LLVM Backend for VRL

Maximizing Performance by Eliminating Runtime Overhead

January 2022

Pablo Sichert
Software Engineer, Vector

VRL Architecture Overview

How to get from VRL program text to expression tree with defined semantics?

Program Text Tokens AST Expression Tree Result

VRL Architecture Overview

How to get from VRL program text to expression tree with defined semantics?

VRL Language Frontend

Program Text | Tokens | AST | Expression Tree | Result

DATADOG

VRL Architecture Overview

```
if .status == 200 {
  .message = "ok"
}
```

Program Text Tokens AST Expression Tree Result

VRL Architecture Overview

```
if .status == 200 {
   .message = "ok"
}
```

DATADOG

VRL Architecture Overview

```
if .status == 200 {
  .message = "ok"
}
```

```
If Dot Identifier("status") Operator("==") IntegerLiteral(200) LBrace
  Dot Identifier("message") Equals StringLiteral("ok")
RBrace
```

DATADOG

VRL Architecture Overview

```
if .status == 200 {
  .message = "ok"
}
```

```
If Dot Identifier("status") Operator("==") IntegerLiteral(200) LBrace
  Dot Identifier("message") Equals StringLiteral("ok")
RBrace
```

DATADOG

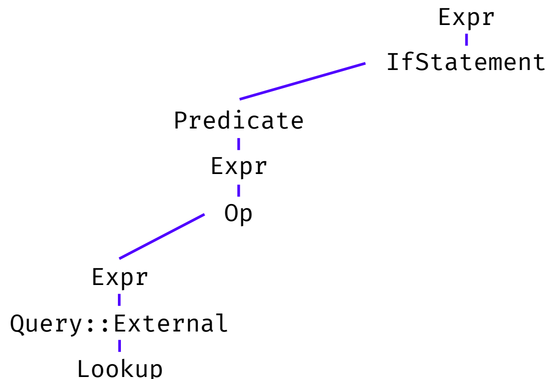
VRL Architecture Overview

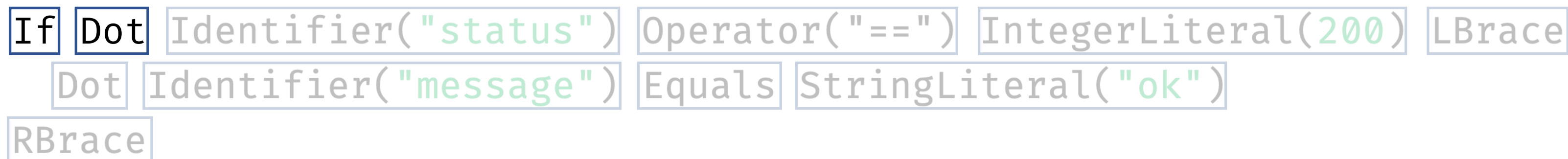
```
Expr
IfStatement
```

```
If Dot Identifier("status") Operator("==") IntegerLiteral(200) LBrace
  Dot Identifier("message") Equals StringLiteral("ok")
RBrace
```

