".
2. Else if PreferredType is hint String, let hint be "string".
3. Else PreferredType is hint Number, let hint be "number".
4. Let exoticToPrim be GetMethod(input, @@toPrimitive).
5. ReturnIfAbrupt(exoticToPrim).
6. If exoticToPrim is not undefined, then
    a. Let result be Call(exoticToPrim, input, «hint»).
    b. ReturnIfAbrupt(result).
    c. If Type(result) is not Object, return result.
    d. Throw a TypeError exception.
7. If hint is "default", let hint be "number".
8. Return OrdinaryToPrimitive(input,hint).
```

When the abstract operation OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

```
1. Assert: Type(O) is Object
2. Assert: Type(hint) is String and its value is either "string" or "number".
3. If hint is "string", then
    a. Let methodNames be «"toString", "valueOf"».
4. Else,
    a. Let methodNames be «"valueOf", "toString"».
5. For each name in methodNames in List order, do
    a. Let method be Get(O, name).
    b. ReturnIfAbrupt(method).
    c. If IsCallable(method) is true, then
        i. Let result be Call(method, O).
        ii. ReturnIfAbrupt(result).
        iii. If Type(result) is not Object, return result.
6. Throw a TypeError exception.
```

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@toPrimitive method. Of the objects defined in this specification only Date objects (see 20.3.4.45) and Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

Glance at Semantics: +



12.7.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lprim* be ToPrimitive(*lval*).
8. ReturnIfAbrupt(*lprim*).
9. Let *rprim* be ToPrimitive(*rval*).
10. ReturnIfAbrupt(*rprim*).
11. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Let *lstr* be ToString(*lprim*).
    b. ReturnIfAbrupt(*lstr*).
    c. Let *rstr* be ToString(*rprim*).
    d. ReturnIfAbrupt(*rstr*).
    e. Return the String that is the result of concatenating *lstr* and *rstr*.
12. Let *lnum* be ToNumber(*lprim*).
13. ReturnIfAbrupt(*lnum*).
14. Let *rnum* be ToNumber(*rprim*).
15. ReturnIfAbrupt(*rnum*).
16. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.7.5.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.

7.1.3 ToNumber (argument)

The abstract operation ToNumber converts *argument* to a value of type Number according to Table 11:

Table 11 — ToNumber Conversions

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToNumber(*argument*.[[value]]).
Undefined	Return NaN.
Null	Return +0.
Boolean	[illegible] *argument* is false.
Number	Return *argument* (no conversion).
String	See grammar and conversion algorithm below.
Symbol	[illegible]
Object	[illegible] 1. Let *primValue* be ToPrimitive(*argument*, hint Number). 2. Return ToNumber(*primValue*).

7.1.3.1 ToNumber Applied to the String Type

ToNumber applied to Strings applies the following grammar to the input String interpreted as a sequence of UTF-16 encoded code points (6.1.4). If the grammar cannot interpret the String as an expansion of *StringNumericLiteral*, then the result of ToNumber is NaN.

NOTE 1 The terminal symbols of this grammar are all composed of Unicode BMP code points so the result will be NaN if the string contains the UTF-16 encoding of any supplementary code points or any unpaired surrogate code points.

Syntax

StringNumericLiteral :::
  StrWhiteSpace$_{opt}$
  StrWhiteSpace$_{opt}$ *StrNumericLiteral* *StrWhiteSpace*$_{opt}$

StrWhiteSpace :::
  StrWhiteSpaceChar *StrWhiteSpace*$_{opt}$

StrWhiteSpaceChar :::
  WhiteSpace
  LineTerminator

StrNumericLiteral :::
  StrDecimalLiteral
  BinaryIntegerLiteral
  OctalIntegerLiteral
  HexIntegerLiteral

[illegible]

7.1.12 ToString (argument)

The abstract operation ToString converts *argument* to a value of type String according to Table 12:

Table 12 — ToString Conversions

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToString(*argument*.[[value]]).
Undefined	Return `"undefined"`.
Null	Return `"null"`.
Boolean	If *argument* is **true**, return `"true"`. If *argument* is **false**, return `"false"`.
Number	See 7.1.12.1.
String	Return *argument*.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(*argument*, hint String). 2. Return ToString(*primValue*).

7.1.12.1 ToString Applied to the Number Type

The abstract operation ToString converts a Number *m* to String format as follows:

1. If *m* is NaN, return the String `"NaN"`.
2. If *m* is +0 or −0, return the String `"0"`.
3. If *m* is less than zero, return the String concatenation of the String `"-"` and ToString(−*m*).
4. If *m* is +∞, return the String `"Infinity"`.
5. Otherwise, let *n*, *k*, and *s* be integers such that $k \ge 1$, $10^{k-1} \le s < 10^k$, the Number value for $s \times 10^{n-k}$ is *m*, and *k* is as small as possible. Note that *k* is the number of digits in the decimal representation of *s*, that *s* is not divisible by 10, and that the least significant digit of *s* is not necessarily uniquely determined by these criteria.
6. If $k \le n \le 21$, return the String consisting of the code units of the *k* digits of the decimal representation of *s* (in order, with no leading zeroes), followed by *n*−*k* occurrences of the code unit 0x0030 (DIGIT ZERO).
7. If $0 < n \le 21$, return the String consisting of the code units of the most significant *n* digits of the decimal representation of *s*, followed by the code unit 0x002E (FULL STOP), followed by the code units of the remaining *k*−*n* digits of the decimal representation of *s*.
8. If $-6 < n \le 0$, return the String consisting of the code unit 0x0030 (DIGIT ZERO), followed by the code unit 0x002E (FULL STOP), followed by −*n* occurrences of the code unit 0x0030 (DIGIT ZERO), followed by the code units of the *k* digits of the decimal representation of *s*.
9. Otherwise, if *k* = 1, return the String consisting of the code unit of the single digit of *s*, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether *n*−1 is positive or negative, followed by the code units of the decimal representation of the integer abs(*n*−1) (with no leading zeroes).
10. Return the String consisting of the code units of the most significant digit of the decimal representation of *s*, followed by code unit 0x002E (FULL STOP), followed by the code units of the remaining *k*−1 digits of the decimal representation of *s*, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether *n*−1 is positive or negative, followed by the code units of the decimal representation of the integer abs(*n*−1) (with no leading zeroes).

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 Table 9:

Table 9 — ToPrimitive Conversions

Input Type	Result
Completion Record	If *input* is an abrupt completion, return *input*. Otherwise return ToPrimitive(*input*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *input*.
Null	Return *input*.
Boolean	Return *input*.
Number	Return *input*.
String	Return *input*.
Symbol	Return *input*.
Object	Perform the steps following this table.

When Type(*input*) is Object, the following steps are taken:

1. If *PreferredType* was not passed, let *hint* be `"default"`.
2. Else if *PreferredType* is hint String, let *hint* be `"string"`.
3. Else *PreferredType* is hint Number, let *hint* be `"number"`.
4. Let *exoticToPrim* be GetMethod(*input*, @@toPrimitive).
5. ReturnIfAbrupt(*exoticToPrim*).
6. If *exoticToPrim* is not **undefined**, then
    a. Let *result* be Call(*exoticToPrim*, *input*, «*hint*»).
    b. ReturnIfAbrupt(*result*).
    c. If Type(*result*) is not Object, return *result*.
    d. Throw a **TypeError** exception.
7. If *hint* is `"default"`, let *hint* be `"number"`.
8. Return OrdinaryToPrimitive(*input*,*hint*).

When the abstract operation OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

1. Assert: Type(*O*) is Object
2. Assert: Type(*hint*) is String and its value is either `"string"` or `"number"`.
3. If *hint* is `"string"`, then
    a. Let *methodNames* be «`"toString"`, `"valueOf"`».
4. Else,
    a. Let *methodNames* be «`"valueOf"`, `"toString"`».
5. For each *name* in *methodNames* in List order, do
    a. Let *method* be Get(*O*, *name*).
    b. ReturnIfAbrupt(*method*).
    c. If IsCallable(*method*) is **true**, then
        i. Let *result* be Call(*method*, *O*).
        ii. ReturnIfAbrupt(*result*).
        iii. If Type(*result*) is not Object, return *result*.
6. Throw a **TypeError** exception.

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@toPrimitive method. Of the objects defined in this specification only Date objects (see 20.3.4.45) and Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

7.3.9 GetMethod (O, P)

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

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

7.3.2 GetV (V, P)

The abstract operation GetV is used to retrieve the value of a specific property of an ECMAScript language value. If the value is not an object, the property lookup is performed using a wrapper object appropriate for the type of the value. The operation is called with arguments *V* and *P* where *V* is the value and *P* is the property key. This abstract operation performs the following steps:

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *O* be ToObject(*V*).
3. ReturnIfAbrupt(*O*).
4. Return *O*.[[Get]](*P*, *V*).

7.1.13 ToObject (argument)

The abstract operation ToObject converts *argument* to a value of type Object according to Table 13:

Table 13 — ToObject Conversions

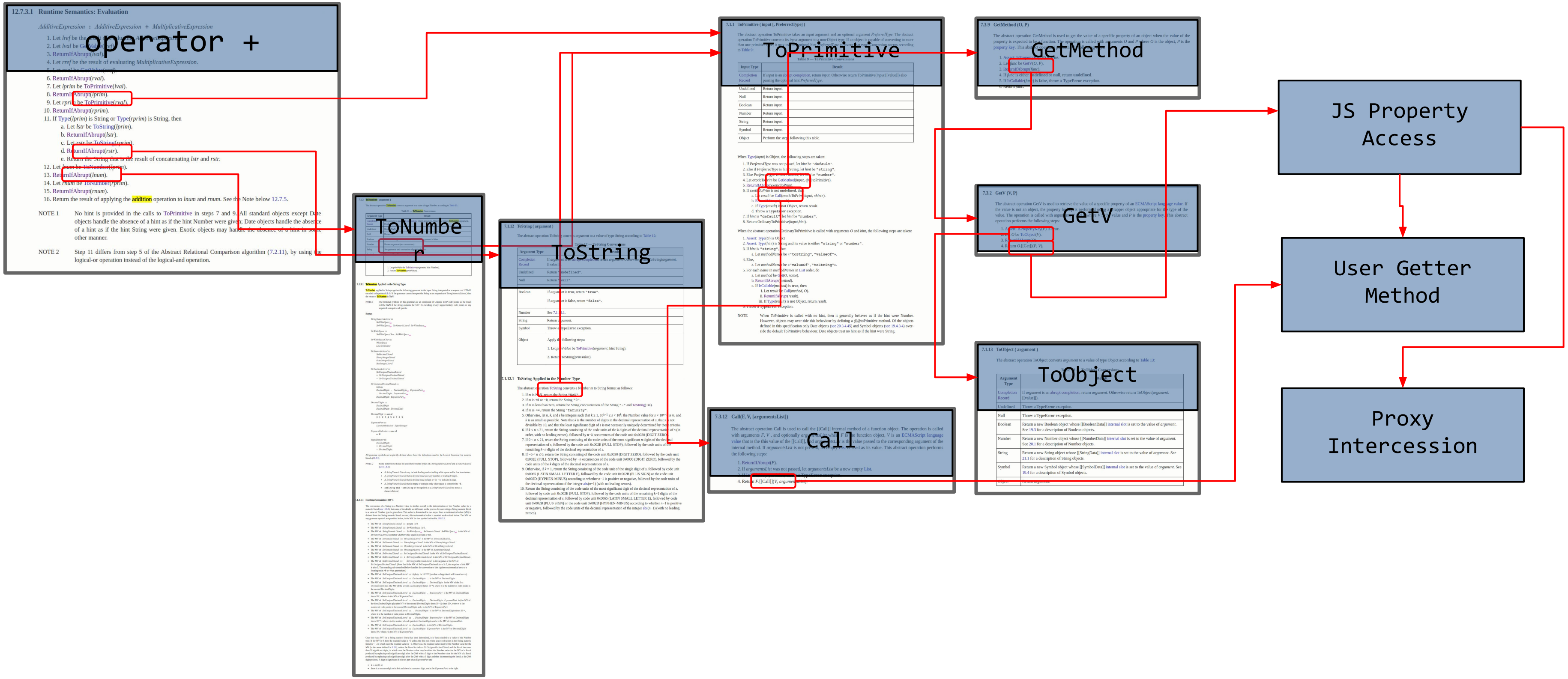
Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToObject(*argument*.[[value]]).
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return a new Boolean object whose [[BooleanData]] internal slot is set to the value of *argument*. See 19.3 for a description of Boolean objects.
Number	Return a new Number object whose [[NumberData]] internal slot is set to the value of *argument*. See 20.1 for a description of Number objects.
String	Return a new String object whose [[StringData]] internal slot is set to the value of *argument*. See 21.1 for a description of String objects.
Symbol	Return a new Symbol object whose [[SymbolData]] internal slot is set to the value of *argument*. See 19.4 for a description of Symbol objects.
Object	Return *argument*.

7.3.12 Call(F, V, [argumentsList])

The abstract operation Call is used to call the [[Call]] internal method of a function object. The operation is called with arguments *F*, *V* , and optionally *argumentsList* where *F* is the function object, *V* is an ECMAScript language value that is the this value of the [[Call]], and *argumentsList* is the value passed to the corresponding argument of the internal method. If *argumentsList* is not present, an empty List is used as its value. This abstract operation performs the following steps:

1. ReturnIfAbrupt(*F*).
2. If *argumentsList* was not passed, let *argumentsList* be a new empty List.
3. If IsCallable(*F*) is false, throw a TypeError exception.
4. Return *F*.[[Call]](*V*, *argumentsList*).

Glance at Semantics: +



12.7.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

```
1. Let lref be the result of evaluating AdditiveExpression.
2. Let lval be GetValue(lref).
3. ReturnIfAbrupt(lval).
4. Let rref be the result of evaluating MultiplicativeExpression.
5. Let rval be GetValue(rref).
6. ReturnIfAbrupt(rval).
7. Let lprim be ToPrimitive(lval).
8. ReturnIfAbrupt(lprim).
9. Let rprim be ToPrimitive(rval).
10. ReturnIfAbrupt(rprim).
11. If Type(lprim) is String or Type(rprim) is String, then
    a. Let lstr be ToString(lprim).
    b. ReturnIfAbrupt(lstr).
    c. Let rstr be ToString(rprim).
    d. ReturnIfAbrupt(rstr).
    e. Return the String that is the result of concatenating lstr and rstr.
12. Let lnum be ToNumber(lprim).
13. ReturnIfAbrupt(lnum).
14. Let rnum be ToNumber(rprim).
15. ReturnIfAbrupt(rnum).
16. Return the result of applying the addition operation to lnum and rnum. See the Note below 12.7.5.
```

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.

7.1.12 ToString (argument)

The abstract operation ToString converts *argument* to a value of type String according to Table 12:

Table 12 — ToString Conversions

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToString(*argument*.[[value]]).
Undefined	Return "undefined".
Null	Return "null".
Boolean	If *argument* is true, return "true". If *argument* is false, return "false".
Number	See 7.1.12.1.
String	Return *argument*.
Symbol	Throw a TypeError exception.
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(*argument*, hint String). 2. Return ToString(*primValue*).

7.1.12.1 ToString Applied to the Number Type

The abstract operation ToString converts a Number *m* to String format as follows:

1. If *m* is NaN, return the String "NaN".
2. If *m* is +0 or −0, return the String "0".
3. If *m* is less than zero, return the String concatenation of the String "-" and ToString(−*m*).
4. If *m* is +∞, return the String "Infinity".
5. Otherwise, let *n*, *k*, and *s* be integers such that $k \geq 1$, $10^{k-1} \leq s < 10^k$, the Number value for $s \times 10^{n-k}$ is *m*, and *k* is as small as possible. Note that *k* is the number of digits in the decimal representation of *s*, that *s* is not divisible by 10, and that the least significant digit of *s* is not necessarily uniquely determined by these criteria.
6. If $k \leq n \leq 21$, return the String consisting of the code units of the *k* digits of the decimal representation of *s* (in order, with no leading zeroes), followed by *n*−*k* occurrences of the code unit 0x0030 (DIGIT ZERO).
7. If $0 < n \leq 21$, return the String consisting of the code units of the most significant *n* digits of the decimal representation of *s*, followed by the code unit 0x002E (FULL STOP), followed by the code units of the remaining *k*−*n* digits of the decimal representation of *s*.
8. If $-6 < n \leq 0$, return the String consisting of the code unit 0x0030 (DIGIT ZERO), followed by the code unit 0x002E (FULL STOP), followed by −*n* occurrences of the code unit 0x0030 (DIGIT ZERO), followed by the code units of the *k* digits of the decimal representation of *s*.
9. Otherwise, if *k* = 1, return the String consisting of the code unit of the single digit of *s*, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by the code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether *n*−1 is positive or negative, followed by the code units of the decimal representation of the integer abs(*n*−1) (with no leading zeroes).
10. Return the String consisting of the code units of the most significant digit of the decimal representation of *s*, followed by code unit 0x002E (FULL STOP), followed by the code units of the remaining *k*−1 digits of the decimal representation of *s*, followed by code unit 0x0065 (LATIN SMALL LETTER E), followed by code unit 0x002B (PLUS SIGN) or the code unit 0x002D (HYPHEN-MINUS) according to whether *n*−1 is positive or negative, followed by the code units of the decimal representation of the integer abs(*n*−1) (with no leading zeroes).

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 Table 9:

Table 9 — ToPrimitive Conversions

Input Type	Result
Completion Record	If *input* is an abrupt completion, return *input*. Otherwise return ToPrimitive(*input*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *input*.
Null	Return *input*.
Boolean	Return *input*.
Number	Return *input*.
String	Return *input*.
Symbol	Return *input*.
Object	Perform the steps following this table.

When Type(*input*) is Object, the following steps are taken:

```
1. If PreferredType was not passed, let hint be "default".
2. Else if PreferredType is hint String, let hint be "string".
3. Else PreferredType is hint Number, let hint be "number".
4. Let exoticToPrim be GetMethod(input, @@toPrimitive).
5. ReturnIfAbrupt(exoticToPrim).
6. If exoticToPrim is not undefined, then
    a. Let result be Call(exoticToPrim, input, «hint»).
    b. ReturnIfAbrupt(result).
    c. If Type(result) is not Object, return result.
    d. Throw a TypeError exception.
7. If hint is "default", let hint be "number".
8. Return OrdinaryToPrimitive(input,hint).
```

When the abstract operation OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

```
1. Assert: Type(O) is Object
2. Assert: Type(hint) is String and its value is either "string" or "number".
3. If hint is "string", then
    a. Let methodNames be «"toString", "valueOf"».
4. Else,
    a. Let methodNames be «"valueOf", "toString"».
5. For each name in methodNames in List order, do
    a. Let method be Get(O, name).
    b. ReturnIfAbrupt(method).
    c. If IsCallable(method) is true, then
        i. Let result be Call(method, O).
        ii. ReturnIfAbrupt(result).
        iii. If Type(result) is not Object, return result.
6. Throw a TypeError exception.
```

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@toPrimitive method. Of the objects defined in this specification only Date objects (see 20.3.4.45) and Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

7.3.12 Call(F, V, [argumentsList])

The abstract operation Call is used to call the [[Call]] internal method of a function object. The operation is called with arguments *F*, *V* , and optionally *argumentsList* where *F* is the function object, *V* is an ECMAScript language value that is the this value of the [[Call]], and *argumentsList* is the value passed to the corresponding argument of the internal method. If *argumentsList* is not present, an empty List is used as its value. This abstract operation performs the following steps:

```
1. ReturnIfAbrupt(F).
2. If argumentsList was not passed, let argumentsList be a new empty List.
3. If IsCallable(F) is false, throw a TypeError exception.
4. Return F.[[Call]](V, argumentsList).
```

7.3.9 GetMethod (O, P)

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

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

7.3.2 GetV (V, P)

The abstract operation GetV is used to retrieve the value of a specific property of an ECMAScript language value. If the value is not an object, the property lookup is performed using a wrapper object appropriate for the type of the value. The operation is called with arguments *V* and *P* where *V* is the value and *P* is the property key. This abstract operation performs the following steps:

```
1. Assert: IsPropertyKey(P) is true.
2. Let O be ToObject(V).
3. ReturnIfAbrupt(O).
4. Return O.[[Get]](P, V).
```

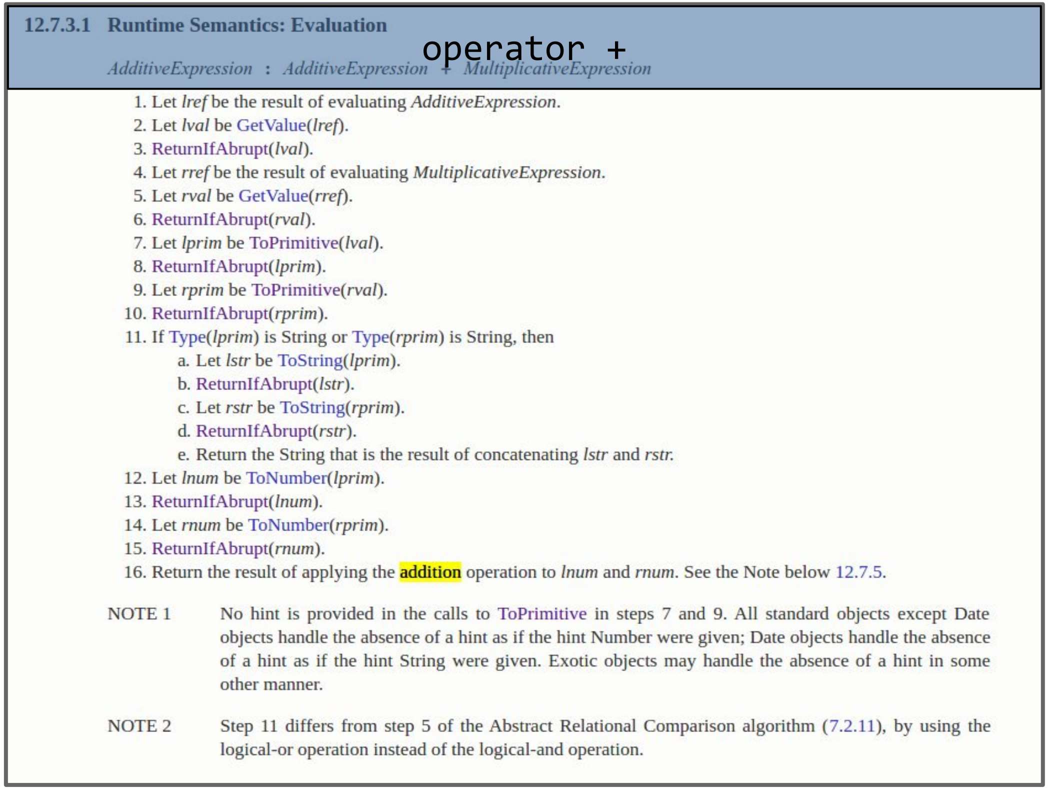
7.1.13 ToObject (argument)

The abstract operation ToObject converts *argument* to a value of type Object according to Table 13:

Table 13 — ToObject Conversions

Argument Type	Result
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToObject(*argument*.[[value]]).
Undefined	Throw a TypeError exception.
Null	Throw a TypeError exception.
Boolean	Return a new Boolean object whose [[BooleanData]] internal slot is set to the value of *argument*. See 19.3 for a description of Boolean objects.
Number	Return a new Number object whose [[NumberData]] internal slot is set to the value of *argument*. See 20.1 for a description of Number objects.
String	Return a new String object whose [[StringData]] internal slot is set to the value of *argument*. See 21.1 for a description of String objects.
Symbol	Return a new Symbol object whose [[SymbolData]] internal slot is set to the value of *argument*. See 19.4 for a description of Symbol objects.
Object	Return *argument*.

Glance at Semantics: +



12.7.3.1 Runtime Semantics: Evaluation

operator +

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

```
1. Let lref be the result of evaluating AdditiveExpression.
2. Let lval be GetValue(lref).
3. ReturnIfAbrupt(lval).
4. Let rref be the result of evaluating MultiplicativeExpression.
5. Let rval be GetValue(rref).
6. ReturnIfAbrupt(rval).
7. Let lprim be ToPrimitive(lval).
8. ReturnIfAbrupt(lprim).
9. Let rprim be ToPrimitive(rval).
10. ReturnIfAbrupt(rprim).
11. If Type(lprim) is String or Type(rprim) is String, then
    a. Let lstr be ToString(lprim).
    b. ReturnIfAbrupt(lstr).
    c. Let rstr be ToString(rprim).
    d. ReturnIfAbrupt(rstr).
    e. Return the String that is the result of concatenating lstr and rstr.
12. Let lnum be ToNumber(lprim).
13. ReturnIfAbrupt(lnum).
14. Let rnum be ToNumber(rprim).
15. ReturnIfAbrupt(rnum).
16. Return the result of applying the addition operation to lnum and rnum. See the Note below 12.7.5.
```

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Exotic objects may handle the absence of a hint in some other manner.

NOTE 2 Step 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.11), by using the logical-or operation instead of the logical-and operation.

Local outcome

`Number Conversion, Number Add`

`String Conversion, String Add`

Side effects

```
JS property access
User method invocations
Proxy method invocations
```

Challenge: untyped variables and operations

```
function add(a, b) {
     return a + b;
}
add(1, 2);
add(“foo”, 1);
add(1, “foo”);
add({foo: “”}, 1);
add(“hello”, {toString: () => “me”});
add(1.01, 3.03);
```

- Variables, parameters, properties, and expressions do not have types
- Operators are overloaded for different types of values

Challenge: eval

```
function add(a, b) {
     return eval(a) + b;
}
add(1, 2);
add(“b = 30”, 1);
```

- The eval operator evaluates a string as if the code was injected directly into the scope
- Can modify locals, introduce new locals, and other horrible things

Challenge: eval

```
function add(a, b) {
     return eval(a) + b;
}
add(1, 2);
add(“b = 30”, 1);
```

- The eval operator evaluates a string as if the code was injected directly into the scope
- Can modify locals, introduce new locals, and other horrible things

Other challenges

```
function one(a, b) {
     var x = a + y;
     var y = 3;  // funky scoping
}

with (o) { print(x); }  // with scopes

function doit(x) {
  print(arguments);  // arguments objects
}

function* all(x) {
  for (y in x) yield y;  // generators
}
```

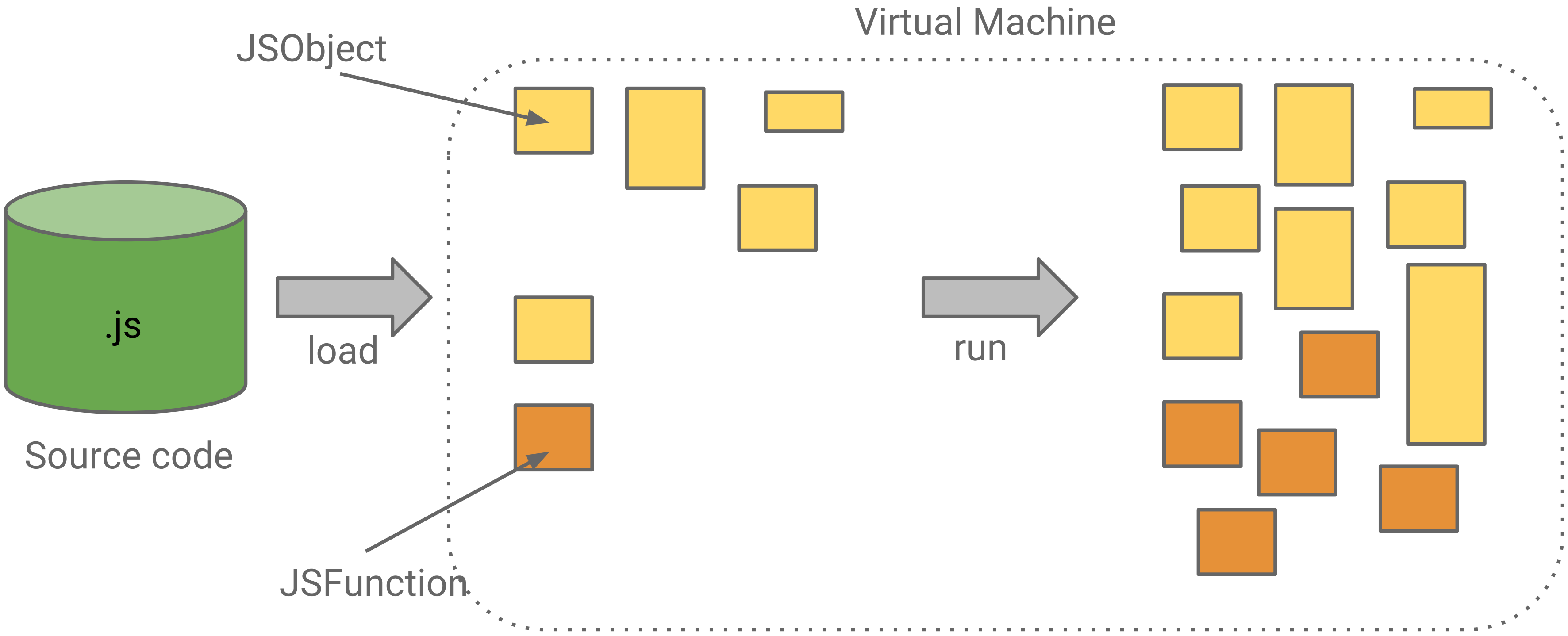
- Lots of neat and surprisingly tricky features
- Most interact poorly
- Conversion gotchas, like the odd falsy object
- Proxies
- Web compatibility issues

The V8 Approach

What makes V8 unique and interesting?

- V8 was the first really fast JavaScript Virtual Machine
  - Launched with Chrome in 2008
  - 10x faster than competition at release
  - 10x faster today than 2008
- Efficient object model using “hidden classes,” a technique from Self VM
- JITs galore
  - Fast AST-walking JIT compiler: fullcodegen (2008) with inline caching
  - Optimizing JIT compiler: Crankshaft (2010) with type feedback and deoptimization
  - Optimizing JIT compiler: TurboFan (2015) with type and range analysis, sea of nodes
- GCs galore
  - Evolution from simple generational collector to incremental and concurrent collector
  - Scheduling GC to reduce jank and save memory

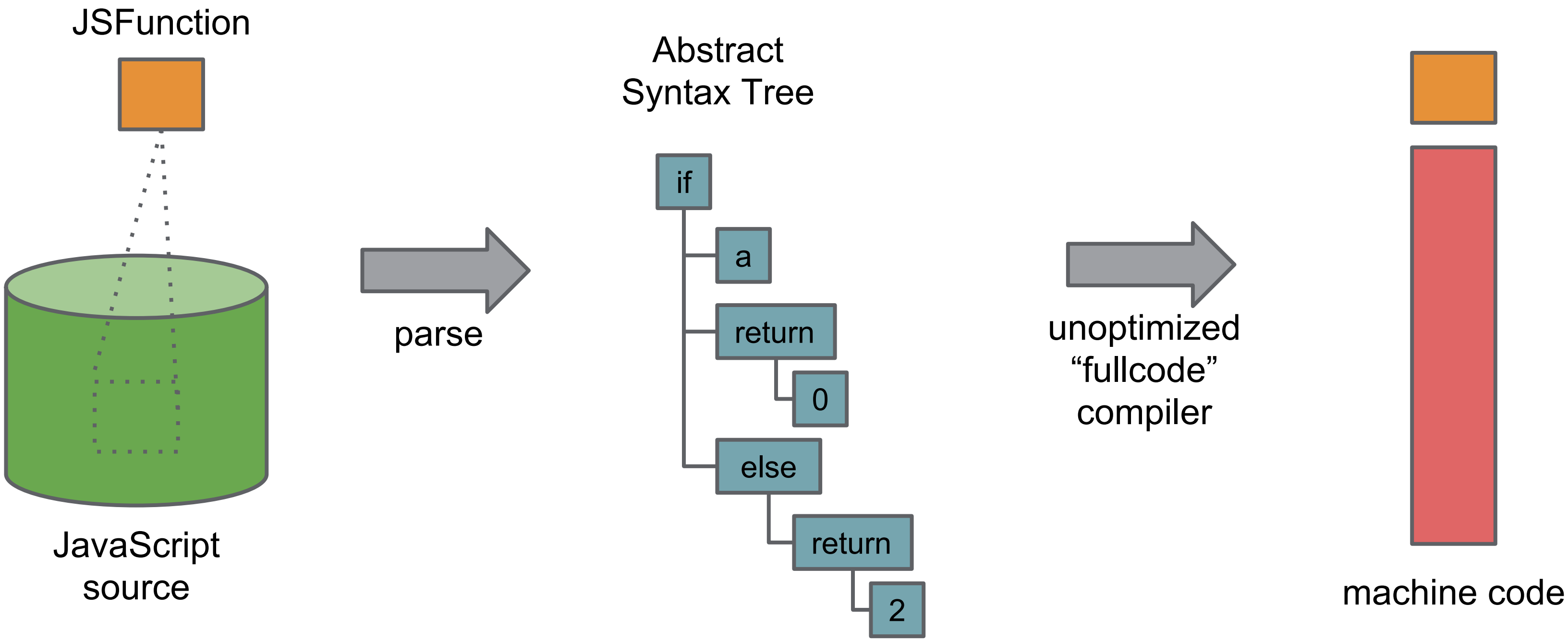
JavaScript Program Lifetime



V8 Approach: parsing

- Parsing has to be fast
    - Parsing JS is hard: hand-written, recursive descent parser
- Two modes:
    - preparse (detect structure only)
    - full (build AST) ~3x slower
- Lazy parsing:
    - A full parse of a function isn’t done until needed to execute it
    - Preparser finds boundaries of functions to quickly parse them later
- Streaming parsing:
    - Parse while script is downloading over the wire

V8 Approach: lazy compilation

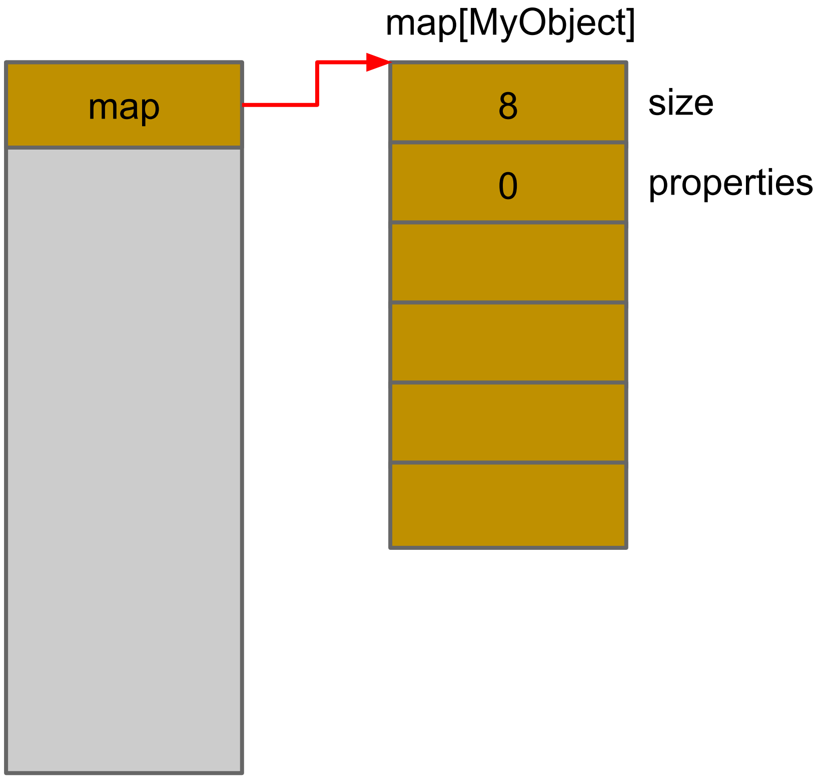


V8 Approach: object model

