Beyond Trees:

Calculating Graph-Based Compilers

FUNCTIONAL PEARL

Patrick Bahr
IT University of Copenhagen, Denmark

Graham Hutton
University of Nottingham, UK

Compiler Calculation

What is Compiler Calculation?

[Bahr, Hutton. *Calculating Correct Compilers*. 2015]

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr


eval :: Expr → Int
eval (Val n)     = n
eval (Add x y)   = eval x + eval y
```

What is Compiler Calculation?

[Bahr, Hutton. *Calculating Correct Compilers*. 2015]

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr

eval :: Expr → Int
eval (Val n)     = n
eval (Add x y)   = eval x + eval y
```

Syntax

What is Compiler Calculation?

[Bahr, Hutton. *Calculating Correct Compilers*. 2015]

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr
```

Syntax

```
eval :: Expr → Int
eval (Val n)     = n
eval (Add x y)   = eval x + eval y
```

Semantics

What is Compiler Calculation?

[Bahr, Hutton. *Calculating Correct Compilers*. 2015]

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr


eval :: Expr → Int
eval (Val n)     = n
eval (Add x y)   = eval x + eval y
```

Compiler

```
compile :: Expr → Code
compile (Val n)     = ???
compile (Add x y)   = ???
```

What is Compiler Calculation?

[Bahr, Hutton. *Calculating Correct Compilers*. 2015]

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr


eval :: Expr → Int
eval (Val n)     = n
eval (Add x y)   = eval x + eval y
```

Compiler

```
compile :: Expr → Code
compile (Val n)     = ???
compile (Add x y)   = ???
```

Proof

Compiler Spec

```
eval e = exec (compile e)
```

What is Compiler Calculation?

[Bahr, Hutton. *Calculating Correct Compilers*. 2015]

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr


eval :: Expr → Int
eval (Val n)     = n
eval (Add x y)   = eval x + eval y
```

Compiler

```
compile :: Expr → Code
compile (Val n)     = ???
compile (Add x y)   = ???
```

Compiler Spec

```
eval e = exec (compile e)
```

What is Compiler Calculation?

[Bahr, Hutton. *Calculating Correct Compilers*. 2015]

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr


eval :: Expr → Int
eval (Val n)     = n
eval (Add x y)   = eval x + eval y
```

Compiler

```
compile :: Expr → Code
compile (Val n)     = ???
compile (Add x y)   = ???
```

Compiler Spec

```
eval e = exec (compile e)
```

Prove the spec before we have a compiler.

What is Compiler Calculation?

[Bahr, Hutton. *Calculating Correct Compilers*. 2015]

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr


eval :: Expr → Int
eval (Val n)     = n
eval (Add x y)   = eval x + eval y
```

Compiler

```
compile :: Expr → Code
compile (Val n)     = ???
compile (Add x y)   = ???
```

Prove the spec before we have a compiler.

Compiler Spec

```
eval e = exec (compile e)
```

Compiler definition falls out of the <u>equational reasoning</u>.

Example Calculation

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr
          | If  Expr Expr Expr

eval :: Expr → Int
eval (Val n)      = n
eval (Add x y)    = eval x + eval y
eval (If x y z) = if eval x == 0
                  then eval z
                  else eval y
```

Example Calculation

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr
          | If  Expr Expr Expr

eval :: Expr → Int
eval (Val n)      = n
eval (Add x y)    = eval x + eval y
eval (If x y z) = if eval x == 0
                  then eval z
                  else eval y
```

Compiler Spec

```
exec c (eval e : s) = exec (comp e c) s
```

Example Calculation

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr
          | If  Expr Expr Expr

eval :: Expr → Int
eval (Val n)      = n
eval (Add x y)    = eval x + eval y
eval (If x y z) = if eval x == 0
                  then eval z
                  else eval y
```

Prove the spec before we have a compiler .

Compiler Spec

```
exec c (eval e : s) = exec (comp e c) s
```

Example Calculation

Syntax & Semantics

```
data Expr = Val Int
          | Add Expr Expr
          | If  Expr Expr Expr

eval :: Expr → Int
eval (Val n)      = n
eval (Add x y)    = eval x + eval y
eval (If x y z) = if eval x == 0
                  then eval z
                  else eval y
```

Prove the spec before we have a compiler .

Compiler Spec

```
exec c (eval e : s) = exec (comp e c) s
```

Compiler + Target Code

```
comp :: Expr → Code → Code
```

```
data Code = …
```

```
exec :: Code → Stack → Stack
```

Compiler + target machine falls out of the <u>equational reasoning</u>.

Calculating....

Result of the Compiler Calculation

Target Language

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Result of the Compiler Calculation

Target Language

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Compiler

```
comp :: Expr → Code → Code
comp (Val n)    c = PUSH n c
comp (Add x y)  c = comp x (comp y (ADD c))
comp (If x y z) c = comp x (JPZ (comp z c) (comp y c))

compile :: Expr → Code
compile e = comp e HALT
```

Result of the Compiler Calculation

Target Language

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Virtual Machine

```
exec :: Code → Stack → Stack
exec (PUSH n c) s         = exec c (n : s)
exec (ADD c) (m : n : s) = exec c ((n + m) : s)
exec (JPZ c' c)  (n : s) = if n == 0 then exec c' s else exec c s
exec HALT s               = s
```

Compiler

```
comp :: Expr → Code → Code
comp (Val n)     c = PUSH n c
comp (Add x y)   c = comp x (comp y (ADD c))
comp (If x y z) c = comp x (JPZ (comp z c) (comp y c))

compile :: Expr → Code
compile e = comp e HALT
```

Result of the Compiler Calculation

Target Language

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Compiler

```
comp :: Expr → Code → Code
comp (Val n)     c = PUSH n c
comp (Add x y)   c = comp x (comp y (ADD c))
comp (If x y z) c = comp x (JPZ (comp z c) (comp y c))

compile :: Expr → Code
compile e = comp e HALT
```

Virtual Machine

```
exec :: Code → Stack → Stack
exec (PUSH n c) s         = exec c (n : s)
exec (ADD c) (m : n : s) = exec c ((n + m) : s)
exec (JPZ c' c)  (n : s) = if n == 0 then exec c' s else exec c s
exec HALT s               = s
```

Correctness Property

```
exec c (eval e : s) = exec (comp e c) s
```

✓

Result of the Compiler Calculation

Target Language

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Compiler

```
comp :: Expr → Code → Code
comp (Val n)    c = PUSH n c
comp (Add x y)  c = comp x (comp y (ADD c))
comp (If x y z) c = comp x (JPZ (comp z c) (comp y c))

compile :: Expr → Code
compile e = comp e HALT
```

Result of the Compiler Calculation

Target Language

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Instructions contain *code continuations*

(= the code to be executed afterwards)

Compiler

```
comp :: Expr → Code → Code
comp (Val n)    c = PUSH n c
comp (Add x y)  c = comp x (comp y (ADD c))
comp (If x y z) c = comp x (JPZ (comp z c) (comp y c))

compile :: Expr → Code
compile e = comp e HALT
```

Result of the Compiler Calculation

Target Language

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Instructions contain *code continuations*
(= the code to be executed afterwards)

Branching instructions have *several* code continuations

Compiler

```
comp :: Expr → Code → Code
comp (Val n)     c = PUSH n c
comp (Add x y)   c = comp x (comp y (ADD c))
comp (If x y z)  c = comp x (JPZ (comp z c) (comp y c))

compile :: Expr → Code
compile e = comp e HALT
```

Result of the Compiler Calculation

Target Language

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Instructions contain *code continuations*

(= the code to be executed afterwards)

Branching instructions have *several* code continuations

Compiler

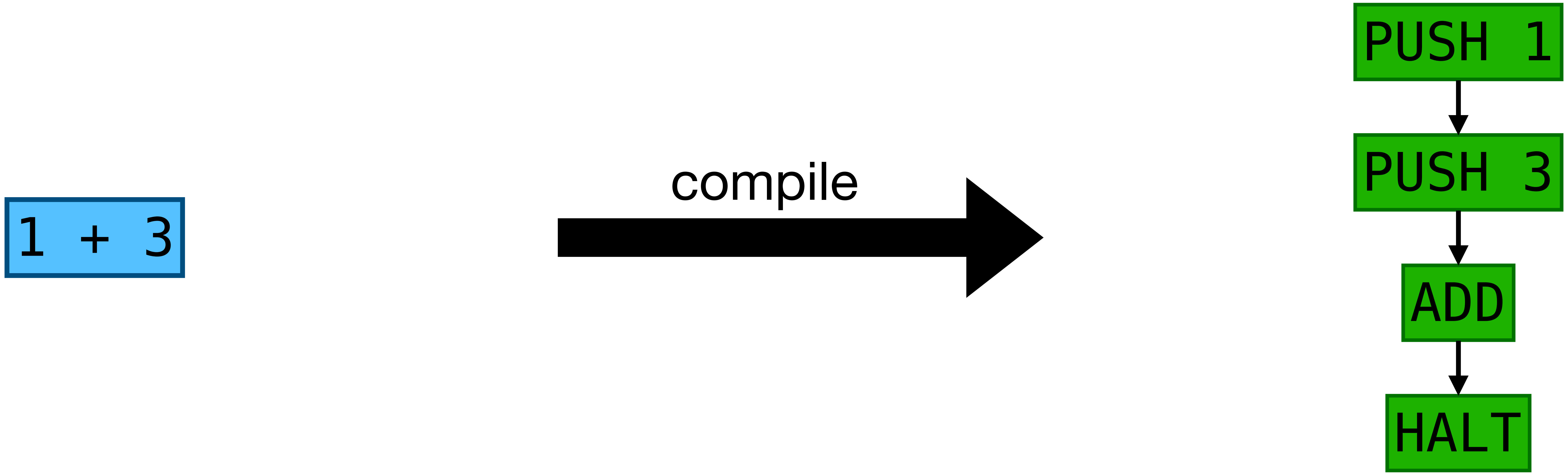
```
comp :: Expr → Code → Code
comp (Val n)     c = PUSH n c
comp (Add x y)   c = comp x (comp y (ADD c))
comp (If x y z) c = comp x (JPZ (comp z c) (comp y c))

compile :: Expr → Code
compile e = comp e HALT
```

Compiler duplicates code

Example

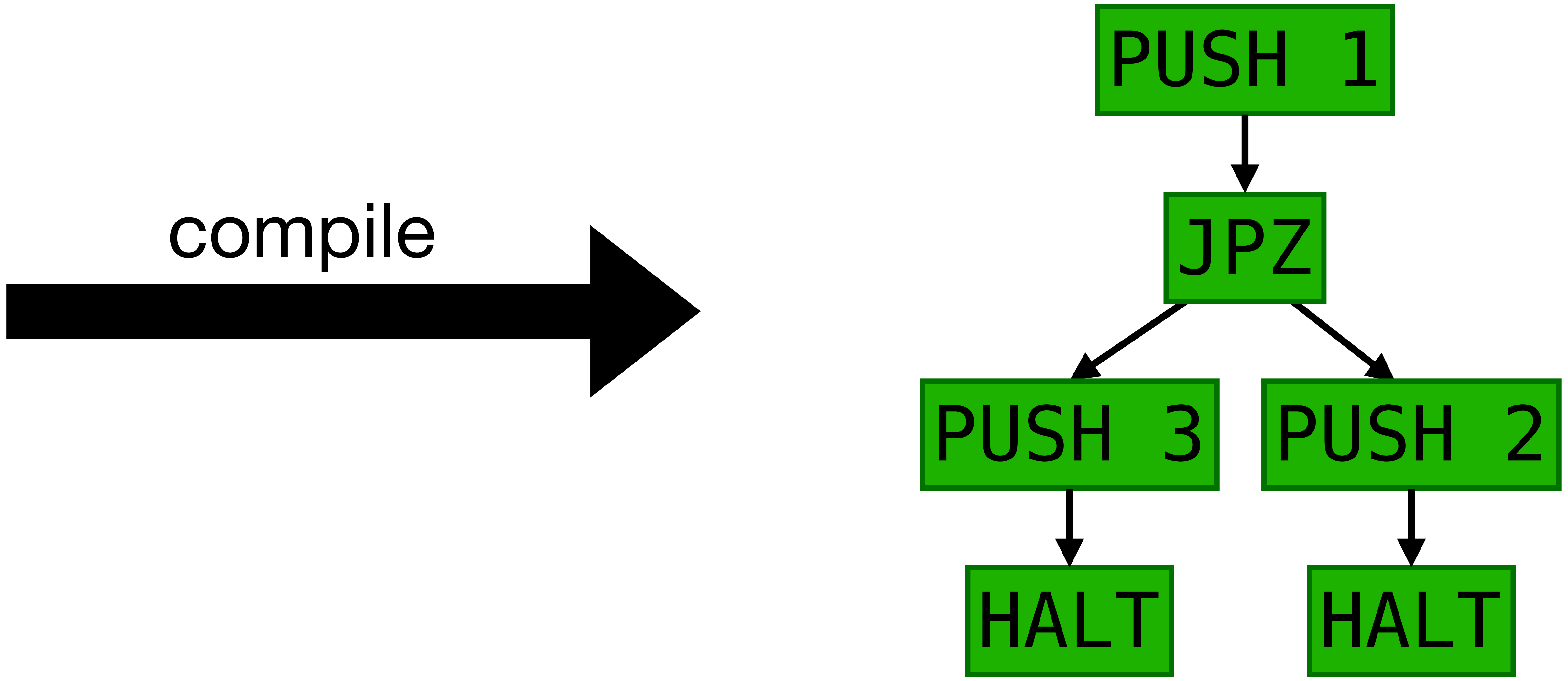
```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```



