Listing 1: C program for testing

```
1
2   /*************************************************
3                           INCLUDES.HPP
4   *************************************************/
5
6   #ifndef I_INCLUDES_H_
7   #define I_INCLUDES_H_
8
9   #include <string>
10  #include <vector>
11  #include <map>
12  #include <utility>
13  #include <stack>
14  #include <algorithm>
15  #include <iostream>
16  #include <cstring>
17  #include <cctype>
18  #include <cstdlib>
19  #include <set>
20
21  using namespace std;
22
23  typedef unsigned long long bit64;
24  typedef char bit8;
25  typedef unsigned int bit32;
26  typedef unsigned short bit16;
27
28  extern bit64 line_no;
29  extern bit64 col_no;
30  extern int indentLevel;
31
32  class I_error{
33      public:
34          I_error(const char *msg):
35              error_message_(msg), line_no_(line_no), col_no_(col_no) {}
36
37          void print(){
38              cout<<"ERROR :: LINE="<<line_no_<<" CHARACTER="<<col_no_
39                  <<" :: "<<error_message_<<endl;
40              exit(0);
41          }
42
43      private:
44          const char *error_message_;
```

```
45          int line_no_;
46          int col_no_;
47  };
48
49  #endif
50
51  /*************************************************
52                              LEXER.HPP
53  *************************************************/
54
55  #ifndef I_LEXER_H_
56  #define I_LEXER_H_
57
58  #include "includes.hpp"
59  using namespace std;
60
61
62  enum TokenType{ SYMBOL = 1536, INTEGER, REAL, OPERATOR, STRING , _NONE, VAR_MARKER };
63  const char *typeString(TokenType t);
64
65  enum Operator{  PLUS = 7654, MINUS, MULT, DIV, MOD, POW,
66                  AND, OR, NOT, XOR,
67                  GT, LT, GTE, LTE, EQ, NEQ ,
68                  COND, ELSE, CONSTRAINT,
69                  LPAREN, RPAREN, ERROR,
70                  LCURL, RCURL, LBRACKET, RBRACKET,
71                  ESCAPE, ASSIGN, COMMA, VAR };
72
73  const char *opString(Operator op);
74  Operator str2op(const char *str);
75
76  union TokenHolder{
77      int integer;
78      double real;
79      char str[1024];
80      char sym[1024];
81      Operator op;
82  };
83
84  class Token{
85      public:
86          TokenType type_;
87          TokenHolder token_;
88
89          Token():type_(_NONE){}
```

```
90          Token(TokenType type, const char *tok_str){
91              makeToken(type, tok_str);
92          }
93
94          Token (const Token &oldToken){
95              type_ = oldToken.type_;
96              token_ = oldToken.token_;
97          }
98
99          void makeToken(TokenType type, const char *tok_str);
100
101         inline TokenType getType(){ return type_; }
102         inline void print(){
103             cout<<"<"<<typeString(type_)<<">";
104             switch(type_){
105                 case INTEGER:   cout<<token_.integer;          break;
106                 case REAL:      cout<<token_.real;             break;
107                 case SYMBOL:    cout<<token_.sym;              break;
108                 case STRING:    cout<<token_.str;              break;
109                 case VAR_MARKER:
110                 case OPERATOR:  cout<<opString(token_.op);     break;
111                 case _NONE:     cout<<"␣";                          break;
112                 default:        cout<<"␣";                          break;
113             }
114             cout<<"</"<<typeString(type_)<<">"<<endl;
115         }
116 };
117
118 class Tokenizer{
119     public:
120         Tokenizer():expr_(NULL),type_(_NONE){}
121         Tokenizer(const char *expr):expr_(expr){}
122
123         inline bool next(Token &t){
124             bool ret = getNextToken();
125             t.makeToken(type_, token_);
126             return ret;
127         }
128
129         inline const char* getExpr(){ return expr_; }
130
131     private:
132         const char *expr_;
133         char token_[1024];
134         TokenType type_;
```

```
135
136         inline bool isOperator(char ch){
137             if(strchr("+-*/%^=(){}[]&|~><?:@,\\", ch)) return true;
138             return false;
139         }
140
141         bool getNextToken();
142 };
143
144 #endif
145
146
147
148 /*************************************************
149                         LEXER.CPP
150 *************************************************/
151
152 #include "lexer.hpp"
153 using namespace std;
154
155 const char *typeString(TokenType t){
156     switch(t){
157         case SYMBOL:
158             return "SYMBOL";
159         case INTEGER:
160             return "INTEGER";
161         case REAL:
162             return "REAL";
163         case OPERATOR:
164             return "OPERATOR";
165         case STRING:
166             return "STRING";
167         case _NONE:
168             return "_NONE";
169         case VAR_MARKER:
170             return "VAR_MARKER";
171         default:
172             return "ERROR ...";
173     }
174 }
175
176 const char *opString(Operator op){
177     switch(op){
178         case PLUS:  return "PLUS";
179         case MINUS: return "MINUS";
```

```
180         case MULT:   return "MULT";
181         case DIV:    return "DIV";
182         case MOD:    return "MOD";
183         case POW:    return "POW";
184         case AND:    return "AND";
185         case OR:     return "OR";
186         case NOT:    return "NOT";
187         case XOR:    return "XOR";
188         case GT:     return "GT";
189         case LT:     return "LT";
190         case GTE:    return "GTE";
191         case LTE:    return "LTE";
192         case EQ:     return "EQ";
193         case NEQ:    return "NEQ";
194         case LPAREN:return "LPAREN";
195         case RPAREN:return "RPAREN";
196         case ERROR: return "ERROR";
197         case ESCAPE:return "ESCAPE";
198         case ASSIGN:return "ASSIGN";
199         case LCURL: return "LCURL";
200         case RCURL: return "RCURL";
201         case COMMA: return "COMMA";
202         case VAR:    return "VAR";
203         case LBRACKET:   return "LBRACKET";
204         case RBRACKET:   return "RBRACKET";
205         case CONSTRAINT:return "CONSTRAINT";
206         case COND:   return "COND";
207         case ELSE:   return "ELSE";
208         default:     return "\0";
209     }
210 }
211
212 Operator str2op(const char *str){
213     switch(str[0]){
214         case '+': return PLUS;
215         case '-': return MINUS;
216         case '*':
217             if( str[1]=='*') return POW;
218             return MULT;
219         case '/': return DIV;
220         case '%': return MOD;
221         case '&': return AND;
222         case '|': return OR;
223         case '~':
224             if( str[1]=='=') return NEQ;
```

```
225             return NOT;
226         case '^': return XOR;
227         case '>':
228             if( str[1]=='=') return GTE;
229             return GT;
230         case '<':
231             if( str[1]=='=') return LTE;
232             return LT;
233         case '=':
234             if(str[1]=='=') return EQ;
235             return ASSIGN;
236         case '(': return LPAREN;
237         case ')': return RPAREN;
238         case '{': return LCURL;
239         case '}': return RCURL;
240         case '\\':return ESCAPE;
241         case ',': return COMMA;
242         case '$': return VAR;
243         case '[': return LBRACKET;
244         case ']': return RBRACKET;
245         case '?': return COND;
246         case ':': return ELSE;
247         case '@': return CONSTRAINT;
248         default:
249             return ERROR;
250     }
251 }
252
253
254 void Token::makeToken(TokenType type, const char *tok_str){
255     type_ = type;
256     switch(type_){
257         case SYMBOL:
258             strcpy( token_.sym, tok_str);
259             break;
260         case INTEGER:
261             token_.integer = atoi(tok_str);
262             break;
263         case REAL:
264             token_.real = atof(tok_str);
265             break;
266         case STRING:
267             strcpy(token_.str, tok_str);
268             break;
269         case VAR_MARKER:
```

```
270             case OPERATOR:
271                 token_.op = str2op(tok_str);
272                 break;
273             case _NONE:
274                 break;
275             default:
276                 cout<<"ERROR OCCURED IN LEXICAL ANALYSIS"<<endl;
277                 break;
278         }
279     }
280
281
282     bool Tokenizer::getNextToken(){
283         char *temp;
284         type_ = _NONE;
285         temp = token_;
286         *temp = '\0';
287
288         if(!*expr_) return false;
289         while(isspace(*expr_)) {
290             expr_++ ;
291             col_no++;
292         }
293
294         if( isOperator(*expr_)){
295             switch(*expr_){
296                 case '*':
297                     *temp++ = *expr_++;
298                     col_no++;
299                     if(*(expr_+1)=='*'){
300                         *temp++ = *expr_++;
301                         col_no++;
302                     }
303                     break;
304                 case '~':
305                 case '>':
306                 case '<':
307                 case '=':
308                     *temp++ = *expr_++;
309                     col_no++;
310                     if(*(expr_+1)=='='){
311                         *temp++ = *expr_++;
312                         col_no++;
313                     }
314                     break;
```

```
315             default:
316                 *temp++ = *expr_++;
317                 col_no++;
318                 break;
319         }
320         type_ = OPERATOR;
321     }
322     else if(*expr_ == '$'){
323         *temp++ = *expr_++;
324         col_no++;
325         type_ = VAR_MARKER;
326     }
327     else if(isalpha(*expr_)){
328         while(isalpha(*expr_) || *expr_ == '_'){
329             *temp++ = *expr_++;
330             col_no++;
331         }
332         type_ = SYMBOL;
333     }
334     else if(isdigit(*expr_)){
335         while(isdigit(*expr_)){
336             *temp++ = *expr_++;
337             col_no++;
338         }
339         if(*expr_ == '.'){
340             *temp++ = *expr_++;
341             col_no++;
342             while(isdigit(*expr_)){
343                 *temp++ = *expr_++;
344                 col_no++;
345             }
346             type_ = REAL;
347         }
348         else type_ = INTEGER;
349     }
350     else if(*expr_ == '\"'){
351         *expr_++;
352         col_no++;
353         while(*expr_ != '\"'){
354             *temp++ = *expr_++;
355             col_no++;
356         }
357         *expr_++;
358         col_no++;
359         type_ = STRING;
```

```
360         }
361         else{
362             while(!isspace(*expr_)){
363                 *temp++ = *expr_++;
364                 col_no++;
365             }
366             type_ = SYMBOL;
367         }
368
369         *temp = '\0';
370         return true;
371     }
372
373
374     /*************************************************
375                             PARSER.HPP
376     *************************************************/
377
378     #ifndef I_PARSER_H_
379     #define I_PARSER_H_
380
381     #include "lexer.hpp"
382     using namespace std;
383
384     enum atomType{ TOKEN, SEXPR, LEXPR, VARIABLE };
385     struct atom{
386         atomType type_;
387         void *atom_;
388
389         void print();
390     };
391
392     struct Logical_not{
393         bool not_;
394         vector<atom> logical_not_;       // Unary operator ~
395
396         Logical_not(){
397             not_ = false;
398         }
399         void print();