```
function MyObject(name, data) {
     this.name = name;
     this.data = data;
     return this;
}
var x = new MyObject(“string”, 0);
x.extra = 44;
```

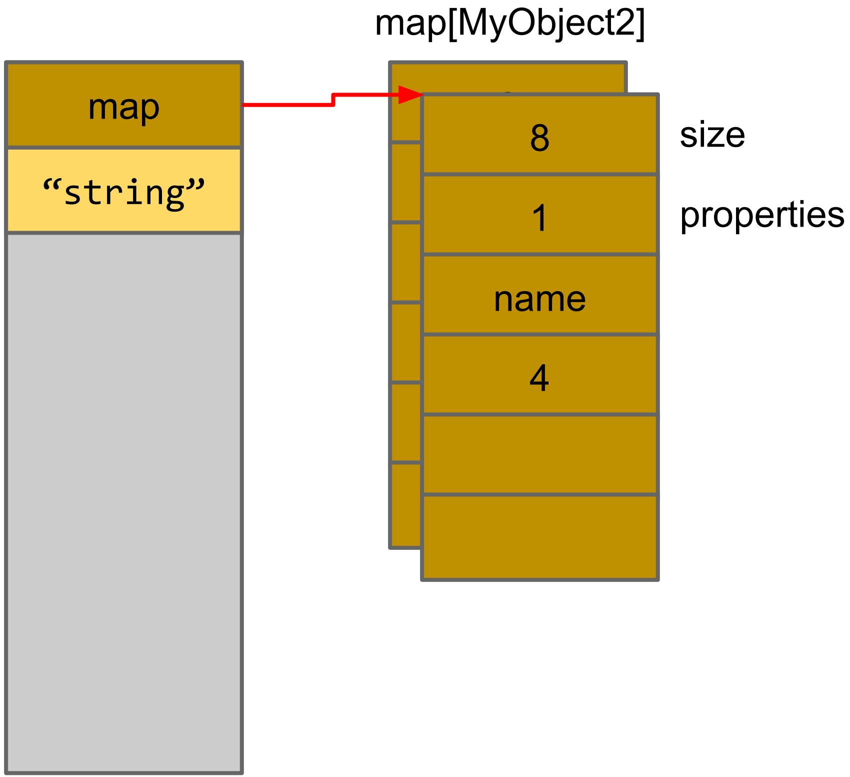
V8 Approach: object model

```
function MyObject(name, data) {
     this.name = name;
     this.data = data;
     return this;
}
var x = new MyObject(“string”, 0);
x.extra = 44;
```



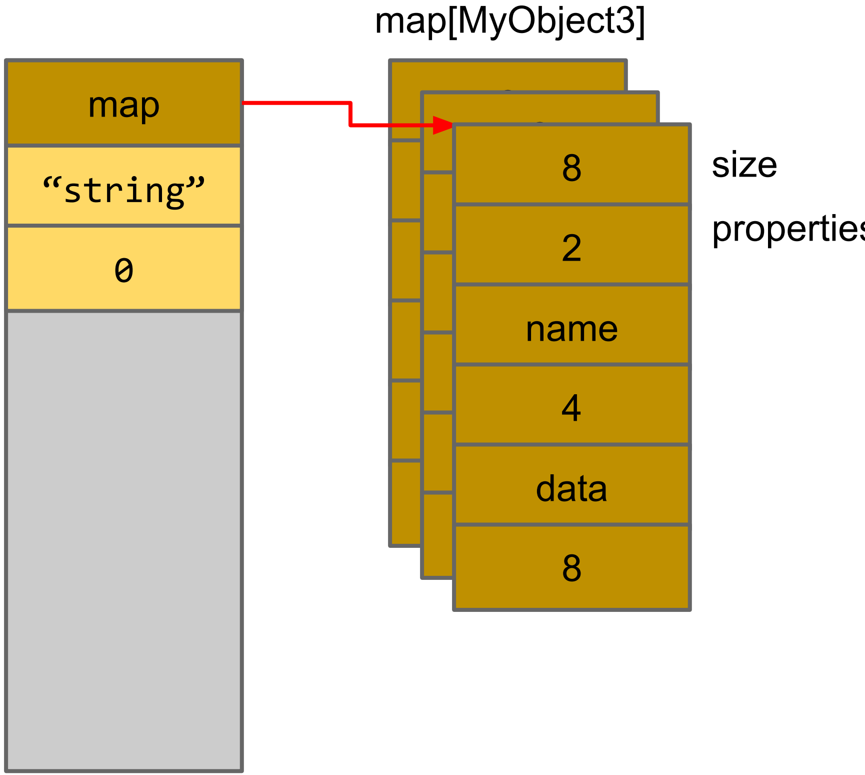
V8 Approach: object model

```
function MyObject(name, data) {
     this.name = name;
     this.data = data;
     return this;
}
var x = new MyObject(“string”, 0);
x.extra = 44;
```



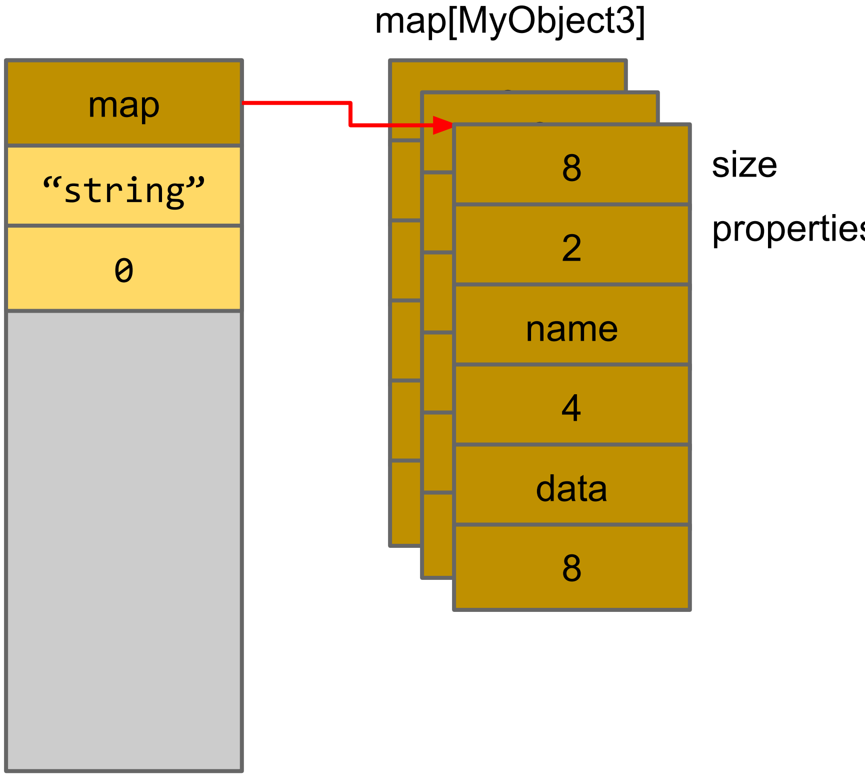
V8 Approach: object model

```
function MyObject(name, data) {
     this.name = name;
     this.data = data;
     return this;
}
var x = new MyObject(“string”, 0);
x.extra = 44;
```



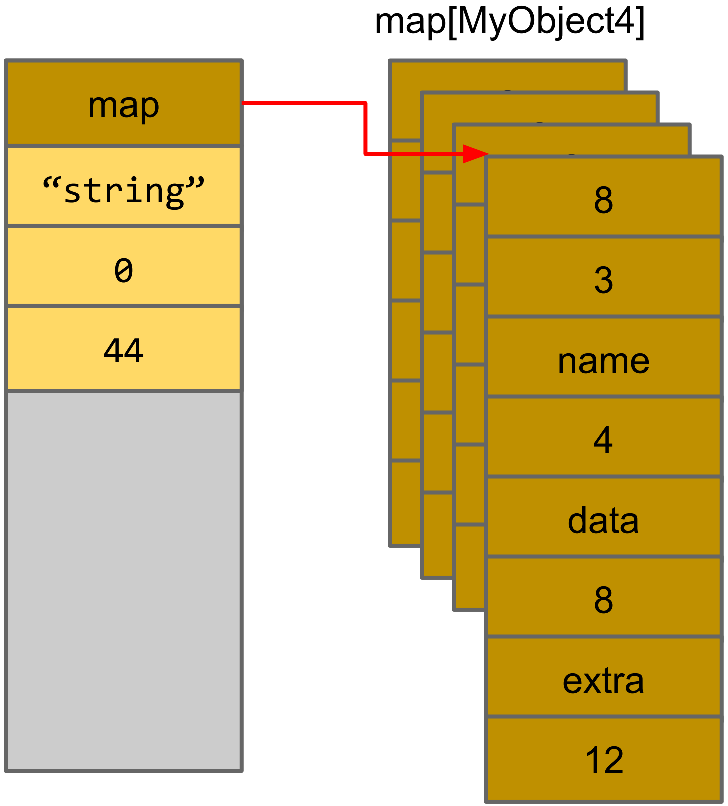
V8 Approach: object model

```
function MyObject(name, data) {
     this.name = name;
     this.data = data;
     return this;
}
var x = new MyObject(“string”, 0);
x.extra = 44;
```

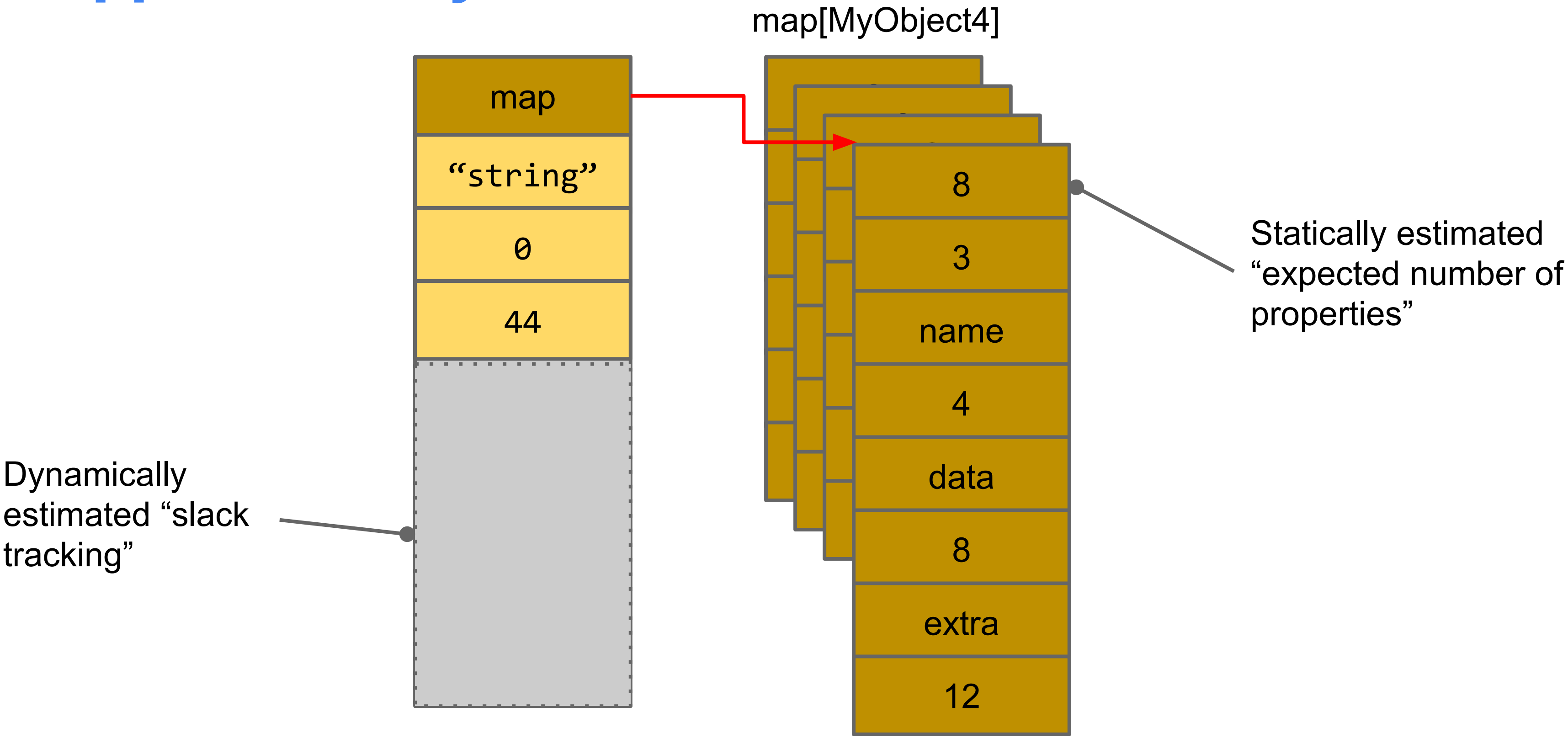


V8 Approach: object model

```
function MyObject(name, data) {
     this.name = name;
     this.data = data;
     return this;
}
var x = new MyObject(“string”, 0);
x.extra = 44;
```



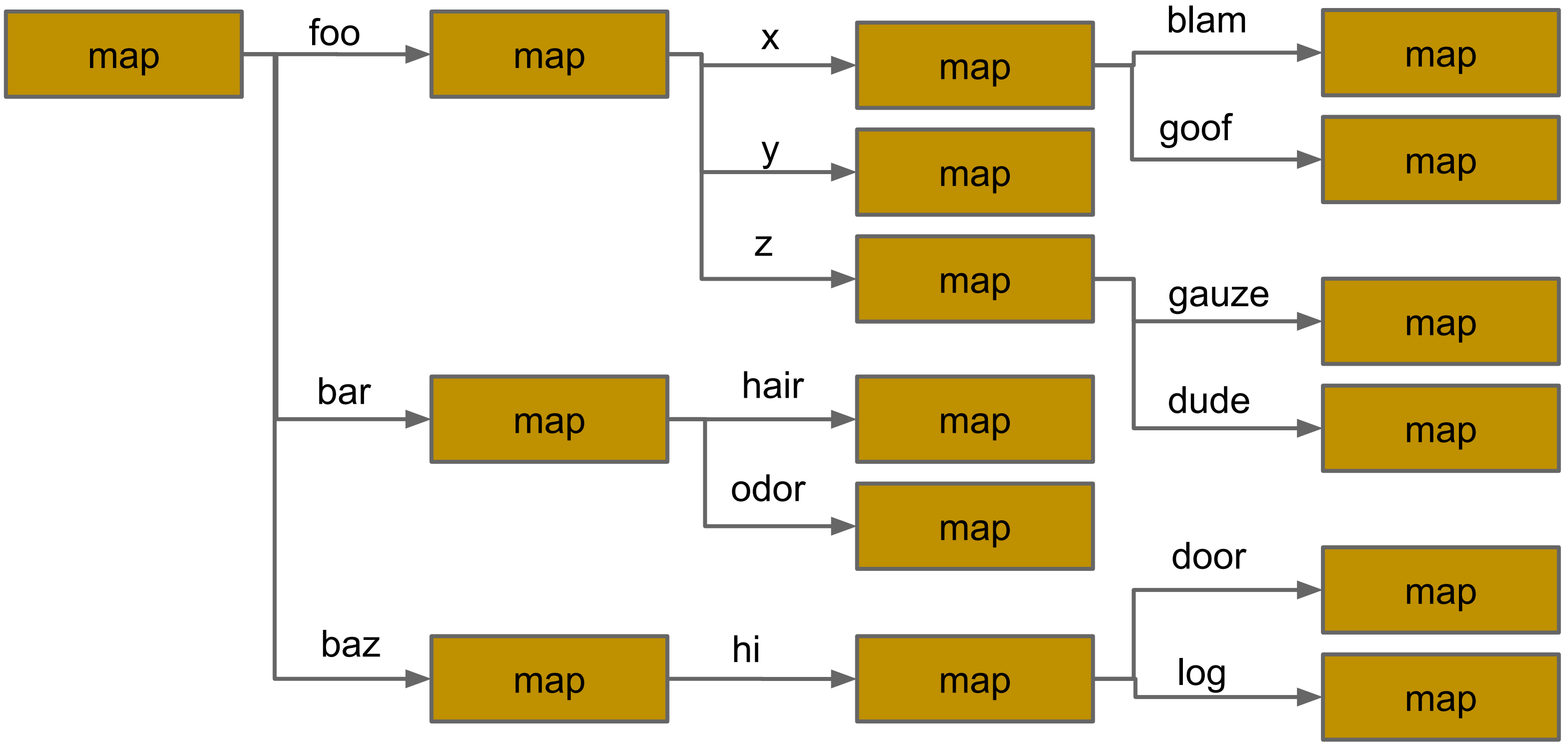
V8 Approach: object model



Statically estimated “expected number of properties”

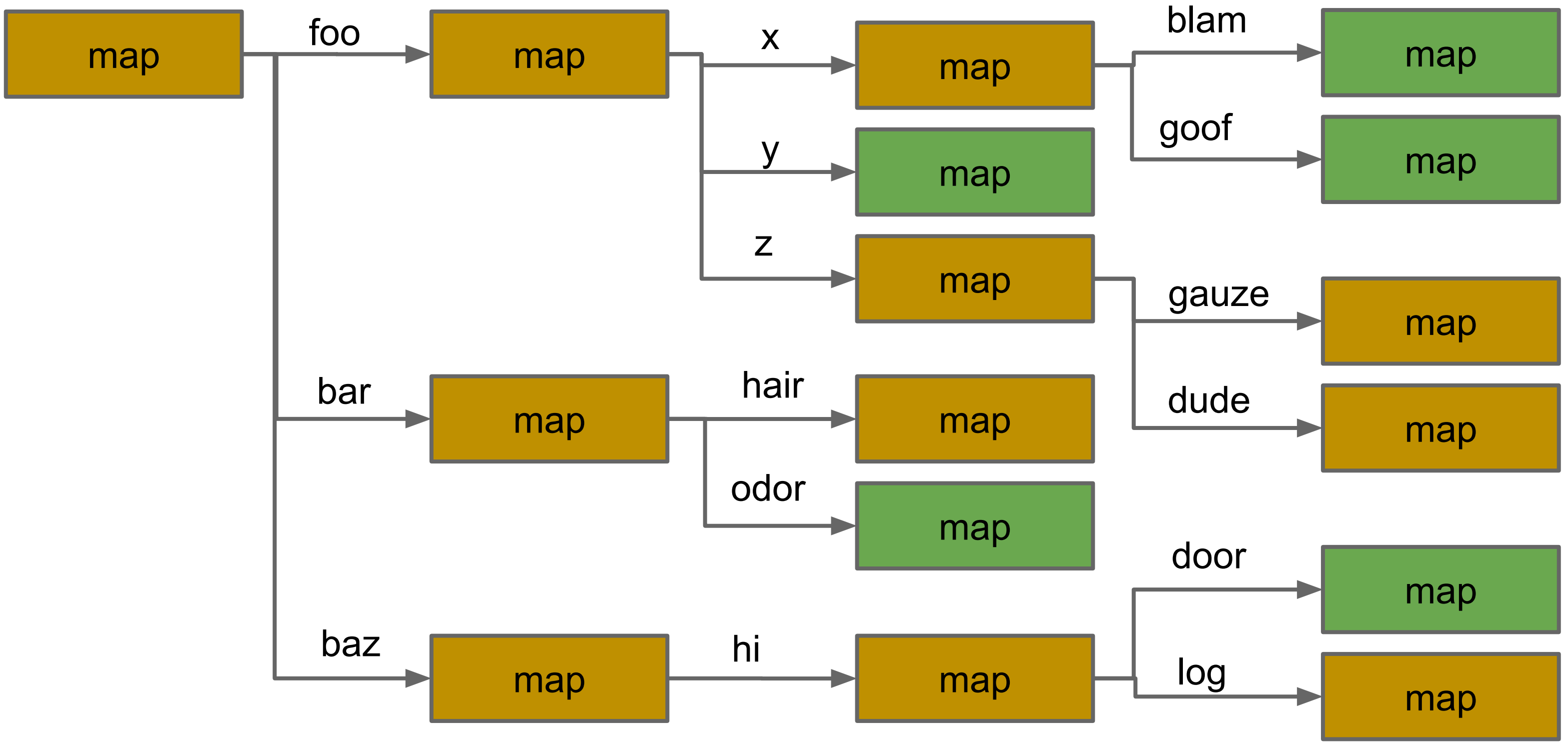
Dynamically estimated “slack tracking”

V8 Approach: map forest



V8 Approach: map forest

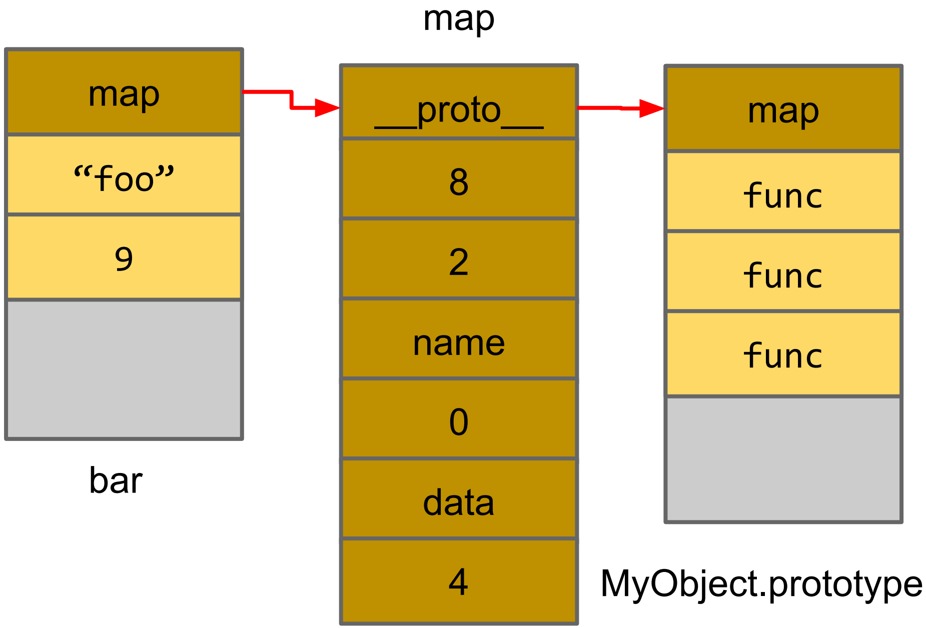
potentially stable map 🟩



Google

V8 Approach: object model

```
function MyObject(name, data) {
      this.name = name;
      this.data = data;
      return this;
}
MyObject.prototype.print =
  function() {
    print(“name: “ + this.name);
    print(“data: “ + this.data);
}
var bar = new MyObject(“foo”, 9);
```

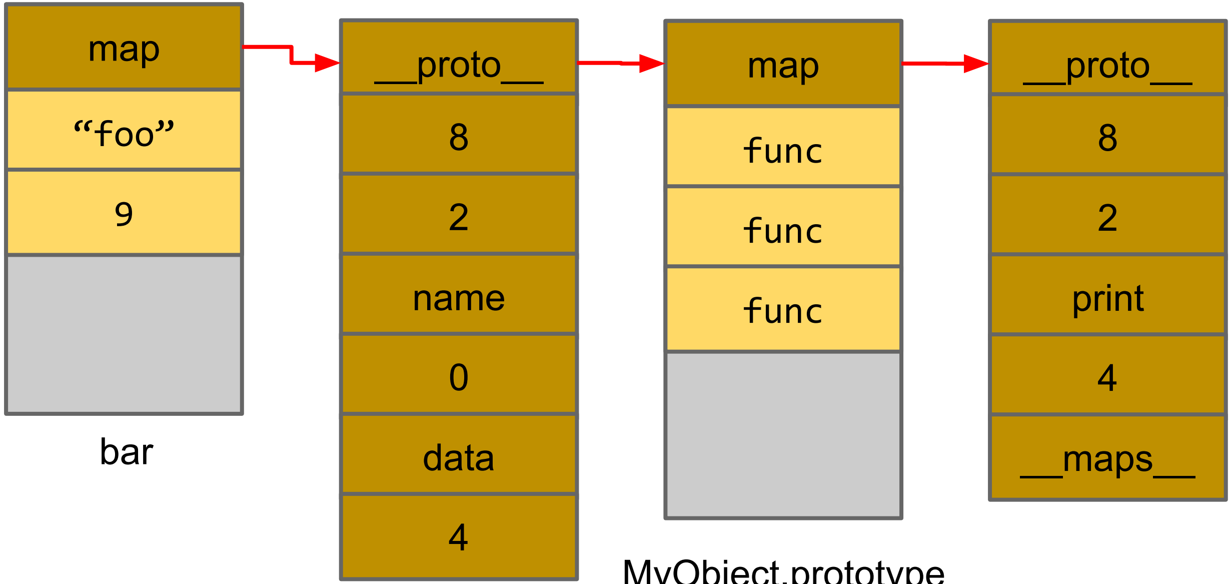


map
“foo”
9
bar

map
__proto__
8
2
name
0
data
4

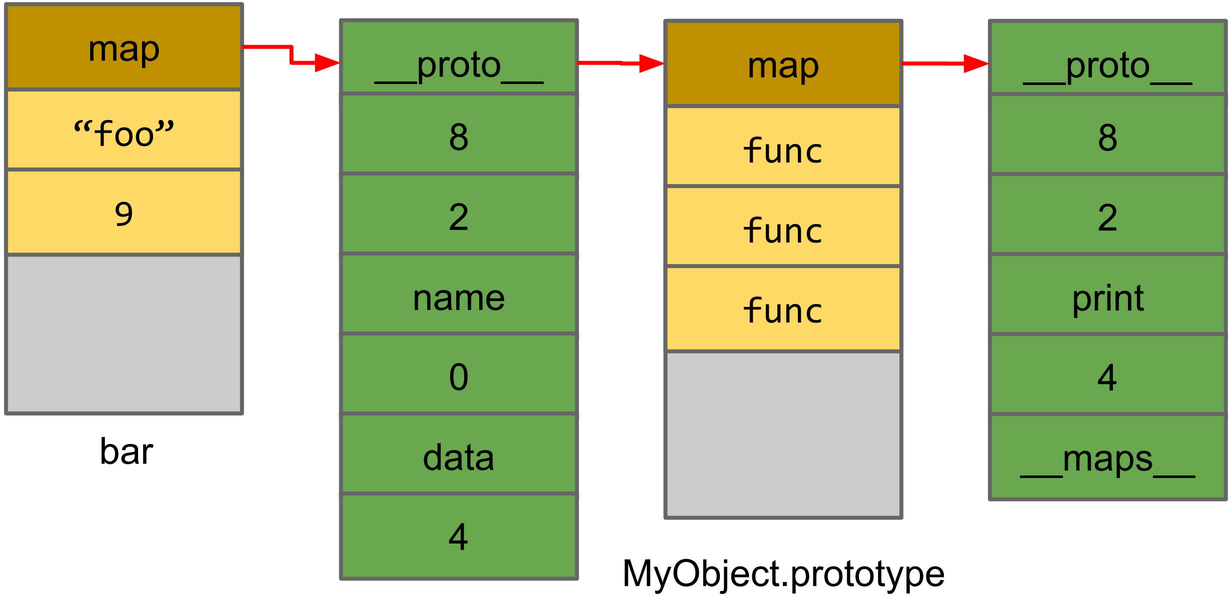
map
func
func
func
MyObject.prototype

V8 Approach: object model



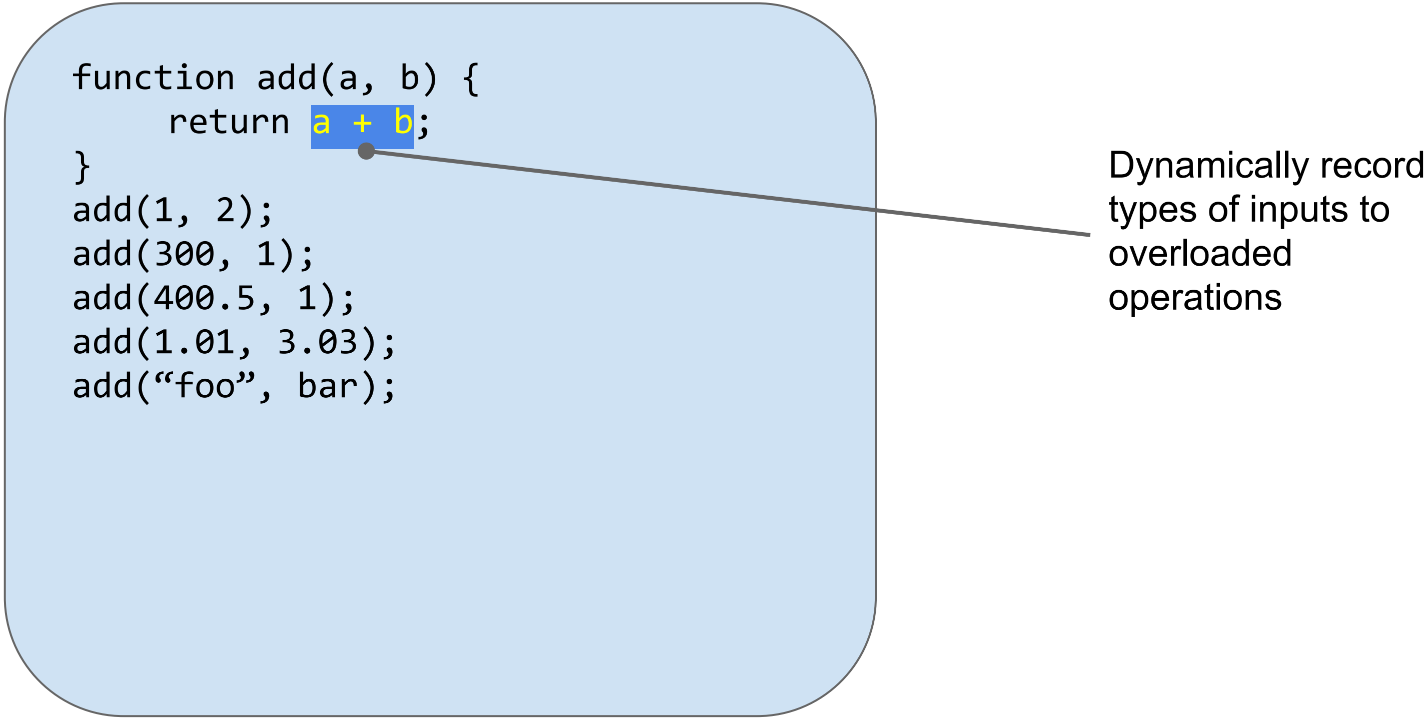
MyObject.prototype

V8 Approach: object model



Google

V8 Approach: untyped variables and operations



