msdscript API

Expr classes

|  |  |  |
| --- | --- | --- |
| Expr class | Expr is the fundamental type in msdscript. It is used to represent mathematical expressions, booleans, functions and conditional statements. This includes a wide range of ideas, including addition, multiplication, functions, if-then statements, and others. No subtraction though. These expressions can be nested within each other using the inheriting classes' member variables. | |
| Inheriting classes: | AddExpr (addExpr.h)  BoolExpr (boolExpr.h)  CallExpr (callExpr.h)  EqExpr (eqExpr.h)  FunExpr (funExpr.h)  IfExpr (ifExpr.h)  LetExpr (letExpr.h)  MultExpr (multExpr.h)  NumExpr (numExpr.h)  VarExpr (varExpr.h) | |
| Member variables: | None | |
| Methods: | equals(PTR(Expr)e) | |
|  | Description: | equals() is used to compare expressions. While each implementation in the inheriting classes differs slightly, they all use some form of casting the Expr given as an argument and seeing if the member variables match the Expr that is calling the method. |
| Arguments: | PTR(Expr) |
| Return: | boolean |
|  | |
| interp(PTR(Env) env) | |
| Description: | interp() takes an expression (including nested espressions) and produces the result. It does this on the stack, and so is limited if you are trying to interpret large expressions, such as counting down from 1,000,000. |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | step\_interp() takes an expression and produces the result, like interp() above. However, it does not perform traditional recursion to do so, so it is not as limited as interp() for large expressions. It can countdown from 1,000,000. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | print() prints an expression (including nested expressions). It follows a fairly simple format, some examples are found in the implementations of this method in the inheriting classes. |
| Arguments: | std::ostream |
| Return: | none |
|  | |
| pretty\_print(std::ostream& stream) | |
| Description: | pretty\_print() is currently set up to just call print(). If desired, you could format this print option however you would like. |
| Arguments: | std::ostream |
| Return: | none |
|  |  |  |
|  | pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
|  | Description: | used in the pretty\_print() method, it takes a mode and prints a certain way accordingly. |
|  | Arguments: | print\_mode\_t  std::ostream |
|  | Return: | none |
|  |  |  |
|  | to\_string() | |
|  | Description: | to\_string takes an expression and converts it to a string by using the print() method above. This method is implemented for the Expr class, and is not changed by inheriting classes. |
|  | Arguments: | none |
|  | Return: | std::string |
|  |  |  |
|  | to\_string\_pretty() | |
|  | Description: | to\_string\_pretty() takes an expression and converts it to a string like to\_string() above, but uses the optional (and currently not implemented) custom printing format in pretty\_print(), rather than the regular print(). |
|  | Arguments: | none |
|  | Return: | std::string |

|  |  |  |
| --- | --- | --- |
| AddExpr class | AddExpr is a type of expression, which specifically represents a mathematical expression where two expressions are being added together.  For example, these inner epressions could be numbers (respresented by the NumExpr class):  1 + 1  variables (VarExpr class):  1 + x  or other larger expressions such as AddExpr, MultExpr, and more:  1 + (1 + 3)  1 + (4 \* 8)  (60 \* 43) + (54 \* x)    (Please refer to the msdscriptGuide.docx file for further illustrations of how this works.)  The plus sign (+) is not accounted for as a member variable in this class, it is known that the left hand side expression (member variable Expr lhs) is being added to the right hand side espression (member variable Expr rhs). The print() method does print the plus sign, and the parser (see parse.h and parse.cpp) uses the plus sign to construct an AddExpr. | |
| Member variables: | PTR(Expr) lhs  PTR(Expr) rhs | |
| Methods: | Constructor:  AddExpr(PTR(Expr)lhs, PTR(Expr)rhs) | |
|  | Description: |  |
| Arguments: | PTR(Expr)  PTR(Expr) |
| Return: | AddExpr |
|  | |
| equals(PTR(Expr)e) | |
| Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the  argument to an AddExpr, see if it was the same class, then compare each member variable. This does not interpret the expression. |
| Arguments: | PTR(Expr) |
| Return: | boolean |
|  | |
| interp(PTR(Env) env) | |
| Description: | As interp() returns a Val object, we interp() each member variable, then use the Val class's add\_to() method (see val.h) to combine them both into one Val using addition. |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | Same idea as interp() above, but it uses the continuation format so we can perform the operation without using traditional recursion. Does not directly use the Val class's add\_to() method, but passes info on to the next step in the continuation loop, which in this case is RightThenAddCont. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | Uses a std::ostream reference to print the member variables separated by a plus sign (with a space on either side of the plus sign) and the whole AddExpr surrounded by parentheses, and no space between the member variables and the parentheses. |
| Arguments: | std::ostream |
| Return: | none |
|  | |
| pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
| Description: | Checks the print mode, but then currently does the exact same thing as AddExpr::print(). |
| Arguments: | print\_mode\_t  std::ostream |
| Return: | none |

