Design Documentation for Classes in Moto

Overview

Classes in moto will allow for the logical grouping of data and methods acting on that data. Moto will allow for the definition of a Class. Declarations of variables within that definition will become Class member variables instantiated whenever an instance of the class is instantiated. Definitions of functions within the Class definition will become methods of that class. Moto code that neither defines a function nor declares a variable nor ends the class definition will be disallowed as it is meaningless. The new operator will be able to declare instances of Moto defined Classes just as it can with Classes defined in extensions. Functions defined within a Moto Class will have access to the special variable 'this' which has a substructure equivalent to the variables defined within the Class.

Design Decisions

We should allow for the access of instance variables external to the class definition. While this often allows for poor programming it will provide a measurable performance increase especially in the interpreter and will save a lot of brainless programming time. ***We might consider dissallowing this functionality for mLight.*** This will force use to use a different meaning for the dot '.' . The new meaning should be that of a two operand dereferencing operator.

Lexer / Parser

Moto code within a class definition that neither defines a function nor declares a variable nor ends the class definition will be disallowed as it is meaningless. This is the case in languages lie C++ an Java. It is debatable however as to what should happen to complex declarations inside of a Class e.g. a member variable is initialized to some value. It is concievable that the code for ininitialization could be copied into the constructor. ***In Java the code for complex declarations of member variables is copied into the beginning of the constructor. We should take the same tact.*** One might then wonder if all code should be copied into the constructor. This would mean that if we only want to allow method definitions and variable declarations at the top level of a Class Definition then perhaps this rule should be implemented in the lexer / parser.

One of the biggest issues here is what to do about the dot. It now must become a full fledged two operand operator (the dereference operator). It can act on any expression (that can return an object at least ... which because of String addition is pretty much any expression) and may be used on either side of an equals sign. ***In the compiled world dereferencing variables can be converted to '->' in C***.

The precendence of the dereferencing operator should be the same as . in Java which is hopefully the same as '->' in C. If it is not than we can change the precendence at code generation time with parens

The lexical scoping rules for functions will need to be changed so that they are allowed at the first level of a Class definition. This may in fact be another way to tell if they are functions or methods though the Current Class Definition method will likely work better. ***Class definitions themselves must be strictly limited to the top level***. This should be done the same way that global variable declarations and function definitions are lexically scoped currently. ***That is in moto.y we need to check if any scopes are***

on the scope stack and if there are, throw an error. At the same time, function definitions must now be allowed within class definitions. ***Thus we should change the behavior for checking weather a function definition is allowed to a check of whether no scope or a class scope is on the stack.*** It should go without saying that we ***need a class scoping identifier in the parser*** now too.

Since the dereference operator '.' is now a major player in moto all the parsing / evaluation stages will require **motoX_dereference_lval and motoX_dereference_rval functions.** As such ***new OpTypes DEREFERENCE_LVAL and DEREFERENCE_RVAL will be required*** to signal to those phases to call these functions

The keyword 'this' should only be usable within a method definition, that is, within a function definition that is within a class definition. The restrictions on the usage of this keyword should be made in the same place the restrictions on the usage of the return keyword are made.

Dynamic Loader

Functions defined within a Moto Class will have access to instance variables via the special variable 'this' (which has a substructure equivalent to the variables defined within the Class) or simply by using one of the variables within a method definition (assuming it was not overloaded by an argument). To do this we must ***get the class structure information out of the motod parsing phase*** since it will be needed in the type verification phase for moto defined methods. When a moto defined method is called in the interpreter (motoi_method, motoi_new), the 'this' variable, all instance variables, and the $rvalue variable can be pushed up at the same time. These MotoVars ***must be the same MotoVars as the ones associated with the Object being stored*** otherwise thier values will not persist between method calls on the object.