```
function add(a, b) {
     return a + b;
}
add(1, 2);
add(300, 1);
add(400.5, 1);
add(1.01, 3.03);
add(“foo”, bar);
```

Dynamically record types of inputs to overloaded operations

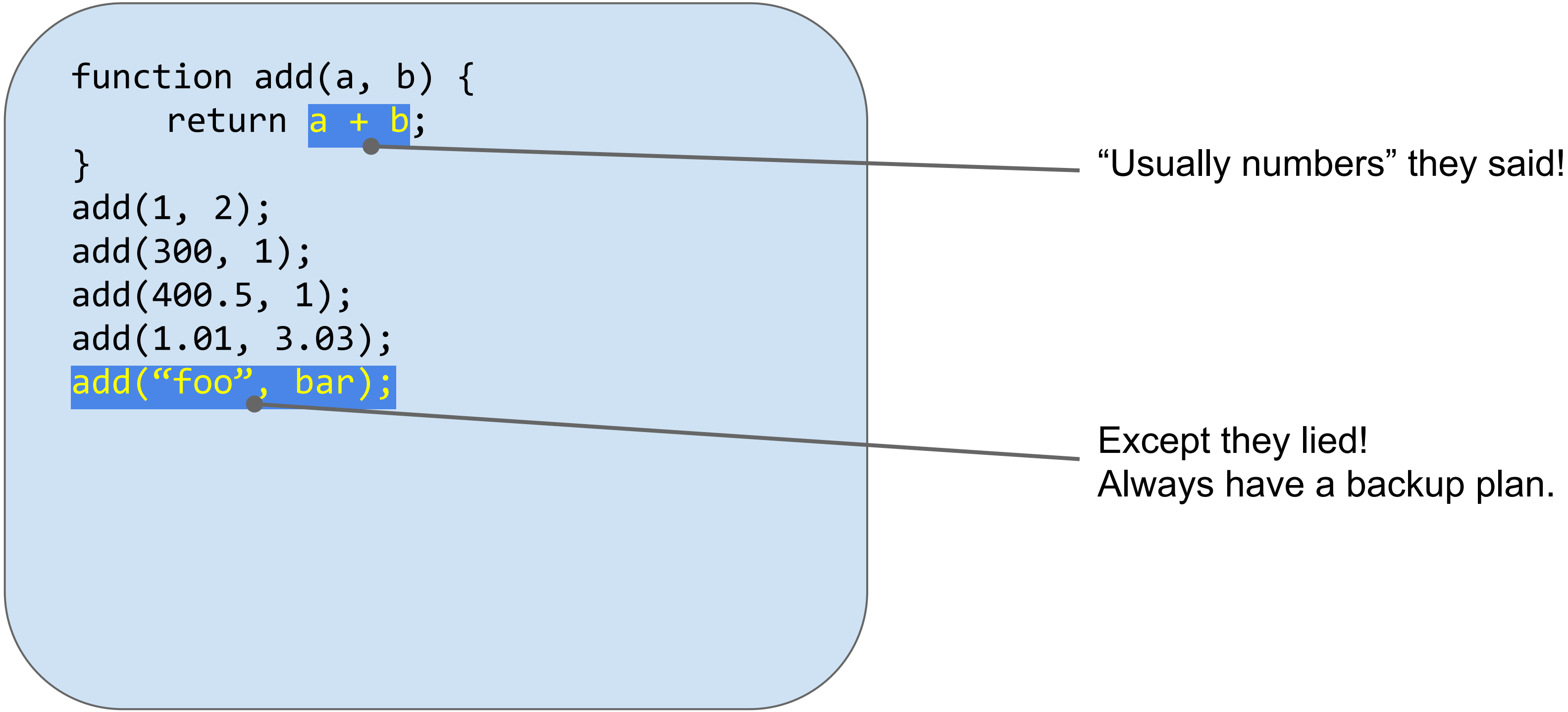
V8 Approach: untyped variables and operations

```
function add(a, b) {
     return a + b;
}
add(1, 2);
add(300, 1);
add(400.5, 1);
add(1.01, 3.03);
add(“foo”, bar);
```

Dynamically record types of inputs to overloaded operations

Most dynamism is site-specific and stable. Normally safe to assume that what happened last time will happen the next time.

V8 Approach: untyped variables and operations



```
function add(a, b) {
     return a + b;
}
add(1, 2);
add(300, 1);
add(400.5, 1);
add(1.01, 3.03);
add(“foo”, bar);
```

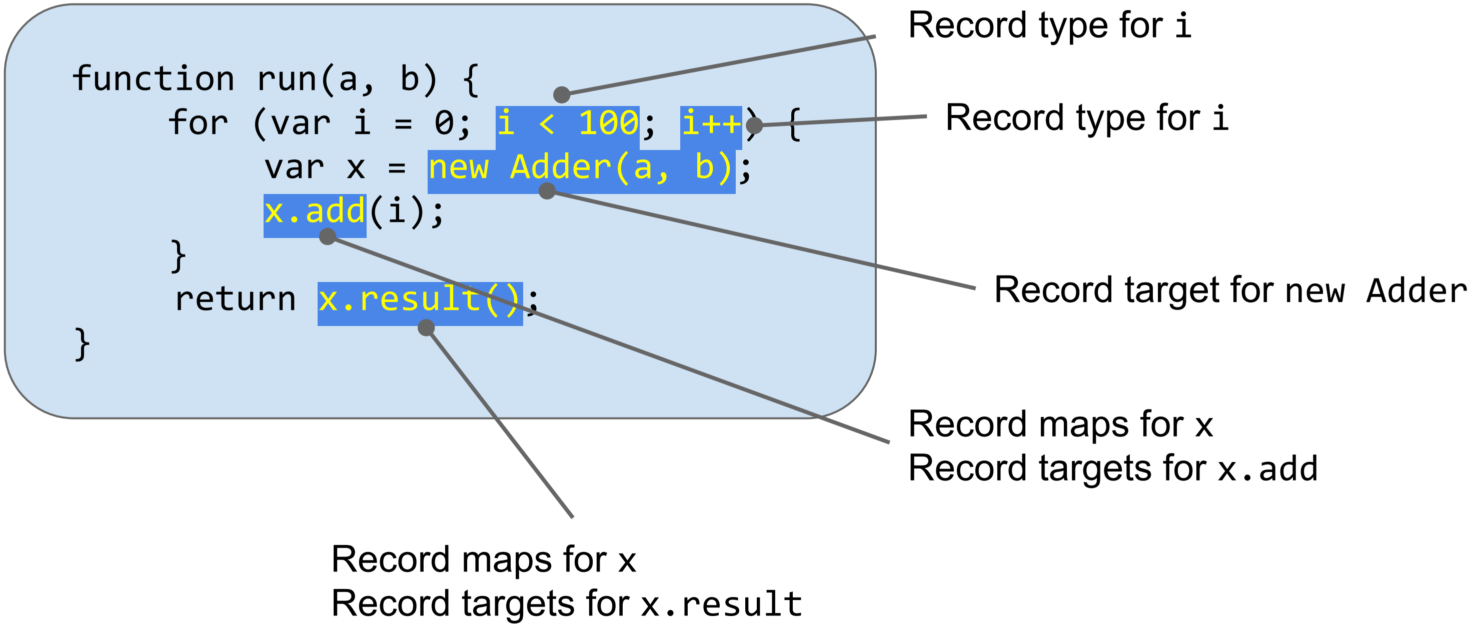
“Usually numbers” they said!

Except they lied!
Always have a backup plan.

V8 Approach: adaptive optimization

```
function run(a, b) {
    for (var i = 0; i < 100; i++) {
        var x = new Adder(a, b);
        x.add(i);
    }
    return x.result();
}
```

V8 Approach: adaptive optimization



```
function run(a, b) {
     for (var i = 0; i < 100; i++) {
          var x = new Adder(a, b);
          x.add(i);
     }
     return x.result();
}
```

Record type for `i`

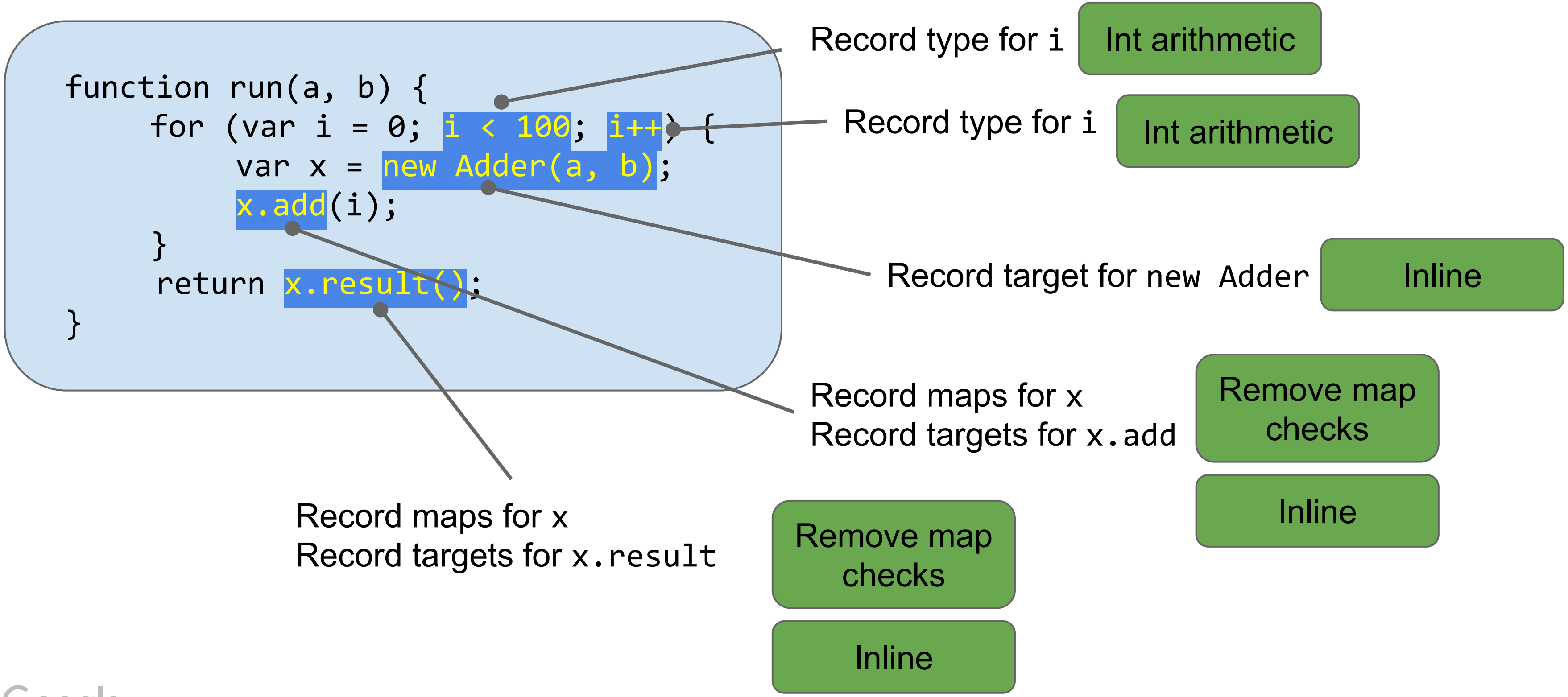
Record type for `i`

Record target for `new Adder`

Record maps for x
Record targets for `x.add`

Record maps for x
Record targets for `x.result`

V8 Approach: adaptive optimization



```
function run(a, b) {
     for (var i = 0; i < 100; i++) {
          var x = new Adder(a, b);
          x.add(i);
     }
     return x.result();
}
```

Record type for `i` — Int arithmetic

Record type for `i` — Int arithmetic

Record target for `new Adder` — Inline

Record maps for x
Record targets for `x.add` — Remove map checks; Inline

Record maps for x
Record targets for `x.result` — Remove map checks; Inline

V8 Approach: adaptive optimization

```
function run(a, b) {
    for (var i = 0; i < 100; i++) {
        var x = new Adder(a, b);
        x.add(i);
    }
    return x.result();
}
```

hotness criteria

Escape analysis

Type analysis

Int arithmetic

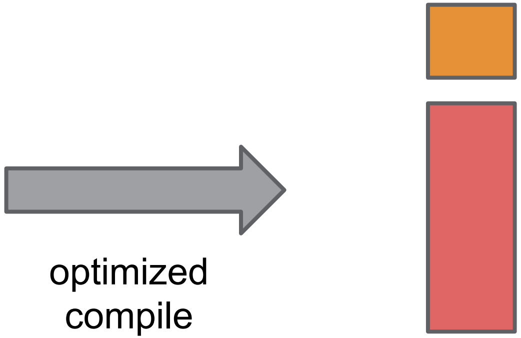
Remove map checks

Inline

GVN

V8 Approach: adaptive optimization

```
function run(a, b) {
    for (var i = 0; i < 100; i++) {
        var x = new Adder(a, b);
        x.add(i);
    }
    return x.result();
}
```

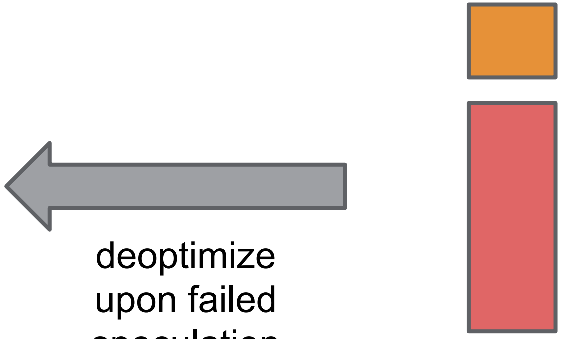


optimized compile

optimized code

V8 Approach: adaptive optimization

```
function run(a, b) {
     for (var i = 0; i < 100; i++) {
          var x = new Adder(a, b);
          x.add(i);
     }
     return x.result();
}
```

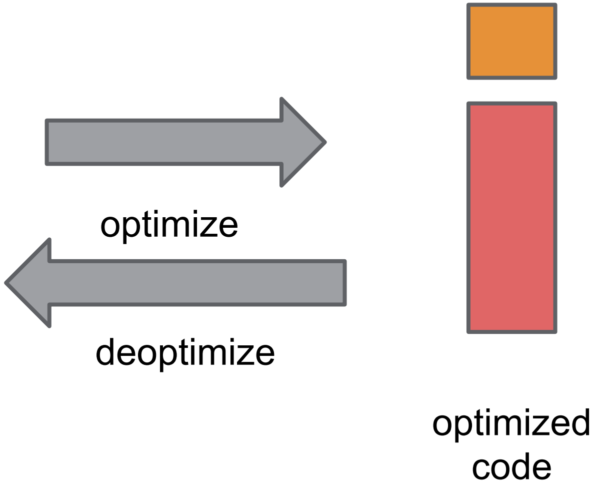


deoptimize
upon failed
speculation

optimized
code

V8 Approach: adaptive optimization

