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`