Languages with different primitives for control-flow can be tricky for automatic translation. That's especially true if you're translating from a language that allows arbitrary `goto` statements, like C, to a language that does not, like pretty much every other widely used programming language.

This module takes care of most of that complexity in two steps.

1. First, it allows you to construct a Control-Flow Graph (CFG) representing all loops, conditionals, and gotos for a function in the source program. (This is usually pretty straight-forward.)
2. Then this module can analyse that CFG and identify which parts should be treated as loops and which should be treated as `if`-statements, and what order those should appear in for the translated function.

If there are `goto` statements in the source, the output of step 2 may look very different than the input to step 1!

```
{-# LANGUAGE Rank2Types #-}
module Language.Rust.Corrode.CFG (
    Label, Terminator'(..), Terminator, BasicBlock(..),
    CFG(..), Unordered, DepthFirst, prettyCFG,
    BuildCFGT, mapBuildCFGT, addBlock, newLabel, buildCFG,
    removeEmptyBlocks, depthFirstOrder,
    prettyStructure, relooperRoot, structureCFG,
) where

import Control.Monad
import Control.Monad.Trans.State
import Data.Foldable
import qualified Data.IntMap.Lazy as IntMap
import qualified Data.IntSet as IntSet
import Data.Maybe
import Data.Traversable
import Text.PrettyPrint.HughesPJClass hiding (empty)
```

Control-Flow Graph representation

A control-flow graph is a collection of "basic blocks" containing sequential code, plus arrows indicating what to execute next when the computer reaches the end of the current basic block.

To be a valid basic block, control flow must enter only at the beginning of the block, and leave only at the end.

Basic blocks have a type parameter, `s`, for whatever type you want to use to represent the code inside the basic block. This module generally doesn't care what representation you use—a reasonable choice might be a list of statements in your target language—but whatever you choose, it should probably have an instance of both `Foldable` and `Monoid`. (The built-in list type provides both of these, for instance.) Otherwise you won't be able to use some key functions that this module provides.

(We'll discuss the `c` type parameter while explaining terminators, next.)

We assign every basic block an arbitrary "label" that we can use to refer to it from elsewhere in the control-flow graph. This could be anything, but it's convenient to use distinct integers as labels.

```
data BasicBlock s c = BasicBlock s (Terminator c)
type Label = Int
```

A basic block ends with a specification of which block to proceed to next, which we'll call the block's "terminator".

We model these cases:

- `Unreachable` indicates that the source language guarantees that control will never reach the end of this block. This is usually because the block ends with a `return` statement. But it can also happen if the block ends with a call to a function that is known to never return, for example.
- `Branch` indicates that when this block completes, control always proceeds to the specified block.
- `CondBranch` is a "conditional branch". If the specified condition is true at runtime, then control goes to the first specified block; otherwise it goes to the second block. Note that we represent conditional branches as always having both a "true" case and a "false" case; there's no implicit "fall-through" behavior like you might find for a conditional jump in assembly language, for instance.

```
data Terminator' c l
    = Unreachable
    | Branch l
    | CondBranch c l l
    deriving Show
```

The above `Terminator'` type has two generic type parameters:

The first is the type to use for condition expressions. This should probably be whatever type you use to represent boolean expressions in your target language, but this module doesn't look at what's inside those condition expressions at all, so you can use any representation you want.

The second type parameter is for whatever type you want to use for labels for basic blocks. Although we've chosen a specific `Label` type above, it's convenient to make this a type parameter so we can define instances of the standard `Functor` and `Foldable` type-classes, for generic access to the outgoing edges.