400     };
401
402     struct Logical_xor{
403         vector<Logical_not> logical_xor_;    // values separated by ^
404         void print();
```

```
405 };
406
407 struct Logical_and{
408     vector<Logical_xor> logical_and_;    // values separated by &
409     void print();
410 };
411
412 struct Logical_or{
413     vector<Logical_and> logical_or_; // Values separated by |
414     void print();
415 };
416
417 struct Unary{
418     bool minus_;
419     vector<Logical_or> unary_;  // Unary operators +, -
420
421     Unary(){
422         minus_ = false;
423     }
424     void print();
425 };
426
427 struct Exponent{
428     vector<Unary> exponent_;     // Values separated by exponent op **
429     void print();
430 };
431
432 struct Modulus{                  // Values separated by %
433     vector<Exponent> modulus_;
434     void print();
435 };
436
437 struct FactorDiv{                    // Values separated by *, /
438     vector<Modulus> factor_div_;
439     void print();
440 };
441
442 struct FactorMult{
443     struct vector<FactorDiv> factor_mult_;
444     void print();
445 };
446
447 struct TermAdd{                  //Values separated by + and -
448     vector<FactorMult> term_add_;
449     void print();
```

```
450 };
451
452 struct TermSub{
453     vector<TermAdd> term_sub_;
454     void print();
455 };
456
457 struct Component{                  //The if-else operator _?_:_
458     vector<TermSub> component_;
459     void print();
460 };
461
462 struct LExpr{                      //Comma separated values
463     vector<Component> lexpr_;
464     void print();
465 };
466
467
468 struct Variable{
469     LExpr *context_;
470     char var_name_[1024];
471
472     Variable(){
473         context_ = NULL;
474     }
475
476     void print();
477 };
478
479
480 struct SExpr{
481     vector<atom> sexpr_;
482     void print();
483 };
484
485 struct Statement{
486     bool store_;
487     SExpr *lval_;
488     SExpr *rval_;
489
490     Statement():store_(true),lval_(NULL),rval_(NULL){}
491     ~Statement(){
492         delete lval_;
493         delete rval_;
494     }
```

```
495
496         inline void print(){
497             cout<<"<STATEMENT>"<<endl;
498             if(lval_ != NULL){
499                 cout<<"<L-VALUE>"<<endl;
500                 lval_->print();
501                 cout<<"</L-VALUE>"<<endl;
502             }
503
504             if(rval_ != NULL){
505                 cout<<"<R-VALUE>"<<endl;
506                 rval_->print();
507                 cout<<"</R-VALUE>"<<endl;
508             }
509             cout<<"</STATEMENT>"<<endl;
510         }
511     };
512
513
514
515
516
517
518
519     class Logical_notParser{
520         private: size_t parse_(size_t begin, Logical_not &expr);
521
522         public:
523             Logical_notParser(const vector<Token> &tok_stream, size_t begin){
524                 for(size_t i=begin; i<tok_stream.size(); i++){
525                     tok_stream_.push_back(tok_stream[i]);
526                 }
527             }
528             Logical_notParser(const vector<Token> &tok_stream)
529                 :tok_stream_(tok_stream){}
530             Logical_notParser(){}
531
532             void construct(const vector<Token> &tok_stream, size_t begin){
533                 for(size_t i=begin; i<tok_stream.size(); i++){
534                     tok_stream_.push_back(tok_stream[i]);
535                 }
536             }
537
538             void destruct(){
539                 vector<Token>().swap(tok_stream_);      //clear and dellocate memory
```

```
540         }
541
542         inline void setDelimiter(Operator op){
543             delimiter_ = op;
544         }
545
546         pair<Logical_not, size_t> getLogical_not();
547
548     private:
549         vector<Token> tok_stream_;
550         Operator delimiter_;
551
552 };
553
554 class Logical_xorParser{
555     public:
556         Logical_xorParser(const vector<Token> &tok_stream, size_t begin){
557             for(size_t i=begin; i<tok_stream.size(); i++){
558                 tok_stream_.push_back(tok_stream[i]);
559             }
560         }
561         Logical_xorParser(const vector<Token> &tok_stream)
562             :tok_stream_(tok_stream){}
563         Logical_xorParser(){}
564
565         void construct(const vector<Token> &tok_stream, size_t begin){
566             for(size_t i=begin; i<tok_stream.size(); i++){
567                 tok_stream_.push_back(tok_stream[i]);
568             }
569         }
570         void destruct(){
571             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
572         }
573         inline void setDelimiter(Operator op){
574             delimiter_ = op;
575         }
576         pair<Logical_xor, size_t> getLogical_xor();
577
578     private:
579         vector<Token> tok_stream_;
580         Operator delimiter_;
581
582         size_t parse_(size_t begin, Logical_xor &expr);
583
584 };
```

```
585
586 class Logical_andParser{
587     public:
588         Logical_andParser(const vector<Token> &tok_stream, size_t begin){
589             for(size_t i=begin; i<tok_stream.size(); i++){
590                 tok_stream_.push_back(tok_stream[i]);
591             }
592         }
593         Logical_andParser(const vector<Token> &tok_stream)
594             :tok_stream_(tok_stream){}
595         Logical_andParser(){}
596
597         void construct(const vector<Token> &tok_stream, size_t begin){
598             for(size_t i=begin; i<tok_stream.size(); i++){
599                 tok_stream_.push_back(tok_stream[i]);
600             }
601         }
602         void destruct(){
603             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
604         }
605
606         inline void setDelimiter(Operator op){
607             delimiter_ = op;
608         }
609
610         pair<Logical_and, size_t> getLogical_and();
611
612     private:
613         vector<Token> tok_stream_;
614         Operator delimiter_;
615
616         size_t parse_(size_t begin, Logical_and &expr);
617
618 };
619
620 class Logical_orParser{
621     public:
622         Logical_orParser(const vector<Token> &tok_stream, size_t begin){
623             for(size_t i=begin; i<tok_stream.size(); i++){
624                 tok_stream_.push_back(tok_stream[i]);
625             }
626         }
627         Logical_orParser(const vector<Token> &tok_stream)
628             :tok_stream_(tok_stream){}
629         Logical_orParser(){}
```

```
630
631         void construct(const vector<Token> &tok_stream, size_t begin){
632             for(size_t i=begin; i<tok_stream.size(); i++){
633                 tok_stream_.push_back(tok_stream[i]);
634             }
635         }
636         void destruct(){
637             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
638         }
639
640         inline void setDelimiter(Operator op){
641             delimiter_ = op;
642         }
643
644         pair<Logical_or, size_t> getLogical_or();
645
646     private:
647         vector<Token> tok_stream_;
648         Operator delimiter_;
649
650         size_t parse_(size_t begin, Logical_or &expr);
651
652 };
653
654 class UnaryParser{
655     public:
656         UnaryParser(const vector<Token> &tok_stream, size_t begin){
657             for(size_t i=begin; i<tok_stream.size(); i++){
658                 tok_stream_.push_back(tok_stream[i]);
659             }
660         }
661         UnaryParser(const vector<Token> &tok_stream)
662             :tok_stream_(tok_stream){}
663         UnaryParser(){}
664
665         void construct(const vector<Token> &tok_stream, size_t begin){
666             for(size_t i=begin; i<tok_stream.size(); i++){
667                 tok_stream_.push_back(tok_stream[i]);
668             }
669         }
670
671         void destruct(){
672             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
673         }
674         inline void setDelimiter(Operator op){
```

```
675             delimiter_ = op;
676         }
677 
678         pair<Unary, size_t> getUnary();
679 
680     private:
681         vector<Token> tok_stream_;
682         Operator delimiter_;
683 
684         size_t parse_(size_t begin, Unary &expr);
685 
686 };
687 
688 class ExponentParser{
689     public:
690         ExponentParser(const vector<Token> &tok_stream, size_t begin){
691             for(size_t i=begin; i<tok_stream.size(); i++){
692                 tok_stream_.push_back(tok_stream[i]);
693             }
694         }
695         ExponentParser(const vector<Token> &tok_stream)
696             :tok_stream_(tok_stream){}
697         ExponentParser(){}
698 
699         void construct(const vector<Token> &tok_stream, size_t begin){
700             for(size_t i=begin; i<tok_stream.size(); i++){
701                 tok_stream_.push_back(tok_stream[i]);
702             }
703         }
704         void destruct(){
705             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
706         }
707 
708         inline void setDelimiter(Operator op){
709             delimiter_ = op;
710         }
711 
712         pair<Exponent, size_t> getExponent();
713 
714     private:
715         vector<Token> tok_stream_;
716         Operator delimiter_;
717 
718         size_t parse_(size_t begin, Exponent &expr);
719 
```

```
720 };
721
722 class ModulusParser{
723     public:
724         ModulusParser(const vector<Token> &tok_stream, size_t begin){
725             for(size_t i=begin; i<tok_stream.size(); i++){
726                 tok_stream_.push_back(tok_stream[i]);
727             }
728         }
729         ModulusParser(const vector<Token> &tok_stream)
730             :tok_stream_(tok_stream){}
731         ModulusParser(){}
732
733         void construct(const vector<Token> &tok_stream, size_t begin){
734             for(size_t i=begin; i<tok_stream.size(); i++){
735                 tok_stream_.push_back(tok_stream[i]);
736             }
737         }
738         void destruct(){
739             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
740         }
741
742         inline void setDelimiter(Operator op){
743             delimiter_ = op;
744         }
745
746         pair<Modulus, size_t> getModulus();
747
748     private:
749         vector<Token> tok_stream_;
750         Operator delimiter_;
751
752         size_t parse_(size_t begin, Modulus &expr);
753
754 };
755
756 class FactorDivParser{
757     public:
758         FactorDivParser(const vector<Token> &tok_stream, size_t begin){
759             for(size_t i=begin; i<tok_stream.size(); i++){
760                 tok_stream_.push_back(tok_stream[i]);
761             }
762         }
763         FactorDivParser(const vector<Token> &tok_stream)
764             :tok_stream_(tok_stream){}
```

```
765         FactorDivParser(){}
766
767         void construct(const vector<Token> &tok_stream, size_t begin){
768             for(size_t i=begin; i<tok_stream.size(); i++){
769                 tok_stream_.push_back(tok_stream[i]);
770             }
771         }
772         void destruct(){
773             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
774         }
775
776         inline void setDelimiter(Operator op){
777             delimiter_ = op;
778         }
779
780         pair<FactorDiv, size_t> getFactorDiv();
781
782     private:
783         vector<Token> tok_stream_;
784         Operator delimiter_;
785
786         size_t parse_(size_t begin, FactorDiv &expr);
787
788 };
789
790 class FactorMultParser{
791     public:
792         FactorMultParser(const vector<Token> &tok_stream, size_t begin){
793             for(size_t i=begin; i<tok_stream.size(); i++){
794                 tok_stream_.push_back(tok_stream[i]);
