Question 1

The Trinket (try-n-catch) language starts with the same semantics and features as Garter, and adds support for try-catch expressions and gives names to existing runtime errors.

As try-catch expressions are fundamentally control-flow operations there are no changes to runtime data representations for operations. Runtime specificities for implementing this control flow will be discussed later as an integration with compilation.

Errors vs Exceptions

At the moment Garter has a notion of error codes, which flow from the compiled snake language to the C runtime when one of a well-defined group of error-ful scenarios occur. In order to give more control to the programmer when writing try-catch expressions, most of the error codes is given an 'Exception' in Trinket for human-readable (and writable) identification. The notable exception is `err_OUT_OF_MEMORY`, since part of the Trinket language's modus operandi is not letting the programmer quibble with memory.

```
err_COMP_NOT_NUM => ComparisonException
err_ARITH_NOT_NUM => ArithmeticException
err_LOGIC_NOT_BOOL => LogicException
err_IF_NOT_BOOL => ConditionalException
err_OVERFLOW => OverflowException
err_ACCESS_NOT_TUPLE => NonTupleAccessException
err_INDEX_NOT_NUM => NonNumberIndexException
err_ACCESS_LOW_INDEX => LowIndexExeption
err_ACCESS_HIGH_INDEX = HighIndexException
err_NIL_DEREF = NilDereferenceException
err_CALL_NOT_CLOSURE => NonClosureCallException
err_CALL_ARITY => ArityException
```

Exceptions in Trinket are not values, they are more of an annotation system to give the user more fine-grained control over their exception handling. Most of the named exceptions are associated with a value - that which caused the error in its own context and would be displayed in a Garter error message. Here are some examples of expressions which would "throw" an Exception and the values they would "contain".

```
false < 3 => ComparisonException false
4 + true => ArithmeticException true
4 || false => LogicException 4
if 2 : 1 else: 3 => ConditionalException 2
12345678912345 * 12345678912345 => OverflowException -1603819415859337999
4[3] => NotTupleException 4
(1, 2, 3)[true] => IndexNotNumException true
(1, 2, 3)[-1] => LowIndexExeption -1
(1, 2, 3)[3] => HighIndexException 3
nil[1] => NilDerefException # no bound value, it is trivially known to be nil
4(1) => NotClosureException 4
(lambda: 4)(3) => ArityException 1
```

Try-Catch Semantics and Examples

Firstly, should any of the aforementioned error-codes occur entirely outside of any applicable try-catch expression, the behavior in Trinket will be the same as Garter.

Try-catch expressions in Trinket are exactly that - expressions. As such, whichever of the two blocks runs to completion provides the answer for the expression as a whole.

A Trinket programmer has the ability to specify exactly which Exception they are concerned with by annotating the catch portion of the expression. Using one of the Exceptions defined above means the catch will apply only to that Exception, so the programmer has to be intent-ful when doing so.

```
try {
    4 + true
} catch (ArithmeticException value) {
    value
} => true

try {
    4 + true
} catch (ArityException value) {
    value
} => <runtime error err_ARITH_NOT_NUM>
```

But an Exception annotation is optional, and without one the catch will handle any error scenario (other than out-of-memory) - at the cost of losing any information about what value caused the error.

```
try {
  4 + true
} catch {
  4 + 2
} => 6
```

Or, if a Trinket programmer wants to catch a specific Exception but ignore its associated value, an underscore can be used.

```
try {
  4 + true
} catch (ArithmeticException _) {
  3
} => 3
```

Try-catch expressions can be nested, and all handlers surrounding an expression handle an error from inside it.

```
try {
  try {
    false || 4
  } catch (ArithmeticExeption value) {
    value
  }
} catch (LogicException value) {
    value
} => 4

try {
  false || 4
} catch (LogicException value) {
  try {
    value[0]
  } catch (NonTupleAccessException nt) {
    nt
  }
} => 4
```

For nested handlers of the same exception, the most interior catches the error.

```
try {
  try {
    4 + nil
  } catch (ArithmeticException value) {
    true
  }
} catch (ArithmeticError value) {
    false
} => true
```

When errors occur in the try-block, execution of the try-block immediately ceases and execution is moved to the handler.

```
try {
    let a = print(4 + true) in
    1
} catch (ArithmeticException value) {
    value
} => true # (no printing)
```

Cross-linguistically, it's not uncommon for the catch-block in a try-catch (or a different handler…) to be concerned with cleaning up potentially incomplete work. Thus, though Trinket has few side-effects that impact data, handlers in a `try-catch` expression are defined to exist in the same scope or environment as the try-block. They do not close over or copy free values.

```
let t = (1, 2, 3) in
  try {
    t[1] := 4;
    t[5] := nil
  } catch (HighIndexException ind) {
    t
  } => (1, 4, 3)
```