The Dynamic Loader must create the ***Moto Class Definition*** (mcd) as well as the MotoType required for motovals and MotoVars to store Objects of this type. This way functions defined before the Class Definition can still make use of the type.***The Class Definition Object should NOT contain MotoTypes or MotoVars or MotoVals*** as it will make it impossible to have two classes that have instance variables of each others type (indirect self reference). Thus the Class Definition Object should store simply the String types of the instance variable types just as mxfn records do. Motov will catch the error if any instance variables are of undefined types. To create the Class Definition Object however motod must start paying attention to declaration statements. Also, ***we will need to add a 'Current Class Definition' variable to the environment*** since within a class definition motod will be adding variables and methods to that definition.

Since there would be much code duplication and it would require scoping hackery within parser to do otherwise ***moto methods should be defined by motod_define and we should know that they are methods by the value of the 'Current Class Definition'*** variable in env. ***Thus we should not create a new phantom MDEFINE token***. This strategy should be true in the verifier and compiler as well and the interpreter doesn't care about definitions.

When motod is parsing through a class definition it will be pulling out declarations in order to build the MCD. What it will need to avoid however are declarations that are nested inside of Class methods. We could add the intelligence to the parser to pass declarations of

member variables with a different OpType. It is not clear though that other parse phases need this facility and there would be much code duplicated between member declarations and regular declarations. ***The 'Current Class Definition' being set in the environment will tell us that we are inside of a Class Defintion. Then, when motod_define gets executed, as long as we don't decend into the statement_list no declarations inside of methods will show up in motod.***

Moto methods are differentiated from moto functions in the mxfn records by way of the motoname attribute. ***If the motoname attribute has the form <class name>::<method name>(<#args>) then the mxfn record codes for a method***. Constructors are further differentiated from methods when the motoname has the form <class name>::<class name>(<# args>). This ***in motod_define, when we discover what we are defining is a method as opposed to a function we must be sure to set the motoname differently***. ***With regard to constructors, we need not change the motoname from other methods, however we must set the return type to that of the class***. Nothing special from other methods needs to be done for the motoname or return type of destructors since the name in the class definition will already be ~<Classname>. To get the current classname however, ***the Current Class Definition must know its name***!

Verifier

Arguments to methods should be able to shadow instance variables. This is the way things work in Java and passing arguments to set() methods with the same names as instance variables is too common a programming pattern to deny. ***Nothing however should be able to shadow 'this'***. Attempting to shadow 'this' should be an error ... but then again attempting to redefine any reserved word should be an error.

With regard to runtime scoping we must make sure that complex declarations within the Class definition do not get access to variables outside of the Class definition. ***This can be done by setting up a new private scope when verification of the class variable declarations begins***.

The de-reference operator '.' should work the same way on either the left side or right side of an equals sign in the verifier. This is exactly what happens with array_lval and array_rval. ***Thus motov_dereference_lval should simply call motov_dereference_rval.***

We may want to conider modifying motov_declare to display better error messages when verifying member variable declarations,

If a no-arg constructor is called for a moto defined class the call must succeed even if no such constructor function was explicitly written.

Interpreter

The destructructor recorded within the memory manager must be the same for each moto defined object. ***This is a necessity for the interpreter as we will not be able to create new C function pointers within the interpreter***. At deletion time an appropriate destruction algorithm can be called. This will have definite implication on when destruction must occur between page views. The destructor parsed code would need to be stored persistently since one page might not load the class definition for an object that

needs to be destroyed. ***Thus before fully implementing classes in moto it may be worthwhile to implement page caching (we can also consider a phased approach in which contructors of moto declared objects simply will not get called)***. The other ... potentially simpler option is to get garbage collection working so we can forget about all this promotion crap.

A Moto Class Definition (mcd) must keep a reference to the OpCell which defines the MCD. This way the code that is not inside any method can be executed as the first part of the constructor.

In the interpreter the Class Definition OpCell itself should be ignored (just as function and method definition opcells are). However, ***when a Moto Defined Class constructor is called we must first call the code associated with the class definition's statement list***. This will take care of complex variable declarations used for member variables of the class. What we must do in fact is ***in motoi_new, call the class definition OpCell, then, before the frame is freed, change all of its vars into member vars and migrate them to a new MCI instance***. Then we can call the Constructor like any other method. This should be done regardless of whether an explicitly defined matching constructor for the Class exists. ***If the type being created is a moto defined class, call the MCD definition OpCell before anything else***.