Example

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

```
if 1 then 2 else 3
```

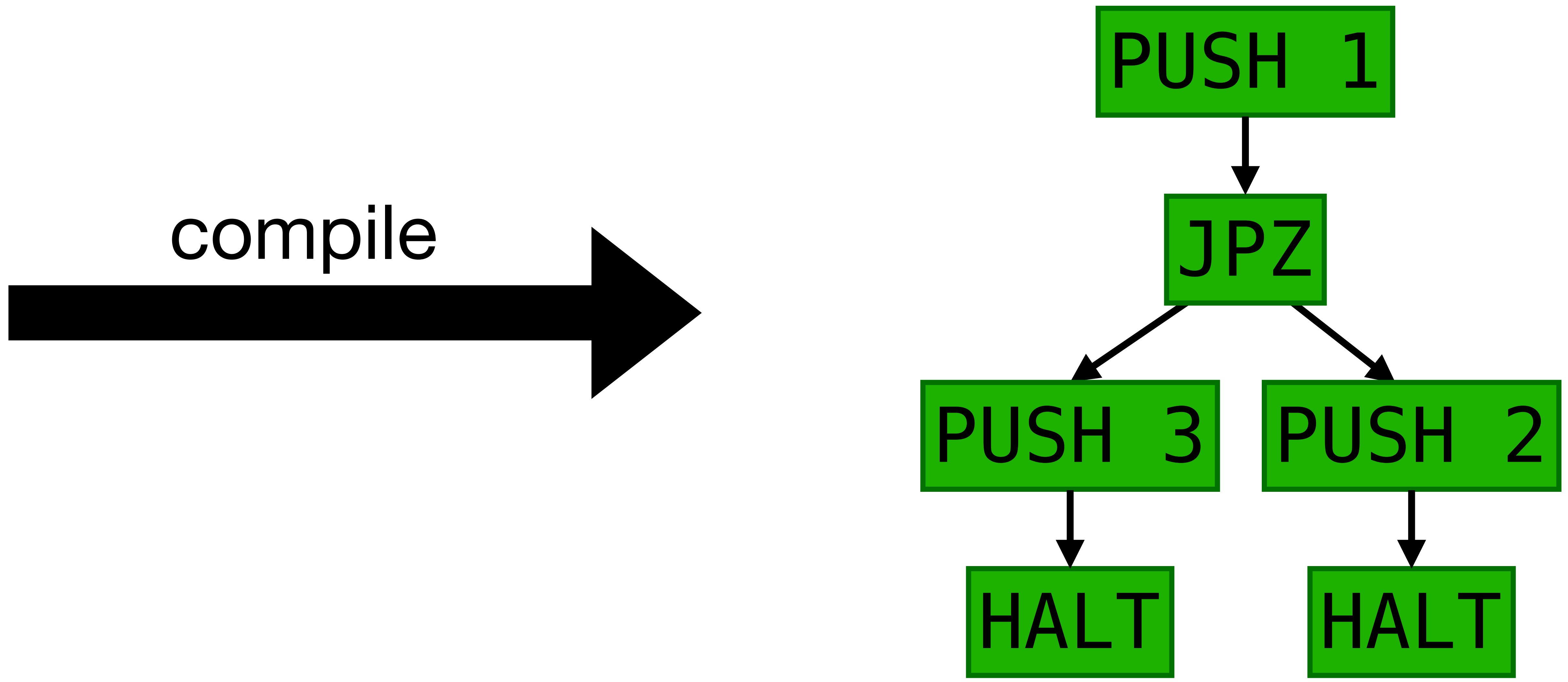


Example

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

if 1 *then* 2 *else* 3

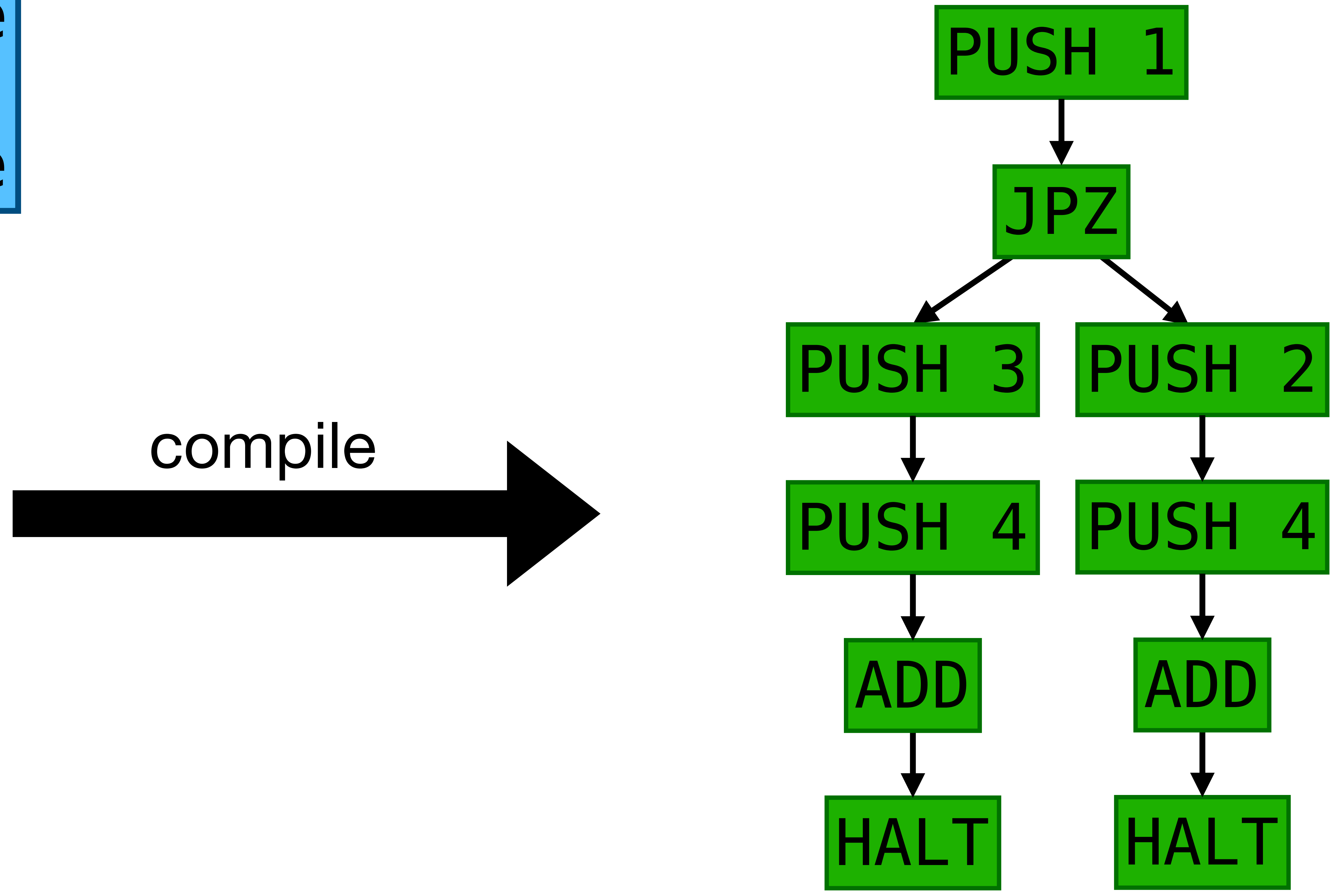
Code is tree-shaped



Example

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

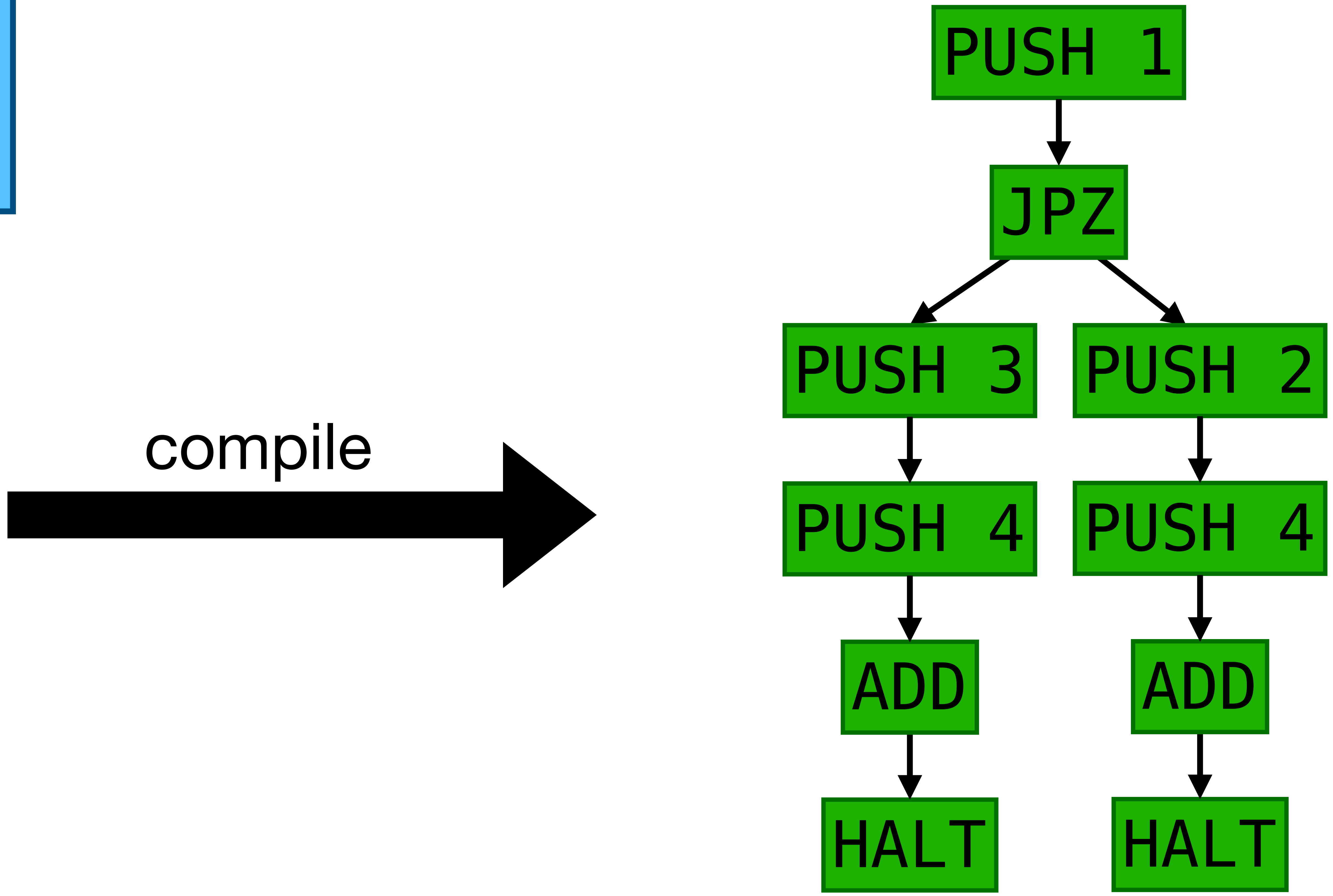
(*if* 1 *then* 2 *else* 3) + 4



Example

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

```
(if 1 then 2 else 3) + 4
```