For convenience, we define a type alias that specifies that the label type is specifically the above-chosen `Label`.

```
type Terminator c = Terminator' c Label

instance Functor (Terminator' c) where
    fmap = fmapDefault

instance Foldable (Terminator' c) where
    foldMap = foldMapDefault

instance Traversable (Terminator' c) where
    traverse _ Unreachable = pure Unreachable
    traverse f (Branch l) = Branch <$> f l
    traverse f (CondBranch c l1 l2) = CondBranch c <$> f l1 <*> f l2
```

Now we can define a complete control-flow graph in terms of the previous types. It has a "start" label, indicating which basic block is the first one to start executing on entrance to a function; and a map from labels to their corresponding basic blocks.

After the CFG has been constructed, there's a pre-processing step we do to sort the basic blocks into a useful order. We use a small type-system trick here to indicate whether that sorting has happened: a value of type `CFG Unordered` has not been sorted yet, while a `CFG DepthFirst` has. A function that accepts any `CFG k` doesn't care whether the blocks have been sorted or not. So keep an eye out for that, below, because the type signatures serve as documentation of an important precondition.

With this type-system trick, the Haskell compiler will enforce that callers pass only sorted CFGs to functions that require them, which is a nice sanity check. However, within this module we still have to be careful to only tag a CFG as sorted if it actually is, and also to tag functions as requiring a sorted CFG as needed. Haskell can't magically figure that out!

```
data Unordered
data DepthFirst
```

```haskell
data CFG k s c = CFG Label (IntMap.IntMap (BasicBlock s c))

instance (Show s, Show c) => Show (CFG k s c) where
    show = render . prettyCFG (text . show) (text . show)
```

When things go wrong, it's handy to be able to print a human-readable version of an entire control-flow graph so we can inspect it. This function takes helper functions for formatting statements and conditional expressions, respectively, and uses them within each basic block to format the entire control-flow graph.

```haskell
prettyCFG :: (s -> Doc) -> (c -> Doc) -> CFG k s c -> Doc
prettyCFG fmtS fmtC (CFG entry blocks) = vcat $
    (text "start @" <> text (show entry)) : blocks'
    where
    blocks' = do
        (label, BasicBlock stmts term) <- IntMap.toList blocks
        let blockHead = text (show label) <> text ":"
        let blockBody = fmtS stmts
        let blockTail = case term of
                Unreachable -> text "// unreachable"
                Branch to -> text ("goto " ++ show to ++ ";")
                CondBranch cond true false ->
                    text "if(" <> fmtC cond
                        <> text ") goto " <> text (show true)
                        <> text "; else goto " <> text (show false)
                        <> text ";"
        blockHead : map (nest 4) [blockBody, blockTail] ++ [text ""]
```

Constructing CFGs

This module provides a small monadic interface for constructing control-flow graphs. It's provided as a "monad transformer", meaning that you can combine this monad with other monads. For example, if you need to keep information about variable declarations that are in scope in order to translate statements and expressions correctly, you can use a `State` monad for that, and layer this `BuildCFGT` monad on top of it. Then you can use actions from either monad as needed.

```haskell
type BuildCFGT m s c = StateT (BuildState s c) m
```

Because this is a monad transformer, you may find you need to perform some operation transforming the underlying monad. For example, `Reader` monads have a `local` operation that runs some specified monadic action with a different value in its environment than the outer computation uses, and similarly with the `Writer` monad's `listen` and `censor` operations. To use these kinds of operations with `BuildCFGT`, you need to wrap them in `mapBuildCFGT`.

The type signature here is a little bit weird. Your function has to preserve the current state of the CFG builder, because we're suspending the usual monad rules that would normally carry that state around behind the scenes. But we don't allow you to peek at or modify the CFG builder state along the way. That's enforced by using the GHC `Rank2Types` language extension (enabled at the top of this module) to declare that your transformation must work for all possible state types: Code that must work for all possible types can't possibly do anything but pass that data along unchanged.

```haskell
mapBuildCFGT
    :: (forall st. m (a, st) -> n (b, st))
    -> BuildCFGT m s c a -> BuildCFGT n s c b
mapBuildCFGT = mapStateT
```

While constructing a new control-flow graph, we need to keep track of two things: any basic blocks constructed so far, and a unique label to use for the next basic block. We keep both in a new data type, `BuildState`.

