```cpp
#ifndef __THREADS_H__
#define __THREADS_H__

#include <pthread.h>


class Thread {
private:
    pthread_t thread;

    static void* runner(void *data);

public:
    Thread() {}

    void start();
    void join();

    virtual void run() = 0;
    virtual ~Thread() {}

private:
    Thread(const Thread&);
    Thread& operator=(const Thread&);
};


#endif
```

```cpp

#include "thread.h"

void* Thread::runner(void *data) {
  Thread* self = (Thread*) data;
  self->run();
  return NULL;
}

void Thread::start() {
  pthread_create(&thread, NULL, Thread::runner, this);
}

void Thread::join() {
  pthread_join(thread, NULL);
}
```

```
#ifndef __PARSER_H__
#define __PARSER_H__

#include <string>

#include "expressions.h"
#include "atoms.h"
#include "factories.h"

enum ParsingContext {
  CommonExpression,
  Setq,
  Sync,
  Defun,
  Fun
};

class Parser {
  Context& globalContext_;
  ParsingContext parsingContext_;

  Expression* getExpressionInstance_(std::string name);

  Atom* getAtomInstance_(std::string s);

  Expression* functionExpression_(Expression* r, const std::string s);
  Expression* parseExpression_(const std::string s);

public:
  explicit Parser(Context& globalContext);

  Expression* parse(const std::string s);
  ParsingContext getParsingContext();
};



#endif
```

```

#include <algorithm>
#include <string>
#include <iostream>
#include <sstream>

#include "parser.h"


bool isNumber(const std::string& s) {
    std::string::const_iterator it = s.begin();
    while (it ≠ s.end() ∧ isdigit(*it)) ++it;
    return ¬s.empty() ∧ it ≡ s.end();
}

bool isExpression_(const std::string& s) {
  return (s[0] ≡ ’(’ ∧ s[s.size() − 1] ≡ ’)’);
}


Parser::Parser(Context& globalContext) : globalContext_(globalContext) {
}

Expression* Parser::parse(const std::string s) {
  parsingContext_ = CommonExpression;
  return parseExpression_(s);
}

ParsingContext Parser::getParsingContext() {
  return parsingContext_;
}


Expression* Parser::getExpressionInstance_(const std::string name) {
  ExpressionFactory& expFact = globalContext_.getExpressionFactory();

  if (name ≡ "print") {
    return expFact.createPrint();
  } else if (name ≡ "+") {
    return expFact.createSum();
  } else if (name ≡ "−") {
    return expFact.createDiff();
  } else if (name ≡ "*") {
    return expFact.createMul();
  } else if (name ≡ "/") {
    return expFact.createDiv();
  } else if (name ≡ "=") {
    return expFact.createEqual();
  } else if (name ≡ "<") {
    return expFact.createLesser();
  } else if (name ≡ ">") {
    return expFact.createGreater();
  } else if (name ≡ "list") {
    return expFact.createList();
  } else if (name ≡ "car") {
    return expFact.createCar();
  } else if (name ≡ "cdr") {
    return expFact.createCdr();
  } else if (name ≡ "append") {
    return expFact.createAppend();
  } else if (name ≡ "if") {
    return expFact.createIf();
  } else if (name ≡ "setq") {
    parsingContext_ = Setq;
    return expFact.createSetq();
  } else if (name ≡ "sync") {
    parsingContext_ = Sync;
    return expFact.createSync();
  } else if (name ≡ "defun") {
    parsingContext_ = Defun;
    return expFact.createDefun();
  } else {
    Expression* e = globalContext_.getExpression(name);
    if (e ≠ NULL) {
      parsingContext_ = Fun;
    }
    return e;
  }
}


Atom* Parser::getAtomInstance_(const std::string s) {
  AtomFactory& atomFact = globalContext_.getAtomFactory();

  if (isNumber(s)) {
    NumericAtom* a = atomFact.createNumeric();
    a→setValue(s);
    return a;
  }

  if (parsingContext_ ≠ Setq) {
    Atom* variable = globalContext_.getAtom(s);
    if (variable ≠ NULL) return variable;
  }

  StringAtom* a = atomFact.createString();
  a→setValue(s);
  return a;
}


Expression* Parser::functionExpression_(Expression* r, const std::string s) {
