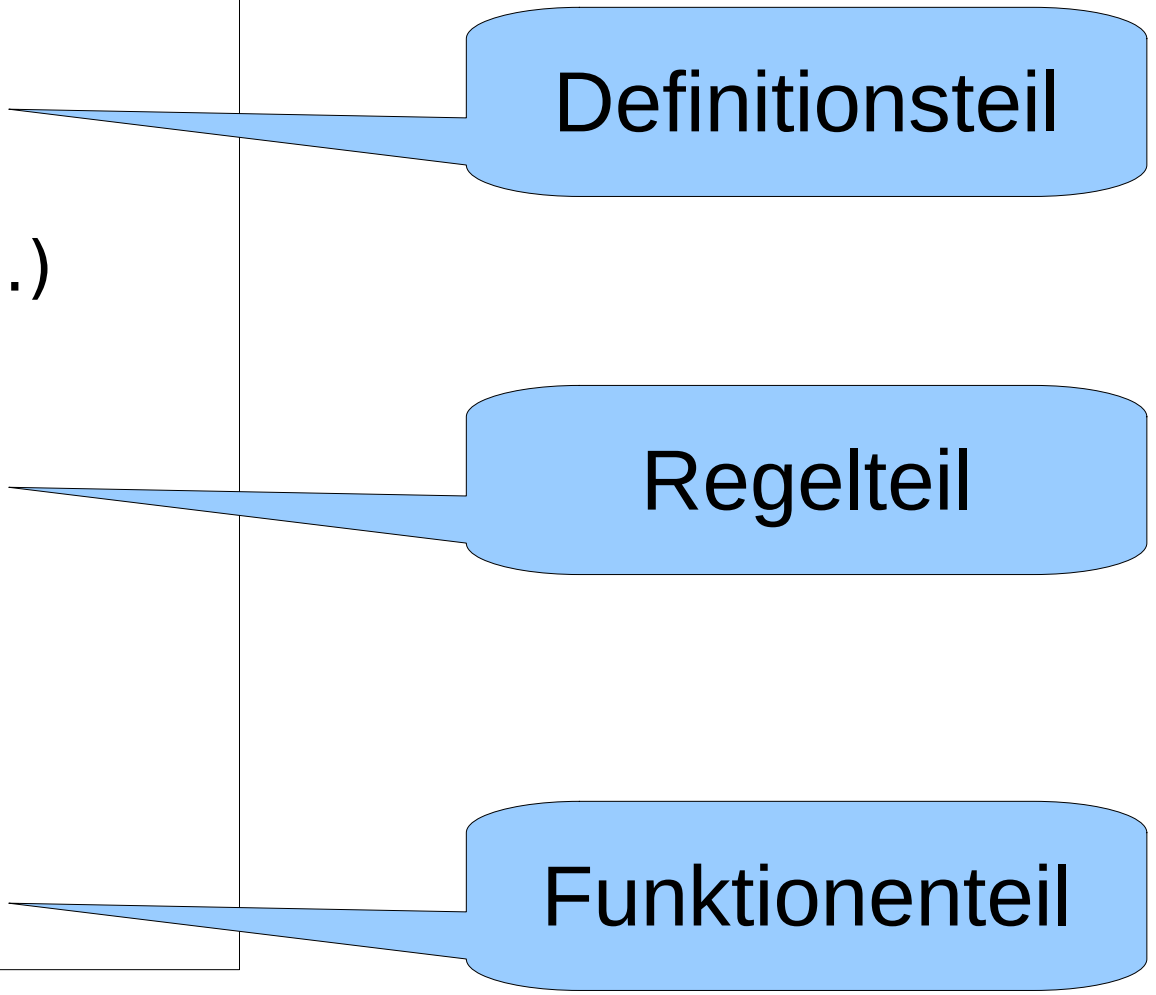
yacc/ bison

- Onlinedocumentation:
  - https://www.geeksforgeeks.org/introduction-to-yacc/
  - http://epaperpress.com/lexandyacc/
- Yet another compiler compiler
- Klassisches Unix Werkzeug, bison ist die GNU-Version von yacc
- Generiert LALR bottom up Parser (look ahead, left recursive Parser)

