==========
Iteration

logically controlled v. enumeration controlled
"while condition is true" v. "for every element of set"
In the latter case, the number of elements (and their identities) are known before we even start the loop (and in general, we don't want the values we iterate over to depend on anything we do in early iterations).

Logically-controlled loops

pre-test (`while`)
post-test (`repeat`)
one-and-a-half loops (loop with exit)
labels for non-closest exit?

implementation options:

```
    L1:
        r1 := <condition>
        if !r1 goto L2
        <loop body>
        goto L1
    L2:
```

That has two branches in every iteration.

```
        r1 := <condition>
        if !r1 goto L2
    L1:
        <loop body>
        r1 := <condition>
        if r1 goto L1
    L2:
```

That evaluates the condition in two different places.
Not a big deal if it doesn't bloat code size.
If it's complicated we can do this instead:

```
        goto L2
    L1:
        <loop body>
    test:
        r1 := <condition>
        if r1 goto L1
```

That has one extra jump, but only one copy of the test.

C-style `for` loop
semantically clean, but ***not*** really a `for` loop
hard to apply the various optimizations possible for "real" `for` loops

```
    for (int i = first; i <= last; i+= step) {
        ...
    }

    ~=

    {
        int i = first;
        while (i <= last) {
            ...
            i += step;
        }
    }
```

Enumeration-controlled loops

```
    for v in my_set /* in my favorite order */ do
        ...
    end
```

Arithmetic progressions are a common case:

```
    /* Modula-2 syntax */
    i : integer;
    ...
    for i := first to last by step do
        ...
    end
```

How might we implement this? Consider

```
        i := first
        goto L2
  L1: ...
        i += step
  L2: if i <= last goto L1
```

Several things can go wrong (generally fixed in Ada and Fortran 90, to some extent in Modula-2)

empty bounds
shouldn't execute (did in Fortran I)

changes to bounds or step size within loop
calculated up front in modern languages

direction of step
constant stepsize
"`downto`" (Pascal)
"`in reverse`" (Ada)

changes to loop variable within loop
not generally allowed in modern languages

value after the loop
especially at end-of-legal range for type (overflow?)

if local to loop, can't even name afterward, so it's just an implementation issue, not a semantic one

iteration count translation technique
needed in Fortran, which has run-time step
helpful any time the end value may be the last valid one
supported by "decrement and branch if nonzero" instruction on many machines:

```
        r1 := first
        r2 := step
        r3 := last
        r3 := ⌊(r3−r1+r2)/r2⌋
        if r3 ≤ 0 goto L2
    L1: <loop body>
        r1 := r2
        if r3 > 0 goto L1
    L2:
```

`goto`s in and out
modern languages allow only out, and structure as `exit`/`break`/`return` (or exception)