Design Documentation for Functions in Moto

Overview

We want to add support for defining **functions** in moto. We want the code in functions to have access only to locally defined variables, the arguments passed in, and external variables declared as **global**. We want to be able to define functions either inside of embedded blocks using a C/ Java like syntax or by way of the moto construct $define. **return** should also be available either as an embedded block keyword or as the moto construct **$define**.

Design Decisions

Functions should **not** have to be defined before they are used. It's just annoying to have to do that.

Nesting one function definition inside another will **not** be allowed.

The scope of local variables declared inside of functions **should** only be within the function definition.

Function definition anywhere outside of the main frame **should be prohibited**

Using an extension library from anywhere outside of the main frame **should be prohibited**

Declaring a global variable anywhere outside of the main frame **should be prohibited**

Variable 'shadowing' **should not be prohibited**.

Design Implications

$return must be nested within $define ... $endef and return an expression of type TYPE
return must be nested within TYPE NAME((TYPE_i NAME_i,)*) { ... } and return an expression of type TYPE

Need to add / modify the following parser rules

```
declaration_list : declaration
                 | declaration list, declaration

declaration      : TYPE NAME
                 | type_qualifier TYPE NAME

type_qualifier   : "global"

definition_stmt  : "$define" "(" declaration "(" declaration_list ")" ")"
statement_list "$enddef"

embedded_definition_stmt : declaration "(" declaration_list ")"
embedded_statement

return_stmt : "$return" "(" expression ")"
```

```
embedded_return_stmt : “return” expression

declare_stmt : “$declare” “(“ declaration “)”
             | “$declare” “(“ declaration “=” assignment_expression “)”

embedded_declare_stmt : declaration
                      | declaration “=” assignment_expression
```

<u>Need to add the following new tokens to the lexer</u>

DEFINE	$define(
ENDDEF	$enddef
RETURN	$return(
ERETURN	return
GLOBAL	global

<u>And the following new nonterminals to the parser</u>

definition_statement - so I can push and pop syntactic scope
function_definition_statement
embedded_definition_statement
embedded_function_definition_statement
return_statement
embedded_return_statement

declaration_list
declaration - type_qualifier TYPE NAME that can be used by both declare statements and definition statements
type_qualifier

<u>And phantom tokens that can be pushed on to the scope stack</u>

DECLARATION	for argument declarations to functions (NAME NAME)
EDEFINE	so I have a scope to push

<u>Parsing will require new parse errors:</u>

```
void moto_illegalReturn(MetaInfo* meta,char embedded);
void moto_illegalDefine(MetaInfo* meta,char embedded);
void moto_illegalUse(MetaInfo* meta,char embedded);
void moto_illegalGlobal(MetaInfo* meta);
void moto_malformedReturn(MetaInfo* meta,char embedded);
void moto_malformedDefine(MetaInfo* meta,char embedded);
void moto_unterminatedDefinition(MetaInfo* meta,char embedded);
void moto_endDefWithoutDefine(MetaInfo* meta);
```

<u>Requiring new error strings:</u>

"IllegalDefine" - function definition within sub-cope
"IllegalUse" - use within sub-cope
"IllegalGlobal" - global within sub-cope

"IllegalReturn" - return not within function definition
"EmbIllegalReturn" - $return not within function defintion
"EmbIllegalDefine" - $define within function definition
"EmbIllegalUse" - $use within sub-scope
"MalformedDefine" - Malformed $define statement, expected $(<type> <name>(<args>))
"MalformedReturn" - Malformed $return statement, expected $return(<expr>)
“MalformedEmbDefine” - Malformed function prototype, expected <type> <name>(<args>)
"MalformedEmbReturn" - Malformed return statement, expected return <expr>;
"UnterminatedDefinition" - Unterminated $define statement
"EmbUnterminatedDefinition" - Unterminated function definition
"EndDefWithoutDefine" - $enddef without $define

Changes needed to mxfunctions

To determine if this function is an extension function or locally defined we check if f->fptr is null. If it is than this function must be locally defined

We need to add a f->opcell field and point it to the opcell containing the function definition (not just the statement list because we need to know what variables to set for the arguments)

We need to set f->opcell to null in loaded functions in dl.c

For convenience we should also add an argnames array to mxfunctions. This way motoi doens’t need to reparse the opcell to know what variables to set for the arguments.

