Exception Design

Design Goals

Provide moto language level support for Exception handling. Specifically support for try, catch and finally blocks. Exceptions should be Objects. All exceptions should have a getMesssage() method method.

Design Questions

Q. How does the exception handling system work now ?

A. Ok, so we we have these TryBlock objects. Try calls **setjmp**(tryblk->jmp) and excp_throw calls **lngjmp**(tryblk->jmp) with code excp->t.code. Catch returns true if etype.code == the thrown exception type. For many exceptions **Finally could just be a noop followed by {...} .** However difficulties can arise when exceptions are thrown within catch or other finally blocks. ***See the discussion on implementing finally***

Q. Why do we need a pre-allocated exception for each process / thread?

A. Suppose we have an out of memory exception. We would not be able to allocate a new exception in that case. This doesn't mean that all functions / methods that throw exceptions need to necessarily throw this one.

Q. So how might we implement this thing in moto

A. The compiler portion is pretty straight forward. The C code generated will be just how we write it using the TRY, CATCH, FINALLY macros. The interpreter is tougher. We can use the macros TRY, CATCH_ALL, and FINALLY inside the motoi_try() funtion as follows

```
TRY
        motoi(try block statements)
CATCH_ALL
        for each catch block
                if the exception caught matches that which should be caught by this block
                        execute catch block
FINALLY
        execute finally block if present
```

Q. What happens in Java when you return through a try block that contains a finally block

A. The finally block gets executed

Q. What happens in Java when you break or continue through a try block ?

A. The finally block gets executed

Q. Ok, so when we return something from within a try block in Java the finally block still gets executed ... What happens if we return something different from within that finally block ?

A. The second return wins

Q. What the hell is happenning in Java ?

A. It seems as though break, continue, and return are nothing more than exceptions that get caught differently.

Q. Ok, how the hell are we going to implement these semantics for return, break, and continue ?

A. See the ***discussion on this subject***

Q. Should we allow for a try block without a corresponding catch or finally block ?

A. Java doesn't and I can't figure out any good reason for this

Q. In the interpreter, will we need to add special handling for break or return statements that get executed from withing a try block since they also longjmp ?

Q. How do we implent 'catch all' in moto,

A. We should allow for catch(Exception e) like Java. We must also make sure this catch all re-throws MotoExceptions. ***Since moto lacks a general inheritance mechanism at this time we could fake this the same way we fake Object inheritance.***

Q. Ok, so if we are going to have a generic Exception Object we need some way of telling moto that all the other Exceptions are subclasses of that Object. How do we do this

A. Since we will be pulling in exceptions from mxc compiled files there should be a way of saying one class is a subclass of another class. Also, the ***functions that act on exceptions must be at least capable of acting on these exception subclasses***. Thus the storage for the portions of the subclasses that are common must have the same locations in the overall structure.

Q. If Exceptions are Objects in moto, must they be Objects in C as well ?

A. ***Yes***. Especially if we are going to bring in all the existing exceptions through mxc. This will require changes to the way exceptions are defined, thrown, and matched. *See the discussion about* ***Re-Implementing Exceptions As Objects***

Q. In compiled code, if we return from a function from within a try block, does the finally block get executed? Should it ?

A. If it should we may need to re-define return or provide a special RETURN macro

Q. How many nested try blocks can we have ?

A. Currently EXCP_SIZE (128)

Q. What does Exception.toString() do in Java ?

A. "*<fully qualified class name>* : *<message>*"

<u>Design Decisions</u>

<u>Implementing Finally</u>

Implementation of Finally will take some work. What we want to see happen is that Finally ***always*** gets executed. There are really five distinct cases to consider

1) No exception gets thrown
   - *try block executes*
   - *finally block executes*
2) An exception gets thrown in the try block but is not caught
   - *try block executes*
   - *finally block executes*
   - *exception gets re-thrown*
3) An exception gets thrown in the try block and is caught in the catch block and dealt with
   - *try block executes*
   - *catch block executes*
   - *finally block executes*
4) An exception gets thrown in the try block and is caught but another exception gets thrown in the block that catches it!
   - *try block executes*
   - *catch block executes*
   - *finally block executes*
   - *new exception gets thrown*
5) An exception gets thrown in the finally block
   - *try block executes*
   - *catch block executes*
   - *finally block executes*
   - *new exception gets thrown*

Cases 1-3 would be handled simply by making finally a no op which appears in the code after all the catches. The trouble occurs when an exception is thrown from within a Catch block since that cause a lngjmp the whole try block frame

What we want to see happen for case 4 is the finally block get executed and the exception get re-thrown.