|  |  |  |
| --- | --- | --- |
| BoolExpr class | BoolExpr is an expression that represents a boolean. It is one of the 'end points' of the Expr subclasses, which means that other expressions cannot be nested inside of it. The only member variable is a boolean, not another Expr. | |
| Member variables: | bool rep | |
| Methods: | Constructor:  BoolExpr(bool rep) | |
|  | Description: |  |
| Arguments: | bool |
| Return: | BoolExpr |
|  | |
| equals(PTR(Expr)e) | |
| Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to an BoolExpr, see if it was the same class, then compare each member variable, which in this case is just a bool. This does not interpret the expression. |
| Arguments: | PTR(Expr) |
| Return: | bool |
|  | |
| interp(PTR(Env) env) | |
| Description: | Returns a new BoolVal object with the same member variable (ie, same boolean value). |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | As this is one of the 'end points' of step interpretation, it saves a new BoolVal in the Step::val field, not unlike how BoolExpr::interp() returns a new BoolVal. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | Prints "\_true" or "\_false" with an underscore preceding the word, depending on the value of the rep member variable of the calling BoolExpr. |
| Arguments: | std::ostream |
| Return: | none |
|  |  |
| pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
| Description: | Not fully implemented. Currently does the same thing as BoolExpr::print() |
| Arguments: | print\_mode\_t  std::ostream |
| Return: | none |

|  |  |  |
| --- | --- | --- |
| CallExpr class | CallExpr is a type of expression, to be used explicitly to 'call' a function, represented by a FunExpr (see funExpr.h). | |
| Member variables: | PTR(Expr) to\_be\_called  PTR(Expr) actual\_arg | |
| Methods: | Constructor:  CallExpr(PTR(Expr) to\_be\_called, PTR(Expr) actual\_arg) | |
|  | Description: |  |
| Arguments: | PTR(Expr)  PTR(Expr) |
| Return: | CallExpr |
|  | |
| equals(PTR(Expr)e) | |
| Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to a CallExpr, see if it was the same class, then compare each member variable. This does not interpret the expression. |
| Arguments: | PTR(Expr) |
| Return: | bool |
|  | |
| interp(PTR(Env) env) | |
| Description: | Uses the Val class's call() method to call the function stored in to\_be\_called() using the actual\_arg member variable. |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | Same idea as interp() above, but it uses the continuation format so we can perform the operation without using traditional recursion. Does not directly use the Val class's call() method. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | Uses a std::ostream erference to print the actual\_arg member variable surrounded by parentheses, with no spaces in between. |
| Arguments: | std::ostream |
| Return: | none |
|  |  |
| pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
| Description: | Currently just calls print(). See expr.h for more description. |
| Arguments: | print\_mode\_t  std::ostream |
| Return: | none |

