emitting strategy

- If-not immediate ⇒ emit.
- If-block ⇒ make-new block, emit the new block.

「Emitting」

Namely we set

(+ (* 3 4) (* 7 8))

⇒ ((let a_1 = (* 7 8)) (let a_2 = (* 3 4))
(let a_3 = (+ a_1 a_2)))

a_3

This however lacks, as we should make blocks!
Lets are fine not making blocks, but fixed depth recursion should.

(loop 10 ((x 1) (y 7)) #1=(def ((x (* x 2)) (y (* 7 y))) (recur x y)))

⇒

(emit: instruction (make-loop
⋮
: body (emit: new block #1#)))

def just acts like our previous calls.

This looks awefully like ANF... I think there is no avoiding it, thus we can deprecate our ANF pass! However we should keep it in case a new form breaks our strategy!

[Looping Mechanism]

It seems we want some kind of loop-recur structure, so we can pass data around between rounds, and even iterate. Thus here is the proposed form

```
(defun array-iterate (func arr)                    2^8 2^16 2^24 ...
  (def ((ele-size (size arr)))
    (loop (length arr) ((array arr) (index 0))
      ;; optimizing index by doing formula manually at 1
      (def ((constraint (x1 x2)) ; arr = rest * 2^8 + ele-at-i!
            (= array (+ x2 (* x1 (expt 2 ele-size))))
        ;; 3,4159 => 3,415+9 = 3,415*10+9
        ;; for an example with digits notice how x2 is what
        ;; we want!
        (funcall func array index)
        (recur x1 (1+ index)))))))
```

; strategy 1 is useful to see what we can improve

We should note an important point, index is a numerical value... In fact it is never a wire. Strategy 1: In fact we should allow any CL value in this. However this gives us a challenge. We want to compile this as an evaluated block. However we can solve this with a bit of analysis. What if we made the body a lambda which is evaluated at the points. The 'recur' would tell us the next value to evaluate it at. We can even store these in a block to denote the loop.

Note; we will back off the highlight next page!

- Downsides are quite large [Looping P2]

+ we don't get a straightforward AST. Thus passes can't be run early on it.
+ we lose structure, our AST has to be unrolled to be analyzed!

Strategy - 2:

- Alternative strategies:

Instead lets say they don't have to be CL values, but instead any vamp_ir value, circuits or constants.
Then this would allow us the same code as before, but any CL-logic on the value would be moot.
thus we instead treat it as an alucard value.
The one drawback of this strategy is that we should be able to treat lengths at compile-time, yet we can't. Thus we would also want an if node that we can use. The other issue is just that of constant propagation. Before it was free, but now we need a pass probably along with elimination.

However a saving grace would be that if we look at the (recur ...) reference and loop value we can determine if it is or not.

- Future extensions

+ With strategy 2, we have some solid foundation.
+ This means we can push it, namely we may want to combine this with if, for a looping block that can end early or even have alternative branches based on a condition.
  * This last point will make constant propagation harder but it is still doable.

[conditional compilation]

branching in circuits is difficult in the sense that we always execute both the then and else branches. Thus expectations about cost has to be tempered by this. We can probably compile like the following.

```
(if pred then else)
=> (let ((then' then)
         (else' else)
         (pred' pred)
         (constraint (r-pred')
           (= 1 (+ pred' r-pred'))))
     (+ (* pred' then')
        (* r-pred' else')))
```

pred' → 0 → else' (1+ pred)
pred' → 1 → then' (1- pred)

$p + x = 1$

because of the lack of laziness we can make this a normal function! since circuits are limited, we actually need to implement this even as a CL function (easy) or perhaps we can implement an if block if we can optimize harder.

I think due to simplicity I'll make this a Cl function. However we should experiment with writing it in the AST for optimization ∧ maybe SMT solvers.

- optimization chance
+ if its a node then I can do lambda lifting of similar logic so we don't duplicate as much work

One addition to def that we desperetly need is to make actual constraints. We can't retrofit it with the same syntax, as analysis on figuring out which are basic binders would be hard, and constraint bindings terms quite difficult. Thus I propose:

```
(constraint (<symbol>*)
  <constraint-code>*)
```

This creates the symbols in the first list as values which will be refined later by the constraint. Since constraints are often (= ...) expressions, we should be able to treat them the same as any other equality!

Since vampir already supports this reasoning, we are solid, and can strictly increase the expression of alucard.

I would also like this to be allowed in (def ...) as well, as it would be klunky otherwise.

比如 (Ex)

```
; Prolog hackery in my alucard?
; The magic of circuits!
(constraint (inverse-bool)
  (= 1 (+ bool inverse-bool))) ; ship it!

(def ((size (length arr))
      (constraint (last-digit rest)
        (= arr (+ last-digit (* rest (ceil (lg 8)))))))
  .....)
```