1 The Factor Compiler

If we could sort programming languages by the fuzzy notions we tend to have about how "high-level" they are, toward the high end we'd find dynamically-typed languages like Python, Ruby, and PHP—all of which are generally more interpreted than compiled. Despite being as high-level as these popular languages, Factor's implementation is driven by performance. Factor source is always compiled to native machine code using either its simple, non-optimizing compiler or (more typically) the optimizing compiler that performs several sorts of data and control flow analyses. In this section, we look at the general architecture of Factor's implementation, after which we place a particular emphasis on the transformations performed by the optimizing compiler.

Though there are projects for this

1.1 Low-level Optimizations

The low-level intermediate representation (IR) in `compiler.cfg` takes the more conventional form of a control flow graph (CFG). A CFG (not to be confused with "context-free grammar") is an arrangement of instructions into *basic blocks*: maximal sequences of "straight-line" code, where control does not transfer out of or into the middle of the block. Directed edges are added between blocks to represent control flow—either from a branching instruction to its target, or from the end of a basic block to the start of the next one. Construction of the low-level IR proceeds by analyzing the control flow of the high-level IR and converting the nodes of **??** into lower-level, more conventional instructions modeled after typical assembly code. There are over a hundred of these instructions, but many are simply different versions of the same operation. For instance, while instructions are generally called on *virtual registers* (represented in Factor simply by integers), there are *immediate* versions of instructions. The `##add` instruction, as an example, represents the sum of the contents of two registers, but `##add-imm` sums the contents of one register and an integer literal. Other instructions are inserted to make stack reads and writes explicit, as well as to balance the height. Below is a categorized list of all the instruction objects (each one is a subclass of the `insn` tuple).

cite

Is the complete list really necessary?

- Loading constants: `##load-integer`, `##load-reference`
- Optimized loading of constants, inserted by representation selection: `##load-tagged`, `##load-float`, `##load-double`, `##load-vector`
- Stack operations: `##peek`, `##replace`, `##replace-imm`, `##inc-d`, `##inc-r`
- Subroutine calls: `##call`, `##jump`, `##prologue`, `##epilogue`, `##return`
- Inhibiting tail-call optimization (TCO): `##no-tco`
- Jump tables: `##dispatch`
- Slot access: `##slot`, `##slot-imm`, `##set-slot`, `##set-slot-imm`
- Register transfers: `##copy`, `##tagged>integer`

- Integer arithmetic: `##add`, `##add-imm`, `##sub`, `##sub-imm`, `##mul`, `##mul-imm`, `##and`, `##and-imm`, `##or`, `##or-imm`, `##xor`, `##xor-imm`, `##shl`, `##shl-imm`, `##shr`, `##shr-imm`, `##sar`, `##sar-imm`, `##min`, `##max`, `##not`, `##neg`, `##log2`, `##bit-count`
- Float arithmetic: `##add-float`, `##sub-float`, `##mul-float`, `##div-float`, `##min-float`, `##max-float`, `##sqrt`
- Single/double float conversion: `##single>double-float`, `##double>single-float`
- Float/integer conversion: `##float>integer`, `##integer>float`
- SIMD operations: `##zero-vector`, `##fill-vector`, `##gather-vector-2`, `##gather-int-vector-2`, `##gather-vector-4`, `##gather-int-vector-4`, `##select-vector`, `##shuffle-vector`, `##shuffle-vector-halves-imm`, `##shuffle-vector-imm`, `##tail>head-vector`, `##merge-vector-head`, `##merge-vector-tail`, `##float-pack-vector`, `##signed-pack-vector`, `##unsigned-pack-vector`, `##unpack-vector-head`, `##unpack-vector-tail`, `##integer>float-vector`, `##float>integer-vector`, `##compare-vector`, `##test-vector`, `##test-vector-branch`, `##add-vector`, `##saturated-add-vector`, `##add-sub-vector`, `##sub-vector`, `##saturated-sub-vector`, `##mul-vector`, `##mul-high-vector`, `##mul-horizontal-add-vector`, `##saturated-mul-vector`, `##div-vector`, `##min-vector`, `##max-vector`, `##avg-vector`, `##dot-vector`, `##sad-vector`, `##horizontal-add-vector`, `##horizontal-sub-vector`, `##horizontal-shl-vector-imm`, `##horizontal-shr-vector-imm`, `##abs-vector`, `##sqrt-vector`, `##and-vector`, `##andn-vector`, `##or-vector`, `##xor-vector`, `##not-vector`, `##shl-vector-imm`, `##shr-vector-imm`, `##shl-vector`, `##shr-vector`
- Scalar/vector conversion: `##scalar>integer`, `##integer>scalar`, `##vector>scalar`, `##scalar>vector`
- Boxing and unboxing aliens: `##box-alien`, `##box-displaced-alien`, `##unbox-any-c-ptr`, `##unbox-alien`
- Zero-extending and sign-extending integers: `##convert-integer`
- Raw memory access: `##load-memory`, `##load-memory-imm`, `##store-memory`, `##store-memory-imm`
- Memory allocation: `##allot`, `##write-barrier`, `##write-barrier-imm`, `##alien-global`, `##vm-field`, `##set-vm-field`
- The foreign function interface (FFI): `##unbox`, `##unbox-long-long`, `##local-allot`, `##box`, `##box-long-long`, `##alien-invoke`, `##alien-indirect`, `##alien-assembly`, `##callback-inputs`, `##callback-outputs`
- Control flow: `##phi`, `##branch`
- Tagged conditionals: `##compare-branch`, `##compare-imm-branch`, `##compare`, `##compare-imm`