|  |  |  |
| --- | --- | --- |
| EqExpr class | EqExpr is a type of expression, which specifically represents a comparison between two expressions.  For example, an EqExpr could be represented visually like this:    1 == 2    where 1 is the lhs member variable and 2 is the rhs member variable.  As the member variables are Expr's, it is possible to nest expressions using this class, as follows:  2 == (1 + 1)  where 2 is the lhs (NumExpr) and (1 + 1) is the rhs (AddExpr).  The double equals sign (==) is not accounted for as a member variable in this class, it is known that the left hand side expression (member variable Expr lhs) is being compared to the right hand side expression (member variable Expr rhs). The print() method does print the double equals sign, and the parser (see parse.h and parse.cpp) uses the double equals sign to construct an EqExpr. | |
| Member variables: | PTR(Expr) lhs  PTR(Expr) rhs | |
| Methods: | Constructor:  EqExpr(PTR(Expr) lhs, PTR(Expr) rhs) | |
|  | Description: |  |
| Arguments: | PTR(Expr)  PTR(Expr) |
| Return: | EqExpr |
|  | |
| equals(PTR(Expr)e) | |
| Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to an EqExpr, see if it was the same class, then compare each member variable. This does not interpret the expression. |
| Arguments: | PTR(Expr) |
| Return: | bool |
|  | |
| interp(PTR(Env) env) | |
| Description: | As interp() returns a Val object, we interp() each member variable, then use the Val class's equals() method (see val.h) to compare them and know whether we should return a BoolVal(true) or BoolVal(false).  See boolVal.h. |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | Same idea as interp() above, but it uses the continuation format so we can perform the operation without using traditional recursion. Does not directly use the Val class's equals() method. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | Checks the print mode, but then currently does the exact same thing as EqExpr::print(). |
| Arguments: | std::ostream |
| Return: | none |
|  |  |
| pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
| Description: | Currently just calls print(). See expr.h for more description. |
| Arguments: | print\_mode\_t  std::ostream |
| Return: | none |

|  |  |  |
| --- | --- | --- |
| FunExpr class | FunExpr is a type of expression, which specifically represents a function. One of the member variables is an Expr, so you can nest other expressions in this class. The FunExpr::body member variable is the function, the FunExpr::formal\_arg member variable is the string to use as a variable in the function. When writing a functionin msdscript, it should be in this format:  \_fun(x)(x + 1)  where \_fun is the keyword to the parser that a function is about to start, (x) is the FunExpr::formal\_arg member variable, and (x + 1) is the FunExpr::body member variable.  Note that when printing, it will just print:  [function] | |
| Member variables: | std::string formal\_arg  PTR(Expr) body | |
| Methods: | Constructor:  FunExpr(std::string formal\_arg, PTR(Expr) body) | |
|  | Description: |  |
| Arguments: | std::string  PTR(Expr) |
| Return: | FunExpr |
|  | |
| equals(PTR(Expr)e) | |
| Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to an FunExpr, see if it was the same class, then compare each member variable. This does not interpret the expression. |
| Arguments: | PTR(Expr) |
| Return: | bool |
|  | |
| interp(PTR(Env) env) | |
| Description: | Returns a new FunVal object using the FunExpr::formal\_arg and FunExpr::body member variables from the object that is calling the function, and adds the Env object taken as a parameter, which will be used to call the function. Only FunVal objects can be called using the Val::call() method. |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | As this is one of the 'end points' of step interpretation, it saves a new FunVal in the Step::val field, not unlike how FunExpr::interp() returns a new FunVal. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | Instead of actually printing the member variables, this prints just the word "function" inside square brackets, as follows:  [function] |
| Arguments: | std::ostream |
| Return: | none |
|  |  |
| pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
| Description: | Currently just calls print(). See expr.h for more description. |
| Arguments: | print\_mode\_t  std::ostream |
| Return: | none |