```
function run(a, b) {
    for (var i = 0; i < 100; i++) {
        var x = new Adder(a, b);
        x.add(i);
    }
    return x.result();
}
```



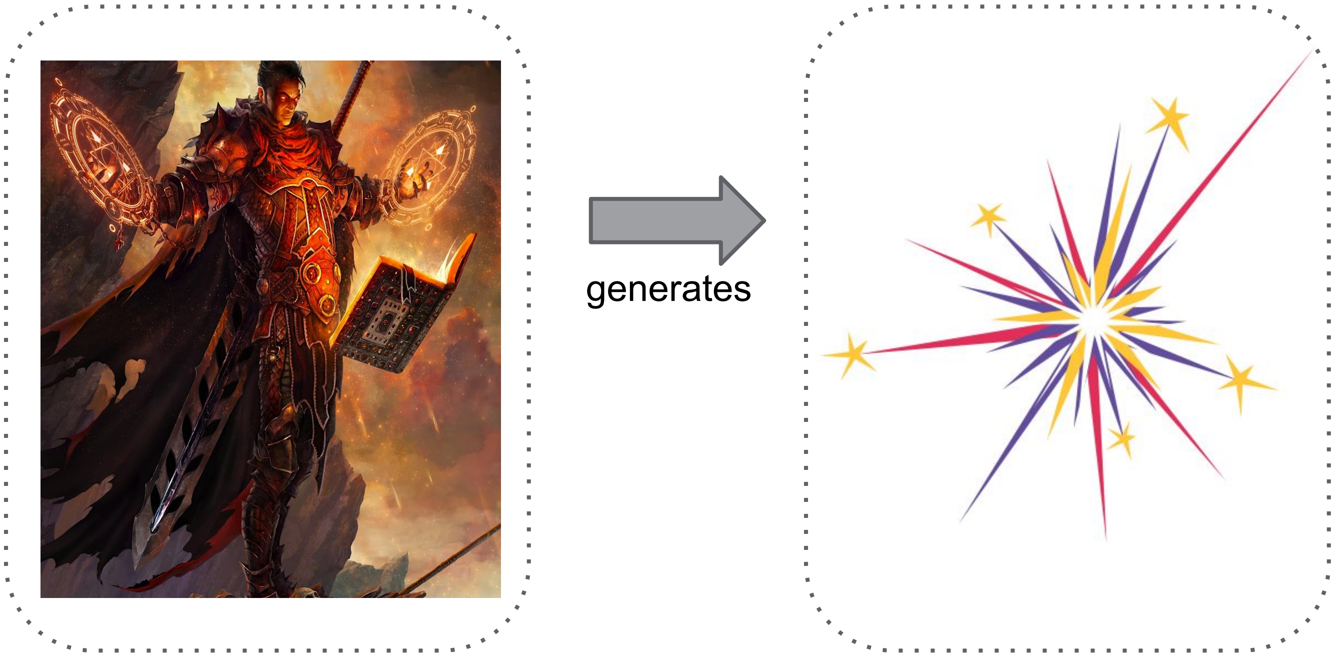
A Zoo of Tiers

FullCodeGen
Unoptimized compiler

CrankShaft
optimizing compiler

TurboFan
optimizing compiler

A Zoo of Tiers (4)



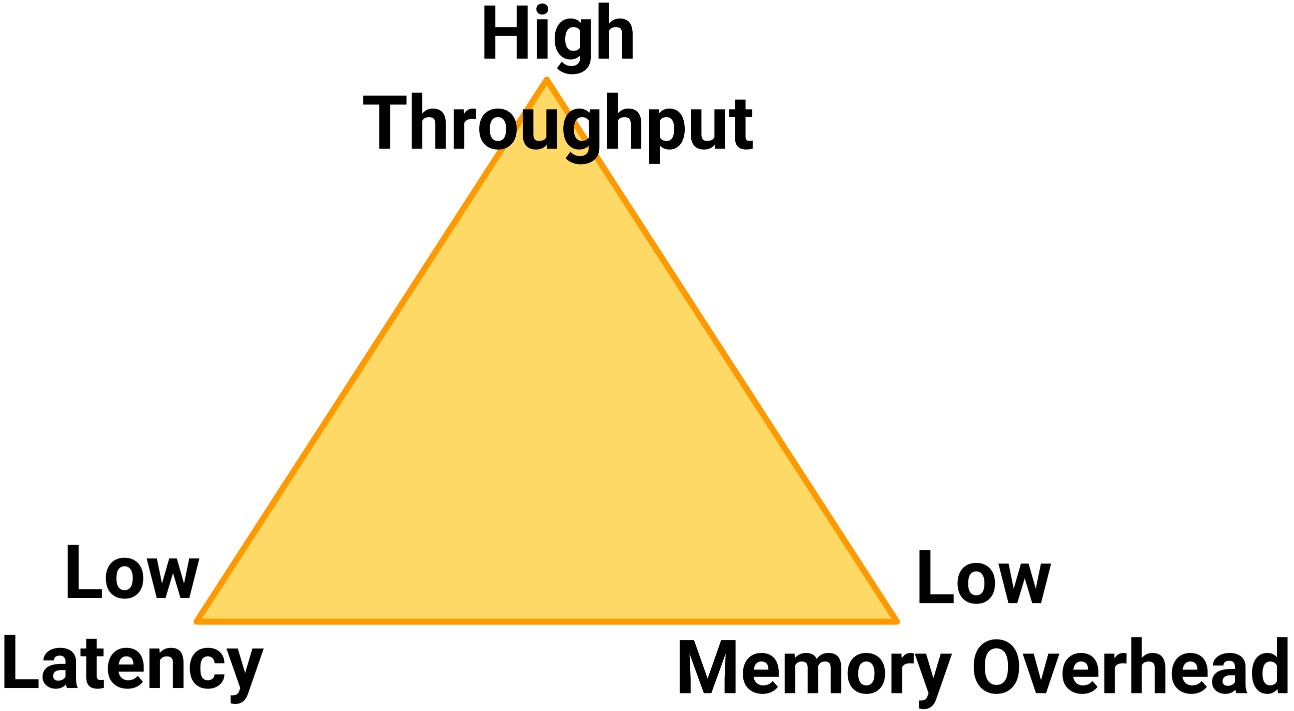
TurboFan
optimizing compiler

generates

Ignition
interpreter

- Faster startup!
- Saves memory!
- Still portable!
  (11 supported TurboFan archs)

The Impossible Garbage Collection Triad



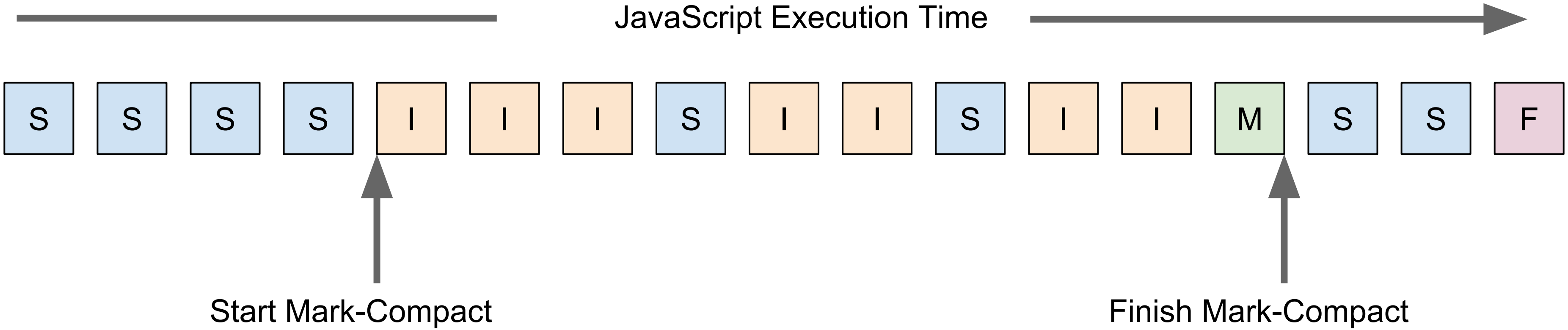
V8 Garbage Collection

S Scavenger (~0-10 ms)

I Incremental Marking (~0.01-CONFIGURABLE ms)

M Final Mark-Compact Collection (~4-40 ms)

F Full Mark-Compact Collection (>40ms)



JavaScript Execution Time

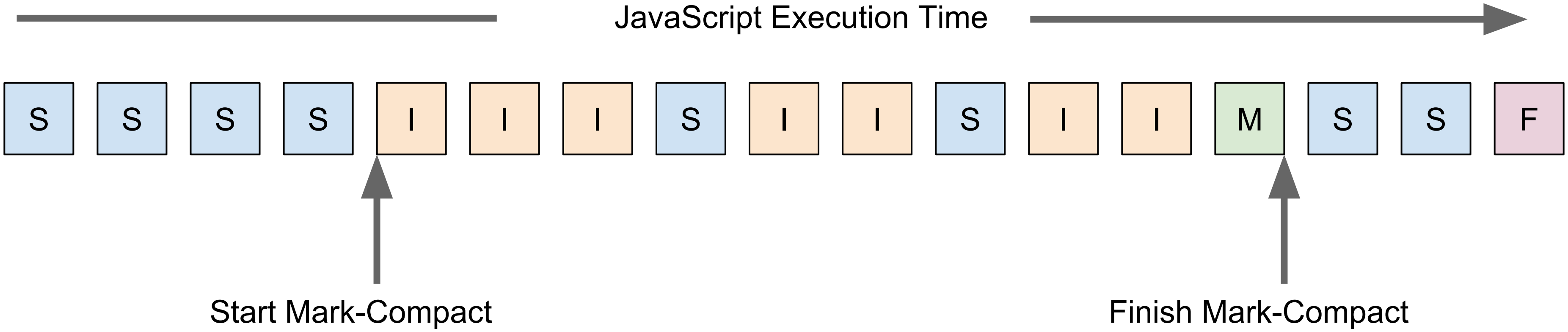
S S S S I I I S I I S I I M S S F

Start Mark-Compact

Finish Mark-Compact

Estimating GC pauses

S	Scavenger (~0-10 ms)
I	Incremental Marking (~0.01-CONFIGURABLE ms)
M	Final Mark-Compact Collection (~4-40 ms)
F	Full Mark-Compact Collection (>40ms)



JavaScript Execution Time

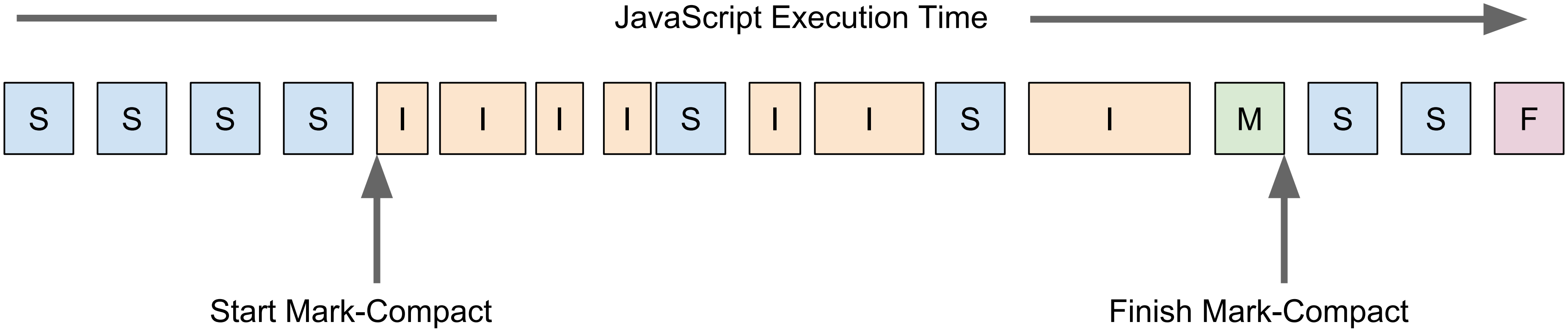
S	S	S	S	I	I	I	S	I	I	S	I	I	M	S	S	F

Start Mark-Compact

Finish Mark-Compact

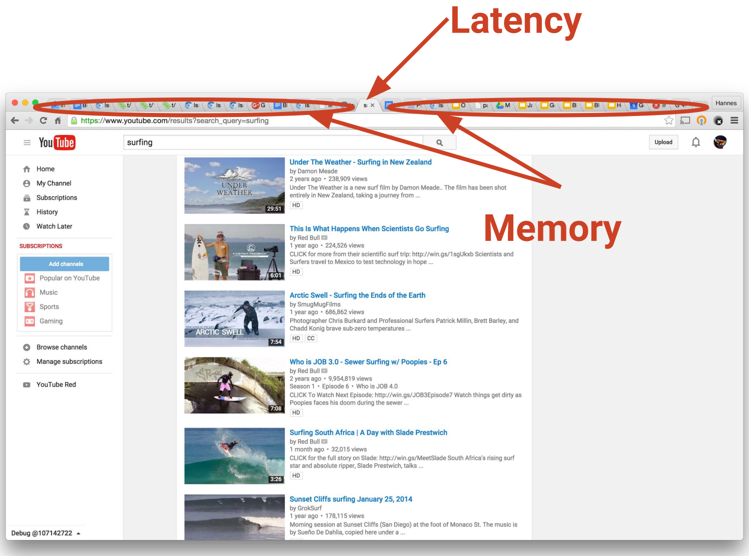
Estimating GC pauses

- **S** Scavenger (~0-10 ms)
- **I** Incremental Marking (~0.01-CONFIGURABLE ms)
- **M** Final Mark-Compact Collection (~4-40 ms)
- **F** Full Mark-Compact Collection (>40ms)



Latency versus Memory Overhead

- Foreground tab
  - Latency is critical
  - New frames are drawn every 16.66 ms when animation or scrolling happens
  - Reducing memory becomes important as soon as the tab becomes inactive
- Background tabs
  - Memory consumption more important than latency
  - Idle tabs can be aggressively garbage collected to save memory



Under The Weather - Surfing in New Zealand
by Damon Meade
2 years ago • 238,909 views
Under The Weather is a new surf film by Damon Meade.. The film has been shot entirely in New Zealand, taking a journey from ...
HD

This Is What Happens When Scientists Go Surfing
by Red Bull
1 year ago • 224,526 views
CLICK for more from their scientific surf trip: http://win.gs/1sgUkxb Scientists and Surfers travel to Mexico to test technology in hope ...
HD

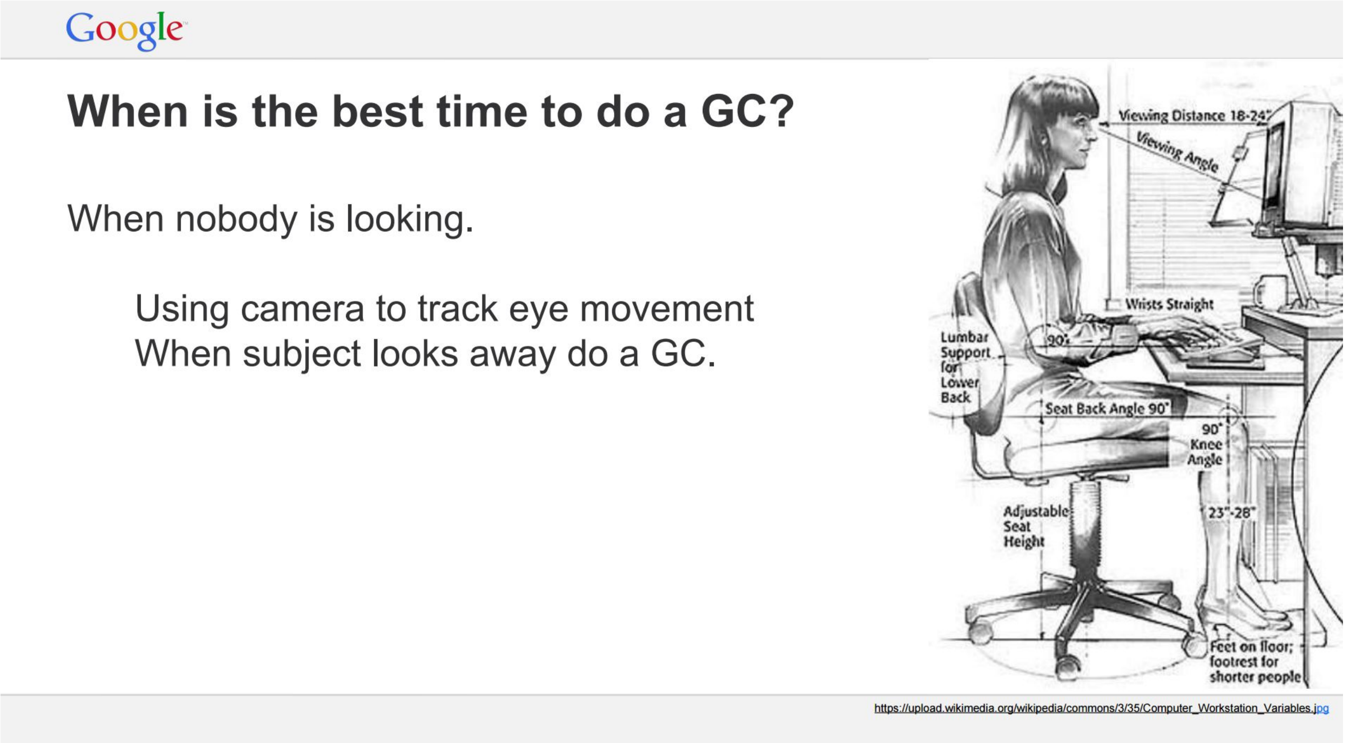
Arctic Swell - Surfing the Ends of the Earth
by SmugMugFilms
1 year ago • 686,862 views
Photographer Chris Burkard and Professional Surfers Patrick Millin, Brett Barley, and Chadd Konig brave sub-zero temperatures ...
HD CC

Who is JOB 3.0 - Sewer Surfing w/ Poopies - Ep 6
by Red Bull
2 years ago • 9,954,819 views
Season 1 • Episode 6 • Who is JOB 4.0
CLICK To Watch Next Episode: http://win.gs/JOB3Episode7 Watch things get dirty as Poopies faces his doom during the sewer ...
HD

Surfing South Africa | A Day with Slade Prestwich
by Red Bull
1 month ago • 32,015 views
CLICK for the full story on Slade: http://win.gs/MeetSlade South Africa's rising surf star and absolute ripper, Slade Prestwich, talks ...
HD

Sunset Cliffs surfing January 25, 2014
by GrokSurf
1 year ago • 178,115 views
Morning session at Sunset Cliffs (San Diego) at the foot of Monaco St. The music is by Sueño De Dahlia, copied here under a ...

Idea: Make garbage collection invisible



When is the best time to do a GC?

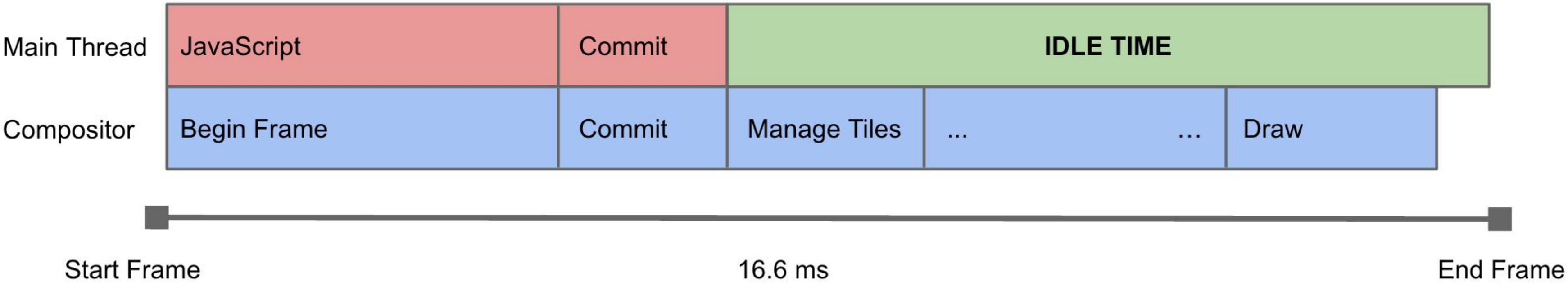
When nobody is looking.

Using camera to track eye movement
When subject looks away do a GC.

https://upload.wikimedia.org/wikipedia/commons/3/35/Computer_Workstation_Variables.jpg

Rick Hudson, GopherCon 2015

Life of an animation Frame



Main Thread

JavaScript

Commit

IDLE TIME

Compositor

Begin Frame

Commit

Manage Tiles

...

...

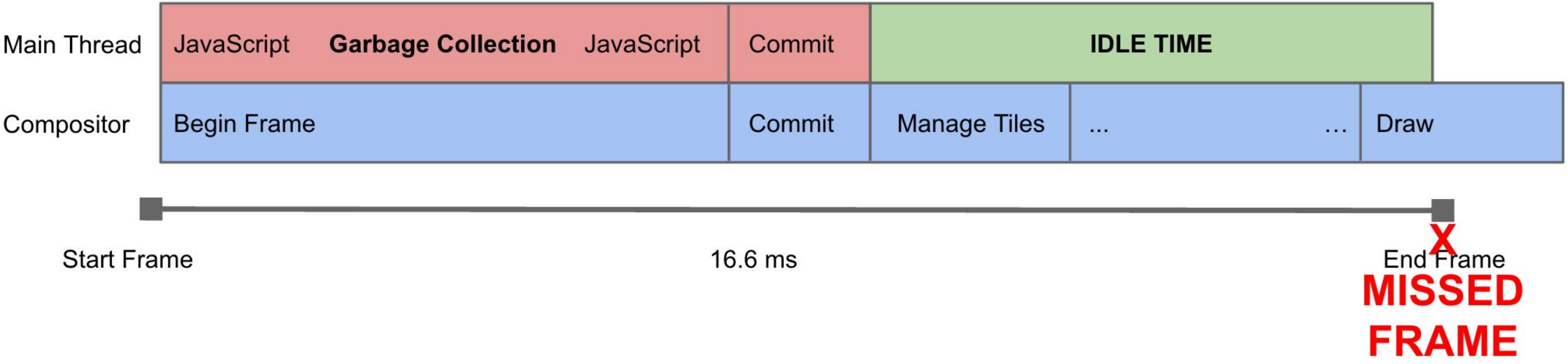
Draw

Start Frame

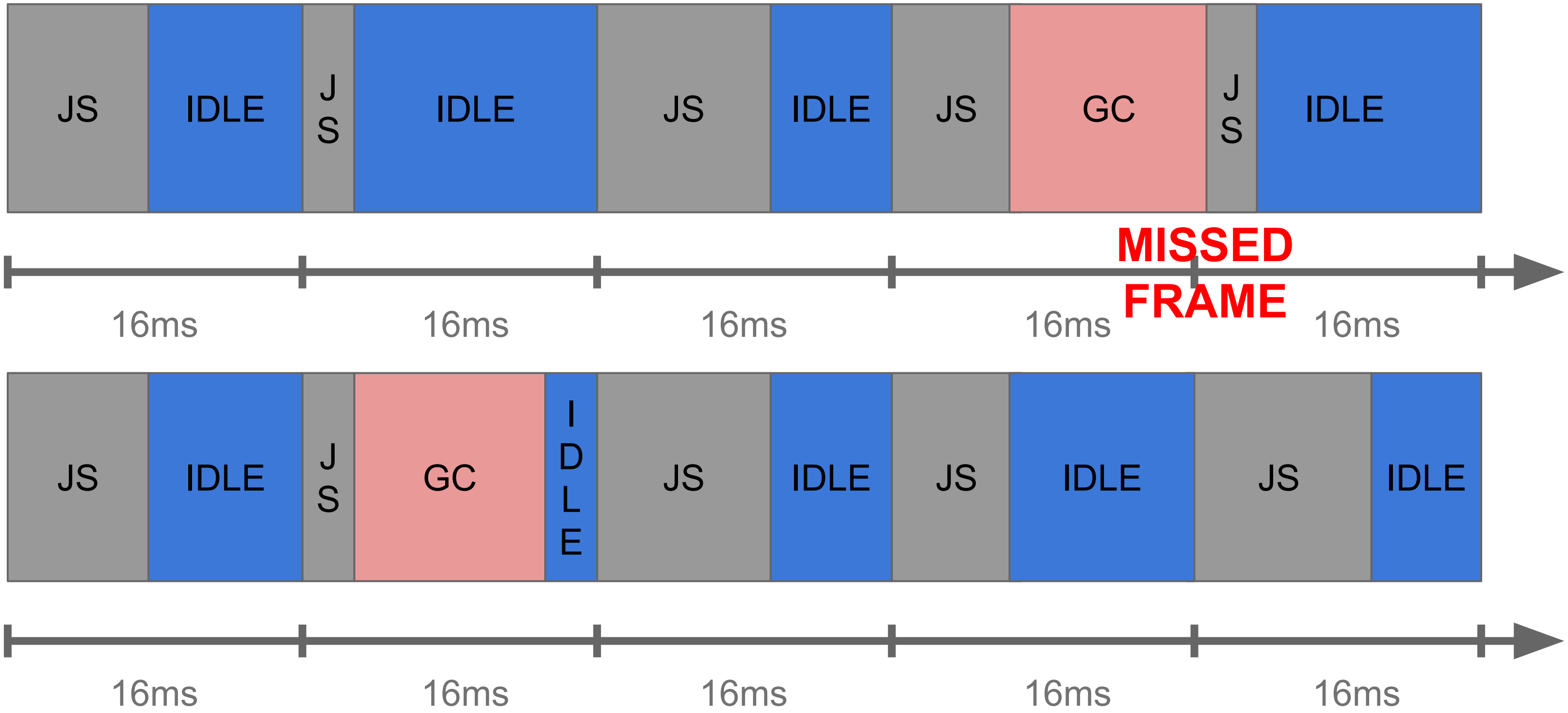
16.6 ms

End Frame

Life of an animation Frame

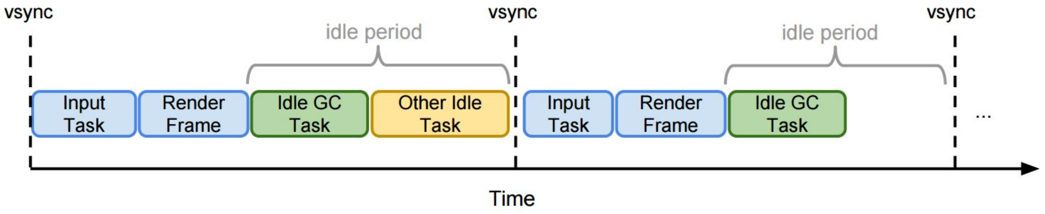


Life of a frame



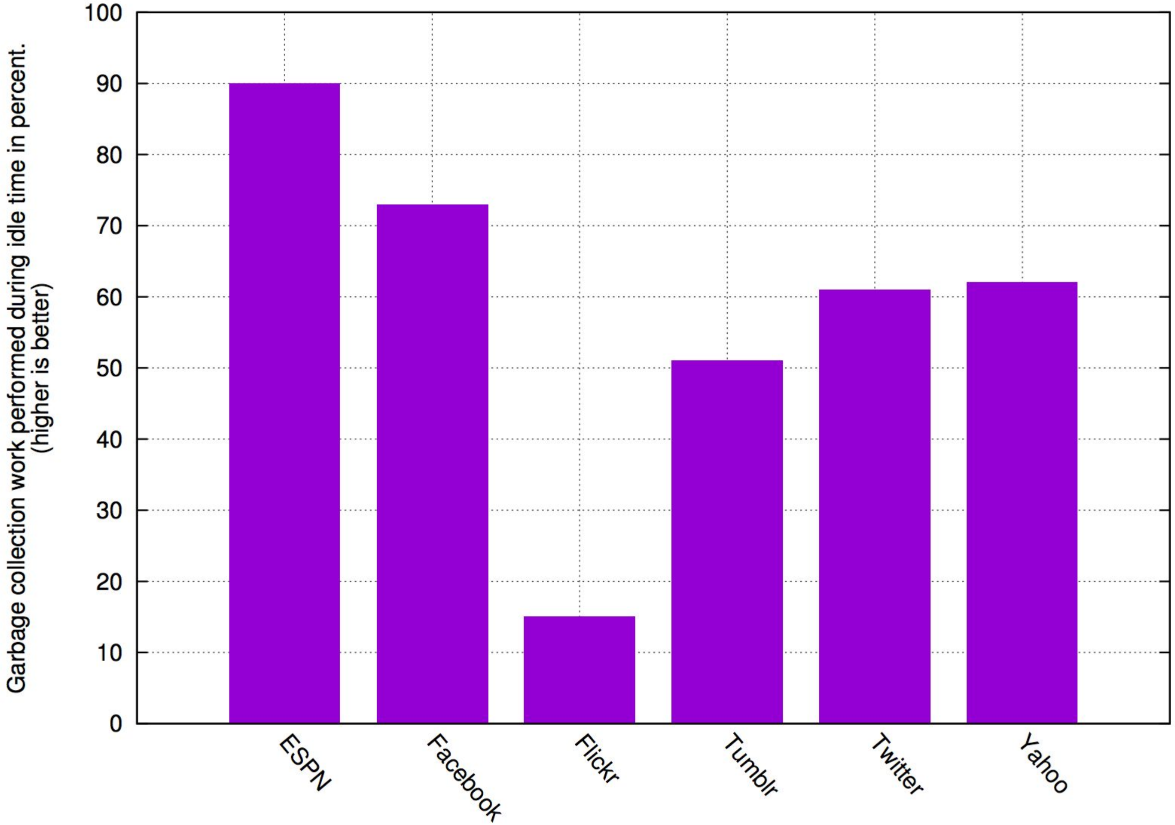
Latency-driven Idle Time GC Scheduling (PLDI16)

- V8 heuristics tries to estimate:
  - average young generation collection speed/MB
  - average incremental marking speed/MB
  - average finalization of mark-compact speed/MB
- V8 registers an *idle garbage collection task* in the Chrome scheduler when a given garbage collection operation should happen soon
- The task scheduler will execute it when there is idle time
  - apportioning up to 50ms to perform garbage collection



Google

Telemetry Infinite Scrolling Benchmarks



WebAssembly
(demo)

Motivation for WebAssembly

- Big pressure to bring native code to the web
  - Competition with installed mobile apps (Android, iOS)
  - Big-time OpenGL apps: games, CAD programs, maps
  - Extensibility: audio/video codecs
- Existing solutions fall short
  - JavaScript increasing contortions to serve as a compilation target
  - PNaCl encountered heavy industry resistance
- Demand for new language capabilities limited by JS bottleneck
  - SIMD
  - SharedArrayBuffer
  - Threads

asm.js? what’s that?

```
a = x + y
```

Normal JavaScript

ToNumber?
ToString?
StringAdd?
IntegerAdd?
DoubleAdd?

x: int32
y: int32

```
a = x + y | 0
```

asm.js

Int32Add
a: int32

x: float64
y: float64

```
a = +(x + y)
```

asm.js

Float64Add
a: float64

asm.js? what’s that? (2)

```
var buffer = new ArrayBuffer(16 * 1024 * 1024);
function module(buffer, stdlib) {
  “use asm”;
  var heap8 = new Int8Array(buffer);
  function foo(a) {
    a = a | 0;
    return heap8[a] + 1 | 0;
  }
  return {foo: foo}
}

var mod = module(buffer, {print: print});
mod.foo(100);
```

asm.js? what’s that? (3)

- Emscripten: A POSIX-like platform with
  - Toolchain based on forked LLVM
  - libc
  - OpenGL (on top of WebGL)
  - a community
  - Game engines
  - Applications
  - Benchmarks

asm.js? what’s that? (4)

- 2 engines specially recognize asm.js subset and *validate* that subset
  - Mozilla Firefox - pioneer
  - Microsoft Edge - fast follow
- V8 uses TurboFan’s advanced type analysis to recover the same information
  - Within ~X% of custom solution on most benchmarks
  - No inter-procedural optimizations
  - Crossover with optimizing normal JavaScript
- V8 can validate asm.js subset and internally translate to WebAssembly

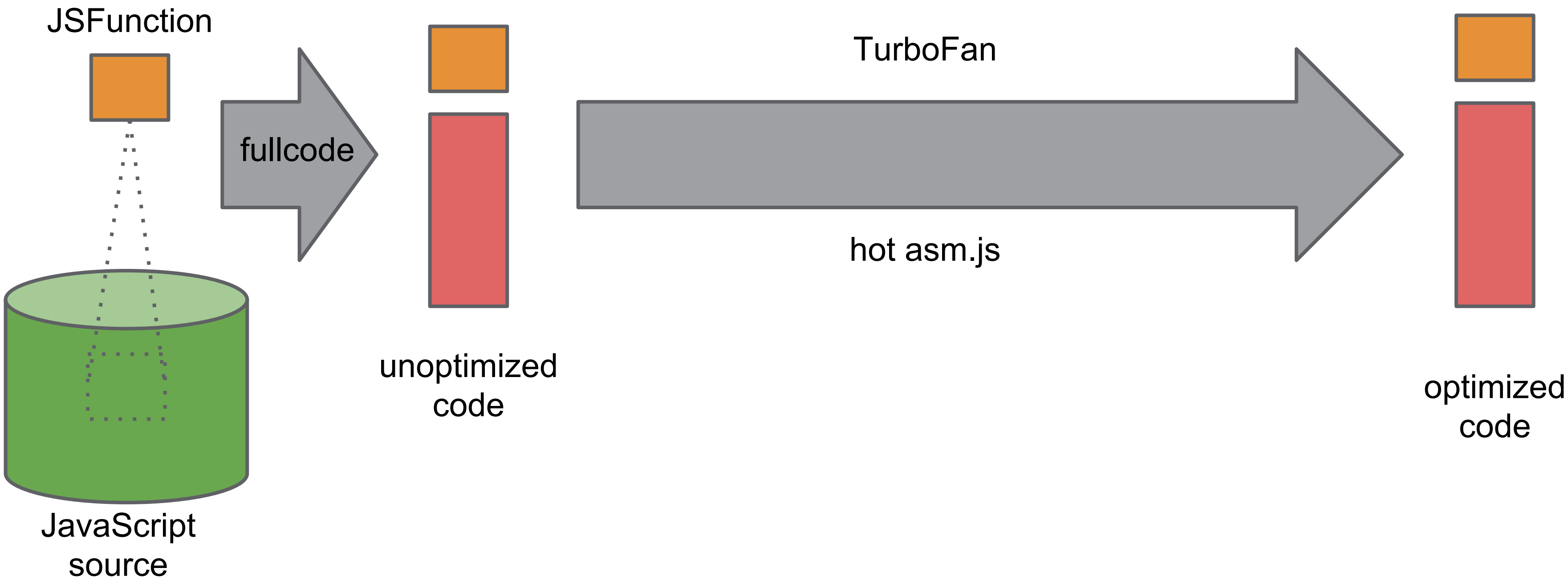
What is WebAssembly?

- A compilation target for native
  - C/C++, other languages -> WASM
- A new capability for the web
  - More than just compressed asm.js
  - float32, int64, threads*, SIMD*
- A complement to JavaScript
  - interface to/from JS code
  - integrate with WebAPIs
- Performance guarantee (ish)
  - Fast calling conventions
  - no boxing, no GC
  - AOT

What is WebAssembly *not*?

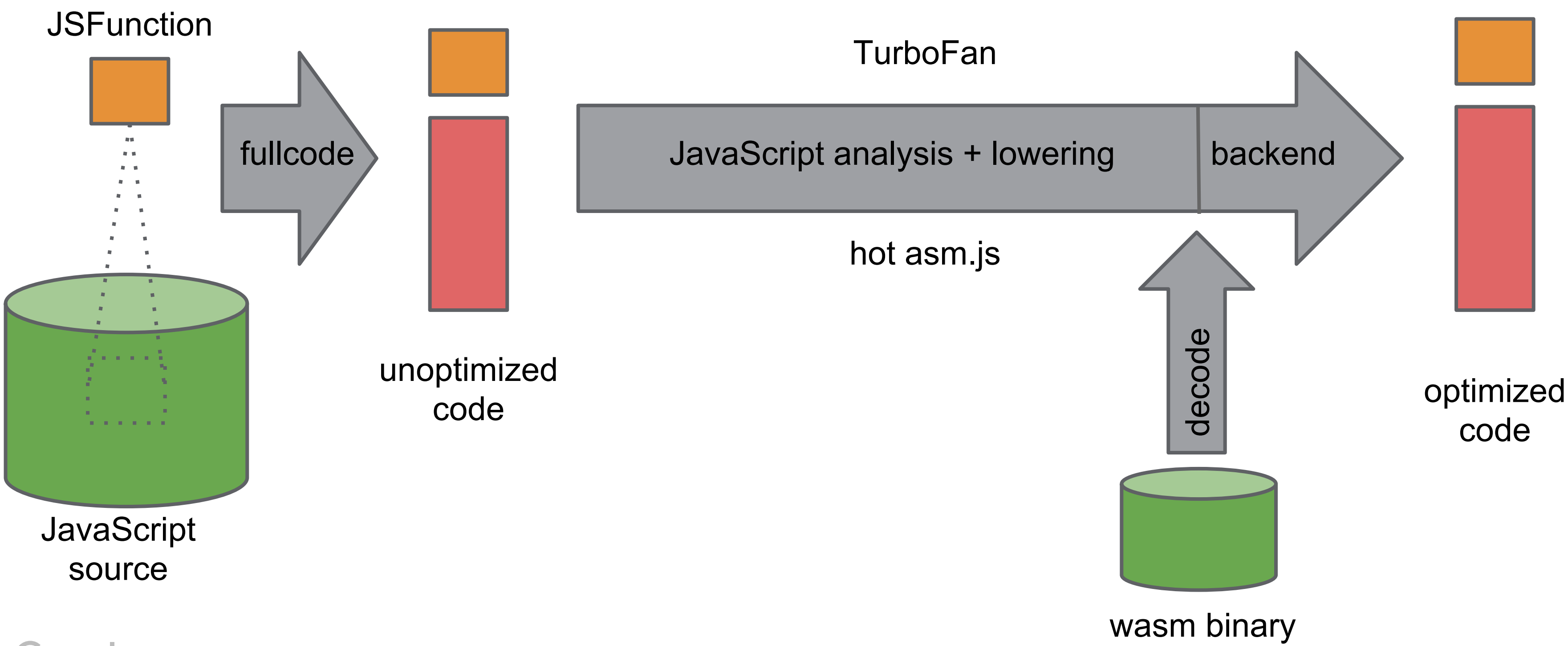
- A value judgment about languages
  - JavaScript vs C++ vs Java vs Dart
- The backend of some C compiler
  - LLVM bitcode, gcc GIMPLE, sea of nodes
- A programming language
  - generated and manipulated by tools
- A separate VM within Chrome
  - instead: built on TurboFan and V8

V8 Pipeline Design (asm.js)

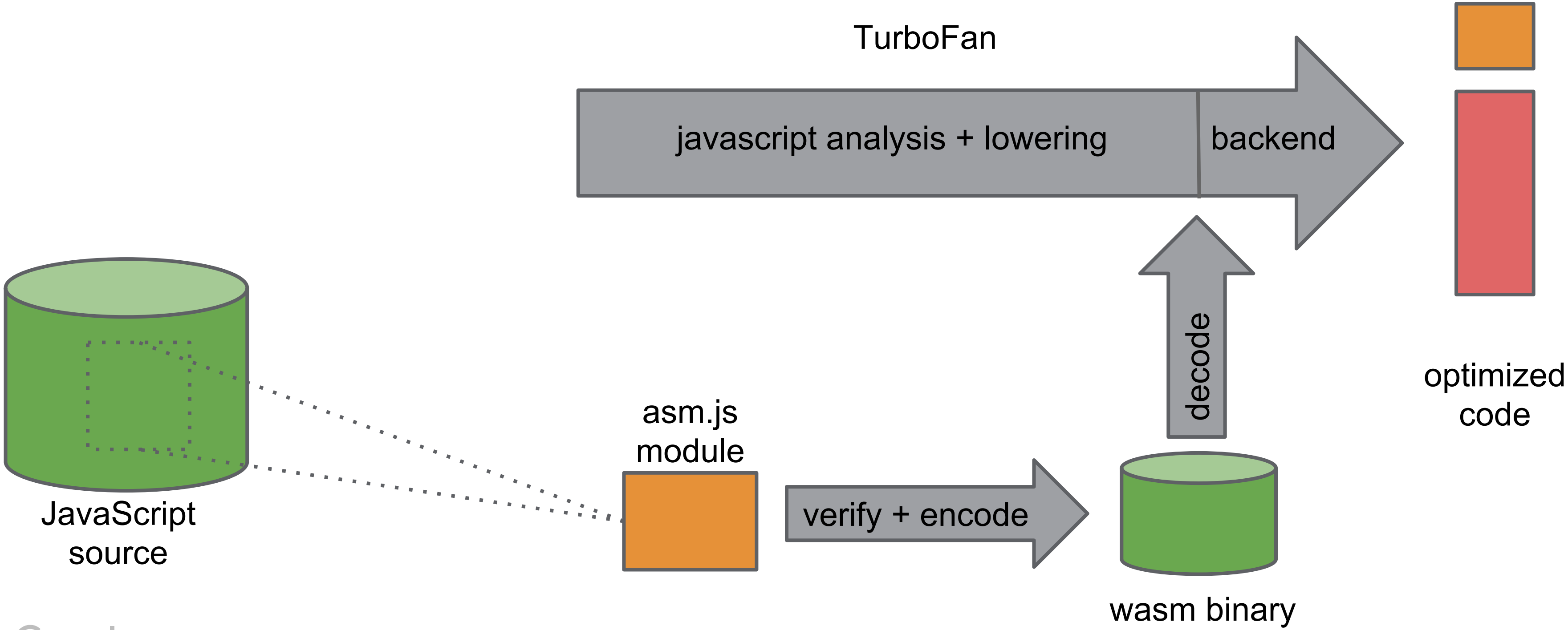


Google

V8 Pipeline Design + WASM



V8 Pipeline Design + asm.js + WASM



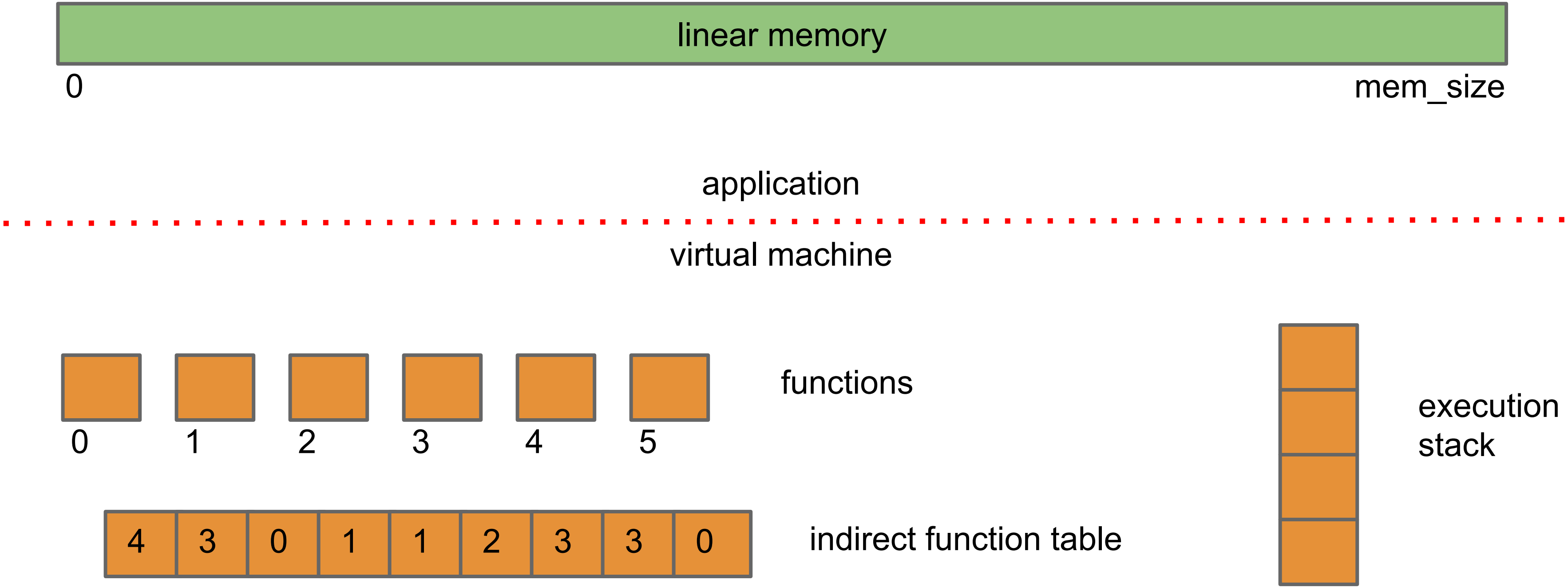
Google

WebAssembly in a nutshell

- Data Types
  - `void i32 i64 f32 f64`
- Functions
  - Flat, single global table
  - Static binding
  - Indirect calls through table
- State: linear memory
  - large, bounds-checked array
- Trusted execution stack

- Data Operations
  - `i32: + - * / % << >> >>> etc`
  - `i64: + - * / % << >> >>> etc`
  - `f32: + - * / sqrt ceil floor`
  - `f64: + - * / sqrt ceil floor`
  - `conversions`
  - `load store`
  - `call_direct call_indirect`
- Structured Control Flow
  - `if loop block br switch`

WebAssembly trusted and untrusted state



linear memory

0

mem_size

application

virtual machine

functions

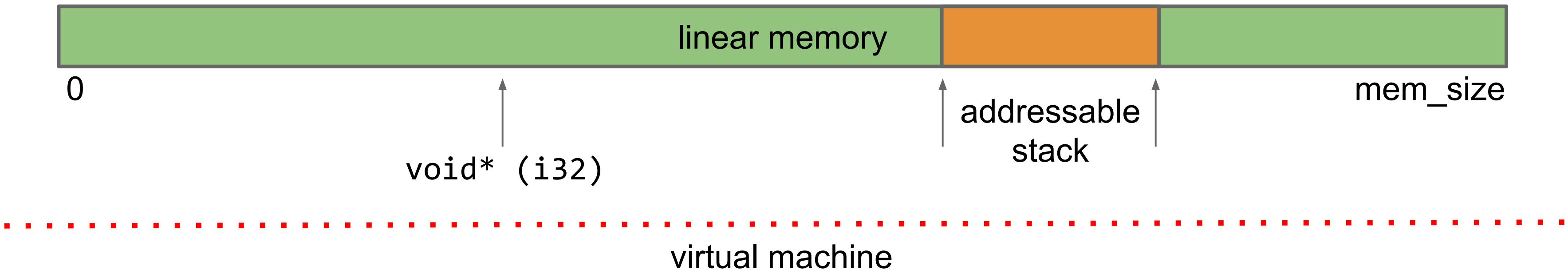
0 1 2 3 4 5

4	3	0	1	1	2	3	3	0

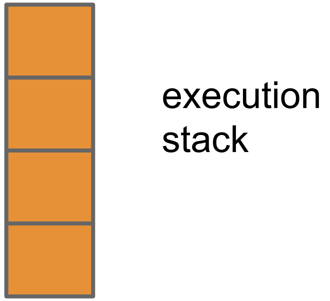
indirect function table

execution stack

Compiling C/C++ to WebAssembly



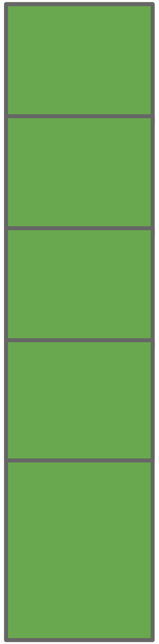
- C compiler translates pointers to `i32` indices
- C compiler places addressable stack in memory
- asm.js bounds checks (~5% overhead)



WebAssembly binary code

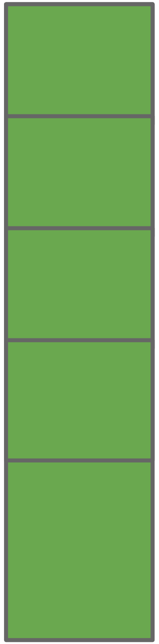
- Goals:
  - compact => smaller than minified JS
  - easy to verify => one linear pass
  - easy to compile => one linear pass to construct IR or baseline JIT
  - extensible => anticipate new bytecodes and types
- Design:
  - AST-based post-order encoding of function bodies
  - All AST nodes are expressions
  - Optional application-specified opcode table

Module structure



- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
- Initialized data

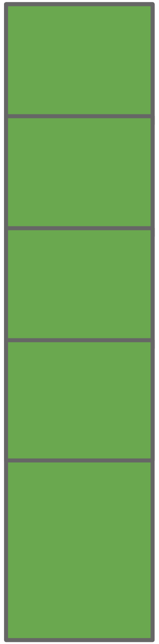
Module structure



- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
- Initialized data