  std::string newExpr = ((DefunExpression*) r)→getExpressionString(s);
  parsingContext_ = CommonExpression;
  return parseExpression_(newExpr);
}


Expression* Parser::parseExpression_(const std::string s) {
  if (¬isExpression_(s)) return NULL;

  std::istringstream iss(s.substr(1, s.size() − 2));

  std::string expressionName;
  iss >> expressionName;

  Expression* result = getExpressionInstance_(expressionName);
  if (result ≡ NULL) return NULL;

  std::string token;

  while (iss >> token) {
    if (token[0] ≡ ’(’) {
      std::string tokenAux;
      int bracketCount = 0;

      bracketCount += count(token.begin(), token.end(), ’(’);
      bracketCount −= count(token.begin(), token.end(), ’)’);

      while (¬iss.eof() ∧ bracketCount) {
        iss >> tokenAux;

```

```
        bracketCount += count(tokenAux.begin(), tokenAux.end(), ’(’);
        bracketCount −= count(tokenAux.begin(), tokenAux.end(), ’)’);

        token.append(" ");
        token.append(tokenAux);
      }

      if (parsingContext_ ≡ Defun) {
        if (token ≡ "(ENV)") {
          continue;
        } else {
          ((DefunExpression*) result)→setExpressionString(token);
          if (¬iss.eof()) return NULL;
          return result;
        }
      }

      if (parsingContext_ ≡ Fun) {
        return functionExpression_(result, token);
      }

      Expression* e = parseExpression_(token);
      if (e ≡ NULL) return NULL;

      result→addArgument(e);
    } else {
      if (parsingContext_ ≡ Fun) {
        return functionExpression_(result, token);
      }

      Atom* atom = getAtomInstance_(token);

      result→addArgument(atom);
    }
    token.clear();
  }

  return result;
}
```

abr 12, 16 16:39 **main.cpp** Page 1/1

```

#include <iostream>

#include "interpreter.h"


int main(int argc, char const *argv[]) {
  if (argc > 1) {
    std::cout << "ERROR: argumentos" << std::endl;
    return 1;
  }

  Reader r;
  Interpreter i(r);

  if (i.run()) return 2;

  return 0;
}
```

abr 12, 16 16:39 **interpreter.h** Page 1/1

```
#ifndef __INTERPRETER_H__
#define __INTERPRETER_H__

#include "expressions.h"
#include "parser.h"

#include <string>

class Reader {
public:
  virtual std::string nextLine();
};


class Interpreter {
  Reader reader_;

public:
  explicit Interpreter(const Reader r);

  int run();
};


#endif
```

```cpp

#include <string>
#include <iostream>

#include "interpreter.h"


std::string Reader::nextLine() {
  std::string buff;
  getline(std::cin, buff);

  // Remuevo trailing spaces
  size_t endpos = buff.find_last_not_of(" \t");
  if (std::string::npos ≠ endpos) {
      buff = buff.substr(0, endpos + 1);
  }
  return buff;
}


Interpreter::Interpreter(const Reader r) : reader_(r) {
}

int Interpreter::run() {
    Context globalContext;
    Parser p(globalContext);

    std::string s = reader_.nextLine();
    while (s.size()) {
      Expression* e = p.parse(s);
      if (e ≡ NULL) {
        std::cout << "ERROR: " << s << std::endl;
        return 1;
      }

      if (p.getParsingContext() ≡ Sync) {
        e→eval(globalContext);
      } else {
        globalContext.runInThread(e);
      }
      // Context c;
      s = reader_.nextLine();
    }
    return 0;
  }

```

```cpp
#ifndef __EXPRESSION_FACTORY_H
#define __EXPRESSION_FACTORY_H

#include <map>
#include <string>
#include <vector>

#include "expressions.h"
#include "atoms.h"
#include "thread.h"


class Context;


template <class T>
class Factory {
  std::vector<T*> instances_;

public:
  template <class U>
  U* createObject() {
    U* var = new U();
    instances_.push_back((U*) var);
    return var;
  }

  virtual ~Factory() {
    for (size_t i = 0; i < instances_.size(); ++i) {
      delete instances_[i];
    }
  }
};


class ExpressionFactory : private Factory<Expression> {
public:
  PrintExpression* createPrint();
  SumExpression* createSum();
  DiffExpression* createDiff();
  MulExpression* createMul();
  DivExpression* createDiv();
  EqualExpression* createEqual();
  LesserExpression* createLesser();
  GreaterExpression* createGreater();
  ListExpression* createList();
  CarExpression* createCar();
  CdrExpression* createCdr();
  AppendExpression* createAppend();
  IfExpression* createIf();
  SetqExpression* createSetq();
  SyncExpression* createSync();
  DefunExpression* createDefun();
};


class AtomFactory : private Factory<Atom> {
public:
  StringAtom* createString();
  NumericAtom* createNumeric();
  ListAtom* createList();
};


class ExpressionRunner : public Thread {
Context* c_;
Expression* e_;
public:
  ExpressionRunner();

  void setParameters(Context* c, Expression* e);
  virtual void run();
};


class ExpressionRunnerFactory : private Factory<Thread> {
public:
  ExpressionRunner* createRunner();
};


class Context {
ExpressionFactory expressionFactory_;
AtomFactory atomFactory_;
ExpressionRunnerFactory runnerFactory_;

std::map<std::string,Atom*> atoms_;
std::map<std::string,Expression*> expressions_;
std::vector<ExpressionRunner*> threads_;

public:
  ExpressionFactory& getExpressionFactory();
  AtomFactory& getAtomFactory();

  void setAtom(std::string key, Atom* value);
  Atom* getAtom(const std::string& key);

  void setExpression(std::string key, Expression* value);
  Expression* getExpression(const std::string& key);

  void runInThread(Expression* e);
  void joinThreads();

  ~Context();
};


#endif
```

```

#include <map>
#include <string>
#include <vector>
#include <iostream>

#include "factories.h"


PrintExpression* ExpressionFactory::createPrint() {
  return createObject<PrintExpression>();
}

SumExpression* ExpressionFactory::createSum() {
  return createObject<SumExpression>();
}

DiffExpression* ExpressionFactory::createDiff() {
  return createObject<DiffExpression>();
}

MulExpression* ExpressionFactory::createMul() {
  return createObject<MulExpression>();
}

DivExpression* ExpressionFactory::createDiv() {
  return createObject<DivExpression>();
}

EqualExpression* ExpressionFactory::createEqual() {
  return createObject<EqualExpression>();
}

LesserExpression* ExpressionFactory::createLesser() {
  return createObject<LesserExpression>();
}

GreaterExpression* ExpressionFactory::createGreater() {
  return createObject<GreaterExpression>();
}

ListExpression* ExpressionFactory::createList() {
  return createObject<ListExpression>();
}

CarExpression* ExpressionFactory::createCar() {
  return createObject<CarExpression>();
}

CdrExpression* ExpressionFactory::createCdr() {
  return createObject<CdrExpression>();
}

AppendExpression* ExpressionFactory::createAppend() {
  return createObject<AppendExpression>();
}

IfExpression* ExpressionFactory::createIf() {
  return createObject<IfExpression>();
}

SetqExpression* ExpressionFactory::createSetq() {
  return createObject<SetqExpression>();
}

SyncExpression* ExpressionFactory::createSync() {
  return createObject<SyncExpression>();
}

DefunExpression* ExpressionFactory::createDefun() {
  return createObject<DefunExpression>();
}


StringAtom* AtomFactory::createString() {
  return createObject<StringAtom>();
}

NumericAtom* AtomFactory::createNumeric() {
  return createObject<NumericAtom>();
}

ListAtom* AtomFactory::createList() {
  return createObject<ListAtom>();
}


ExpressionRunner::ExpressionRunner() : c_(NULL), e_(NULL) {
}

void ExpressionRunner::setParameters(Context* c, Expression* e) {
  c_ = c;
  e_ = e;
}

void ExpressionRunner::run() {
  e_→eval(*c_);
}


ExpressionRunner* ExpressionRunnerFactory::createRunner() {
  return createObject<ExpressionRunner>();
}


ExpressionFactory& Context::getExpressionFactory() {
