

Application multi-parsers réentrants en C++ avec Flex&Bison

Principes

Réalisation minimale

Fabrice HARROUET

École Nationale d'Ingénieurs de Brest

harrouet@enib.fr

<http://www.enib.fr/~harrouet/>

Origine des outils

▷ **Lex&Yacc**

- ◇ Générateurs d'analyseurs lexicaux/syntaxiques en C
- ◇ \simeq années 1970, laboratoires *Bell*
- ◇ Outils *UNIX* (*Posix*) standards

▷ **Flex&Bison**

- ◇ Version *GNU* de *Lex&Yacc*
- ◇ *Flex* : “fast *Lex*” *Bison* : Jeu de mot sur *Yacc*
- ◇ Beaucoup de possibilités supplémentaires !
- ◇ Disponible pour un très grand nombre de plateformes

▷ **(Flex++&Bison++)**

- ◇ Versions modifiées, dédiées à C++
- ◇ Multi-parsers/réentrant, mais plus maintenu depuis 1993 !

Réalisation envisagée

- ▷ **Une unique classe Parser en C++**
 - ◇ Encapsuler `yylex()`, `yyparse()` ...
 - ◇ Cohérent avec une approche orientée-objets
 - ◇ Une unique classe → communication simplifiée
- ▷ **Application multi-parsers**
 - ◇ Analyser des langages différents dans la même application
 - ◇ → éviter les conflits de nom !
- ▷ **Parser réentrant**
 - ◇ Un *parser* peut en invoquer un autre (langage = ou ≠)
 - ◇ Utilisation dans une application multi-threads
 - ◇ → ne pas utiliser de variables globales/statiques !

Démarche envisagée

- ▷ **Au niveau de Flex**
 - ◇ Un fichier `.l`
 - ◇ Mode `C++` : génération de classes
 - implicitement multi-scanners et réentrant
- ▷ **Au niveau de Bison**
 - ◇ Un fichier `.y`
 - ◇ Option de réentrance
 - `yyval` n'est plus globale, doit être transmise à `yylex()`
 - Tables globales mais `static` (visibilité limitée)
 - ◇ Macros pour que les fonctions deviennent des méthodes
- ▷ **Association Flex&Bison**
 - ◇ Rédaction d'un fichier `.h` commun
 - ◇ Génération d'un unique fichier `.cpp` depuis le `.l` et le `.y`

Définition de la classe

```

#ifndef MINIPARSER_H
#define MINIPARSER_H

#include <iostream>
using namespace std;
#define _CPP_BACKWARD_Iostream_H 1
#define yyFlexLexer _MiniFlexLexer
#include <FlexLexer.h>
#undef yyFlexLexer

class MiniParser : protected _MiniFlexLexer
{
public:
    MiniParser(istream * input);
    virtual ~MiniParser(void);
    virtual int parse(void);
protected:
    virtual int yylex(void * arg);
    virtual void yyerror(const char * msg);
};

#endif // MINIPARSER_H

```

Définition de la classe

▷ Inclusions

- ◇ `iostream` standard → espace `std`
→ `_CPP_BACKWARD_Iostream_H` évite l'ancien `iostream.h`
- ◇ `FlexLexer.h` installé en standard avec *Flex*
- ◇ Définit l'abstraction `FlexLexer` et l'implémentation `yyFlexLexer`
- ◇ On renomme `yyFlexLexer` en vue d'une utilisation multi-*parsers*

▷ Définition de la classe Parser

- ◇ Hérite de l'implémentation fournie par *Flex*
→ `protected` : “*dissimuler*” les détails d'implémentation
- ◇ Communiquer un flot d'entrée à l'implémentation
- ◇ `parse()/yylex()` générées par *Bison/Flex*
- ◇ `yyerror()` est à fournir impérativement
- ◇ Possibilité de compléter la classe (applicatif)

Le fichier destiné à Flex

```

%{
  #undef YY_DECL
  #define YY_DECL int MiniParser::yylex(void * arg)
  #define yylval (*((YYSTYPE*)arg))
%}

%option c++
%option 8bit
%option noyywrap
%option prefix="_Mini"
%option yyclass="MiniParser"

  /* expressions rationnelles nommees */

%%

  /* expressions rationnelles --> actions semantiques */
  /* Utilisation de yylval et des membres de notre classe */

%%

```

Le fichier destiné à Flex

▷ Les macros

- ◊ YY_DECL remplace le prototype de `yylex()`
- ◊ `yylex()` attend l'@ du `yylval` local à `yyparse()` (réentrance)
 - `void *` car le type de `%union (YYSTYPE)` est inconnu dans le `.h`
 - Le type de `%union (YYSTYPE)` sera en amont dans le `.cpp`
 - la macro `yylval` permet les actions sémantiques habituelles

▷ Options

- ◊ `c++` : Générer du `C++`
- ◊ `8bit` : Accepter les caractères accentués
- ◊ `noyywrap` : Pas de fonction `yywrap()`
- ◊ `prefix` : Implémentation dans `_MiniFlexLexer`
- ◊ `yyclass` : Classe "utilisable" `MiniParser` (contient `yylex()`)

