

abr 12, 16 16:39

thread.h

Page 1/1

```

1  #ifndef __THREADS_H__
2  #define __THREADS_H__
3
4  #include <pthread.h>
5
6
7  class Thread {
8  private:
9      pthread_t thread;
10
11      static void* runner(void *data);
12
13  public:
14      Thread() {}
15
16      void start();
17      void join();
18
19      virtual void run() = 0;
20      virtual ~Thread() {}
21
22  private:
23      Thread(const Thread&);
24      Thread& operator=(const Thread&);
25  };
26
27
28  #endif

```

abr 12, 16 16:39

thread.cpp

Page 1/1

```

1
2  #include "thread.h"
3
4  void* Thread::runner(void *data) {
5      Thread* self = (Thread*) data;
6      self->run();
7      return NULL;
8  }
9
10 void Thread::start() {
11     pthread_create(&thread, NULL, Thread::runner, this);
12 }
13
14 void Thread::join() {
15     pthread_join(thread, NULL);
16 }

```

abr 12, 16 16:39

parser.h

Page 1/1

```

1  #ifndef __PARSER_H__
2  #define __PARSER_H__
3
4  #include <string>
5
6  #include "expressions.h"
7  #include "atoms.h"
8  #include "factories.h"
9
10 enum ParsingContext {
11     CommonExpression,
12     Setq,
13     Sync,
14     Defun,
15     Fun
16 };
17
18 class Parser {
19     Context& globalContext_;
20     ParsingContext parsingContext_;
21
22     Expression* getExpressionInstance_(std::string name);
23
24     Atom* getAtomInstance_(std::string s);
25
26     Expression* functionExpression_(Expression* r, const std::string s);
27     Expression* parseExpression_(const std::string s);
28
29 public:
30     explicit Parser(Context& globalContext);
31
32     Expression* parse(const std::string s);
33     ParsingContext getParsingContext();
34 };
35
36
37
38 #endif

```

abr 12, 16 16:39

parser.cpp

Page 1/3

```

1
2  #include <algorithm>
3  #include <string>
4  #include <iostream>
5  #include <sstream>
6
7  #include "parser.h"
8
9
10 bool isNumber(const std::string& s) {
11     std::string::const_iterator it = s.begin();
12     while (it != s.end() & isdigit(*it)) ++it;
13     return !s.empty() & it == s.end();
14 }
15
16 bool isExpression_(const std::string& s) {
17     return (s[0] == '(' & s[s.size() - 1] == ')');
18 }
19
20
21 Parser::Parser(Context& globalContext) : globalContext_(globalContext) {
22 }
23
24 Expression* Parser::parse(const std::string s) {
25     parsingContext_ = CommonExpression;
26     return parseExpression_(s);
27 }
28
29 ParsingContext Parser::getParsingContext() {
30     return parsingContext_;
31 }
32
33
34 Expression* Parser::getExpressionInstance_(const std::string name) {
35     ExpressionFactory& expFact = globalContext_.getExpressionFactory();
36
37     if (name == "print") {
38         return expFact.createPrint();
39     } else if (name == "+") {
40         return expFact.createSum();
41     } else if (name == "-") {
42         return expFact.createDiff();
43     } else if (name == "*") {
44         return expFact.createMul();
45     } else if (name == "/") {
46         return expFact.createDiv();
47     } else if (name == "=") {
48         return expFact.createEqual();
49     } else if (name == "<") {
50         return expFact.createLesser();
51     } else if (name == ">") {
52         return expFact.createGreater();
53     } else if (name == "list") {
54         return expFact.createList();
55     } else if (name == "car") {
56         return expFact.createCar();
57     } else if (name == "cdr") {
58         return expFact.createCdr();
59     } else if (name == "append") {
60         return expFact.createAppend();
61     } else if (name == "if") {
62         return expFact.createIf();
63     } else if (name == "setq") {
64         parsingContext_ = Setq;
65         return expFact.createSetq();
66     } else if (name == "sync") {