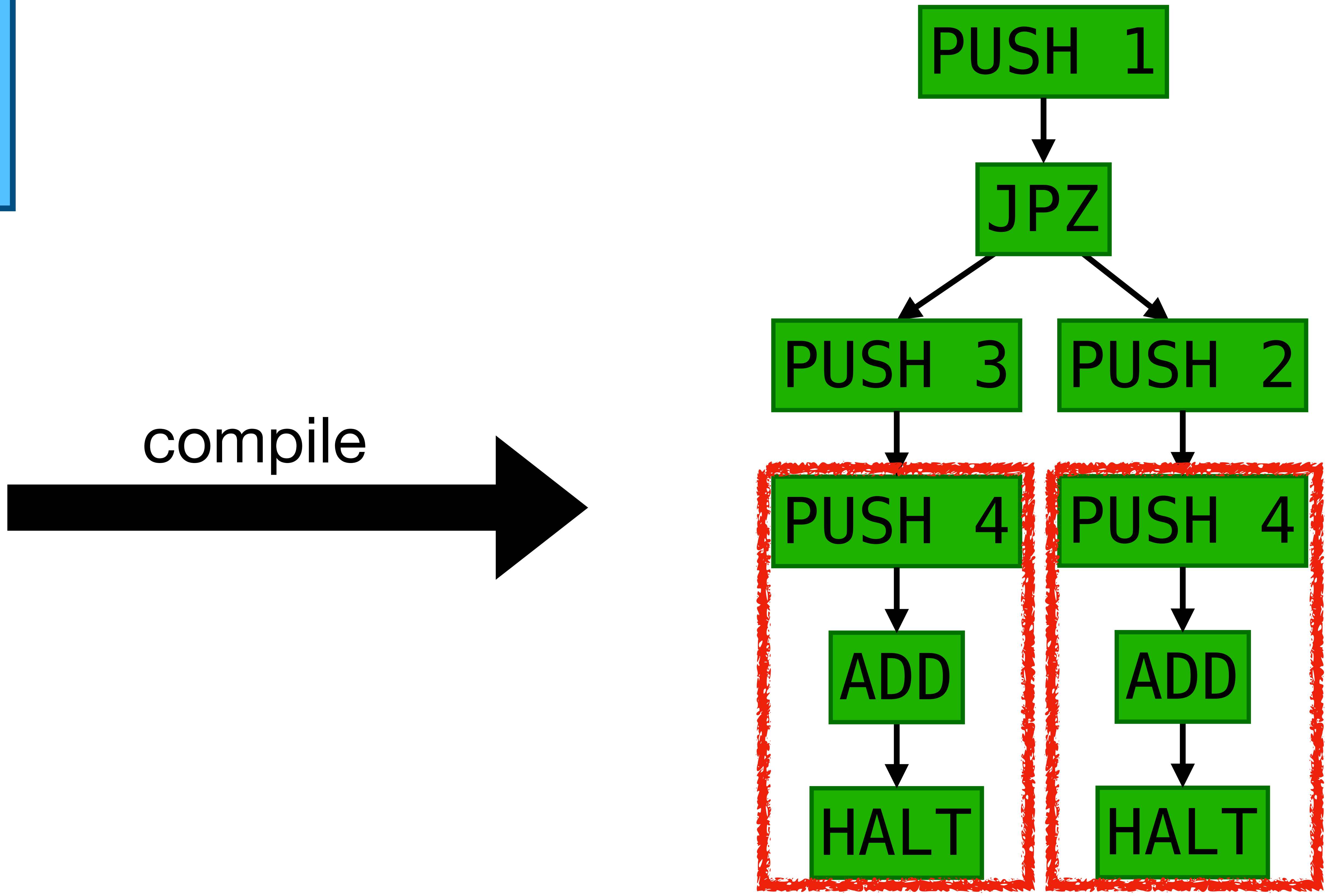


Example

Code is duplicated!