795             }
796         }
797         FactorMultParser(const vector<Token> &tok_stream)
798             :tok_stream_(tok_stream){}
799         FactorMultParser(){}
800
801         void construct(const vector<Token> &tok_stream, size_t begin){
802             for(size_t i=begin; i<tok_stream.size(); i++){
803                 tok_stream_.push_back(tok_stream[i]);
804             }
805         }
806         void destruct(){
807             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
808         }
809
```

```
810         inline void setDelimiter(Operator op){
811             delimiter_ = op;
812         }
813 
814         pair<FactorMult, size_t> getFactorMult();
815 
816     private:
817         vector<Token> tok_stream_;
818         Operator delimiter_;
819 
820         size_t parse_(size_t begin, FactorMult &expr);
821 
822 };
823 
824 class TermAddParser{
825     public:
826         TermAddParser(const vector<Token> &tok_stream, size_t begin){
827             for(size_t i=begin; i<tok_stream.size(); i++){
828                 tok_stream_.push_back(tok_stream[i]);
829             }
830         }
831         TermAddParser(const vector<Token> &tok_stream)
832             :tok_stream_(tok_stream){}
833         TermAddParser(){}
834 
835         void construct(const vector<Token> &tok_stream, size_t begin){
836             for(size_t i=begin; i<tok_stream.size(); i++){
837                 tok_stream_.push_back(tok_stream[i]);
838             }
839         }
840         void destruct(){
841             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
842         }
843 
844         inline void setDelimiter(Operator op){
845             delimiter_ = op;
846         }
847 
848         pair<TermAdd, size_t> getTermAdd();
849 
850     private:
851         vector<Token> tok_stream_;
852         Operator delimiter_;
853 
854         size_t parse_(size_t begin, TermAdd &expr);
```

```
855
856     };
857
858     class TermSubParser{
859         public:
860             TermSubParser(const vector<Token> &tok_stream, size_t begin){
861                 for(size_t i=begin; i<tok_stream.size(); i++){
862                     tok_stream_.push_back(tok_stream[i]);
863                 }
864             }
865             TermSubParser(const vector<Token> &tok_stream)
866                 :tok_stream_(tok_stream){}
867             TermSubParser(){}
868
869             void construct(const vector<Token> &tok_stream, size_t begin){
870                 for(size_t i=begin; i<tok_stream.size(); i++){
871                     tok_stream_.push_back(tok_stream[i]);
872                 }
873             }
874             void destruct(){
875                 vector<Token>().swap(tok_stream_);       //clear and dellocate memory
876             }
877
878             inline void setDelimiter(Operator op){
879                 delimiter_ = op;
880             }
881
882             pair<TermSub, size_t> getTermSub();
883
884         private:
885             vector<Token> tok_stream_;
886             Operator delimiter_;
887
888             size_t parse_(size_t begin, TermSub &expr);
889
890     };
891
892     class ComponentParser{
893         public:
894             ComponentParser(const vector<Token> &tok_stream, size_t begin){
895                 for(size_t i=begin; i<tok_stream.size(); i++){
896                     tok_stream_.push_back(tok_stream[i]);
897                 }
898             }
899             ComponentParser(const vector<Token> &tok_stream)
```

```
900             :tok_stream_(tok_stream){}
901         ComponentParser(){}
902
903         void construct(const vector<Token> &tok_stream, size_t begin){
904             for(size_t i=begin; i<tok_stream.size(); i++){
905                 tok_stream_.push_back(tok_stream[i]);
906             }
907         }
908         void destruct(){
909             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
910         }
911
912         inline void setDelimiter(Operator op){
913             delimiter_ = op;
914         }
915
916         pair<Component, size_t> getComponent();
917
918     private:
919         vector<Token> tok_stream_;
920         Operator delimiter_;
921
922         size_t parse_(size_t begin, Component &expr);
923
924 };
925
926 class LExprParser{
927     public:
928         LExprParser(const vector<Token> &tok_stream, size_t begin){
929             for(size_t i=begin; i<tok_stream.size(); i++){
930                 tok_stream_.push_back(tok_stream[i]);
931             }
932         }
933         LExprParser(const vector<Token> &tok_stream):tok_stream_(tok_stream){}
934         LExprParser(){
935             delimiter_ = RCURL;
936         }
937
938         void construct(const vector<Token> &tok_stream, size_t begin){
939             for(size_t i=begin; i<tok_stream.size(); i++){
940                 tok_stream_.push_back(tok_stream[i]);
941             }
942         }
943         void destruct(){
944             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
```

```
945         }
946
947         inline void setDelimiter(Operator op){
948             delimiter_ = op;
949         }
950
951         pair<LExpr *,size_t> getLExpr();
952
953     private:
954         const char *expr_;
955         vector<Token> tok_stream_;
956         Token tok_;
957         LExpr *lexpr_;
958         bit32 curl_match_;
959         Operator delimiter_;
960
961         size_t parse_(size_t begin, LExpr *lexpr);
962 };
963
964 class VariableParser{
965     public:
966         VariableParser(const vector<Token> &tok_stream, size_t begin){
967             for(size_t i=begin; i<tok_stream.size(); i++){
968                 tok_stream_.push_back(tok_stream[i]);
969             }
970         }
971         VariableParser(const vector<Token> &tok_stream):tok_stream_(tok_stream){}
972         VariableParser(){}
973
974         void construct(const vector<Token> &tok_stream, size_t begin){
975             for(size_t i=begin; i<tok_stream.size(); i++){
976                 tok_stream_.push_back(tok_stream[i]);
977             }
978         }
979         void destruct(){
980             vector<Token>().swap(tok_stream_);       //clear and dellocate memory
981         }
982
983         inline void setDelimiter(Operator op){
984             delimiter_ = op;
985         }
986
987         pair<Variable,size_t> getVariable();
988
989
```

```
990         private:
991             vector<Token> tok_stream_;
992             Token tok_;
993             Operator delimiter_;
994 
995             size_t parse_(size_t begin, Variable &var);
996     };
997 
998 
999     class SExprParser{
1000        public:
1001            SExprParser(const char *expr):expr_(expr),lex_(expr){
1002                while(lex_.next(tok_)) tok_stream_.push_back(tok_);
1003                bareDelimited_ = false;
1004            }
1005            SExprParser(const vector<Token> &tok_stream, size_t begin){
1006                for(size_t i=begin; i<tok_stream.size(); i++){
1007                    tok_stream_.push_back(tok_stream[i]);
1008                }
1009                bareDelimited_ = false;
1010            }
1011            SExprParser(const vector<Token> &tok_stream):tok_stream_(tok_stream){
1012                bareDelimited_ = false;
1013            }
1014            SExprParser(){
1015                bareDelimited_ = false;
1016            }
1017 
1018            void construct(const vector<Token> &tok_stream, size_t begin){
1019                for(size_t i=begin; i<tok_stream.size(); i++){
1020                    tok_stream_.push_back(tok_stream[i]);
1021                }
1022                delimiter_ = RPAREN;
1023            }
1024            void destruct(){
1025                vector<Token>().swap(tok_stream_);      //clear and dellocate memory
1026            }
1027 
1028            inline void setDelimiter(Operator op){
1029                delimiter_ = op;
1030            }
1031 
1032            inline void setBareDelimited(){
1033                bareDelimited_ = true;
1034            }
```

```
1035
1036            pair<SExpr *,size_t> getSExpr();
1037
1038        private:
1039            const char *expr_;
1040            vector<Token> tok_stream_;
1041            Token tok_;
1042            SExpr *sexpr_;
1043            Tokenizer lex_;
1044            bit32 paren_match;
1045            Operator delimiter_;
1046            bool bareDelimited_;
1047
1048            size_t parse_(size_t begin, SExpr *sexpr);
1049    };
1050
1051
1052    class StatementParser{
1053        public:
1054            StatementParser(const char *expr):expr_(expr){
1055                Tokenizer lex_(expr);
1056                Token t;
1057                vector<Token> tok_stream_l;
1058                vector<Token> tok_stream_r;
1059
1060                stmt_ = new Statement;
1061
1062                while(lex_.next(t)){
1063                    if(t.type_ == OPERATOR){
1064                        if(t.token_.op == ASSIGN){
1065                            break;
1066                        }
1067                        else tok_stream_l.push_back(t);
1068                    }
1069                    else tok_stream_l.push_back(t);
1070                }
1071                SExprParser lval_parser_(tok_stream_l);
1072                stmt_->lval_ = lval_parser_.getSExpr().first;
1073                int count = 0;
1074                while(lex_.next(t)){
1075                    count++;
1076                    tok_stream_r.push_back(t);
1077                }
1078                if(!count)stmt_->store_=false;
1079                else{
```

```
1080                SExprParser rval_parser_(tok_stream_r);
1081                stmt_->rval_ = rval_parser_.getSExpr().first;
1082            }
1083        }
1084
1085        inline Statement *getStmt(){
1086            return stmt_;
1087        }
1088
1089
1090    private:
1091        const char *expr_;
1092        vector <Token> tok_stream_;
1093        Statement *stmt_;
1094 };
1095
1096 #endif
1097
1098 /*************************************************
1099                         PARSER.CPP
1100 *************************************************/
1101
1102 #include "parser.hpp"
1103 using namespace std;
1104 size_t SExprParser::parse_(size_t begin, SExpr *sexpr){
1105     size_t i;
1106     atom myAtom;
1107     SExpr *newSexpr;
1108     LExprParser lParser;
1109     VariableParser vParser;
1110     pair<Variable, size_t> vres;
1111     Variable *var;
1112     SExprParser sParser;
1113     pair<SExpr *, size_t> sres;
1114     pair<LExpr *,size_t> lres;
1115
1116
1117     for(i=begin; i<tok_stream_.size(); i++){
1118         //tok_stream_[i].print();
1119         if(tok_stream_[i].type_ == OPERATOR){
1120             switch(tok_stream_[i].token_.op){
1121                 case LPAREN:
1122                     paren_match++;
1123                     sParser.construct(tok_stream_, ++i);
1124                     sParser.setDelimiter(RPAREN);
```

```
1125                    sres = sParser.getSExpr();
1126                    i += sres.second;
1127                    newSexpr = sres.first;
1128                    myAtom.type_=SEXPR;
1129                    myAtom.atom_ = (void *)newSexpr;
1130                    sexpr->sexpr_.push_back(myAtom);
1131                    break;
1132                case LCURL:
1133                    lParser.construct(tok_stream_, ++i);
1134                    lParser.setDelimiter(RCURL);
1135                    lres = lParser.getLExpr();
1136                    i += lres.second;
1137                    myAtom.type_ = LEXPR;
1138                    myAtom.atom_ = (void *)lres.first;