Program Text Tokens AST Expression Tree Result

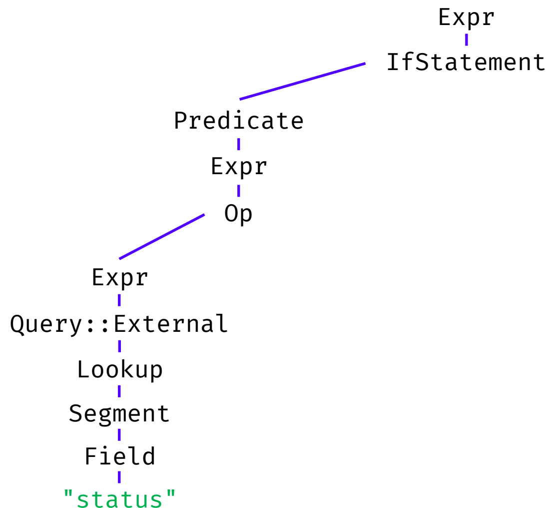
VRL Architecture Overview

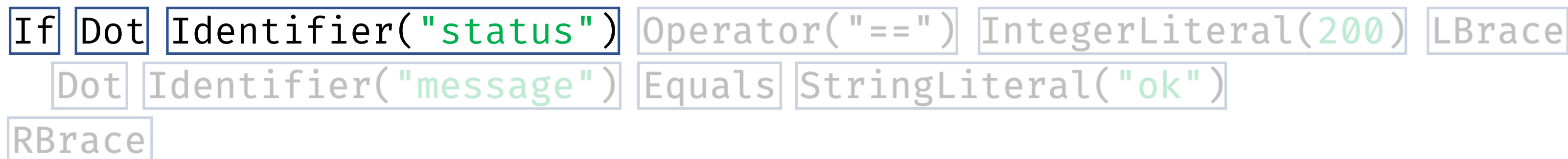




DATADOG

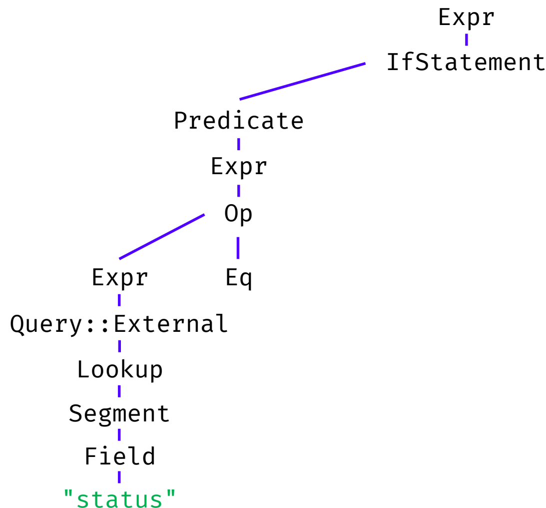
VRL Architecture Overview

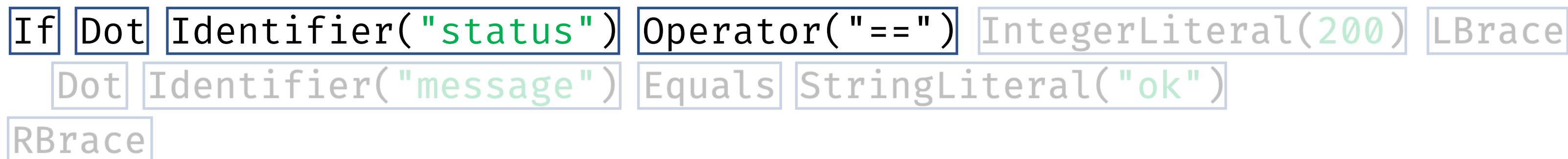




DATADOG

VRL Architecture Overview

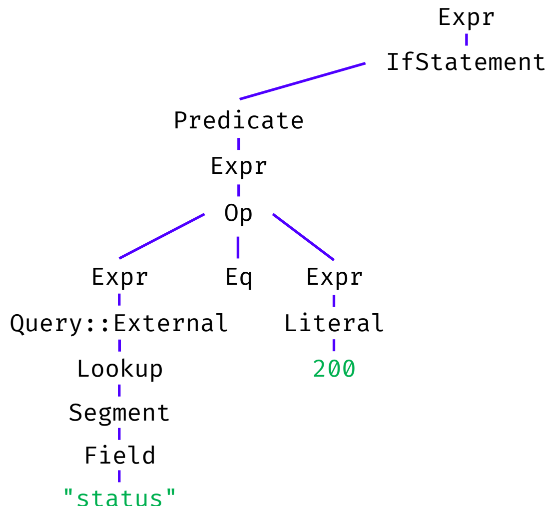


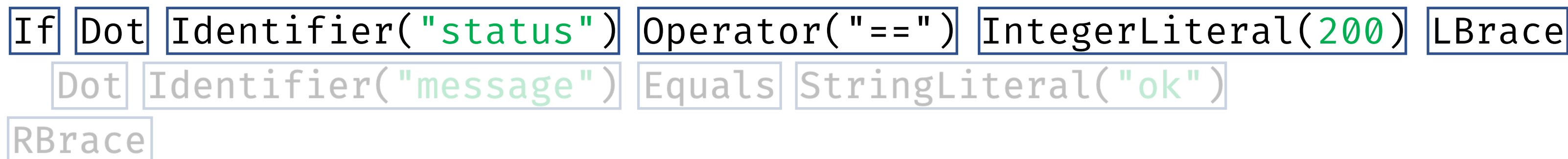


Expression Tree Result

DATADOG

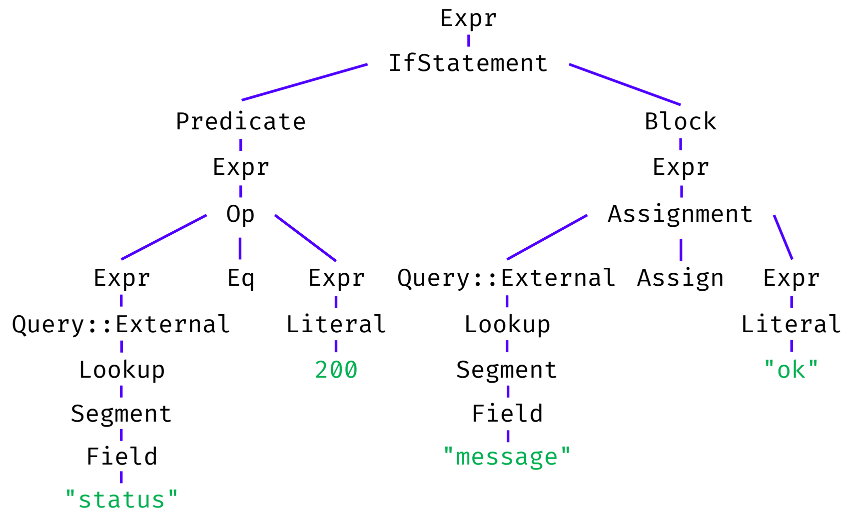
VRL Architecture Overview

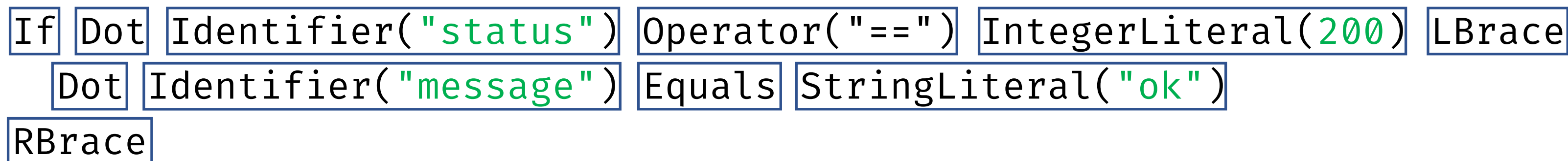




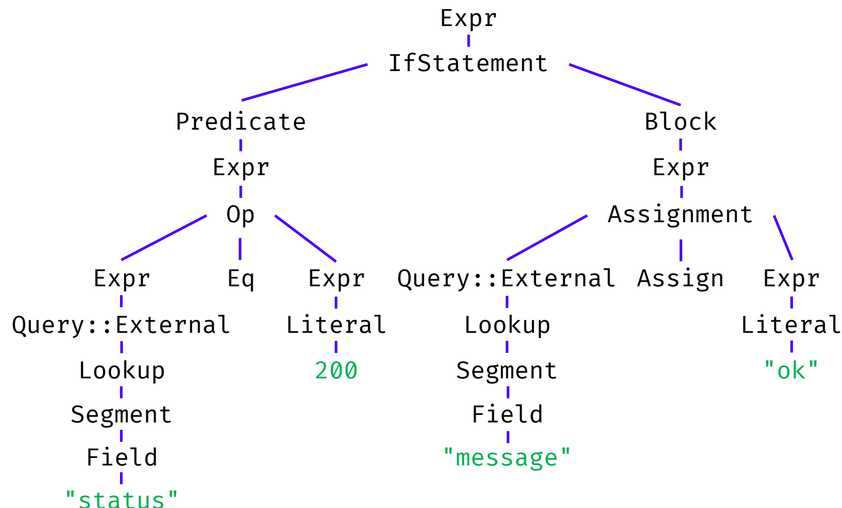
DATADOG

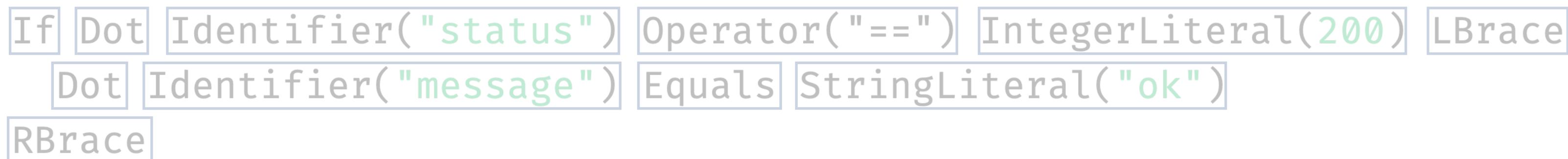
VRL Architecture Overview



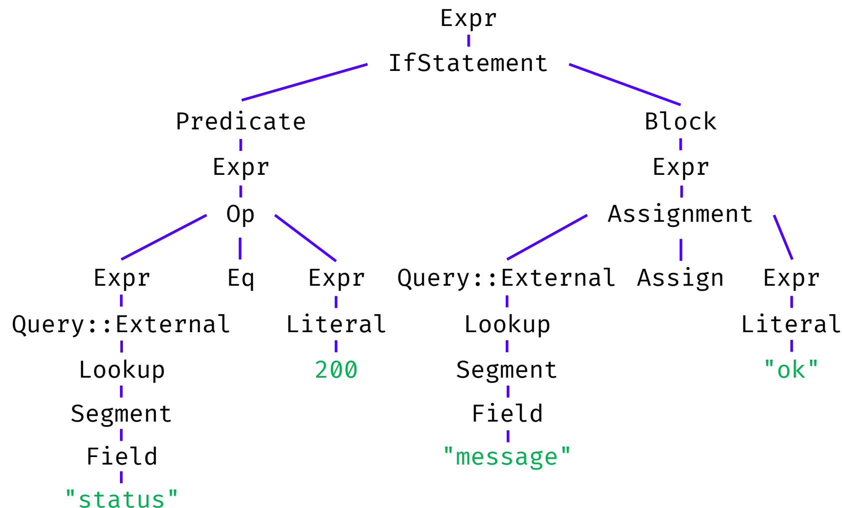


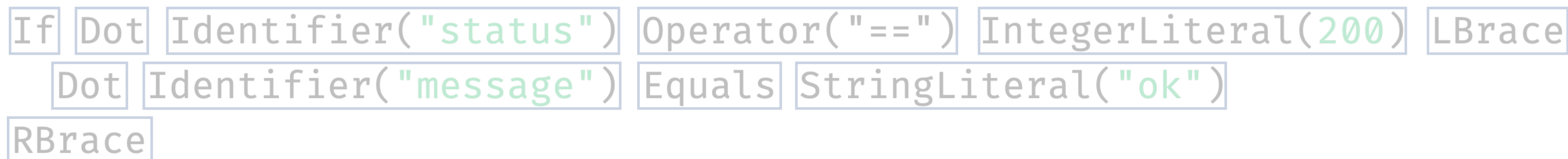
VRL Architecture Overview

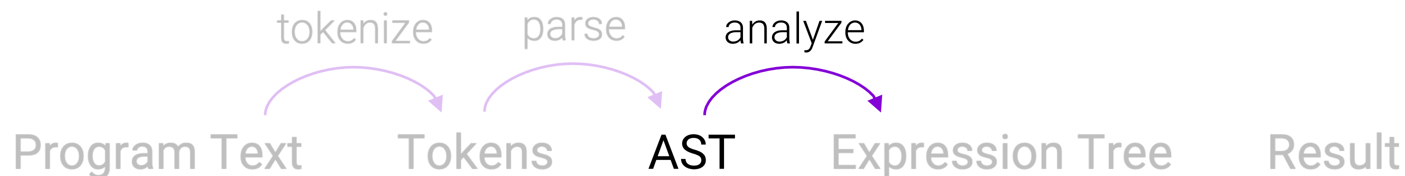




VRL Architecture Overview



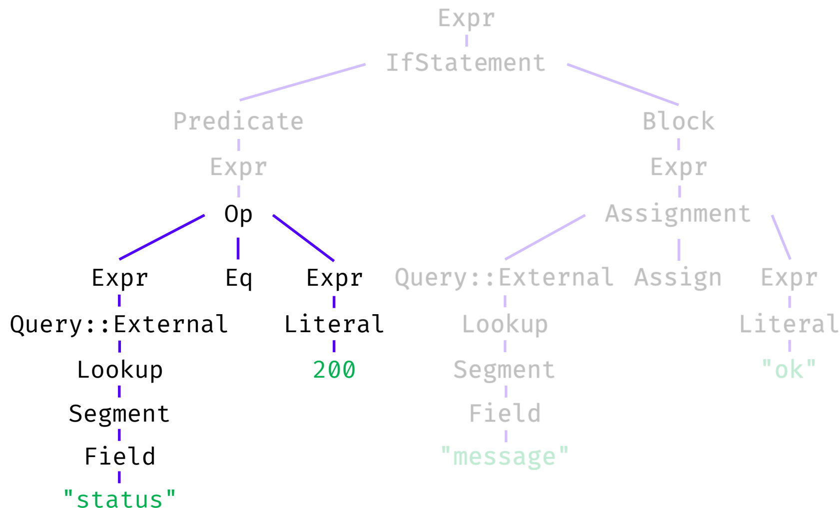


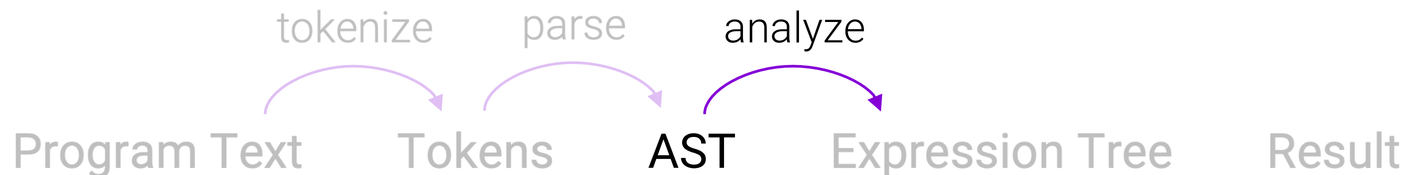


DATADOG

VRL Architecture Overview

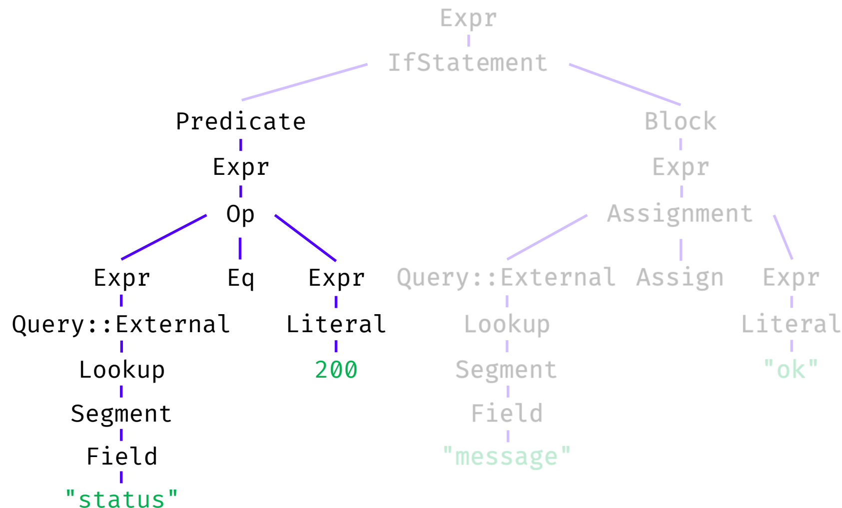
Are operand types comparable?