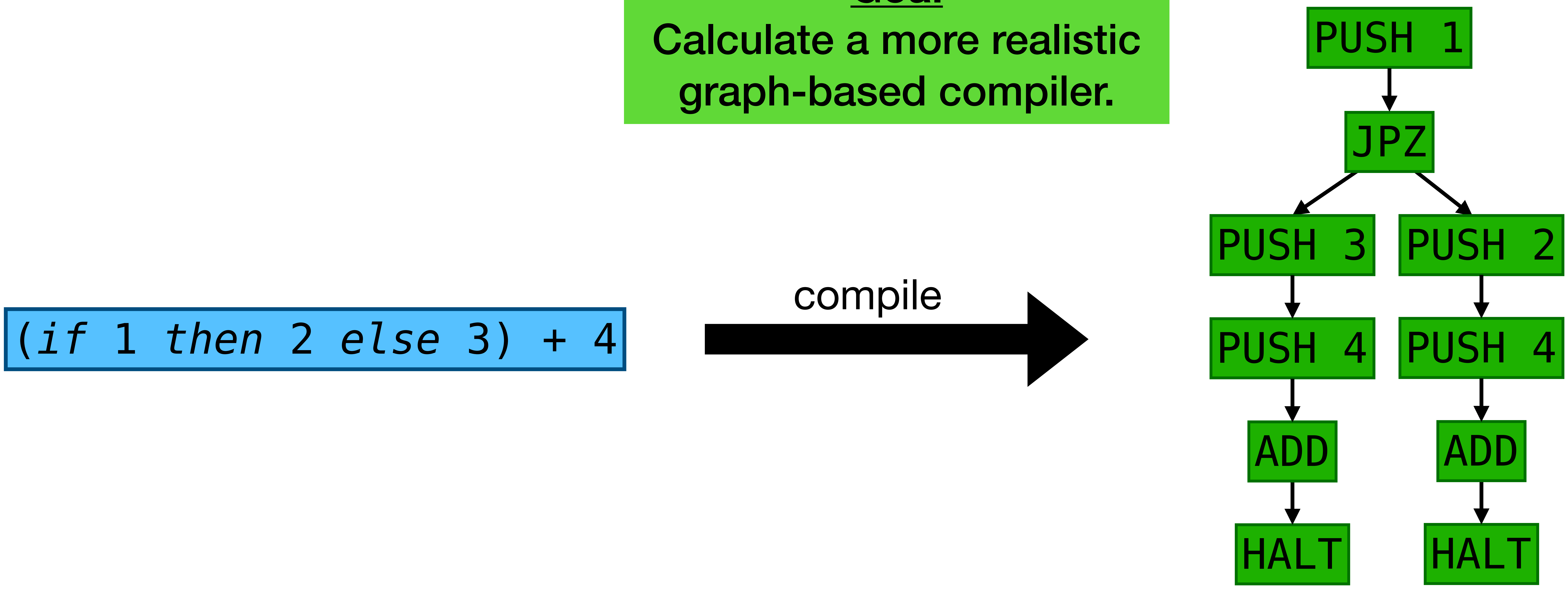
```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

(*if* 1 *then* 2 *else* 3) + 4



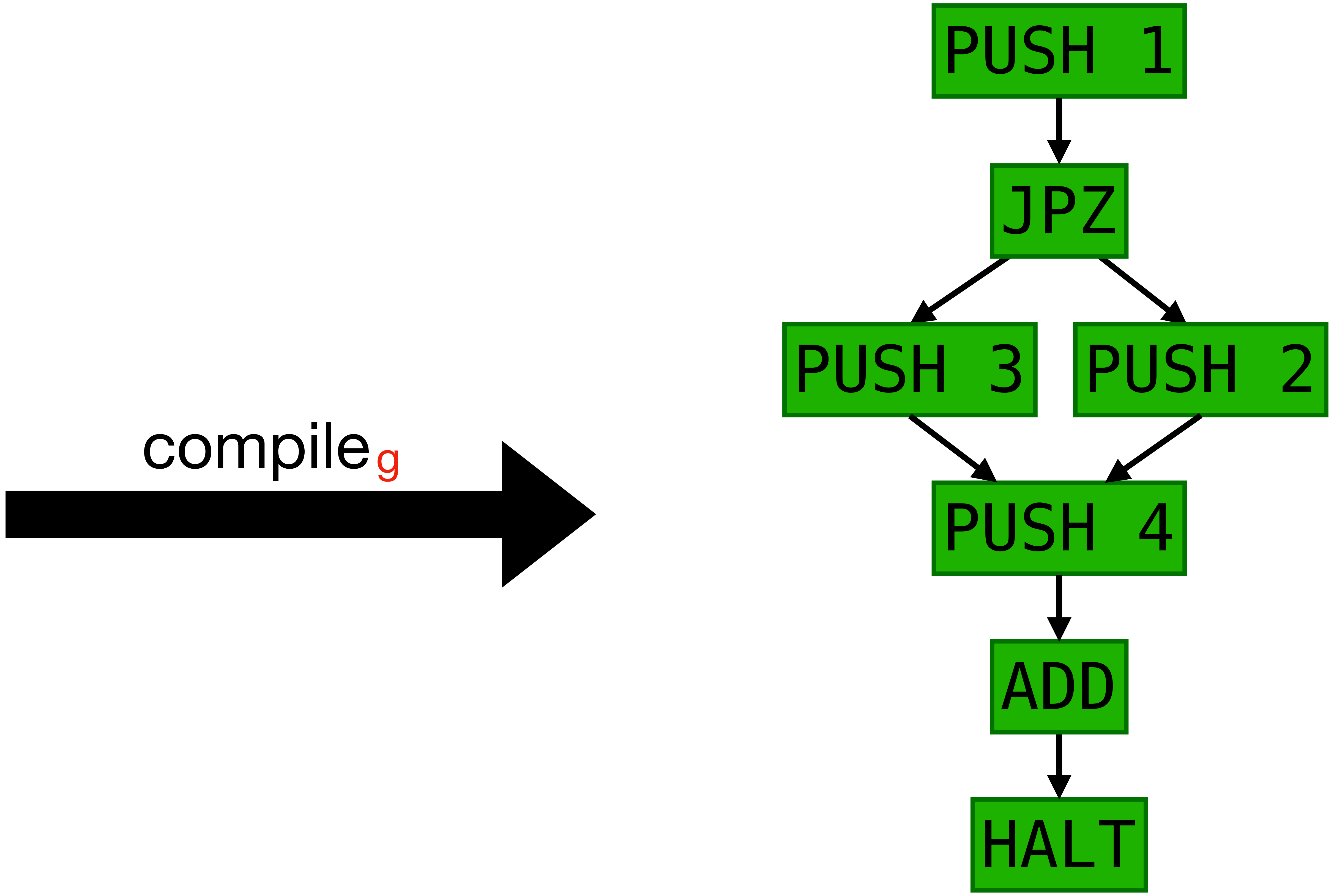
From Trees to Graphs

Goal
Calculate a more realistic graph-based compiler.



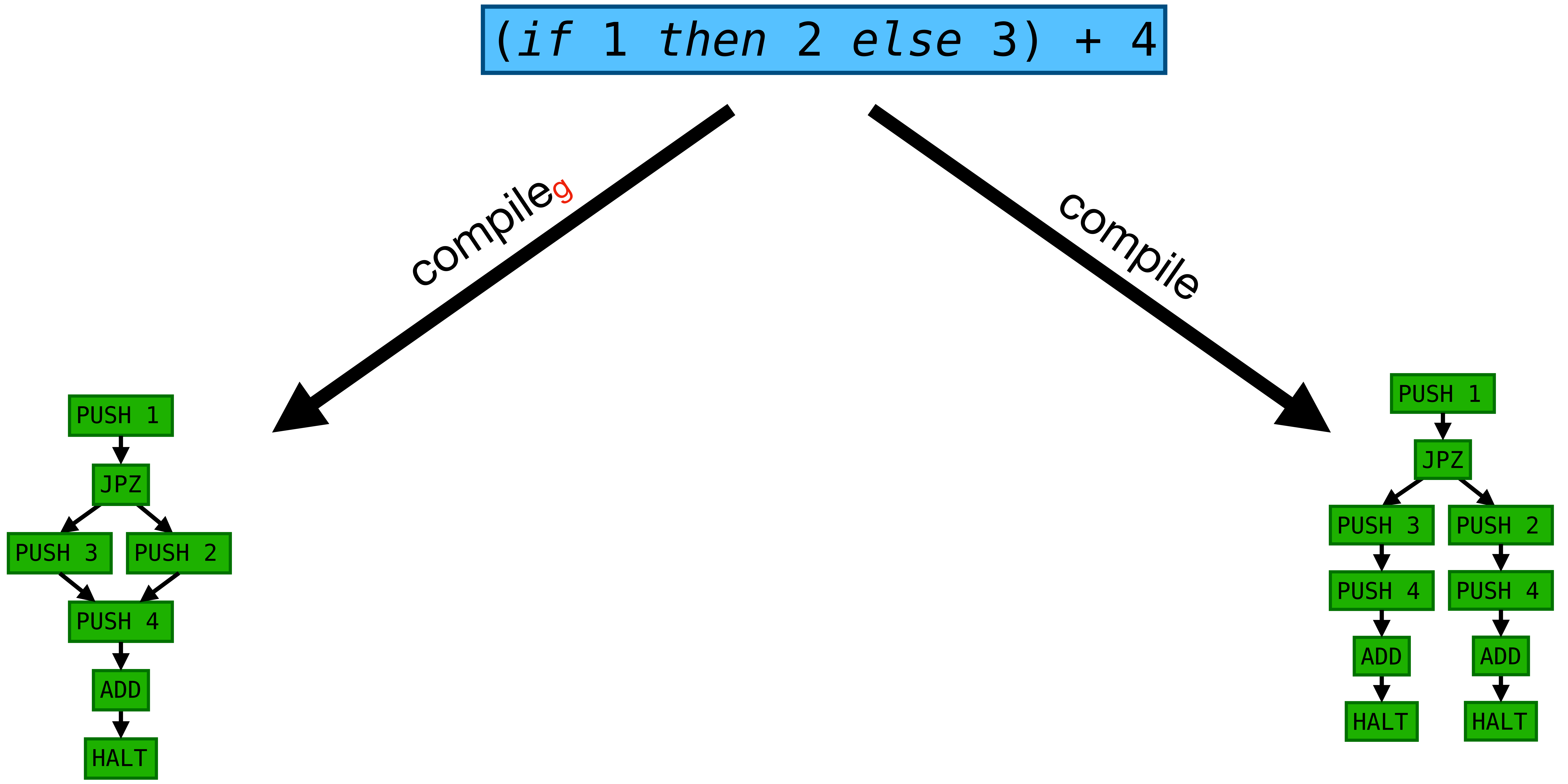
From Trees to Graphs

`(if 1 then 2 else 3) + 4`

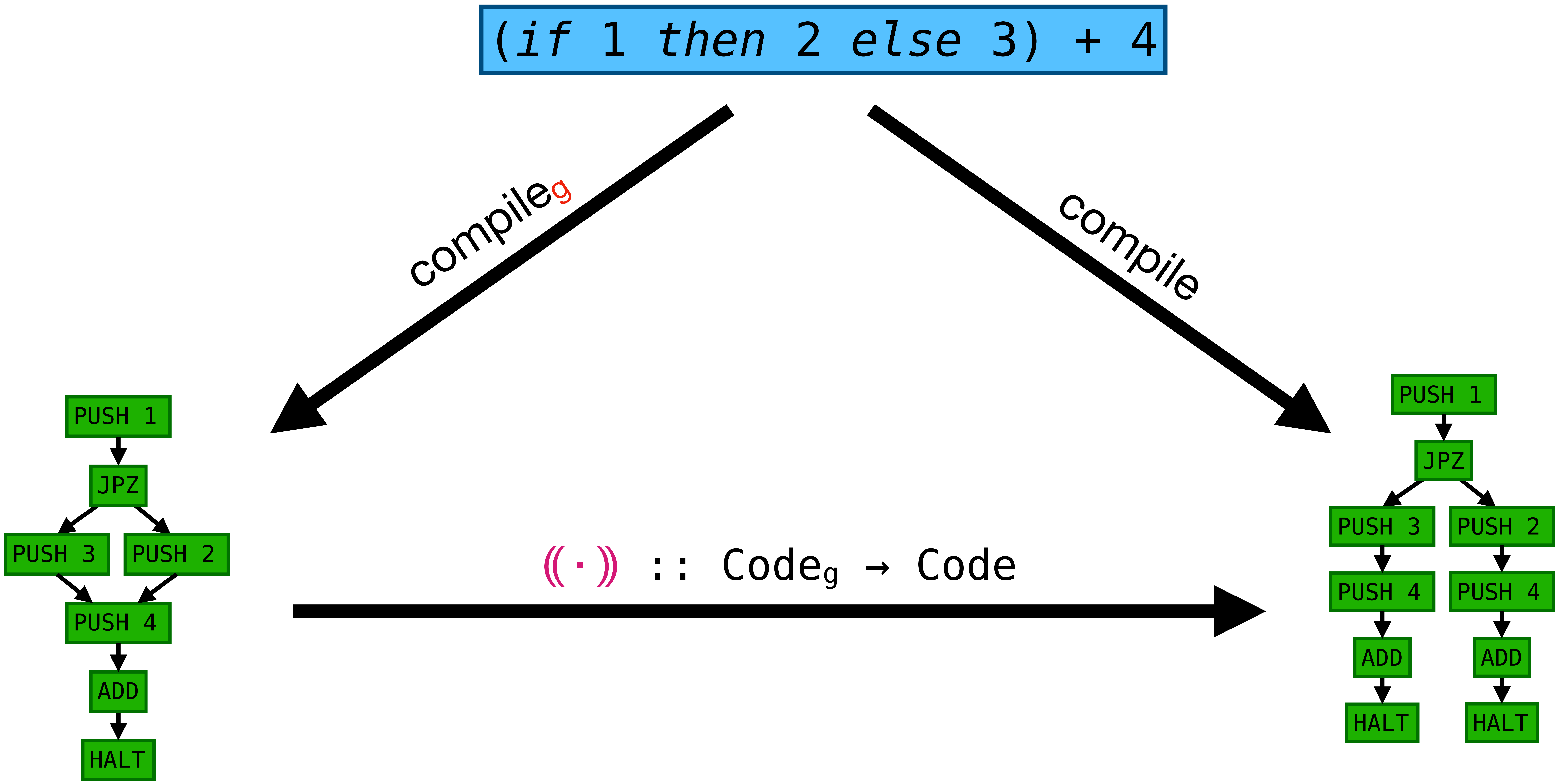


Compiler Spec

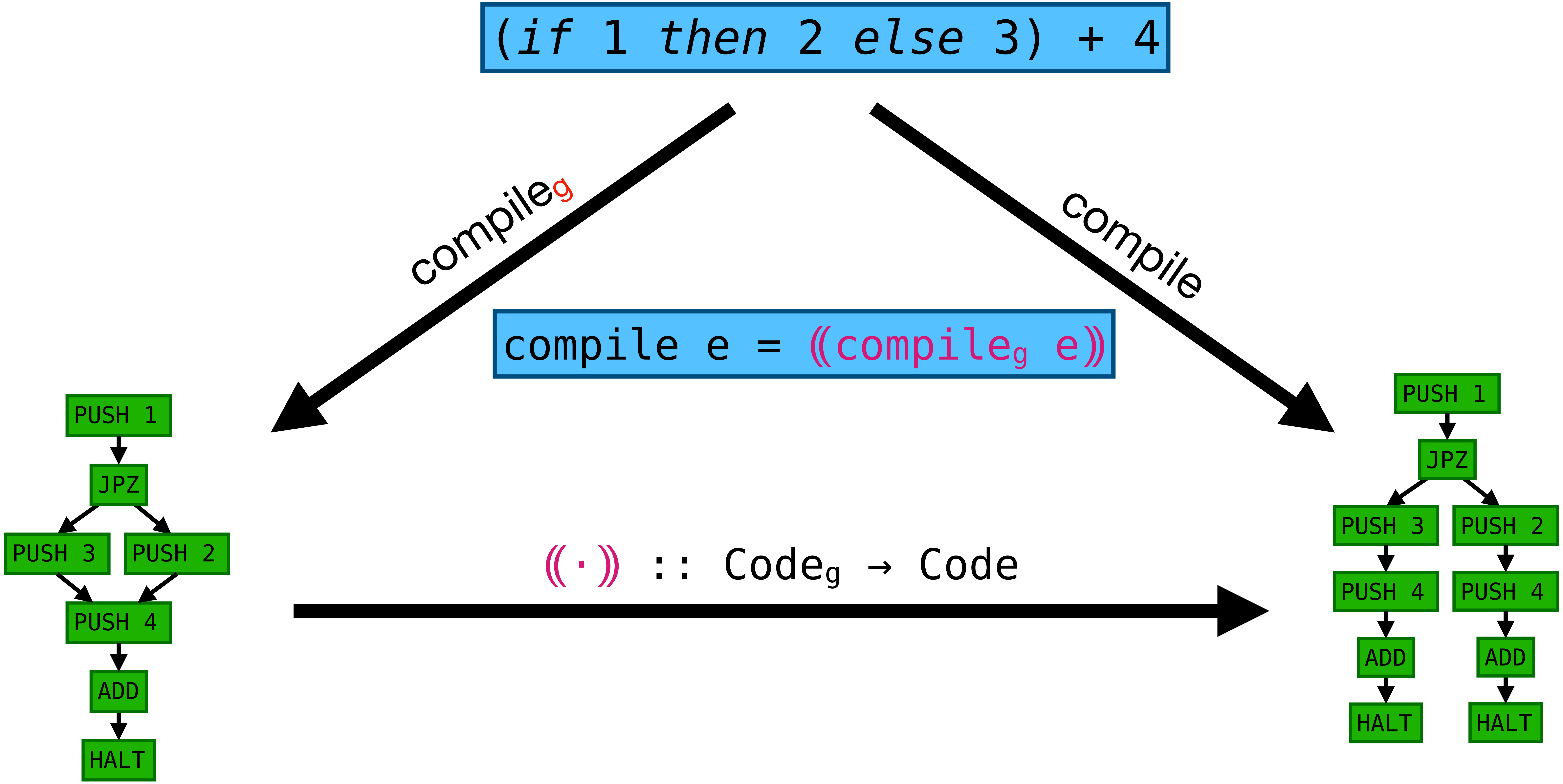
Correctness Specification



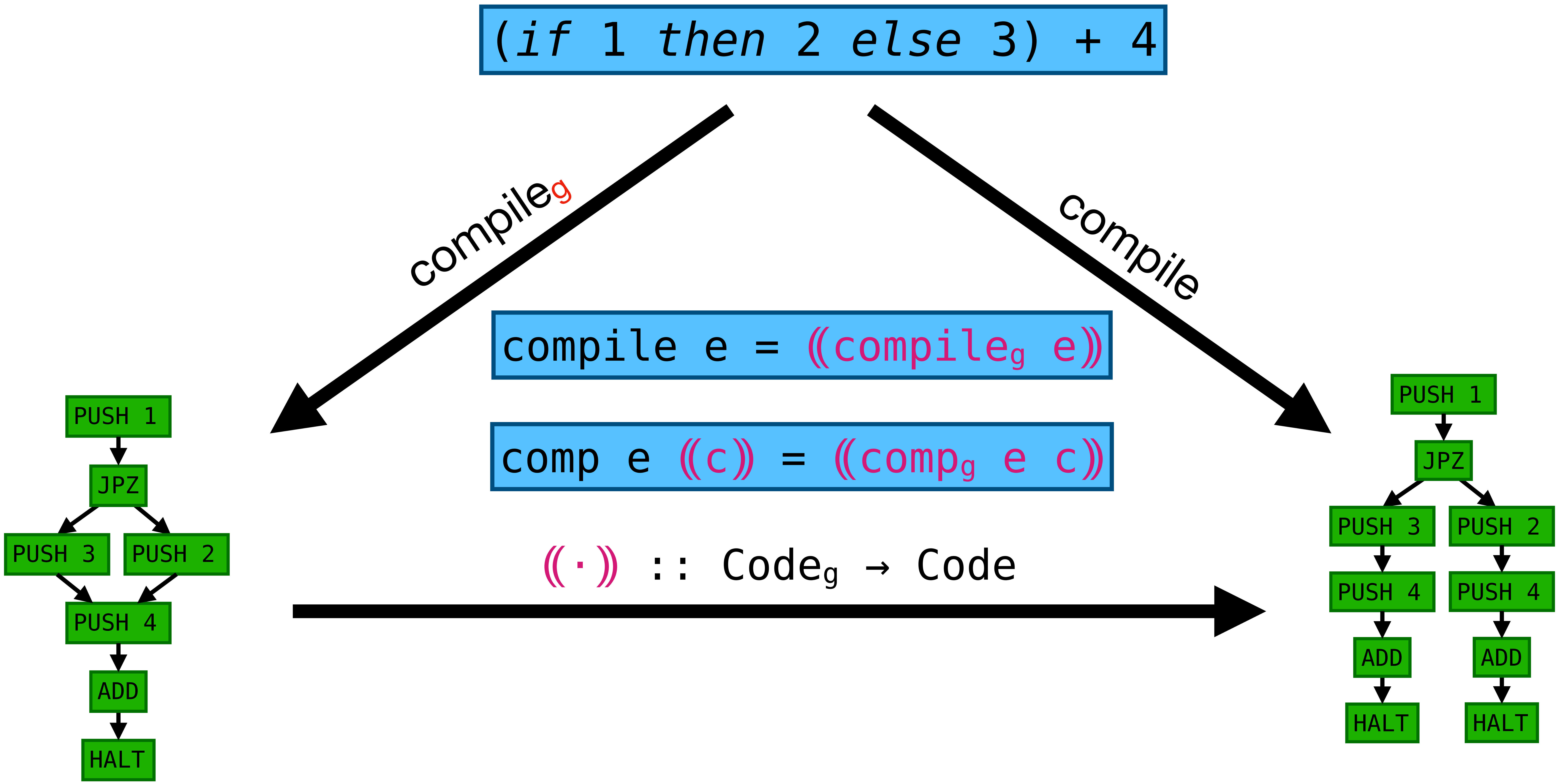
Correctness Specification



Correctness Specification



Correctness Specification



Calculating a Graph-Based Compiler

Tree-Based Compiler