```

abr 12, 16 16:39

parser.cpp

Page 2/3

```

67     parsingContext_ = Sync;
68     return expFact.createSync();
69 } else if (name == "defun") {
70     parsingContext_ = Defun;
71     return expFact.createDefun();
72 } else {
73     Expression* e = globalContext_.getExpression(name);
74     if (e != NULL) {
75         parsingContext_ = Fun;
76     }
77     return e;
78 }
79 }
80
81 Atom* Parser::getAtomInstance_(const std::string s) {
82     AtomFactory& atomFact = globalContext_.getAtomFactory();
83
84     if (isNumber(s)) {
85         NumericAtom* a = atomFact.createNumeric();
86         a->setValue(s);
87         return a;
88     }
89
90     if (parsingContext_ != Setq) {
91         Atom* variable = globalContext_.getAtom(s);
92         if (variable != NULL) return variable;
93     }
94
95     StringAtom* a = atomFact.createString();
96     a->setValue(s);
97     return a;
98 }
99
100
101
102 Expression* Parser::functionExpression_(Expression* r, const std::string s) {
103     std::string newExpr = ((DefunExpression*) r)->getExpressionString(s);
104     parsingContext_ = CommonExpression;
105     return parseExpression_(newExpr);
106 }
107
108
109 Expression* Parser::parseExpression_(const std::string s) {
110     if (!isExpression_(s)) return NULL;
111
112     std::istringstream iss(s.substr(1, s.size() - 2));
113
114     std::string expressionName;
115     iss >> expressionName;
116
117     Expression* result = getExpressionInstance_(expressionName);
118     if (result == NULL) return NULL;
119
120     std::string token;
121
122     while (iss >> token) {
123         if (token[0] == '(') {
124             std::string tokenAux;
125             int bracketCount = 0;
126
127             bracketCount += count(token.begin(), token.end(), '(');
128             bracketCount -= count(token.begin(), token.end(), ')');
129
130             while (!iss.eof() ^ bracketCount) {
131                 iss >> tokenAux;
132             }

```

abr 12, 16 16:39

parser.cpp

Page 3/3

```

133     bracketCount += count(tokenAux.begin(), tokenAux.end(), '(');
134     bracketCount -= count(tokenAux.begin(), tokenAux.end(), ')');
135
136     token.append(" ");
137     token.append(tokenAux);
138 }
139
140 if (parsingContext_ == Defun) {
141     if (token == "(ENV)") {
142         continue;
143     } else {
144         ((DefunExpression*) result)->setExpressionString(token);
145         if (!iss.eof()) return NULL;
146         return result;
147     }
148 }
149
150 if (parsingContext_ == Fun) {
151     return functionExpression_(result, token);
152 }
153
154 Expression* e = parseExpression_(token);
155 if (e == NULL) return NULL;
156
157 result->addArgument(e);
158 } else {
159     if (parsingContext_ == Fun) {
160         return functionExpression_(result, token);
161     }
162
163     Atom* atom = getAtomInstance_(token);
164
165     result->addArgument(atom);
166 }
167 token.clear();
168 }
169
170 return result;
171 }

```

abr 12, 16 16:39

main.cpp

Page 1/1

```
1
2 #include <iostream>
3
4 #include "interpreter.h"
5
6
7 int main(int argc, char const *argv[]) {
8     if (argc > 1) {
9         std::cout << "ERROR: argumentos" << std::endl;
10        return 1;
11    }
12
13    Reader r;
14    Interpreter i(r);
15
16    if (i.run()) return 2;
17
18    return 0;
19 }
```

abr 12, 16 16:39

interpreter.h

Page 1/1

```
1 #ifndef __INTERPRETER_H__
2 #define __INTERPRETER_H__
3
4 #include "expressions.h"
5 #include "parser.h"
6
7 #include <string>
8
9 class Reader {
10 public:
11     virtual std::string nextLine();
12 };
13
14
15 class Interpreter {
16     Reader reader_;
17
18 public:
19     explicit Interpreter(const Reader r);
20
21     int run();
22 };
23
24
25 #endif
```