It might seem like we shouldn't need to keep a separate counter for unique labels. Couldn't we just look at the label for the last block that was constructed, add 1, and use that as the next block's label?

Unfortunately, during CFG construction we often need to refer to blocks that we haven't constructed yet. For example, to construct a loop, we might construct the body of the loop with a branch back to the loop header, and only then construct the loop header with a branch into the body.

That means we may have to generate any number of labels before finishing the corresponding blocks, so we have to keep track of which IDs we already handed out.

Note that this also means that this intermediate representation of a CFG is generally not a valid CFG, because it includes blocks that branch to other blocks that haven't been constructed yet. It's the caller's responsibility to ensure that all blocks get added eventually.

```
data BuildState s c = BuildState
    { buildLabel :: Label
    , buildBlocks :: IntMap.IntMap (BasicBlock s c)
    }
```

`newLabel` just returns a unique `Label`.

```
newLabel :: Monad m => BuildCFGT m s c Label
newLabel = do
    old <- get
    put old { buildLabel = buildLabel old + 1 }
    return (buildLabel old)
```

`addBlock` saves the given statements and terminator in the state.

```
addBlock :: Monad m => Label -> s -> Terminator c -> BuildCFGT m s c ()
addBlock label stmt terminator = do
    modify $ \ st -> st
        { buildBlocks = IntMap.insert label (BasicBlock stmt terminator)
            (buildBlocks st)
        }
```

Finally we have the function that runs a builder and returns the CFG that it built. The builder's return value must be the label to use as the entry-point for the control-flow graph.

Note that the constructed CFG is tagged as `Unordered` because we haven't sorted it yet.

```
buildCFG :: Monad m => BuildCFGT m s c Label -> m (CFG Unordered s c)
buildCFG root = do
    (label, final) <- runStateT root (BuildState 0 IntMap.empty)
    return (CFG label (buildBlocks final))
```

It's normal to write simple translations for building the CFG that produce some pretty silly-looking control-flow graphs. For example, they may produce a lot of basic blocks that have no statements in them and just unconditionally branch somewhere else. Those blocks can be safely removed, if we're a little careful, without changing the meaning of the CFG, and that's what `removeEmptyBlocks` does.

> **NOTE**: I don't think this is necessary; all of the following algorithms should produce the same output even with empty blocks present, as far as I can figure. But when something goes wrong and we need to report an error, it's nice to have a simpler CFG to examine. So I'm not deleting this, but I'm not going to bother documenting how it works because it isn't important.
>
> **TODO**: Think about whether this can be folded into `depthFirstOrder` without making that function too complicated.

```
removeEmptyBlocks :: Foldable f => CFG k (f s) c -> CFG Unordered (f s) c
removeEmptyBlocks (CFG start blocks) = CFG (rewrite start) blocks'
```

```
    where
    go = do
        (empties, done) <- get
        case IntMap.minViewWithKey empties of
            Nothing -> return ()
            Just ((from, to), empties') -> do
                put (empties', done)
                step from to
                go
    step from to = do
        (empties, done) <- get
        case IntMap.splitLookup to empties of
            (_, Nothing, _) -> return ()
            (e1, Just to', e2) -> do
                put (e1 `IntMap.union` e2, done)
                step to to'
        (empties', done') <- get
        let to' = IntMap.findWithDefault to to done'
        put (empties', IntMap.insert from to' done')
    isBlockEmpty (BasicBlock s (Branch to)) | null s = Just to
    isBlockEmpty _ = Nothing
    rewrites = snd $ execState go (IntMap.mapMaybe isBlockEmpty blocks, IntMap.empty)
    rewrite to = IntMap.findWithDefault to to rewrites
    discards = IntMap.keysSet (IntMap.filterWithKey (/=) rewrites)
    rewriteBlock from _ | from `IntSet.member` discards = Nothing
    rewriteBlock _ (BasicBlock b term) = Just (BasicBlock b (fmap rewrite term))
    blocks' = IntMap.mapMaybeWithKey rewriteBlock blocks
```

Transforming CFGs to structured programs

Once we've constructed a CFG, the real challenge is to turn that messy pile of basic blocks back into structured control flow.