- Integer conditionals: `##compare-integer-branch`, `##compare-integer-imm-branch`, `##test-branch`, `##test-imm-branch`, `##compare-integer`, `##compare-integer-imm`, `##test`, `##test-imm`
- Float conditionals: `##compare-float-ordered-branch`, `##compare-float-unordered-branch`, `##compare-float-ordered`, `##compare-float-unordered`
- Overflowing arithmetic: `##fixnum-add`, `##fixnum-sub`, `##fixnum-mul`
- Garbage collector (GC) checks: `##save-context`, `##check-nursery-branch`, `##call-gc`
- Spills and reloads, inserted by the register allocator: `##spill`, `##reload`

```
: optimize-cfg ( cfg -- cfg' )
    optimize-tail-calls
    delete-useless-conditionals
    split-branches
    join-blocks
    normalize-height
    construct-ssa
    alias-analysis
    value-numbering
    copy-propagation
    eliminate-dead-code ;
```

Listing 1: Optimization passes on the low-level IR

By translating the high-level IR into instructions that manipulate registers directly, we reveal further redundancies that can be optimized away. The `optimize-cfg` word in Listing 1 shows the passes performed in doing this. The first word, `optimize-tail-calls`, performs tail call elimination on the CFG. *Tail calls* are those that occur within a procedure and whose results are immediately returned by that procedure. Instead of allocating a new call stack frame, we may convert tail calls into simple jumps, since afterwards the current procedure's call frame isn't really needed. In the case of recursive tail calls, we can convert special cases of recursion into loops in the CFG, so that we won't trigger call stack overflows. For instance, consider Figure 1 on the next page, which shows the effect of `optimize-tail-calls` on the following definition:

used in **??**, not defined

```
: tail-call ( -- ) tail-call ;
```

Note the recursive call (trivially) occurs at the end of the definition, just before the return point. When translated to a CFG, this is a `##call` instruction, as seen in block 4 to the left of Figure 1 on the following page. This is also just before the final `##epilogue` and `##return` instructions in block 8, as blocks 5–7 are effectively empty (these excessive `##branch`es will be eliminated in a later pass). Because of this, rather than make a whole new subroutine call, we can convert it into a `##branch` back to the beginning of the word, as in the CFG to the right.