abr 12, 16 16:39

interpreter.cpp

Page 1/1

```

1
2 #include <string>
3 #include <iostream>
4
5 #include "interpreter.h"
6
7
8 std::string Reader::nextLine() {
9     std::string buff;
10    getline(std::cin, buff);
11
12    // Remuevo trailing spaces
13    size_t endpos = buff.find_last_not_of(" \t");
14    if (std::string::npos != endpos) {
15        buff = buff.substr(0, endpos + 1);
16    }
17    return buff;
18 }
19
20
21 Interpreter::Interpreter(const Reader r) : reader_(r) {
22 }
23
24 int Interpreter::run() {
25     Context globalContext;
26     Parser p(globalContext);
27
28     std::string s = reader_.nextLine();
29     while (s.size()) {
30         Expression* e = p.parse(s);
31         if (e == NULL) {
32             std::cout << "ERROR: " << s << std::endl;
33             return 1;
34         }
35
36         if (p.getParsingContext() == Sync) {
37             e->eval(globalContext);
38         } else {
39             globalContext.runInThread(e);
40         }
41         // Context c;
42         s = reader_.nextLine();
43     }
44     return 0;
45 }
46

```

abr 12, 16 16:39

factories.h

Page 1/2

```

1 #ifndef __EXPRESSION_FACTORY_H
2 #define __EXPRESSION_FACTORY_H
3
4 #include <map>
5 #include <string>
6 #include <vector>
7
8 #include "expressions.h"
9 #include "atoms.h"
10 #include "thread.h"
11
12
13 class Context;
14
15
16 template <class T>
17 class Factory {
18     std::vector<T*> instances_;
19
20 public:
21     template <class U>
22     U* createObject() {
23         U* var = new U();
24         instances_.push_back((U*) var);
25         return var;
26     }
27
28     virtual ~Factory() {
29         for (size_t i = 0; i < instances_.size(); ++i) {
30             delete instances_[i];
31         }
32     };
33 };
34
35
36 class ExpressionFactory : private Factory<Expression> {
37 public:
38     PrintExpression* createPrint();
39     SumExpression* createSum();
40     DiffExpression* createDiff();
41     MulExpression* createMul();
42     DivExpression* createDiv();
43     EqualExpression* createEqual();
44     LesserExpression* createLesser();
45     GreaterExpression* createGreater();
46     ListExpression* createList();
47     CarExpression* createCar();
48     CdrExpression* createCdr();
49     AppendExpression* createAppend();
50     IfExpression* createIf();
51     SetqExpression* createSetq();
52     SyncExpression* createSync();
53     DefunExpression* createDefun();
54 };
55
56
57 class AtomFactory : private Factory<Atom> {
58 public:
59     StringAtom* createString();
60     NumericAtom* createNumeric();
61     ListAtom* createList();
62 };
63
64
65 class ExpressionRunner : public Thread {
66     Context* c_;
67

```

abr 12, 16 16:39

factories.h

Page 2/2

```

67 Expression* e_;
68 public:
69     ExpressionRunner();
70
71     void setParameters(Context* c, Expression* e);
72     virtual void run();
73 };
74
75
76 class ExpressionRunnerFactory : private Factory<Thread> {
77 public:
78     ExpressionRunner* createRunner();
79 };
80
81
82 class Context {
83     ExpressionFactory expressionFactory_;
84     AtomFactory atomFactory_;
85     ExpressionRunnerFactory runnerFactory_;
86
87     std::map<std::string, Atom*> atoms_;
88     std::map<std::string, Expression*> expressions_;
89     std::vector<ExpressionRunner*> threads_;
90
91 public:
92     ExpressionFactory& getExpressionFactory();
93     AtomFactory& getAtomFactory();
94
95     void setAtom(std::string key, Atom* value);
96     Atom* getAtom(const std::string& key);
97
98     void setExpression(std::string key, Expression* value);
99     Expression* getExpression(const std::string& key);
100
101     void runInThread(Expression* e);
102     void joinThreads();
103
104     ~Context();
105 };
106
107
108 #endif