This implementation would work for a pretty wide variety of languages. It assumes the target language has:

1. If-then-else,
2. Loops,
3. Multi-level exits from loops.

That last point needs some explanation. Most languages with loops provide some way for the programmer to break out of a loop early, or restart at the beginning of the loop without finishing the current iteration. (Let's call both kinds of control-flow "loop exits".) Of those languages, many but not all of them allow the programmer to exit more than one loop in one go, by giving loops names and specifying which loop to exit by name. This code assumes that your target language is one of the latter kind.

```
data StructureLabel s c
    = GoTo { structureLabel :: Label }
    | ExitTo { structureLabel :: Label }
    | Nested [Structure s c]
    deriving Show

type StructureTerminator s c = Terminator' c (StructureLabel s c)
type StructureBlock s c = (s, StructureTerminator s c)

data Structure' s c a
```

```
    = Simple s (StructureTerminator s c)
    | Loop a
    | Multiple (IntMap.IntMap a) a
    deriving Show

data Structure s c = Structure
    { structureEntries :: IntSet.IntSet
    , structureBody :: Structure' s c [Structure s c]
    }
    deriving Show

prettyStructure :: (Show s, Show c) => [Structure s c] -> Doc
prettyStructure = vcat . map go
    where
    go (Structure _ (Simple s term)) = text (show s ++ ";") $+$ text (show term)
    go (Structure entries (Loop body)) = prettyGroup entries "loop" (prettyStructure body)
    go (Structure entries (Multiple handlers unhandled)) = prettyGroup entries "match" $
        vcat [ text (show entry ++ " =>") $+$ nest 2 (prettyStructure handler) | (entry, handler) <- In
        $+$ if null unhandled then mempty else (text "_ =>" $+$ nest 2 (prettyStructure unhandled))

    prettyGroup entries kind body =
        text "{" <> hsep (punctuate (text ",") (map (text . show) (IntSet.toList entries))) <> text ("}
        $+$ nest 2 body

relooperRoot :: Monoid s => CFG k s c -> [Structure s c]
relooperRoot (CFG entry blocks) = relooper (IntSet.singleton entry) $
    IntMap.map (\ (BasicBlock s term) -> (s, fmap GoTo term)) blocks

relooper :: Monoid s => IntSet.IntSet -> IntMap.IntMap (StructureBlock s c) -> [Structure s c]
relooper entries blocks =
```

First we partition the entry labels into those that some block may branch to versus those that none can branch to. The key idea is that entry labels need to be placed early in the output, but if something later can branch to them, then we need to wrap them in a loop so we can send control flow back to the entry point again.

Each of these cases makes at least one recursive call. To ensure that this algorithm doesn't get stuck in an infinite loop, we need to make sure that every recursive call has a "simpler" problem to solve, such that eventually each subproblem has been made so simple that we can finish it off immediately. We'll show that the subproblems truly are simpler in each case.

```
    let (returns, noreturns) = partitionMembers entries $ IntSet.unions $ map successors $ IntMap.elems
        (present, absent) = partitionMembers entries (IntMap.keysSet blocks)
    in case (IntSet.toList noreturns, IntSet.toList returns) of
```

If there are no entry points, then the previous block can't reach any remaining blocks, so we don't need to generate any code for them. This is the primary recursive base case for this algorithm.

```
    ([], []) -> []
```

Simple blocks

If there's only one label and it is *not* the target of a branch in the current set of blocks, then simply place that label next in the output.