```
comp :: Expr → Code → Code
comp (Val n)      c = PUSH n c
comp (Add x y)    c = ...
comp (If x y z) c = ...
```

Calculating a Graph-Based Compiler

Tree-Based Compiler

```
comp :: Expr → Code → Code
comp (Val n)      c = PUSH n c
comp (Add x y)    c = ...
comp (If x y z)   c = ...
```

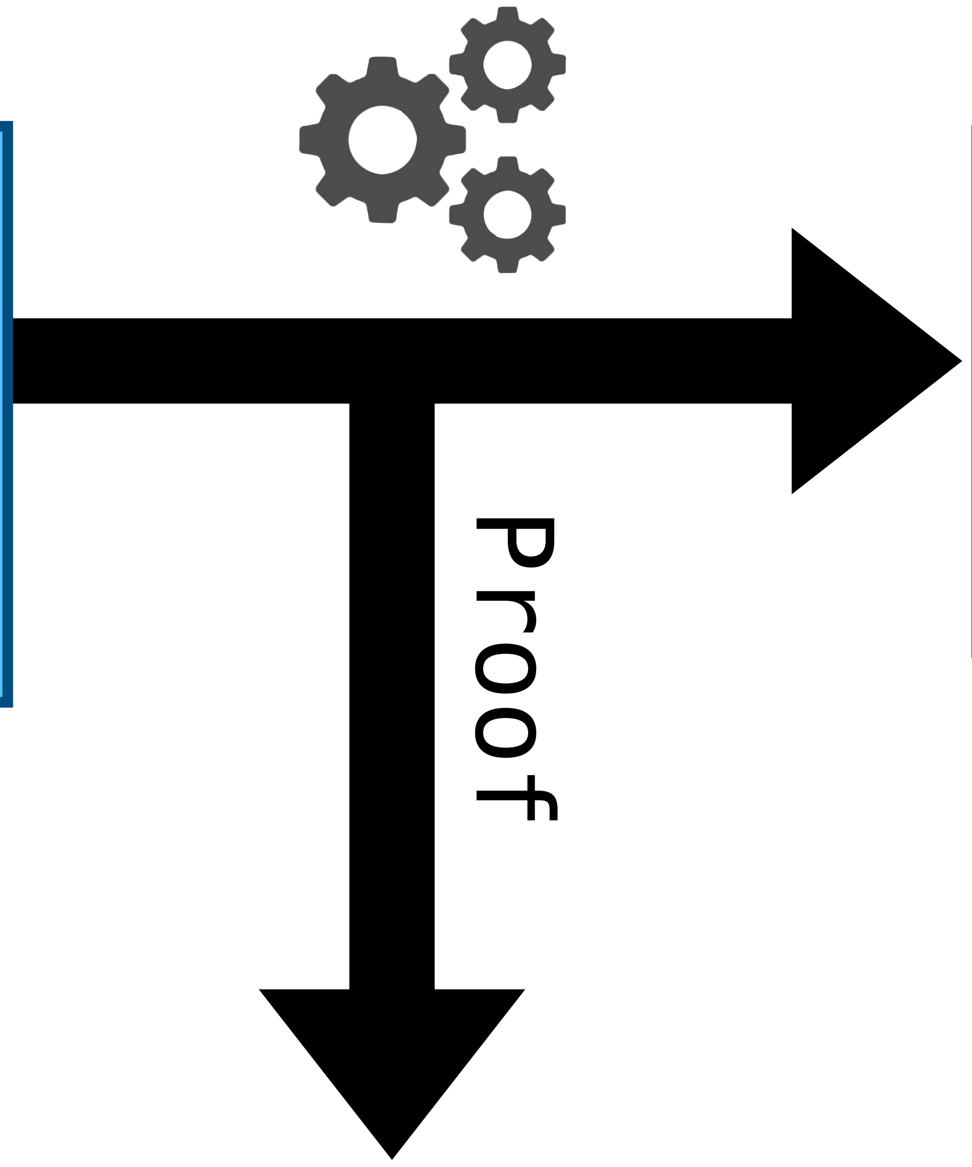
Graph-Based Compiler

```
comp_g :: Expr → Code_g → Code_g
comp_g (Val n)      c = ???
comp_g (Add x y)    c = ???
comp_g (If x y z)   c = ???
```

Calculating a Graph-Based Compiler

Tree-Based Compiler

```
comp :: Expr → Code → Code
comp (Val n)      c = PUSH n c
comp (Add x y)    c = ...
comp (If x y z) c = ...
```



Graph-Based Compiler

```
comp_g :: Expr → Code_g → Code_g
comp_g (Val n)      c = ???
comp_g (Add x y)    c = ???
comp_g (If x y z) c = ???
```

Compiler Spec

```
comp e ⟪c⟫ = ⟪comp_g e c⟫
```

Calculating a Graph-Based Compiler

Tree-Based Compiler

```
comp :: Expr → Code → Code
comp (Val n)     c = PUSH n c
comp (Add x y)   c = ...
comp (If x y z)  c = ...
```

Graph-Based Compiler

```
comp_g :: Expr → Code_g → Code_g
comp_g (Val n)    c = ???
comp_g (Add x y)  c = ???
comp_g (If x y z) c = ???
```

Compiler Spec

```
comp e ⦅c⦆ = ⦅comp_g e c⦆
```

Calculating a Graph-Based Compiler

Tree-Based Compiler

```
comp :: Expr → Code → Code
comp (Val n)      c = PUSH n c
comp (Add x y)    c = ...
comp (If x y z) c = ...
```

Graph-Based Compiler

```
comp_g :: Expr → Code_g → Code_g
comp_g (Val n)     c = ???
comp_g (Add x y)   c = ???
comp_g (If x y z) c = ???
```

Compiler Spec

```
comp e ⟦c⟧ = ⟦comp_g e c⟧
```

Prove the spec before we have $\texttt{comp}_g$.

Calculating a Graph-Based Compiler

Tree-Based Compiler

```
comp :: Expr → Code → Code
comp (Val n)      c = PUSH n c
comp (Add x y)    c = ...
comp (If x y z)   c = ...
```

Graph-Based Compiler

```
comp_g :: Expr → Code_g → Code_g
comp_g (Val n)    c = ???
comp_g (Add x y)  c = ???
comp_g (If x y z) c = ???
```

Compiler Spec

```
comp e ⟦c⟧ = ⟦comp_g e c⟧
```

Prove the spec before we have comp_g .

Definition of comp_g falls out of the <u>equational reasoning</u>.

How to Represent Graphs

How to Represent Graphs
to Support Equational Reasoning

Representing Graph-Based Code

Tree-Based Code

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Representing Graph-Based Code

Tree-Based Code

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Graph-Based Code

```
data Code_g l = HALT_g
              | PUSH_g Int (Code_g l)
              | ADD_g (Code_g l)
              | JPZ_g l (Code_g l)
```

Representing Graph-Based Code

Tree-Based Code

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Graph-Based Code

```
data Code_g l = HALT_g
              | PUSH_g Int (Code_g l)
              | ADD_g (Code_g l)
              | JPZ_g l (Code_g l)
```

Parametric Higher-Order Abstract Syntax

Representing Graph-Based Code

Tree-Based Code

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Parametrised by an abstract type of labels

Graph-Based Code

```
data Code_g l = HALT_g
              | PUSH_g Int (Code_g l)
              | ADD_g (Code_g l)
              | JPZ_g l (Code_g l)
```

Parametric Higher-Order Abstract Syntax

Representing Graph-Based Code

Tree-Based Code

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Parametrised by an abstract type of labels

Graph-Based Code

```
data Code_g l = HALT_g
              | PUSH_g Int (Code_g l)
              | ADD_g (Code_g l)
              | JPZ_g l (Code_g l)
```

Parametric Higher-Order Abstract Syntax

Representing Graph-Based Code

Tree-Based Code

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Parametrised by an abstract type of labels

Graph-Based Code

```
data Code_g l = HALT_g
              | PUSH_g Int (Code_g l)
              | ADD_g (Code_g l)
              | JPZ_g l (Code_g l)
              | JMP_g l
              | LAB_g (l → Code_g l) (Code_g l)
```

Parametric Higher-Order Abstract Syntax

Representing Graph-Based Code

Tree-Based Code

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Parametrised by an abstract type of labels

Jump to a label

Graph-Based Code