|  |  |  |
| --- | --- | --- |
| IfExpr class | IfExpr is a type of expression, which specifically represents an if-then-else statement, where the IfExpr::if\_side is evaluated as a bool, and if it is true the IfExpr::then\_side executes, but if false the IfExpr::else\_side executes.  When entering in the program, using underscores before the keywords "if", "then", and "else" is necessary. Therefore:  \_if \_true \_then 4 \_else 5  would execute 4. Since each member variable is an Expr, you can nest expressions here, as shown:  \_if (2 == 1) \_then x + 234 \_else x + 12  would execute x + 12. | |
| Member variables: | PTR(Expr) if\_side  PTR(Expr) then\_side  PTR(Expr) else\_side | |
| Methods: | Constructor:  IfExpr(PTR(Expr) if\_side, PTR(Expr) then\_side, PTR(Expr) else\_side) | |
|  | Description: |  |
| Arguments: | PTR(Expr)  PTR(Expr)  PTR(Expr) |
| Return: | IfExpr |
|  | |
| equals(PTR(Expr)e) | |
| Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to a FunExpr, see if it was the same class, then compare each member variable. This does not interpret the expression. |
| Arguments: | PTR(Expr) |
| Return: | bool |
|  | |
| interp(PTR(Env) env) | |
| Description: | Checks to see if the IfExpr::if\_side interprets to true. If it does, then the IfExpr::then\_side is interpreted and returned. If not, the the IfExpr::else\_side is interpreted returned. |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | Same idea as interp() above, but it uses the continuation format so we can perform the operation without using traditional recursion. Does not compare against a BoolVal or determine whether to use the IfExpr::then\_side or IfExpr::else\_side, it passes that to the next step in the continuation loop, which in this case is IfBranchCont. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | Uses a std::ostream reference to print the whole expression in parentheses, with each member variable preceded by an underscore, its respective keyword, and a space; "-if ", "\_then ", or "\_else ".  Example:  (\_if (1 == 1) \_then 1 \_else 2) |
| Arguments: | std::ostream |
| Return: | none |
|  |  |
| pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
| Description: | Currently just calls print(). See expr.h for more description. |
| Arguments: | print\_mode\_t  std::ostream |
| Return: | none |

|  |  |  |
| --- | --- | --- |
| LetExpr class | LetExpr is almost like a function that calls itself. The LetExpr::lhs member variable is a std::string that probably (but not necessarily) exists in the LetExpr::body member variable. The LetExpr::rhs member variable is what will replace the variable (LetExpr::lhs) in LetExpr::body when the expression is interpreted. This is the syntax, with the member variable names in []:  \_let [lhs] = [rhs] \_in [body]  So:  \_let x = 1 \_in x + 1  would interpret to 2.  Since two of the member variables are Exprs, you can nest expressions in those variables. | |
| Member variables: | std::string lhs  PTR(Expr) rhs  PTR(Expr) body | |
| Methods: | Constructor:  LetExpr(std::string lhs, PTR(Expr) rhs, PTR(Expr) body) | |
|  | Description: |  |
| Arguments: | std::string  PTR(Expr)  PTR(Expr) |
| Return: | LetExpr |
|  | |
| equals(PTR(Expr)e) | |
| Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to a FunExpr, see if it was the same class, then compare each member variable, which in this case is a std::string and two Exprs. This does not interpret the expression. |
| Arguments: | PTR(Expr) |
| Return: | bool |
|  | |
| interp(PTR(Env) env) | |
| Description: | This uses an environment to interpret the LetExpr::rhs Expr, then uses that Val along with LetExpr::lhs to make a new ExtendedEnv, which is used to interpret the LetExpr::body Expr and return the Val that is produced. This is how the lhs string is replaced with the rhs Expr in the body. |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | Same idea as interp() above, but it uses the continuation format so we can perform the operation without using traditional recursion. It passes a lot of the work done in LetExpr::interp() on to the next step in the continuation loop, which in this case is LetBodyCont. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | Uses a std::ostream reference to print the whole expression in parentheses, with the lhs and body preceded by an underscore and its respective keyword and a space, and the rhs preceded by a space, an equals sign, and a space.  Example:  (\_let x = 1 \_in (x + 1))  LetExpr::body will be printed with parentheses around it as well, but that happens when print is called on AddExpr, not here. |
| Arguments: | std::ostream |
| Return: | none |
|  |  |
| pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
| Description: | Currently just calls print(). See expr.h for more description. |
| Arguments: | print\_mode\_t  std::ostream |
| Return: | none |