This case always removes one block from consideration before making the recursive call, so the subproblem is one block smaller.

```
([entry], []) -> case IntMap.updateLookupWithKey (\ _ _ -> Nothing) entry blocks of
    (Just (s, term), blocks') -> Structure
        { structureEntries = entries
        , structureBody = Simple s term
        } : relooper (successors (s, term)) blocks'
```

If the target is a block that we've already decided to place somewhere later, then we need to construct a fake block that tells the code generator to set the current-block state variable appropriately.

```
    (Nothing, _) -> Structure
        { structureEntries = entries
        , structureBody = Simple mempty (Branch (GoTo entry))
        } : []
```

Skipping to blocks placed later

When there are multiple entry labels and some or all of them refer to blocks that we have already decided to place somewhere later, we need some way to skip over any intervening code until control flow reaches wherever we actually placed these blocks. (Note that if we need to branch to a block that we placed somewhere earlier, then we'll have already constructed a loop for that, so we don't need to handle that case here.)

We accomplish this by constructing a `Multiple` block with an empty branch for each absent entry label, and an else-branch that contains the code we may want to skip. This gets control flow to the end of the enclosing block. If the target block isn't there either, then we'll do this again at that point, until we've gotten all the way out to a block that does contain the target label.

However, if we don't have any code to place in the else-branch, then this procedure would generate a no-op `Multiple` block, so we can avoid emitting anything at all in that case.

```
_ | not (IntSet.null absent) ->
    if IntSet.null present then [] else Structure
        { structureEntries = entries
        , structureBody = Multiple
            (IntMap.fromSet (const []) absent)
            (relooper present blocks)
        } : []
```

Loops

If all the entry labels are targets of branches in some block somewhere, then construct a loop with all those labels as entry points.

To keep the generated code simple, we want to eliminate any absent entries (the previous case) before constructing a loop. If we generate a loop with absent entry points, then to handle those inside the loop we'd need to `break` out of the loop. By doing it in this order instead, we don't need any code at all in the handlers for the absent branches.

In this case, we have one recursive call for the body of the loop, and another for the labels that go after the loop.

- The loop body has the same entry labels. However, for the recursive call we remove all the branches that made it a loop, so we're guaranteed to not hit this case again with the same set of entry labels. As long as the other cases reduce the number of blocks, we're set.
- For the labels following the loop, we've removed at least the current entry labels from consideration, so there are fewer blocks we still need to structure.

```
([], _) -> Structure
    { structureEntries = entries
```

```
    , structureBody = Loop (relooper entries blocks')
    } : relooper followEntries followBlocks
    where
```

The labels that should be included in this loop's body are all those which can eventually come back to one of the entry points for the loop.

Note that `IntMap.keysSet returns' == entries`. If some entry were not reachable from any other entry, then we would have split it off into a `Multiple` block first.

```
    returns' = (strictReachableFrom `IntMap.intersection` blocks) `restrictKeys` entries
    bodyBlocks = blocks `restrictKeys`
        IntSet.unions (IntMap.keysSet returns' : IntMap.elems returns')
```

Now that we've identified which labels belong in the loop body, we can partition the current blocks into those that are inside the loop and those that follow it.

```
    followBlocks = blocks `IntMap.difference` bodyBlocks
```

Any branches that go from inside this loop to outside it form the entry points for the block following this one. (There can't be any branches that go to someplace earlier in the program because we've already removed those before recursing into some loop that encloses this one.)

```
    followEntries = outEdges bodyBlocks
```

At this point we've identified some branches as either a `break` (so it's in `followEntries`) or a `continue` (because it was one of this loop's entry points) branch. When we recurse to structure the body of this loop, we must not consider those branches again, so we delete them from the successors of all blocks inside the loop.

Note that `structureEntries` for this loop block records the labels that are `continue` edges, and `structureEntries` for the subsequent block records the labels that are `break` edges, so we don't need to record any additional information here.