Since we cannot create actual C 'structs' in memory (*is this true ?*) we will likely need a new Objects that will mimic the behavior of *Class Definitions* and *Class Instances* in memory for the interpreter and verifier. For the Class Definition object we should be able to

Get the name and type of all member variables (so that we canpush all of them onto the stack when calling the motoi_method function or the motov_mdefine function)

Get a particular member variable's value out of an instance.

Set a particular member variable's value within an instance.

As a first cut it may be worthwhile to represent a Class Definition (mcd***) as a hastable of String types and a Class Instance (***mci***) as a hastable of MotoVars both keyed of the instance variable names. The performance of constructing and deconstructing MotoVars with each method call is likely a greater burden than the space savings of packing in the data to a fake structure is worth. The reason we couldn't say that a class definition wouldn't be just a hashtable of mototypes is because arrays are not unique types (though they probably should be)***

In the interpreter it will be very important ***not to free the MotoVars associated with a given MCI when the stack frame is freed***. Since we will be adding a new boolean attribute to MotoVars that says whether or not they are Class member variables (the reasoning for this is in the section on code generation) ***we can use this attribute in the interpreter to make sure these vars are not freed when the Frame they are added to is popped off the frame stack***.

It seems as though a simpler mechanism for dealing with lval/ rval difficulties with regard MCIs will suffice here. ***One function (motoi_dereference_lval) can push the MotoVar associated with the dereference onto the OpStack. The other (motoi_dereference_rval) can push the MotoVal associated with the dereference***. This is a much cleaner view of the responsibilites than what happens with arrays and it seems like this is essentially what must happen with other compiled language (e.g. a reference gets pushed onto the stack when we deal with lvalues). As with arrays ***the parser can determine whether a given dereference should result in an lvalue (MotoVar) or an rvalue (MotoVal)***. We should consider rewriting motoi_assign for

arrays and even simple variable names with this design in mind. This way, xxx_rval can always call xxx_lval and simply return the MotoVal associated with the MotoVar.

Code Generator

The C Code generated for a Moto Defined Class should be a C struct that includes all the classes member variables and a set of functions which take a pointer to that structure as their first argument. The naming for these functions should be consistent with C++ / Java symbol mangling (***on that note its about time the same thing was done with mxc generated functions***) . Should we choose at a later date to allow for interospection it will become necessary to include a pointer to an Object's structure as part of its structure. ***This should likely be done across the board to both internally and externally defined classes when it gets done.***

The very first thing a Class Instance constructor must do is allocate the memory for the generated C structure and set the instance variables to thier default values. Note that ***the constructor will be the only function that does not take a reference to the class as it's first argument***. The very last thing a Class instance constructor must do is return the Class Instance constructed. ***We will need to output an extra method for the implicit constructor. This should be done in motoc_define.*** This extra method must be called by all the other constructors (or perhaps its output could be passed in as the first argument to all the other constructors which would mean they would act just like other methods).

Class variables referred to inside of the generated methods must be referred to by this->'variable name' . Thus the code generate must be able to distinguish between arguments and instance variables. ***We can accomplish this by adding a boolean marker to MotoVars which signifies that they are instance variables.*** One very specific implication here is that ***motoc_id must output the 'this->' prefix to vars that have isMember set to true as must motoc_declare***.

When code is being generated for a class the code outside of any method should be cached seperately somewhere. When the Constructor code for that class is itself written out that cached code (for complex variable declarations) should be copied in at the beginning of the constructor. The motoc_class should in fact return nothing in the way of MotoVals just as motoc_define does not. It seems as though ***as good a place as any for this code would be in the MCD***. Then we could ***add a new function to env.c moto_emitCStructures (which should be emitted above the globals section).*** The ***methods and constructors (including the implicit constructor) should be output with the rest of the the prototypes and function definitions***. This is not a problem for ordinary methods and constructors but it is a problem for the implicit constructor since it will not have a record in env->fdefs. The naming of implicit constructors will also likely be somehwat different from other cnames. ***Thus two more environment functions are needed: moto_emitCImplicitConstructorPrototypes and moto_emitCImplicitConstructors***.