```

abr 12, 16 16:39

factories.cpp

Page 1/3

```

1
2 #include <map>
3 #include <string>
4 #include <vector>
5 #include <iostream>
6
7 #include "factories.h"
8
9
10 PrintExpression* ExpressionFactory::createPrint() {
11     return createObject<PrintExpression>();
12 }
13
14 SumExpression* ExpressionFactory::createSum() {
15     return createObject<SumExpression>();
16 }
17
18 DiffExpression* ExpressionFactory::createDiff() {
19     return createObject<DiffExpression>();
20 }
21
22 MulExpression* ExpressionFactory::createMul() {
23     return createObject<MulExpression>();
24 }
25
26 DivExpression* ExpressionFactory::createDiv() {
27     return createObject<DivExpression>();
28 }
29
30 EqualExpression* ExpressionFactory::createEqual() {
31     return createObject<EqualExpression>();
32 }
33
34 LesserExpression* ExpressionFactory::createLesser() {
35     return createObject<LesserExpression>();
36 }
37
38 GreaterExpression* ExpressionFactory::createGreater() {
39     return createObject<GreaterExpression>();
40 }
41
42 ListExpression* ExpressionFactory::createList() {
43     return createObject<ListExpression>();
44 }
45
46 CarExpression* ExpressionFactory::createCar() {
47     return createObject<CarExpression>();
48 }
49
50 CdrExpression* ExpressionFactory::createCdr() {
51     return createObject<CdrExpression>();
52 }
53
54 AppendExpression* ExpressionFactory::createAppend() {
55     return createObject<AppendExpression>();
56 }
57
58 IfExpression* ExpressionFactory::createIf() {
59     return createObject<IfExpression>();
60 }
61
62 SetqExpression* ExpressionFactory::createSetq() {
63     return createObject<SetqExpression>();
64 }
65
66 SyncExpression* ExpressionFactory::createSync() {

```

abr 12, 16 16:39

factories.cpp

Page 2/3

```

67     return createObject<SyncExpression>();
68 }
69
70 DefunExpression* ExpressionFactory::createDefun() {
71     return createObject<DefunExpression>();
72 }
73
74
75 StringAtom* AtomFactory::createString() {
76     return createObject<StringAtom>();
77 }
78
79 NumericAtom* AtomFactory::createNumeric() {
80     return createObject<NumericAtom>();
81 }
82
83 ListAtom* AtomFactory::createList() {
84     return createObject<ListAtom>();
85 }
86
87
88 ExpressionRunner::ExpressionRunner() : c_(NULL), e_(NULL) {
89 }
90
91 void ExpressionRunner::setParameters(Context* c, Expression* e) {
92     c_ = c;
93     e_ = e;
94 }
95
96 void ExpressionRunner::run() {
97     e_>eval(*c_);
98 }
99
100
101 ExpressionRunner* ExpressionRunnerFactory::createRunner() {
102     return createObject<ExpressionRunner>();
103 }
104
105
106 ExpressionFactory& Context::getExpressionFactory() {
107     return expressionFactory_;
108 }
109
110 AtomFactory& Context::getAtomFactory() {
111     return atomFactory_;
112 }
113
114 void Context::setAtom(std::string key, Atom* value) {
115     atoms_[key] = value;
116 }
117
118 Atom* Context::getAtom(const std::string& key) {
119     std::map<std::string, Atom*>::iterator it = atoms_.find(key);
120     if (it == atoms_.end()) return NULL;
121
122     return (*it).second;
123 }
124
125 void Context::setExpression(std::string key, Expression* value) {
126     expressions_[key] = value;
127 }
128
129 Expression* Context::getExpression(const std::string& key) {
130     std::map<std::string, Expression*>::iterator it = expressions_.find(key);
131     if (it == expressions_.end()) return NULL;
132