```
min_size = 16mb
max_size = 1gb
exported_to_js = false
```

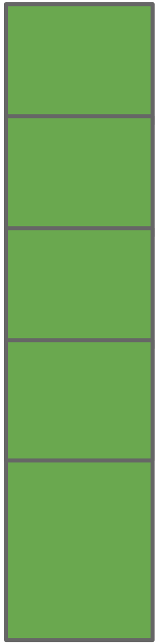
Module structure



- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
- Initialized data

```
(i32, i32) -> i32
(i64, i32) -> i32
(f32) -> i32
```

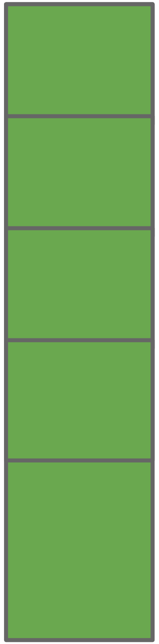
Module structure



- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
- Initialized data

```
myfunc:
   <sig>
   <flags>
   <code>
```

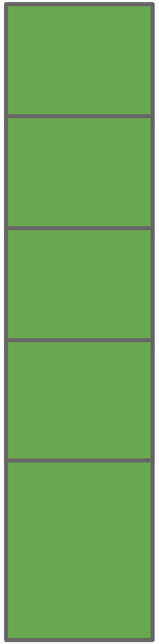
Module structure



- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
- Initialized data

```
0: myfunc1
1: myfunc2
2: myfunc2
```

Module structure



- Memory declaration
- Function signatures
- Functions
- Indirect Function Table
- Initialized data

```
0x01099de8: <data>
0x0f0a9c12: <data>
0x00034a00: <data>
```

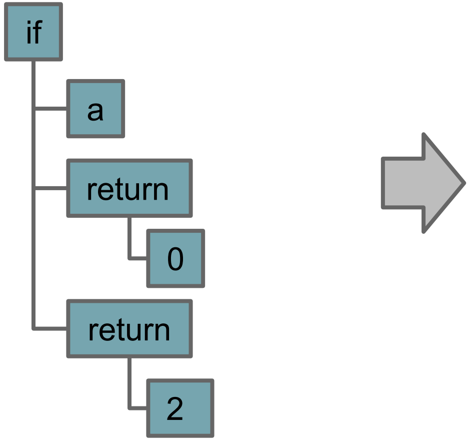
Bytecode => TurboFan

- One Linear pass to construct sea of nodes
    - SSA environment tracks control and effect dependencies
    - Stack of if, blocks, and loops
    - Conservative phi insertion at loop headers
    - Reduction steps generate nodes in the IR graph
- Machine-level graph
    - Immediately suitable for code generation
    - Correct sea-of-nodes can go through scheduling
    - Can apply machine-level and machine-independent optimizations
- Fast calling convention
    - No boxing of double arguments
    - All arguments in registers
    - No extra JSFunction / context arguments

Pre-order encoding of an AST

```
if (a) return 0; else return 2;
```

```
return a?0:2
```



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

0	ret
1	if
2	local
3	#0
4	iconst
5	#0
6	iconst
7	#2

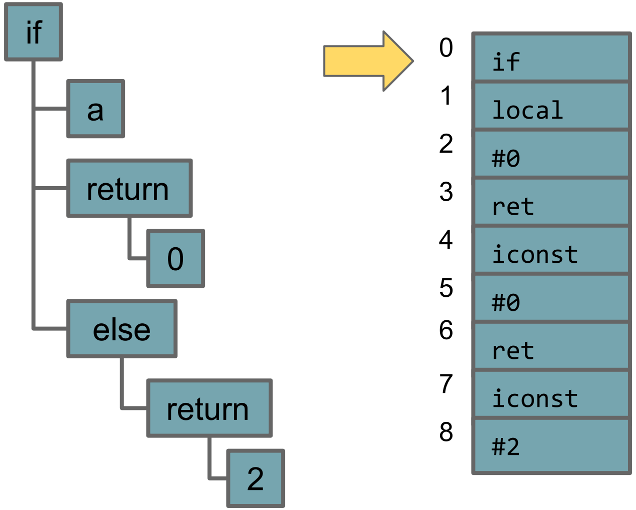
0	ret
1	if
2	local0
3	iconst0
4	iconst2

0	if
1	local0
2	iconst0
3	iconst2

Decoding preorder to IR

```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫

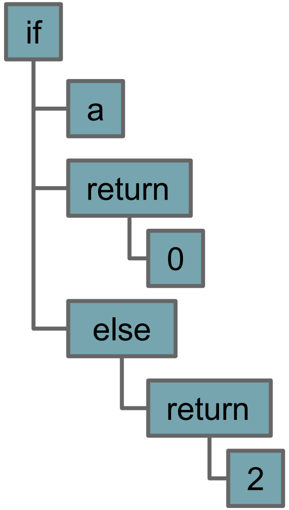


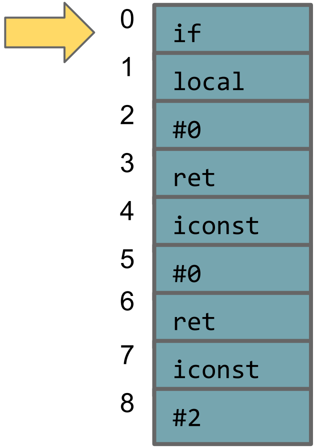
0	`if`
1	`local`
2	`#0`
3	`ret`
4	`iconst`
5	`#0`
6	`ret`
7	`iconst`
8	`#2`

Decoding preorder to IR

```
if (a) return 0; else return 2;
```

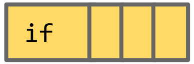
unfinished 🟨 finished 🟫





0	`if`
1	`local`
2	`#0`
3	`ret`
4	`iconst`
5	`#0`
6	`ret`
7	`iconst`
8	`#2`

Production stack

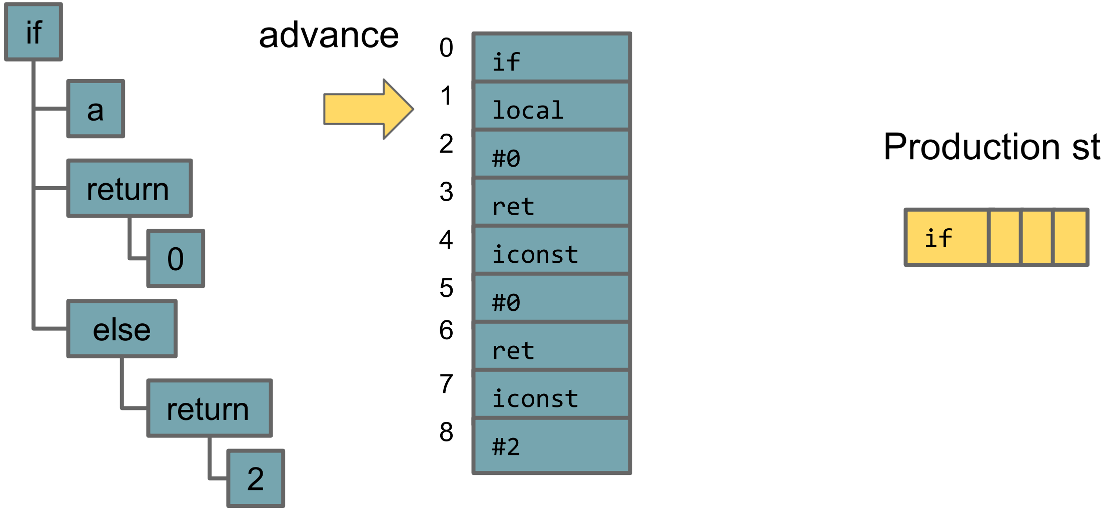


shift

Decoding preorder to IR

```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫

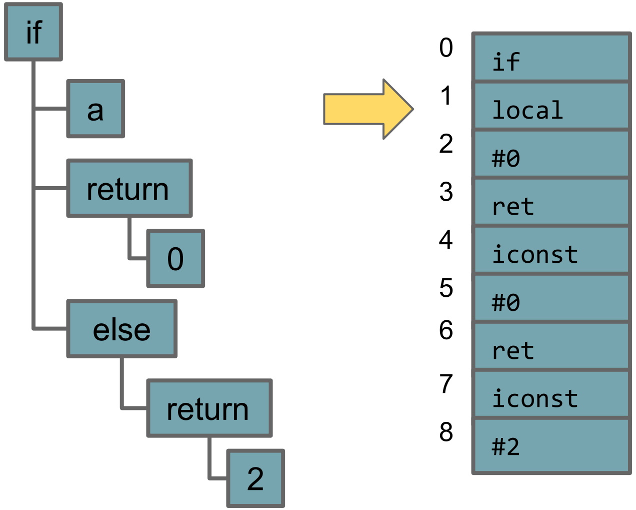


0	`if`
1	`local`
2	`#0`
3	`ret`
4	`iconst`
5	`#0`
6	`ret`
7	`iconst`
8	`#2`

Decoding preorder to IR

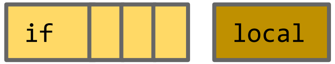
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Production stack

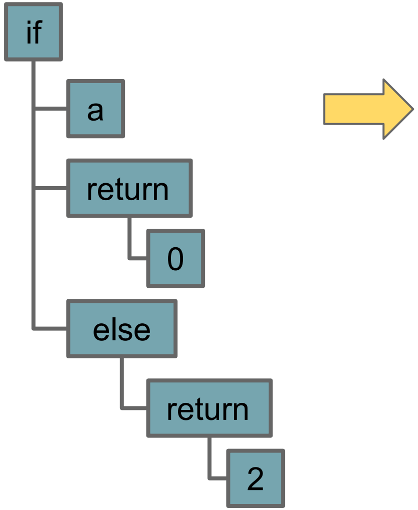


shift

Decoding preorder to IR

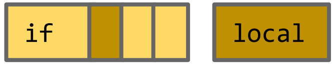
```
if (a) return 0; else return 2;
```

unfinished □ finished ■



0	`if`
1	`local`
2	`#0`
3	`ret`
4	`iconst`
5	`#0`
6	`ret`
7	`iconst`
8	`#2`

Production stack

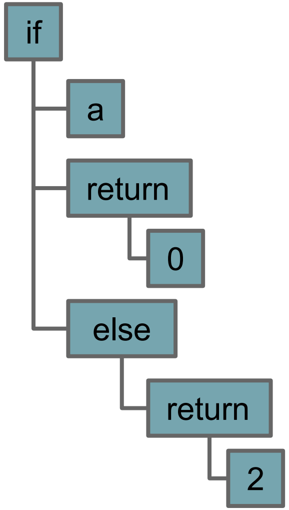


reduce

Decoding preorder to IR

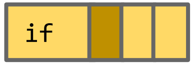
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Production stack

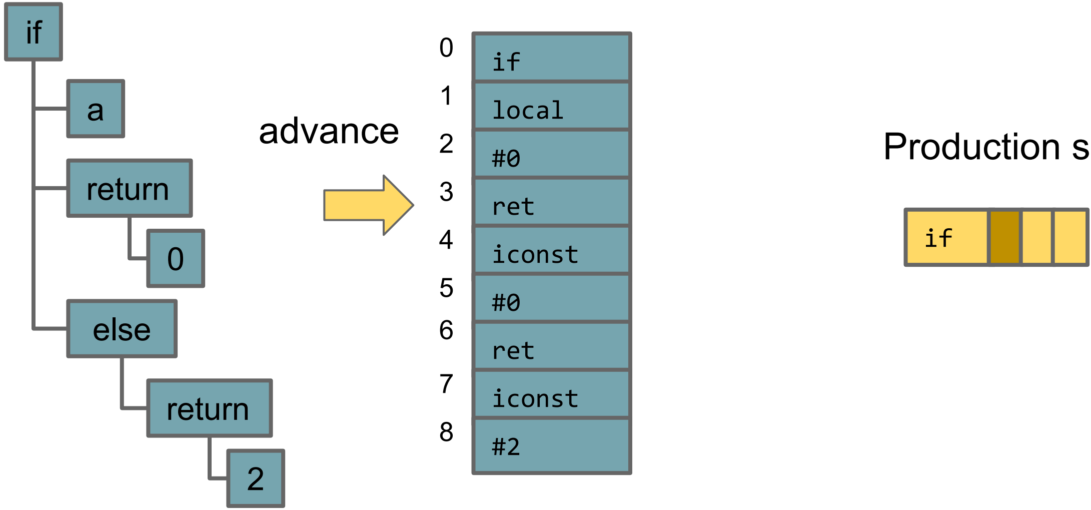


reduce

Decoding preorder to IR

```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



if
a
return
0
else
return
2

advance

0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

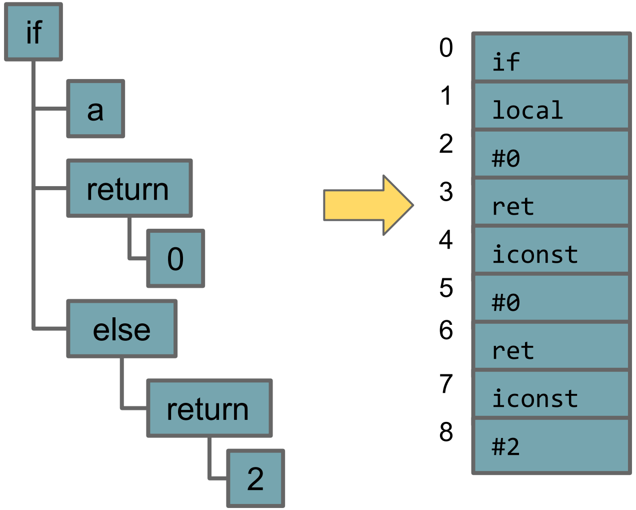
Production stack

if

Decoding preorder to IR

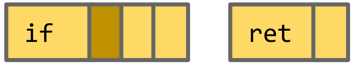
```
if (a) return 0; else return 2;
```

unfinished □ finished ■



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Production stack

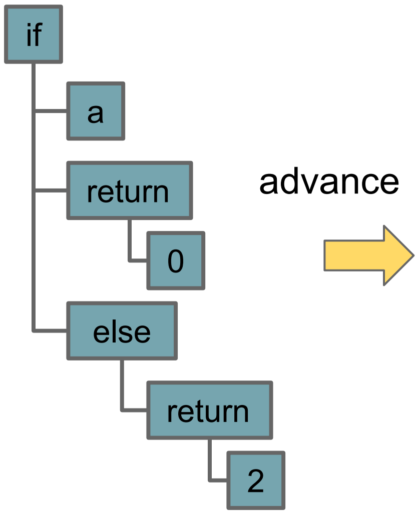


shift

Decoding preorder to IR

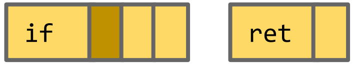
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

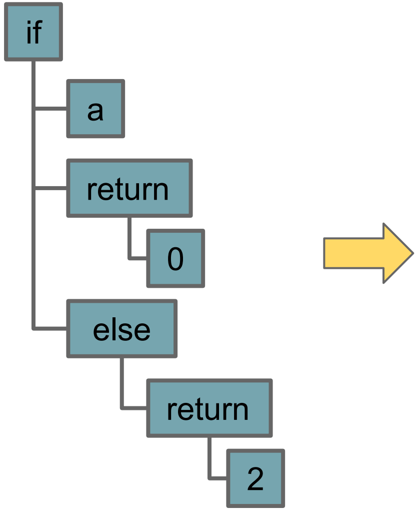
Production stack



Decoding preorder to IR

```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



if

a

return

0

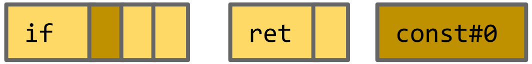
else

return

2

0	`if`
1	`local`
2	`#0`
3	`ret`
4	`iconst`
5	`#0`
6	`ret`
7	`iconst`
8	`#2`

Production stack



`if`

`ret`

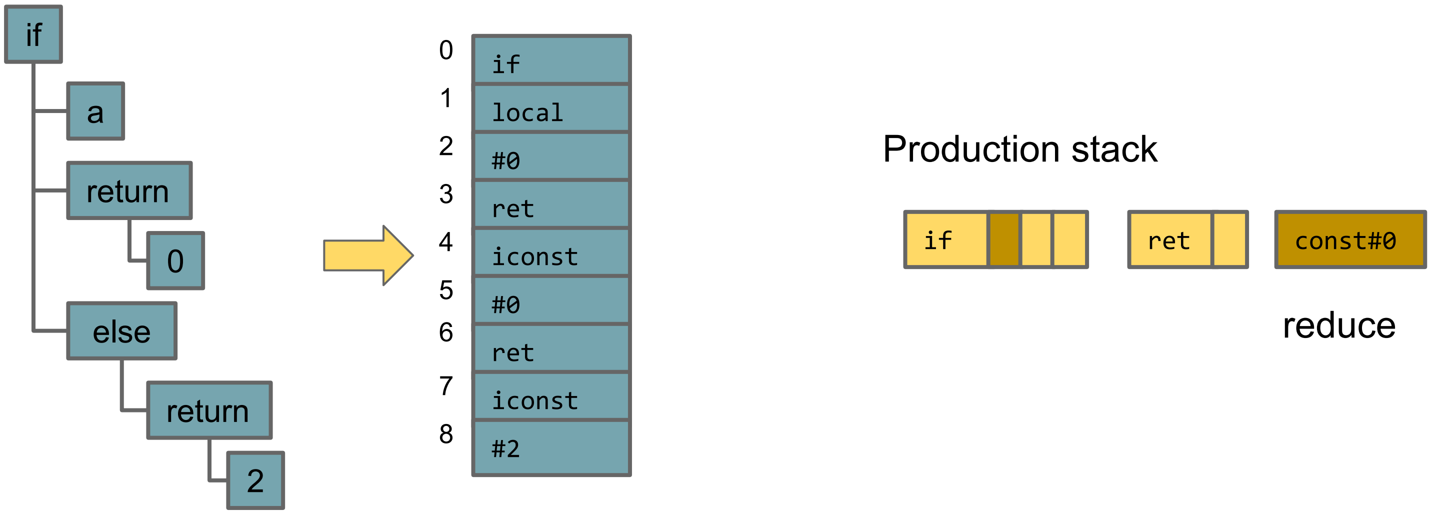
`const#0`

shift

Decoding preorder to IR

```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

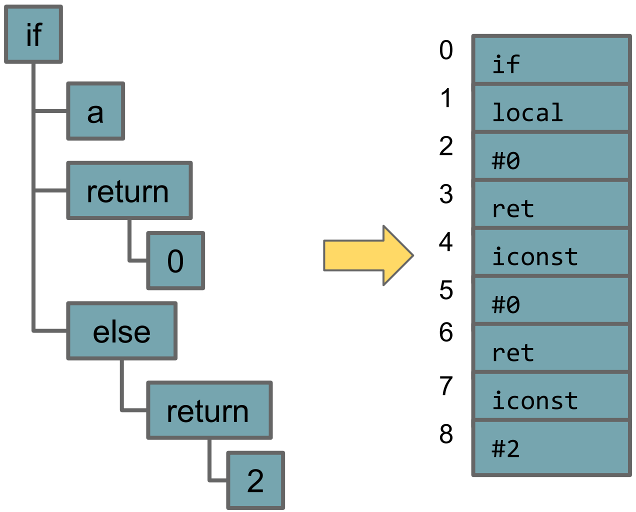
Production stack

reduce

Decoding preorder to IR

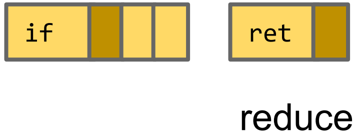
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

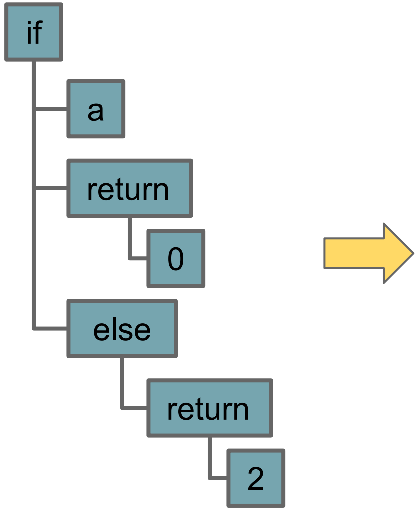
Production stack



Decoding preorder to IR

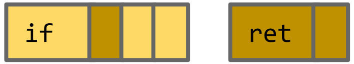
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	`if`
1	`local`
2	`#0`
3	`ret`
4	`iconst`
5	`#0`
6	`ret`
7	`iconst`
8	`#2`

Production stack

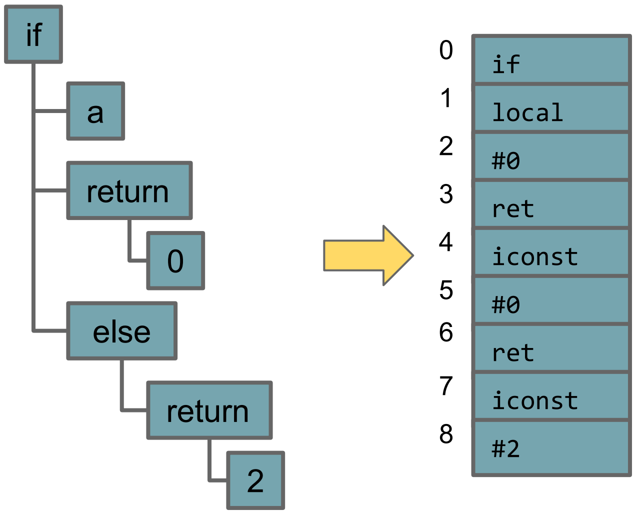


reduce

Decoding preorder to IR

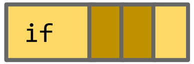
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Production stack

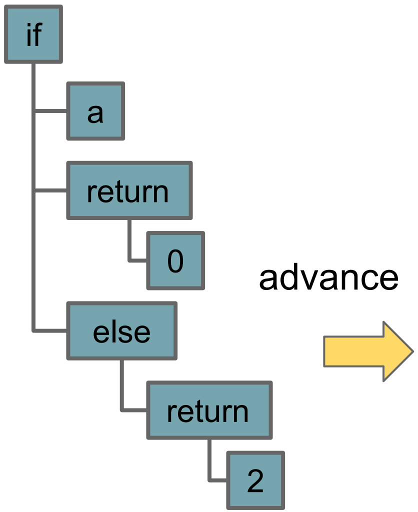


reduce

Decoding preorder to IR

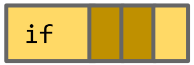
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	`if`
1	`local`
2	`#0`
3	`ret`
4	`iconst`
5	`#0`
6	`ret`
7	`iconst`
8	`#2`

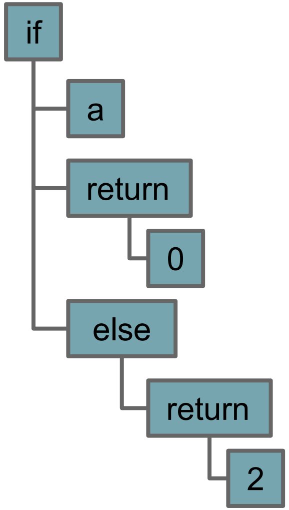
Production stack



Decoding preorder to IR

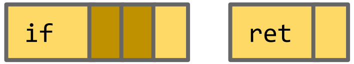
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	`if`
1	`local`
2	`#0`
3	`ret`
4	`iconst`
5	`#0`
6	`ret`
7	`iconst`
8	`#2`