Le fichier destiné à *Bison*

```

%{
#include "miniParser.h"

#define yyparse MiniParser::parse
%}

%pure_parser

/* %token %left %right %nonassoc %union %type %start ... */

%%

/* Grammaire --> actions semantiques */
/* Utilisation des membres de notre classe */

%%
MiniParser::MiniParser(istream * input) : _MiniFlexLexer(input) {}
MiniParser::~MiniParser(void) {}
void MiniParser::yyerror(const char * msg) { cerr << msg << endl; }

```

Le fichier destiné à *Bison*

▷ L'entête

- ◊ Inclure la définition de la classe
- ◊ Éventuellement d'autres inclusions ou déclarations
- ◊ Renommer `yyparse()` pour correspondre à notre méthode `parse()`
- ◊ L'option `%pure_parser` permet la réentrance
 - `yyval` n'est plus globale mais locale
 - son adresse est passée à l'appel de `yylex()`

▷ L'implémentation des méthodes

- ◊ Le constructeur doit passer le flot d'entrée à la classe parent
- ◊ Il reste à traiter les détails applicatifs

La génération du code

▷ Démarche globale

- ◇ Les passes *Flex* et *Bison* génèrent leur code dans le même fichier
- ◇ Pas besoin de générer de `.h` pour les *token* et l'*union*
- ◇ Un seul fichier `.cpp` contient toute l'implémentation du parser
- ◇ Il sera compilé comme les autres fichiers `.cpp`

```

${PARSER}.cpp : ${PARSER}.y ${PARSER}.l
    bison -v -o${PARSER}.cpp ${PARSER}.y
    cat ${PARSER}.cpp | sed -e s/"int yyparse .*/"/ > ${PARSER}.cpp.tmp
    mv ${PARSER}.cpp.tmp ${PARSER}.cpp
    flex -Cfa -t ${PARSER}.l | \
        sed -e s/"class istream;"/"/ \
            -e s/"#include <FlexLexer.h>"/"/ \
            -e s/"extern \"C\" int isatty .*/"/ >> ${PARSER}.cpp

```

La génération du code

▷ La passe *Bison*

- ◇ Option `-v` pour un fichier `.output` (mise au point)
- ◇ Option `-o` pour le nom du fichier de sortie
- ◇ `sed` supprime la déclaration embarrassante de `yyparse()` (macro !)

▷ La passe *Flex*

- ◇ Option `-Cfa` pour l'efficacité (facultative)
- ◇ Option `-t` pour générer le code sur la sortie standard
- ◇ Supprimer la déclaration de `istream` (conflit avec `std::istream`)
- ◇ Une inclusion embarrassante (déjà réalisée) est supprimée
- ◇ La déclaration de `isatty()` est supprimée (*warning* possible)
- ◇ Le résultat de ce filtrage complète le fichier généré par *Bison*

L'utilisation du *parser*

▷ Dans l'application

- ◇ L'inclusion du `.h` est suffisante
- ◇ Le *parser* doit être instancié avec un flot d'entrée
- ◇ On invoque sa méthode `parse()`
- ◇ C'est une classe comme les autres (méthodes, attributs ...)

```
#include "miniParser.h"

#include <fstream>
using namespace std;

int main(int argc, char ** argv)
{
    istream * input=&cin;
    if(argc>1) input=new ifstream(argv[1]); // Check result !
    MiniParser parser(input);
    parser.parse();
    if(input!=&cin) delete input;
    return(0);
}
```

enib, F.H ... 13/19

Améliorer le signalement des erreurs

▷ Localiser précisément l'erreur signalée avec `yyerror()`

- ◇ Indiquer le numéro de ligne
- ◇ Afficher la portion de la ligne qui précède l'erreur
- ◇ Compter les erreurs pour bilan final

▷ Traitements à mettre en place

- ◇ Compter les lignes (depuis l'analyse lexicale)
 - Repérer tous les `\n`
- ◇ Reconstituer la ligne courante (depuis l'analyse lexicale)
 - Chaque *ER* doit enregistrer `yytext` dans la ligne
 - Les `\n` vident la ligne courante
- ◇ Dans `yyerror()`
 - Incrémenter le compteur d'erreurs
 - Afficher la ligne courante et son numéro

enib, F.H ... 14/19

Signalement des erreurs — Définition de la classe

```

#ifndef MINIPARSER_H
#define MINIPARSER_H

#include <string>
// ...
class MiniParser : protected _MiniFlexLexer
{
public:
    MiniParser(istream * input);
    virtual ~MiniParser(void);
    virtual int parse(void);
    virtual unsigned int getNbErrors(void) const;
protected:
    virtual int yylex(void * arg);
    virtual void yyerror(const char * msg);
    virtual void _store(const char * txt);
    virtual void _nextLine(void);
    unsigned int _nbErrors;
    unsigned int _nbLines;
    string _line;
};
#endif // MINIPARSER_H