  return expressionFactory_;
}

AtomFactory& Context::getAtomFactory() {
  return atomFactory_;
}

void Context::setAtom(std::string key, Atom* value) {
  atoms_[key] = value;
}

Atom* Context::getAtom(const std::string& key) {
  std::map<std::string,Atom*>::iterator it = atoms_.find(key);
  if (it ≡ atoms_.end()) return NULL;

  return (*it).second;
}

void Context::setExpression(std::string key, Expression* value) {
  expressions_[key] = value;
}

Expression* Context::getExpression(const std::string& key) {
  std::map<std::string,Expression*>::iterator it = expressions_.find(key);
  if (it ≡ expressions_.end()) return NULL;

```

```
  return (*it).second;
}

void Context::runInThread(Expression* e) {
  ExpressionRunner* er = runnerFactory_.createRunner();
  er→setParameters(this, e);
  threads_.push_back(er);
  er→start();
}

void Context::joinThreads() {
  std::vector<ExpressionRunner*>::iterator it = threads_.begin();
  for (; it ≠ threads_.end();) {
    (*it)→join();
    it = threads_.erase(it);
  }
}

Context::~Context() {
  joinThreads();
}
```

```cpp
#ifndef __LIST_EXPRESSIONS_H__
#define __LIST_EXPRESSIONS_H__

#include <deque>
#include <vector>
#include <string>

#include "atoms.h"


class Context;
class Expression;


class Argument {
  Atom* a_;
  Expression* e_;
  bool isAtom_;

public:
  explicit Argument(Atom* a);
  explicit Argument(Expression* e);
  Argument();

  void setAtom(Atom* a);
  void setExpression(Expression* e);

  Atom* getAtom();
  Expression* getExpression();
  bool isAtom();
};


class Expression {
std::deque<Argument*> args_;

public:
  void addArgument(Expression* e);
  void addArgument(Atom* a);

  Atom* getArgumentValue(Argument* a, Context& c);
  std::deque<Argument*>& getArguments();

  ListAtom* createNil(Context &c);

  virtual Atom* eval(Context& c) = 0;
  virtual ~Expression();
};


class PrintExpression : public Expression {
public:
  virtual Atom* eval(Context& c);
};


class MathExpression : public Expression {
public:
  virtual int operation(int a, int v) = 0;
  virtual Atom* eval(Context& c);
};

class SumExpression : public MathExpression {
  virtual int operation(int a, int b);
};

```

```cpp
class DiffExpression : public MathExpression {
  virtual int operation(int a, int b);
};

class MulExpression : public MathExpression {
  virtual int operation(int a, int b);
};

class DivExpression : public MathExpression {
  virtual int operation(int a, int b);
};


class ListExpression : public Expression {
public:
  virtual Atom* eval(Context& c);
};


class CarExpression : public Expression {
public:
  virtual Atom* extractAtom(std::vector<Atom*>& values, Context& c);
  virtual Atom* eval(Context& c);
};


class EqualExpression : public Expression {
public:
  virtual Atom* eval(Context& c);
  virtual bool compare(const std::string& a, const std::string& b);
};


class LesserExpression : public EqualExpression {
public:
  virtual bool compare(const std::string& a, const std::string& b);
};


class GreaterExpression : public EqualExpression {
public:
  virtual bool compare(const std::string& a, const std::string& b);
};


class CdrExpression : public CarExpression {
public:
  virtual Atom* extractAtom(std::vector<Atom*>& values, Context& c);
};


class AppendExpression : public Expression {
public:
  virtual Atom* eval(Context& c);
};


class IfExpression : public Expression {
public:
  virtual Atom* eval(Context& c);
};


class SetqExpression : public Expression {
public:
  virtual Atom* eval(Context& c);
};

class SyncExpression : public Expression {
public:
  virtual Atom* eval(Context& c);
};

class DefunExpression : public Expression {
std::string expression_;

public:
  virtual Atom* eval(Context& c);

  void setExpressionString(std::string s);
  std::string getExpressionString(std::string parameters);