1139                    sexpr->sexpr_.push_back(myAtom);
1140                    break;
1141                default:
1142                    if(bareDelimited_){
1143                        return i;
1144                    }
1145                    if(tok_stream_[i].token_.op == delimiter_){
1146                        return ++i;
1147                    }
1148                    myAtom.type_=TOKEN;
1149                    Token *myToken = new Token(tok_stream_[i]);
1150                    myAtom.atom_ = (void *)myToken;
1151                    sexpr->sexpr_.push_back(myAtom);
1152                    break;
1153            }
1154        }
1155        else if(tok_stream_[i].type_ == VAR_MARKER){
1156            vParser.construct(tok_stream_, ++i);
1157            vres = vParser.getVariable();
1158            i += vres.second;
1159            var = new Variable(vres.first);
1160            myAtom.type_ = VARIABLE;
1161            myAtom.atom_ = (void *)var;
1162            sexpr->sexpr_.push_back(myAtom);
1163        }
1164        else{
1165            myAtom.type_ = TOKEN;
1166            Token *myToken = new Token(tok_stream_[i]);
1167            myAtom.atom_ = (void *)myToken;
1168            sexpr->sexpr_.push_back(myAtom);
1169        }
```

```
1170        }
1171        //cout<<"\n\n";
1172        return i;
1173    }
1174
1175
1176    pair<SExpr *,size_t> SExprParser::getSExpr(){
1177        pair<SExpr *, size_t> ret;
1178        paren_match = 0;
1179        ret.first = new SExpr;
1180        ret.second = parse_(0,ret.first);
1181        destruct();
1182        /*if(paren_match){
1183            I_error e("Parentheses mismatch");
1184            e.print();
1185        }*/
1186        return ret;
1187    }
1188
1189
1190    pair<Variable, size_t> VariableParser::getVariable(){
1191        pair<Variable, size_t> ret;
1192        Variable var;
1193        ret.first = var;
1194        ret.second = parse_(0, ret.first);
1195        destruct();
1196        return ret;
1197    }
1198
1199    size_t VariableParser::parse_(size_t begin, Variable &var){
1200        size_t i;
1201        LExprParser lParser;
1202        pair<LExpr *,size_t> lres;
1203
1204        i=begin;
1205        if(tok_stream_[i].type_ == OPERATOR){
1206            if(tok_stream_[i].token_.op==LBRACKET){
1207                lParser.construct(tok_stream_,++i);
1208                lParser.setDelimiter(RBRACKET);
1209                lres = lParser.getLExpr();
1210                i += lres.second;
1211
1212                var.context_ = lres.first;
1213                //i++;
1214            }
```

```
1215            else{
1216                I_error e("1 Wrong usage of VARIABLE");
1217                e.print();
1218            }
1219        }
1220        else if(tok_stream_[i].type_==SYMBOL);
1221        else{
1222            I_error e("2 Wrong usage of VARIABLE");
1223            e.print();
1224        }
1225        //tok_stream_[i].print();
1226        if(tok_stream_[i].type_ == SYMBOL){
1227            strcpy(var.var_name_, tok_stream_[i].token_.sym);
1228            return ++i;
1229        }
1230        else{
1231            I_error e("3 Wrong usage of VARIABLE");
1232            e.print();
1233        }
1234    }
1235
1236
1237
1238
1239    pair<Logical_not, size_t> Logical_notParser::getLogical_not(){
1240        pair<Logical_not, size_t> ret;
1241        Logical_not expr;
1242        ret.first = expr;
1243        ret.second = parse_(0, ret.first);
1244        destruct();
1245        return ret;
1246    }
1247
1248    size_t Logical_notParser::parse_(size_t begin, Logical_not &expr){
1249        size_t i = begin;
1250        SExprParser sParser;
1251        VariableParser vParser;
1252        LExprParser lParser;
1253        Variable *var;
1254        pair<Variable, size_t> vres;
1255        pair<SExpr *, size_t> sres;
1256        pair<LExpr *, size_t> lres;
1257        atom myAtom;
1258
1259        if(tok_stream_[begin].type_==OPERATOR){
```

```
1260            if(tok_stream_[begin].token_.op==NOT){
1261                expr.not_ = true;
1262                i++;
1263            }
1264            else expr.not_ = false;
1265        }
1266
1267        size_t count = 0,j;
1268        for(j=i; j<tok_stream_.size(); j++){
1269            //tok_stream_[j].print();
1270            if(tok_stream_[j].type_ == VAR_MARKER){
1271                vParser.construct(tok_stream_, ++j);
1272                vres = vParser.getVariable();
1273                j += vres.second;
1274            }
1275
1276            if(tok_stream_[j].type_==OPERATOR){
1277                if(tok_stream_[j].token_.op == LPAREN){
1278                    sParser.construct(tok_stream_, ++j);
1279                    sres = sParser.getSExpr();
1280                    j += sres.second;
1281                    j--;
1282                }
1283                else if(tok_stream_[j].token_.op == LCURL){
1284                    lParser.construct(tok_stream_, ++j);
1285                    lres = lParser.getLExpr();
1286                    j += lres.second;
1287                    j--;
1288                }
1289                else if(tok_stream_[j].token_.op == delimiter_){
1290                    //count--;
1291                    break;
1292                }
1293                else{
1294                    break;
1295                }
1296            }
1297            count++;
1298        }
1299        //cout<<"\n\n";
1300        if(count > 1){
1301            sParser.construct(tok_stream_, i);
1302            sParser.setBareDelimited();
1303            //cout<<endl<<"BARE DELIMITED SET\n"<<endl;
1304            sres = sParser.getSExpr();
```

```
1305            i += sres.second;
1306            myAtom.type_ = SEXPR;
1307            myAtom.atom_ = (void *)sres.first;
1308            expr.logical_not_.push_back(myAtom);
1309        }
1310        else{
1311            //tok_stream_[i].print();
1312            if(tok_stream_[i].type_ == VAR_MARKER){
1313                myAtom.type_ = VARIABLE;
1314                var = new Variable(vres.first);
1315                myAtom.atom_ = (void *)var;
1316                expr.logical_not_.push_back(myAtom);
1317            }
1318            else if(tok_stream_[i].type_ == OPERATOR){
1319                if(tok_stream_[i].token_.op == LPAREN){
1320                    myAtom.type_ = SEXPR;
1321                    myAtom.atom_ = (void *)(sres.first);
1322                    expr.logical_not_.push_back(myAtom);
1323                }
1324                else if(tok_stream_[i].token_.op == LCURL){
1325                    myAtom.type_ = LEXPR;
1326                    myAtom.atom_ = (void *)(lres.first);
1327                    expr.logical_not_.push_back(myAtom);
1328                }
1329                else if(tok_stream_[i].token_.op == delimiter_){
1330                    return ++i;
1331                }
1332                else{
1333                    I_error e("WRONG FORMAT OF ARITHMATIC EXPRESSION, AMBIGUOUS");
1334                    e.print();
1335                }
1336            }
1337            else{
1338                myAtom.type_ = TOKEN;
1339                Token *myTok = new Token(tok_stream_[i]);
1340                myAtom.atom_ = (void *)myTok;
1341                expr.logical_not_.push_back(myAtom);
1342            }
1343            //cout<<"\n\n"<<j<<"\n\n";
1344        }
1345        return j;
1346    }
1347
1348
1349    pair<Logical_xor, size_t> Logical_xorParser::getLogical_xor(){
```

```
1350     pair<Logical_xor, size_t> ret;
1351     Logical_xor expr;
1352     ret.first = expr;
1353     ret.second = parse_(0, ret.first);
1354     destruct();
1355     return ret;
1356 }
1357
1358 size_t Logical_xorParser::parse_(size_t begin, Logical_xor &expr){
1359     Logical_notParser lnParser;
1360     pair<Logical_not, size_t> lnres;
1361
1362     size_t i=begin;
1363     lnParser.construct(tok_stream_, i);
1364     lnParser.setDelimiter(delimiter_);
1365     lnres = lnParser.getLogical_not();
1366
1367     i += lnres.second;
1368     expr.logical_xor_.push_back(lnres.first);
1369
1370
1371     while(tok_stream_[i].type_ == OPERATOR){
1372         if(tok_stream_[i].token_.op == XOR){
1373             lnParser.construct(tok_stream_, ++i);
1374             lnParser.setDelimiter(delimiter_);
1375             lnres = lnParser.getLogical_not();
1376
1377             i+= lnres.second;
1378             expr.logical_xor_.push_back(lnres.first);
1379         }
1380         else break;
1381     }
1382
1383     return i;
1384 }
1385
1386
1387 pair<Logical_and, size_t> Logical_andParser::getLogical_and(){
1388     pair<Logical_and, size_t> ret;
1389     Logical_and expr;
1390     ret.first = expr;
1391     ret.second = parse_(0, ret.first);
1392     destruct();
1393     return ret;
1394 }
```

```
1395
1396 size_t Logical_andParser::parse_(size_t begin, Logical_and &expr){
1397     Logical_xorParser lxParser;
1398     pair<Logical_xor, size_t> lxres;
1399
1400     size_t i=begin;
1401     lxParser.construct(tok_stream_, i);
1402     lxParser.setDelimiter(delimiter_);
1403     lxres = lxParser.getLogical_xor();
1404
1405     i += lxres.second;
1406     expr.logical_and_.push_back(lxres.first);
1407
1408
1409     while(tok_stream_[i].type_ == OPERATOR){
1410         if(tok_stream_[i].token_.op == AND){
1411             lxParser.construct(tok_stream_, ++i);
1412             lxParser.setDelimiter(delimiter_);
1413             lxres = lxParser.getLogical_xor();
1414
1415             i+= lxres.second;
1416             expr.logical_and_.push_back(lxres.first);
1417         }
1418         else break;
1419     }
1420
1421     return i;
1422 }
1423
1424 pair<Logical_or, size_t> Logical_orParser::getLogical_or(){
1425     pair<Logical_or, size_t> ret;
1426     Logical_or expr;
1427     ret.first = expr;
1428     ret.second = parse_(0, ret.first);
1429     destruct();
1430     return ret;
1431 }
1432
1433 size_t Logical_orParser::parse_(size_t begin, Logical_or &expr){
1434     Logical_andParser loParser;
1435     pair<Logical_and, size_t> lores;
1436
1437     size_t i=begin;
1438     loParser.construct(tok_stream_, i);
1439     loParser.setDelimiter(delimiter_);
```

```
1440        lores = loParser.getLogical_and();
1441
1442        i += lores.second;
1443        expr.logical_or_.push_back(lores.first);
1444
1445
1446        while(tok_stream_[i].type_ == OPERATOR){
1447            if(tok_stream_[i].token_.op == OR){
1448                loParser.construct(tok_stream_, ++i);
1449                loParser.setDelimiter(delimiter_);
1450                lores = loParser.getLogical_and();
1451
1452                i+= lores.second;
1453                expr.logical_or_.push_back(lores.first);
1454            }
1455            else break;
1456        }
1457
1458        return i;
1459    }
1460
1461    pair<Unary, size_t> UnaryParser::getUnary(){
1462        pair<Unary, size_t> ret;
1463        Unary expr;
1464        ret.first = expr;
1465        ret.second = parse_(0, ret.first);
1466        destruct();
1467        return ret;
1468    }
1469
1470    size_t UnaryParser::parse_(size_t begin, Unary &expr){
1471        Logical_orParser loParser;
1472        pair<Logical_or, size_t> lores;
1473        size_t i = begin;
1474        bool unary_op_ = false;
1475
1476        if(tok_stream_[i].type_ == OPERATOR){
1477            if(tok_stream_[i].token_.op==PLUS || tok_stream_[i].token_.op==MINUS){
1478                unary_op_ = true;
1479                if(tok_stream_[i].token_.op == MINUS) expr.minus_ =true;
1480                loParser.construct(tok_stream_, ++i);
1481                loParser.setDelimiter(delimiter_);
1482                lores = loParser.getLogical_or();