```

Signalement des erreurs — Le fichier destiné à Flex

```

%{
// ...
%}

/* ... */

%%

"keyword" { _store(yytext); return(KEYWORD); }

"+"      { _store(yytext); return(PLUS); }

/* ... */

[ \t]+   { _store(yytext); }

"\n"     { _nextLine(); }

.        { _store(yytext); yyerror("lexical error"); }

%%

```

Signalement des erreurs — Le fichier destiné à *Bison*

```

%{ // ...
%}
/* ... */
%%
/* ... */
%%
MiniParser::MiniParser(istream * input)
: _MiniFlexLexer(input), _nbErrors(0), _nbLines(1), _line() {}

MiniParser::~MiniParser(void) {}

unsigned int MiniParser::getNbErrors(void) const { return(_nbErrors); }

void MiniParser::yyerror(const char * msg)
{
++_nbErrors;
cerr << "line " << _nbLines << ": " << msg << endl;
if(!_line.empty()) cerr << _line << " <---" << endl;
}

void MiniParser::_store(const char * txt) { _line+=txt; }
void MiniParser::_nextLine(void) { _line.clear(); _nbLines++; }

```

Signalement des erreurs — L'utilisation du *parser*

```

#include "miniParser.h"

#include <fstream>
using namespace std;

int main(int argc, char ** argv)
{
istream * input=&cin;
if(argc>1) input=new ifstream(argv[1]); // Check result !
MiniParser parser(input);
parser.parse();
if(parser.getNbErrors())
{
cerr << "Failure ! (" << parser.getNbErrors() << " errors)" << endl;
}
else
{
cerr << "Success !" << endl;
}
if(input!&cin) delete input;
return(0);
}

```

Quelques remarques

▷ %union désigne une union !

- ◇ Les champs ne peuvent pas avoir de constructeurs !
- ◇ ex : pas de `std::string` mais `char [0x100]` → dimension !?!
 - Trop petit → débordement
 - Trop grand → saturation de la pile
- ◇ Si type complexes → passer par des pointeurs

▷ Pointeurs dans l'%union

- ◇ Risque de fuites mémoire (après erreur syntaxique ...)
- ◇ Mémoriser ce qu'on alloue dans un vecteur de pointeurs
- ◇ À l'utilisation du pointeur → le retirer du vecteur
- ◇ À la destruction du *parser* → détruire le contenu du vecteur

```
//-----  
#ifndef CALCPARSER_H  
#define CALCPARSER_H 1  
  
#include <string>  
#include <vector>  
#include <iostream>  
using namespace std;  
#define _CPP_BACKWARD_Iostream_H 1  
  
#define yyFlexLexer _CalcFlexLexer  
#include <FlexLexer.h>  
#undef yyFlexLexer  
  
class Calculator;  
class InstrNode;  
class ExprNode;  
  
class CalcParser : protected _CalcFlexLexer  
{  
public:  
  
    CalcParser(istream * input,  
               Calculator * calc);  
  
    virtual  
    ~CalcParser(void);  
  
    virtual  
    int  
    parse(void);  
  
    virtual  
    unsigned int  
    getNbErrors(void) const;  
  
protected:  
  
    virtual  
    int  
    yylex(void * arg);  
  
    virtual  
    void  
    yyerror(const string & msg);  
  
    virtual  
    void  
    _store(const char * txt);  
  
    virtual  
    void  
    _nextLine(void);  
  
    virtual  
    string *  
    _saveString(string * str);  
  
    virtual  
    string *  
    _relaxString(string * str);  
  
    virtual  
    InstrNode *  
    _saveInstr(InstrNode * instr);  
  
    virtual
```

```
InstrNode *
_relaxInstr(InstrNode * instr);

virtual
ExprNode *
_saveExpr(ExprNode * expr);

virtual
ExprNode *
_relaxExpr(ExprNode * expr);

Calculator * _calc;
unsigned int _nbErrors;
unsigned int _nbLines;
string _line;
vector<string *> _strings;
vector<InstrNode *> _iNodes;
vector<ExprNode *> _eNodes;
};

#endif // CALCPARSER_H
//-----
```

```

/*-----*/
%{
#undef YY_DECL
#define YY_DECL int CalcParser::yylex(void * arg)
#define yylval (*(YYSTYPE*)arg)
%}

%option c++
%option 8bit
%option noyywrap
%option prefix="_Calc"
%option yyclass="CalcParser"

%x strEnv

integer      [0-9]+
r1           [0-9]*\.[0-9]+
r2           [0-9]+\.[0-9]*
r_exp       [eE][+-]?[0-9]+
real        ({r1}|{r2}){r_exp}?|{integer}{r_exp}
value       {integer}|{real}
ident       [A-Za-z_][0-9a-zA-Z_]*

%%

"+"         {
            _store(yytext);
            return(CALC_PLUS);
        }

"-"         {
            _store(yytext);
            return(CALC_MINUS);
        }

"*"         {
            _store(yytext);
            return(CALC_MULT);
        }

"/"         {
            _store(yytext);
            return(CALC_DIV);
        }

"("         {
            _store(yytext);
            return(CALC_LP);
        }

")"         {
            _store(yytext);
            return(CALC_RP);
        }

"="         {
            _store(yytext);
            return(CALC_ASSIGN);
        }

";"         {
            _store(yytext);
            return(CALC_SEMICOLON);
        }

">>"       {
            _store(yytext);
            return(CALC_IN);
        }

"<<"       {
            _store(yytext);
            return(CALC_OUT);
        }

{value}     {
            _store(yytext);
        }