Verifier and Interpreter Implications

Need to create a new global scope which all other scopes can look for variables in. This should be done by adding a new symbol table to env (env->globals). When searching for variables, code in all frames should have access to env->globals.

Need tag a frame as ‘private’ (FN_FRAME). When searching for variable declarations up the scope stack we should not go beyond private frames because private frames (frames used in function definitions) should not inherit variable declarations from parent scopes

Need to be able to register function prototypes as they are defined. We need to add a pass in between parsing and motov (motod). I do not want to require explicit ‘function prototypes. So instead we will need to add a pass after syntax checking but before type checking to grab all the function prototypes and make MXFunctions out of them (might as well do this for $uses also)

I will put the opcell corresponding to the define in some sort of mapping table and every time the function is called from the interpreter motoi() that opcell. This is likely what has to happen. Motod should be responsible for this so that we can call functions used before they are defined.

Changes needed to moto frames

Need to add a new frame type FN_FRAME

Need to change moto_createFrame to create frames of this type

Need to change moto_freeFrame to free frames of this type

Need to modify moto_getVal to not descend past FN_FRAMEs when looking for variables. Need to make it search env->globals also. It should be looked at last to allow for shadowing.

Need to modify moto_getVar to not descend past FN_FRAMEs when looking for variables. Need to make it search env->globals also. It should be looked at last to allow for shadowing.

<u>Need to add new verifier errors</u>

void moto_varTypeReturnError(char *t1, char *t2)

and corresponding explanation

"VarTypeReturnError" - Return type <%s> cannot be implicitly cast to type <%s>

<u>Standard moto ivc implications</u>

Need to switch on the following new operands (use motoi as example for all 3)

DEFINE -> motoi_define
RETURN -> motoi_return

need to add the following new functions

void motoi_define(const UnionCell *p)
void motoi_return(const UnionCell *p)

psuedocode of motod_define

```
1) extract the type and function name
2) if the type is not defined generate an error
3) extract the arguments
4) for each type that's not defined generate an error
4) generate an mxfunction record and put in in the ftable
- 5) if the prototype is already declared generate an error
```

psuedocode of motov_define

```
1) extract the type and function name
2) if the type is not defined generate an error
3) extract the arguments
4) for each type that's not defined generate an error
- 5) if the prototype is already declared generate an error
6) create a private frame
7) push arguments onto that frame as motovars by declaring them
8) Declare the special rvalue argument
9) call motov on the statement list
10) pop off the private frame
```

psuedocode of motov_return

```
1) get the rvalue variable
2) verify the type of the argument to return matches the type of the rvalue variable.
3)  if not (and it's not castable) generate an error
- 4) push it on to the stack ?
```

psuedocode of modifications to motov_fn
- looks like we don't need any

psuedocode of motoi_define

```
1) don't do anything
```

psuedocode for motov_declare modifications

```
1) Instead of the typename and the varname being the first and second operands to the opcell, a new DECLARATION opcell will become the first operand and they must be extracted from that.
2) If the DECLARATION opcell specified that this is a global declaration then
        2.1) We should check whether another global variable with that name has already been declared by calling moto_getGlobalVar(env, varn);
        2.2) If not that var should be declared global by calling moto_declareGlobal(env, type, varn)
```

psuedocode for motoi_declare modifications

```
1) Instead of the typename and the varname being the first and second operands to the opcell, a new DECLARATION opcell will become the first operand and they must be extracted from that.
2) If the DECLARATION opcell specified that this is a global declaration then
        2.1) The var should be declared global by calling moto_declareGlobal(env, type, varn)
```

psuedocode for motoc_declare modifications

```
1) Instead of the typename and the varname being the first and second operands to the opcell, a new DECLARATION opcell will become the first operand and they must be extracted from that.
2) If the DECLARATION opcell specified that this is a global declaration then
        2.1) The var should be declared global by calling moto_declareGlobal(env, type, varn)
```

psuedocode of motoi_fn modifications

```
1) create a private frame for the execution
2) declare variables for each of the arguments setting their values to the values passed in
3) declare the special rvalue variable
4) setjmp for motoi_return to jump to
5) Call motoi on statement list of the opcell matching the function prototype
6) pop the private frame and push rvalue onto the stack
```