When no Constructor is specified for a Class ***a default constructor must still be written out***. ***The code for the default constructor can be generated at the end of motoc_class. It should allocate space for the class. Provide default values for all the class variables. Do any work associated with complex member variable declaraion. Then finally return a pointer to the newly allocated class.***

The scheme for generating C names and prototypes for mxfn functions must now

change to incorporate method naming. ***The name change must be made in the environment function moto_fnToCName which is called by motod_define***. The strategy used in the codex libraries of naming 'methods' as <short class name>_<method name> is a very readable one. Unfortunately there is no good way to devise a 'short class name' programattically. Thus we could just say that ***the entire classname should be prefixed***. The function names for destructors happen to include the restricted character '~' (which reminds me that this may need ot be added to the lexer). ***Thus the '~' character would need to be turned into something else (potentially an underscore)***. The arglists and argtypes output would also need to change for all methods except the constructor since in C we want the first agument passed to be a pointer to the Object. This change would need to be made in ***moto_fnToCPrototype***

Class member vars as well as the variable 'this' must be defined in the function frame created by motoc_define for both methods and constructors. Otherwise method compilation will fail. ***These must all be defined as member vars except for 'this'*** which will require special handling. We should ***modify the declaration output procedure in moto_freeFrame to never output 'this'***

Lexer / Parser Changes

New Lexer Tokens

$class(- returns CLASS
$endclass - returns ENDCLASS
class - returns ECLASS
this - returns THIS

New Or Modified Parser Rules

```
definition_statement
      : function_definition_statement
      |  class_definition_statement
;

class_definition_statement
      :  CLASSDEF  statement_list  ENDCLASSDEF
      |  CLASSDEF  ENDCLASSDEF
;

embedded_class_definition_statement
      :  ECLASS  NAME  embedded_statement
;

assignment_expression
      : NAME assignment_operator assignment_expression
      | array_subscript_expression assignment_operator assignment_expression
      | dereference_expression assignment_operator assignment_expression
      | conditional_expression
;

primary_expression
      : INTEGER
```

```
      | FPNUM
      | STRING
      | CHAR
      | BOOLEAN
      | MOTONULL
      | THIS
      | NAME
      | REGEX
      | OPENPAREN
;

postfix_expression
      : postfix_expression INC
      | postfix_expression DEC
      | dereference_expression
      | array_subscript_expression
      | NAME OPENPAREN CLOSEPAREN
      | NAME OPENPAREN expression_list CLOSEPAREN
      | primary_expression { $$ = $1; }
;

embedded_statement
      : embedded_conditional_statement
      | embedded_iterative_statement
      | embedded_definition_statement
      | embedded_class_definition_statement
      | 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_block
      | error SC
      | SC
;

dereference_expression
      : postfix_expression DOT NAME OPENPAREN CLOSEPAREN
      | postfix_expression DOT NAME OPENPAREN expression_list
CLOSEPAREN
      | postfix_expression DOT NAME
```

New terminals

THIS - matches “this”
CLASSDEF - matches “$class(“
ECLASSDEF - matches “class”
CLASSENDDEF - matches “$endclass”

New non-terminals

dereference_expression
embedded_class_definition_statement
class_definition_statement

New parser scoping tokens

CLASSDEF - Scopes between $class and $endclass
ECLASSDEF - Scopes for embedded class definitions

New OpTypes

CLASSDEF - OpType returned when a class definition is encountered
DEREFERENCE_LVAL - OpType returned when dot '.' used on the left side of an '='
DEREFERENCE_RVAL - OpType returned when dot '.' used anywhere but the immediate left side of an equals