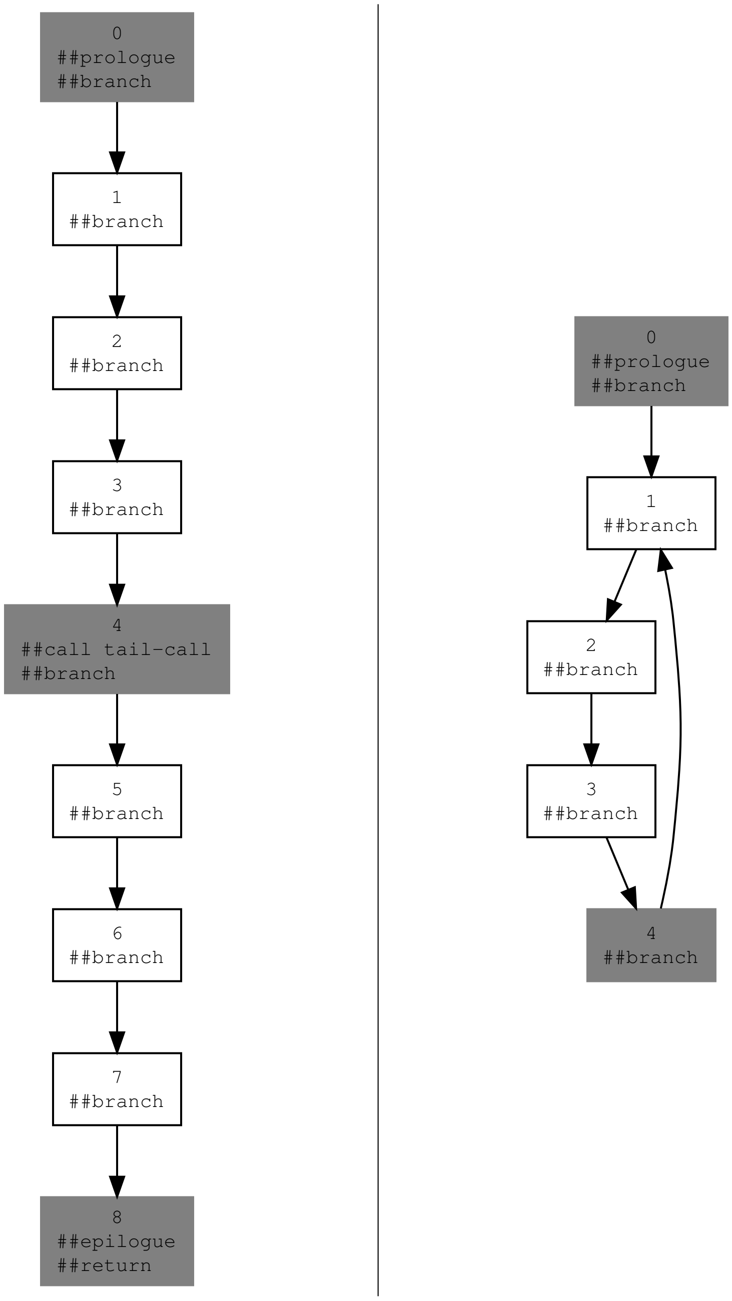


Figure 1: `tail-call` before and after `optimize-tail-calls`