psuedocode of motoi_return

```
1) execute the expression operand
```

```
2) set value of rvalue
3) pop the return stack
4) lngjump back into motoi_fn
```

```
psuedocode of motoc_define
1) Create a 'definition' frame
2) Generate all the things we need to retrieve the MXFunction
     for this definition i.e. the fname, the argcount, and the
     argtypes array
3) While we're at it, push arguments onto the definition
     frame by declaring them
4) Get the function this definition corresponds to
5) Push on a scoping frame
6) generate code for a c function
7) Free the scoping frame
8) Free the definition frame
9) Map the function to it's definition in env->fdefs
10) Clear the fcodebuffer for the next guy
```

```
psuedocode for motoc_return
1) set code to “return expr;”
```

<u>env.c modifications</u>

moto_freeFrame

1) when freeing an FN_FRAME, if we are in compiler mode then instead of concatenating the frame.out to the parent frame.out the code should go into a hashtable in env mapping function prototypes to the code they generate (env->fdefs)

2) since freeFrame doesn't know what the current function is it should write out to a currentFunctionBuffer of some sort (env->fcodebuffer)

3) moto_freeFrame does need to know something about the current function tho so it doesn't try writing out arguments as declarations ...

- We can accomplish this in a number of ways.
    1) we can add a boolean to MotoVars to specify whether it's an argument variable or not
    2) we can associate the function with the frame and for each declaration, check to see if the var is listed as an argument. This may be problematic if someone specifically tries to shadow a var
    *** 3) we can have motoc immediately push a FN frame followed by a SUB frame on to the stack. Then make the rule that no declarations are written out for FN frames. This will allow people to shadow vars (they won't get multiply defined errors ... they maybe it would be nice if they could get warnings)**

modified function

MotoVar *moto_declare(MotoEnv *env, MotoType *type, char *name, char global);

new functions:

MotoVar *moto_getGlobalVar(MotoEnv *env, char *name);

char* moto_fnToCName(MotoEnv* env, char* fname, int argc,char** argtypes);
char* moto_fnToCPrototype(MotoEnv* env, MXFunction* f);

moto_fnToCName(MotoEnv* env, MXFunction *fn)

1) to work similar to `char *mfn_functionName(MFN *p)`

New moto_emitCXXX functions needed

1) moto_emitCPrototypes - emit C prototypes for moto functions
2) moto_emitCGlobals - emit C declarations of global varables
3) moto_emitCFunctions -emit C definitions for moto functions

void moto_emitCPrototypes(MotoEnv *env, StringBuffer *out);
void moto_emitCGlobals(MotoEnv *env, StringBuffer *out);
void moto_emitCFunctions(MotoEnv *env, StringBuffer *out);

char* moto_fnToCName(MotoEnv* env, char* fname, int argc,char** argtypes);
Called by motov to generate a mangled c name

char* moto_fnToCPrototype(MotoEnv* env, MXFunction* f);
Called by moto_emitCPrototypes and moto_emitCFunctions when outputting locall defined function prototypes and definitions

These will need to be called by moto_out() in motofn.c

For the compiler we need an env->fdefs symbol table could be used to map MXFunctions to the generated code for function bodies.

Two jmpstacks are be better than One

Right now env maintains one jumpstack. That's because break and continue always jump back to the most recently opened loop. We can't use that jump stack because we may be returning from within a loop and actually want to jump right past it

Implications:

Need a new jmpstack ... will call it returnjmpstack

C code generation

Variables declared as global should be declared outside the __MAIN__ function and **before** function definitions. They will need to be declared as static so that mmc doesn't give us symbol collision errors at compile time.

Functions should **NOT** map directly to C functions since C functions don't differentiate based upon the argument list. Rather moto function names should be mangled in the same way mxcg generated function names are mangled. Both of these mechanisms *should* likely be modified to generate symbol names in the same way C++ does.