We need to longjmp a second time but make sure none of the catches get executed this time around! So right now excp_catch sets the current tryblock scope to catch scope. ***Scope is really a misnomer, what we are talking about here is state this should be change.*** excp_throw uses this field to say 'If I'm already in a catch scope than decrement the tryblock count and pass the exception to the parent tryblock'. What we'd like to see happen instead is

1) If we are already in a catch state than excp_catch should not try to match exceptions
2) excp_catch and catchAll should only take us into catch scope when they have actually caught (matched) an exception
3) excp_throw should NOT try to decrement the TRYBLOCK. Instead it should jump straight on back to the same setjmp.

excp_catch, throw, catchAll, etc.. are misnomers since they work on the try block stack and are not methods of exception, these should be changed to extb_ methods.

<u>Re-implementation of Try Block States to deal with FINALLY</u>

TRYBLOCK is the current try block

Upon entering a try block TRYBLOCK should have its scope set to TRY_STATE its code set to 0 and its exception set to NULL

A catch block should only be entered if the current Try Block state is TRY_STATE

Upon entering a catch block TRYBLOCK should have its scope set to CATCH_STATE

The finally block should only be entered if the current try block state is TRY_STATE or CATCH_STATE

Upon entering a finally block TRYBLOCK should have its scope set to FINALLY_STATE

When an exception gets thrown TRYBLOCK should have its excp field set to that exception and its code set to 1

If at the end of a try block the code is still 1 meaning an exception was thrown but not caught the try block stack should be popped and the exception re-thrown

Re-Implementing Break Return and Continue with Java Semantics

Ok, so we need all **breaks**, **returns** and **continues** to pass through any active finally blocks

This shouldn't be too hard in the interpreter if we simply convert break, return, and continue to exceptions **MotoBreakException, MotoReturnException, and MotoContinueException** that cannot be caught by motoi_excp_catch

This means we can get rid of returnjmpstack from moto_env

And with some fairly minor changes to switch and case we can do the same to jmpstack

The Compiled code changes are tougher

break and continue currently just become break and continue in C . We could devise a BREAK and CONTINUE macro that throws a BreakException and a ContinueException that when END_TRY see's it converts to good old break and continue but does not re-throw. There are complications here however since try blocks can themselves be nested in other try blocks in the same frame. We really have two implementation options here

1) Add some sort of counter to exceptions as to how many try blocks they should jump back through if not caught ... this would be a big friggen mess

2) When generating loop code, generate a try block that catches **MotoBreakException** and **MotoContinueException** and does the right thing for both of them, then **have break and continue actually become THROW statements for these two exceptions** ... this can not be your run-of-the-mill try block however since the C functions break and continue will not let END_TRY execute. We will need to have code which does when END_TRY does (in terms of popping the block stack) before calling break; or continue

We are going to go with **option 2.** The following C code should do the trick

```
while / for(...) {
        TRY {
                ...
        } CATCH (“MotoBreakException”) {
                excp_try_i--;
                break;
        } CATCH (“MotoContinueException”) {
                excp_try_i--;
                continue;
        } END_TRY
}
```

Return from function is even more curious. I just converts to ‘return’ or ‘return s’ in C . It seems like what we will need to do is to wrap every generated function in a try block which catches MotoReturnException. $return should convert to something which sets an rvalue variable and then throws a MotoReturnException. We must also make sure that the last thing done in the moto function try block is to throw this same exception because we cannot have the return occur in the FINALLY block since if an exception gets thrown from within the function the finally block will still get executed.