Production stack

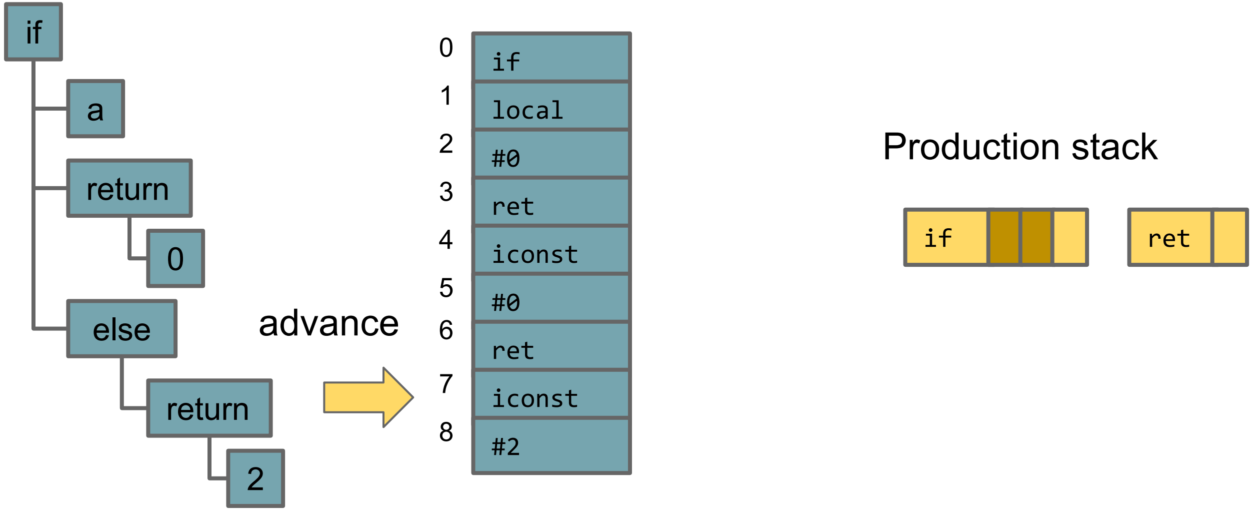


shift

Decoding preorder to IR

```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫

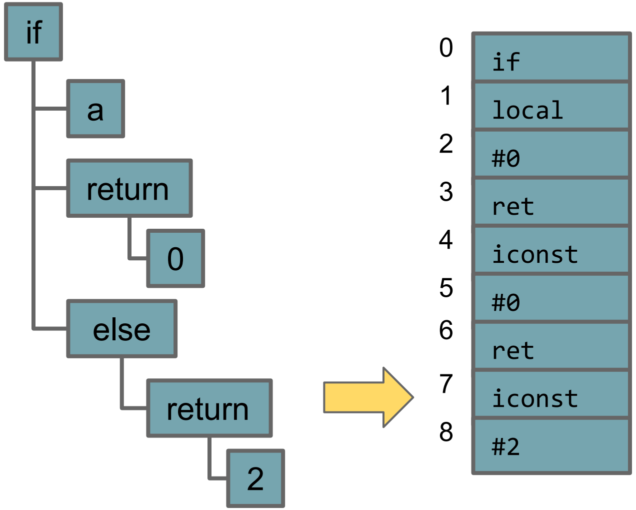


0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Decoding preorder to IR

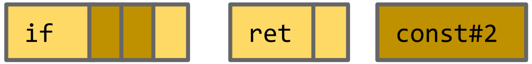
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Production stack

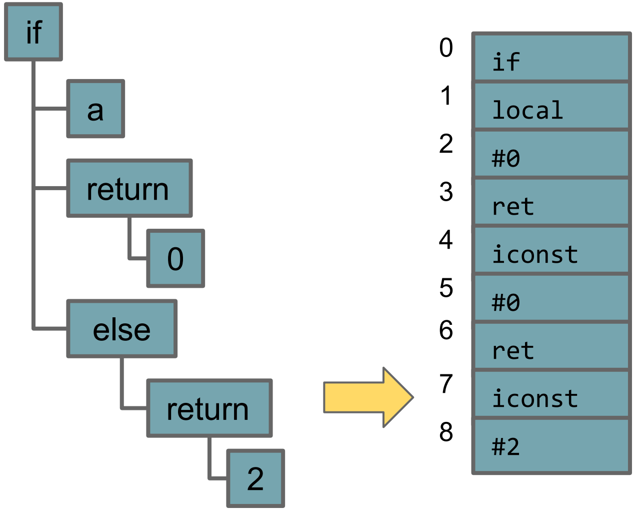


shift

Decoding preorder to IR

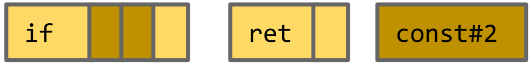
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Production stack

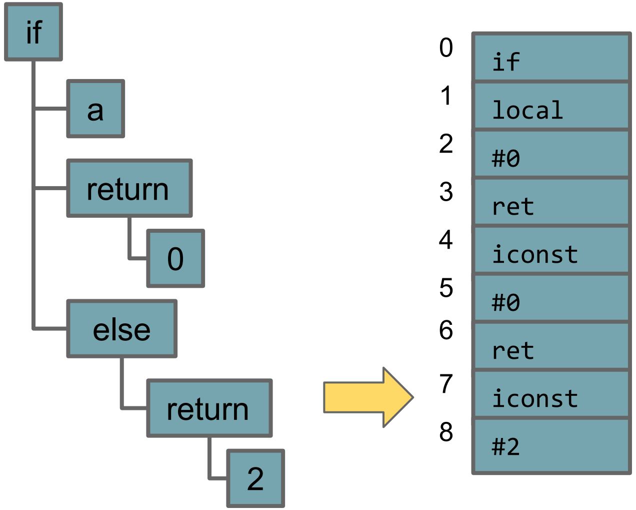


reduce

Decoding preorder to IR

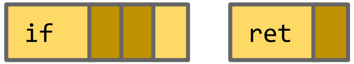
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Production stack

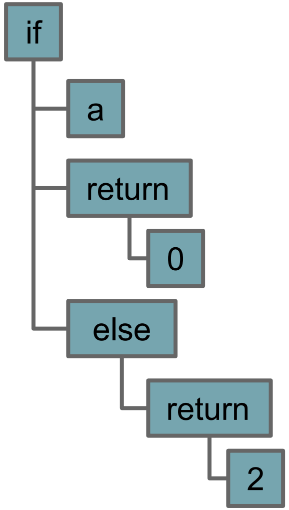


reduce

Decoding preorder to IR

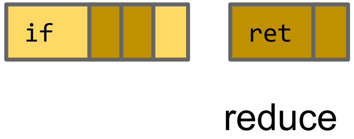
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	`if`
1	`local`
2	`#0`
3	`ret`
4	`iconst`
5	`#0`
6	`ret`
7	`iconst`
8	`#2`

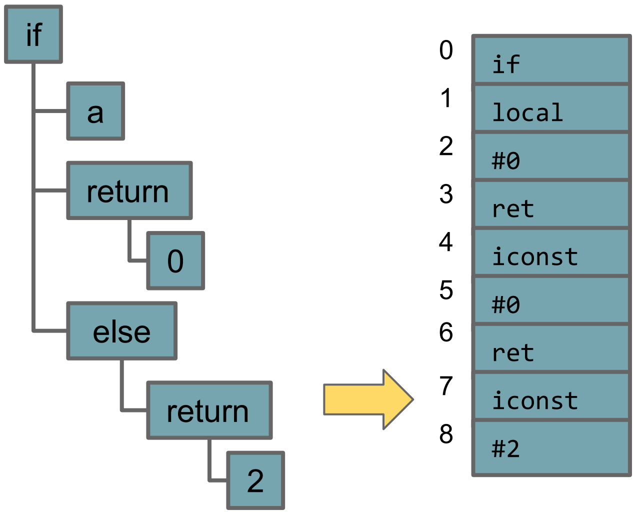
Production stack



Decoding preorder to IR

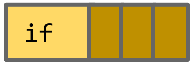
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Production stack

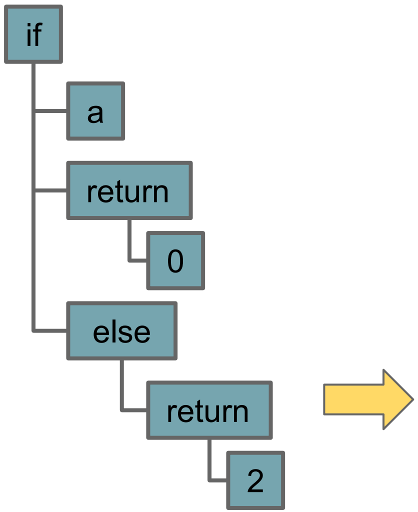


reduce

Decoding preorder to IR

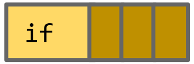
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Production stack

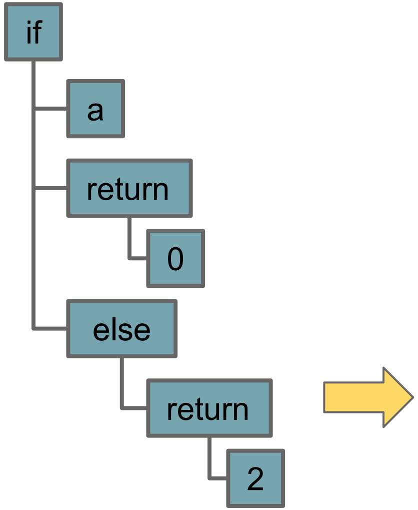


reduce

Decoding preorder to IR

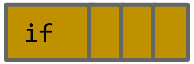
```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Production stack

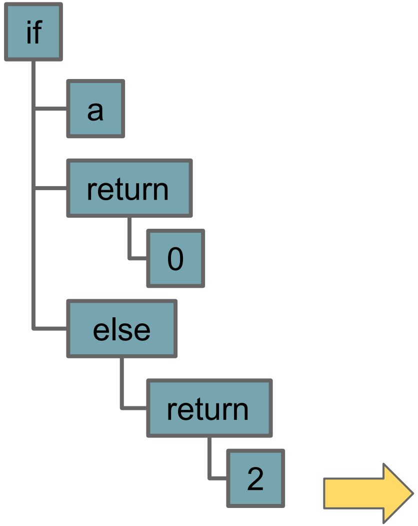


reduce

Decoding preorder to IR

```
if (a) return 0; else return 2;
```

unfinished 🟨 finished 🟫



finish

0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

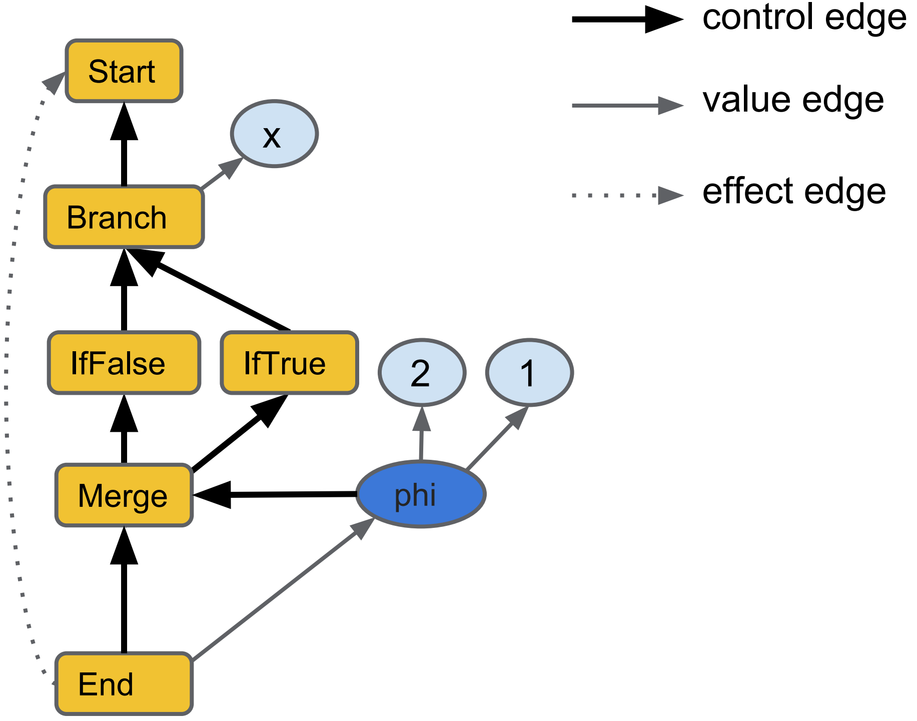
Production stack

Bytecode ⇒ TurboFan

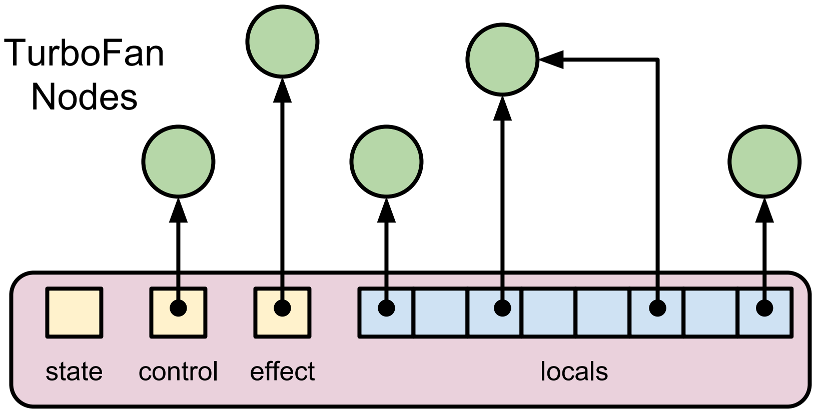
- One Linear pass to construct sea of nodes
  - SSA environment tracks control and effect dependencies
  - Stack of if, blocks, and loops
  - Conservative phi insertion at loop headers
  - Reduction steps generate nodes in the IR graph
- Machine-level graph
  - Immediately suitable for code generation
  - Correct sea-of-nodes can go through scheduling
  - Can apply machine-level and machine-independent optimizations

TurboFan graph example

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

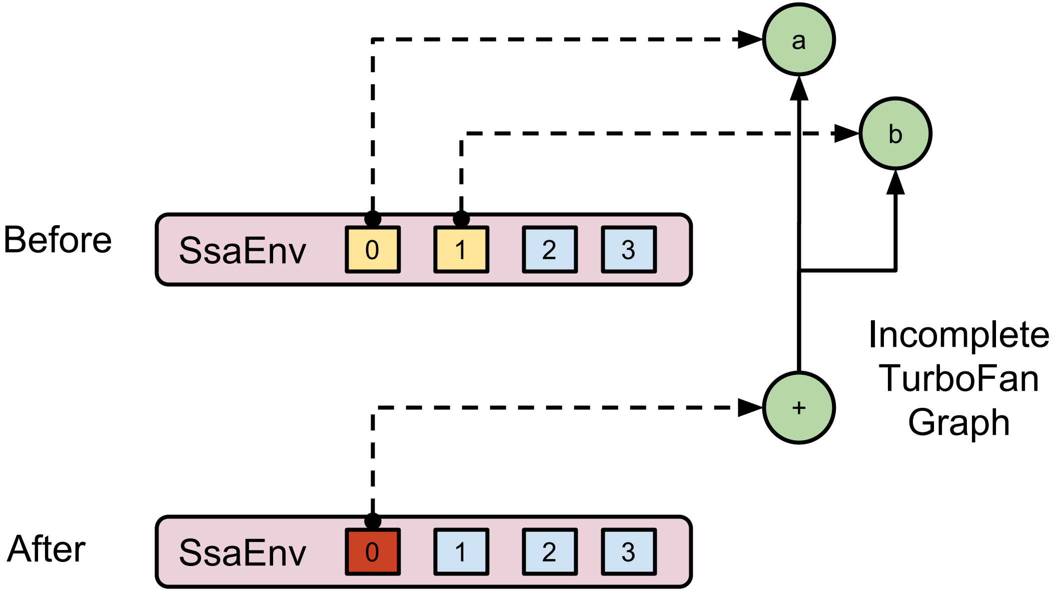


TurboFan SSA Environment



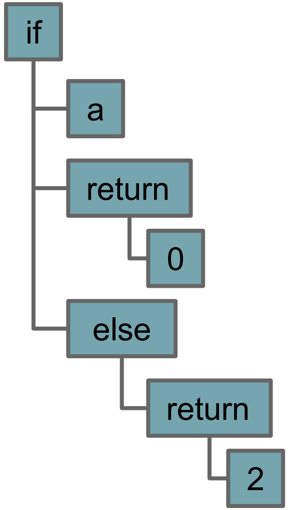
Using the SSA environment

```
bytecode: local[0] = local[0] + local[1]
```



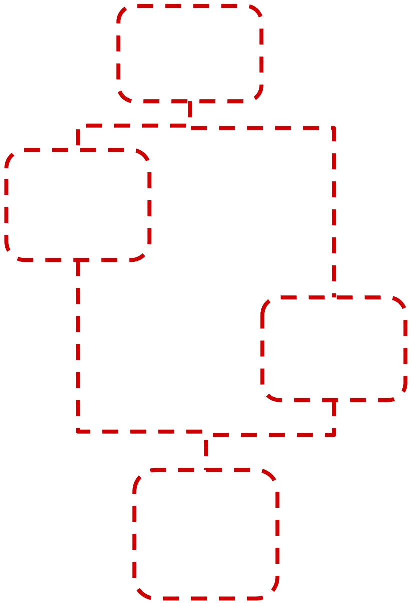
Minimal SSA Renaming in one pass

```
if (a) return 0; else return 2;
```



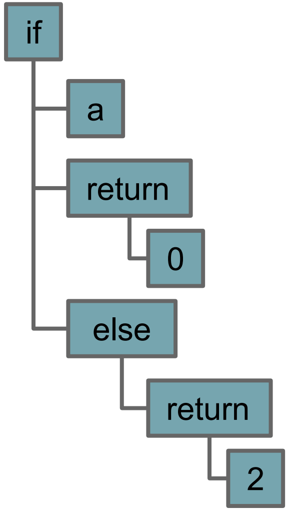
0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Virtual CFG



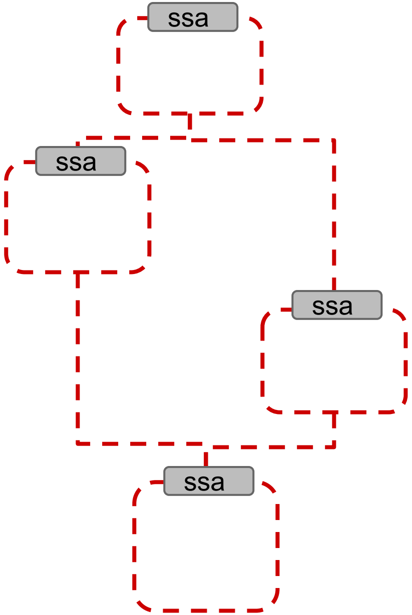
Minimal SSA Renaming in one pass

```
if (a) return 0; else return 2;
```



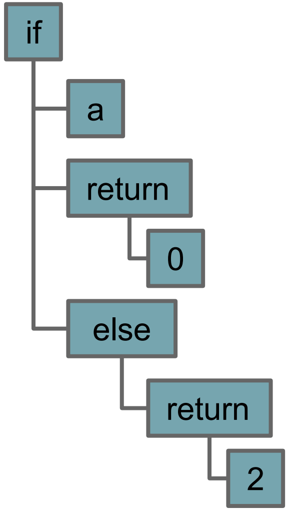
0	if
1	local
2	#0
3	ret
4	iconst
5	#0
6	ret
7	iconst
8	#2

Virtual CFG



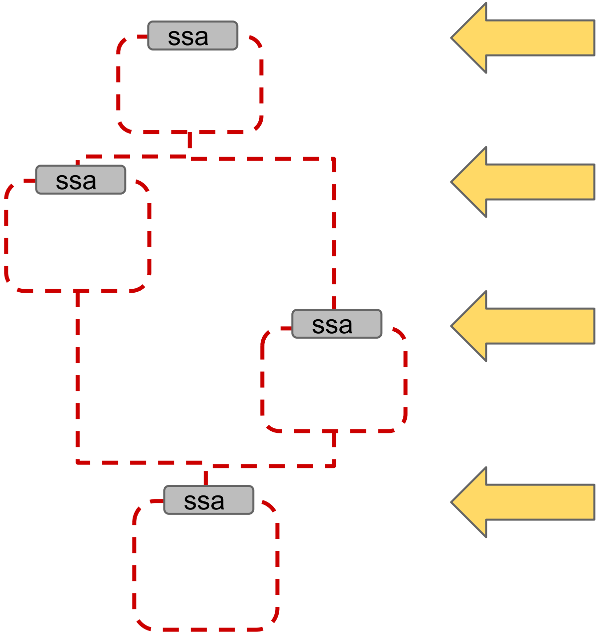
Minimal SSA Renaming in one pass

```
if (a) return 0; else return 2;
```



0	`if`
1	`local`
2	`#0`
3	`ret`
4	`iconst`
5	`#0`
6	`ret`
7	`iconst`
8	`#2`

Virtual CFG



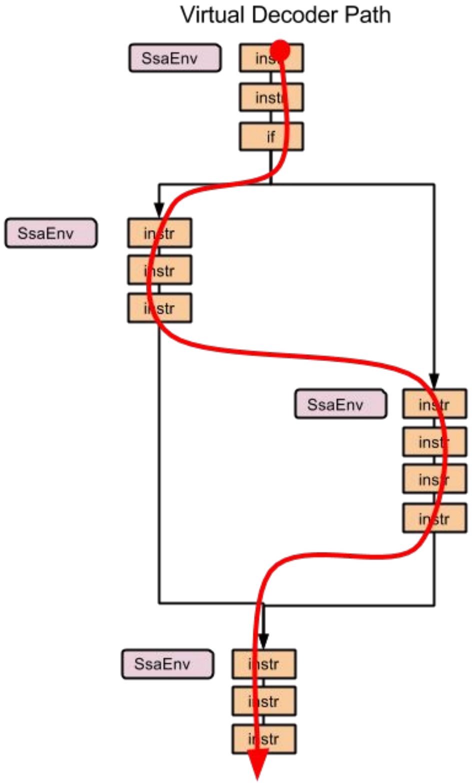
begin `if`

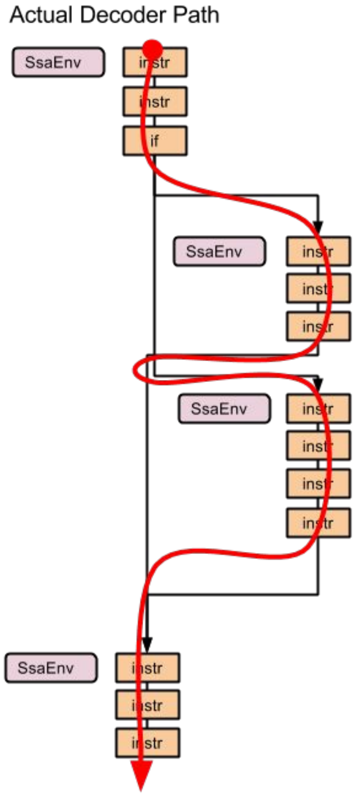
`true` (split)

`false` (split)

end `if` (merge)

Stack of SSA environments





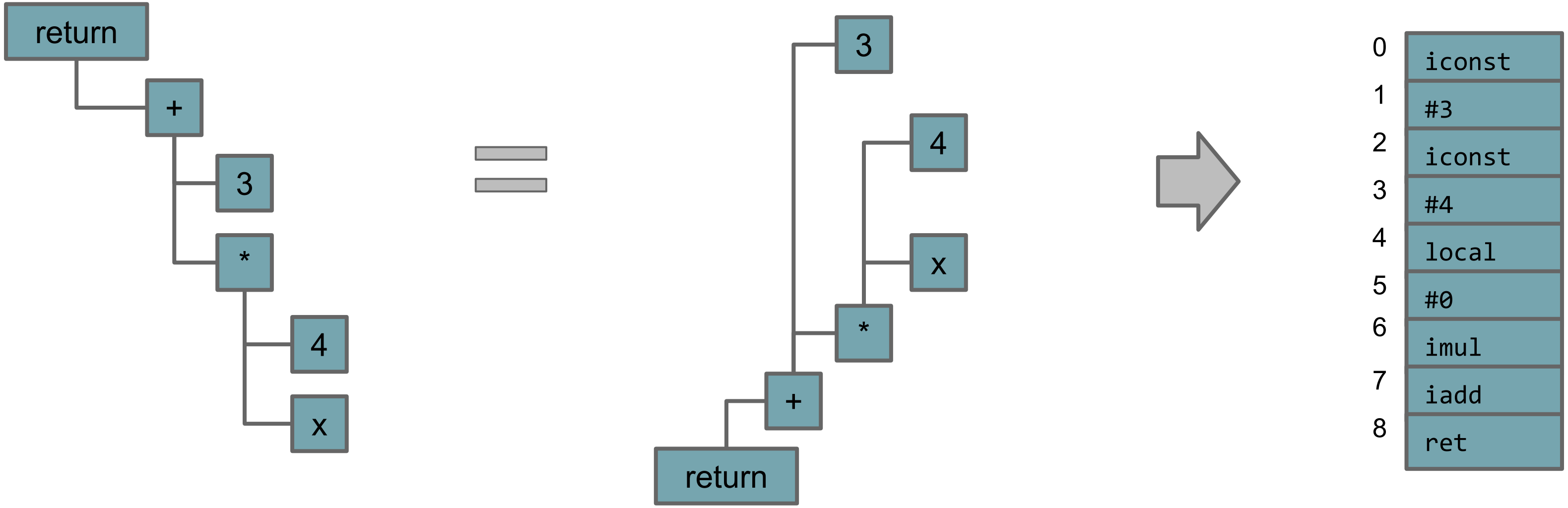
Google

The same great AST: now in postorder!

Function Bodies

Post-order encoding of an AST

```
return 3 + x * 4
```



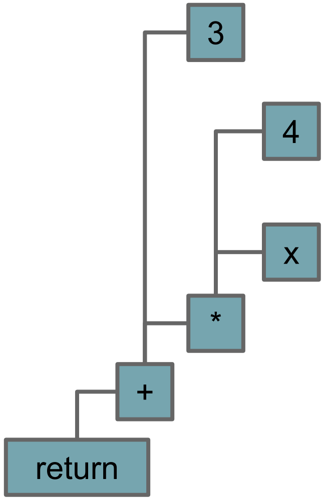
0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

Google

Decoding post-order to an AST

finished 🟨

```
return 3 + x * 4
```

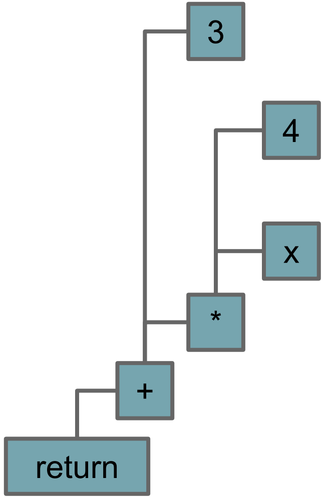


0	`iconst`
1	`#3`
2	`iconst`
3	`#4`
4	`local`
5	`#0`
6	`imul`
7	`iadd`
8	`ret`

Decoding post-order to an AST

```
return 3 + x * 4
```

finished ▨



0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

const#3

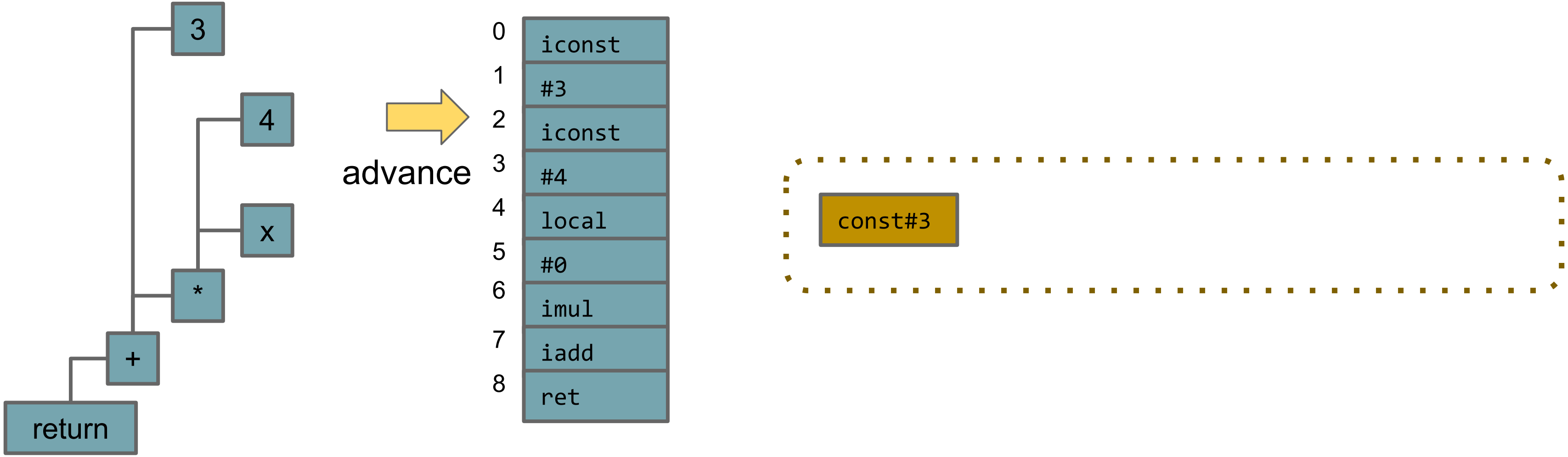
push

Google

Decoding post-order to an AST

```
return 3 + x * 4
```

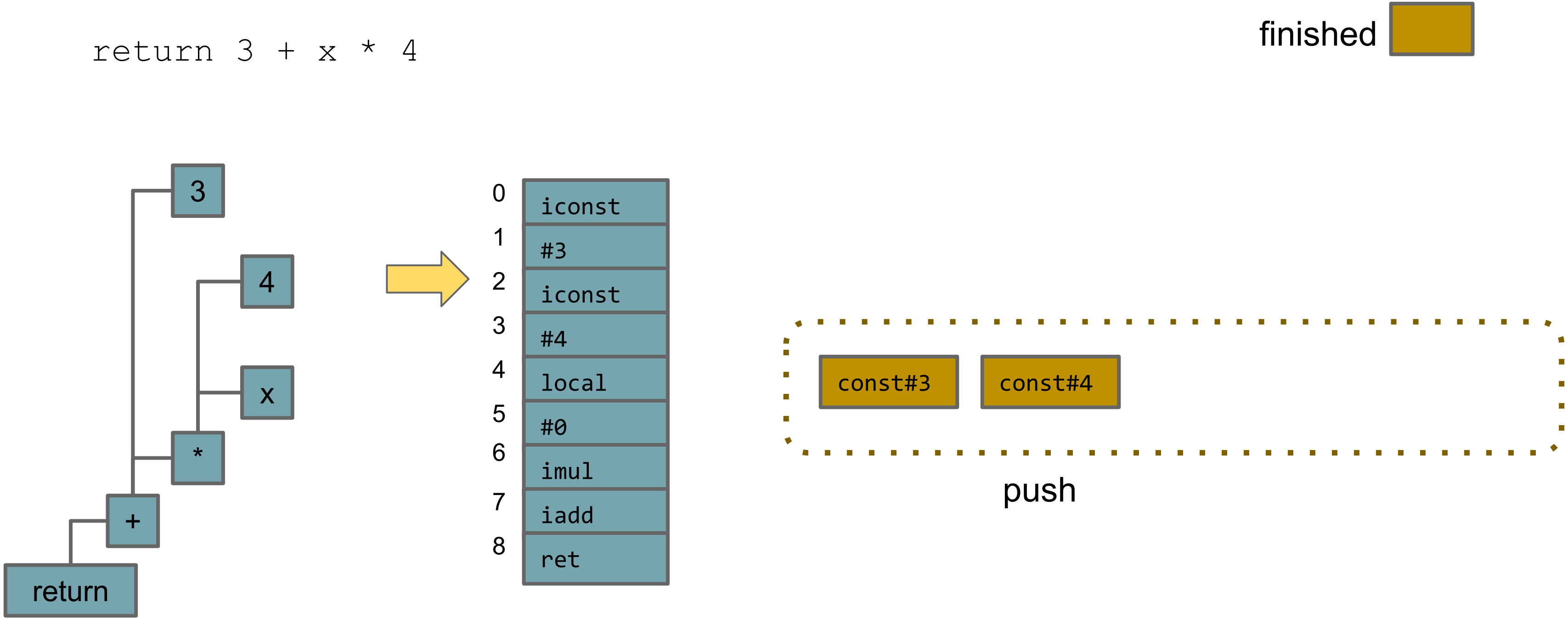
finished 🟫



```
0 iconst
1 #3
2 iconst
3 #4
4 local
5 #0
6 imul
7 iadd
8 ret
```