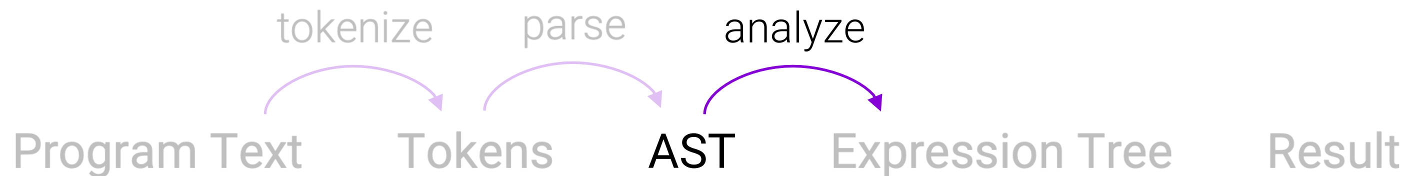
DATADOG

VRL Architecture Overview

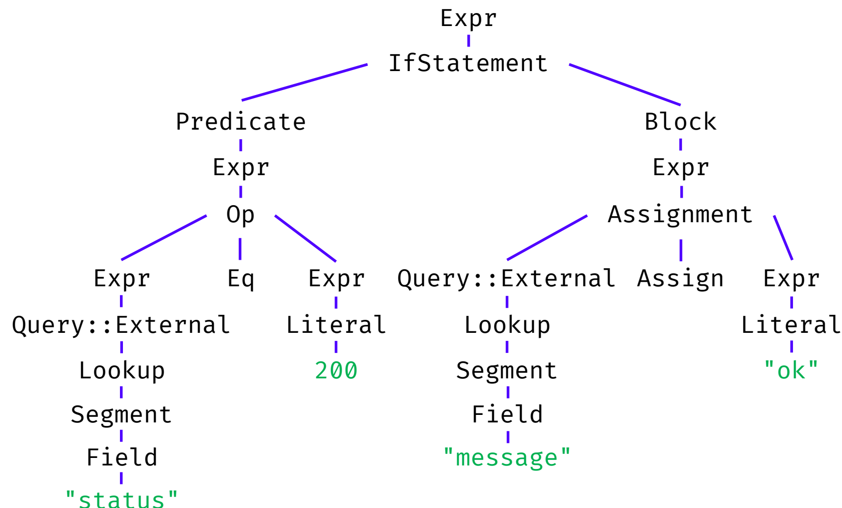


Are operand types comparable?

Is predicate boolean?



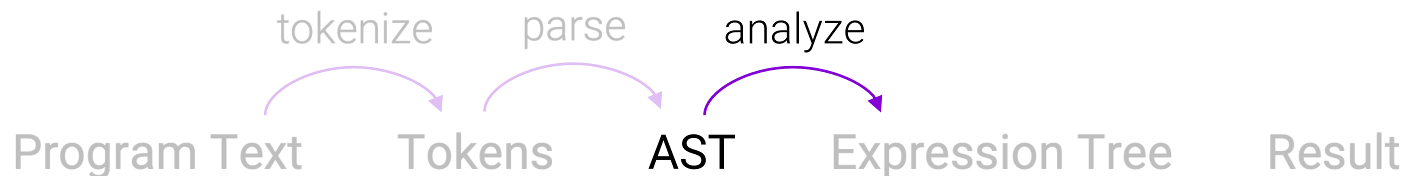
VRL Architecture Overview



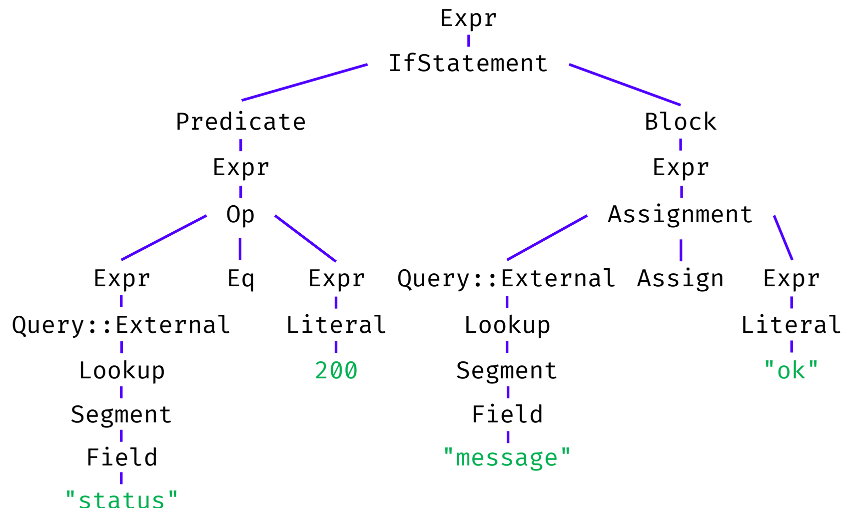
Are operand types comparable?

Is predicate boolean?

Is expression infallible?



VRL Architecture Overview



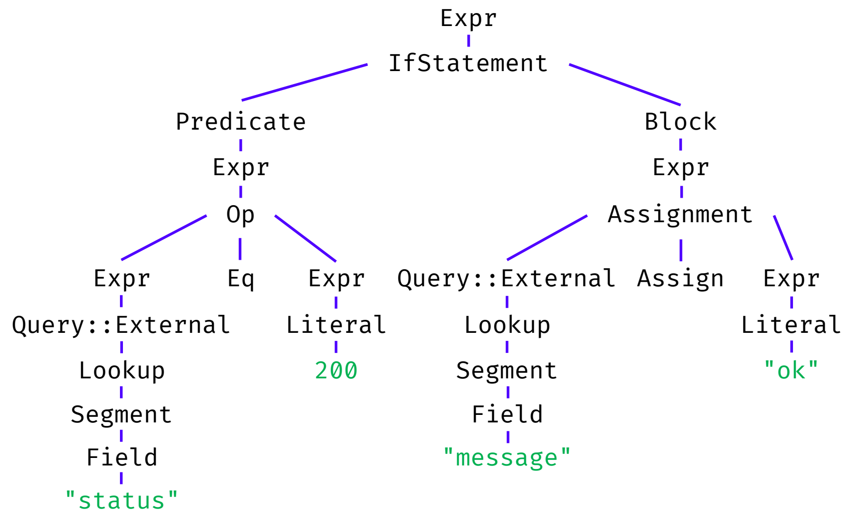
Are operand types comparable?

Is predicate boolean?

Is expression infallible?

DATADOG

VRL Architecture Overview



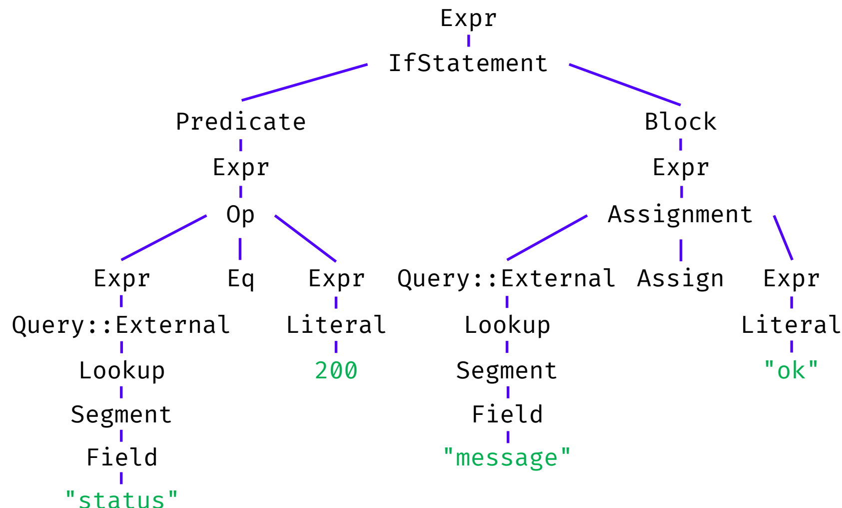
Are operand types comparable?

Is predicate boolean?

Is expression infallible?

DATADOG

VRL Architecture Overview



How to evaluate expression tree?