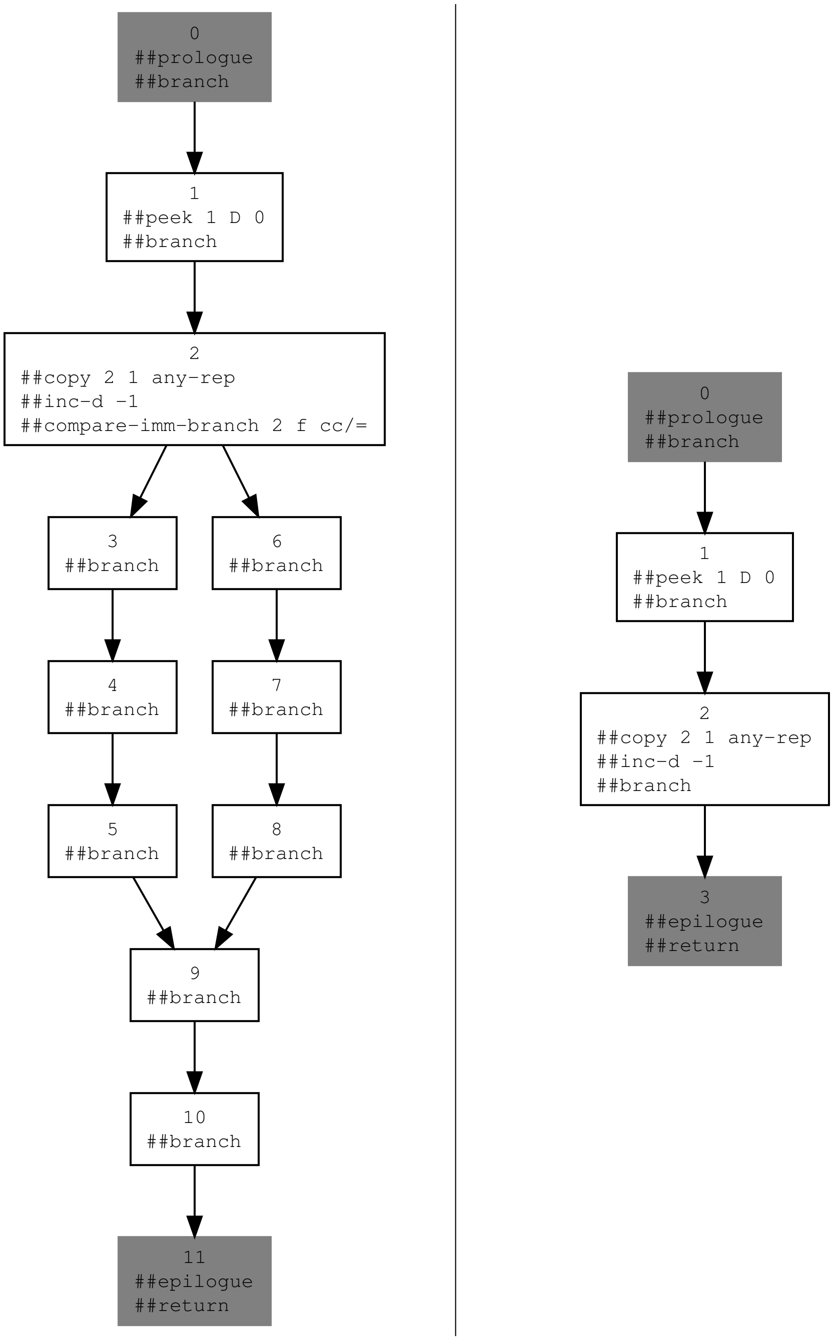
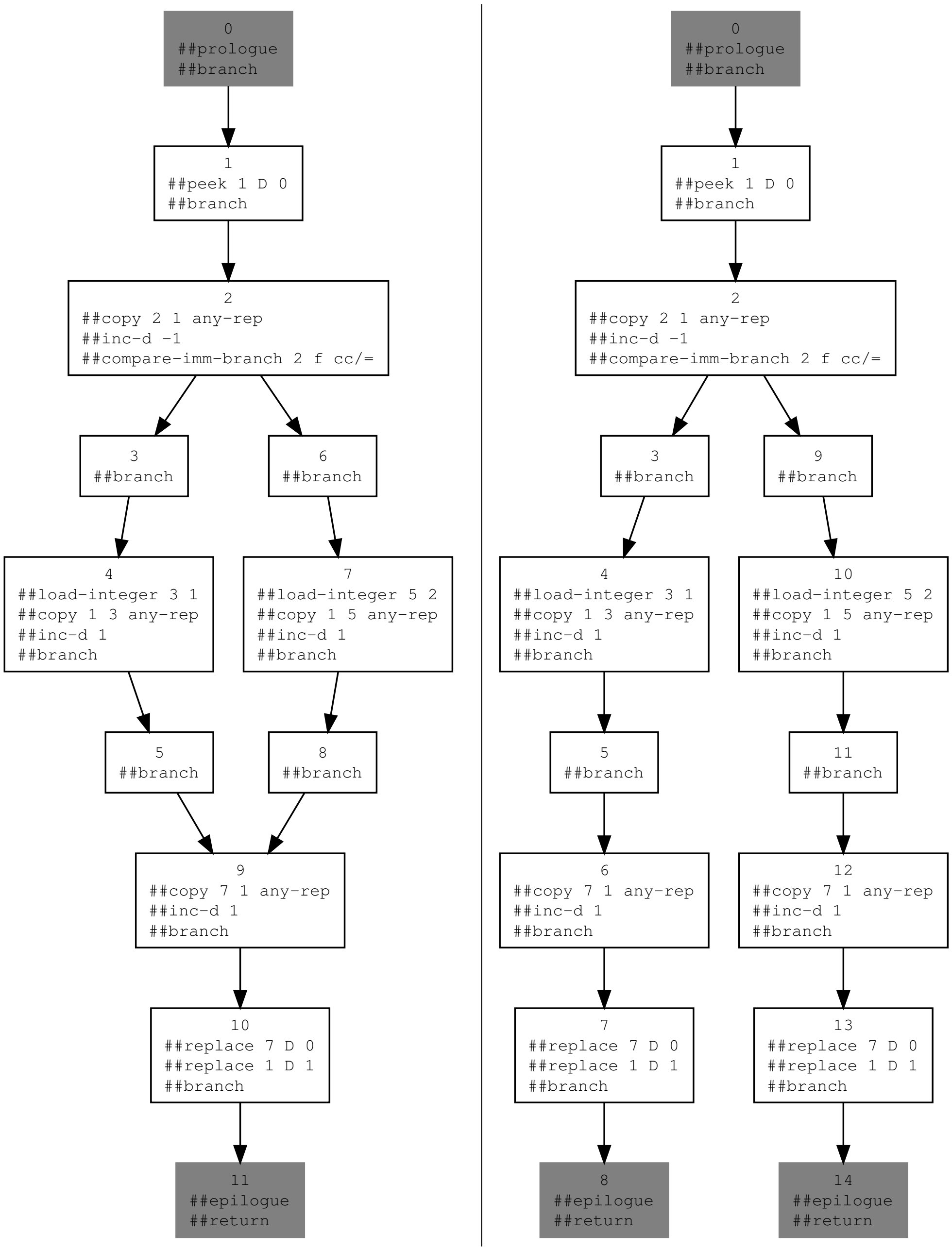