```

abr 12, 16 16:39

factories.cpp

Page 3/3

```

133     return (*it).second;
134 }
135
136 void Context::runInThread(Expression* e) {
137     ExpressionRunner* er = runnerFactory_.createRunner();
138     er->setParameters(this, e);
139     threads_.push_back(er);
140     er->start();
141 }
142
143 void Context::joinThreads() {
144     std::vector<ExpressionRunner*>::iterator it = threads_.begin();
145     for (; it != threads_.end(); ) {
146         (*it)->join();
147         it = threads_.erase(it);
148     }
149 }
150
151 Context::~Context() {
152     joinThreads();
153 }

```

abr 12, 16 16:39

expressions.h

Page 1/3

```

1  #ifndef __LIST_EXPRESSIONS_H__
2  #define __LIST_EXPRESSIONS_H__
3
4  #include <deque>
5  #include <vector>
6  #include <string>
7
8  #include "atoms.h"
9
10
11 class Context;
12 class Expression;
13
14
15 class Argument {
16     Atom* a_;
17     Expression* e_;
18     bool isAtom_;
19
20 public:
21     explicit Argument(Atom* a);
22     explicit Argument(Expression* e);
23     Argument();
24
25     void setAtom(Atom* a);
26     void setExpression(Expression* e);
27
28     Atom* getAtom();
29     Expression* getExpression();
30     bool isAtom();
31 };
32
33
34 class Expression {
35     std::deque<Argument*> args_;
36
37 public:
38     void addArgument(Expression* e);
39     void addArgument(Atom* a);
40
41     Atom* getArgumentValue(Argument* a, Context& c);
42     std::deque<Argument*> getArguments();
43
44     ListAtom* createNil(Context &c);
45
46     virtual Atom* eval(Context& c) = 0;
47     virtual ~Expression();
48 };
49
50
51 class PrintExpression : public Expression {
52 public:
53     virtual Atom* eval(Context& c);
54 };
55
56
57 class MathExpression : public Expression {
58 public:
59     virtual int operation(int a, int v) = 0;
60     virtual Atom* eval(Context& c);
61 };
62
63 class SumExpression : public MathExpression {
64     virtual int operation(int a, int b);
65 };
66

```

abr 12, 16 16:39

expressions.h

Page 2/3

```

67 class DiffExpression : public MathExpression {
68     virtual int operation(int a, int b);
69 };
70
71 class MulExpression : public MathExpression {
72     virtual int operation(int a, int b);
73 };
74
75 class DivExpression : public MathExpression {
76     virtual int operation(int a, int b);
77 };
78
79
80 class ListExpression : public Expression {
81 public:
82     virtual Atom* eval(Context& c);
83 };
84
85
86 class CarExpression : public Expression {
87 public:
88     virtual Atom* extractAtom(std::vector<Atom*>& values, Context& c);
89     virtual Atom* eval(Context& c);
90 };
91
92
93 class EqualExpression : public Expression {
94 public:
95     virtual Atom* eval(Context& c);
96     virtual bool compare(const std::string& a, const std::string& b);
97 };
98
99
100 class LesserExpression : public EqualExpression {
101 public:
102     virtual bool compare(const std::string& a, const std::string& b);
103 };
104
105
106 class GreaterExpression : public EqualExpression {
107 public:
108     virtual bool compare(const std::string& a, const std::string& b);
109 };
110
111
112 class CdrExpression : public CarExpression {
113 public:
114     virtual Atom* extractAtom(std::vector<Atom*>& values, Context& c);
115 };
116
117
118 class AppendExpression : public Expression {
119 public:
120     virtual Atom* eval(Context& c);
121 };
122
123
124 class IfExpression : public Expression {
125 public:
126     virtual Atom* eval(Context& c);
127 };
128
129
130 class SetqExpression : public Expression {
131 public:
132     virtual Atom* eval(Context& c);

```


abr 12, 16 16:39

expressions.h

Page 3/3

```

133 };
134
135 class SyncExpression : public Expression {
136 public:
137     virtual Atom* eval(Context& c);
138 };
139
140 class DefunExpression : public Expression {
141     std::string expression_;
142
143 public:
144     virtual Atom* eval(Context& c);
145
146     void setExpressionString(std::string s);
147     std::string getExpressionString(std::string parameters);
148 };
149
150 #endif