DATADOG

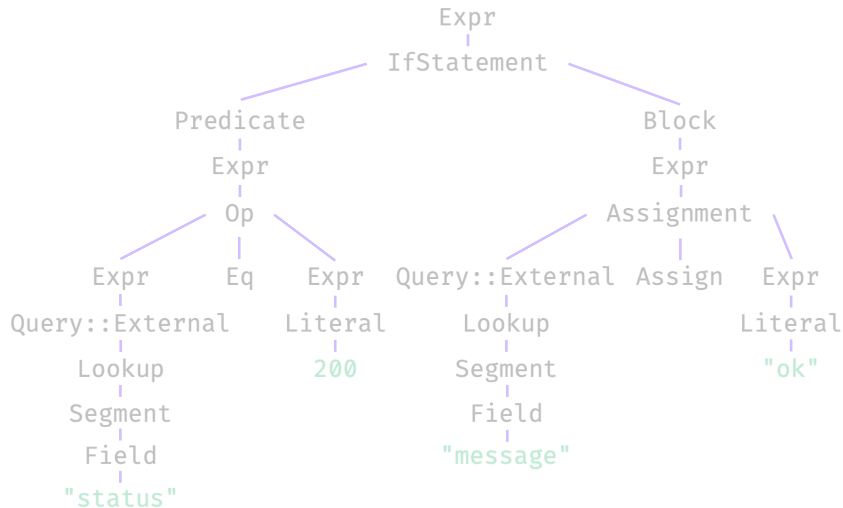
VRL Execution Model – Expression Traversal

```
                         Expr
                          |
                     IfStatement
              /                      \
         Predicate                  Block
             |                        |
           Expr                     Expr
             |                        |
            Op                   Assignment
        /    |    \            /      |      \
    Expr    Eq    Expr  Query::External  Assign  Expr
     |             |          |                   |
Query::External  Literal    Lookup              Literal
     |             |          |                   |
   Lookup         200      Segment               "ok"
     |                        |
  Segment                   Field
     |                        |
   Field                  "message"
     |
  "status"
```

```rust
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

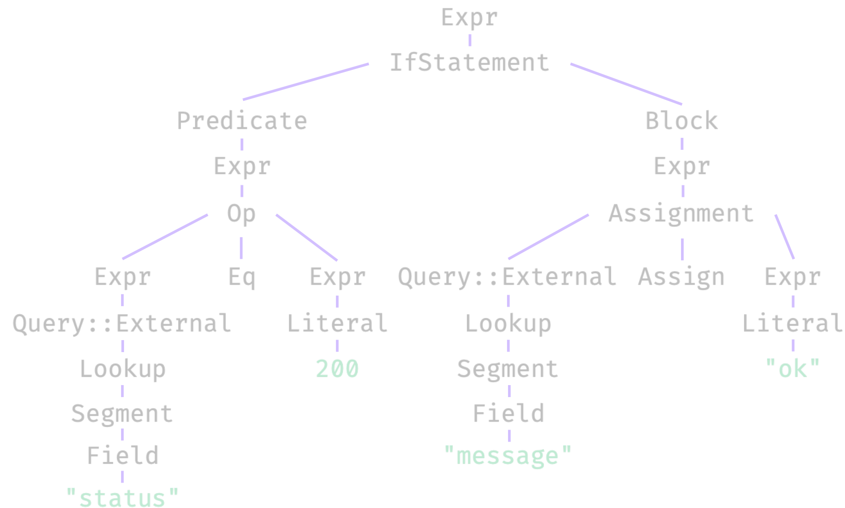
DATADOG

VRL Execution Model – Expression Traversal




```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

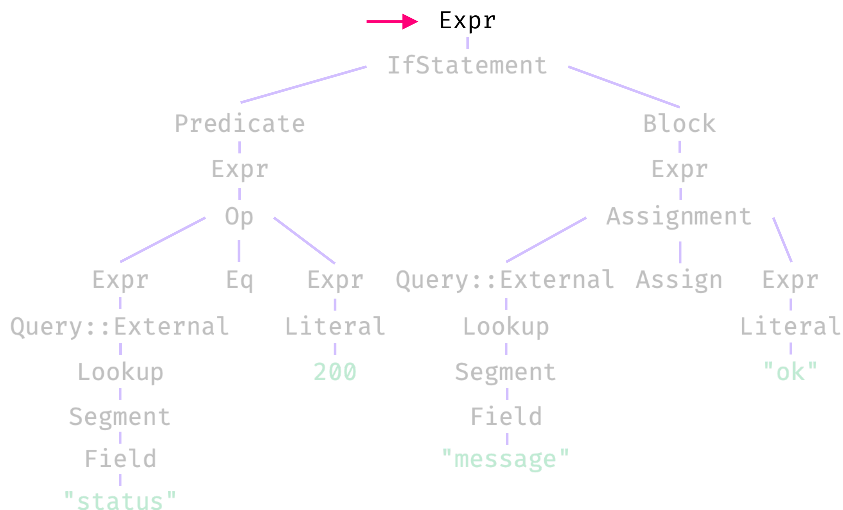
VRL Execution Model – Expression Traversal




```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

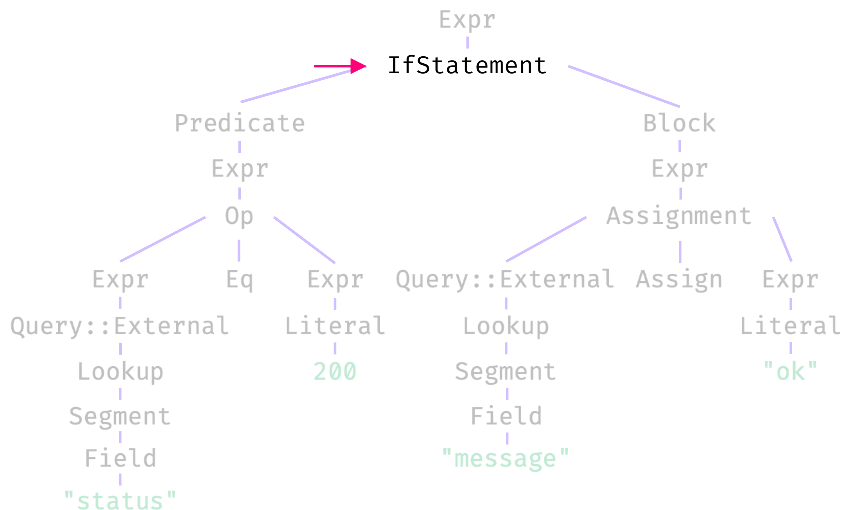



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for Expr {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

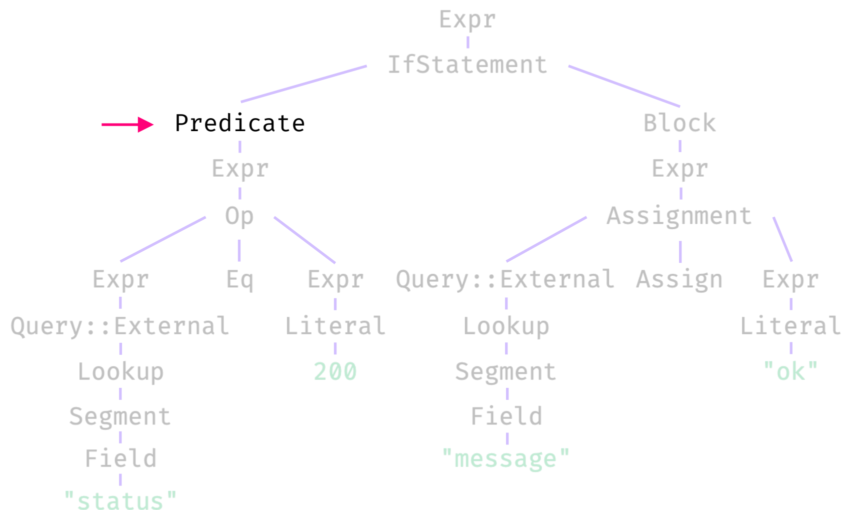



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for IfStatement {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

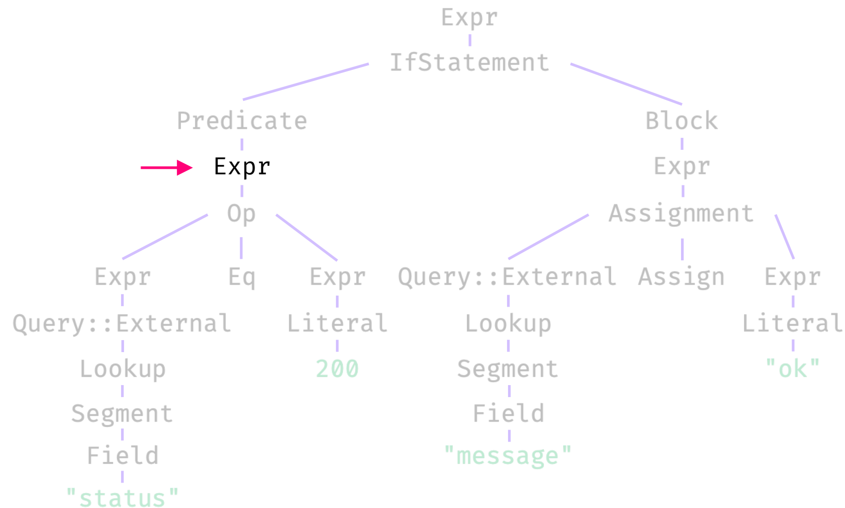



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for Predicate {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

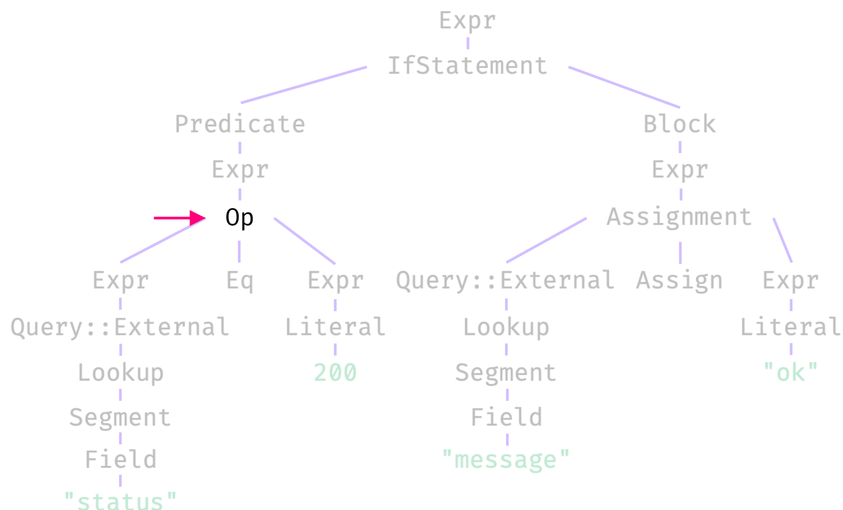



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for Expr {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

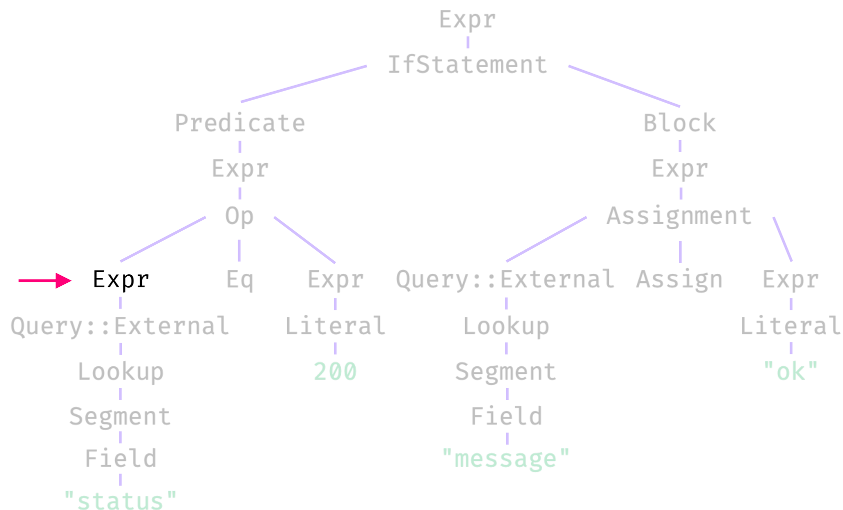



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for Op {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

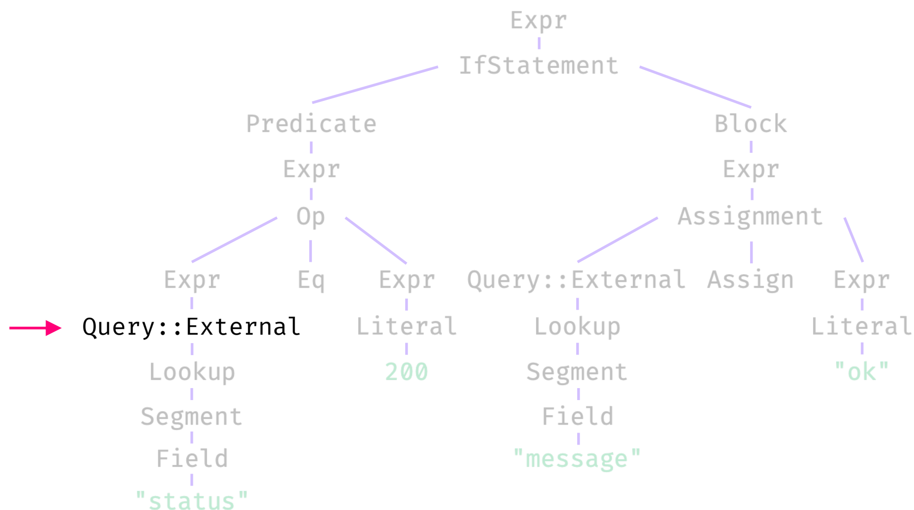



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for Expr {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

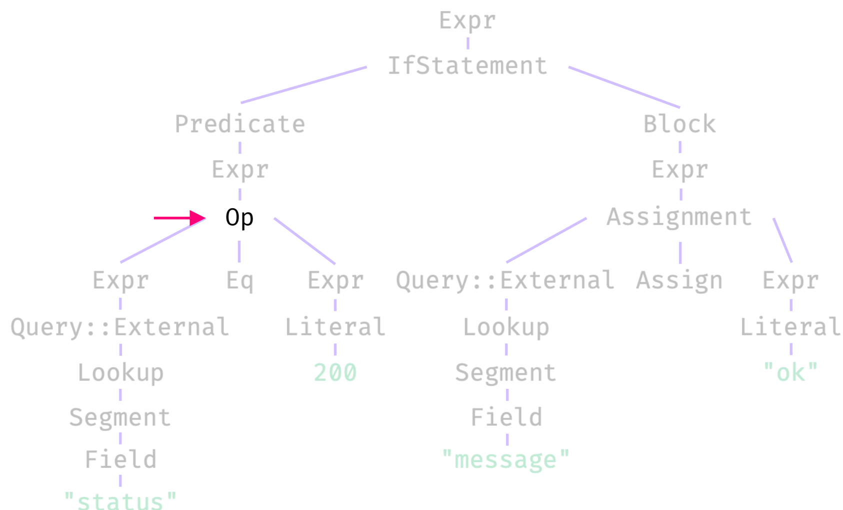



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for Query {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

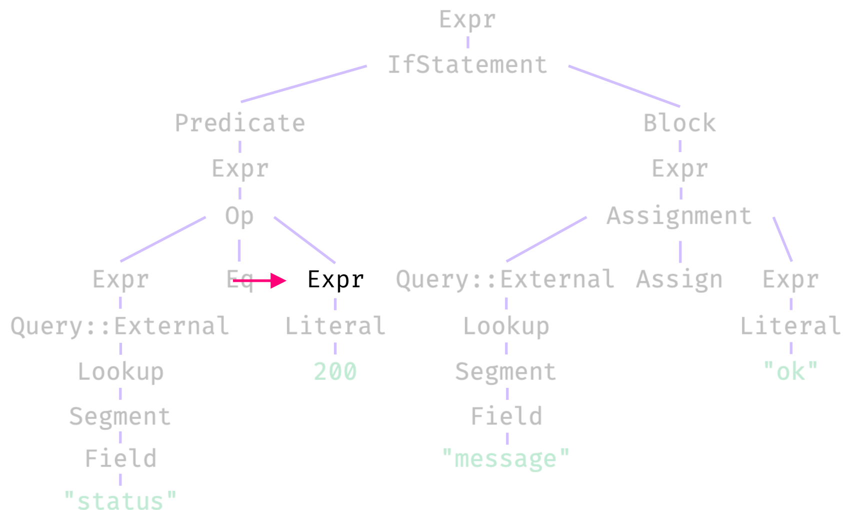