|  |  |  |
| --- | --- | --- |
| MultExpr class | MultExpr is a type of expression, which specifically represents a mathematical expression where two expressions are being multiplied together.  For example, these inner expressions could be numbers (represented by the NumExpr class):  2 \* 2  variables (VarExpr class):  3 \* x  or other larger expressions such as AddExpr, MultExpr, and more:  1 \* (1 + 1)  2 \* (2 \* 3)  (12 + 1) \* (8 + 12)  (Please refer to the msdscriptGuide.docx file for further illustrations of how this works.)  The multiply sign (\*) is not accounted for as a member variable in this class, it is known that the left hand side expression (MultExpr::lhs) is being multiplied to the right hand side expression (MultExpr::rhs). The print() method does print the multiply sign, and the parser (see parse.h and parse.cpp) uses the multiply sign to construct a MultExpr. | |
| Member variables: | PTR(Expr) lhs  PTR(Expr) rhs | |
| Methods: | Constructor:  MultExpr(PTR(Expr) lhs, PTR(Expr) rhs) | |
|  | Description: |  |
| Arguments: | PTR(Expr)  PTR(Expr) |
| Return: | MultExpr |
|  | |
| equals(PTR(Expr)e) | |
| Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to an MultExpr, see if it was the same class, then compare each member variable. This does not interpret the expression. |
| Arguments: | PTR(Expr) |
| Return: | bool |
|  | |
| interp(PTR(Env) env) | |
| Description: | As interp() returns a Val object, we interp() each member variable, then use the Val class's mult\_to() method (see val.h) to combine them both into one Val using addition. |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | Same idea as interp() above, but it uses the continuation format so we can perform the operation without using traditional recursion. Does not directly use the Val class's mult\_to() method, but passes info on to the next step in the continuation loop, which in this case is RightThenMultCont. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | Uses a std::ostream reference to print the member variables separated by a plus sign (with a space on either side of the multiply sign) and the whole AddExpr surrounded by parentheses, and no space between the member variables and the parentheses.  Example:  (1 \* 1) |
| Arguments: | std::ostream |
| Return: | none |
|  |  |
| pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
| Description: | Currently does the same thing as print(). See expr.h for more description. |
| Arguments: | print\_mode\_t  std::ostream |
| Return: | none |

|  |  |  |
| --- | --- | --- |
| NumExpr class | NumExpr is a type of expression, which specifically represents a number. It is an 'end point' Expr, meaning that it does not have an Expr as a member variable, it just has the NumExpr::rep, which is an integer. | |
| Member variables: | int rep | |
| Methods: | Constructor:  NumExpr(int rep) | |
|  | Description: |  |
| Arguments: | int |
| Return: | NumExpr |
|  | |
| equals(PTR(Expr)e) | |
| Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to a NumExpr, see if it was the same class, then compare each member variable, which is just an int. This does not interpret the expression. |
| Arguments: | PTR(Expr) |
| Return: | bool |
|  | |
| interp(PTR(Env) env) | |
| Description: | As this is one of the 'end points' interpretation, it just returns a NumVal with the same int as a number variable. |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | As this is one of the 'end points' of step interpretation, it saves a new NumVal in the Step::val field, not unlike how NumExpr::interp() returns a new NumVal. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | Prints the int value stored in NumExpr::rep, without parentheses or spaces. |
| Arguments: | std::ostream |
| Return: | none |
|  |  |
| pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
| Description: | Not fully implemented. Currently does the exact same thing as NumExpr::print() above. |
| Arguments: | print\_mode\_t  std::ostream |
| Return: | none |