Hence we should generate the following C code for moto functions and methods :

```
<type> fn (...) {
        <type> MOTO_RValue;
        TRY {
                ...
                MOTO_RValue = <return value>;
                THROW_D(“MotoReturnException”)
        } CATCH (“MotoReturnException”) {
                excp_try_i--;
                return MOTO_RValue;
        } END_TRY
}
```

Continue, Break, and Return Exceptions won’t fly! Read on to find out why !

Once more into the breach

While the above design works it has awful performance implications as seen here running the performance test *methcall*:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 40.9       0.45     0.45                             _sigprocmask [1]
  7.3       0.53     0.08                             __setjmp [5]
  5.5       0.59     0.06                             __longjmp [6]
  5.5       0.65     0.06                             _excp_getFillAndThrow
[7]
  5.5       0.71     0.06                             _strcpy [8]
  4.5       0.76     0.05                             _extb_catch [9]
  4.5       0.81     0.05                             _vfprintf [10]
  2.7       0.84     0.03   100000     0.00     0.00  __NthToggle_activate__
[11]
```

Its actually even worse than it looks (performance cost increased more than 10 fold), the cost of setjmp and longjmp are way way high on some systems ... specifically Mac OS X.

Now setjmp and longjmp are for non-local gotos. It seems like what we need here are local go-tos. Returning to the idea of counting ... we could implement a system like :

```
for
        ...
        try
                ...
                /* break *. goto break_label
        finally
                break_label :

                ...
                if coming from break or continue :
                        excp_block --
                        break;
        end_try
```

or

```
for {
        ...
        try {
                ...
                try {
                        ...
                        /* on break */ goto break_label_1
                } finally {
                        break_label_1:

                        /* on break */ goto break_label_2

                        if coming from break or continue
                                goto break_label_2:
                } end_try
        } finally {
                break_label_2 :
                ...
                /* on break */  excp_block --; break;

                if coming from break or continue :
                        excp_block --; break;
        } end_try
}
```

For return we would need

```
fn
        /* here return would generate */
        return rvalue;
```

```
try {
        /* here return would generate */
        rvalue = ...;
        goto return_label_1:
} finally {
        return_lable_1 :
        ...

        /* here return would generate */
        rvalue = ...;
                excp_block --; return rvalue;

        if coming from return
                excp_block --; return rvalue;
} end_try
```

So a possible algorithm would go like this :

open_try's is the number of open try blocks on the frame stack before the first fn frame

motoc_return could be changed to generate

```
if open_try's is == 0
      generate "return expression"
if open_try's is > 0 and we are not in a finally block
      generate "rvalue = expression; MOTO_FNAction=RETURN; goto
fnreturn_start_try_id";
if open_try;s is > 0 and we are in a finally block
      generate "rvalue = expression; MOTO_FNAction=RETURN; goto
fnreturn_finished_try_id";
```

motoc_try could be changed to generate

```
FINALLY {
      fnreturn_start_try_id :