```

abr 12, 16 16:39

expressions.cpp

Page 1/5

```

1
2 #include <deque>
3 #include <iostream>
4 #include <sstream>
5 #include <string>
6 #include <vector>
7
8 #include "expressions.h"
9 #include "factories.h"
10
11
12 Argument::Argument(Atom* a): a_(a), isAtom_(true) {
13 }
14
15 Argument::Argument(Expression* e): e_(e), isAtom_(false) {
16 }
17
18 Argument::Argument(): isAtom_(false) {
19 }
20
21 Atom* Argument::getAtom() {
22     return a_;
23 }
24
25 Expression* Argument::getExpression() {
26     return e_;
27 }
28
29 void Argument::setAtom(Atom* a) {
30     a_ = a;
31     isAtom_ = true;
32 }
33
34 void Argument::setExpression(Expression* e) {
35     e_ = e;
36     isAtom_ = false;
37 }
38
39 bool Argument::isAtom() {
40     return isAtom_;
41 }
42
43
44 void Expression::addArgument(Expression* e) {
45     Argument* arg = new Argument(e);
46     args_.push_back(arg);
47 }
48
49 void Expression::addArgument(Atom* a) {
50     Argument* arg = new Argument(a);
51     args_.push_back(arg);
52 }
53
54 std::deque<Argument*>& Expression::getArguments() {
55     return args_;
56 }
57
58 Atom* Expression::getArgumentValue(Argument* a, Context& c) {
59     if (a->isAtom()) {
60         return a->getAtom();
61     } else {
62         Expression* e = a->getExpression();
63         return e->eval(c);
64     }
65 }
66

```

abr 12, 16 16:39

expressions.cpp

Page 2/5

```

67 ListAtom* Expression::createNil(Context& c) {
68     return c.getAtomFactory().createList();
69 }
70
71 Expression::~Expression() {
72     std::deque<Argument*>::iterator it = args_.begin();
73     for (; it != args_.end(); ++it) {
74         delete *it;
75         it = args_.erase(it);
76     }
77 }
78
79
80 Atom* PrintExpression::eval(Context& c) {
81     std::deque<Argument*> args = getArguments();
82
83     std::deque<Argument*>::iterator it = args.begin();
84     for (; it != args.end(); ++it) {
85         Atom* a = getArgumentValue(*it, c);
86         std::cout << a->getValue();
87         if (it != args.end() - 1) {
88             std::cout << " ";
89         }
90     }
91     std::cout << std::endl;
92
93     return createNil(c);
94 }
95
96
97 Atom* MathExpression::eval(Context& c) {
98     std::deque<Argument*> args = getArguments();
99
100     std::deque<Argument*>::iterator it = args.begin();
101
102     int value = ((NumericAtom*) getArgumentValue(*it, c))>getNumericValue();
103
104     for (++it; it != args.end(); ++it) {
105         NumericAtom* a = (NumericAtom*) getArgumentValue(*it, c);
106         value = operation(value, a->getNumericValue());
107     }
108
109     std::stringstream ss;
110     ss << value;
111     NumericAtom* result = c.getAtomFactory().createNumeric();
112     result->setValue(ss.str());
113     return result;
114 }
115
116
117 int SumExpression::operation(int a, int b) {
118     return a + b;
119 }
120
121
122 int DiffExpression::operation(int a, int b) {
123     return a - b;
124 }
125
126
127 int MulExpression::operation(int a, int b) {
128     return a * b;
129 }
130
131
132 int DivExpression::operation(int a, int b) {