|  |  |  |
| --- | --- | --- |
| VarExpr class | VarExpr is a type of expression, which specifically represents a variable. It is an 'end point' Expr, meaning that it does not have an Expr as a member variable, it just has the VarExpr::rep, which is a std::string. | |
| Member variables: | std::string rep | |
| Methods: | Constructor:  VarExpr(std::string rep) | |
|  | Description: |  |
| Arguments: | std::string |
| Return: | VarExpr |
|  | |
| equals(PTR(Expr)e) | |
| Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to a VarExpr, see if it was the same class, then compare each member variable, which is just a std::string. This does not interpret the expression. |
| Arguments: | PTR(Expr) |
| Return: | bool |
|  | |
| interp(PTR(Env) env) | |
| Description: | This uses the Env class's lookup() method to look through the Env provided as an argument to see if the VarExpr::rep is in the environment. If it is, it will return the value associated with it. If not, then it will eventually reach an EmptyEnv and will throw an error. |
| Arguments: | PTR(Env) |
| Return: | PTR(Val) |
|  | |
| step\_interp() | |
| Description: | Same idea as interp() above, but it uses the continuation loop so we can perform the operation without using tradtional recursion. Instead of returning a val, it stores the Val that comes from the Env::lookup() methos in the Step::val field. |
| Arguments: | none |
| Return: | none |
|  |  |
| print(std::ostream& stream) | |
| Description: | Prints the std::string value stored in VarExpr::rep, without parentheses or spaces. |
| Arguments: | std::ostream |
| Return: | none |
|  |  |
| pretty\_print\_at(print\_mode\_t mode, std::ostream& stream) | |
| Description: | Not fully implemented. Currently does the exact same thing as VarExpr::print() above. |
| Arguments: | print\_mode\_t  std::ostream |
| Return: | none |

Val classes

|  |  |  |
| --- | --- | --- |
| Val class | Val is another of the main classes of msdscript, along with Expr. A Val pointer is what is returned when Expr classes call interp(), and we use this class for that because its methods add\_to() and mult\_to() become very important when using step\_interp(), as well as using call() and call\_step with functions.  These classes and their corresponding Expr version are the 'end point' classes for the interp process. Tests for each inheriting class and its methods are included in the .cpp file for each class. | |
| Inheriting classes: | BoolVal (boolVal.h)  FunVal (funVal.h)  NumVal (numVal.h) | |
| Member variables: | None | |
| Methods: | equals(PTR(Val) val) | |
|  | Description: | equals() is used to compare values (Val). While each implementation in the inheriting classes differ slightly, they all use some form of casting the Val given as an argument and seeing if the member variables match the Val that is calling the method. This matches what the Expr::equals() methods do. |
| Arguments: | PTR(Val) |
| Return: | boolean |
|  | |
| add\_to(PTR(Val) val) | |
| Description: | add\_to() only actually performs with the NumVal class, where it adds the member variable of that class to the member variable of the Val provided as an argument. If the other Val classes call this, an error is thrown. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  | |
| mult\_to(PTR(Val) val) | |
| Description: | mult\_to() only actually performs with the NumVal class, where it multiplies the member variable of that class to the member variable of the Val provided as an argument. If the other Val classes call this, an error is thrown. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  |  |
| to\_string() | |
| Description: | to\_string returns a std::string. For NumVal it returns the member variable as a string, for FunVal and BoolVal it returns a custom string. |
| Arguments: | none |
| Return: | std::string |
|  | |
| call(PTR(Val) val) | |
| Description: | call() only actually performs with the FunVal class, where it interps the FunVal::body member variable with the FunVal::env member variable. If NumVal or BoolVal try to call it, an error is thrown. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  |  |  |
|  | call\_step(PTR(Val) actual\_arg\_val, PTR(Cont) rest) | |
|  | Description: | Same idea as call() above, but using the continuation loop instead of traditional recursion. |
|  | Arguments: | PTR(Val)  PTR(Cont) |
|  | Return: | none |