All functions and variables declared global should actually be declared static in C to start with so operating systems like Mac OS X don't hand us a bunch of multiply defined symbol warnings when we compile an app with mmc

So that we don't get errors in our generated C code we should output function prototypes at the top of the file

For moto functions we could say f->fname = f->cname = the same mangleing done in mxc

e.g. something along the lines of

```
char *mfn_functionName(MFN *p) {
   if (p->cname == NULL) {
      int i;
      int argc = vec_size(p->argv);
      StringBuffer *buf = buf_create(32);
      buf_printf(buf, "__MOTO_%s__", p->fname);
      for (i = 0; i < argc; i++) {
         FNArg *arg = (FNArg *)vec_get(p->argv, i);
         if (i > 0) {
            buf_puts(buf, "_");
         }
         buf_puts(buf, arg->type);
      }
      buf_replaceChar(buf, ':', '_');
      buf_replaceChar(buf, '~', '_');
      p->cname = buf_toString(buf);
   }
   return p->cname;
}
```

To generate C prototypes and definitions we are going to need the argument names. These are currently not in the MXFunction struct so we must add them.

Other thoughts

How should moto programmers be able to use functions defined on one page from another (without using #include)

Should there be a new command line tool while allows the creation of extension libraries from moto files ?

ToDo List

- Update parser and lexer with required new terminals and rules (global,return , $return, $define,$enddef)
- Be sure to push DEFINE onto the scope stack in moto.y so we can verify we only return from within functions and also don't try to nest function definitions
- Add new optypes to motoi,motov, and motoc (DEFINE,RETURN)
- Add field to function record specifying if it came from an extension or a moto definition
- modify motoi_fn to deal with calls to functions defined in moto
- Add new hashtable in env.c to hold global variable declarations
- Add new hashtable in env.c to hold C function definitions (for motoc)
- Add field to scopes (a boolean that says whether this is a private or function scope or

whether it should inherit vars from parent scopes)

Examples

```
---

${
global int hello_count=0;

String counter(){
    hello_count++;
   return “hello “+str(hello_count);
}

print hello();
print hello();
print hello();
}$
```

```
--
```

1) The following file

```
${
void hello1() { print "hello world\n"; }
hello1();
}$

${
String hello2() { return "hello world\n"; }
print hello2();
}$

$define(String hello3())
    $return("hello world")
$enddef
$(hello3())

${
   String inputTag(String type, String name, String value){
      String tag = "<input type=\""+type+"\" name=\""+name+"\"
value=\""+value+"\">";
      return tag;
   }
}$
```

```
<form>
$(inputTag("text","foo","bar"))
$(inputTag("submit","op","bar"))
</form>
```

Should output

```
hello world

hello world

hello world

<form>
<input type="text" name="foo" value="bar">
<input type="submit" name="op" value="bar">
</form>
```

2) The following file

```
$define()              $* Malformed $define *$
$enddef                $* $enddef without $define *$
$return()              $* Malformed $return *$
$return(1)             $* $return w/out $define *$

$define(int foo())
   $define(int bar()) $* $define inside of a definition *$
   $enddef
$enddef

${ return 12 ; }$      $* return outside of a definition *$
${
   int foo(){
      int bar(){}      // nested definition
   }
}$

$define(int foo())     $* $define without enddef *$
```

Should result in the following syntax errors

```
fn_yacc_err.moto:1: Malformed $define statement, expected $(<type>
<name>(<args>))
fn_yacc_err.moto:2: $enddef without $define
fn_yacc_err.moto:3: Malformed $return statement, expected
```

```
$return(<expr>)
fn_yacc_err.moto:4: $return not within function defintion
fn_yacc_err.moto:7: $define within function definition
fn_yacc_err.moto:11: return not within function definition
fn_yacc_err.moto:14: function definition within function
definition
fn_yacc_err.moto:18: Unterminated $define statement
```

3) The following file

```
$define(int foo())
   $return('a')               $* Wrong type for return value *$
$enddef

$define(Stringle baz())       $* Type stringle not defined *$
$enddef

$define(String bar(Apple a))  $* Type Apple not defined *$
$enddef

$(foo('a'))                   $* foo(char) not defined *$
```

Should result in the following verification errors

```
fn_verr_err.moto:2: Return type <int> cannot be implicitly cast to
type <char>
fn_verr_err.moto:5: Type not defined: <Stringle>
fn_verr_err.moto:8: Type not defined: <Apple>
fn_verr_err.moto:11: No such function: foo(<char>)
```

If someone declares a variable outside of any nested scope and declares it global than it will become available inside of function definitions e.g.

```
$declare(global foo=”bar”)

${int printFoo() { print foo; } }$
```

should work fine whereas

```
$declare(foo=”bar”)

${int printFoo() { print foo; } }$
```

should cause a variable not defined before used error