      if (MOTO_Action == RETURN) {
            extb_block --;
            fnreturn_finished_try_id :
                  goto parent fn_return_start or if there isn't one
                  return rvalue;
      }
} END_TRY
```

To do this we would need a loop_parent_stack, a function_parent_stack, a try_state_stack, and a try_counter int.

each element in the loop_parent_stack would be a intStack of the tryBlock ids occurring within the most recently openned loop

each element in the function_parent_stack would be a intHashtable of the tryBlock ids occurring within the most recently openned function or method definition

Elements get pushed and pop from the loop_parent_stack when we enter or leave a while loop

Elements get pushed and pop from the function_parent_stack when we enter or leave a definition

When we enter a try block
1) the try_counter gets incremented
2) its value gets pushed on the current loop_parent_stack and function_parent_stack
3) 'TRY' gets pushed onto the try_state_stack

When we enter a finally block
4) The 'TRY' gets replaced with 'FINALLY' on the try_state_stack

When we leave the try block altogether
1) The loop_parent_stack, fn_parent_stack, and try_state_stack all get popped

<u>But what if we could throw locally ?</u>

A more optimal implementation of the above algorithm would be one that needed to keep track only of **one scope id stack** and could use a global **action** variable.

The **action** would get set whenever we want to do a **local throw** like for a break, continue, or return and unset **whenever the local jump is handled** or **when we do a non-local jump** like throw

try scopes, catch scopes, finally scopes, loop scopes and definition scopes all **get assigned a unique scope id** and pushed onto, and popped from, the **scope stack** which keeps track of the parent child relationships between scopes.

At the end of a **tryblock**, **catchblock**, and **finally block** we add do nothing labels **scope_end_i** where i is the id of the scope. **After the finally block if we are not handling an actual exception but instead have an active action, pop the tryblock stack and goto the end of the parent scope.**

At the end of a loopblock we nest the **scope_end_i** label inside of an unreachable block and handle break and continue actions, **re-throwing** (i.e. going to end of parent scope) return actions.

At the end of a definition block add a label to **scope_end_i** and handle return actions

Whenever we want to do a local jump, we
1) **set the action variable**
2) **mark any active exception as handled**
3) **throw local** e.g. **goto the label ending the current scope**

Here is the example generated C code for a while loop with a single try block inside

```
for(i=0;i<10;i++){
   TRY{
      printf("### In TRY block on iteration %d\n",i);
```

```
      if (i<2) SCOPE_CONTINUE(1)
      else SCOPE_BREAK(1)

      SCOPE_END_1: ;
   }CATCH_ALL{

      SCOPE_END_2: ;
   }FINALLY {
      printf("### In FINALLY block on iteration %d\n",i);

      SCOPE_END_3: ;
   }END_TRY_ACTIONS{
      printf("### In END TRY ACTIONS %d WITH ACTION %d\n",i,_MOTO_Action_1);
      if(_MOTO_Action_1 != NOOP_ACTION) {excp_try_i--; goto SCOPE_END_0;}
   }END_TRY

   if(0) {
      SCOPE_END_0:
      printf("### In END WHILE ACTIONS %d WITH ACTION %d\n",i,_MOTO_Action_1);
      if(_MOTO_Action_1 == BREAK_ACTION) { _MOTO_ACTION == NOOP_ACTION; break;
}
      if(_MOTO_Action_1 == CONTINUE_ACTION) { _MOTO_ACTION == NOOP_ACTION;
continue; }
   }
}
```

How do we get operator caused Exceptions to be thrown from Moto line numbers

There are a couple places where this is going to pop up

1) NullPointerExceptions caused by dereferencing null

```
"((%s)(excp_file=__FILE__,excp_line=__LINE__,inline_checkForNullDereference(%s
)))->%s"

void * eCheckNull(void* p,int line, char* file){
      if (p==NULL){
            excp_file = file;
            excp_line = line;
            getFillAndThrowDefault(“NullPointerException”)
      }
      return p;
}
```

2) NullPointerExceptions caused by calling a method on null

```
(<type>)eCheckNull(<code>,__LINE__,__FILE__)

void * eCheckNull(void* p,int line, char* file){
      if (p==NULL){
            excp_file = file;
            excp_line = line;
```

```
            getFillAndThrowDefault("NullPointerException")
      }
      return p;
}
```

3) NullPointerExceptions caused by subscripting null

```
eISub (<code for var>,<code for index>,int line, char* file)
eLSub (<code for var>,<code for index>,int line, char* file)
eFSub (<code for var>,<code for index>,int line, char* file)
eDSub (<code for var>,<code for index>,int line, char* file)
eBSub (<code for var>,<code for index>,int line, char* file)
eCSub (<code for var>,<code for index>,int line, char* file)
eYSub (<code for var>,<code for index>,int line, char* file)
```

4) ArrayBoundsExceptions caused by subscripting out of bounds