```
data Code_g l = HALT_g
              | PUSH_g Int (Code_g l)
              | ADD_g (Code_g l)
              | JPZ_g l (Code_g l)
              | JMP_g l
              | LAB_g (l → Code_g l) (Code_g l)
```

Parametric Higher-Order Abstract Syntax

Representing Graph-Based Code

Tree-Based Code

```
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | JPZ Code Code
```

Parametrised by an abstract type of labels

Graph-Based Code

```
data Code_g l = HALT_g
              | PUSH_g Int (Code_g l)
              | ADD_g (Code_g l)
              | JPZ_g l (Code_g l)
              | JMP_g l
              | LAB_g (l → Code_g l) (Code_g l)
```

Jump to a label

Label a piece of code with a fresh label

Calculating a Graph-Based Compiler

Calculating a Graph-Based Compiler

We will prove the spec by induction on e

```
comp e ⟪c⟫ = ⟪comp_g e c⟫
```

Calculating a Graph-Based Compiler

`e = Add x y:`

We will prove the spec by induction on e

```
comp e ⟪c⟫ = ⟪comp_g e c⟫
```

Calculating a Graph-Based Compiler

e = Add x y:

```
  comp (Add x y) ⟪c⟫
=
  ...
=
  ...
=
  ...
=
  ⟪ c' ⟫              for some c'
```

We will prove the spec by induction on e

$$\texttt{comp e } \langle\!\langle c \rangle\!\rangle = \langle\!\langle \texttt{comp}_g \texttt{ e c} \rangle\!\rangle$$

Calculating a Graph-Based Compiler

e = Add x y:

```
comp (Add x y) ⟪c⟫
=
  ...
=
  ...
=
  ...
=
⟪ c' ⟫
```

for some c'

We will prove the spec by induction on e

comp e ⟪c⟫ = ⟪comp_g e c⟫

Calculating a Graph-Based Compiler

<u>`e = Add x y:`</u>

```
    comp (Add x y) ⟪c⟫
=
    ...
=
    ...
=
    ...
=
    ⟪ c' ⟫
```

for some `c'`

We will prove the spec by <u>induction on e</u>

`comp e ⟪c⟫ = ⟪comp`$_g$` e c⟫`

Calculating a Graph-Based Compiler

e = Add x y:

```
  comp (Add x y) ⦅c⦆
=
  ...
=
  ...
=
  ...
=
  ⦅ c' ⦆
```

for some c'

We will prove the spec by induction on e

comp e ⦅c⦆ = ⦅comp_g e c⦆

Calculating a Graph-Based Compiler

```
e = Add x y:

  comp (Add x y) ⦇c⦈
=
  ...
=
  ...
=
  ...
=
  ⦇ c' ⦈
```

for some c'

We will prove the spec by induction on e

```
comp e ⦇c⦈ = ⦇compg e c⦈
```

Once we have completed the calculation, we can conclude

```
compg (Add x y) c = c'
```

Calculating a Graph-Based Compiler

<u>`e = Add x y:`</u>

```
    comp (Add x y) ⦅c⦆
```

$\texttt{comp e ⦅c⦆ = ⦅comp}_g\texttt{ e c⦆}$

Calculating a Graph-Based Compiler

<u>e = Add x y:</u>

```
  comp (Add x y) ⦅c⦆
= { definition of comp }
  comp x (comp y (ADD ⦅c⦆))
```

```
comp e ⦅c⦆ = ⦅comp_g e c⦆
```

Calculating a Graph-Based Compiler

<u>e = Add x y:</u>

```
   comp (Add x y) ⦅c⦆
 = { definition of comp }
   comp x (comp y (ADD ⦅c⦆))
 = { definition of ⦅·⦆ }
   comp x (comp y ⦅ADDg c⦆)
```

$\texttt{comp e } ⦅\texttt{c}⦆ = ⦅\texttt{comp}_g \texttt{ e c}⦆$

$⦅\texttt{ADD}_g \texttt{ c}⦆ = \texttt{ADD } ⦅\texttt{c}⦆$

Calculating a Graph-Based Compiler

e = Add x y:

```
 comp (Add x y) ⦇c⦈
= { definition of comp }
 comp x (comp y (ADD ⦇c⦈))
= { definition of ⦇·⦈ }
 comp x (comp y ⦇ADDg c⦈)
= { induction hypothesis for y }
 comp x ⦇compg y (ADDg c)⦈
```

```
comp e ⦇c⦈ = ⦇compg e c⦈
```

```
⦇ADDg c⦈ = ADD ⦇c⦈
```

Calculating a Graph-Based Compiler

e = Add x y:

```
 comp (Add x y) ⟪c⟫
= { definition of comp }
 comp x (comp y (ADD ⟪c⟫))
= { definition of ⟪·⟫ }
 comp x (comp y ⟪ADD_g c⟫)
= { induction hypothesis for y }
 comp x ⟪comp_g y (ADD_g c)⟫
= { induction hypothesis for x }
 ⟪comp_g x (comp_g y (ADD_g c))⟫
```

comp e ⟪c⟫ = ⟪comp_g e c⟫

⟪ADD_g c⟫ = ADD ⟪c⟫

Calculating a Graph-Based Compiler

e = Add x y:

```
  comp (Add x y) ⟪c⟫
= { definition of comp }
  comp x (comp y (ADD ⟪c⟫))
= { definition of ⟪·⟫ }
  comp x (comp y ⟪ADDg c⟫)
= { induction hypothesis for y }
  comp x ⟪compg y (ADDg c)⟫
= { induction hypothesis for x }
  ⟪compg x (compg y (ADDg c))⟫
```

```
comp e ⟪c⟫ = ⟪compg e c⟫
```

```
⟪ADDg c⟫ = ADD ⟪c⟫
```

Hence we can conclude that

```
compg (Add x y) c
    = compg x (compg y (ADDg c))
```

Calculating a Graph-Based Compiler

This calculation is entirely mechanical & syntax-directed!

<u>e = Add x y:</u>

```
  comp (Add x y) ⦅c⦆
= { definition of comp }
  comp x (comp y (ADD ⦅c⦆))
= { definition of ⦅·⦆ }
  comp x (comp y ⦅ADD_g c⦆)
= { induction hypothesis for y }
  comp x ⦅comp_g y (ADD_g c)⦆
= { induction hypothesis for x }
  ⦅comp_g x (comp_g y (ADD_g c))⦆
```

```
comp e ⦅c⦆ = ⦅comp_g e c⦆
```

```
⦅ADD_g c⦆ = ADD ⦅c⦆
```

Hence we can conclude that

```
comp_g (Add x y) c
    = comp_g x (comp_g y (ADD_g c))
```

Calculating a Graph-Based Compiler

```
e = If x y z:
```

e = If x y z:

e = If x y z:

```
   comp (If x y z) ⦅c⦆
 = { definition of comp }
   comp x (JPZ (comp z ⦅c⦆) (comp y ⦅c⦆))
 = { abstract over ⦅c⦆ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⦅c⦆)
 = { definition of ⦅·⦆ }
   comp x ((λ l → JPZ (comp z ⦅JMPg l⦆) (comp y ⦅JMPg l⦆)) ⦅c⦆)
 = { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⦅compg z (JMPg l)⦆ ⦅compg y (JMPg l)⦆) ⦅c⦆)
 = { abstract over ⦅compG z (JMPg l)⦆ }
   comp x ((λ l → (λ l' → JPZ l' ⦅compg y (JMPg l)⦆)⦅compg z (JMPg l)⦆)⦅c⦆)
 = { definition of ⦅·⦆ }
   comp x ((λ l → (λ l' → ⦅JPZg l' (compg y (JMPg l))⦆)⦅compg z (JMPg l)⦆)⦅c⦆)
 = { definition of ⦅·⦆ }
   comp x ((λ l → ⦅LABg (λl' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l))⦆) ⦅c⦆)
 = { definition of ⦅·⦆ }
   comp x ⦅LABg (λ l → LABg (λ l' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l))) c)⦆
 = { induction hypothesis for x }
   ⦅compg x (LABg (λ l → LABg (λ l' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l))) c))⦆
```

e = If x y z:

Mechanical, syntax-directed steps

```
   comp (If x y z) ⟪c⟫
= { definition of comp }
   comp x (JPZ (comp z ⟪c⟫) (comp y ⟪c⟫))
= { abstract over ⟪c⟫ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⟪c⟫)
= { definition of ⟪·⟫ }
   comp x ((λ l → JPZ (comp z ⟪JMP_g l⟫) (comp y ⟪JMP_g l⟫)) ⟪c⟫)
= { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⟪comp_g z (JMP_g l)⟫ ⟪comp_g y (JMP_g l)⟫) ⟪c⟫)
= { abstract over ⟪compG z (JMPg l)⟫ }
   comp x ((λ l → (λ l' → JPZ l' ⟪comp_g y (JMP_g l)⟫)⟪comp_g z (JMP_g l)⟫)⟪c⟫)
= { definition of ⟪·⟫ }
   comp x ((λ l → (λ l' → ⟪JPZ_g l' (comp_g y (JMP_g l))⟫)⟪comp_g z (JMP_g l)⟫)⟪c⟫)
= { definition of ⟪·⟫ }
   comp x ((λ l → ⟪LAB_g (λl' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))⟫) ⟪c⟫)
= { definition of ⟪·⟫ }
   comp x ⟪LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c)⟫
= { induction hypothesis for x }
   ⟪comp_g x (LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c))⟫
```

e = If x y z:

Mechanical, syntax-directed steps

Goal: Avoid code duplication

```
   comp (If x y z) ⟪c⟫
=  { definition of comp }
   comp x (JPZ (comp z ⟪c⟫) (comp y ⟪c⟫))
=  { abstract over ⟪c⟫ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⟪c⟫)
=  { definition of ⟪·⟫ }
   comp x ((λ l → JPZ (comp z ⟪JMPg l⟫) (comp y ⟪JMPg l⟫)) ⟪c⟫)
=  { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⟪compg z (JMPg l)⟫ ⟪compg y (JMPg l)⟫) ⟪c⟫)
=  { abstract over ⟪compG z (JMPg l)⟫ }
   comp x ((λ l → (λ l' → JPZ l' ⟪compg y (JMPg l)⟫)⟪compg z (JMPg l)⟫)⟪c⟫)
=  { definition of ⟪·⟫ }
   comp x ((λ l → (λ l' → ⟪JPZg l' (compg y (JMPg l))⟫)⟪compg z (JMPg l)⟫)⟪c⟫)
=  { definition of ⟪·⟫ }
   comp x ((λ l → ⟪LABg (λl' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l))⟫) ⟪c⟫)
=  { definition of ⟪·⟫ }
   comp x ⟪LABg (λ l → LABg (λ l' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l))) c)⟫
=  { induction hypothesis for x }
   ⟪compg x (LABg (λ l → LABg (λ l' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l))) c))⟫
```

Mechanical, syntax-directed steps

Goal: Avoid code duplication

e = If x y z:

```
   comp (If x y z) ⦅c⦆
= { definition of comp }
   comp x (JPZ (comp z ⦅c⦆) (comp y ⦅c⦆))
= { abstract over ⦅c⦆ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ((λ l → JPZ (comp z ⦅JMPg l⦆) (comp y ⦅JMPg l⦆)) ⦅c⦆)
= { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⦅compg z (JMPg l)⦆ ⦅compg y (JMPg l)⦆) ⦅c⦆)
= { abstract over ⦅compG z (JMPg l)⦆ }
   comp x ((λ l → (λ l' → JPZ l' ⦅compg y (JMPg l)⦆)⦅compg z (JMPg l)⦆)⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ((λ l → (λ l' → ⦅JPZg l' (compg y (JMPg l))⦆)⦅compg z (JMPg l)⦆)⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ((λ l → ⦅LABg (λl' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l))⦆) ⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ⦅LABg (λ l → LABg (λ l' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l))) c)⦆
= { induction hypothesis for x }
   ⦅compg x (LABg (λ l → LABg (λ l' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l))) c)⦆
```

e = If x y z:

Mechanical, syntax-directed steps

Goal: Avoid code duplication