1483                i += lores.second;
1484                expr.unary_.push_back(lores.first);
```

```
1485                return i;
1486            }
1487        }
1488
1489        if(! unary_op_){
1490            loParser.construct(tok_stream_, i);
1491            loParser.setDelimiter(delimiter_);
1492            lores = loParser.getLogical_or();
1493            i += lores.second;
1494            expr.unary_.push_back(lores.first);
1495            return i;
1496        }
1497    }
1498
1499
1500    pair<Exponent, size_t> ExponentParser::getExponent(){
1501        pair<Exponent, size_t> ret;
1502        Exponent expr;
1503        ret.first = expr;
1504        ret.second = parse_(0, ret.first);
1505        destruct();
1506        return ret;
1507    }
1508
1509    size_t ExponentParser::parse_(size_t begin, Exponent &expr){
1510        UnaryParser unParser;
1511        pair<Unary, size_t> unres;
1512
1513        size_t i=begin;
1514        unParser.construct(tok_stream_, i);
1515        unParser.setDelimiter(delimiter_);
1516        unres = unParser.getUnary();
1517
1518        i += unres.second;
1519        expr.exponent_.push_back(unres.first);
1520
1521
1522        while(tok_stream_[i].type_ == OPERATOR){
1523            if(tok_stream_[i].token_.op == POW){
1524                unParser.construct(tok_stream_, ++i);
1525                unParser.setDelimiter(delimiter_);
1526                unres = unParser.getUnary();
1527
1528                i+= unres.second;
1529                expr.exponent_.push_back(unres.first);
```

```
1530            }
1531            else break;
1532        }
1533
1534        return i;
1535    }
1536
1537
1538
1539    pair<Modulus, size_t> ModulusParser::getModulus(){
1540        pair<Modulus, size_t> ret;
1541        Modulus expr;
1542        ret.first = expr;
1543        ret.second = parse_(0, ret.first);
1544        destruct();
1545        return ret;
1546    }
1547
1548    size_t ModulusParser::parse_(size_t begin, Modulus &expr){
1549        ExponentParser exParser;
1550        pair<Exponent, size_t> exres;
1551
1552        size_t i=begin;
1553        exParser.construct(tok_stream_, i);
1554        exParser.setDelimiter(delimiter_);
1555        exres = exParser.getExponent();
1556
1557        i += exres.second;
1558        expr.modulus_.push_back(exres.first);
1559
1560
1561        while(tok_stream_[i].type_ == OPERATOR){
1562            if(tok_stream_[i].token_.op == MOD){
1563                exParser.construct(tok_stream_, ++i);
1564                exParser.setDelimiter(delimiter_);
1565                exres = exParser.getExponent();
1566
1567                i+= exres.second;
1568                expr.modulus_.push_back(exres.first);
1569            }
1570            else break;
1571        }
1572
1573        return i;
1574    }
```

```
1575
1576 pair<FactorDiv, size_t> FactorDivParser::getFactorDiv(){
1577     pair<FactorDiv, size_t> ret;
1578     FactorDiv expr;
1579     ret.first = expr;
1580     ret.second = parse_(0, ret.first);
1581     destruct();
1582     return ret;
1583 }
1584
1585 size_t FactorDivParser::parse_(size_t begin, FactorDiv &expr){
1586     ModulusParser moParser;
1587     pair<Modulus, size_t> mores;
1588
1589     size_t i=begin;
1590     moParser.construct(tok_stream_, i);
1591     moParser.setDelimiter(delimiter_);
1592     mores = moParser.getModulus();
1593
1594     i += mores.second;
1595     expr.factor_div_.push_back(mores.first);
1596
1597
1598     while(tok_stream_[i].type_ == OPERATOR){
1599         if(tok_stream_[i].token_.op == DIV){
1600             moParser.construct(tok_stream_, ++i);
1601             moParser.setDelimiter(delimiter_);
1602             mores = moParser.getModulus();
1603
1604             i+= mores.second;
1605             expr.factor_div_.push_back(mores.first);
1606         }
1607         else break;
1608     }
1609
1610     return i;
1611 }
1612
1613
1614 pair<FactorMult, size_t> FactorMultParser::getFactorMult(){
1615     pair<FactorMult, size_t> ret;
1616     FactorMult expr;
1617     ret.first = expr;
1618     ret.second = parse_(0, ret.first);
1619     destruct();
```

```
1620        return ret;
1621    }
1622
1623    size_t FactorMultParser::parse_(size_t begin, FactorMult &expr){
1624        FactorDivParser fdParser;
1625        pair<FactorDiv, size_t> fdres;
1626
1627        size_t i=begin;
1628
1629        fdParser.construct(tok_stream_, i);
1630        fdParser.setDelimiter(delimiter_);
1631        fdres = fdParser.getFactorDiv();
1632
1633        i += fdres.second;
1634        expr.factor_mult_.push_back(fdres.first);
1635
1636        while(tok_stream_[i].type_ == OPERATOR){
1637            if(tok_stream_[i].token_.op == MULT){
1638                fdParser.construct(tok_stream_, ++i);
1639                fdParser.setDelimiter(delimiter_);
1640                fdres = fdParser.getFactorDiv();
1641
1642                i+= fdres.second;
1643                expr.factor_mult_.push_back(fdres.first);
1644            }
1645            else break;
1646        }
1647
1648        return i;
1649    }
1650
1651    pair<TermAdd, size_t> TermAddParser::getTermAdd(){
1652        pair<TermAdd, size_t> ret;
1653        TermAdd expr;
1654        ret.first = expr;
1655        ret.second = parse_(0, ret.first);
1656        destruct();
1657        return ret;
1658    }
1659
1660    size_t TermAddParser::parse_(size_t begin, TermAdd &expr){
1661        FactorMultParser fmParser;
1662        pair<FactorMult, size_t> fmres;
1663
1664        size_t i=begin;
```

```
1665        fmParser.construct(tok_stream_, i);
1666        fmParser.setDelimiter(delimiter_);
1667        fmres = fmParser.getFactorMult();
1668
1669        i += fmres.second;
1670        expr.term_add_.push_back(fmres.first);
1671
1672        while(tok_stream_[i].type_ == OPERATOR){
1673            if(tok_stream_[i].token_.op == PLUS){
1674
1675                fmParser.construct(tok_stream_, ++i);
1676                fmParser.setDelimiter(delimiter_);
1677                fmres = fmParser.getFactorMult();
1678
1679                i+= fmres.second;
1680                expr.term_add_.push_back(fmres.first);
1681            }
1682            else break;
1683        }
1684
1685        return i;
1686    }
1687
1688    pair<TermSub, size_t> TermSubParser::getTermSub(){
1689        pair<TermSub, size_t> ret;
1690        TermSub expr;
1691        ret.first = expr;
1692        ret.second = parse_(0, ret.first);
1693        destruct();
1694        return ret;
1695    }
1696
1697    size_t TermSubParser::parse_(size_t begin, TermSub &expr){
1698        TermAddParser taParser;
1699        pair<TermAdd, size_t> tares;
1700
1701        size_t i=begin;
1702        taParser.construct(tok_stream_, i);
1703        taParser.setDelimiter(delimiter_);
1704        tares = taParser.getTermAdd();
1705
1706        i += tares.second;
1707        expr.term_sub_.push_back(tares.first);
1708
1709
```

```
1710        while(tok_stream_[i].type_ == OPERATOR){
1711            if(tok_stream_[i].token_.op == MINUS){
1712                taParser.construct(tok_stream_, ++i);
1713                taParser.setDelimiter(delimiter_);
1714                tares = taParser.getTermAdd();
1715
1716                i+= tares.second;
1717                expr.term_sub_.push_back(tares.first);
1718            }
1719            else break;
1720        }
1721
1722        return i;
1723    }
1724
1725    pair<Component, size_t> ComponentParser::getComponent(){
1726        pair<Component, size_t> ret;
1727        Component expr;
1728        ret.first = expr;
1729        ret.second = parse_(0, ret.first);
1730        destruct();
1731        return ret;
1732    }
1733
1734    size_t ComponentParser::parse_(size_t begin, Component &expr){
1735        TermSubParser tsParser;
1736        pair<TermSub, size_t> tsres;
1737        size_t i = begin;
1738        tsParser.construct(tok_stream_, i);
1739        tsParser.setDelimiter(delimiter_);
1740
1741        tsres = tsParser.getTermSub();
1742        i += tsres.second;
1743        expr.component_.push_back(tsres.first);
1744
1745        if(tok_stream_[i].type_ == OPERATOR){
1746            if(tok_stream_[i].token_.op == COND){
1747                tsParser.construct(tok_stream_, ++i);
1748                tsParser.setDelimiter(delimiter_);
1749                tsres = tsParser.getTermSub();
1750                i += tsres.second;
1751                expr.component_.push_back(tsres.first);
1752
1753                if(tok_stream_[i].type_ == OPERATOR){
1754                    if(tok_stream_[i].token_.op == ELSE){
```

```
1755                        tsParser.construct(tok_stream_, ++i);
1756                        tsParser.setDelimiter(delimiter_);
1757                        tsres = tsParser.getTermSub();
1758                        expr.component_.push_back(tsres.first);
1759                        //tsres.first.print();
1760                        //cout<<"\n\n";
1761                        i += tsres.second;
1762
1763                    }
1764                }
1765            }
1766        }
1767        return i;
1768    }
1769
1770
1771    pair<LExpr *, size_t> LExprParser::getLExpr(){
1772        pair<LExpr *, size_t> ret;
1773        ret.first = new LExpr;
1774        ret.second = parse_(0, ret.first);
1775        destruct();
1776        return ret;
1777    }
1778
1779    size_t LExprParser::parse_(size_t begin, LExpr *lexpr){
1780        ComponentParser coParser;
1781        pair<Component, size_t> cores;
1782        size_t i = begin;
1783
1784        coParser.construct(tok_stream_, i);
1785        coParser.setDelimiter(delimiter_);
1786        cores = coParser.getComponent();
1787        i += cores.second;
1788        lexpr->lexpr_.push_back(cores.first);
1789
1790        while(tok_stream_[i].type_ == OPERATOR){
1791            if(tok_stream_[i].token_.op == COMMA){
1792                coParser.construct(tok_stream_, ++i);
1793                coParser.setDelimiter(delimiter_);
1794                cores = coParser.getComponent();
1795
1796                i += cores.second;
1797                lexpr->lexpr_.push_back(cores.first);
1798            }
1799            else if(tok_stream_[i].token_.op == delimiter_)
```

```
1800            return ++i;
1801        else break;
1802    }
1803    return i;
1804 }
1805
1806
1807 void atom::print(){
1808    Token *tok;
1809    SExpr *sexpr;
1810    LExpr *lexpr;
1811    Variable *var;
1812    cout<<"<ATOM>"<<endl;
1813    switch(type_){
1814        case TOKEN:
1815            //cout<<"<ATOM>"<<endl;
1816            tok = (Token *)atom_;
1817            tok->print();
1818            //cout<<"</ATOM>"<<endl;
1819            break;
1820        case SEXPR:
1821            //cout<<"<S-EXPR>"<<endl;
1822            sexpr = (SExpr *)atom_;
1823            sexpr->print();
1824            //cout<<"</S-EXPR>"<<endl;
1825            break;
1826        case LEXPR:
1827            //cout<<"<L-EXPR>"<<endl;
1828            lexpr = (LExpr *)atom_;
1829            lexpr->print();
1830            //cout<<"</L-EXPR>"<<endl;
1831            break;
1832        case VARIABLE:
1833            //cout<<"<VARIABLE>"<<endl;
1834            var = (Variable *)atom_;
1835            var->print();
1836            //cout<<"</VARIABLE>"<<endl;
1837            break;
1838        default:
1839            cout<<"<ERROR/> "<<endl;
1840    }
1841    cout<<"</ATOM>"<<endl;
1842 }
1843
1844 void Logical_not::print(){
```

```
1845        cout<<"<LOGICAL_NOT is_not=\""<<(not_?"true":"false")<<"\">"<<endl;
1846        for(size_t i=0; i<logical_not_.size(); i++)
1847            logical_not_[i].print();
1848        cout<<"</LOGICAL_NOT>"<<endl;