```

```

        sscanf(yytext, "%lf", &yylval.value);
        return(CALC_VALUE);
    }
    "\\"
    {
        _store(yytext);
        _saveString(new string);
        BEGIN(strEnv);
    }
    <strEnv> "\\"
    {
        _store(yytext);
        BEGIN(INITIAL);
        yylval.str=_strings.back();
        return(CALC_STRING);
    }
    <strEnv> "\\a"
    { _store(yytext); (*_strings.back())+="a"; }
    <strEnv> "\\b"
    { _store(yytext); (*_strings.back())+="b"; }
    <strEnv> "\\f"
    { _store(yytext); (*_strings.back())+="f"; }
    <strEnv> "\\n"
    { _store(yytext); (*_strings.back())+="n"; }
    <strEnv> "\\r"
    { _store(yytext); (*_strings.back())+="r"; }
    <strEnv> "\\t"
    { _store(yytext); (*_strings.back())+="t"; }
    <strEnv> "\\v"
    { _store(yytext); (*_strings.back())+="v"; }
    <strEnv> "\\\\"
    { _store(yytext); (*_strings.back())+="\\"; }
    <strEnv> "\\\""
    { _store(yytext); (*_strings.back())+="\""; }
    <strEnv> "\\n"
    { _nextLine(); (*_strings.back())+="n"; }
    <strEnv> "\\n"
    { (*_strings.back())+="n"; }
    <strEnv> "\\."
    { _store(yytext); (*_strings.back())+=yytext[1]; }
    <strEnv><<EOF>>
    {
        BEGIN(INITIAL);
        yylval.str=_strings.back();
        return(CALC_STRING);
    }
    <strEnv> .
    {
        _store(yytext);
        (*_strings.back())+=yytext;
    }
    {ident}
    {
        _store(yytext);
        yylval.str=_saveString(new string(yytext));
        return(CALC_IDENT);
    }
    "/*" . *
    { /* comment */; }
    [ \t]+
    { _store(yytext); }
    "\n"
    { _nextLine(); }
    .
    {
        _store(yytext);
        yyerror("lexical error");
    }
    %%
    /*-----*/

```

```

/*-----*/
%{
#include "calcParser.h"

#include "calculator.h"
#include "instrNode.h"
#include "exprNode.h"

#define yyparse CalcParser::parse
%}

%pure_parser

%token CALC_PLUS CALC_MINUS CALC_MULT CALC_DIV
%token CALC_LP CALC_RP CALC_ASSIGN CALC_SEMICOLON
%token CALC_IN CALC_OUT
%token CALC_VALUE CALC_STRING CALC_IDENT

%left CALC_PLUS CALC_MINUS
%left CALC_MULT CALC_DIV
%right CALC_UMINUS

%union
{
    double value;
    string * str;
    InstrNode * instr;
    ExprNode * expr;
};
%type<value> CALC_VALUE
%type<str> CALC_STRING CALC_IDENT
%type<instr> instrList instr outList outInstr
%type<expr> expr

%start program
%%

program
    : instrList
      {
          if($1)
          {
              _calc->setCode(_relaxInstr($1));
          }
      }
    ;

instrList
    : instrList instr
      {
          if($1)
          {
              $$=_saveInstr(new SeqNode(_relaxInstr($1),_relaxInstr($2)));
          }
          else
          {
              $$=$2;
          }
      }
    | /* empty */
      {
          $$=(InstrNode *)0;
      }
    ;

instr
    : CALC_IDENT CALC_ASSIGN expr CALC_SEMICOLON