```rust
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```rust
impl Expression for Op {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

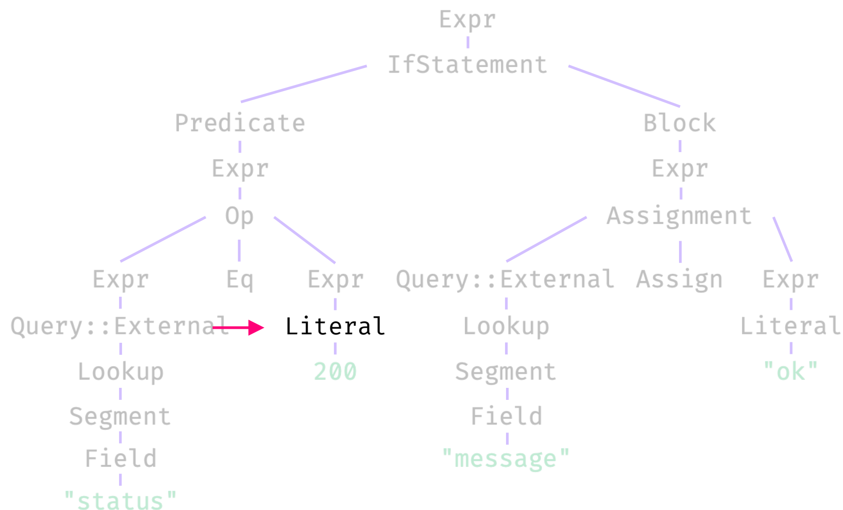



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for Expr {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

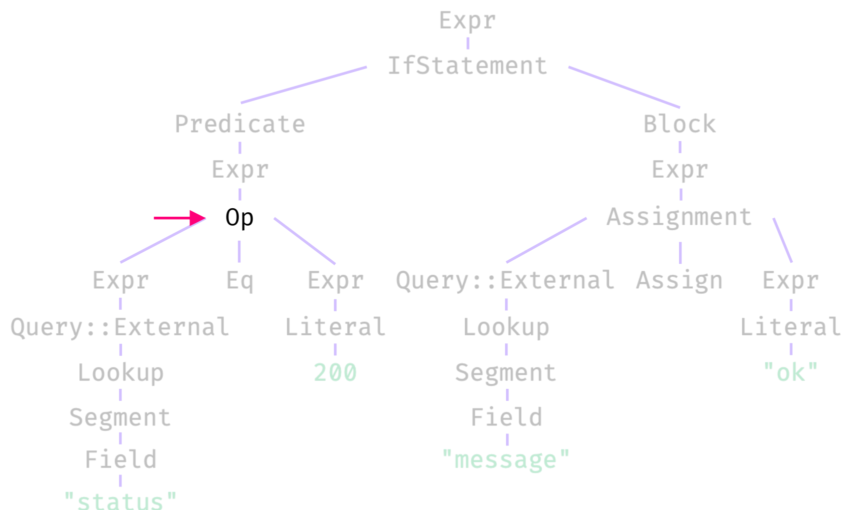



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for Literal {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

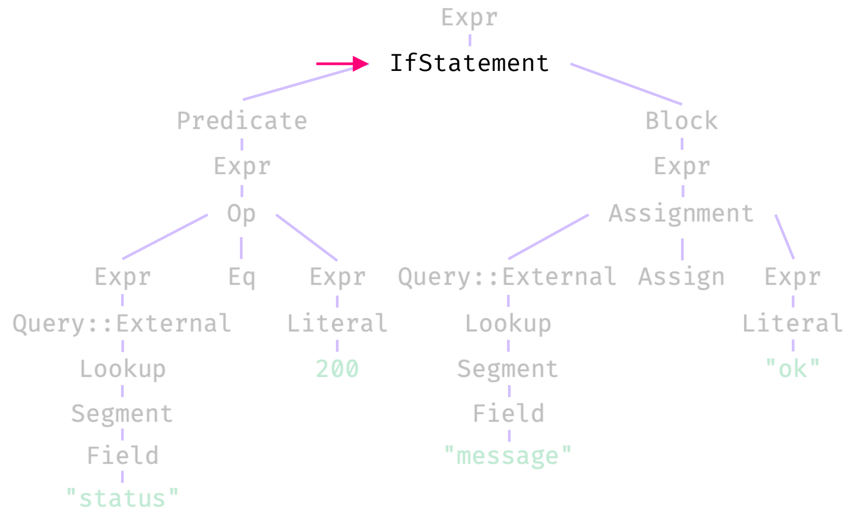



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for Op {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