If we fail to delete some branch back to the loop entry, then when we recurse we'll generate another `Loop` block, which might mean the algorithm never terminates.

If we fail to delete some branch that exits the loop, I think the result will still be correct, but will have more `Multiple` blocks than necessary.

```
    markEdge (GoTo label)
        | label `IntSet.member` (followEntries `IntSet.union` entries)
        = ExitTo label
    markEdge edge = edge
    blocks' = IntMap.map (\ (s, term) -> (s, fmap markEdge term)) bodyBlocks
```

Multiple-entry blocks

Otherwise, we need to merge multiple control flow paths at this point, by constructing code that will dynamically check which path we're supposed to be on.

In a `Multiple` block, we construct a separate handler for each entry label that we can safely split off. We make a recursive call for each handler, and one more for all the blocks we couldn't handle in this block.

- If there are unhandled blocks, then each handler contains fewer blocks than we started with. If we were able to handle all the entry labels, then we've partitioned the blocks into at least two non-empty groups, so each one is necessarily smaller than we started with. There must be at least two entry labels because if there weren't any no-return entries then we'd have constructed a loop, and if there were only one no-return entry and no entries that can be returned to, we'd have constructed a simple block.
- Each handler consumes at least its entry label, so as long as we generate at least one handler, the recursive call for the unhandled blocks will have a smaller subproblem. We can only handle an entry label if none of the other entry labels can, through any series of branches, branch to this label. But

because we aren't in the case above for constructing a loop, we know that at least one entry label has no branches into it, so we're guaranteed to consume at least one block in this pass.

```
_ -> Structure
    { structureEntries = entries
    , structureBody = Multiple handlers unhandled
    } : relooper followEntries followBlocks
    where
```

The elements in the `singlyReached` map are disjoint sets. Proof: keys in an `IntMap` are distinct by definition, and the values after `filter` are singleton sets; so after `flipEdges`, each distinct block can only be attached to one entry label.

```
reachableFrom = IntMap.unionWith IntSet.union (IntMap.fromSet IntSet.singleton entries) strictR
singlyReached = flipEdges $ IntMap.filter (\ r -> IntSet.size r == 1) $ IntMap.map (IntSet.inte
```

Some subset of the entries are now associated with sets of labels that can only be reached via that entry. Mapping these to their corresponding blocks preserves the property that they're disjoint.

In addition, only labels that are permitted to appear inside this `Multiple` block will remain after this. Labels which have already been assigned to a later block won't get duplicated into this one, so we'll have to generate code to ensure that control continues to the later copy.

```
handledEntries = IntMap.map (\ within -> blocks `restrictKeys` within) singlyReached
```

If one of the entry labels can reach another one, then the latter can't be handled in this `Multiple` block because we'd have no way to make control flow from one to the other. These unhandled entries must be handled in subsequent blocks.

```
unhandledEntries = entries `IntSet.difference` IntMap.keysSet handledEntries
```

All labels that are reachable only from the entry points that we *are* handling, however, will be placed somewhere inside this `Multiple` block. Labels that are left over will be placed somewhere after this block.

```
handledBlocks = IntMap.unions (IntMap.elems handledEntries)
followBlocks = blocks `IntMap.difference` handledBlocks
```

The block after this one will have an entry point for each of this block's unhandled entries, and in addition, one for each branch that leaves this `Multiple` block.

```
followEntries = unhandledEntries `IntSet.union` outEdges handledBlocks
```

Finally, we've partitioned the entries and labels into those which should be inside this `Multiple` block and those which should follow it. Recurse on each handled entry point.

```
makeHandler entry blocks' = relooper (IntSet.singleton entry) blocks'
allHandlers = IntMap.mapWithKey makeHandler handledEntries
```

At this point we could throw all the handlers into a `Multiple` block and leave the `unhandled` portion empty. However, that generates code that is both more complicated than necessary, and sometimes wrong, in the case where we have a handler for every entry label. In that case, if control reaches the guard for the last handler, then the condition must always evaluate true, so we can replace a final `else if` statement with an unconditional `else`.