```

abr 12, 16 16:39

expressions.cpp

Page 3/5

```

133     return a / b;
134 }
135
136
137 Atom* EqualExpression::eval(Context& c) {
138     std::deque<Argument*> args = getArguments();
139
140     if (!args.size()) return createNil(c);
141
142     std::deque<Argument*>::iterator it = args.begin();
143
144     Atom* a = getArgumentValue(*it, c);
145     Atom* b = getArgumentValue(*(it + 1), c);
146     if (compare(a->getValue(), b->getValue())) {
147         NumericAtom* result = c.getAtomFactory().createNumeric();
148         result->setValue("1");
149         return result;
150     } else {
151         return createNil(c);
152     }
153 }
154
155 bool EqualExpression::compare(const std::string& a, const std::string& b) {
156     return a == b;
157 }
158
159
160 bool LesserExpression::compare(const std::string& a, const std::string& b) {
161     return a < b;
162 }
163
164
165 bool GreaterExpression::compare(const std::string& a, const std::string& b) {
166     return a > b;
167 }
168
169
170 Atom* ListExpression::eval(Context& c) {
171     ListAtom* result = createNil(c);
172
173     std::deque<Argument*> args = getArguments();
174
175     std::deque<Argument*>::iterator it = args.begin();
176
177     for (; it != args.end(); ++it) {
178         Atom* a = getArgumentValue(*it, c);
179         result->addValue(a);
180     }
181
182     return result;
183 }
184
185
186 Atom* CarExpression::eval(Context& c) {
187     std::deque<Argument*> args = getArguments();
188
189     if (!args.size()) return createNil(c);
190
191     ListAtom* list = (ListAtom*) getArgumentValue(args.front(), c);
192     std::vector<Atom*>& values = list->getValues();
193     if (!values.size()) return createNil(c);
194
195     return extractAtom(values, c);
196 }
197
198

```

abr 12, 16 16:39

expressions.cpp

Page 4/5

```

199 Atom* CarExpression::extractAtom(std::vector<Atom*>& values, Context& c) {
200     return values[0];
201 }
202
203
204 Atom* CdrExpression::extractAtom(std::vector<Atom*>& values, Context& c) {
205     ListAtom* result = createNil(c);
206
207     if (values.size() < 2) return result;
208
209     std::vector<Atom*>::iterator it = values.begin() + 1;
210     for (; it != values.end(); ++it) {
211         result->addValue(*it);
212     }
213
214     return result;
215 }
216
217
218 Atom* AppendExpression::eval(Context& c) {
219     ListAtom* result = createNil(c);
220
221     std::deque<Argument*> args = getArguments();
222     std::deque<Argument*>::iterator it = args.begin();
223     for (; it != args.end(); ++it) {
224         ListAtom* atom = (ListAtom*) getArgumentValue(*it, c);
225
226         std::vector<Atom*>& values = atom->getValues();
227         for (size_t i = 0; i < values.size(); ++i) {
228             result->addValue(values[i]);
229         }
230     }
231
232     return result;
233 }
234
235
236 Atom* IfExpression::eval(Context& c) {
237     std::deque<Argument*> args = getArguments();
238     std::deque<Argument*>::iterator it = args.begin();
239     Atom* r;
240     if (getArgumentValue(*it, c)->isTrue()) {
241         r = getArgumentValue(*(it + 1), c);
242     } else {
243         r = getArgumentValue(*(it + 2), c);
244     }
245     return r;
246 }
247
248
249
250 Atom* SetqExpression::eval(Context& c) {
251     std::deque<Argument*> args = getArguments();
252     std::deque<Argument*>::iterator it = args.begin();
253
254     std::string key = getArgumentValue(*it, c)->getValue();
255     c.setAtom(key, getArgumentValue(*(it + 1), c));
256
257     return createNil(c);
258 }
259
260
261 Atom* SyncExpression::eval(Context& c) {
262     c.joinThreads();
263     return createNil(c);
264 }

```

abr 12, 16 16:39

expressions.cpp

Page 5/5

```

265
266
267 Atom* DefunExpression::eval(Context& c) {
268     std::deque<Argument*> args = getArguments();
269     std::deque<Argument*>::iterator it = args.begin();
270
271     std::string key = getArgumentValue(*it, c)->getValue();
272
273     c.setExpression(key, this);
274
275     return createNil(c);
276 }
277
278 void DefunExpression::setExpressionString(std::string s) {
279     expression_ = s;
280 }
281
282 std::string DefunExpression::getExpressionString(std::string param) {
283     std::string result = expression_;
284     size_t index = 0;
285
286     while (true) {
287         index = result.find("ENV", index);
288         if (index == std::string::npos) break;
289         result.replace(index, 3, param, 0, param.size());
290
291         index += 3;
292     }
293
294     return result;
295 }