Additional Syntax

The grammar for Trinket will have one addition to Garter in the `expr` data-type:

```
type 'a expr =
  ...
  (* try_expr, catch_expr, annot_except *)
  | ETryCatch of 'a expr * 'a expr * 'a annotated_exception * 'a
```

Where `annotated_exeption` is as follows. For those Exceptions which are associated with a value, `Some string` means the user provided a bound name for this value and `None` means they provided an underscore. `Exception` represents an un-annotated `try-catch`.

```
type 'a annotated_exception =
  | Exception 'a
  | ComparisonException of string option * 'a
  | ArithmeticException of string option * 'a
  | LogicException of string option * 'a
  | ConditionalException of string option * 'a
  | OverflowException of string option * 'a
  | NotTupleException of string option * 'a
  | IndexNotNumException of string option * 'a
  | LowIndexExeption of string option * 'a
  | HighIndexException of string option * 'a
  | NilDerefException * 'a
  | NotClosureException of string option * 'a
  | ArityException of string option * 'a
```

Changes to Pre-compilation Passes

Parsing

Rely on the parser for using the appropriate `annotated_exception` constructor based off the source code, parser error if invalid Exception name or wrong number of bound values.

Well-formed

The only allowable `bind`'s in an `annotated_exception` are `BBLank` and `BName`.

Desugaring

No additional desugaring.

Tag/Untag

Trivially recursive over all of `ETryCatch`'s components.

Renaming

If there is a `Some string` in the `annotated_exception`, rename its contained string and replace all usages in `catch_expr`. Then recur into both component expressions.

ANF

The ANF-ed form of `ETryCatch` is a `cexpr`, but both internal expressions are `aexpr`'s like `CIf`. Something like:

```
type 'a cexpr =
  ...
  (* try_expr, catch_expr, annot_except *)
  | CTryCatch of 'a aexpr * 'a aexpr * 'a annotated_exception * 'a
```

Stack Allocation

Stack allocation will continue recursively into both `aexpr` components of the ANF-ed `try-catch`, but there are a few special things to take care of first.

1. Allocate one stack slot that will be live in the `try-catch` expression as a whole - this will be used for book-keeping where to jump back to upon catching an error. Associate it with a name that can be procedurally generated during compilation.
2. If the `annotated_exception` contains a `Some name`, allocate a stack slot that will be live throughout `catch_expr` and associate it with `name` in the environment.

Runtime

Try-catch blocks/expressions are a typical example of continuations, and the canonical way of implementing continuations in C/Unix-land is using the `jmp_buf` data structure and the accompanying functions `setjmp` and `longjmp`. This is done to preserve the execution context of the jump destination and thus to ensure the continuation proceeds as written despite whatever happened in the aborted computation.

These are C library functions, and thus it will likely be easiest to deal with them directly inside the existing C runtime in Trinket.

The following is based off my understanding of setjmp and longjmp from documentation, having never used them.

However, since the code for our catch handler lives in Trinket, not C, we need to call `setjmp` from inside Trinket. Additionally, `longjmp` returns you to the exact place where `setjmp` was called, and we want that to be in the Trinket program *not* the C runtime. Thus we want a helper runtime function to allocate a `jmpbuf` and return its pointer to the Trinket program for storage and (potentially) later use. This is what the first stack slot mentioned above is for.

Since a `jmp_buf` is well bigger than a word (though its exact size is platform dependent), it won't fit nicely into our Trinket stack. They will need to be heap-allocated - the question is where? If they go on the Trinket heap garbage collection will need to be able to handle them and all their internal values. If we allocate them in a C heap using `malloc` then we'll need to be careful to free them in order to prevent memory leaks.

Either way could work, but it is likely much more straightforward to combine `malloc` with a bit of C book-keeping instead of making such large modifications to Trinket's garbage collection.

Compilation

To compile a `try-catch` we start off by getting a `jmp_buf` from our runtime, in the form of its pointer. Stash the pointer in the stack slot allocated to its procedurally generated name from stack allocation.

Call `setjmp` with the dereferenced value of the `jmp_buf` pointer, and check what was returned. If it returned 0, then the `jmp_buf` has been initialized with the current execution information of the program. In this scenario, we would fall-through and execute the compiled try-block.

If the value wasn't 0 then we have jumped back to this point from somewhere in the "future" via `longjmp` (we "threw" an exception which was caught by this catch). In this scenario we jump to the beginning of the catch-block and then move the returned value of `setjmp` (whatever is in RAX) into the allocated slot for the exception's bound value (if there is one). This value is provided by the `longjmp` which brought use back to here.

The layout of the compiled try-catch expression is something like:

```
<get jmp_buf pointer from runtime>
<call set_jmp>
<if RAX = 0 then (fall through) else (jmp catch-block)
<try_block>
<jmp try_catch_done>
<catch_block>
<label: try_catch_done>
```

However, our existing compilation does not actually do the work of ”throwing” exceptions.