1849    }
1850
1851    void Logical_xor::print(){
1852        cout<<"<LOGICAL_XOR>"<<endl;
1853        for(size_t i=0; i<logical_xor_.size(); i++)
1854            logical_xor_[i].print();
1855        cout<<"</LOGICAL_XOR>"<<endl;
1856    }
1857
1858
1859    void Logical_and::print(){
1860        cout<<"<LOGICAL_AND>"<<endl;
1861        for(size_t i=0; i<logical_and_.size(); i++)
1862            logical_and_[i].print();
1863        cout<<"</LOGICAL_AND>"<<endl;
1864    }
1865
1866    void Logical_or::print(){
1867        cout<<"<LOGICAL_OR>"<<endl;
1868        for(size_t i=0; i<logical_or_.size(); i++)
1869            logical_or_[i].print();
1870        cout<<"</LOGICAL_OR>"<<endl;
1871    }
1872
1873    void Unary::print(){
1874        cout<<"<UNARY-OPERATOR is_minus=\""<<(minus_?"true":"false")<<"\">"<<endl;
1875        for(size_t i=0; i<unary_.size(); i++)
1876            unary_[i].print();
1877        cout<<"</UNARY-OPERATOR>"<<endl;
1878    }
1879
1880    void Exponent::print(){
1881        cout<<"<EXPONENT>"<<endl;
1882        for(size_t i=0; i<exponent_.size(); i++)
1883            exponent_[i].print();
1884        cout<<"</EXPONENT>"<<endl;
1885    }
1886
1887    void Modulus::print(){
1888        cout<<"<MODULUS>"<<endl;
1889        for(size_t i=0; i<modulus_.size(); i++)
```

```
1890            modulus_[i].print();
1891        cout<<"</MODULUS>"<<endl;
1892    }
1893
1894    void FactorDiv::print(){
1895        cout<<"<DIVISION>"<<endl;
1896        for(size_t i=0; i<factor_div_.size(); i++)
1897            factor_div_[i].print();
1898        cout<<"</DIVISION>"<<endl;
1899    }
1900
1901    void FactorMult::print(){
1902        cout<<"<MULTIPLICATION>"<<endl;
1903        for(size_t i=0; i<factor_mult_.size(); i++)
1904            factor_mult_[i].print();
1905        cout<<"</MULTIPLICATION>"<<endl;
1906    }
1907
1908    void TermAdd::print(){
1909        cout<<"<ADDITION>"<<endl;
1910        for(size_t i=0; i<term_add_.size(); i++)
1911            term_add_[i].print();
1912        cout<<"</ADDITION>"<<endl;
1913    }
1914
1915    void TermSub::print(){
1916        cout<<"<SUBTRACTION>"<<endl;
1917        for(size_t i=0; i<term_sub_.size(); i++)
1918            term_sub_[i].print();
1919        cout<<"</SUBTRACTION>"<<endl;
1920    }
1921
1922    void Component::print(){
1923        cout<<"<CONDITIONAL>"<<endl;
1924        if(component_.size()==0){
1925            cout<<"<ERROR/>"<<endl;
1926        }
1927        else if(component_.size()==1){
1928            cout<<"<ALWAYS>"<<endl;
1929            component_[0].print();
1930            cout<<"</ALWAYS>"<<endl;
1931        }
1932        else if(component_.size()==2){
1933            cout<<"<CONDITION>"<<endl;
1934            component_[0].print();
```

```
1935            cout<<"</CONDITION>"<<endl;
1936
1937            cout<<"<SATISFIED>"<<endl;
1938            component_[1].print();
1939            cout<<"</SATISFIED>"<<endl;
1940        }
1941        else if(component_.size()==3){
1942            cout<<"<CONDITION>"<<endl;
1943            component_[0].print();
1944            cout<<"</CONDITION>"<<endl;
1945
1946            cout<<"<SATISFIED>"<<endl;
1947            component_[1].print();
1948            cout<<"</SATISFIED>"<<endl;
1949
1950            cout<<"<ELSE>"<<endl;
1951            component_[2].print();
1952            cout<<"</ELSE>"<<endl;
1953        }
1954        else{
1955            cout<<"<FAULTY-CONDITIONAL>"<<endl;
1956            for(size_t i=0; i<component_.size(); i++)
1957                component_[i].print();
1958            cout<<"</FAULTY-CONDITIONAL>"<<endl;
1959        }
1960        cout<<"</CONDITIONAL>"<<endl;
1961    }
1962
1963
1964    void LExpr::print(){
1965        cout<<"<L-EXPRESSION>"<<endl;
1966        for(size_t i=0; i<lexpr_.size(); i++)
1967            lexpr_[i].print();
1968        cout<<"</L-EXPRESSION>"<<endl;
1969    }
1970
1971    void Variable::print(){
1972        cout<<"<VARIABLE>"<<endl;
1973        if(context_ != NULL){
1974            cout<<"<CONTEXT>"<<endl;
1975            context_->print();
1976            cout<<"</CONTEXT>"<<endl;
1977        }
1978        cout<<"<VAR-NAME>"<<var_name_<<"</VAR-NAME>"<<endl;
1979        cout<<"</VARIABLE>"<<endl;
```

```
}

void SExpr::print(){
    cout<<"<S-EXPRESSION>"<<endl;
    for(size_t i=0; i<sexpr_.size(); i++)
        sexpr_[i].print();
    cout<<"</S-EXPRESSION>"<<endl;
}



/*************************************************
                        MAIN.CPP
*************************************************/

#include "parser.hpp"
using namespace std;

bit64 line_no = 1;
bit64 col_no = 0;
int indentLevel = 0;

int main(){
    StatementParser parse("$s+{$[]a+$[~(-7),8,true?x:{y+z}]b}=5");
    Statement *expr = parse.getStmt();
    expr->print();
    cout<<endl;
    return 0;
}


/*************************************************
                    PARSE_TREE.XML
*************************************************/

<STATEMENT>
<L-VALUE>
<S-EXPRESSION>
<ATOM>
<VARIABLE>
<VAR-NAME>s</VAR-NAME>
</VARIABLE>
</ATOM>
<ATOM>
<L-EXPRESSION>
```

```
<CONDITIONAL>
<ALWAYS>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<VARIABLE>
<CONTEXT>
<L-EXPRESSION>
<CONDITIONAL>
<ALWAYS>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</ALWAYS>
</CONDITIONAL>
</L-EXPRESSION>
</CONTEXT>
```

```
2070    <VAR_NAME>a</VAR_NAME>
2071    </VARIABLE>
2072    </ATOM>
2073    </LOGICAL_NOT>
2074    </LOGICAL_XOR>
2075    </LOGICAL_AND>
2076    </LOGICAL_OR>
2077    </UNARY_OPERATOR>
2078    </EXPONENT>
2079    </MODULUS>
2080    </DIVISION>
2081    </MULTIPLICATION>
2082    <MULTIPLICATION>
2083    <DIVISION>
2084    <MODULUS>
2085    <EXPONENT>
2086    <UNARY_OPERATOR is_minus="false">
2087    <LOGICAL_OR>
2088    <LOGICAL_AND>
2089    <LOGICAL_XOR>
2090    <LOGICAL_NOT is_not="false">
2091    <ATOM>
2092    <VARIABLE>
2093    <CONTEXT>
2094    <L_EXPRESSION>
2095    <CONDITIONAL>
2096    <ALWAYS>
2097    <SUBTRACTION>
2098    <ADDITION>
2099    <MULTIPLICATION>
2100    <DIVISION>
2101    <MODULUS>
2102    <EXPONENT>
2103    <UNARY_OPERATOR is_minus="false">
2104    <LOGICAL_OR>
2105    <LOGICAL_AND>
2106    <LOGICAL_XOR>
2107    <LOGICAL_NOT is_not="true">
2108    <ATOM>
2109    <S_EXPRESSION>
2110    <ATOM>
2111    <OPERATOR>MINUS</OPERATOR>
2112    </ATOM>
2113    <ATOM>
2114    <INTEGER>7</INTEGER>
```

```
</ATOM>
</S-EXPRESSION>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</ALWAYS>
</CONDITIONAL>
<CONDITIONAL>
<ALWAYS>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<INTEGER>8</INTEGER>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</ALWAYS>
</CONDITIONAL>
```

```
<CONDITIONAL>
<CONDITION>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<SYMBOL>true</SYMBOL>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</CONDITION>
<SATISFIED>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<SYMBOL>x</SYMBOL>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
```

```
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</SATISFIED>
<ELSE>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<L-EXPRESSION>
<CONDITIONAL>
<ALWAYS>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<SYMBOL>y</SYMBOL>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
```

```
2250 </EXPONENT>
2251 </MODULUS>
2252 </DIVISION>
2253 </MULTIPLICATION>
2254 <MULTIPLICATION>
2255 <DIVISION>
2256 <MODULUS>
2257 <EXPONENT>
2258 <UNARY_OPERATOR is_minus="false">
2259 <LOGICAL_OR>
2260 <LOGICAL_AND>
2261 <LOGICAL_XOR>
2262 <LOGICAL_NOT is_not="false">
2263 <ATOM>
2264 <SYMBOL>z</SYMBOL>
2265 </ATOM>
2266 </LOGICAL_NOT>
2267 </LOGICAL_XOR>
2268 </LOGICAL_AND>
2269 </LOGICAL_OR>
2270 </UNARY_OPERATOR>
2271 </EXPONENT>
2272 </MODULUS>
2273 </DIVISION>
2274 </MULTIPLICATION>
2275 </ADDITION>
2276 </SUBTRACTION>
2277 </ALWAYS>
2278 </CONDITIONAL>
2279 </L_EXPRESSION>
2280 </ATOM>
2281 </LOGICAL_NOT>
2282 </LOGICAL_XOR>
2283 </LOGICAL_AND>
2284 </LOGICAL_OR>
2285 </UNARY_OPERATOR>
2286 </EXPONENT>
2287 </MODULUS>
2288 </DIVISION>
2289 </MULTIPLICATION>
2290 </ADDITION>
2291 </SUBTRACTION>
2292 </ELSE>
2293 </CONDITIONAL>
2294 </L_EXPRESSION>
```

```
2295 </CONTEXT>
2296 <VAR_NAME>b</VAR_NAME>
2297 </VARIABLE>
2298 </ATOM>
2299 </LOGICAL_NOT>
2300 </LOGICAL_XOR>
2301 </LOGICAL_AND>
2302 </LOGICAL_OR>
2303 </UNARY_OPERATOR>
2304 </EXPONENT>
2305 </MODULUS>
2306 </DIVISION>
2307 </MULTIPLICATION>
2308 </ADDITION>
2309 </SUBTRACTION>
2310 </ALWAYS>
2311 </CONDITIONAL>
2312 </L_EXPRESSION>
2313 </ATOM>
2314 </S_EXPRESSION>
2315 </L_VALUE>
2316 <R_VALUE>
2317 <S_EXPRESSION>
2318 <ATOM>
2319 <INTEGER>5</INTEGER>
2320 </ATOM>
2321 </S_EXPRESSION>
2322 </R_VALUE>
2323 </STATEMENT>
2324
2325
2326 /*************************************************
2327                         TESTER.CPP
2328 *************************************************/
2329
2330 /*************************************************************************
2331
2332       PROTOTYPE OF I PROGRAMMING LANGUAGE INTERPRETATION SYSTEM VERSION-3
2333
2334 *************************************************************************/
2335
2336 #ifndef LEXER_H_
2337 #define LEXER_H_
2338
2339
```

```
2340 #include "includes.hpp"
2341 using namespace std;
2342
2343 bit64 line_no;
2344 bit64 col_no;
2345
2346 class I_error{
2347     public:
2348         I_error(const char *msg):
2349             error_message_(msg), line_no_(line_no), col_no_(col_no) {}
2350
2351         void print(){
2352             cout<<"ERROR : LINE="<<line_no_<<" CHARACTER="<<col_no_
2353                 <<" : "<<error_message_<<endl;
2354             exit(0);
2355         }
2356
2357     private:
2358         const char *error_message_;
2359         int line_no_;
2360         int col_no_;
2361 };
2362
2363
2364 enum TokenType{ SYMBOL = 0, INTEGER, REAL, OPERATOR, STRING , _NONE };
2365 const char *typeString(TokenType t){
2366     switch(t){
2367         case SYMBOL:
2368             return "SYMBOL";
2369         case INTEGER:
2370             return "INTEGER";
2371         case REAL:
2372             return "REAL";
2373         case OPERATOR:
2374             return "OPERATOR";
2375         case STRING:
2376             return "STRING";
2377         case _NONE:
2378             return "_NONE";
2379         default:
2380             return "ERROR ...";
2381     }
2382 }
2383
2384 enum Operator{  PLUS, MINUS, MULT, DIV, MOD, POW,
```

```
2385                    AND, OR, NOT, XOR,
2386                    GT, LT, GTE, LTE, EQ, NEQ ,
2387                    LPAREN, RPAREN, ERROR,
2388                    ESCAPE, ASSIGN, LCURL, RCURL, COMMA };
2389
2390    const char *opString(Operator op){
2391        switch(op){
2392            case PLUS:  return "PLUS";
2393            case MINUS: return "MINUS";
2394            case MULT:  return "MULT";
2395            case DIV:   return "DIV";
2396            case MOD:   return "MOD";
2397            case POW:   return "POW";
2398            case AND:   return "AND";
2399            case OR:    return "OR";
2400            case NOT:   return "NOT";
2401            case XOR:   return "XOR";
2402            case GT:    return "GT";
2403            case LT:    return "LT";
2404            case GTE:   return "GTE";
2405            case LTE:   return "LTE";
2406            case EQ:    return "EQ";
2407            case NEQ:   return "NEQ";
2408            case LPAREN:return "LPAREN";
2409            case RPAREN:return "RPAREN";
2410            case ERROR: return "ERROR";
2411            case ESCAPE:return "ESCAPE";
2412            case ASSIGN:return "ASSIGN";
2413            case LCURL: return "LCURL";
2414            case RCURL: return "RCURL";
2415            case COMMA: return "COMMA";