Figure 2: `[ ] [ ]` **`if`** before and after `delete-useless-conditionals`

The next pass in Listing 1 on page 3 is `delete-useless-conditionals`, which removes branches that go to the same basic block. This situation might occur as a result of optimizations performed in the high-level IR. To see it in action, Figure 2 on the preceding page shows the transformation on a purposefully useless conditional, `[ ] [ ]` **`if`**. Before removing the useless conditional, the CFG **`##peek`**s at the top of the data stack (`D 0`), storing the result in the virtual register `1`. This value is popped, so we decrement the stack height (**`##inc-d`** `-1`). Then, **`##compare-imm-branch`** in block 2 compares the value in the virtual register `2` (which is a copy of `1`, the top of the stack) to the immediate value **`f`** to see if it's not equal (signified by `cc/=`). However, both branches jump through several empty blocks and merge at the same destination. Thus, we can remove both branches and replace `##compare-imm-branch` with an unconditional **`##branch`** to the eventual destination. We see this on the right of Figure 2 on the previous page.

In order to expose more opportunities for optimization, **`split-branches`** will actually duplicate code. We use the fact that code immediately following a conditional will be executed along either branch. If it's sufficiently short, we copy it up into the branches individually. That is, we change `[ A ] [ B ]` **`if`** `C` into `[ A C ] [ B C ]` **`if`**, as long as `C` is small enough. Later analyses may then, for example, more readily eliminate one of the branches if it's never taken. Figure 3 on the following page shows what such a transformation looks like on a CFG. The example `[ 1 ] [ 2 ]` **`if dup`** is essentially changed into `[ 1` **`dup`** `] [ 2` **`dup`** `]` **`if`**, thus splitting the block with two predecessors (block 9) on the left.



```
0
##prologue
##branch

1
##peek 1 D 0
##branch

2
##copy 2 1 any-rep
##inc-d -1
##compare-imm-branch 2 f cc/=

3
##branch

6
##branch

4
##load-integer 3 1
##copy 1 3 any-rep
##inc-d 1
##branch

7
##load-integer 5 2
##copy 1 5 any-rep
##inc-d 1
##branch

5
##branch

8
##branch

9
##copy 7 1 any-rep
##inc-d 1
##branch

10
##replace 7 D 0
##replace 1 D 1
##branch

11
##epilogue
##return
```

```
0
##prologue
##branch

1
##peek 1 D 0
##branch

2
##copy 2 1 any-rep
##inc-d -1
##compare-imm-branch 2 f cc/=

3
##branch

9
##branch

4
##load-integer 3 1
##copy 1 3 any-rep
##inc-d 1
##branch

10
##load-integer 5 2
##copy 1 5 any-rep
##inc-d 1
##branch

5
##branch

11
##branch

6
##copy 7 1 any-rep
##inc-d 1
##branch

12
##copy 7 1 any-rep
##inc-d 1
##branch

7
##replace 7 D 0
##replace 1 D 1
##branch

13
##replace 7 D 0
##replace 1 D 1
##branch

8
##epilogue
##return

14
##epilogue
##return
```

Figure 3: `[ 1 ] [ 2 ]` **`if dup`** before and after `split-branches`