Change the signature of the compilation functions (at least for `aexpr` and `cexpr`) to include an accumulator that associates Exception names to catch handlers. This can likely be done multiple ways (tag, stack slot, name).

As we recursively compile the try-block of a try-catch expression, add the associated catch handler to this accumulator. Make sure to overwrite existing entries for any given Exception name to achieve the previously described semantics for nested try-catch's.

This accumulated mapping comes into play when compiling our collection of value checks. At compile time, for any given potential error, we know if we have a surrounding catch for its associated Exception. If we do, we replace the `call` / `jmp` to the runtime's `error` function with a call to `longjmp` - passing in the `jmpbuf` and the value which caused the exception. Make sure what gets passed to `longjmp` is the `jmp_buf` itself and not its pointer.

Question 2

Trinket now gains a `finally` block as an addition to the existing `try-catch` expressions - making them `try-catch-finally` (or `try-finally`) expressions.

Semantics

As said above, a `finally` block can occur with or without a `catch` - but it does need a `try`. The general rule for `finally` execution is that whenever the program leaves its associated `try/catch` (via an exception caught in an exterior handler, an entirely uncaught error, or simply finishing execution) the `finally` block will run.

```
try {
  print(1);
  4 + true;
  print(2)
} finally {
  print(3);
}
=>
1 # printed
3 # printed
<runtime error msg>

try {
  print(1);
  4 + true;
  print(2)
} catch {
  print(3);
} finally {
  print(4);
}
=>
1
3
4
```

For nested `try/catch-finally` blocks the `finally`'s are executed inside-out.

```
try {
    try {
      1
    } catch {
      4
    } finally {
      print(false);
    }
} finally {
  print(true);
}
=>
false # printed
true  # printed
true  # return value
```

If an exception is caught by a catch exterior of some finally's, the interior finally's are executed before the catch handler.

```
try {
    try {
      4 + true
    } finally{
      print(false);
    }
} catch (ArithmeticException _) {
  print(4)
} finally {
  print(true);
}
=>
false # printed
4     # printed
true  # printed
true  # return value
```

Additional Syntax

The grammar for Trinket will have another addition to the `expr` data-type:

```
type 'a expr =
  ...
  (* try(_catch)_expr, finally_expr *)
  | EFinally of 'a expr * 'a expr * 'a
```

Changes to Pre-compilation Passes

Parsing

If it is a `try-catch-finally` expression then the first `expr` will be a `ETryCatch`. If it is a `try-finally` expression then the first expression can be any `expr`, representing the body of the try-block.

Well-formed

No additional well-formedness checks as the parser enforces what goes into the `EFinally` constructor.

Desugaring

No additional desugaring.

Tag/Untag

Trivially recursive over all of `EFinally`'s components.

Rename

Trivially recursive over all of `EFinally`'s components.

ANF

The ANF-ed form of `EFinally` is a `cexpr`, but both internal expressions are `aexpr`'s like `CTryCatch`. Something like:

```
type 'a cexpr =
  ...
  (* try(_catch)_expr, finally_expr, annot_except *)
  | CFinally of 'a aexpr * 'a aexpr * 'a
```

Stack Allocation

Trivially recursive into both of the component `aexpr`'s.

Runtime

There are no explicit changes to the runtime needed to support `finally`.

Compilation

Our book-keeping for finally is going to build on the added accumulator mapping for `try-catch`. In addition to knowing where to jump for each catch handler, the RHS of the map is also going to keep track of all the `finally` blocks that need to be executed *before* running that handler (all the finally blocks inside the handler's `try`).

That way, before we do the `longjmp` to a handler we can run through all of the `finally`'s in its list. And, for an entirely uncaught error, we can run through *all* the `finally` blocks in the program before bailing out to the runtime.

Unfortunately, the `setjmp` / `longjmp` system doesn't lend itself to this chaining of `finally`'s because it loses track of where it jumped from. Thus we will be compiling `finally`'s as ASM functions and using the `call=/=ret` pattern to sequentially execute multiple `finally`'s in the same code location.

When compiling a `CFinally`, first compile the `try-expr` side recursively after adding the new `finally` block to the appropriate entries in the continuation accumulator. Then add a call to the new `finally` block after all the code for the `try-expr` so that no matter what path it took, the `finally` gets executed.

Next, compile the `finally` block as a function in ASM (with a final `ret` in addition to a starting label). Make sure there are appropriate jumps and labels so execution does not fall through the compiled `finally` blocks after executing the `try-expr` (we don't want double execution of the `finally`). Also ensure that the environment for the `finally` is the same as the `try`, and that the local variables on the stack are accessed correctly despite the `call`'s stack manipulation. This should be predictable at compile-time.