```

```

        {
            double * var=_calc->writeVar(*($1));
            $$=_saveInstr(new AssignNode(var,_relaxExpr($3)));
            delete _relaxString($1);
        }
| CALC_IN CALC_IDENT CALC_SEMICOLON
        {
            double * var=_calc->writeVar(*($2));
            $$=_saveInstr(new ReadNode(var));
            delete _relaxString($2);
        }
| outList CALC_SEMICOLON
        {
            $$=$1;
        }
| error CALC_SEMICOLON
        {
            $$=_saveInstr(new WriteStrNode("error")); // default instruction
            yyerrok;
        }
;

outList
: outList outInstr
    {
        $$=_saveInstr(new SeqNode(_relaxInstr($1),_relaxInstr($2)));
    }
| outInstr
    {
        $$=$1;
    }
;

outInstr
: CALC_OUT expr
    {
        $$=_saveInstr(new WriteNode(_relaxExpr($2)));
    }
| CALC_OUT CALC_STRING
    {
        $$=_saveInstr(new WriteStrNode(*($2)));
        delete _relaxString($2);
    }
;

expr
: expr CALC_PLUS expr
    {
        $$=_saveExpr(new PlusNode(_relaxExpr($1),_relaxExpr($3)));
    }
| expr CALC_MINUS expr
    {
        $$=_saveExpr(new MinusNode(_relaxExpr($1),_relaxExpr($3)));
    }
| expr CALC_MULT expr
    {
        $$=_saveExpr(new MultNode(_relaxExpr($1),_relaxExpr($3)));
    }
| expr CALC_DIV expr
    {
        $$=_saveExpr(new DivNode(_relaxExpr($1),_relaxExpr($3)));
    }
| CALC_MINUS expr %prec CALC_UMINUS
    {
        $$=_saveExpr(new UminusNode(_relaxExpr($2)));
    }
| CALC_VALUE

```

```

        {
            $$=_saveExpr(new ConstNode($1));
        }
    | CALC_IDENT
        {
            const double * var=_calc->readVar(*($1));
            if(var)
            {
                $$=_saveExpr(new VarNode(var));
            }
            else
            {
                yyerror("Cannot read var "+*($1));
                $$=_saveExpr(new ConstNode(0.0)); // default expression
            }
            delete _relaxString($1);
        }
    | CALC_LP expr CALC_RP
        {
            $$=$2;
        }
;

%%

```

```

CalcParser::CalcParser(istream * input,
                        Calculator * calc)

```

```

: _CalcFlexLexer(input),
  _calc(calc),
  _nbErrors(0),
  _nbLines(1),
  _line(),
  _strings(),
  _iNodes(),
  _eNodes()
{
    // Nothing to be done
}

```

```

CalcParser::~CalcParser(void)

```

```

{
if(!_strings.empty())
{
    cerr << _strings.size() << " string(s) left!" << endl;
    do
    {
        delete _strings.back();
        _strings.pop_back();
    } while(!_strings.empty());
}
if(!_iNodes.empty())
{
    cerr << _iNodes.size() << " instruction(s) left!" << endl;
    do
    {
        delete _iNodes.back();
        _iNodes.pop_back();
    } while(!_iNodes.empty());
}
if(!_eNodes.empty())
{
    cerr << _eNodes.size() << " expression(s) left!" << endl;
    do
    {
        delete _eNodes.back();
        _eNodes.pop_back();
    } while(!_eNodes.empty());
}
}