|  |  |  |
| --- | --- | --- |
| BoolVal class | BoolVal is a value that is returned when interp() is called on a BoolExpr. Its member variable is just a bool, like BoolExpr, so the main difference is when we use it and what methods it has. When it prints, it prints a std::string to represent true or false, as shown: | |
| Member variables: | bool rep | |
| Methods: | equals(PTR(Val) val) | |
|  | Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to an BoolVal, see if it was the same class, then compare each member variable. |
| Arguments: | PTR(Val) |
| Return: | boolean |
|  | |
| add\_to(PTR(Val) val) | |
| Description: | This method is intended to use with NumVals only, so here it throws a std::runtime error. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  | |
| mult\_to(PTR(Val) val) | |
| Description: | This method is intended to use with NumVals only, so here it throws a std::runtime error. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  |  |
| to\_string() | |
| Description: | Depending on the bool stored in BoolVal::rep, returns one of two strings:  \_true  \_false |
| Arguments: | none |
| Return: | std::string |
|  | |
| call(PTR(Val) val) | |
| Description: | This method is intended to use with FunVals only, so here it throws a std::runtime error. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  |  |  |
|  | call\_step(PTR(Val) actual\_arg\_val, PTR(Cont) rest) | |
|  | Description: | This method is intended to use with FunVals only, so here it throws a std::runtime error. |
|  | Arguments: | PTR(Val)  PTR(Cont) |
|  | Return: | none |

|  |  |  |
| --- | --- | --- |
| FunVal class | FunVal is a value that is returned when interp() is called on a FunExpr. its member variables include a std::string that represents an argument in the function, an Expr body of the function, and an Env environment to interpret the variable in the function. | |
| Member variables: | std::string formal\_arg  PTR(Expr) body  PTR(Env) env | |
| Methods: | equals(PTR(Val) val) | |
|  | Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to an FunVal, see if it was the same class, then compare each member variable. Makes use of the Expr class's equals() method to compare the FunExpr::body member variables. |
| Arguments: | PTR(Val) |
| Return: | boolean |
|  | |
| add\_to(PTR(Val) val) | |
| Description: | This method is intended to use with NumVals only, so here it throws a std::runtime error. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  | |
| mult\_to(PTR(Val) val) | |
| Description: | This method is intended to use with NumVals only, so here it throws a std::runtime error. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  |  |
| to\_string() | |
| Description: | Rather than converting the various member variables to a string, this just returns a std::string that says function in square brackets. |
| Arguments: | none |
| Return: | std::string |
|  | |
| call(PTR(Val) val) | |
| Description: | This is how we interpret functions. It is basically a wrapper to call interp() on the FunVal::body member variable, by creating a new ExtendedEnv using the Val provided as an argument and FunExpr::formal\_arg. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  |  |  |
|  | call\_step(PTR(Val) actual\_arg\_val, PTR(Cont) rest) | |
|  | Description: | Basically does the same thing as call() above, but using the continuation loop. Instead of calling interp() on the body, it stores the FunVal::body and ExtendedEnv in the appropriate Step fields. |
|  | Arguments: | PTR(Val)  PTR(Cont) |
|  | Return: | none |