New parse errors and associated Error Strings:

moto_malformedClassDefinition(MetaInfo* meta,char embedded)
MalformedClassDefintion - "Malformed $class statement, expected $class(<name>)\n"
MalformedEmbClassDefintion - "Malformed class defintion, expected class <name> {...} \n"

moto_illegalLVAL(MetaInfo* meta)
IllegalLVAL - "Operand of the left of '=' may not be assigned to\n"

moto_illegalClassDefininition(MetaInfo* meta,char embedded) -
IllegalClassDefinition - "$class within sub-scope\n"
EmbIllegalClassDefinition - "class definition within sub-scope\n"

moto_illegalThis(MetaInfo* meta) -
IllegalThis - "'this' not used within method defintion\n"

moto_unterminatedClassDefinition(MetaInfo* meta,char embedded) -
EmbUnterminatedClassDefinition - "Unterminated class definition\n"
UnterminatedClassDefinition - "Unterminated $class statement\n"

moto_endClassWithoutClass(MetaInfo* meta) -
EndClassWithoutClass - "$endclass without $class\n"

Environment Modifications

```
typedef struct motoClassDefinition {
      char* classn; // The class name
      SymbolTable memberTypes; // name -> type mapping
      SymbolTable memberDimensions; // name -> dimension mapping
      OpCell definition; // The OpCell associated with the class
definition, this gets interpreted as part of implicit construction
      char* code; // The implicit constructor code generated my motoc_define
} MCD;
```

typedef struct motoClassInstance {
SymbolTable memberVars; // name -> MotoVar mapping
} MCI;

void mcd_create(char* classn, const UnionCell* definition) - creates a new Moto Class Definition

void mcd_setMemberType(MCD* mcd,char* name, char* type); - sets / adds a new member variable and assocites it with the specified type

void mcd_setMemberDimension(MCD* mcd, char* name, char* type); - associes a member variable with its declared dimension

char* mcd_getMemberType(MCD* mcd, char* name); - returns a String representation of a member variable's type

int mcd_getMemberDimension(MCD* mcd, char* name); - returns a member variable's dimension

MCI* mci_createDefault(MCD* mcd) - creates a new Moto Class Instance from a Moto Class Definition

MotoVar* mci_getMemberVar(MCD* mcd,char* name); - retrieves the MotoVar for a member variable

MotoType{
boolean isExternallyDefined // True if this is an externally defined type, false if this type corresponds to an MCD
}

MotoVar{
char isMemberVar // True if this is a member variable of a class. In the interpreter this value will be used to determine which vars should not get free'd when a stack frame is popped. In the compiler this value will be used to determine which variabes should always get printed with a 'this->' prefix.
}

MotoEnv{
SymbolTable cdefs; // Used by getMotoClassDefinition and setMotoClassDefinition to associate and Moto Defined Class name with an MCD
MCD* ccdef; // Used by motod to record the current class definition being built;
}

moto_declare(MotoEnv *env, MotoType *type, char *name, char isGlobal)
- needs to be modified to set isMemberVar attribute to false

MCD* moto_getMotoClassDefinition(MotoEnv* env, char* name)
void moto_setMotoClassDefinition(MotoEnv* env, char* name, MCD* mcd)

```
MotoEnv *
moto_createEnv(int flags,char* mxpath,char* filename) - needs to be modified
to initialize cdefs and ccdef

void
moto_freeEnv(MotoEnv *env) - needs to be modified to free the cdefs

moto_freeFrame() - needs to be modified so as not to free classMemberVars and
not to output them in compiler mode.
```