Aufbau einer yacc Datei

```
%{

 ...c- definitions ...
%}
...definitions (tokens ...)

%%

... rules ...

%%

... c-functions ...
```



Programmbuild

- yacc -d xyz.y
- lex xyz.lex
- gcc lex.yy.c y.tab.c
- Arbeitsschritte:
  - yacc-Datei aus Grammatik bauen
  - yacc -d xyz.y
  - Lex-Datei bauen, verwendet y.tab.h
  - lex xyz.lex
- compilieren

yacc -d erzeugt zusätzlich Headerfile für lex mit Tokenbeschreibungen

Zusammenspiel Lex - yacc

- Yacc ruft die Funktion **`int yylex()`** immer auf, wenn ein Token benötigt wird.
- Der Tokencode, bei Sonderzeichen oft das Zeichen selbst, wird als Returnwert zurückgegeben, viele Token sind damit bereits vollständig beschrieben.
- Ein darüber hinausgehender Tokenwert kann über yylval übergeben werden. Der Wert wird in der Aktion einer Regel nach yylval geschrieben.
- Der Defaulttyp von yylval ist int, sollen Werte verschiedener Typen übergeben werden, so erfolgt dies über einen speziellen union Typ.

Definitionsteil

- C-Code, der in das zu generierende c-File übernommen wird, eingeschlossen in %{ .. %}
  - Includes, defines, prototypes, Variablendefinitionen
- Yacc Deklarationen
  - Tokendefinitionen (%token numeral) daraus wird dann das Includefile y.tab.h generiert bei Aufruf von yacc -d xyz.y (für Schlüsselworte und zusammengesetzte Sonderzeichen, wie :=)
  - Präzedenzdeklarationen
  - Spezielle union Definition, aus der YYSTYPE generiert wird(für numaerals, identifier).

Regelteil

- Regeln in BNF ähnlicher Notation beginnen **linksbündig** mit einem Metasymbol
- Alternativen beginnen üblicher Weise auf neuer Zeile mit '|', nicht unbedingt linksbündig
- Nach der Regel kann eine Aktion als Block angegeben werden, dieser kann über mehrere Zeilen gehen. Die C-Code-Zeilen dürfen **nicht linksbündig** beginnen!
- Werte der einzelnen Zeichen der rechten Seite der Regel stehen in den Aktionen unter $1, $2 … entsprechend Ihrer Position zur Verfügung. Ein Returnwert kann durch $$ gebildet werden.
- Jede Regel wird mit ';' abgeschlossen

Beispiel 1

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}
%token INTEGER
%left '+' '-'
%%
program:program expr '\n'          { printf("%d\n", $2); }
        |
        ;
expr:   INTEGER
        | expr '+' expr            { $$ = $1 + $3; }
        | expr '-' expr            { $$ = $1 - $3; }
        ;
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}
int main(void) {
    yyparse();
    return 0;
}
```

y.tab.h, generiert durch yacc -d xyz.y

leicht gekürzt

```
/* Tokens.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE

   enum yytokentype {
     INTEGER = 258
   };
#endif
/* Tokens.  */
#define INTEGER 258

#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef int YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn
*/
# define YYSTYPE_IS_DECLARED 1
#endif

extern YYSTYPE yylval;
```

Tokencode

Typ des Tokenwertes, hier int
In anderen Fällen ein union

.lex Datei dazu

```
    /* calculator #1 */
%{
    #Ainclude "y.tab.h"
    #include <stdlib.h>
    void yyerror(char *);
%}

%%

[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

[-+*/()\n]  { return *yytext; }

[ \t]       ;       /* skip whitespace */

.           yyerror("Unknown character");

%%

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

Tokenwert

Tokencode

'+', '-' oder '\n' wird als Token zurückgegeben

Beispiel 2 expr, term, factor

```
%{
#include <stdio.h> /* Printf */
#include <stdlib.h>/* exit   */
#define YYSTYPE int
int yyparse(void);
int yylex(void);
void yyerror(char *mes);
%}
%token number
%token QUIT 254
%%
   . . . → next page
%%
int main()
{
 printf("Enter expression with + - * / ( ) \n");
 yyparse(); return 0;
}
void yyerror(char *mes) {fprintf(stderr,"%s\n", mes);}
```