2416            default:    return "\0";
2417        }
2418    }
2419
2420    union TokenHolder{
2421        int integer;
2422        double real;
2423        char str[1024];
2424        char sym[1024];
2425        Operator op;
2426    };
2427
2428    Operator str2op(const char *str){
2429        switch(str[0]){
```

```
2430            case '+': return PLUS;
2431            case '-': return MINUS;
2432            case '*':
2433                if( str[1]=='*') return POW;
2434                return MULT;
2435            case '/': return DIV;
2436            case '%': return MOD;
2437            case '&': return AND;
2438            case '|': return OR;
2439            case '~':
2440                if( str[1]=='=') return NEQ;
2441                return NOT;
2442            case '^': return XOR;
2443            case '>':
2444                if( str[1]=='=') return GTE;
2445                return GT;
2446            case '<':
2447                if( str[1]=='=') return LTE;
2448                return LT;
2449            case '=':
2450                if(str[1]=='=') return EQ;
2451                return ASSIGN;
2452            case '(': return LPAREN;
2453            case ')': return RPAREN;
2454            case '{': return LCURL;
2455            case '}': return RCURL;
2456            case '\\':return ESCAPE;
2457            case ',': return COMMA;
2458            default:
2459                return ERROR;
2460        }
2461    }
2462
2463    class Token{
2464        public:
2465
2466            Token():type_(_NONE){}
2467            Token(TokenType type, const char *tok_str){
2468                makeToken(type, tok_str);
2469            }
2470
2471            Token (const Token &oldToken){
2472                type_ = oldToken.type_;
2473                token_ = oldToken.token_;
2474            }
```

```
2475
2476        void print(){
2477            cout<<"< "<<typeString(type_)<<" , ";
2478            switch(type_){
2479                case INTEGER:   cout<<token_.integer<<" >";          break;
2480                case REAL:      cout<<token_.real<<" >";             break;
2481                case SYMBOL:    cout<<token_.sym<<" >";              break;
2482                case STRING:    cout<<token_.str<<" >";              break;
2483                case OPERATOR:  cout<<opString(token_.op)<<" >";     break;
2484                case _NONE:     cout<<" >";                          break;
2485                default:        cout<<" >";                          break;
2486            }
2487        }
2488
2489        void makeToken(TokenType type, const char *tok_str){
2490            type_ = type;
2491            switch(type_){
2492                case SYMBOL:
2493                    strcpy( token_.sym, tok_str);
2494                    break;
2495                case INTEGER:
2496                    token_.integer = atoi(tok_str);
2497                    break;
2498                case REAL:
2499                    token_.real = atof(tok_str);
2500                    break;
2501                case STRING:
2502                    strcpy(token_.str, tok_str);
2503                    break;
2504                case OPERATOR:
2505                    token_.op = str2op(tok_str);
2506                    break;
2507                case _NONE:
2508                    break;
2509                default:
2510                    cout<<"ERROR OCCURED IN LEXICAL ANALYSIS"<<endl;
2511                    break;
2512            }
2513        }
2514
2515        TokenType getType(){ return type_; }
2516
2517        TokenType type_;
2518        TokenHolder token_;
2519    };
```

```
2520
2521
2522  class Tokenizer{
2523      public:
2524          Tokenizer():expr_(NULL),type_(_NONE){}
2525          Tokenizer(const char *expr):expr_(expr){}
2526
2527          bool next(Token &t){
2528              bool ret = getNextToken();
2529              t.makeToken(type_, token_);
2530              return ret;
2531          }
2532
2533          const char* getExpr(){ return expr_; }
2534
2535      private:
2536          const char *expr_;
2537          char token_[1024];
2538          TokenType type_;
2539
2540          bool isOperator(char ch){
2541              if(strchr("+-*/%^=(){}[]&|~><?:$@,\\", ch)) return true;
2542              return false;
2543          }
2544
2545          bool getNextToken(){
2546              char *temp;
2547              type_ = _NONE;
2548              temp = token_;
2549              *temp = '\0';
2550
2551              if(!*expr_) return false;
2552              while(isspace(*expr_)) {
2553                  expr_++ ;
2554                  col_no++;
2555              }
2556
2557              if( isOperator(*expr_)){
2558                  while(isOperator(*expr_)){
2559                      *temp++ = *expr_++;
2560                      col_no++;
2561                  }
2562                  type_ = OPERATOR;
2563              }
2564              else if(isalpha(*expr_)){
```

```
2565                while(isalpha(*expr_) || *expr_ == '_'){
2566                    *temp++ = *expr_++;
2567                    col_no++;
2568                }
2569                type_ = SYMBOL;
2570            }
2571            else if(isdigit(*expr_)){
2572                while(isdigit(*expr_)){
2573                    *temp++ = *expr_++;
2574                    col_no++;
2575                }
2576                if(*expr_ == '.'){
2577                    *temp++ = *expr_++;
2578                    col_no++;
2579                    while(isdigit(*expr_)){
2580                        *temp++ = *expr_++;
2581                        col_no++;
2582                    }
2583                    type_ = REAL;
2584                }
2585                else type_ = INTEGER;
2586            }
2587            else if(*expr_ == '\"'){
2588                *expr_++;
2589                col_no++;
2590                while(*expr_ != '\"'){
2591                    *temp++ = *expr_++;
2592                    col_no++;
2593                }
2594                *expr_++;
2595                col_no++;
2596                type_ = STRING;
2597            }
2598            else{
2599                while(!isspace(*expr_)){
2600                    *temp++ = *expr_++;
2601                    col_no++;
2602                }
2603                type_ = SYMBOL;
2604            }
2605
2606            *temp = '\0';
2607            return true;
2608        }
2609
```

```
2610 };
2611
2612 enum atomType{ TOKEN, SEXPR, LEXPR };
2613 struct atom{
2614     atomType type_;
2615     void *atom_;
2616 };
2617
2618
2619 //--------------------------------------------------------------------------
2620 //       FORWARD DECLARATION OF L-EXPRESSION AND ITS PARSER FOR USE IN
2621 //                       CLASSES USED FOR PARSING S-EXPRESSIONS
2622 //--------------------------------------------------------------------------
2623
2624 int indentLevel;
2625
2626 struct Logical_not{
2627     vector<atom> logical_not_;       // Unary operator ~
2628     void print();
2629 };
2630
2631 struct Logical_xor{
2632     vector<Logical_not> logical_xor_;   // values separated by ^
2633     void print();
2634 };
2635
2636 struct Logical_and{
2637     vector<Logical_xor> logical_and_;   // values separated by &
2638     void print();
2639 };
2640
2641 struct Logical_or{
2642     vector<Logical_and> logical_or_; // Values separated by |
2643     void print();
2644 };
2645
2646 struct Unary{
2647     vector<Logical_or>  unary_; // Unary operators +, -
2648     void print();
2649 };
2650
2651 struct Exponent{
2652     vector<Unary> exponent_;     // Values separated by exponent op **
2653     void print();
2654 };
```

```
2655
2656    struct Factor{                    // Values separated by *, / and %
2657        vector<Exponent> factor_;
2658        void print();
2659    };
2660
2661    struct Term{                      //Values separated by + and -
2662        vector<Factor> term_;
2663        void print();
2664    };
2665
2666    struct Component{                 //The if-else operator _?_:_
2667        vector<Term> component_;
2668        void print();
2669    };
2670
2671    struct LExpr{                     //Comma separated values
2672        vector<Component> lexpr_;
2673        void print();
2674    };
2675
2676
2677    class Logical_notParser{
2678        public:
2679            Logical_notParser(const vector<Token> &tok_stream, size_t begin){
2680                for(size_t i=begin; i<tok_stream.size(); i++){
2681                    tok_stream_.push_back(tok_stream[i]);
2682                }
2683            }
2684            Logical_notParser(const vector<Token> &tok_stream)
2685                :tok_stream_(tok_stream){}
2686            Logical_notParser(){}
2687
2688            void construct(const vector<Token> &tok_stream, size_t begin){
2689                for(size_t i=begin; i<tok_stream.size(); i++){
2690                    tok_stream_.push_back(tok_stream[i]);
2691                }
2692            }
2693
2694            pair<Logical_not, size_t> getLogical_not();
2695
2696        private:
2697            vector<Token> tok_stream_;
2698
2699            size_t parse_(size_t begin);
```

```
2700
2701    };
2702
2703    class Logical_xorParser{
2704        public:
2705            Logical_xorParser(const vector<Logical_not> &tok_stream, size_t begin){
2706                for(size_t i=begin; i<tok_stream.size(); i++){
2707                    tok_stream_.push_back(tok_stream[i]);
2708                }
2709            }
2710            Logical_xorParser(const vector<Logical_not> &tok_stream)
2711                :tok_stream_(tok_stream){}
2712            Logical_xorParser(){}
2713
2714            void construct(const vector<Logical_not> &tok_stream, size_t begin){
2715                for(size_t i=begin; i<tok_stream.size(); i++){
2716                    tok_stream_.push_back(tok_stream[i]);
2717                }
2718            }
2719
2720            pair<Logical_xor, size_t> getLogical_xor();
2721
2722        private:
2723            vector<Logical_not> tok_stream_;
2724
2725            size_t parse_(size_t begin);
2726
2727    };
2728
2729    class LExprParser{
2730        public:
2731            LExprParser(const vector<Token> &tok_stream, size_t begin){
2732                for(size_t i=begin; i<tok_stream.size(); i++){
2733                    tok_stream_.push_back(tok_stream[i]);
2734                }
2735            }
2736            LExprParser(const vector<Token> &tok_stream):tok_stream_(tok_stream){}
2737            LExprParser(){}
2738
2739            void construct(const vector<Token> &tok_stream, size_t begin){
2740                for(size_t i=begin; i<tok_stream.size(); i++){
2741                    tok_stream_.push_back(tok_stream[i]);
2742                }
2743            }
2744
```

```
2745        pair<LExpr *,size_t> getLExpr();
2746
2747    private:
2748        const char *expr_;
2749        vector<Token> tok_stream_;
2750        Token tok_;
2751        LExpr *lexpr_;
2752        bit32 curl_match_;
2753        bool comma_delimited_;
2754
2755        size_t parse_(size_t begin, LExpr *lexpr);
2756    };
2757
2758    //-----------------------------------------------------------------------------
2759
2760    struct SExpr{
2761        vector<atom> sexpr_;
2762
2763        void print(){
2764            for(size_t i=0; i<sexpr_.size(); i++){
2765                if(sexpr_[i].type_ == TOKEN){
2766                    Token *t = (Token *)sexpr_[i].atom_;
2767                    t->print();
2768                    cout<<" ";
2769                }
2770                else if(sexpr_[i].type_ == SEXPR){