Code Generation Functions

```
moto_fnToCName() - modify to output method names

moto_fnToCPrototype()

moto_emitCStructures()
      - for each MCD
            - output 'typedef struct _<typename> {'
                  - for each var in the MCD
                        - if the var is another MCD
                              - output 'struct _<var typename>'
                        - else
                              - output <typename>
                        - output varname
                        - output ';\n'
            - output '}; <typename'

moto_emitCImplicitConstructorPrototypes()
      - for each MCD
            - output an implicit constructor prototype

moto_emitCImplicitConstructors()
      - for each MCD
            - output the implicit constructor prototype for this MCD
            - allocate the space for the type being constructed and set 'this'
            - output the mcd->code
            - return 'this'

moto_emitCHeader(MotoEnv *env, StringBuffer *out) -
      - call moto_emitCStructures (before emitting globals)
      - call moto_emitCImplicitConstructorPrototypes (immediately before
emitCPrototypes)
      - call moto_emitCImplicitConstructors (immediately before
emitCFunctions)
```

Loader Modifications

New Functions

motod_class - this function will create a Moto Class Definition (MCD) and the associated MotoType

- Grab the Class name (operand 1)
- Create a new MCD
- Add a new MotoType for this MCD
- Set the Environment Variable for the 'Current Class Definition' to the MCD just created
- Call motod on the Statement List
- Un Set the 'Current Class Definition'
- Add MCD to the Class Definitions Table

motod_declare - this function will add add an instance method to the current class definition

- If the 'Current Class Definition' is not set
    - return
- Grab the variable name and type and map that name to the appropriate type in the Current MCD

Modified Functions

motod_define - the function must now create the associated mxfn record for methods as well as functions

- If the 'Current Class Definition' is set
    - prepend the class name and '**::**' to the generated motoname
    - pass the classname to moto_fnToCName to get a properly generated method name

New Loader Errors

Verifier Modifications

New Functions

motov_classdef
- push on a new private scope
- set the environments current MCD to the current class
- motov the statement list
- un-set the environment's current MCD

motov_dereference_rval -
- Evaluate operand 0 to get the motoval associated with the structure
- If operand 0 is not of type->kind REF_TYPE
    - throw error("The dereference operator '.' may only be used on Objects")
- retreive the MCD for the type of the associated motoval
- get the name of the member variable to dereference to from operand 2
- if the MCD defines no such member variable
    - throw error ("Class %s has no member named %s")
- Pushes the MotoVal associated with the dereference onto the OpStack

motov_dereference_lval - Just call motov_dereference_rval

Modified Functions

motov_assign **-** Should be refactored to simply evaluate the first operand, check types, and verify that the optype is one that is assignable to

motov_define **-** the function will now need to verify methods as well as functions

- If the current MCD is set then we are looking at a method definition
  - push all the MotoVars defined in the current mcd onto the frame
  - push 'this' onto the frame

motoc_new **-** if a no-arg constructor is called for a moto defined class, let this succeed even if no such constructor function is registered

New Verification Errors

void moto_cantDereferenceType(char *type)
CantDereferenceType - "Cannot dereference this type: <%s>\n"

void moto_noSuchMember(char *classname, char *membername)
NoSuchMember - "No such member: %s::%s\n"

New Runtime Errors

Interpreter Modifications

New Functions

motoi_dereference_rval **-** Calls motoi_dereference_lval then pushes the MotoVal associated with the MotoVar popped off of the OpStack onto the OpStack

motoi_dereference_lval **-** Pushes the MotoVar associated with the dereference onto the OpStack

- Evaluate operand 0 to get the MotoVal associated with the structure
- retreive the MCI from the MotoVal
- If the MCI is null throw a NullPointerException
- get the name of the member variable to dereference to from operand 2
- retrieve the MotoVar from the MCI with this name and push it onto the OpStack

Modified Functions

motoi_method **-** this funtion will need to be modified in order to distinguish between methods loaded from extensions and methods defined in moto

- If the type specified for created is a Moto Defined Class
  - create the private frame
  - push the arg vars onto the frame
  - declare the rvalue variable

- push all the MCI vars that are not shadowed by arg vars onto the frame
- declare the variable 'this' and set it's value to MCI (the value from Operand 1 of motoi_method
- call the UnionCell associated with the method
- free the frame

motoi_new - this function will need to be modified in order to distinguish between constructors loaded from extensions and constructors defined in moto

- If the type specified for created is a Moto Defined Class
  - Retrieve the MCD for the class being created
  - Create a new private stack frame
  - Call the opcell associated with the MCD (this will declare all and potentailly define all the member variables)
  - Instantiate a new MCI
  - Migrate all the MotoVars out of the current Frame and into the MCI (setting the isMember attribute on each MotoVar as we go)
  - Free the Stack Frame.