Regelteil zu expr, term, factor

```
program : program expr '\n' {printf("= %d\n", $2);}
        |
        ;

expr    : expr '+' term {puts("expr 1");$$ = $1 + $3;}
        | expr '-' term {puts("expr 2");$$ = $1 - $3;}
        | term          {puts("expr 3");$$ = $1;}
        | QUIT {exit(0);}
        ;

term    : term '*' fact {puts("term 1");$$ = $1 * $3;}
        | term '/' fact {puts("term 2");$$ = $1 / $3;}
        | fact          {puts("term 3");$$ = $1;}
        ;

fact    : number        {puts("fact 1");$$ = $1;}
        | '-' number    {puts("fact 1");$$ = -$2;}
        | '(' expr ')'  {puts("fact 2");$$ = $2;}
        ;
```

Lexer dazu

```
%{
    #include "t2.h"        /* eigentlich y.tab.h !!  */
    #include <stdlib.h>    /* yacc -d -o t2.c erzeugt*/
    void yyerror(char *); /* auch t2.h               */
%}

%%
[0-9]+       { yylval = atoi(yytext);
               return number;
             }
[-+()=/*\n] { return *yytext; }
[ \t]   ;    /* skip whitespace */
.            yyerror("Unknown character");

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

Präzedenzregeln

- Präzendenz: Vorrang, Priorität
- Präzedenzregeln von yacc regeln die Priotrität und die Assoziativität von Operatoren
- Die Präzedenzregeln werden im Definitionsteil der yacc Datei angegeben und gelten für jedes Vorkommen des betreffenden Tokens
- Präzedenz steigt zeilenweise von oben nach unten (→ Beispiel)

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
```

* und / hat höhere Präzedenz als + und -

Keine Regel ohne Ausnahme?

- Für das unäre '-' klappt das nicht! Das negative Vorzeichen hat eine höhere Priorität, als die multiplikativen Operatoren.
- Vergabe einer neuen Präzedenz für das Token '-' in der betreffenden Regel.

```
expr    :         expr  '+'  expr
        |         expr  '-'  expr
        |         expr  '*'  expr
        |         expr  '/'  expr
        |  '-'    expr %prec  '*'
```

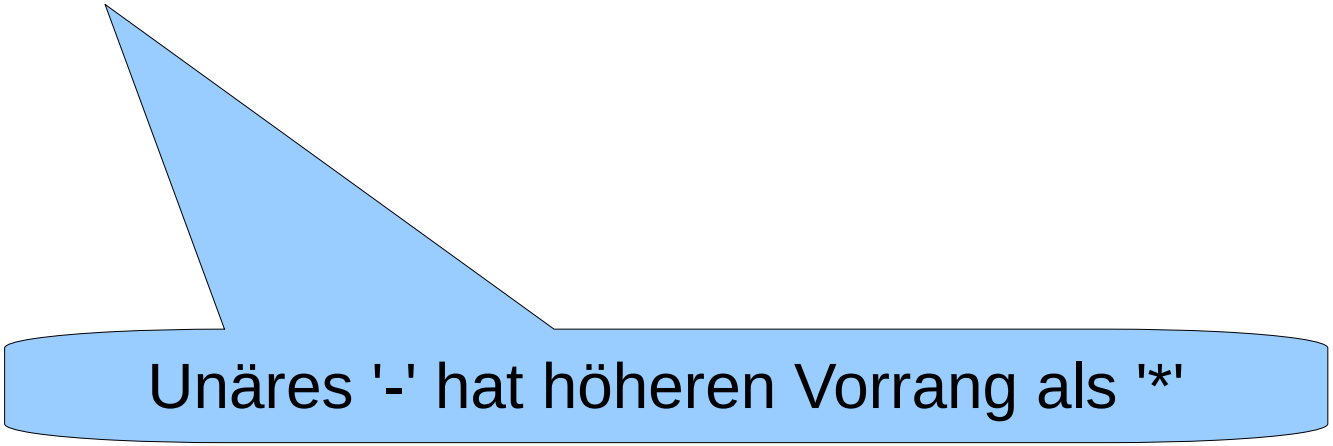
Beispiel 3 Expressions mit Päzedenzregeln

```
%{
#include <stdio.h>
%}

%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%{

void yyerror(char *);
int yylex(void);
int sym[26];
%}
%%
. . . → next Page

%%
void yyerror(char *s) {fprintf(stderr, "%s\n", s);}
int main(void)
{
 yyparse();  return 0;
}
```