```

```
}
}

unsigned int
CalcParser::getNbErrors(void) const
{
return(_nbErrors);
}

void
CalcParser::yyerror(const string & msg)
{
++_nbErrors;
cerr << "line " << _nbLines << ":" << msg << endl;
if(!_line.empty()) cerr << _line << " <---" << endl;
}

void
CalcParser::_store(const char * txt)
{
_line+=txt;
}

void
CalcParser::_nextLine(void)
{
_line.clear();
++_nbLines;
}

string *
CalcParser::_saveString(string * str)
{
_strings.push_back(str);
return(str);
}

string *
CalcParser::_relaxString(string * str)
{
for(unsigned int i=_strings.size();i--;)
{
if(_strings[i]==str)
{
_strings.erase(_strings.begin()+i);
return(str);
}
}
return(str);
}

InstrNode *
CalcParser::_saveInstr(InstrNode * instr)
{
_iNodes.push_back(instr);
return(instr);
}

InstrNode *
CalcParser::_relaxInstr(InstrNode * instr)
{
for(unsigned int i=_iNodes.size();i--;)
{
if(_iNodes[i]==instr)
{
_iNodes.erase(_iNodes.begin()+i);
return(instr);
}
}
}
```

```
    }  
  }  
  return(instr);  
}  
  
ExprNode *  
CalcParser::_saveExpr(ExprNode * expr)  
{  
  _eNodes.push_back(expr);  
  return(expr);  
}  
  
ExprNode *  
CalcParser::_relaxExpr(ExprNode * expr)  
{  
  for(unsigned int i=_eNodes.size();i-->0)  
  {  
    if(_eNodes[i]==expr)  
    {  
      _eNodes.erase(_eNodes.begin()+i);  
      return(expr);  
    }  
  }  
  return(expr);  
}  
  
/*-----*/
```

```
//-----  
#ifndef CALCULATOR_H  
#define CALCULATOR_H 1  
  
#include <map>  
#include <string>  
using namespace std;  
  
class InstrNode;  
  
class Calculator  
{  
public:  
  
    Calculator(void);  
  
    virtual  
    ~Calculator(void);  
  
    virtual  
    void  
    setCode(InstrNode * code);  
  
    virtual  
    const InstrNode *  
    getCode(void) const;  
  
    virtual  
    double *  
    writeVar(const string & name);  
  
    virtual  
    const double *  
    readVar(const string & name) const;  
  
protected:  
  
    map<string,double *> _vars;  
    InstrNode * _code;  
};  
  
#endif // CALCULATOR_H  
  
//-----
```

```
//-----  
#include "calculator.h"  
#include "instrNode.h"  
  
Calculator::Calculator(void)  
: _vars(),  
  _code((InstrNode *)0)  
{  
}  
  
Calculator::~~Calculator(void)  
{  
if(_code) delete _code;  
while(!_vars.empty())  
{  
    delete (*_vars.begin()).second;  
    _vars.erase(_vars.begin());  
}  
}  
  
void  
Calculator::setCode(InstrNode * code)  
{  
if(_code) delete _code;  
_code=code;  
}  
  
const InstrNode *  
Calculator::getCode(void) const  
{  
return(_code);  
}  
  
double *  
Calculator::writeVar(const string & name)  
{  
map<string,double *>::iterator it=_vars.find(name);  
if(it==_vars.end())  
{  
    double * d=new double;  
    _vars.insert(map<string,double *>::value_type(name,d));  
    return(d);  
}  
else  
{  
    return((*it).second);  
}  
}  
  
const double *  
Calculator::readVar(const string & name) const  
{  
map<string,double *>::const_iterator it=_vars.find(name);  
if(it==_vars.end())  
{  
    return((const double *)0);  
}  
else  
{  
    return((*it).second);  
}  
}  
  
//-----
```

```
//-----  
#ifndef INSTRNODE_H  
#define INSTRNODE_H 1  
  
#include <string>  
using namespace std;  
  
class ExprNode;  
  
//-----  
class InstrNode  
{  
public:  
  
    InstrNode(void);  
  
    virtual  
    ~InstrNode(void);  
  
    virtual  
    void  
    exec(void) const=0;  
};  
  
//-----  
class SeqNode : public InstrNode  
{  
public:  
  
    SeqNode(InstrNode * left,  
            InstrNode * right);  
  
    virtual  
    ~SeqNode(void);  
  
    virtual  
    void  
    exec(void) const;  
  
protected:  
  