```

abr 12, 16 16:39

atoms.h

Page 1/1

```

1  #ifndef __LISP_ATOMS_H__
2  #define __LISP_ATOMS_H__
3
4  #include <string>
5  #include <vector>
6
7
8  class Atom {
9  public:
10     virtual bool isTrue() { return true; }
11
12     virtual std::string getValue() = 0;
13
14     virtual ~Atom() {}
15 };
16
17 class StringAtom : public Atom {
18     std::string value_;
19 public:
20     void setValue(const std::string s) { value_ = s; }
21
22     virtual std::string getValue() { return value_; }
23 };
24
25 class NumericAtom : public Atom {
26     int value_;
27 public:
28     void setValue(std::string s);
29     virtual std::string getValue();
30     int getNumericValue();
31 };
32
33
34 class ListAtom : public Atom {
35     std::vector<Atom*> values;
36 public:
37     void setValue(std::string s) {}
38
39     virtual bool isTrue();
40
41     void addValue(Atom* value);
42     virtual std::string getValue();
43
44     std::vector<Atom*>& getValues();
45 };
46
47
48 #endif

```

abr 12, 16 16:39

atoms.cpp

Page 1/1

```

1
2  #include <string>
3  #include <sstream>
4  #include <vector>
5
6  #include "atoms.h"
7
8
9  void NumericAtom::setValue(std::string s) {
10     std::stringstream ss(s);
11     ss >> value_;
12 }
13
14 std::string NumericAtom::getValue() {
15     std::stringstream ss;
16     ss << value_;
17     return ss.str();
18 }
19
20 int NumericAtom::getNumericValue() {
21     return value_;
22 }
23
24
25
26 bool ListAtom::isTrue() {
27     return values.size();
28 }
29
30 void ListAtom::addValue(Atom* value) {
31     values.push_back(value);
32 }
33
34 std::string ListAtom::getValue() {
35     std::stringstream ss;
36     ss << "(";
37     for (size_t i = 0; i < values.size(); ++i) {
38         ss << (values[i] ->getValue());
39         if (i != values.size() - 1) ss << " ";
40     }
41     ss << ")";
42
43     return ss.str();
44 }
45
46 std::vector<Atom*>& ListAtom::getValues() {
47     return values;
48 }

```

abr 12, 16 16:39

Table of Content

Page 1/1

1	Table of Contents					
2	1	<i>thread.h</i>	sheets	1 to	1 (1) pages 1- 1 29 lines
3	2	<i>thread.cpp</i>	sheets	1 to	1 (1) pages 2- 2 17 lines
4	3	<i>parser.h</i>	sheets	2 to	2 (1) pages 3- 3 39 lines
5	4	<i>parser.cpp</i>	sheets	2 to	3 (2) pages 4- 6 172 lines
6	5	<i>main.cpp</i>	sheets	4 to	4 (1) pages 7- 7 20 lines
7	6	<i>interpreter.h</i>	sheets	4 to	4 (1) pages 8- 8 26 lines
8	7	<i>interpreter.cpp</i>	sheets	5 to	5 (1) pages 9- 9 47 lines
9	8	<i>factories.h</i>	sheets	5 to	6 (2) pages 10- 11 109 lines
10	9	<i>factories.cpp</i>	sheets	6 to	7 (2) pages 12- 14 154 lines
11	10	<i>expressions.h</i>	sheets	8 to	9 (2) pages 15- 17 151 lines
12	11	<i>expressions.cpp</i>	sheets	9 to	11 (3) pages 18- 22 296 lines
13	12	<i>atoms.h</i>	sheets	12 to	12 (1) pages 23- 23 49 lines
14	13	<i>atoms.cpp</i>	sheets	12 to	12 (1) pages 24- 24 49 lines