  - Identify the appropriate constructor
  - Call the OpCell associated with the appropriate Constructor
    - create the private frame
    - push the arg vars onto the frame
    - push all the MCI vars that are not shadowed by arg vars onto the frame
    - declare the variable 'this' and set it's value to the new MCI
    - Call the OpCell associated with the Constructor
    - free the frame (we actually don't need to worry about rvalue since all we care about is the MCI for constructors)
  - Set the return var->v->value to the MCI

motoi_assign -
- Evaluate the first operand to get the Member MotoVar
- Set the vars value to the new value
- Return a clone of the vars values

Compiler Modifications

New Functions

motoc_class - Generate the code generated for the implicit constructor
- Grab the Class name (operand 1)
- Get the MCD for this class name
- Set the Environment Variable for the 'Current Class Definition' to this MCD
- Create a private frame
- Generate the C code for the implicit constructor
- Set the MCD code to the of the definition frame
  Luckily this will not have variable declarations in it
- Free the definition frame

- UnSet the Environment Variable for the 'Current Class Definition'

motoc_dereference_rval -
- Evaluate Operand 0 to get the motoval associated with the structure
- Retreive the MCD for the type of the associated motoval
- Get the name of the member variable to dereference to from operand 1
- Create the return val of appropriate type and dimension
- Set the codeval of the return val to instanceVal.code + "->" + op2 name

motoc_dereference_lval - Just call motoc_dereference_rval

Modified Functions

motoc_define - the function will now need to generate code for methods as well as functions.
- The motoname generated needs to be updated to include possible method definitions
- Member vars and 'this' need to be pushed onto the definition frame if this is a method or constructor definition.

motoc_declare

motoc_id - if the var is a member var prefix 'this->'

void motoc_postfixIncDec(const UnionCell *p) - needs to be modified to not retrieve the var directly but call motoc_id !

motoc_new
- need to be able to call implicit constructors
- need to modify the call to any MDC constructor to pass as the first argument the call to the implicit constructor

Examples

```
$class(ABook)
      $declare(int i = 27*3)
$endclass
$declare(ABook a = new ABook())
$(a.i)

${
      class BBook{
            int chapters;
            String chapterContents[];

            BBook BBook(int chapters){
                  this.chapters = chapters;
                  chapterContents = new String[chapters];

                  int i;
                  for(i=0;i<chapters;i++)
                        chapterContents[i]="Chapter "+str(i+1);
```

```
            }

            String getChapter(int i){
                  if ( i >= 0 && i< chapters)
                        return this.chapterContents[i];

                  print "Chapter "+str(i)+" out of range\n";
                  return null;
            }
      }
      BBook b = new BBook(3);
        print b.getChapter(2)+"\n";
        print b.getChapter(7);
}$
$(b.chapters)
```

Yacc Errors

```
$endclass               // EndClassWithoutClass

$(this)                 // illegalThis
$do("goo".size() = 1)   // illegalLVAL
$if(true)
      $class(foo)       // illegalClassDefinition
      $endclass
$endif
$class(2+2)             // malformedClass
$class(bar)             // unterminateClassDefinition
```

Loader Errors

```
${
      class foo {}
      class foo {} //  Class foo has already been defined
      class bar {
            int foo() {}
            int foo() {} // Method bar::foo(0) of has already ben defined
      }
}$
```

Verifier Errors

```
${
      class foo {
            String bar;
            int maka[];

            foo(String bar,int maka[]){
                  this.bar = bar;
                  this.foo = var;   // No Such Member
            }
      }
```

```
    23.maka;    // Cannot Dereference Ints
    “hello”.maka;     // No Such Member
}$
```