    InstrNode * _left;  
    InstrNode * _right;  
};  
  
//-----  
class AssignNode : public InstrNode  
{  
public:  
  
    AssignNode(double * var,  
              ExprNode * expr);  
  
    virtual  
    ~AssignNode(void);  
  
    virtual  
    void  
    exec(void) const;  
  
protected:  
  
    double * _var;  
};
```

```
ExprNode * _expr;
};

//-----

class ReadNode : public InstrNode
{
public:

    ReadNode(double * var);

    virtual
    ~ReadNode(void);

    virtual
    void
    exec(void) const;

protected:

    double * _var;
};

//-----

class WriteNode : public InstrNode
{
public:

    WriteNode(ExprNode * expr);

    virtual
    ~WriteNode(void);

    virtual
    void
    exec(void) const;

protected:

    ExprNode * _expr;
};

//-----

class WriteStrNode : public InstrNode
{
public:

    WriteStrNode(const string & str);

    virtual
    ~WriteStrNode(void);

    virtual
    void
    exec(void) const;

protected:

    string _str;
};

#endif // INSTRNODE_H

//-----
```

```
//-----  
#include "instrNode.h"  
#include "exprNode.h"  
  
#include <iostream>  
using namespace std;  
  
//-----  
  
InstrNode::InstrNode(void)  
{  
}  
  
InstrNode::~InstrNode(void)  
{  
}  
  
// void  
// InstrNode::exec(void) const=0;  
  
//-----  
  
SeqNode::SeqNode(InstrNode * left,  
                 InstrNode * right)  
: InstrNode(),  
  _left(left),  
  _right(right)  
{  
}  
  
SeqNode::~SeqNode(void)  
{  
delete _left;  
delete _right;  
}  
  
void  
SeqNode::exec(void) const  
{  
  _left->exec();  
  _right->exec();  
}  
  
//-----  
  
AssignNode::AssignNode(double * var,  
                       ExprNode * expr)  
: InstrNode(),  
  _var(var),  
  _expr(expr)  
{  
}  
  
AssignNode::~AssignNode(void)  
{  
delete _expr;  
}  
  
void  
AssignNode::exec(void) const  
{  
  *_var=_expr->eval();  
}  
  
//-----
```

```
ReadNode::ReadNode(double * var)
: InstrNode(),
  _var(var)
{
}
```

```
ReadNode::~~ReadNode(void)
{
}
```

```
void
ReadNode::exec(void) const
{
  cin >> *_var;
}
```

```
//-----
```

```
WriteNode::WriteNode(ExprNode * expr)
: InstrNode(),
  _expr(expr)
{
}
```

```
WriteNode::~~WriteNode(void)
{
  delete _expr;
}
```

```
void
WriteNode::exec(void) const
{
  cout << _expr->eval();
}
```

```
//-----
```

```
WriteStrNode::WriteStrNode(const string & str)
: InstrNode(),
  _str(str)
{
}
```

```
WriteStrNode::~~WriteStrNode(void)
{
}
```

```
void
WriteStrNode::exec(void) const
{
  cout << _str;
}
```

```
//-----
```

```
//-----  
#ifndef EXPRNODE_H  
#define EXPRNODE_H 1  
  
//-----  
  
class ExprNode  
{  
public:  
  
    ExprNode(void);  
  
    virtual  
    ~ExprNode(void);  
  
    virtual  
    double  
    eval(void) const=0;  
};  
  
//-----  
  
class PlusNode : public ExprNode  
{  
public:  
  
    PlusNode(ExprNode * left,  
             ExprNode * right);  
  
    virtual  
    ~PlusNode(void);  
  
    virtual  
    double  
    eval(void) const;  
  
protected:  
  
    ExprNode * _left;  
    ExprNode * _right;  
};  
  
//-----  
  
class MinusNode : public ExprNode  
{  
public:  
  
    MinusNode(ExprNode * left,  
             ExprNode * right);  
  
    virtual  
    ~MinusNode(void);  
  
    virtual  
    double  
    eval(void) const;  
  
protected:  
  
    ExprNode * _left;  
    ExprNode * _right;  
};  
  
//-----
```

```
class MultNode : public ExprNode
{
public:

    MultNode(ExprNode * left,
             ExprNode * right);

    virtual
    ~MultNode(void);

    virtual
    double
    eval(void) const;

protected:

    ExprNode * _left;
    ExprNode * _right;
};
```