Präzedenzregeln

Array für 26 Variable

```
Program:    program statement '\n'
          |
          ;
Statement: expr { printf("%d\n", $1); }
          | VARIABLE '=' expr { sym[$1] = $3; }
          ;
Expr: INTEGER
    | VARIABLE                  { $$ = sym[$1]; }
    | expr '+' expr             { $$ = $1 + $3; }
    | expr '-' expr             { $$ = $1 - $3; }
    | expr '*' expr             { $$ = $1 * $3; }
    | expr '/' expr             { $$ = $1 / $3; }
    | '(' expr ')'              { $$ = $2; }
    | '-'  expr  %prec '*'      { $$ =-$2; }
    ;
```



.lex dazu

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}
%%
 /* variables */
[a-z]   { yylval = *yytext - 'a'; return VARIABLE; }

 /* integers */
[0-9]+ { yylval = atoi(yytext); return INTEGER; }
 /* operators */
[-+()=/*\n] { return *yytext; }
 /* skip whitespace */
[  \t] ;
 /* anything else is an error */
. yyerror("invalid character");
%%
int yywrap(void) {return 1;}
```

Berechnen des Variablenindex aus dem Buchstaben

yacc und lex und pl/0

- Umformung der Grammatik in einfache BNF
- Erstellen der yacc Datei
- Erstellen des Headerfiles
- Erstellen der lex-Datei
- Alles zusammenbauen → Parser fertig
- Semantikroutinen anpassen (Parameterübergaben)
- Alles zusammenbauen → Compiler fertig

pl0.y - Der Vereinbarungsteil

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "list.h"
#include "code.h"
#include "semrtype.h"
#include "semr.h"
%}
%union{
  long num;
  char*idnt;
  char*strg;
}
. . . → siehe rechts

%%
```

```
%token T_BEGIN     276
%token T_CALL      257
%token T_CONST     258
%token T_DO        259
%token T_ELSE      260
%token T_END       261
%token T_IF        262
%token T_ODD       263
%token T_PROCEDURE 264
%token T_THEN      265
%token T_VAR       266
%token T_WHILE     267
%token <idnt>T_Ident     268
%token <num>T_Num  269
%token T_ERG       270
%token T_LEQ       271
%token T_GEQ       272
%token <strg>T_String     273
```

pl0.y – der Funktionenteil

```
%%
main(int argc, char**argv)
{
 if (argc>1)
  if (initLex(argv[1]))
  {
    initMain();
    openOFile(argv[1]);
    yyparse();
    closeOFile();
  }
  else printf("Dateifehler\n");
  return 0;
}

yyerror(char* ps) {  printf("%s\n",ps); }
```

Öffnen der Quelldatei
und Initialisierung des Lexers

Anlegen aller erforderlichen
Datenstrukturen

EBNF

```
programm    = block '.'
block       = [“CONST“ ident=num{“,“ ident=num}“;“]
              ['VAR' ident { ',' ident } ';' ]
              {'PROCEDURE' ident ';' block ';' } statement

statement   = [ident ':=' expression |
              'CALL' ident |
              '?' ident |
              '!' expression |
              'BEGIN' statement { ';' statement } 'END' |
              'IF' condition THEN statement |
              'WHILE' condition DO statement]

condition   = 'ODD' expression |
              expression ( '=' | '#' | '<' | '<=' | '>' | '>=' ) expression

expression  =['+' | '-'] term {('+' | '-') term }

term        =faktor { ('*'|'/') faktor }

factor      =ident | number | '(' expression ')'
```