VRL Execution Model – Expression Traversal

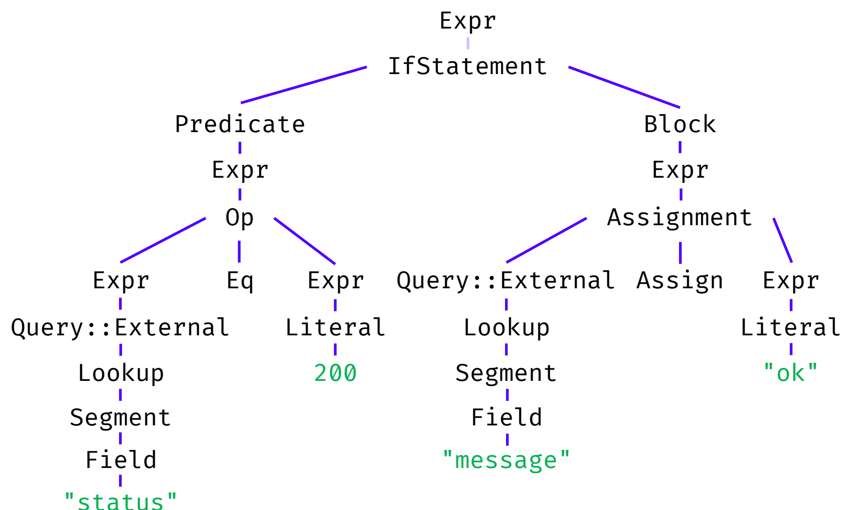



```
pub trait Expression {
  fn resolve(&self, ctx: &mut Context) -> Resolved;
}
```

```
impl Expression for IfStatement {
  fn resolve(&self, ctx: &mut Context) -> Resolved {
    ...
  }
}
```

Input: { "status": 400 }

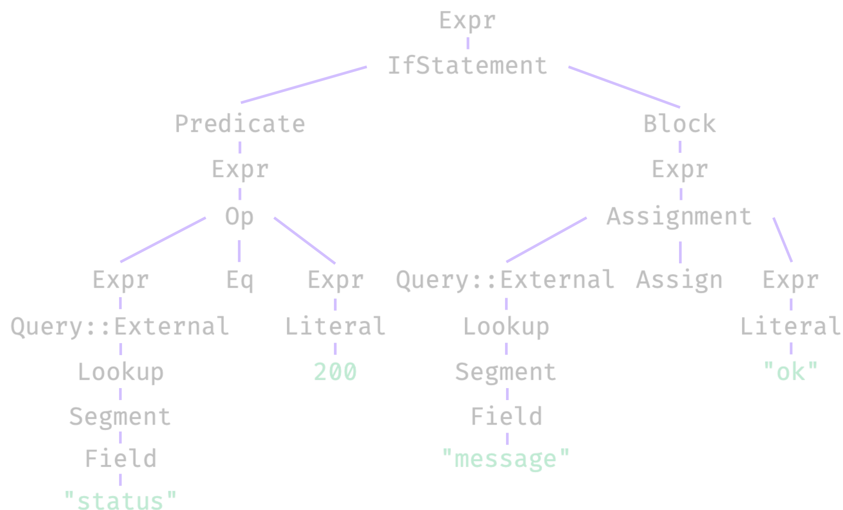
VRL Execution Model – Virtual Machine




```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

DATADOG

VRL Execution Model – Virtual Machine




```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

DATADOG

VRL Execution Model – Virtual Machine

```rust
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

VRL Execution Model – Virtual Machine

```rust
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

```
Input: { "status": 400 }
```

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

```
Input: { "status": 400 }
```

VRL Execution Model – Virtual Machine

```rust
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
→     OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath) ←
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

```
Input: { "status": 400 }
```

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

```
Input: { "status": 400 }
```

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

Input: { "status": 400 }

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

```
Input: { "status": 400 }
```

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

Input: { "status": 400 }

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

```
Input: { "status": 400 }
```

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

```
Input: { "status": 400 }
```

VRL Execution Model – Virtual Machine

```rust
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal) ←
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

Input: { "status": 400 }

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
    → OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse) ←
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

Input: { "status": 400 }

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

```
Input: { "status": 400 }
```

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

Input: { "status": 400 }

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
→   let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1) ←
OpCode(Pop)
OpCode(Return)
```

Input: { "status": 400 }

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
  →   OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop) ←
OpCode(Return)
```

```
Input: { "status": 400 }
```

VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
      OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return)
```

```
Input: { "status": 400 }
```

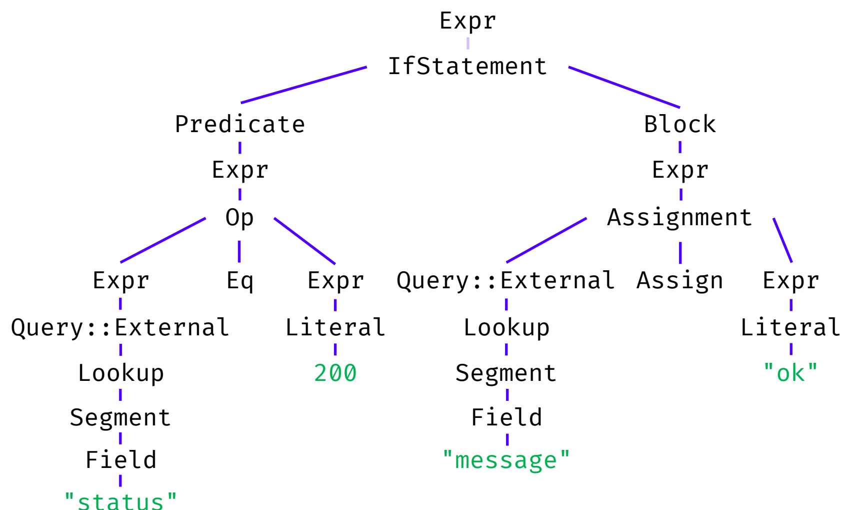
VRL Execution Model – Virtual Machine

```
pub fn interpret<'a>(
  &self, ctx: &mut Context<'a>
) -> Result<Value, ExpressionError> {
  let mut state = VmState::new(self);
  loop {
    let next = state.next()?;
    match next {
      OpCode::Constant => { ... }
      OpCode::Equal => { ... }
      OpCode::GetPath => { ... }
      OpCode::Jump => { ... }
      OpCode::JumpIfFalse => { ... }
      OpCode::Pop => { ... }
  →   OpCode::Return => { ... }
      OpCode::SetPath => { ... }
      ...
    }
  }
}
```

```
OpCode(GetPath)
Primitive(0)
OpCode(Constant)
Primitive(0)
OpCode(Equal)
OpCode(JumpIfFalse)
Primitive(7)
OpCode(Pop)
OpCode(Constant)
Primitive(1)
OpCode(SetPath)
Primitive(1)
OpCode(Jump)
Primitive(1)
OpCode(Pop)
OpCode(Return) ←
```

```
Input: { "status": 400 }
```

VRL Execution Model – Code Generation

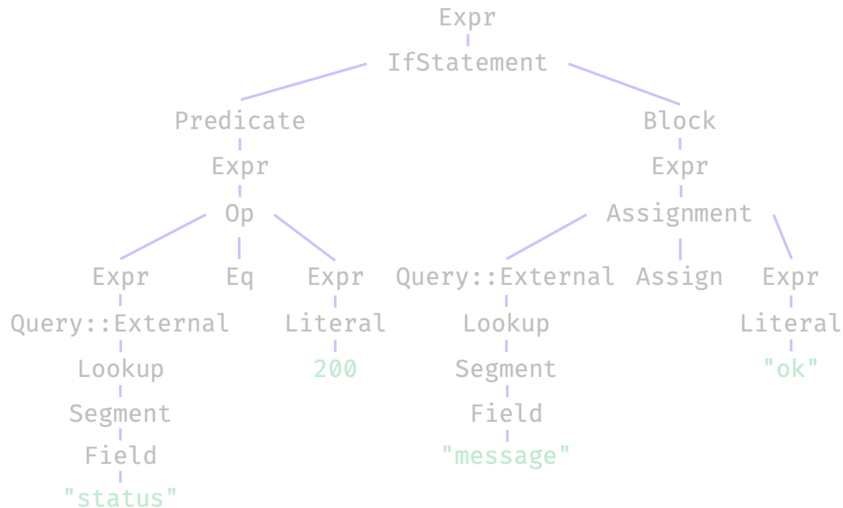



```
define void @vrl_execute(%"Context"* align 8 %context,
start:
  call void @vrl_expression_query_target_external_impl
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"*
  call void @vrl_expression_literal_impl(%"Value"* bit
  call void @vrl_expression_op_eq_impl(%"Result<Value>
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolve
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs
  br i1 %vrl_resolved_boolean_is_true, label %if_state

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bit
  call void @vrl_expression_assignment_target_insert_e
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

VRL Execution Model – Code Generation




```
define void @vrl_execute(%"Context"* align 8 %context,
start:
  call void @vrl_expression_query_target_external_impl
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"*
  call void @vrl_expression_literal_impl(%"Value"* bit
  call void @vrl_expression_op_eq_impl(%"Result<Value>
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolve
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs
  br i1 %vrl_resolved_boolean_is_true, label %if_state

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bit
  call void @vrl_expression_assignment_target_insert_e
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

Input: { "status": 400 }

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

Input: { "status": 400 }

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
→ %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

Input: { "status": 400 }

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
  %rhs = alloca %"Result<Value>"*, align 8
→ call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

Input: { "status": 400 }

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
→ call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

Input: { "status": 400 }

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
→ call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

Input: { "status": 400 }

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
→ %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

Input: { "status": 400 }

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
→ call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

Input: { "status": 400 }

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
→ br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

Input: { "status": 400 }

VRL Execution Model – Code Generation

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Va
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcas
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
→ ret void
}
```