```
   comp (If x y z) ⟦c⟧
= { definition of comp }
   comp x (JPZ (comp z ⟦c⟧) (comp y ⟦c⟧))
= { abstract over ⟦c⟧ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⟦c⟧)
= { definition of ⟦·⟧ }
   comp x ((λ l → JPZ (comp z ⟦JMP_g l⟧) (comp y ⟦JMP_g l⟧)) ⟦c⟧)
= { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⟦comp_g z (JMP_g l)⟧ ⟦comp_g y (JMP_g l)⟧) ⟦c⟧)
= { abstract over ⟦compG z (JMPg l)⟧ }
   comp x ((λ l → (λ l' → JPZ l' ⟦comp_g y (JMP_g l)⟧)⟦comp_g z (JMP_g l)⟧)⟦c⟧)
= { definition of ⟦·⟧ }
   comp x ((λ l → (λ l' → ⟦JPZ_g l' (comp_g y (JMP_g l))⟧)⟦comp_g z (JMP_g l)⟧)⟦c⟧)
= { definition of ⟦·⟧ }
   comp x ((λ l → ⟦LAB_g (λl' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))⟧) ⟦c⟧)
= { definition of ⟦·⟧ }
   comp x ⟦LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c)⟧
= { induction hypothesis for x }
   ⟦comp_g x (LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c)⟧
```

e = If x y z:

Mechanical, syntax-directed steps

Goal: Avoid code duplication

Goal: First argument of JPZ_g must have type l

```
   comp (If x y z) ⦅c⦆
 = { definition of comp }
   comp x (JPZ (comp z ⦅c⦆) (comp y ⦅c⦆))
 = { abstract over ⦅c⦆ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⦅c⦆)
 = { definition of ⦅·⦆ }
   comp x ((λ l → JPZ (comp z ⦅JMP_g l⦆) (comp y ⦅JMP_g l⦆)) ⦅c⦆)
 = { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⦅comp_g z (JMP_g l)⦆ ⦅comp_g y (JMP_g l)⦆) ⦅c⦆)
 = { abstract over ⦅compG z (JMPg l)⦆ }
   comp x ((λ l → (λ l' → JPZ l' ⦅comp_g y (JMP_g l)⦆)⦅comp_g z (JMP_g l)⦆)⦅c⦆)
 = { definition of ⦅·⦆ }
   comp x ((λ l → (λ l' → ⦅JPZ_g l' (comp_g y (JMP_g l))⦆)⦅comp_g z (JMP_g l)⦆)⦅c⦆)
 = { definition of ⦅·⦆ }
   comp x ((λ l → ⦅LAB_g (λl' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))⦆) ⦅c⦆)
 = { definition of ⦅·⦆ }
   comp x ⦅LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c)⦆
 = { induction hypothesis for x }
   ⦅comp_g x (LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c)⦆
```

e = If x y z:

Mechanical, syntax-directed steps

Goal: Avoid code duplication

Goal: First argument of JPZ_g must have type l

```
   comp (If x y z) ⦇c⦈
= { definition of comp }
   comp x (JPZ (comp z ⦇c⦈) (comp y ⦇c⦈))
= { abstract over ⦇c⦈ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ((λ l → JPZ (comp z ⦇JMP_g l⦈) (comp y ⦇JMP_g l⦈)) ⦇c⦈)
= { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⦇comp_g z (JMP_g l)⦈ ⦇comp_g y (JMP_g l)⦈) ⦇c⦈)
= { abstract over ⦇compG z (JMPg l)⦈ }
   comp x ((λ l → (λ l' → JPZ l' ⦇comp_g y (JMP_g l)⦈)⦇comp_g z (JMP_g l)⦈)⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ((λ l → (λ l' → ⦇JPZ_g l' (comp_g y (JMP_g l))⦈)⦇comp_g z (JMP_g l)⦈)⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ((λ l → ⦇LAB_g (λl' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))⦈) ⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ⦇LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c)⦈
= { induction hypothesis for x }
   ⦇comp_g x (LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c)⦈
```

e = If x y z:

Mechanical, syntax-directed steps

Goal: Avoid code duplication

Goal: First argument of JPZ_g must have type l

```
   comp (If x y z) ⦇c⦈
= { definition of comp }
   comp x (JPZ (comp z ⦇c⦈) (comp y ⦇c⦈))
= { abstract over ⦇c⦈ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ((λ l → JPZ (comp z ⦇JMP_g l⦈) (comp y ⦇JMP_g l⦈)) ⦇c⦈)
= { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⦇comp_g z (JMP_g l)⦈ ⦇comp_g y (JMP_g l)⦈) ⦇c⦈)
= { abstract over ⦇compG z (JMPg l)⦈ }
   comp x ((λ l → (λ l' → JPZ l' ⦇comp_g y (JMP_g l)⦈)⦇comp_g z (JMP_g l)⦈)⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ((λ l → (λ l' → ⦇JPZ_g l' (comp_g y (JMP_g l))⦈)⦇comp_g z (JMP_g l)⦈)⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ((λ l → ⦇LAB_g (λl' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))⦈) ⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ⦇LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c)⦈
= { induction hypothesis for x }
   ⦇comp_g x (LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c)⦈
```

e = If x y z:

```
   comp (If x y z) ⦅c⦆
= { definition of comp }
   comp x (JPZ (comp z ⦅c⦆) (comp y ⦅c⦆))
= { abstract over ⦅c⦆ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ((λ l → JPZ (comp z ⦅JMP_g l⦆) (comp y ⦅JMP_g l⦆)) ⦅c⦆)
= { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⦅comp_g z (JMP_g l)⦆ ⦅comp_g y (JMP_g l)⦆) ⦅c⦆)
= { abstract over ⦅compG z (JMPg l)⦆ }
   comp x ((λ l → (λ l' → JPZ l' ⦅comp_g y (JMP_g l)⦆) ⦅comp_g z (JMP_g l)⦆) ⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ((λ l → (λ l' → ⦅JPZ_g l' (comp_g y (JMP_g l))⦆) ⦅comp_g z (JMP_g l)⦆) ⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ((λ l → ⦅LAB_g (λl' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))⦆) ⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ⦅LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c)⦆
= { induction hypothesis for x }
   ⦅comp_g x (LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))) c))⦆
```

Mechanical, syntax-directed steps

Goal: Avoid code duplication

Goal: First argument of JPZ_g must have type l

e = If x y z:

Mechanical, syntax-directed steps

Goal: Avoid code duplication

Both steps are just β-expansion

Goal: First argument of JPZ_g must have type l

```
   comp (If x y z) ⦇c⦈
= { definition of comp }
   comp x (JPZ (comp z ⦇c⦈) (comp y ⦇c⦈))
= { abstract over ⦇c⦈ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ((λ l → JPZ (comp z ⦇JMPg l⦈) (comp y ⦇JMPg l⦈)) ⦇c⦈)
= { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⦇compg z (JMPg l)⦈ ⦇compg y (JMPg l)⦈) ⦇c⦈)
= { abstract over ⦇compG z (JMPg l)⦈ }
   comp x ((λ l → (λ l' → JPZ l' ⦇compg y (JMPg l)⦈) ⦇compg z (JMPg l)⦈) ⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ((λ l → (λ l' → ⦇JPZg l' (compg y (JMPg l))⦈)⦇compg z (JMPg l)⦈)⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ((λ l → ⦇LABg (λl' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l))⦈) ⦇c⦈)
= { definition of ⦇·⦈ }
   comp x ⦇LABg (λ l → LABg (λ l' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l)) c)⦈
= { induction hypothesis for x }
   ⦇compg x (LABg (λ l → LABg (λ l' → JPZg l' (compg y (JMPg l))) (compg z (JMPg l)) c))⦈
```

e = If x y z:

```
   comp (If x y z) ⦅c⦆
= { definition of comp }
   comp x (JPZ (comp z ⦅c⦆) (comp y ⦅c⦆))
= { abstract over ⦅c⦆ }
   comp x ((λ l → JPZ (comp z l) (comp y l)) ⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ((λ l → JPZ (comp z ⦅JMP_g l⦆) (comp y ⦅JMP_g l⦆)) ⦅c⦆)
= { induction hypothesis for y and z }
   comp x ((λ l → JPZ ⦅comp_g z (JMP_g l)⦆ ⦅comp_g y (JMP_g l)⦆) ⦅c⦆)
= { abstract over ⦅compG z (JMPg l)⦆ }
   comp x ((λ l → (λ l' → JPZ l' ⦅comp_g y (JMP_g l)⦆) ⦅comp_g z (JMP_g l)⦆) ⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ((λ l → (λ l' → ⦅JPZ_g l' (comp_g y (JMP_g l))⦆) ⦅comp_g z (JMP_g l)⦆) ⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ((λ l → ⦅LAB_g (λl' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l))⦆) ⦅c⦆)
= { definition of ⦅·⦆ }
   comp x ⦅LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l)) c)⦆
= { induction hypothesis for x }
   ⦅comp_g x (LAB_g (λ l → LAB_g (λ l' → JPZ_g l' (comp_g y (JMP_g l))) (comp_g z (JMP_g l)) c)⦆
```

Mechanical, syntax-directed steps

Goal: Avoid code duplication

Both steps are just β-expansion

Goal: First argument of JPZ_g must have type l

We can read off the definition of $comp_g$

Final Graph-Based Compiler

```
compg :: Expr → Codeg l → Codeg l
compg (Val n)     c = PUSHg n c
compg (Add x y)   c = compg x (compg y (ADDg c))
compg (If x y z) c = compg x (LABg (λ l → LABg (λ l' →
           JPZg l' (compg y (JMPg l))) (compg z (JMPg l))) c)
```

Final Graph-Based Compiler

```
compg :: Expr → Codeg l → Codeg l
compg (Val n)     c = PUSHg n c
compg (Add x y)   c = compg x (compg y (ADDg c))
compg (If x y z) c = compg x (LABg (λ l → LABg (λ l' →
           JPZg l' (compg y (JMPg l))) (compg z (JMPg l))) c)
```

Final Graph-Based Compiler