```
eISub (<code for var>,<code for index>,int line, char* file)
eLSub (<code for var>,<code for index>,int line, char* file)
eFSub (<code for var>,<code for index>,int line, char* file)
eDSub (<code for var>,<code for index>,int line, char* file)
eBSub (<code for var>,<code for index>,int line, char* file)
eCSub (<code for var>,<code for index>,int line, char* file)
eYSub (<code for var>,<code for index>,int line, char* file)
```

5) MathExceptions caused by division by zero

```
int32_t eCheckIntZero(int32_t n,int line, char* file)
float eCheckFloatZero(float n,int line, char* file)
int64_t eCheckLongZero(int64_t n,int line, char* file)
double eCheckDoubleZero(double n,int line, char* file)
char eCheckCharZero(char n,int line, char* file)
char eCheckByteZero(char n,int line, char* file)
```

5) Regex match exceptions cause by either a null regex or a null string

<u>Re-Implementing Exceptions As Objects</u>

If Exceptions are really going to be implemented as objects we will need to do the following

1) Have the **excp_define** macro typedef a new struct for each exception and create default constructors so that there is a C structure (which will become a moto object) for this exception.

1.1) This may be nothing more than typedeffing the exception name to **Exception** however if we do this (which we pretty much have to for current mxc compatibility) we will need to do something about how e_types get defined so we don't get collission between the e_type variable and the structure name . **What we should probably do is move all existing Exception type names to <*Exception*>Type** e.g.

<u>in .c file:</u>

```
EType AllocationFailureExceptionType = {&AllocationFailureExceptionType,
```

```
"AllocationFailureExceptionType" };

AllocationFailureException* AllocationFailureException_create(){
      AllocationFailureException* e =
malloc(sizeof(AllocationFailureException));
      e->type = AllocationFailureExceptionType;
      e->msg = NULL;
}
AllocationFailureException* AllocationFailureException_create(char* msg){
      AllocationFailureException* e =
malloc(sizeof(AllocationFailureException));
      e->type = AllocationFailureExceptionType;
      e->msg = msg;
}
```

in .h file :

```
typedef Exception AllocationFailureException;
extern EType AllocationFailureExceptionType;
AllocationFailureException* AllocationFailureException_create();
AllocationFailureException* AllocationFailureException_create(char* msg);
```

To do the necessary concatenation with a CPP macro to get the EType name use the ## operator ... I have no idea how standard this is though but I'm probably already wed to gcc.

This strategy causes some real unfortunate compilations for the THROW macro. It actually doesn't take arguments (nor can it) but rather passes the args right on to **excp_marshal** and **excp_marshalDefault** . excp_marshal has a variable argument list (which is why THROW can't take args) . **This means that everywhere a current exception is being thrown we need to replace <Exception name> with <Exception name>Type ... yuck.**

1.2) **Or** we could start to untie moto types from C types ... but that would require much more work.

1.3) **Or we could change exception types to Strings** . Eventually we will want to let people define exceptions from within Moto. When this happens all moto objects must effectively know their own ***type*** and that *type* is what must be compared in try catch (as opposed to the symbol address of the EType variable being used now). There are a number of issues with the EType variable. First off the address is a void * yet lngjmp returns an int (although its really not clear to me why we compare the address returned by the lngjmp as opposed to the stored exception).

1.3.1) The first implication of this would be that **exception types become / contain strings of some kind**. This is essentially what we had planned for all types down the road. This will give us instight as to how it should occur.

1.3.2) The second implication of this is that catch will work essentially **by comparing Strings for equality**. To get away from comparing the return value of the lngjmp, throw should always set that return value to 1 on exception and set the tryblock exception to the current exception as it does currently. Then the catch function should compare the current tryblock type to the current exception type based on String equality. On a match it should mark the exception as handled by setting the tryblock code back to 1.

1.3.3) A hidden implication of this strategy is that while the strategy in 1.1 required changes to all code using the THROW macro that could not be accomplished

easily programatically, by moving to String types THROW, THROW_D, and CATCH could all effectively work on Strings. This change could be made in all C code programatically by the following perl script:

```
#!/usr/bin/perl
open(INPUT, "$ARGV[0]");
open(OUTPUT, ">$ARGV[0].new");

foreach (<INPUT>) {
   $_ =~ s/(THROW(_D)?[(]|CATCH[(])([A-Za-z]+)/$1\"$3\"/g;
   print OUTPUT $_;
}

close(INPUT);
close(OUTPUT);
```

1.3.4) The macro code to implement this strategy would look like :

```
/*
 * macro for throwing exceptions
 */
#define THROW \
   excp_file=__FILE__,\
   excp_line=__LINE__,\
   excp_getFillAndThrow

/*
 * macro for constructing and throwing exceptions without messages
 */
#define THROW_D \
   excp_file=__FILE__,\
   excp_line=__LINE__,\
   excp_getFillAndThrowDefault

/*
 * macro for catching exceptions
 */
#define CATCH(etype) else if (excp_catch(tryblk, etype))
```

1.3.5) The C code for this strategy would look like

```
void excp_getFillAndThrow( char* etype , char*s, ...){
   Exception* e = excp_get();
   int maxchars = MAX_EMSG_SIZE - 6;
   int nchars;

   va_list ap;
   va_start(ap, s);

   e->file = excp_file;
   e->line = excp_line;

   strcpy(e->t,etype);
```

```
   nchars = vsnprintf(excp_e.msg, maxchars, s, ap);
   if (nchars < 0 || nchars >= maxchars) {
      strcat(e->msg, " ...\n");
   }

   excp_throw(&TRYBLK,e);
}

void excp_getFillAndThrowDefault( char* etype ){
   excp_getFillAndThrow( etype , "", "");
}

/*
 * accessor for the current exception (for use in a catch block)
 */
Exception *excp_get() {
   return &excp_e;
}
```

1.3.6) The downside of using Strings in the C macros THROW and CATCH is **we don't have type safety in our C code**.

2) Since people are going to be writing **new Exception()** we should not use the THROW macro or excp_create functions. Instead we should create (*allocate*) a new Exception object and do what excp_create does to it when it gets thrown.

2.1) etype should be set upon construction (in default constructors created by excp_define)
2.2) messages should be set in either the constructor or by a method (setMessage() getMessage())
2.3) file and line should be set upon throw

2.4) **excp_create()** is really a misnomer when compared to other xxx_create() functions because it doesn't actually instantiate anything, rather it just fills in the process's static exception structure. A better model would be to pass it the address of an exception Object and have it fill that in. It should be changed to **excp_fill** and work on generic exception objects. Really what excp_fill is is a default initializer.

Exception Macro Changes

```
/*
 * Macro for defining new exception types in .c files
 */
#define excp_declare(e_type) \
      \
      EType e_type ## Type = {&(e_type ## Type), #e_type }; \
      \
      e_type * e_type ## _createDefault(){ \
            e_type * e = (e_type * )malloc(sizeof(e_type)); \
            e->t = e_type ## Type; \
            return e; \
      } \
      \
```

```
      e_type * e_type ## _create(char* msg){ \
            e_type * e = (e_type * )malloc(sizeof(e_type)); \
            e->t = e_type ## Type; \
            return e; \
      }

/*
 * Macro for externing exception definitions in .h files
 */

#define excp_extern(e_type) \
      typedef Exception e_type; \
      extern EType e_type ## Type; \
      e_type* e_type ## _create(); \
      e_type* e_type ## _create(char* msg);

/*
 * Macro for catching exceptions
 */
#define CATCH(etype) else if (excp_catch(tryblk, etype ## Type))

/*
 * macro for executing code both on success and failure
 */
#define FINALLY /* finally block */
```

Lex Changes

EXCP_TRY - $try
EXCP_CATCH - $catch(
EXCP_FINALLY - $finally
EXCP_ENDTRY - $endtry
EXCP_THROW - $throw

EXCP_ETRY - try
EXCP_ECATCH - catch
EXCP_EFINALLY - finally
EXCP_ETHROW - throw

Yacc Changes