Input: { "status": 400 }

Performance Characteristics of Modern CPUs

DATADOG

Performance Characteristics of Modern CPUs

Latencies of data access

Register
1 cycle

L1 Cache
4 cycles

L2 Cache
14 cycles

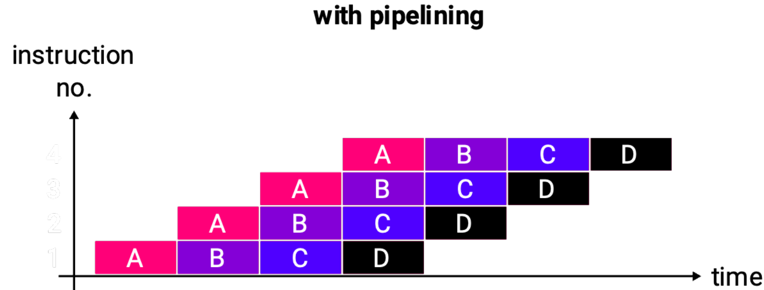
L3 Cache
~50 cycles

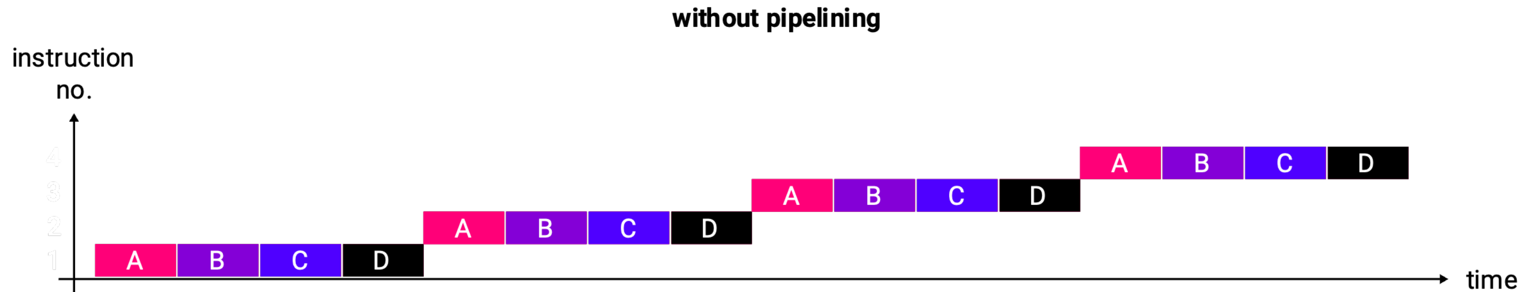
Main Memory
~200 cycles

https://cvw.cac.cornell.edu/codeopt/memtimes

Performance Characteristics of Modern CPUs

Instruction processing





https://de.wikipedia.org/wiki/Pipeline_(Prozessor)#/media/Datei:Befehlspipeline.svg

DATADOG

POV: Branch Predictor

Runtime Interpretation

Code Generation

LLVM Architecture Overview

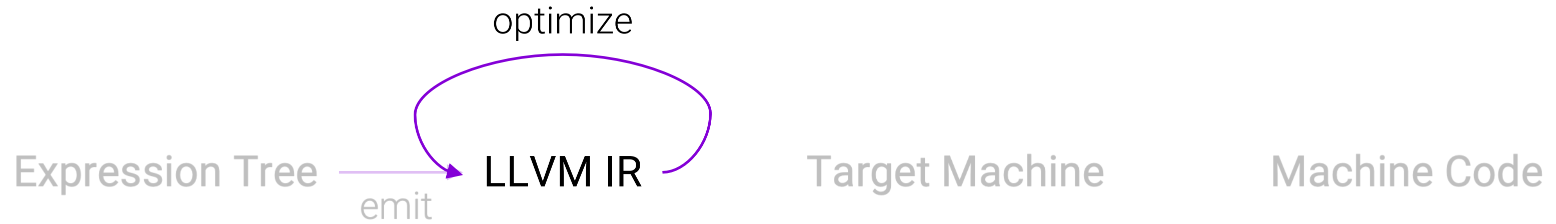
Expression Tree LLVM IR Target Machine Machine Code

LLVM Architecture Overview

Target Machine

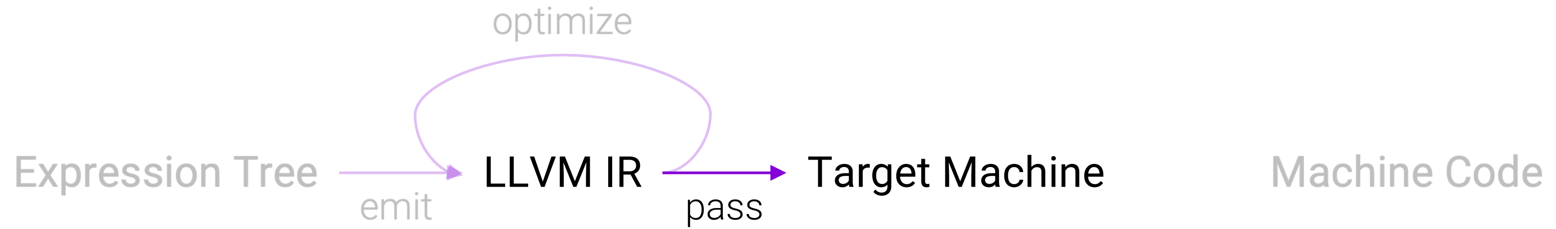
Machine Code

LLVM Architecture Overview



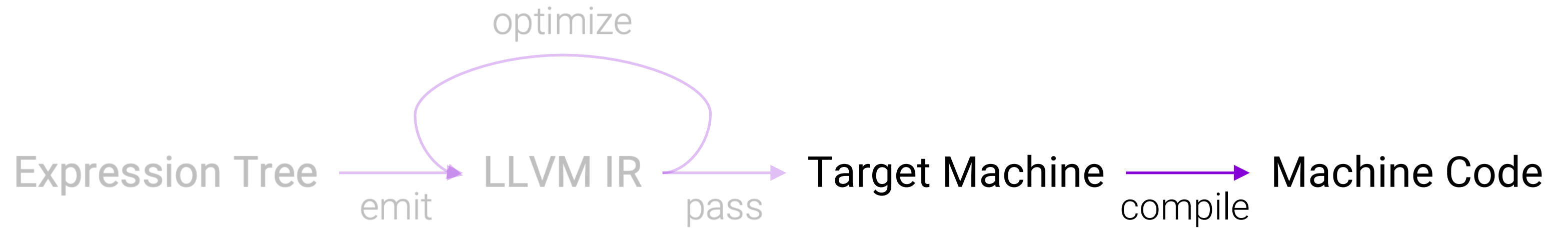
DATADOG

LLVM Architecture Overview



DATADOG

LLVM Architecture Overview



DATADOG

Code Generation – General Idea

```rust
#[no_mangle]
pub extern "C" fn vrl_resolved_initialize(result: *mut Resolved) {
    unsafe { result.write(Ok(Value::Null)) };
}

#[no_mangle]
pub extern "C" fn vrl_resolved_drop(result: *mut Resolved) {
    drop(unsafe { result.read() });
}

#[no_mangle]
pub extern "C" fn vrl_resolved_is_err(result: &mut Resolved) -> bool {
    result.is_err()
}

#[no_mangle]
pub extern "C" fn vrl_resolved_boolean_is_true(result: &Resolved) -> bool {
    result.as_ref().unwrap().as_boolean().unwrap()
}

#[no_mangle]
pub extern "C" fn vrl_expression_assignment_target_insert_external_impl(
    ctx: &mut Context,
    path: &LookupBuf,
    resolved: &Resolved,
) {
    let value = resolved.as_ref().unwrap().clone();
    let _ = ctx.target_mut().insert(path, value);
}

...
```

```llvm
; Function Attrs: uwtable
define void @vrl_resolved_initialize(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr
start:
  %self.dbg.spill.i = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  %_4.i = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::Little
  %result.dbg.spill = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  %_5 = alloca %"memmem::SearcherKind", align 8
  %_4 = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  %0 = bitcast %"memmem::SearcherKind"* %_5 to i8*, !dbg !465333
  store i8 8, i8* %0, align 8, !dbg !465333
  %1 = bitcast %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  %2 = getelementptr inbounds %"std::result::Result<vrl_compiler::Value, vrl_compiler::ExpressionError>::Ok", %"std::result
  %3 = bitcast %"memmem::SearcherKind"* %2 to i8*, !dbg !465334
  %4 = bitcast %"memmem::SearcherKind"* %_5 to i8*, !dbg !465334
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 8 %3, i8* align 8 %4, i64 40, i1 false), !dbg !465334
  %5 = bitcast %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  store i64 0, i64* %5, align 8, !dbg !465334
  call void @llvm.experimental.noalias.scope.decl(metadata !465335), !dbg !465338
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  %6 = bitcast %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  %7 = bitcast %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 8 %6, i8* align 8 %7, i64 88, i1 false), !dbg !465346
  call void @_ZN4core3ptr5write17h5e38451d006cd550E(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::E
  br label %bb1, !dbg !465338

bb1:                                              ; preds = %start
  ret void, !dbg !465348
}

; Function Attrs: uwtable
define void @vrl_resolved_drop(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line:
start:
  %self.dbg.spill.i = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  %result.dbg.spill = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  %_3 = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  call void @llvm.experimental.noalias.scope.decl(metadata !465353), !dbg !465356
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  call void @_ZN4core3ptr4read17h7567ccb1bb42450aE(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::En
  br label %bb1, !dbg !465356

bb1:                                              ; preds = %start
  call void @_ZN4core3mem4drop17h1d6703fac067675cE(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::En
  br label %bb2, !dbg !465365

bb2:                                              ; preds = %bb1
  ret void, !dbg !465366
}

; Function Attrs: uwtable
define zeroext i1 @vrl_resolved_is_err(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
start:
  %result.dbg.spill = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  %0 = call zeroext i1 @"_ZN4core6result19Result$LT$T$C$E$GT$6is_err17h17f59dc30d2e082fE"(%"core::result::Result<addr2line:
  br label %bb1, !dbg !465390

bb1:                                              ; preds = %start
  ret i1 %0, !dbg !465391
}

; Function Attrs: uwtable
define zeroext i1 @vrl_resolved_boolean_is_true(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::Endia
start:
  %result.dbg.spill = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  %0 = call { i64, i8* } @"_ZN4core6result19Result$LT$T$C$E$GT$6as_ref17h23eef4ff0fa33476E"(%"core::result::Result<addr2lin
  %_5.0 = extractvalue { i64, i8* } %0, 0, !dbg !465396
  %_5.1 = extractvalue { i64, i8* } %0, 1, !dbg !465396
  br label %bb1, !dbg !465396

bb1:                                              ; preds = %start
  %_4 = call align 8 dereferenceable(40) %"memmem::SearcherKind"* @"_ZN4core6result19Result$LT$T$C$E$GT$6unwrap17hfd2cb2ff2
  br label %bb2, !dbg !465396

bb2:                                              ; preds = %bb1
```