Decoding post-order to an AST

```
return 3 + x * 4
```

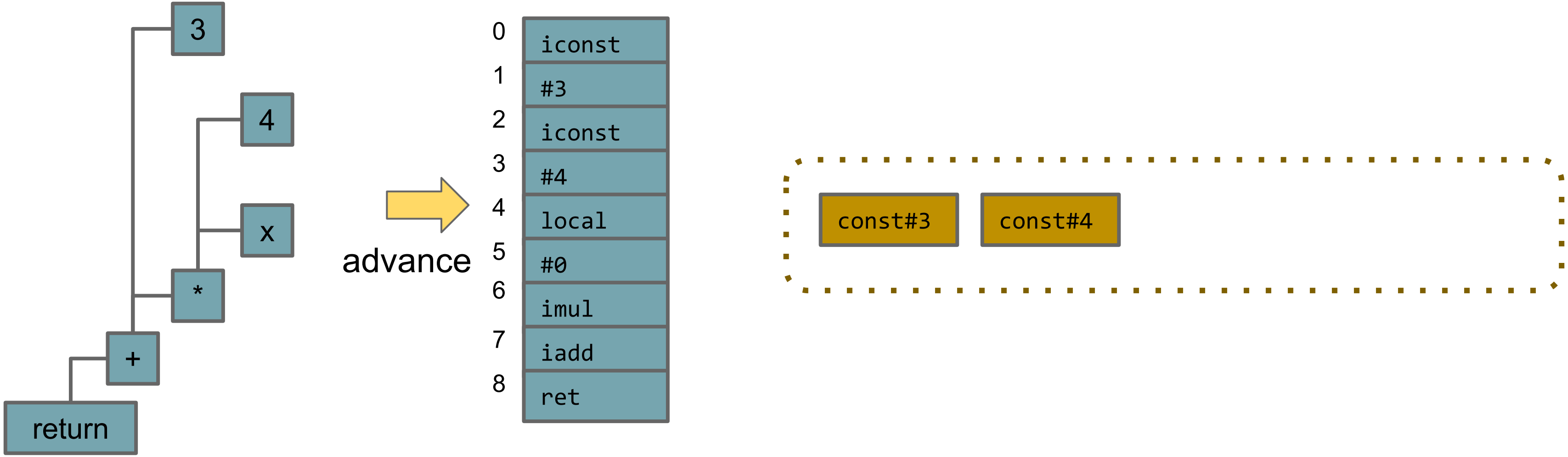


0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

Decoding post-order to an AST

finished 🟨

```
return 3 + x * 4
```

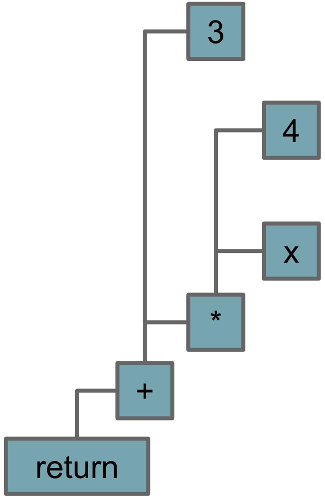


0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

Decoding post-order to an AST

finished ■

```
return 3 + x * 4
```



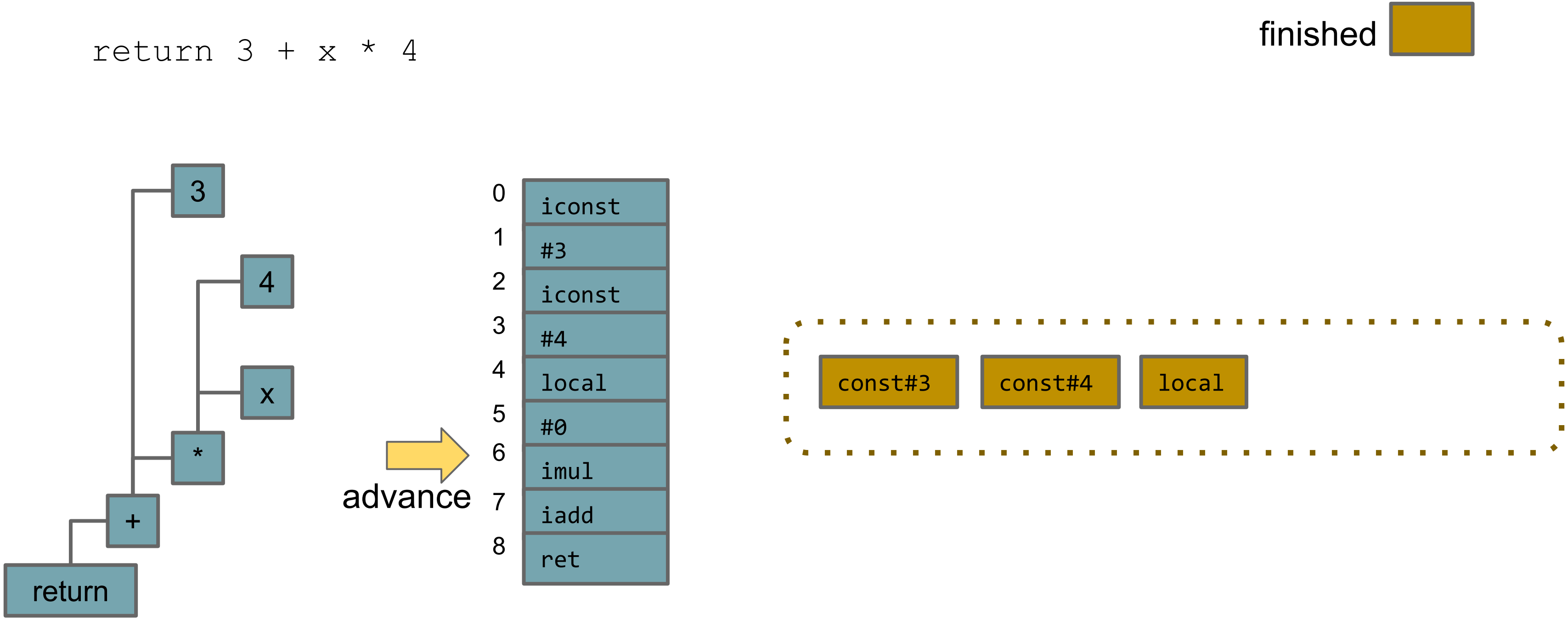
0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

`const#3` `const#4` `local`

push

Decoding post-order to an AST

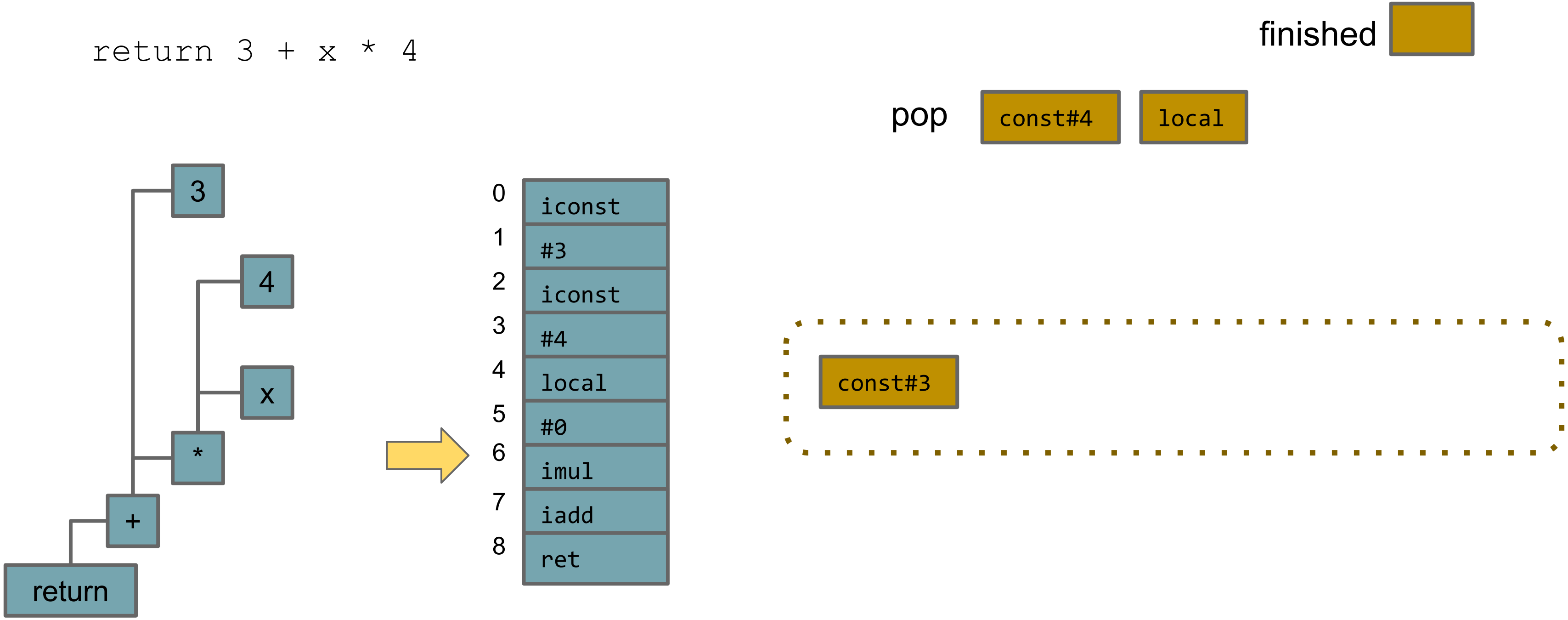
```
return 3 + x * 4
```



0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

Decoding post-order to an AST

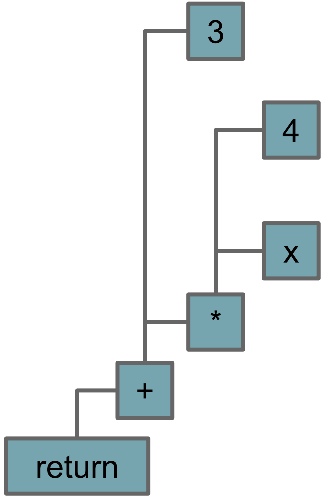
```
return 3 + x * 4
```



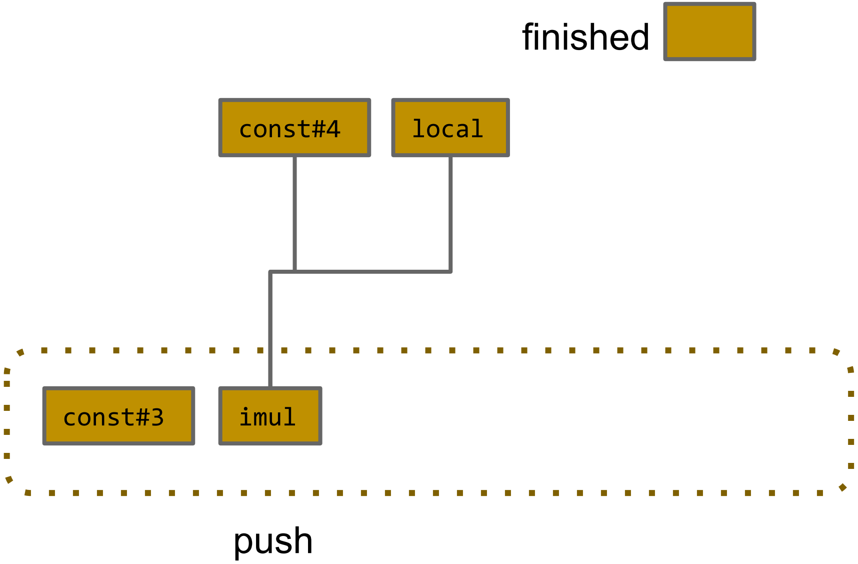
0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

Decoding post-order to an AST

```
return 3 + x * 4
```

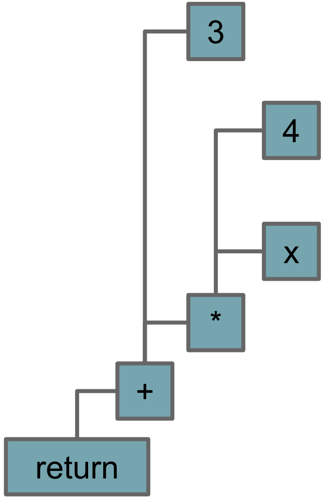


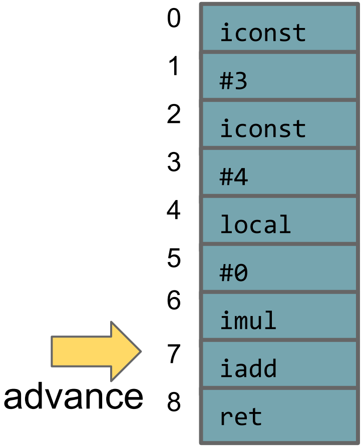
0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret



Decoding post-order to an AST

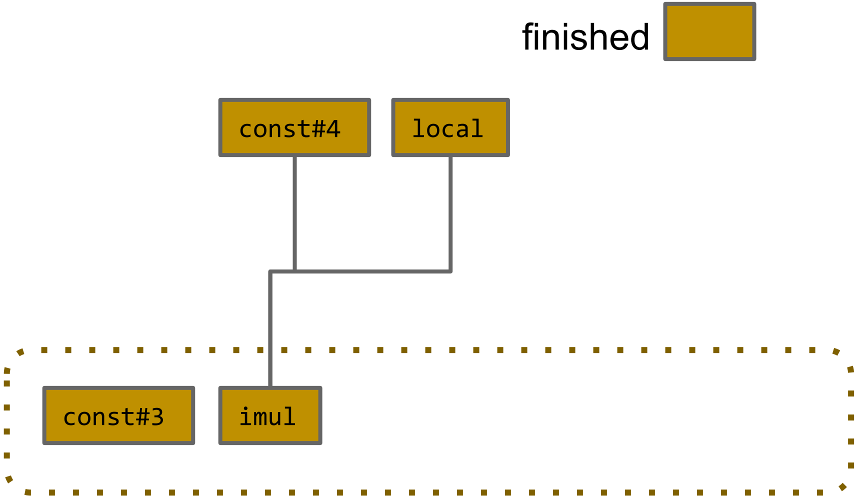
```
return 3 + x * 4
```





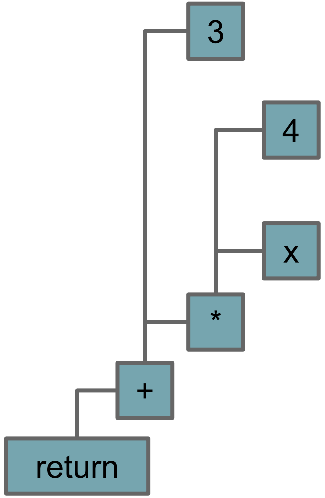
advance

0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

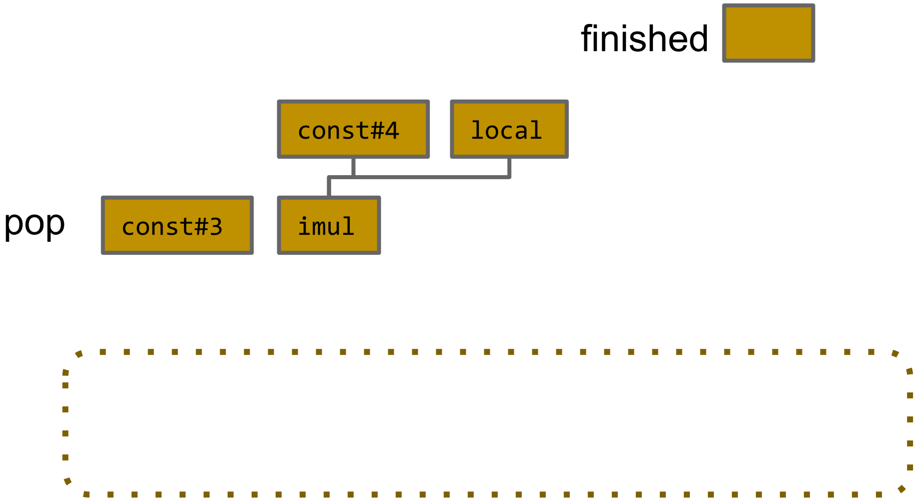


Decoding post-order to an AST

```
return 3 + x * 4
```

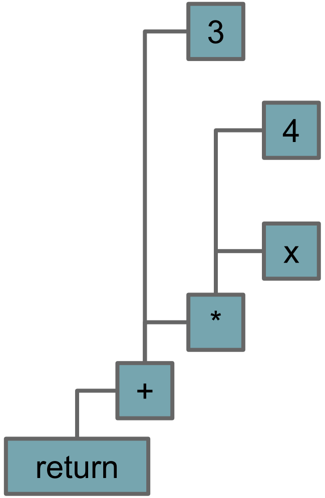


0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

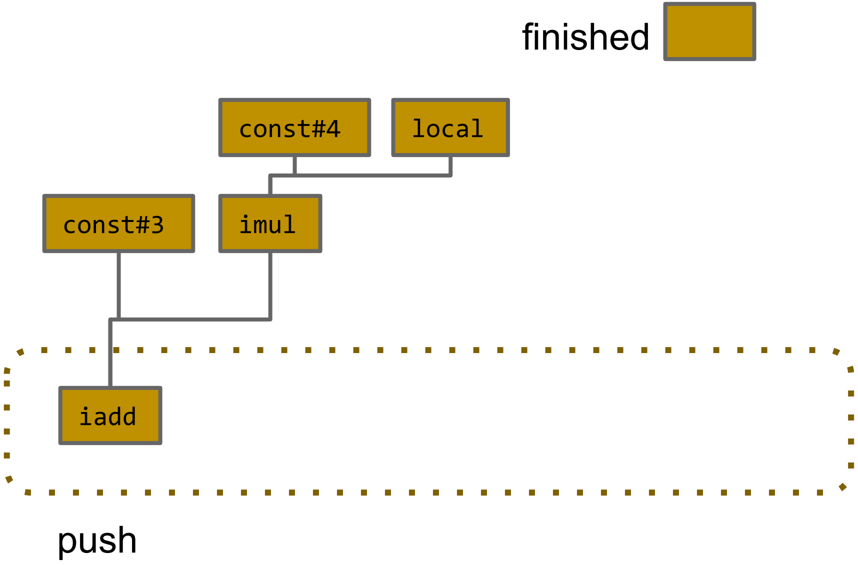


Decoding post-order to an AST

```
return 3 + x * 4
```

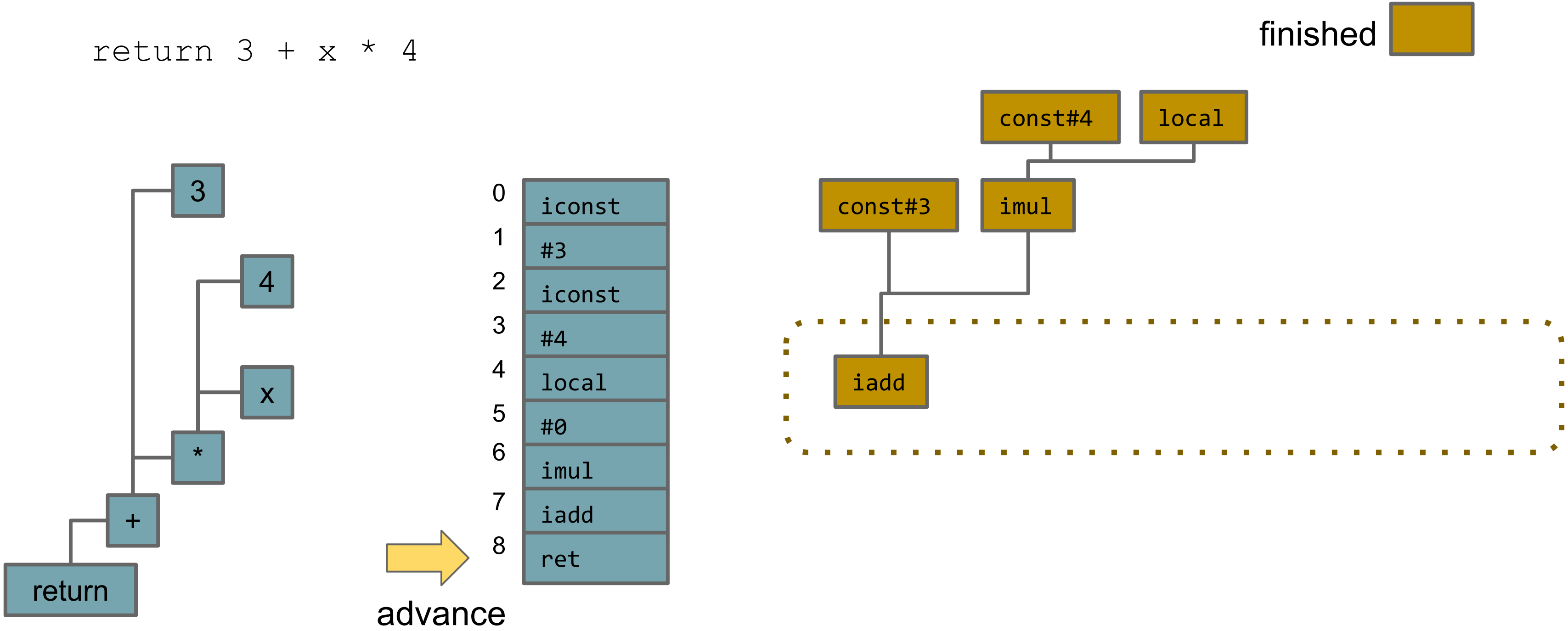


0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret



Decoding post-order to an AST

```
return 3 + x * 4
```

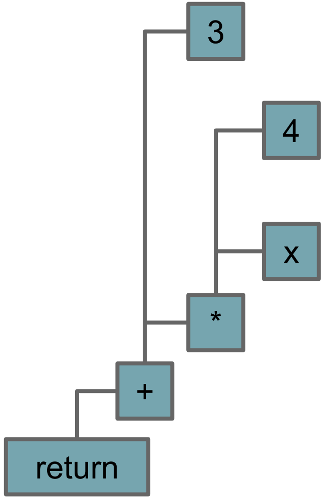


finished 🟨

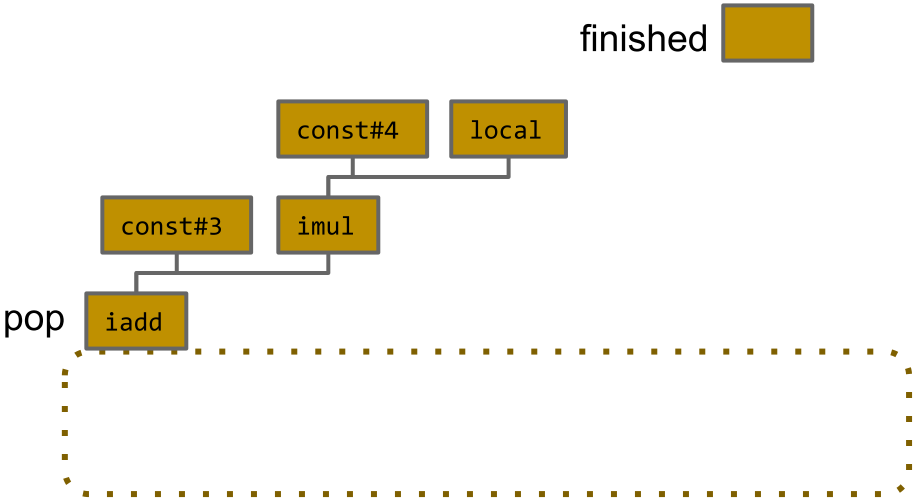
```
0  iconst
1  #3
2  iconst
3  #4
4  local
5  #0
6  imul
7  iadd
8  ret
```

Decoding post-order to an AST

```
return 3 + x * 4
```

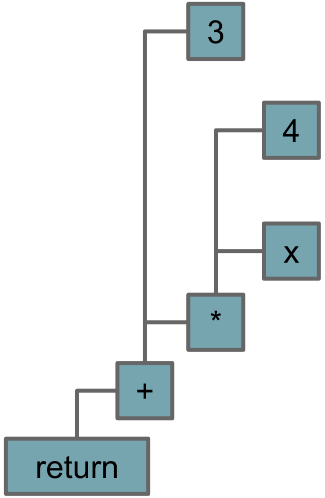


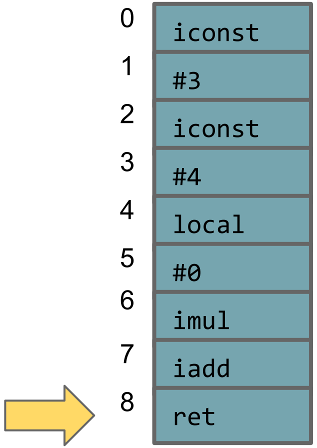
0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret



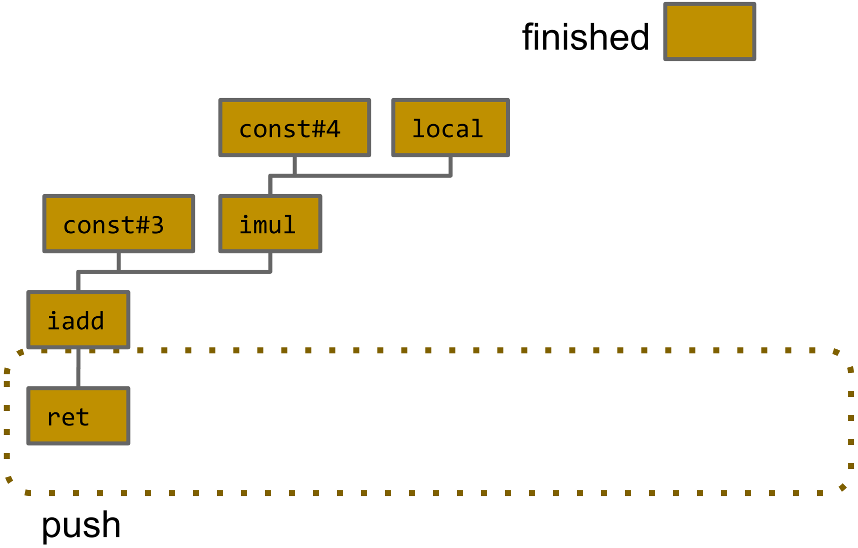
Decoding post-order to an AST

```
return 3 + x * 4
```



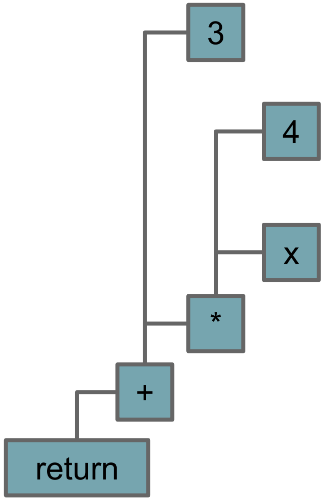


0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret



Decoding post-order to an AST

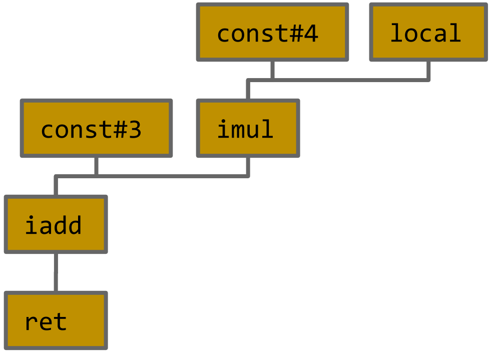
```
return 3 + x * 4
```



0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

finish

finished ▨

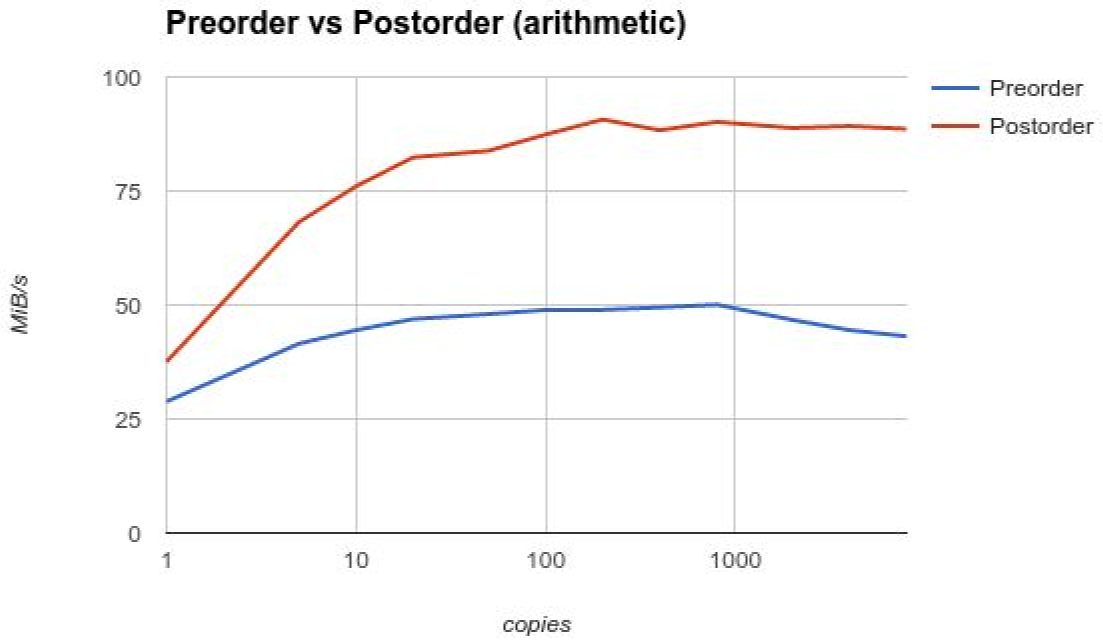


Decode+Verify performance

```
return 3 + x * 4
```

0	ret
1	iadd
2	iconst
3	#3
4	imul
5	iconst
6	#4
7	local
8	#0

preorder



0	iconst
1	#3
2	iconst
3	#4
4	local
5	#0
6	imul
7	iadd
8	ret

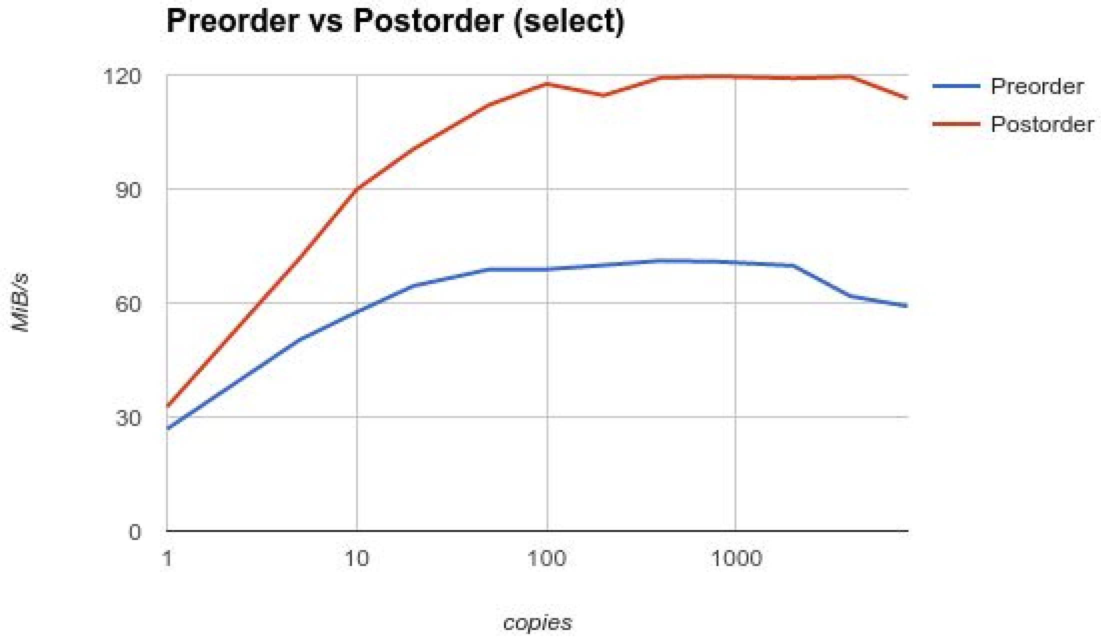
postorder

Decode+Verify performance

`select(2, 3, x)`

0	`select`
1	`iconst`
2	`#2`
3	`iconst`
4	`#3`
5	`local`
6	`#0`

preorder



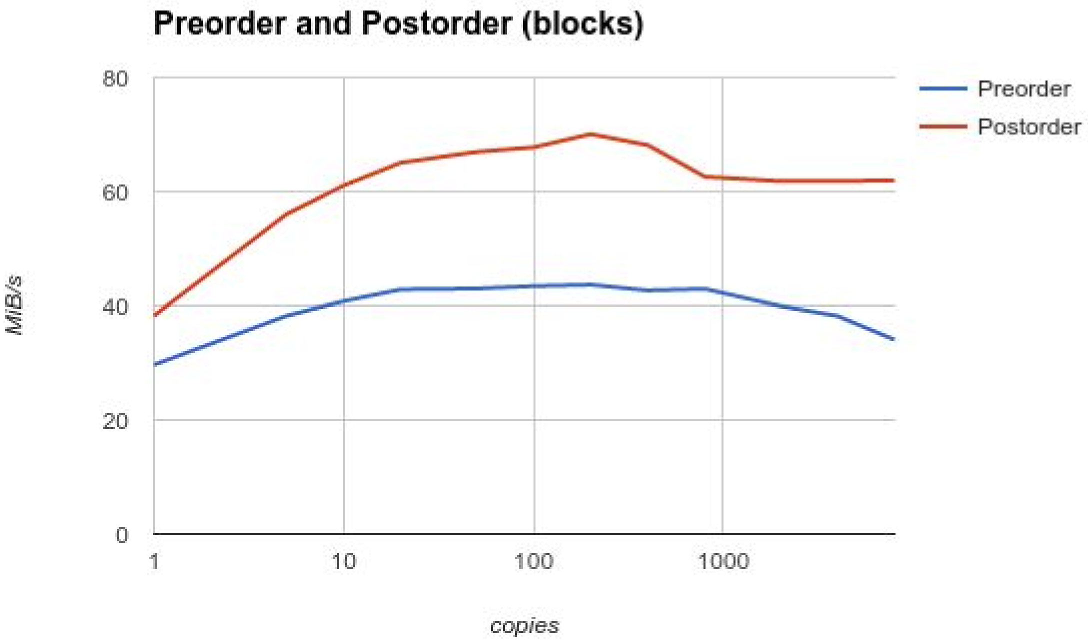
0	`iconst`
1	`#2`
2	`iconst`
3	`#3`
4	`local`
5	`#0`
6	`select`

postorder

Decode+Verify performance

```
block(block(br_if $0 x) br($1, #3)) #2)
```

0	block
1	2
2	block
3	2
4	br_if
5	$0
6	local
7	#0
8	nop
9	br
10	$1
11	iconst
12	#3
13	iconst
14	#2