};

#endif
```

```

#include <deque>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

#include "expressions.h"
#include "factories.h"


Argument::Argument(Atom* a): a_(a), isAtom_(true) {
}

Argument::Argument(Expression* e): e_(e), isAtom_(false) {
}

Argument::Argument(): isAtom_(false) {
}

Atom* Argument::getAtom() {
  return a_;
}

Expression* Argument::getExpression() {
  return e_;
}

void Argument::setAtom(Atom* a) {
  a_ = a;
  isAtom_ = true;
}

void Argument::setExpression(Expression* e) {
  e_ = e;
  isAtom_ = false;
}

bool Argument::isAtom() {
  return isAtom_;
}


void Expression::addArgument(Expression* e) {
  Argument* arg = new Argument(e);
  args_.push_back(arg);
}

void Expression::addArgument(Atom* a) {
  Argument* arg = new Argument(a);
  args_.push_back(arg);
}

std::deque<Argument*>& Expression::getArguments() {
  return args_;
}

Atom* Expression::getArgumentValue(Argument* a, Context& c) {
  if (a→isAtom()) {
    return a→getAtom();
  } else {
    Expression* e = a→getExpression();
    return e→eval(c);
  }
}

```

```
ListAtom* Expression::createNil(Context& c) {
  return c.getAtomFactory().createList();
}

Expression::~Expression() {
  std::deque<Argument*>::iterator it = args_.begin();
  for (; it ≠ args_.end();) {
    delete *it;
    it = args_.erase(it);
  }
}


Atom* PrintExpression::eval(Context& c) {
  std::deque<Argument*> args = getArguments();

  std::deque<Argument*>::iterator it = args.begin();
  for (; it ≠ args.end(); ++it) {
    Atom* a = getArgumentValue(*it, c);
    std::cout << a→getValue();
    if (it ≠ args.end() − 1) {
      std::cout << " ";
    }
  }
  std::cout << std::endl;

  return createNil(c);
}


Atom* MathExpression::eval(Context& c) {
  std::deque<Argument*> args = getArguments();

  std::deque<Argument*>::iterator it = args.begin();

  int value = ((NumericAtom*) getArgumentValue(*it, c))→getNumericValue();

  for (++it; it ≠ args.end(); ++it) {
    NumericAtom* a = (NumericAtom*) getArgumentValue(*it, c);
    value = operation(value, a→getNumericValue());
  }

  std::stringstream ss;
  ss << value;
  NumericAtom* result = c.getAtomFactory().createNumeric();
  result→setValue(ss.str());
  return result;
}


int SumExpression::operation(int a, int b) {
  return a + b;
}


int DiffExpression::operation(int a, int b) {
  return a − b;
}


int MulExpression::operation(int a, int b) {
  return a * b;
}


int DivExpression::operation(int a, int b) {
```

```
  return a / b;
}


Atom* EqualExpression::eval(Context& c) {
  std::deque<Argument*> args = getArguments();

  if (¬args.size()) return createNil(c);

  std::deque<Argument*>::iterator it = args.begin();

  Atom* a = getArgumentValue(*it, c);
  Atom* b = getArgumentValue(*(it + 1), c);
  if (compare(a→getValue(), b→getValue())) {
    NumericAtom* result = c.getAtomFactory().createNumeric();
    result→setValue("1");
    return result;
  } else {
    return createNil(c);
  }
}

bool EqualExpression::compare(const std::string& a, const std::string& b) {
  return a ≡ b;
}


bool LesserExpression::compare(const std::string& a, const std::string& b) {
  return a < b;
}


bool GreaterExpression::compare(const std::string& a, const std::string& b) {
  return a > b;
}


Atom* ListExpression::eval(Context& c) {
  ListAtom* result = createNil(c);

  std::deque<Argument*> args = getArguments();

  std::deque<Argument*>::iterator it = args.begin();

  for (; it ≠ args.end(); ++it) {
    Atom* a = getArgumentValue(*it, c);
    result→addValue(a);
  }

  return result;
}


Atom* CarExpression::eval(Context& c) {
  std::deque<Argument*> args = getArguments();

  if (¬args.size()) return createNil(c);

  ListAtom* list = (ListAtom*) getArgumentValue(args.front(), c);
  std::vector<Atom*>& values = list→getValues();