DATADOG

Code Generation – General Idea

```rust
#[no_mangle]
pub extern "C" fn vrl_resolved_initialize(result: *mut Resolved) {
    unsafe { result.write(Ok(Value::Null)) };
}

#[no_mangle]
pub extern "C" fn vrl_resolved_drop(result: *mut Resolved) {
    drop(unsafe { result.read() });
}

#[no_mangle]
pub extern "C" fn vrl_resolved_is_err(result: &mut Resolved) -> bool {
    result.is_err()
}

#[no_mangle]
pub extern "C" fn vrl_resolved_boolean_is_true(result: &Resolved) -> bool {
    result.as_ref().unwrap().as_boolean().unwrap()
}

#[no_mangle]
pub extern "C" fn vrl_expression_assignment_target_insert_external_impl(
    ctx: &mut Context,
    path: &LookupBuf,
    resolved: &Resolved,
) {
    let value = resolved.as_ref().unwrap().clone();
    let _ = ctx.target_mut().insert(path, value);
}

...
```

```llvm
; Function Attrs: uwtable
define void @vrl_resolved_initialize(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr
start:
  %self.dbg.spill.i = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  %_4.i = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::Little
  %result.dbg.spill = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  %_5 = alloca %"memmem::SearcherKind", align 8
  %_4 = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  %0 = bitcast %"memmem::SearcherKind"* %_5 to i8*, !dbg !465333
  store i8 8, i8* %0, align 8, !dbg !465333
  %1 = bitcast %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  %2 = getelementptr inbounds %"std::result::Result<vrl_compiler::Value, vrl_compiler::ExpressionError>::Ok", %"std::result
  %3 = bitcast %"memmem::SearcherKind"* %2 to i8*, !dbg !465334
  %4 = bitcast %"memmem::SearcherKind"* %_5 to i8*, !dbg !465334
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 8 %3, i8* align 8 %4, i64 40, i1 false), !dbg !465334
  %5 = bitcast %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  store i64 0, i64* %5, align 8, !dbg !465334
  call void @llvm.experimental.noalias.scope.decl(metadata !465335), !dbg !465338
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  %6 = bitcast %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  %7 = bitcast %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 8 %6, i8* align 8 %7, i64 88, i1 false), !dbg !465346
  call void @_ZN4core3ptr5write17h5e38451d006cd550E(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::E
  br label %bb1, !dbg !465338

bb1:                                              ; preds = %start
  ret void, !dbg !465348
}

; Function Attrs: uwtable
define void @vrl_resolved_drop(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line:
start:
  %self.dbg.spill.i = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  %result.dbg.spill = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  %_3 = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEn
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  call void @llvm.experimental.noalias.scope.decl(metadata !465353), !dbg !465356
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  call void @_ZN4core3ptr4read17h7567ccb1bb42450aE(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::En
  br label %bb1, !dbg !465356

bb1:                                              ; preds = %start
  call void @_ZN4core3mem4drop17h1d6703fac067675cE(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::En
  br label %bb2, !dbg !465365

bb2:                                              ; preds = %bb1
  ret void, !dbg !465366
}

; Function Attrs: uwtable
define zeroext i1 @vrl_resolved_is_err(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
start:
  %result.dbg.spill = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  %0 = call zeroext i1 @"_ZN4core6result19Result$LT$T$C$E$GT$6is_err17h17f59dc30d2e082fE"(%"core::result::Result<addr2line:
  br label %bb1, !dbg !465390

bb1:                                              ; preds = %start
  ret i1 %0, !dbg !465391
}

; Function Attrs: uwtable
define zeroext i1 @vrl_resolved_boolean_is_true(%"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::Endia
start:
  %result.dbg.spill = alloca %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::g
  store %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<addr2line::gimli::LittleEndian>,
  call void @llvm.dbg.declare(metadata %"core::result::Result<addr2line::gimli::UnitHeader<addr2line::gimli::EndianSlice<ad
  %0 = call { i64, i8* } @"_ZN4core6result19Result$LT$T$C$E$GT$6as_ref17h23eef4ff0fa33476E"(%"core::result::Result<addr2lin
  %_5.0 = extractvalue { i64, i8* } %0, 0, !dbg !465396
  %_5.1 = extractvalue { i64, i8* } %0, 1, !dbg !465396
  br label %bb1, !dbg !465396

bb1:                                              ; preds = %start
  %_4 = call align 8 dereferenceable(40) %"memmem::SearcherKind"* @"_ZN4core6result19Result$LT$T$C$E$GT$6unwrap17hfd2cb2ff2
  br label %bb2, !dbg !465396

bb2:                                              ; preds = %bb1
```

Code Generation – General Idea

```
#[no_mangle]
pub extern "C" fn vrl_resolved_initialize(result: *mut Resolved) {
    unsafe { result.write(Ok(Value::Null)) };
}

#[no_mangle]
pub extern "C" fn vrl_resolved_drop(result: *mut Resolved) {
    drop(unsafe { result.read() });
}

#[no_mangle]
pub extern "C" fn vrl_resolved_is_err(result: &mut Resolved) -> bool {
    result.is_err()
}

#[no_mangle]
pub extern "C" fn vrl_resolved_boolean_is_true(result: &Resolved) -> bool {
    result.as_ref().unwrap().as_boolean().unwrap()
}

#[no_mangle]
pub extern "C" fn vrl_expression_assignment_target_insert_external_impl(
    ctx: &mut Context,
    path: &LookupBuf,
    resolved: &Resolved,
) {
    let value = resolved.as_ref().unwrap().clone();
    let _ = ctx.target_mut().insert(path, value);
}

...
```

```
define void @vrl_execute(
  %"Context"* align 8 %context,
  %"Result<Value>"* align 8 %result
) {
start:
  ret void
}
```

Code Generation – Function Calls

```
let fn_ident = "vrl_resolved_initialize";
let fn_impl = ctx
  .module()
  .get_function(fn_ident)
  .ok_or(format!(r#"failed to get "{}" function"#, fn_ident))?;

ctx.builder()
  .build_call(fn_impl, &[argument_ref.into()], fn_ident)
```

```
call void @vrl_resolved_initialize(%"Result<Value>"** %resolved)
```

DATADOG

Code Generation – Literals

```
let size = std::mem::size_of::<T>();
let global_type = self.context.i8_type().array_type(size as _);
let global = self.module.add_global(global_type, None, name);
global.set_linkage(inkwell::module::Linkage::Private);
global.set_alignment(std::mem::align_of::<T>() as _);
let value = self.into_i8_array_value(value);
global.set_initializer(&value);
global
```

```
@"200" = private global [40 x i8]
c"\01\E2\F7\06\00`\00\00\C8\00\00\00\00\00\00\00\00\00\00\00\00
\00\00\00\00\00\EDn\01\00\00\00\00\80q\EC\01\00\00\00", align 8
```

DATADOG

Code Generation – Allocations

```
let resolved_type = self
  .function()
  .get_nth_param(1)
  .unwrap()
  .get_type();

self.builder.build_alloca(resolved_type, name)
```

```
%result = alloca %"Result<Value>"*, align 8
```

DATADOG

Code Generation – Branching

```
let if_block = ctx
  .context()
  .append_basic_block(function, "if_block");

let else_block = ctx
  .context()
  .append_basic_block(function, "else_block");

ctx.builder()
  .build_conditional_branch(predicate, if_block, else_block);
```

```
 br i1 %predicate, label %if_block, label %else_block

if_statement_if_branch:

block_end:
```

DATADOG

Code Generation – Full Example

```
if .status == 200 {
  .message = "ok"
}
```

```
define void @vrl_execute(%"Context"* align 8 %context, %"Result<Value>"* align 8 %result) {
start:
  call void @vrl_expression_query_target_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]* @status to %"Looku
  %rhs = alloca %"Result<Value>"*, align 8
  call void @vrl_resolved_initialize(%"Result<Value>"** %rhs)
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"200" to %"Value"*), %"Result<Value>"** %rhs)
  call void @vrl_expression_op_eq_impl(%"Result<Value>"** %rhs, %"Result<Value>"* %result)
  %vrl_resolved_boolean_is_true = call i1 @vrl_resolved_boolean_is_true(%"Result<Value>"* %result)
  call void @vrl_resolved_drop(%"Result<Value>"** %rhs)
  br i1 %vrl_resolved_boolean_is_true, label %if_statement_if_branch, label %block_end

if_statement_if_branch: ; preds = %start
  call void @vrl_expression_literal_impl(%"Value"* bitcast ([40 x i8]* @"\22ok\22" to %"Value"*), %"Result<Value>"* %result)
  call void @vrl_expression_assignment_target_insert_external_impl(%"Context"* %context, %"LookupBuf"* bitcast ([32 x i8]* @messa
  br label %block_end

block_end: ; preds = %if_statement_if_branch, %start
  ret void
}
```

Questions