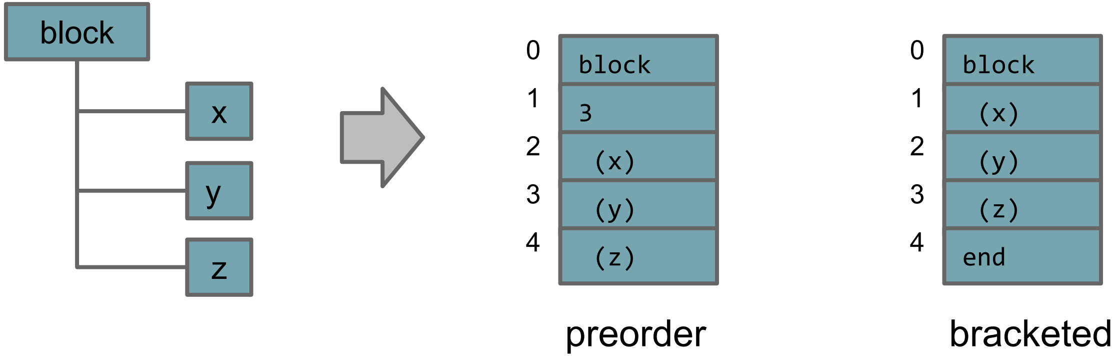
0	block
1	block
2	nop
3	local
4	#0
5	br_if
6	$0
7	iconst
8	#3
9	br
10	$1
11	end
12	iconst
13	#2
14	end

Postorder encodings of control

```
block
br
br_if
if
if_else
tableswitch
```

Preorder vs. Postorder `block`

```
(block x, y, z)
```



block

x

y

z

0	`block`
1	`3`
2	`(x)`
3	`(y)`
4	`(z)`

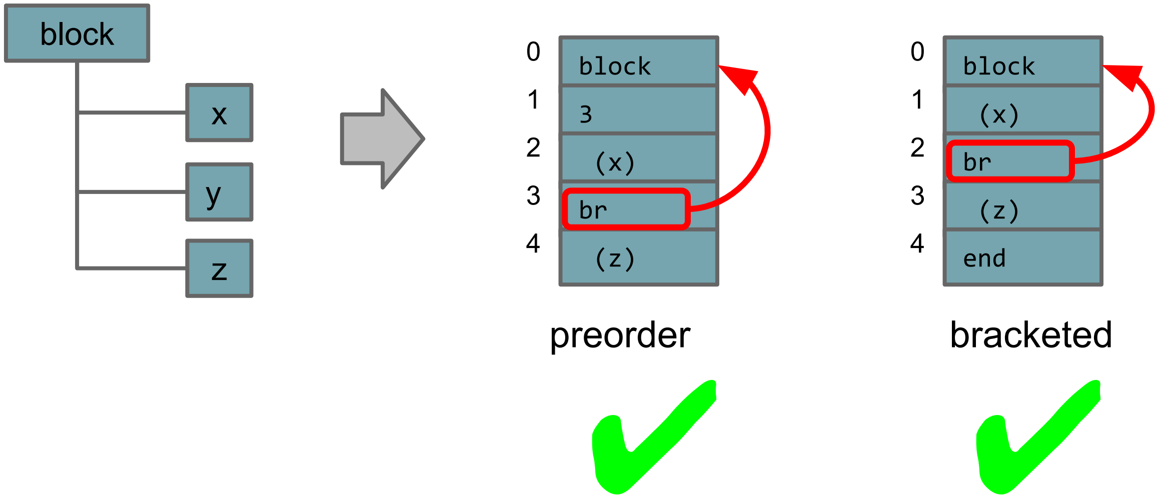
preorder

0	`block`
1	`(x)`
2	`(y)`
3	`(z)`
4	`end`

bracketed

Preorder vs. Postorder `block` verification

```
(block x, br $0, z)
```



block

x

y

z

	preorder
0	block
1	3
2	(x)
3	br
4	(z)

preorder ✓

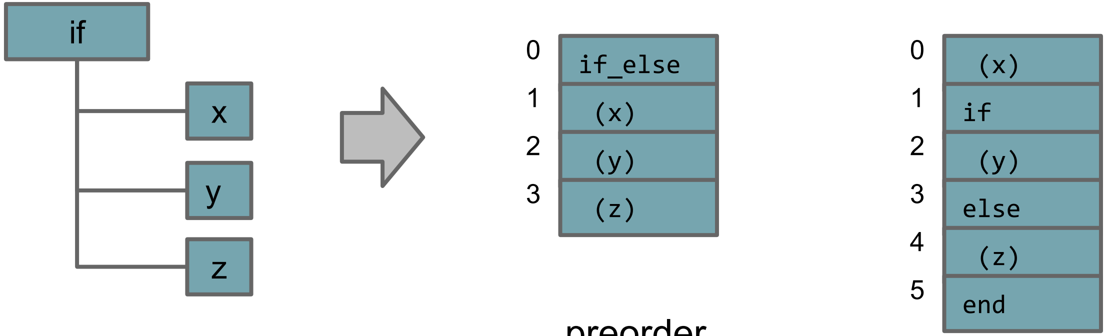
	bracketed
0	block
1	(x)
2	br
3	(z)
4	end

bracketed ✓

single-pass verification

Preorder vs. Postorder `if/else`

```
(if_else x, y, z)
```



0	`if_else`
1	`(x)`
2	`(y)`
3	`(z)`

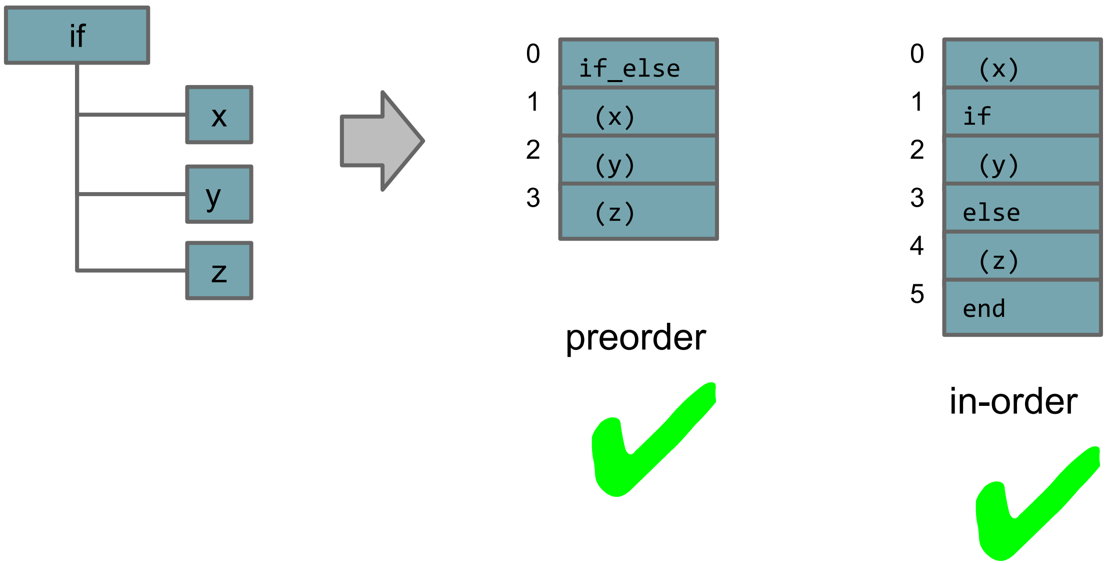
preorder

0	`(x)`
1	`if`
2	`(y)`
3	`else`
4	`(z)`
5	`end`

in-order

Preorder vs. Postorder `if/else`

```
(if_else x, y, z)
```



0	if_else
1	(x)
2	(y)
3	(z)

preorder ✓

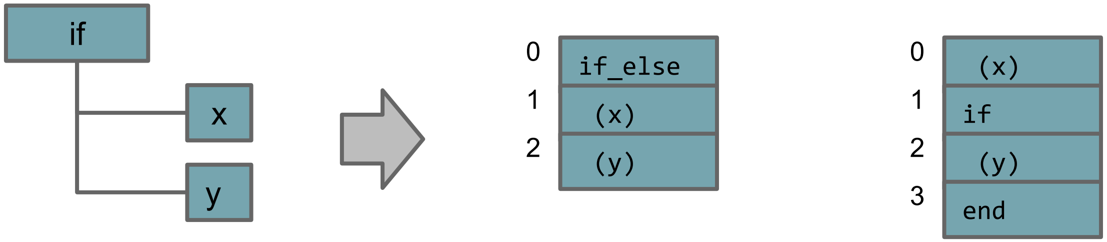
0	(x)
1	if
2	(y)
3	else
4	(z)
5	end

in-order ✓

single-pass verification

Preorder vs. Postorder `if/else`

```
(if x, y)
```



0	if_else
1	(x)
2	(y)

preorder

✓

0	(x)
1	if
2	(y)
3	end

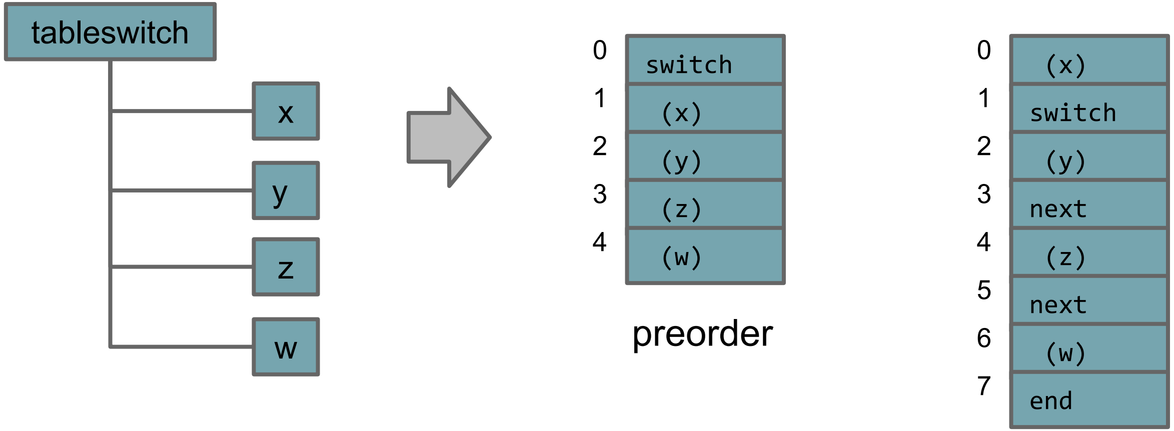
in-order

✓

single-pass verification

Preorder vs. Postorder `tableswitch`

```
(tableswitch x, y, z, w)
```



0	switch
1	(x)
2	(y)
3	(z)
4	(w)

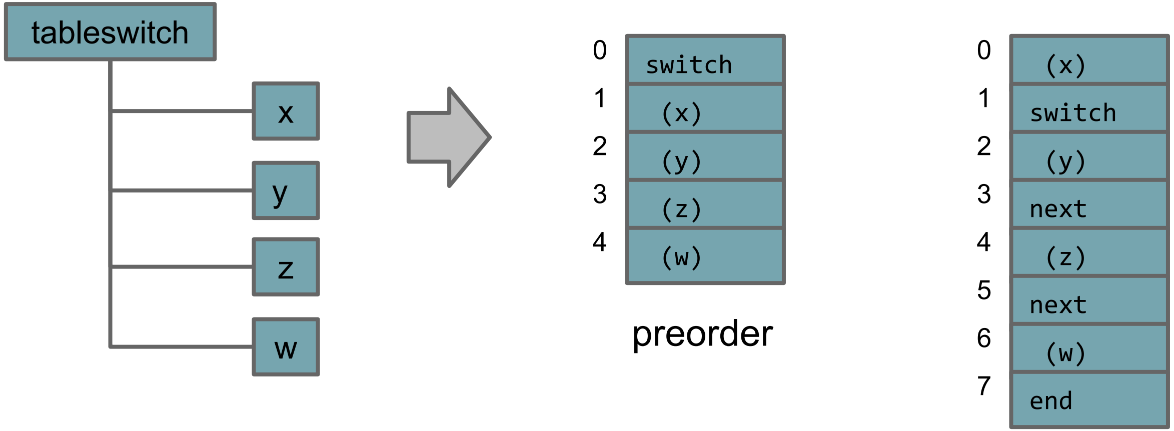
preorder

0	(x)
1	switch
2	(y)
3	next
4	(z)
5	next
6	(w)
7	end

in-order

Preorder vs. Postorder `tableswitch`

```
(tableswitch x, y, z, w)
```



0	switch
1	(x)
2	(y)
3	(z)
4	(w)

preorder

✓

0	(x)
1	switch
2	(y)
3	next
4	(z)
5	next
6	(w)
7	end

in-order

✓

single-pass verification

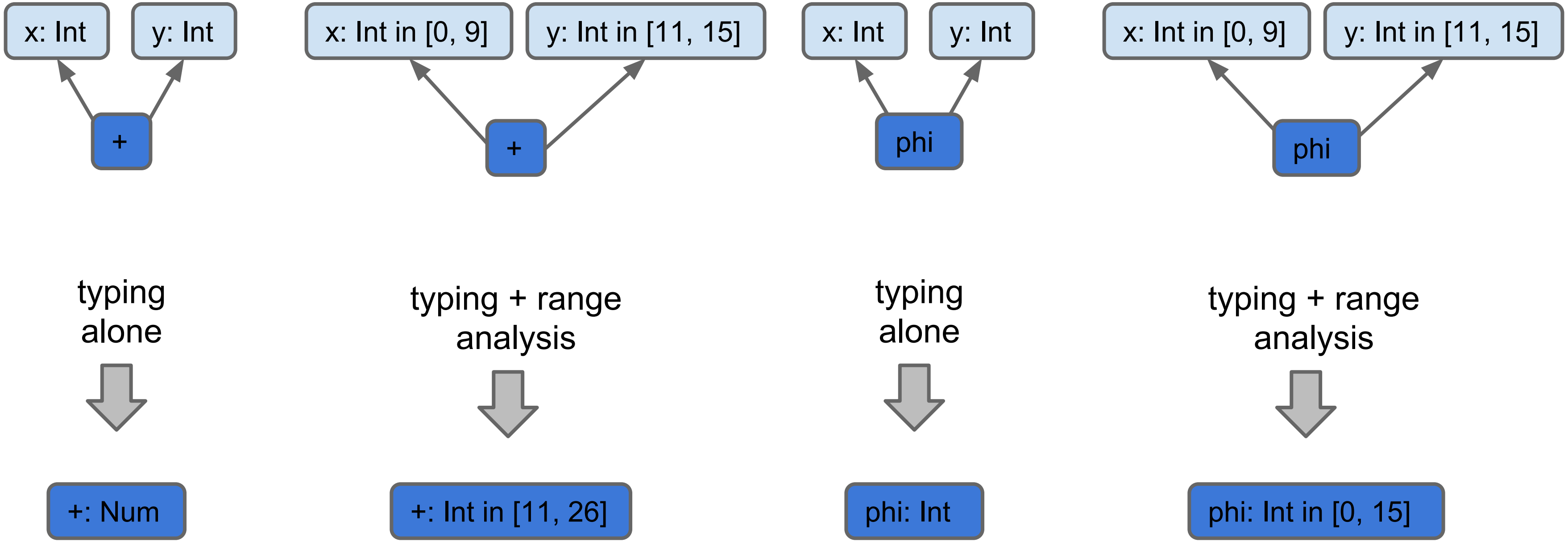
WebAssembly binary code

- Goals:
  - compact => smaller than minified JS
  - easy to verify => one linear pass
  - easy to compile => one linear pass to construct IR or baseline JIT
  - extensible => anticipate new bytecodes and types
- Did we deliver?
  - Fast single-pass decode+verify (> 100MB/s)
  - Single-pass to compiler IR demonstrated (V8/TurboFan)
  - Fast optimizing compiler (1.8MB/s single thread, 7MB/s with 8 threads)
  - Within 20% of native code execution speed (geomean; vs 80% for asm.js)
  - Single-pass compiler in development (Mozilla)

Compiling WASM vs. Compiling asm.js

- JavaScript is not statically typed
  - Values have types, not variables
  - 8 is a number, “`foo`” is a string
  - All basic operators (`+ - / * % << >>`) are overloaded or have implicit conversions
- All arithmetic is done in 64-bit floating point
  - Empirically most programs use small integers (<= 31 bits)
  - Overflow to double causes bailout to slow path, allocation, etc
  - Troublesome cases {`-0.0 NaN Infinity -Infinity`}
- Type “annotations” in asm.js
  - `a + b | 0` is integer arithmetic
  - `+(a + b)` is double arithmetic
  - `(a >>> 0) < (b >>> 0)` is an unsigned comparison

Type and Range Analysis (asm.js)



x: Int

y: Int

+

typing alone

+: Num

x: Int in [0, 9]

y: Int in [11, 15]

+

typing + range analysis

+: Int in [11, 26]

x: Int

y: Int

phi

typing alone

phi: Int

x: Int in [0, 9]

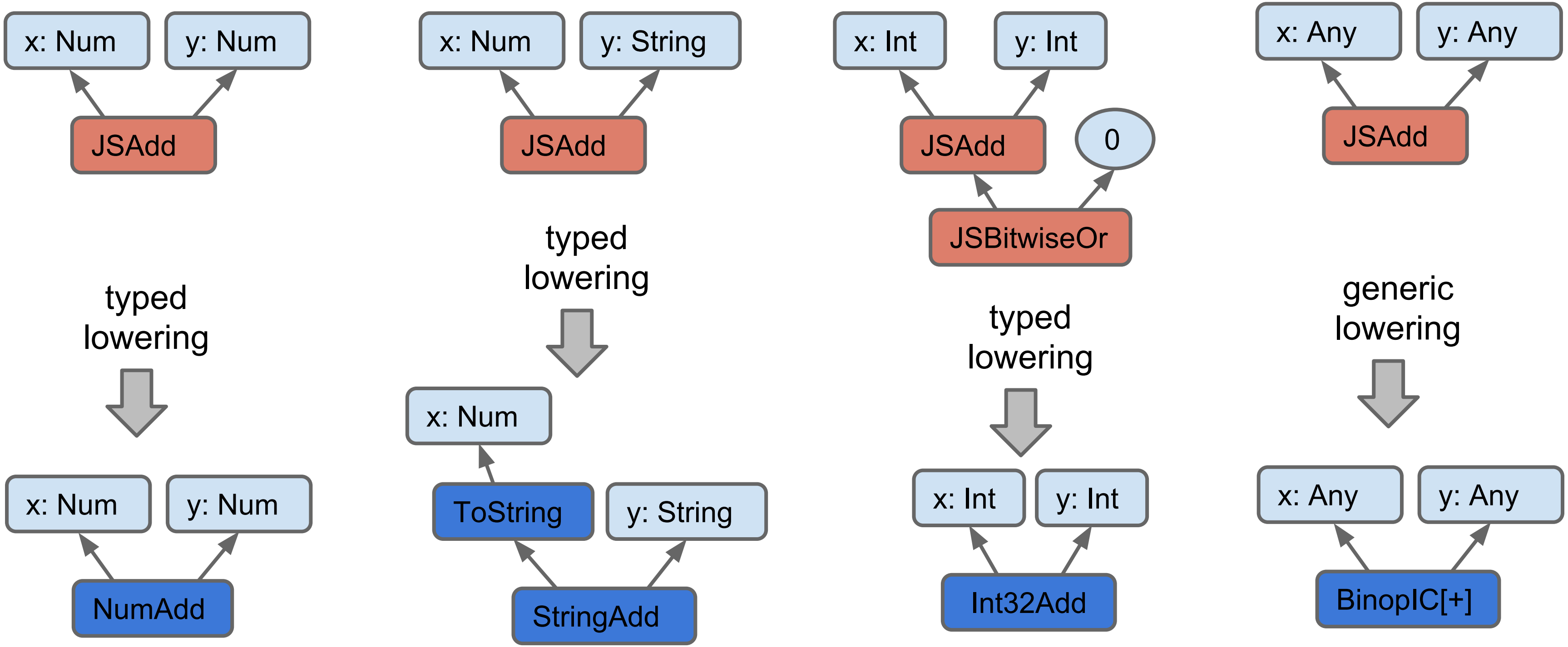
y: Int in [11, 15]

phi

typing + range analysis

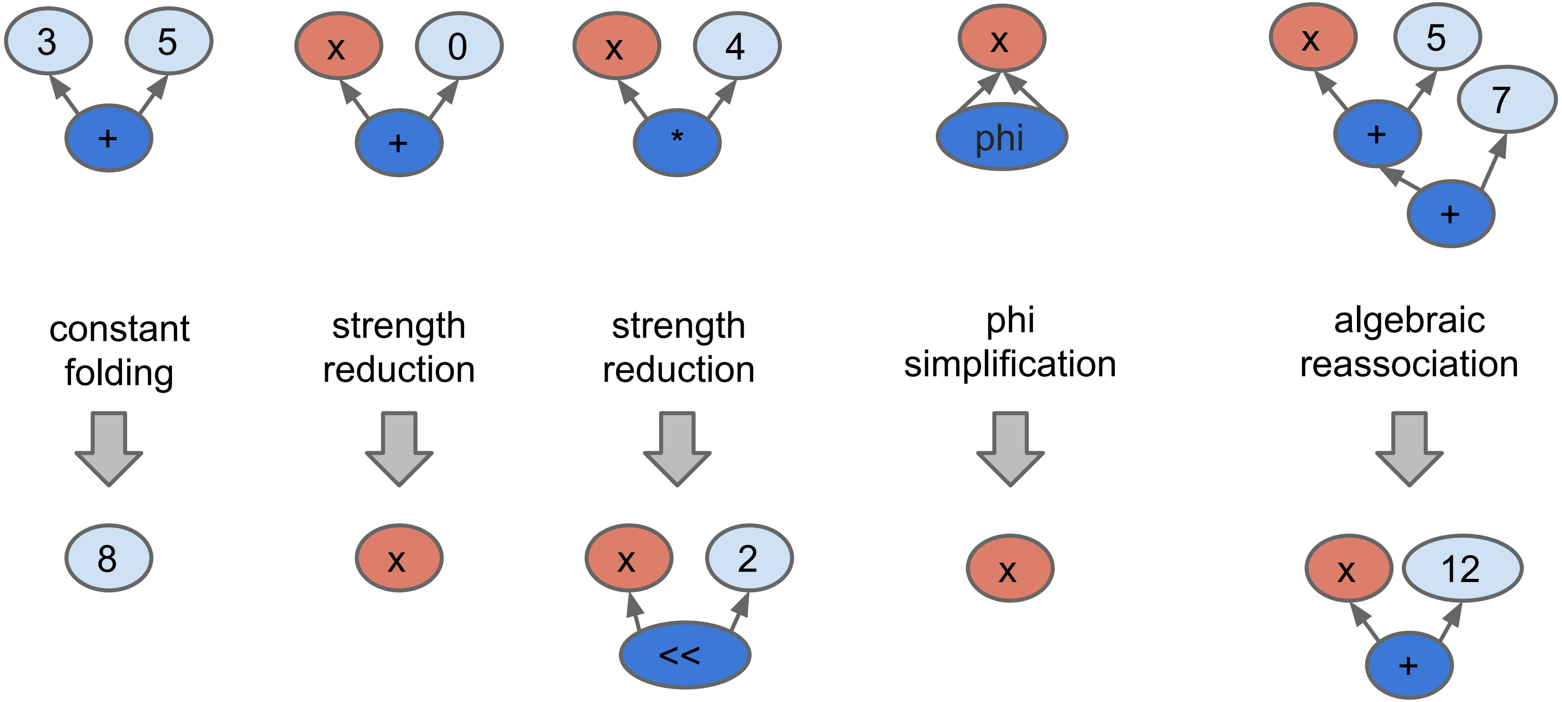
phi: Int in [0, 15]

Typed lowering as Reduction (asm.js)



WASM = no lowering necessary!

General Reductions



constant folding

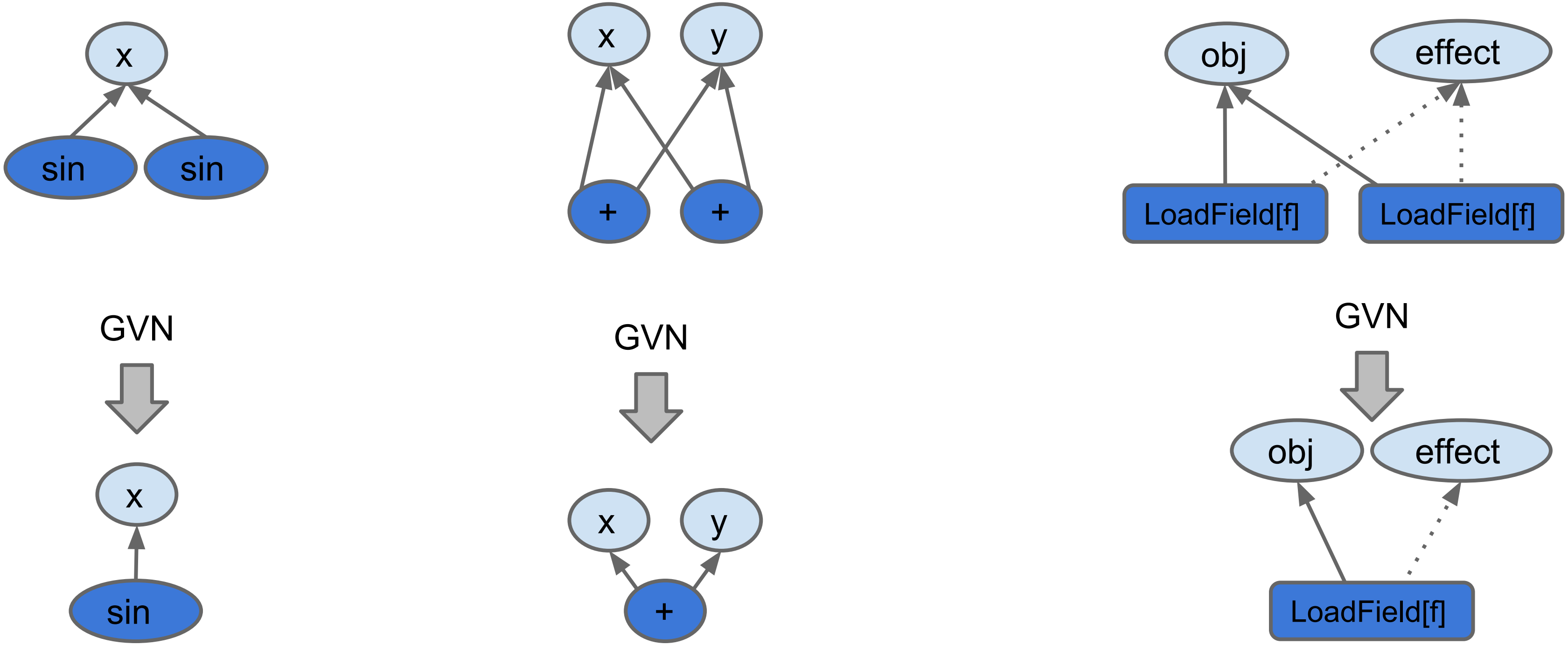
strength reduction

strength reduction

phi simplification

algebraic reassociation

General Reductions (2)



WebAssembly Status

- LLVM backend upstream
- Lots of tools
- Reference implementation (spec) in Standard ML
- 3 Browser engines have native support in various stages
  - Google Chrome Beta: fully spec compliant on all architectures, behind a flag
  - Mozilla Firefox: optimized for ia32 and x64, behind a flag
  - Microsoft Edge: support in an experimental build
- MVP (Version 1.0) expected to be shipped this summer
- Standardization expected by the end of the year

`https://github.com/WebAssembly/`

Questions?