|  |  |  |
| --- | --- | --- |
| NumVal class | NumVal is a value that is returned when interp() is called on a NumExpr. Its member variable is an int, like a NumExpr. The add\_to() and mult\_to() methods are used for this class in the continuation loop. | |
| Member variables: | int rep | |
| Methods: | equals(PTR(Val) val) | |
|  | Description: | Uses std::dynamic\_cast (in the CAST() macro, see pointer.h) to cast the argument to an NumVal, see if it was the same class, then compare each member variable. |
| Arguments: | PTR(Val) |
| Return: | boolean |
|  | |
| add\_to(PTR(Val) val) | |
| Description: | Method to add one NumVal to another. Uses a similar technique as the equals() methods, where it casts the input to a NumVal, but in this case throws an error if the argument Val was not a NumVal. It then returns a new NumVal where the NumVal::rep is the sum of the two NumVals' reps. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  | |
| mult\_to(PTR(Val) val) | |
| Description: | Method to multiply ove NumVal with another. Uses a similar technique as the equals() methods, where it casts the input to a NumVal, but in this case throws an error if the argument Val was not a NumVal. It then returns a new NumVal where the NumVal::rep is the product of the two NumVals' reps. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  |  |
| to\_string() | |
| Description: | Method that returns the int stored in NumVal::rep as a std::string. |
| Arguments: | none |
| Return: | std::string |
|  | |
| call(PTR(Val) val) | |
| Description: | This method is intended to use with FunVals only, so here it throws a std::runtime error. |
| Arguments: | PTR(Val) |
| Return: | PTR(Val) |
|  |  |  |
|  | call\_step(PTR(Val) actual\_arg\_val, PTR(Cont) rest) | |
|  | Description: | This method is intended to use with FunVals only, so here it throws a std::runtime error. |
|  | Arguments: | PTR(Val)  PTR(Cont) |
|  | Return: | none |

Step class

|  |  |  |  |
| --- | --- | --- | --- |
| Step class | This is the class that makes the alternative to Expr::interp() work. Instead of returning values like Expr::interp() does, calling Expr::step\_interp() stores the relevant values in the Step class member variables, and uses those along with the Cont class to interpret the expressions without the same memory restrictions that Expr::interp() has when it recursively calls functions and allocates more and more space on the stack. | | |
| Inheriting classes: | BoolVal (boolVal.h)  FunVal (funVal.h)  NumVal (numVal.h) | | |
| Member variables: | static mode\_t mode | | Mode indicates whether the next step is to start interpreting an expression or start delivering a value to a continuation. |
|  | static PTR(Expr) expr | | The expression to interpret, meaningful only when 'mode' is 'interp\_mode' |
| static PTR(Env) env | | The environment to be used when interpreting, only when 'mode' is 'interp\_mode' |
| static PTR(Val) val | | The value to be delivered to the continuation, meaningful only when 'mode' is 'continue\_mode’ |
| static PTR(Cont) cont | | The continuation to receive a value, meaningful only when 'mode' is 'continue\_mode' |
| Methods: | interp\_by\_steps(PTR(Expr) e) | | |
|  | Description: | This method sets the end of the continuation loop, with an EmptyEnv in Step::env, a nullptr in Step::val, and a DoneCont in Step::cont, and then calls step\_interp() on Step::expr or step\_continue on Step::cont. | |
| Arguments: | PTR(Expr) | |
| Return: | PTR(Val) | |

Cont classes

|  |  |  |
| --- | --- | --- |
| Cont class | The Cont class is basically what the continuation loop uses to determine what to do with the next step, which 'side' to continue with, what to do with the other info, when to finish, etc. The description is basically in the name of the classes. | |
| Inheriting classes: | DoneCont  RightThenAddCont  AddCont  RightThenMultCont  MultCont  RightThenEqCont  EqCont  IfBranchCont  LetBodyCont  ArgThenCallCont  CallCont | |
| Member variables: | static PTR(Cont) done | |
| Methods: | step\_continue() | |
|  | Description: |  |
| Arguments: | none |
| Return: | none |

|  |  |  |
| --- | --- | --- |
| Function: | use\_arguments(int argc, char \*\*argv) | |
|  | Description: | This function takes arguments from the command line to first decide which mode to enter, and then to parse the script that is entered afterward. It makes use of the various parsing methods found in parse.h. |
| Arguments: | int  char\*\* |
| Return: | none |