```
compg :: Expr → Codeg l → Codeg l
compg (Val n)     c = PUSHg n c
compg (Add x y)   c = compg x (compg y (ADDg c))
compg (If x y z) c = compg x (LABg (λ l → LABg (λ l' →
           JPZg l' (compg y (JMPg l))) (compg z (JMPg l))) c)
```

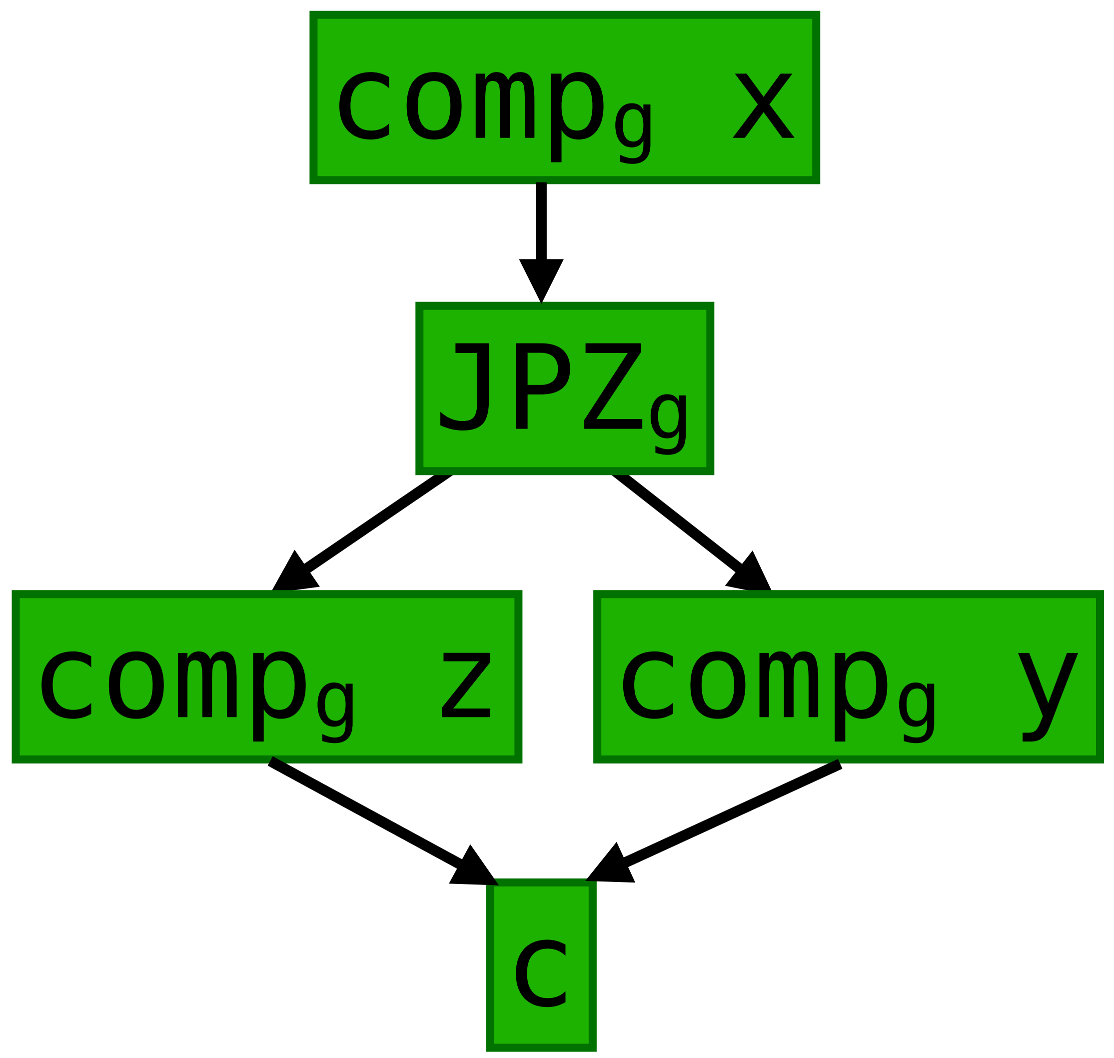
```
compg (If x y z) c =  compg x
                      JPZg l'
                      compg y
                      JMPg l
                 l': compg z
                      JMPg l
                 l : c
```

Final Graph-Based Compiler

```
compg :: Expr → Codeg l → Codeg l
compg (Val n)     c = PUSHg n c
compg (Add x y)   c = compg x (compg y (ADDg c))
compg (If x y z) c = compg x (LABg (λ l → LABg (λ l' →
           JPZg l' (compg y (JMPg l))) (compg z (JMPg l))) c)
```

```
compg (If x y z) c =  compg x
                      JPZg l'
                      compg y
                      JMPg l
                  l': compg z
                      JMPg l
                  l : c
```

≅



More in the Paper

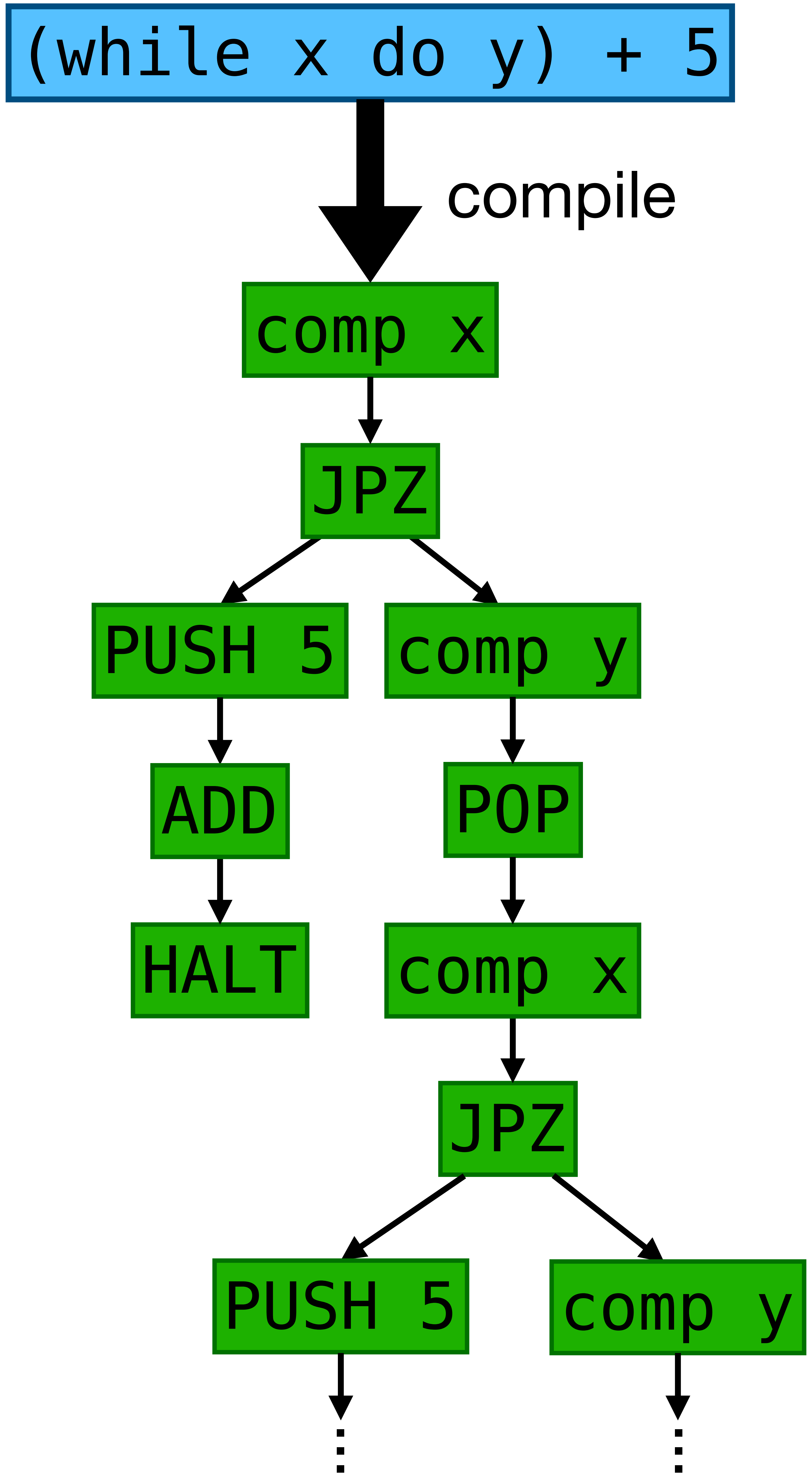
Calculation technique for compilers that produce **cyclic code** (e.g. when compiling loops)

- First calculate tree-based compiler that produces **infinite code**
- Then calculate graph-based compiler using $(\!(\cdot)\!)$
- This requires **coinductive** reasoning & **monadic** semantics

More in the Paper

Calculation technique for compilers that produce **cyclic code** (e.g. when compiling loops)

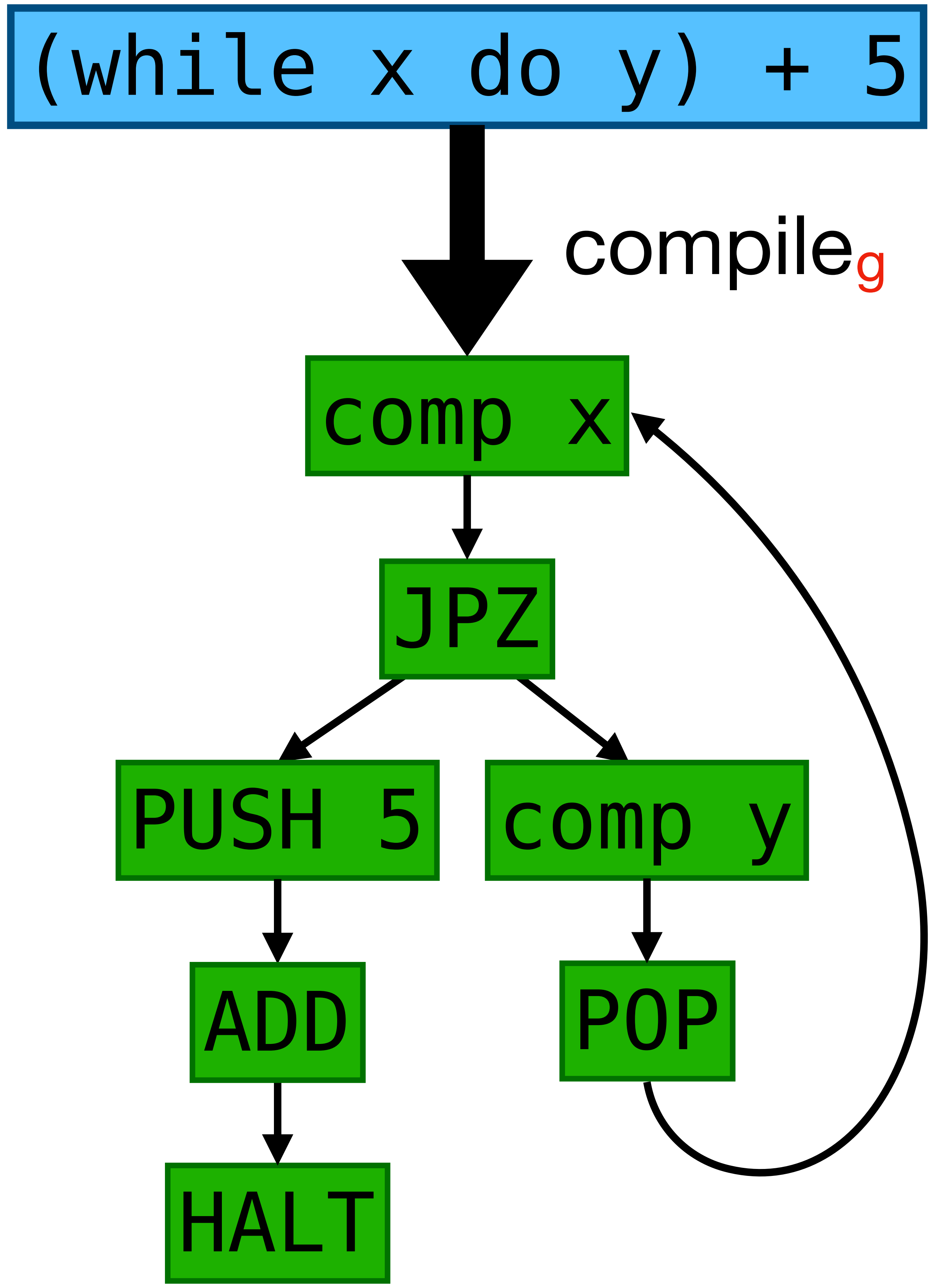
- First calculate tree-based compiler that produces **infinite code**
- Then calculate graph-based compiler using ((·))
- This requires **coinductive** reasoning & **monadic** semantics



More in the Paper

Calculation technique for compilers that produce **cyclic code** (e.g. when compiling loops)

- First calculate tree-based compiler that produces **infinite code**
- Then calculate graph-based compiler using $(\!(\cdot)\!)$
- This requires **coinductive** reasoning & **monadic** semantics



More in the Agda Formalisation

All calculations are formalised in Agda.

- **Compiler calculations for**
  - While loops
  - Lambda calculus
  - Expression language with exceptions
- Calculate the **virtual machine** for the graph-based code

specification: $\mathrm{exec}_g\ c\ s = \mathrm{exec}\ (\!(c)\!)\ s$

Beyond Trees:

Calculating Graph-Based Compilers

FUNCTIONAL PEARL

Patrick Bahr
IT University of Copenhagen, Denmark

Graham Hutton
University of Nottingham, UK