Sample Yacc file
PL/306/2 language
Vladimir Vacic, Fran Jarnjak

```
%{
#include <stdio.h>
#include "lex.yy.c"
extern int lineNumber;
FILE *fp;
%}

%token  K_START
%token  K_END
%token  K_STOP
%token  K_DECLARE
%token  K_INTEGER
%token  K_FLOAT
%token  K_PROCEDURE
%token  K_GET
%token  K_GOTO
%token  K_PUT
%token  K_SKIP
%token  K_IF
%token  K_THEN
%token  K_ELSE
%token  K_ENDIF

%token  T_IDENT
%token  T_INTEGER
%token  T_FLOAT

%token  E_IDENT_LONG
%token  E_INV_CHAR

%token  O_ASSIGN
%token  ',' ';' ':'
%token  '(' ')'

%left '!'
%left   '*' '/' '%' '&'
%left   '-' '+' '|'
%left   '=' '>' '<'
%left   O_NEG

%start Program

%%

Program:    DeclareStatementList ProcedureStatementList K_START ';'
StatementList K_END ';' { fprintf(fp,"Parsing sucesfull.\n"); printf("Parsing
succesfull.\n"); }
            | E_IDENT_LONG    { err_prog(E_IDENT_LONG); }
            | E_INV_CHAR      { err_prog(E_INV_CHAR); }
;
```

```
DeclareStatementList:   DeclareStatement DeclareStatementList { ; }
                  | DeclareStatement { ; }
;
DeclareStatement:       K_DECLARE '(' List ')' K_INTEGER ';' {
fprintf(fp,"DECLARE integer\n"); }
                  | K_DECLARE '(' List ')' K_FLOAT ';' { fprintf(fp,"DECLARE
float\n"); }
;
Id:   T_IDENT           {;}
      | E_IDENT_LONG  { err_prog(E_IDENT_LONG);}
;
List: Id ',' List
      | Id
;
ProcedureStatementList: ProcedureStatement ProcedureStatementList
                  | ProcedureStatement
;
ProcedureStatement: Id ':'K_PROCEDURE '(' List ')' ';' StatementList K_END Id
';' { fprintf(fp,"PROCEDURE\n");}
;
StatementList: Statement StatementList
            | Statement
;
Statement: AssignmentStatement { fprintf(fp,"ASSIGNMENT statement\n"); }
            | Label           { fprintf(fp,"LABEL statement\n");}
            | CallStatement { fprintf(fp,"CALL statement\n"); }
            | GetStatement { fprintf(fp,"GET statement\n"); }
            | PutStatement { fprintf(fp,"PUT statement\n"); }
            | ConditionalStatement { fprintf(fp,"CONDITIONAL statement\n"); }
            | GotoStatement { fprintf(fp,"GOTO statement\n"); }
            | K_STOP ';'{ fprintf(fp,"STOP\n"); }
            | ';'     { fprintf(fp,"EMPTY statement\n"); }
;

AssignmentStatement: List O_ASSIGN Expression ';' { fprintf(fp,"ASSIGN\n"); }
;

Expression:      T_INTEGER                    { ;}
             | T_FLOAT
             | Id                           { ;}
               | Expression '-' Expression                  { ; }
               | Expression '+' Expression                  { ; }
             | Expression '*' Expression             {; }
             | Expression '/' Expression             {; }
             | Expression '%' Expression             {; }
             | '-' Expression %prec O_NEG          { ; }
               | '(' Expression ')'                 { ; }
;

Label:      Id':' Statement { fprintf(fp,"LABEL\n"); }
;
CallStatement: Id '(' List ')' ';' { fprintf(fp,"CALL\n"); }
;
GetStatement: K_GET '(' List ')' ';' { fprintf(fp,"GET\n"); }
;
GotoStatement: K_GOTO Id ';'  { fprintf(fp,"GOTO\n");}
```

```
;
PutStatement:     K_PUT '(' ParameterList ')' ';'  { fprintf(fp,"PUT\n"); }
            | K_PUT K_SKIP '(' ParameterList ')' ';'  { fprintf(fp,"PUT
SKIP\n"); }
;
ParameterList: T_INTEGER ',' ParameterList
            | T_FLOAT ',' ParameterList
            | T_INTEGER
            | T_FLOAT
            | Id ',' ParameterList
            | Id
;
ConditionalStatement: K_IF Condition K_THEN Statement K_ENDIF ';'  {
fprintf(fp,"IF-THEN\n"); }
                  | K_IF Condition K_THEN Statement K_ELSE Statement K_ENDIF
';'  { fprintf(fp,"IF-THEN-ELSE\n"); }
;

Condition:  '!''('Condition')'
            | '(' Condition ')'
            | Condition2
            | '(' Condition ')' '&' '(' Condition2 ')'
            | '(' Condition ')' '|' '(' Condition2 ')'
;

Condition2:       Expression '<' Expression
            | Expression '>' Expression
            | Expression '=' Expression
;


%%

int err_prog(int err)
{
  printf("\n\nError: 'DeclarationBlock ProcedureBlock START; StatementList
END;'\n");
  printf("construct expected.");

  if (err == E_IDENT_LONG)
    err_ident_long();
  if (err == E_INV_CHAR)
    err_inv_char();

  return -1;
}

int err_ident_long(void)
{
  printf("\n\nError: Identifiers cannot exceed ");
  printf("sixteen characters in length - Line: %d\n\n", lineNumber);
  return -1;
}

int yywrap(void) {
      return 1;
}
```

```
int yyerror(char *msg) {
  printf("ERROR Parsing: %s - %s - Line: %d\n", yytext, msg, lineNumber);
  fprintf(fp,"\nERROR Parsing: %s - %s\n - Line: %d", yytext, msg,
lineNumber);
  return 0;
}

int err_inv_char(void)
{
  printf("\n\nError: Invalid character encountered - Line: %d\n",
lineNumber);
  return -1;
}

int main(int argc, char *argv[])
{

       printf("\nPL/306/2 Parser v1.0\n");
       printf("(C) 2000 Fran Jarnjak, Vladimir Vacic\n\n");

       if (argc == 2) {
                yyin = fopen(argv[1], "r");
              fp = fopen("pl306.out","a+");
       }
         else if (argc == 3) {
                yyin = fopen(argv[1], "r");
              fp = fopen(argv[2], "a+");
       }
         else {
               printf("Usage: pl306c <input_file> <ouput_file>\n");
             printf("Error code: %d\n\n",argc);
            return 1;
       }
         yyparse();
         return 0;
}
```