//-----

```
class DivNode : public ExprNode
{
public:

    DivNode(ExprNode * left,
            ExprNode * right);

    virtual
    ~DivNode(void);

    virtual
    double
    eval(void) const;

protected:

    ExprNode * _left;
    ExprNode * _right;
};
```

//-----

```
class UminusNode : public ExprNode
{
public:

    UminusNode(ExprNode * right);

    virtual
    ~UminusNode(void);

    virtual
    double
    eval(void) const;

protected:

    ExprNode * _right;
};
```

//-----

```
class ConstNode : public ExprNode
{
public:
```

```
ConstNode(double value);

virtual
~ConstNode(void);

virtual
double
eval(void) const;
```

```
protected:
```

```
    double _value;
};
```

```
//-----
```

```
class VarNode : public ExprNode
{
public:
```

```
    VarNode(const double * var);
```

```
virtual
~VarNode(void);
```

```
virtual
double
eval(void) const;
```

```
protected:
```

```
    const double * _var;
};
```

```
#endif // EXPRNODE_H
```

```
//-----
```

```
//-----  
#include "exprNode.h"  
  
//-----  
  
ExprNode::ExprNode(void)  
{  
}  
  
ExprNode::~ExprNode(void)  
{  
}  
  
// double  
// ExprNode::eval(void) const=0;  
  
//-----  
  
PlusNode::PlusNode(ExprNode * left,  
                   ExprNode * right)  
: ExprNode(),  
  _left(left),  
  _right(right)  
{  
}  
  
PlusNode::~PlusNode(void)  
{  
  delete _left;  
  delete _right;  
}  
  
double  
PlusNode::eval(void) const  
{  
  return(_left->eval()+_right->eval());  
}  
  
//-----  
  
MinusNode::MinusNode(ExprNode * left,  
                     ExprNode * right)  
: ExprNode(),  
  _left(left),  
  _right(right)  
{  
}  
  
MinusNode::~MinusNode(void)  
{  
  delete _left;  
  delete _right;  
}  
  
double  
MinusNode::eval(void) const  
{  
  return(_left->eval()-_right->eval());  
}  
  
//-----  
  
MultNode::MultNode(ExprNode * left,  
                   ExprNode * right)  
: ExprNode(),  
  _left(left),
```

```
    _right(right)
}

MultNode::~MultNode(void)
{
    delete _left;
    delete _right;
}

double
MultNode::eval(void) const
{
    return(_left->eval()*_right->eval());
}

//-----

DivNode::DivNode(ExprNode * left,
                 ExprNode * right)
: ExprNode(),
  _left(left),
  _right(right)
{
}

DivNode::~DivNode(void)
{
    delete _left;
    delete _right;
}

double
DivNode::eval(void) const
{
    return(_left->eval()/_right->eval());
}

//-----

UminusNode::UminusNode(ExprNode * right)
: ExprNode(),
  _right(right)
{
}

UminusNode::~UminusNode(void)
{
    delete _right;
}

double
UminusNode::eval(void) const
{
    return(-_right->eval());
}

//-----

ConstNode::ConstNode(double value)
: ExprNode(),
  _value(value)
{
}

ConstNode::~ConstNode(void)
{
}
```

```
}  
  
double  
ConstNode::eval(void) const  
{  
    return(_value);  
}  
  
//-----  
  
VarNode::VarNode(const double * var)  
: ExprNode(),  
  _var(var)  
{  
}  
  
VarNode::~VarNode(void)  
{  
}  
  
double  
VarNode::eval(void) const  
{  
    return(*_var);  
}  
  
//-----
```

```
//-----  
#include "calcParser.h"  
#include "calculator.h"  
#include "instrNode.h"  
  
#include <fstream>  
using namespace std;  
  
int  
main(int argc,  
      char ** argv)  
{  
if(argc<=1)  
{  
    cerr << "usage: " << argv[0] << " fichier" << endl;  
    return(1);  
}  
ifstream input(argv[1]);  
if(input.fail())  
{  
    cerr << "Cannot read from " << argv[1] << endl;  
    return(1);  
}  
Calculator calc;  
CalcParser parser(&input,&calc);  
parser.parse();  
if(parser.getNbErrors())  
{  
    cerr << endl << "Failure!(" << parser.getNbErrors() << " errors)" << endl;  
}  
else  
{  
    const InstrNode * code=calc.getCode();  
    if(code)  
    {  
        code->exec();  
    }  
    cerr << endl << "Success!" << endl;  
}  
return(0);  
}  
  
//-----
```

```
#####
#---- le programme a obtenir (cible principale) ----
TARGET=calc
PARSER=calcParser

all : ${TARGET}

#---- les fichiers sources ----

FILES=${TARGET}.cpp \
      ${PARSER}.cpp \
      calculator.cpp \
      instrNode.cpp \
      exprNode.cpp

#####

#---- les variables ----

INCDIR=.
CC=g++
CCFLAGS=-W -Wall -Werror -pedantic -I${INCDIR}
DEPFLAGS=-MM
LDFLAGS=

#####

#---- les fichiers objets ----

OBJECTS=${FILES:.cpp=.o}

#---- la regle par default (.cpp --> .o) ----

.cpp.o :
    @echo
    ${CC} -o $*.o -c ${CCFLAGS} $<
    @echo

#---- realisation de l'executable (edition de liens) ----

${TARGET} : ${OBJECTS}
    @echo
    ${CC} -o ${TARGET} ${CCFLAGS} ${OBJECTS} ${LDFLAGS}
    @echo

#----- ${PARSER}.cpp -----
${PARSER}.cpp : ${PARSER}.y ${PARSER}.l
    @echo
    bison -v -o${PARSER}.cpp ${PARSER}.y
    @echo
    cat ${PARSER}.cpp | sed -e s/"int yyparse.*"/"/ > ${PARSER}.cpp.tmp
    mv ${PARSER}.cpp.tmp ${PARSER}.cpp
    @echo
    flex -Cfa -t ${PARSER}.l | \
    sed -e s/"class istream;"/"/ \
    -e s/"#include <FlexLexer.h>"/"/ \
    -e s/"extern \"C\" int isatty.*"/"/ >> ${PARSER}.cpp
    @echo

#---- inclusion des dependances ----

-include .depend

#---- generation des dependances ----

dep : ${FILES}
```

```
@echo "=====  
Building dependencies ===== "  
@rm -f .depend  
@for i in ${FILES} ; do \  
    echo $$i ; \  
    ${CC} ${DEPFLAGS} ${CCFLAGS} $$i > .tmpdepend ; \  
    OBJNAME=`echo $$i | sed -e s%\\.cpp%.o% ` ; \  
    cat .tmpdepend | \  
    sed -e s%`basename $$i .cpp`\\.o%$$OBJNAME% >> .depend ; \  
    echo >> .depend ; \  
done  
@rm -f .tmpdepend
```

```
#---- nettoyage ----
```

```
clean :
```

```
    rm -f ${TARGET} *.o .depend a.out core \  
        ${PARSER}.cpp ${PARSER}.output
```

```
#####
```

```
$
$ cat test1.txt
<< "a=? "; // read a
>> a;
<< "b=? "; // read b
>> b;
average=0.5*(a+b); // compute average
<< "average=" << average << "\n"; // write average
$
$ ./calc test1.txt
a=? 123
b=? 456
average=289.5
```

```
Success !
$
```

```
$
$ cat test2.txt
<< "a=? "; // read a
>> a;
<< "b=? "; // read b
>> b;
average=0.5*(a+@b); // compute average
<< "average=" << average << "\n"; // write average
$
$ ./calc test2.txt
line 5: lexical error
average=0.5*(a+@ <---
```

```
Failure ! (1 errors)
$
```

```
$
$ cat test3.txt
<< "a=? "; // read a
>> a;
<< "b=? "; // read b
>> b;
average=0.5*(a+); // compute average
<< "average=" << average << "\n"; // write average
$
$ ./calc test3.txt
line 5: parse error
average=0.5*(a+) <---
line 6: Cannot read var average
<< "average=" << average <---
```

```
Failure ! (2 errors)
1 string(s) left !
2 expression(s) left !
$
```