We can prove this using our precise knowledge of the set of values that the current-block variable could have at this point. But very few compilers could prove it, because for the general case, tracking precise value sets is hard and compiler writers don't usually consider the effort worth-while.

As a result, if this block is the last one in a function and every handler is supposed to return a value, a compiler that verifies that some value is returned on every path will conclude that some path is missing a `return` statement, even though we know that path is unreachable.

So, if we have a handler for every entry point, pick one to be the `else` branch of this block. Otherwise, there is no `else` branch.

```
        (unhandled, handlers) = if IntMap.keysSet allHandlers == entries
            then
                let (lastHandler, otherHandlers) = IntMap.deleteFindMax allHandlers
                in (snd lastHandler, otherHandlers)
            else ([], allHandlers)

    where
    strictReachableFrom = flipEdges (go (IntMap.map successors blocks))
        where
        grow r = IntMap.map (\ seen -> IntSet.unions $ seen : IntMap.elems (r `restrictKeys` seen)) r
        go r = let r' = grow r in if r /= r' then go r' else r'

restrictKeys :: IntMap.IntMap a -> IntSet.IntSet -> IntMap.IntMap a
restrictKeys m s = m `IntMap.intersection` IntMap.fromSet (const ()) s

outEdges :: IntMap.IntMap (StructureBlock s c) -> IntSet.IntSet
outEdges blocks = IntSet.unions (map successors $ IntMap.elems blocks) `IntSet.difference` IntMap.keysS

partitionMembers :: IntSet.IntSet -> IntSet.IntSet -> (IntSet.IntSet, IntSet.IntSet)
partitionMembers a b = (a `IntSet.intersection` b, a `IntSet.difference` b)

successors :: StructureBlock s c -> IntSet.IntSet
successors (_, term) = IntSet.fromList [ target | GoTo target <- toList term ]

flipEdges :: IntMap.IntMap IntSet.IntSet -> IntMap.IntMap IntSet.IntSet
flipEdges edges = IntMap.unionsWith IntSet.union [ IntMap.fromSet (const (IntSet.singleton from)) to |
```

Eliminating unnecessary multiple-entry blocks

```
simplifyStructure :: Monoid s => [Structure s c] -> [Structure s c]
simplifyStructure = foldr go [] . map descend
    where
    descend structure = structure { structureBody =
        case structureBody structure of
        Simple s term -> Simple s term
        Multiple handlers unhandled ->
            Multiple (IntMap.map simplifyStructure handlers) (simplifyStructure unhandled)
        Loop body -> Loop (simplifyStructure body)
    }
```

If there's a `Simple` block immediately followed by a `Multiple` block, then we know several useful facts immediately:

- The `Simple` block terminates with a conditional branch, where both targets are distinct `GoTo` labels. Otherwise, the next block wouldn't have enough entry points to be a `Multiple` block.
- Each target of the conditional branch either has a handler it can be replaced by from the `Multiple` block, or it can be replaced with the unhandled blocks.
- Every non-empty branch of the `Multiple` block will be used by this process, so no code will be lost.
- This simplification never duplicates code.

The one tricky thing here is that under some circumstances we need to ensure that there's a `mkGoto` statement emitted in some branches. Conveniently, here we can unconditionally insert an empty `Simple` block ending