  if (¬values.size()) return createNil(c);

  return extractAtom(values, c);
}
```

```
Atom* CarExpression::extractAtom(std::vector<Atom*>& values, Context& c) {
  return values[0];
}


Atom* CdrExpression::extractAtom(std::vector<Atom*>& values, Context& c) {
  ListAtom* result = createNil(c);

  if (values.size() < 2) return result;

  std::vector<Atom*>::iterator it = values.begin() + 1;
  for (; it ≠ values.end(); ++it) {
    result→addValue(*it);
  }

  return result;
}


Atom* AppendExpression::eval(Context& c) {
  ListAtom* result = createNil(c);

  std::deque<Argument*> args = getArguments();
  std::deque<Argument*>::iterator it = args.begin();
  for (; it ≠ args.end(); ++it) {
    ListAtom* atom = (ListAtom*) getArgumentValue(*it, c);;

    std::vector<Atom*>& values = atom→getValues();
    for (size_t i = 0; i < values.size(); ++i) {
      result→addValue(values[i]);
    }
  }

  return result;
}


Atom* IfExpression::eval(Context& c) {
  std::deque<Argument*> args = getArguments();
  std::deque<Argument*>::iterator it = args.begin();
  Atom* r;
  if (getArgumentValue(*it, c)→isTrue()) {
    r = getArgumentValue(*(it + 1), c);

  } else {
    r =  getArgumentValue(*(it + 2), c);
  }
  return r;
}


Atom* SetqExpression::eval(Context& c) {
  std::deque<Argument*> args = getArguments();
  std::deque<Argument*>::iterator it = args.begin();

  std::string key = getArgumentValue(*it, c)→getValue();
  c.setAtom(key, getArgumentValue(*(it + 1), c));

  return createNil(c);
}


Atom* SyncExpression::eval(Context& c) {
  c.joinThreads();
  return createNil(c);
}
```

```


Atom* DefunExpression::eval(Context& c) {
  std::deque<Argument*> args = getArguments();
  std::deque<Argument*>::iterator it = args.begin();

  std::string key = getArgumentValue(*it, c)→getValue();

  c.setExpression(key, this);

  return createNil(c);
}

void DefunExpression::setExpressionString(std::string s) {
  expression_ = s;
}

std::string DefunExpression::getExpressionString(std::string param) {
  std::string result = expression_;
  size_t index = 0;

  while (true) {
    index = result.find("ENV", index);
    if (index ≡ std::string::npos) break;
    result.replace(index, 3, param, 0, param.size());

    index += 3;
  }

  return result;
}
```

```
#ifndef __LISP_ATOMS_H__
#define __LISP_ATOMS_H__

#include <string>
#include <vector>


class Atom {
public:
  virtual bool isTrue() { return true; }

  virtual std::string getValue() = 0;

  virtual ~Atom() {}
};

class StringAtom : public Atom {
std::string value_;
public:
  void setValue(const std::string s) { value_ = s; }

  virtual std::string getValue() { return value_; }
};

class NumericAtom : public Atom {
int value_;
public:
  void setValue(std::string s);
  virtual std::string getValue();
  int getNumericValue();
};


class ListAtom : public Atom {
std::vector<Atom*> values;
public:
  void setValue(std::string s) {}

  virtual bool isTrue();

  void addValue(Atom* value);
  virtual std::string getValue();

  std::vector<Atom*>& getValues();
};


#endif
```

```

#include <string>
#include <sstream>
#include <vector>

#include "atoms.h"


void NumericAtom::setValue(std::string s) {
  std::stringstream ss(s);
  ss >> value_;
}

std::string NumericAtom::getValue() {
  std::stringstream ss;
  ss << value_;
  return ss.str();
}

int NumericAtom::getNumericValue() {
  return value_;
}



bool ListAtom::isTrue() {
  return values.size();
}

void ListAtom::addValue(Atom* value) {
  values.push_back(value);
}

std::string ListAtom::getValue() {
  std::stringstream ss;
  ss << "(";
  for (size_t i = 0; i < values.size(); ++i) {
    ss << (values[i])→getValue();
    if (i ≠ values.size() − 1) ss << " ";
  }
  ss << ")";

  return ss.str();
}

std::vector<Atom*>& ListAtom::getValues() {
  return values;
}
```

Table of Contents