```
try_block
      :  EXCP_TRY  statement_list  EXCP_ENDTRY
      |  EXCP_TRY  error  EXCP_ENDTRY
;

try_handlers
      :  catch_block_list
      |  catch_block_list  finally_block
      |  finally_block
;

catch_block_list
```

```
      :  catch_block
      |  catch_block_list  catch_block
;

catch_block
      :  EXCP_CATCH  NAME  NAME  CLOSEPAREN  statement_list
      |  EXCP_CATCH  NAME  NAME  CLOSEPAREN
      |  EXCP_CATCH  error  CLOSEPAREN
;

finally_block
      :  EXCP_FINALLY  statement_list
      |  EXCP_FINALLY
;

throw_statement
      :  EXCP_THROW  expression  CLOSEPAREN
      |  EXCP_THROW  error  CLOSEPAREN
;

statement
      : conditional_statement
      | iterative_statement
      | declare_statement
      | definition_statement
      | try_block
      | do_statement
      | jump_statement
      | use_statement
      | print_statement
      | throw_statement
      | return_statement
      | DOLLAROPEN embedded_statement_list CLOSEDOLLAR
      | DOLLAROPEN CLOSEDOLLAR
      | DOLLAROPEN error CLOSEDOLLAR
      | ANYCHARS
      | WS
      | error
;

embedded_try_catch_block
      :  EXCP_ETRY  embedded_block  embedded_try_catch_handlers
;

embedded_try_catch_handlers
      :  embedded_catch_block_list  embedded_finally_block
      |  embedded_finally_block
      |  embedded_catch_block_list
;

embedded_catch_block_list
      :  embedded_catch_block
      |  embedded_catch_block_list  embedded_catch_block
;
```

```
embedded_catch_block
      :  EXCP_ECATCH  OPENPAREN  NAME  NAME  CLOSEPAREN  embedded_block
      |  EXCP_ECATCH  OPENPAREN  error  CLOSEPAREN
;

embedded_finally_block
      :  EXCP_EFINALLY  embedded_block
      |  EXCP_EFINALLY
;

embedded_throw_statement
      :  EXCP_ETHROW  expression
      |  EXCP_ETHROW  error
;