in a `GoTo` branch, and let `structureCFG` decide later whether that requires emitting any actual code.

```
    go (Structure entries (Simple s term))
       (Structure _ (Multiple handlers unhandled) : rest) =
        Structure entries (Simple s (fmap rewrite term)) : rest
        where
        rewrite (GoTo to) = Nested
            $ Structure (IntSet.singleton to) (Simple mempty (Branch (GoTo to)))
            : IntMap.findWithDefault unhandled to handlers
        rewrite _ = error ("simplifyStructure: Simple/Multiple invariants violated in " ++ show entries)

    go block rest = block : rest

-- We no longer care about ordering, but reachability needs to only include
-- nodes that are reachable from the function entry, and this has the side
-- effect of pruning unreachable nodes from the graph.
depthFirstOrder :: CFG k s c -> CFG DepthFirst s c
depthFirstOrder (CFG start blocks) = CFG start' blocks'
    where
    search label = do
        (seen, order) <- get
        unless (label `IntSet.member` seen) $ do
            put (IntSet.insert label seen, order)
            case IntMap.lookup label blocks of
                Just (BasicBlock _ term) -> traverse_ search term
                _ -> return ()
            modify (\ (seen', order') -> (seen', label : order'))
    final = snd (execState (search start) (IntSet.empty, []))
    start' = 0
    mapping = IntMap.fromList (zip final [start'..])
    rewrite label = IntMap.findWithDefault (error "basic block disappeared") label mapping
    rewriteBlock label (BasicBlock body term) = (label, BasicBlock body (fmap rewrite term))
    blocks' = IntMap.fromList (IntMap.elems (IntMap.intersectionWith rewriteBlock mapping blocks))
```

Generating final structured code

With all the preliminary analyses out of the way, we're finally ready to turn a control-flow graph back into a structured program full of loops and `if`-statements!

Since this module is not language-specific, the caller needs to provide functions for constructing `break`, `continue`, loop, and `if` statements. The loop-related constructors take a label and generate a loop name from it, to support multi-level exits.

```
structureCFG
    :: Monoid s
    => (Maybe Label -> s)
    -> (Maybe Label -> s)
    -> (Label -> s -> s)
    -> (c -> s -> s -> s)
    -> (Label -> s)
    -> ([(Label, s)] -> s -> s)
    -> CFG DepthFirst s c
    -> (Bool, s)
structureCFG mkBreak mkContinue mkLoop mkIf mkGoto mkMatch cfg =
    (hasMultiple root, foo [] mempty root)
    where
```

```
    root = simplifyStructure (relooperRoot cfg)
    foo exits next' = snd . foldr go (next', mempty)
        where
        go structure (next, rest) = (structureEntries structure, go' structure next `mappend` rest)

        go' (Structure entries (Simple body term)) next = body `mappend` case term of
                Unreachable -> mempty
                Branch to -> branch to
                CondBranch c t f -> mkIf c (branch t) (branch f)
            where
            branch (Nested nested) = foo exits next nested
            branch to | structureLabel to `IntSet.member` next =
                insertGoto (structureLabel to) (next, mempty)
            branch (ExitTo to) | isJust target = insertGoto to (fromJust target)
                where
                inScope immediate (label, local) = do
                    (follow, mkStmt) <- IntMap.lookup to local
                    return (follow, mkStmt (immediate label))
                target = msum (zipWith inScope (const Nothing : repeat Just) exits)
            branch to = error ("structureCFG: label " ++ show (structureLabel to) ++ " is not a valid e

            insertGoto _ (target, s) | IntSet.size target == 1 = s
            insertGoto to (_, s) = mkGoto to `mappend` s

        go' (Structure _ (Multiple handlers unhandled)) next =
            mkMatch [ (label, foo exits next body) | (label, body) <- IntMap.toList handlers ] (foo exi

        go' (Structure entries (Loop body)) next = mkLoop label (foo exits' entries body)
            where
            label = IntSet.findMin entries
            exits' =
                ( label
                , IntMap.union
                    (IntMap.fromSet (const (entries, mkContinue)) entries)
                    (IntMap.fromSet (const (next, mkBreak)) next)
                ) : exits

hasMultiple :: [Structure s c] -> Bool
hasMultiple = any (go . structureBody)
    where
    go (Multiple{}) = True
    go (Simple _ term) = or [ hasMultiple nested | Nested nested <- toList term ]
    go (Loop body) = hasMultiple body
```