2771                    indentLevel++;
2772                    cout<<"\n";
2773                    for(int j=0;j<indentLevel;j++)cout<<"\t";cout<<"( ";
2774                    SExpr *t = (SExpr *)sexpr_[i].atom_;
2775                    t->print();
2776                    indentLevel--;
2777                    cout<<" )"<<endl;
2778                    for(int j=0;j<indentLevel;j++)cout<<"\t";
2779                }
2780                else if(sexpr_[i].type_ == LEXPR){
2781                    LExpr *l = (LExpr *)sexpr_[i].atom_;
2782                    l->print();
2783                }
2784                else;
2785            }
2786        }
2787    };
2788
2789    class SExprParser{
```

```
2790    public:
2791        SExprParser(const char *expr):expr_(expr),lex_(expr){
2792            while(lex_.next(tok_)) tok_stream_.push_back(tok_);
2793        }
2794        SExprParser(const vector<Token> &tok_stream, size_t begin){
2795            for(size_t i=begin; i<tok_stream.size(); i++){
2796                tok_stream_.push_back(tok_stream[i]);
2797            }
2798        }
2799        SExprParser(const vector<Token> &tok_stream):tok_stream_(tok_stream){}
2800        SExprParser(){}
2801
2802        void construct(const vector<Token> &tok_stream, size_t begin){
2803            for(size_t i=begin; i<tok_stream.size(); i++){
2804                tok_stream_.push_back(tok_stream[i]);
2805            }
2806        }
2807
2808        pair<SExpr *,size_t> getSExpr(){
2809            pair<SExpr *, size_t> ret;
2810            paren_match = 0;
2811            ret.first = new SExpr;
2812            ret.second = parse_(0,ret.first);
2813            if(paren_match){
2814                I_error e("Parentheses␣mismatch");
2815                e.print();
2816            }
2817            return ret;
2818        }
2819
2820    private:
2821        const char *expr_;
2822        vector<Token> tok_stream_;
2823        Token tok_;
2824        SExpr *sexpr_;
2825        Tokenizer lex_;
2826        bit32 paren_match;
2827
2828        size_t parse_(size_t begin, SExpr *sexpr){
2829            size_t i;
2830            atom myAtom;
2831            SExpr *newSexpr;
2832            LExprParser lParser;
2833            pair<LExpr *,size_t> lres;
2834            for(i=begin; i<tok_stream_.size(); i++){
```

```
2835                    if(tok_stream_[i].type_ == OPERATOR){
2836                        switch(tok_stream_[i].token_.op){
2837                            case LPAREN:
2838                                paren_match++;
2839                                newSexpr = new SExpr;
2840                                i = parse_(i+1, newSexpr);
2841                                //atom myAtom;
2842                                myAtom.type_=SEXPR;
2843                                myAtom.atom_ = (void *)newSexpr;
2844                                sexpr->sexpr_.push_back(myAtom);
2845                                break;
2846                            case RPAREN:
2847                                paren_match--;
2848                                return i;
2849                            case LCURL:
2850                                lParser.construct(tok_stream_, i+1);
2851                                //i += lParser.parse_(0,);
2852                                lres = lParser.getLExpr();
2853                                i += lres.second;
2854                                //atom myAtom;
2855                                myAtom.type_ = LEXPR;
2856                                myAtom.atom_ = (void *)lres.first;
2857                                sexpr->sexpr_.push_back(myAtom);
2858                                break;
2859                            default:
2860                                //atom myAtom;
2861                                myAtom.type_=TOKEN;
2862                                Token *myToken = new Token(tok_stream_[i]);
2863                                myAtom.atom_ = (void *)myToken;
2864                                sexpr->sexpr_.push_back(myAtom);
2865                                break;
2866                        }
2867                    }
2868                    else{
2869                        //atom myAtom;
2870                        myAtom.type_ = TOKEN;
2871                        Token *myToken = new Token(tok_stream_[i]);
2872                        myAtom.atom_ = (void *)myToken;
2873                        sexpr->sexpr_.push_back(myAtom);
2874                    }
2875                }
2876                return i;
2877            }
2878    };
2879
```

```
2880
2881 //-------------------------------------------------------------------------
2882 //       FUNCTION DEFINITIONS FOR THE L-EXPRESSION STRUCT AND ITS PARSER
2883 //-------------------------------------------------------------------------
2884
2885 size_t LExprParser::parse_(size_t begin, LExpr* lexpr){
2886     size_t i;
2887     atom myAtom;
2888     pair<SExpr *,size_t> sres;
2889     SExprParser sParser;
2890     for(i=begin; i<tok_stream_.size(); i++){
2891         if(tok_stream_[i].type_ == OPERATOR){
2892             switch(tok_stream_[i].token_.op){
2893                 case LPAREN:
2894                     sParser.construct(tok_stream_, i+1);
2895                     sres = sParser.getSExpr();
2896                     i += sres.second;
2897                     //atom myAtom;
2898                     myAtom.type_ = SEXPR;
2899                     myAtom.atom_ = (void *)sres.first;
2900                     //lexpr->lexpr_.push_back(myAtom);
2901                     break;
2902
2903                 case COMMA:
2904                     comma_delimited_ = true;
2905                     break;
2906             }
2907         }
2908     }
2909 }
2910
2911
2912
2913
2914 inline void LExpr::print(){
2915     /*cout<<"\n";
2916     indentLevel++;
2917     for(int i=0;i<indentLevel;i++) cout<<"\t";cout<<"{ ";
2918     for(size_t i=0; i<lexpr_.size(); i++){
2919         if(lexpr_[i].type_ == TOKEN){
2920             Token *t=(Token *) lexpr_[i].atom_;
2921             t->print();
2922             cout<<" ";
2923         }
2924         else if(lexpr_[i].type_ == SEXPR){
```

```
2925                indentLevel++;
2926                cout<<endl;
2927                for(int j=0; j<indentLevel; j++) cout<<"\t";cout<<"( ";
2928                SExpr *s=(SExpr *)lexpr_[i].atom_;
2929                s->print();
2930                indentLevel--;
2931                cout<<" )"<<endl;
2932                for(int j=0; j<indentLevel; j++) cout<<"\t";
2933            }
2934            else if(lexpr_[i].type_ == LEXPR){
2935                LExpr *l=(LExpr *)lexpr_[i].atom_;
2936                l->print();
2937            }
2938            else;
2939
2940            if(!arithmatic_){
2941                cout<<" , ";
2942            }
2943        }
2944        indentLevel--;
2945        cout<<" } "<<endl;
2946        for(int j=0; j<indentLevel; j++) cout<<"\t";*/
2947    }
2948
2949    pair<LExpr *,size_t> LExprParser::getLExpr(){}
2950
2951
2952    //-------------------------------------------------------------------------------------------
2953
2954    struct Statement{
2955        bool store_;
2956        SExpr *lval_;
2957        SExpr *rval_;
2958
2959        Statement():store_(true),lval_(NULL),rval_(NULL){}
2960        void print(){
2961            cout<<endl<<" L - V A L U E"<<endl<<"================"<<endl;
2962            lval_->print();
2963            if(store_){
2964                cout<<endl<<" R - V A L U E"<<endl<<"================"<<endl;
2965                rval_->print();
2966            }
2967        }
2968
2969        ~Statement(){
```

```
2970        delete lval_;
2971        delete rval_;
2972    }
2973};
2974
2975class StatementParser{
2976    public:
2977        StatementParser(const char *expr):expr_(expr){
2978            Tokenizer lex_(expr);
2979            Token t;
2980            vector<Token> tok_stream_l;
2981            vector<Token> tok_stream_r;
2982
2983            stmt_ = new Statement;
2984
2985            while(lex_.next(t)){
2986                if(t.type_ == OPERATOR){
2987                    if(t.token_.op == ASSIGN){
2988                        break;
2989                    }
2990                    else tok_stream_l.push_back(t);
2991                }
2992                else tok_stream_l.push_back(t);
2993            }
2994            SExprParser lval_parser_(tok_stream_l);
2995            stmt_->lval_ = lval_parser_.getSExpr().first;
2996            int count = 0;
2997            while(lex_.next(t)){
2998                count++;
2999                tok_stream_r.push_back(t);
3000            }
3001            if(!count)stmt_->store_=false;
3002            else{
3003                SExprParser rval_parser_(tok_stream_r);
3004                stmt_->rval_ = rval_parser_.getSExpr().first;
3005            }
3006        }
3007
3008        Statement *getStmt(){
3009            return stmt_;
3010        }
3011
3012
3013    private:
3014        const char *expr_;
```

```
3015        vector <Token> tok_stream_;
3016        Statement *stmt_;
3017    };
3018
3019
3020
3021    class SymTbl{
3022        public:
3023            typedef SymTbl type;
3024            typedef string key_type;
3025            typedef bit64 value_type;
3026            typedef bit64 count_type;
3027
3028            SymTbl():offset(0),sym_count(0){}
3029            bool insert(const key_type key){
3030                if(table.count(key)) return false;
3031                table[key]=++sym_count;
3032                return true;
3033            }
3034
3035            value_type operator [](const key_type &key){
3036                return offset+table[key];
3037            }
3038
3039            const bool exists(const key_type &key) const{
3040                if(table.count(key)) return true;
3041                return false;
3042            }
3043
3044            const bool empty() const{
3045                return sym_count?false:true;
3046            }
3047
3048        private:
3049            count_type offset;
3050            count_type sym_count;
3051            map<key_type, value_type> table;
3052    };
3053
3054
3055    class machine{
3056        public:
3057            typedef bit64 word_type;
3058            typedef bit8 byte_type;
3059    //-------------------------------------------------------------------------
```

```
3060    //      TO WORK FROM HERE. IMPLEMENT THE VIRTUAL MACHINE FOR INTERPRETATION
3061    //      ALSO IMPLEMENT AN INTERMEDIATE REPRESENTATION IN BINARY IF POSSIBLE
3062    //-------------------------------------------------------------------------
3063            machine():current_state(0){}
3064            bool storeStatement(Statement *s){}
3065            bool eval(SExpr *expr){}
3066
3067
3068        private:
3069            word_type current_state;
3070            SymTbl sym_table;
3071            map<pair<word_type,word_type>,word_type> transition_table;
3072            set<word_type> final_states;
3073            vector<vector<SExpr *> > eval_rules;
3074            map<word_type, size_t> binder_table;
3075    };
3076
3077
3078
3079    int main(){
3080        line_no = 1;
3081        col_no = 0;
3082        cout<<"This is a (testing sentence (a+b) > 7.5 ) = (5! is 120 ) and"
3083            <<" I am \"Ganesh Prasad 1993\""<<endl;
3084        StatementParser parse("This is a (testing sentence (a+b) > 7.5 )"
3085                              " = (5! is 120) and I am \"Ganesh Prasad 1993\"");
3086        Statement *expr = parse.getStmt();
3087        expr->print();
3088        cout<<endl;
3089        return 0;
3090    }
3091
3092    #endif
```