embedded_statement
      : embedded_conditional_statement
      | embedded_iterative_statement
      | embedded_definition_statement
      | embedded_class_definition_statement
      | embedded_try_block
      | embedded_declare_statement SC
      | embedded_do_statement SC
      | embedded_jump_statement SC
      | embedded_use_statement SC
      | embedded_print_statement SC
      | embedded_return_statement SC
      | embedded_throw_statement SC
      | embedded_block
      | error SC
      | SC
;
```

New Non-terminals

try_block
try_handlers
catch_block_list
catch_block
finally_block
throw_statement
embedded_try_block
embedded_catch_block_list
embedded_catch_block
embedded_finally_block
embedded_throw_statement

New Parser Scoping Tokens

EXCP_TRY
EXCP_ETRY

New Op Types

EXCP_TRY(LIST try_block,LIST catch_blocks, LIST finally_block)
EXCP_CATCH(NAME excp_type, NAME excp_var_name, LIST catch_block)

New Parse errors

void moto_malformedCatch(MetaInfo* meta, char embedded) -
"MalformedCatch" - "Malformed catch statement, expected $catch(<type> <name>)"
"MalformedEmbCatch" - "Malformed catch, expected catch(<exception type> <name>) {...}\n"
void moto_malformedThrow(MetaInfo* meta, char embedded) -
"MalformedThrow" - "malformed throw statement, expected $throw(<excpression)"
"MalformedEmbThrow" - "Malformed throw, expected throw <expression>\n"
void moto_unterminatedTry(MetaInfo* meta, char embedded) -
"UnterminatedTry" - "unterminated $try block"
"EmbUnterminatedTry" - "unterminated try block"
void moto_endTryWithoutTry(MetaInfo* meta) -
"EndTryWithoutTry" - "$endtry without $try"
void moto_illegalCatch(MetaInfo* meta, char embedded) -
"IllegalCatch" - "$catch not within $try ... $endtry\n"
"EmbIllegalCatch" - "catch not within try block\n"
void moto_illegalFinally(MetaInfo* meta, char embedded) -
"IllegalFinally" - "$finally not within $try ... $endtry\n"
"EmbIllegalFinally" - "finally not within try block"
void moto_illegalTry(MetaInfo* meta, char embedded) -
"IllegalTry" - "$try without $catch or $finally \n"
"EmbIllegalTryy" - "try block without catch or finally blocks"

environment changes

need new built in type Exception

new type verifier errors

void moto_typeNotCatchable(char* typename);
TypeNotCatchable - Cannot catch <%s>. Only exceptions may be caught

void moto_typeNotThrowable(char* typename);
TypeNotThrowable - Cannot throw <%s>. Only exceptions may be thrown

void moto_typeAlreadyCaught(char* typename);
TypeAlreadyCaught - The exception of type <%s> will have already been caught by a prior catch

Verifier Changes

motov_excp_try
- Push on a new sub-frame
- Verify the try block code
- Pop the sub-frame
- For each catch block

- Verify the exception type exists
    - If not throw a type not defined err
- Verify the type is a subclass of Exception
    - If not throw a type not exception err
- if the type is Exception than consider all other exceptions as already being caught
- add the exception type to a handled exceptions stringset
    - if it's already present throw a typeAlreadyCaught err
- push on a new frame
- Declare the exception var
- verify the catch block code
- pop off the frame

- push frame
- Verify the finally block code
- pop frame

motov_excp_throw

- Evaluate the expression operand
- Verify that what's being thrown is and exception (for now a reference type is fine)
    - If it is not a subclass of Exception throw a type not throwable error

<u>Interpreter Changes</u>

motoi_excp_throw

- execute the expression being thrown
- set the line number and file for the exception object
- call excp_throw on the result

motoi_excp_try

- Open TRY block
- push frame
- record the current frame
- execute try block code
- pop frame
- CATCH_ALL_BUT("Moto")
    - pop off all frames from the marked frame (just like with return)
    - Get the type of the caught exception
    - If it was any sort of MotoException that was thrown don't even try to catch it
    - Record the fact that this exception has not yet been handled
    - for each catch block
        - if catch block exception type matches caught exception type
            - push frame
            - define exception variable, set its value to that of caught exception
            - execute catch block code
            - pop frame
            - Record the fact that this exception was handled
    - If the exception was not handled, re-throw it
- FINALLY
    - push frame
    - execute finally block code
    - pop frame

- modify motoi_while to catch MotoBreakException and MotoContinueException
- modify motoi_for to catch MotoBreakException and MotoContinueException
- modify motoi_jump to throw either MotoBreakException or MotoContinueException as appropriate
- modify motoi_return to throw MotoReturnException
- modify motoi_fn to catch MotoReturnExceptions
- modify motoi_new to catch MotoReturnExceptions

Compiler Changes

motoc_excp_throw
- evalute the expression being thrown
- output excp_marshal(expression,__FILE__,__LINE__) on the result

motoc_excp_try
- Output “TRY { “
- Push on a new sub-frame
- Evaluate the try block code
- Pop the sub-frame
- Output the try block code
- Output “ } “
- For each catch block
- Output 'CATCH("<Exception Type>") {' or 'CATCH_ALL_BUT(\”Moto\”) {' if the exception type
is 'Exception'
- push on a new frame
- Declare the exception var
- Output an assignment of the current exception to this var
- Evaluate the catch block code
- pop off the frame
- output the catch block code
- output ‘}’
- push frame
- Verify the finally block code
- pop frame

modify motoc_while / motoc_for to generate the following C code

```
while / for(...) {
        TRY {
                ...
        } CATCH (“MotoBreakException”) {
                excp_try_i--;
                break;
        } CATCH (“MotoContinueException”) {
                excp_try_i--;
                continue;
        } END_TRY
}

fn(...) {
        <type> MOTO_RValue;
        TRY {
```

```
        ...
        MOTO_RValue = <return value>;
        THROW_D("MotoReturnException")
    } CATCH ("MotoReturnException") {
        excp_try_i--;
        return MOTO_RValue;
    } END_TRY
}
```