Die Regeln

```
Program   : block '.'                              {AdestroyPl0();};

Block     : blockDecl statement;

BlockDecl: constDecl varDecl procDeclList   {AenterProc();};

ConstDecl: T_CONST constList ';'

         | ;

ConstList: constList ',' T_Ident '=' T_Num {AcreateConst($5,$3);}

         | T_Ident '=' T_Num                {AcreateConst($3,$1);};

varDecl:   T_VAR varList ';'

         | ;

varList:   varList ',' T_Ident               {AcreateVar($3);}

         | T_Ident                           {AcreateVar($1);};
```

```
ProcDeclList: procDeclList procDecl
            | ;


ProcDecl: T_PROCEDURE T_Ident ';'          {AcreateProc($2);}
          Block ';'                        {AdestroyProc();};


StatementList: statementList ';' statement
             |  statement
             |  ;


Statement:    T_BEGIN statementList T_END
            | T_Ident T_ERG {AassLeft($1);}
              expression                          {Aass();}
            | T_IF condition T_THEN                {Aif1();}
              Statement                           {Aif2();}
            | T_WHILE                             {Awhile1();}
              condition T_DO                      {Awhile2();}
              Statement                           {Awhile();}
            | T_CALL T_Ident                      {Acall($2);}
            | '?' T_Ident                         {Ain($2);}
            | '!' expression                      {AstOutExpr();}
            | '!' T_String                        {AstOutStrng($2);};
```

```
expression: term1 '+' exprList      {Aoperator(0);}
          | term1 '-' exprList      {Aoperator(1);}
          | term1;
term1:      '-' term                {AvzMinus();}
          | '+' term
          | term;
ExprList:   exprList '+' term       {Aoperator(0);}
          | exprList '-' term       {Aoperator(1);}
          | term;



Term:       term '*' factor         {Aoperator(2);}
          | term '/' factor         {Aoperator(3);}
          | factor;

Factor:     T_Num                   {AfacNum($1);}
          | T_Ident                 {AfacId ($1);}
          | '(' expression ')';

CmpOP:      '='                     {AcondPush(cmpEQ);}
          | '#'                     {AcondPush(cmpNE);}
          | '<'                     {AcondPush(cmpLT);}
          | '>'                     {AcondPush(cmpGT);}
          | T_LEQ                   {AcondPush(cmpLE);}
          | T_GEQ                   {AcondPush(cmpGE);};
```

```
Condition: T_ODD expression                  {AcondOdd();}
         | expression cmpOP expression       {Acond();};
```

```
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   enum yytokentype {
     T_BEGIN = 276,
     T_CALL = 257,
     T_CONST = 258,
     T_DO = 259,
     T_ELSE = 260,
     T_END = 261,
     T_IF = 262,
     T_ODD = 263,
     T_PROCEDURE = 264,
     T_THEN = 265,
     T_VAR = 266,
     T_WHILE = 267,
     T_Ident = 268,
     T_Num = 269,
     T_ERG = 270,
     T_LEQ = 271,
     T_GEQ = 272,
     T_String = 273
   };
#endif
```

y.tab.h

```
/* Tokens.  */
#define T_BEGIN 276
#define T_CALL 257
#define T_CONST 258
#define T_DO 259
#define T_ELSE 260
#define T_END 261
#define T_IF 262
#define T_ODD 263
#define T_PROCEDURE 264
#define T_THEN 265
#define T_VAR 266
#define T_WHILE 267
#define T_Ident 268
#define T_Num 269
#define T_ERG 270
#define T_LEQ 271
#define T_GEQ 272
#define T_String 273
```

```
#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE
{

/* Line 1676 of yacc.c  */
#line 9 "p10.y"

  long num;
  char*idnt;
  char*strg;

/* Line 1676 of yacc.c  */
#line 100 "y.tab.h"
} YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn
*/
# define YYSTYPE_IS_DECLARED 1
#endif

extern YYSTYPE yylval;
```

Der Lexer

```
%{
/*
lexikalische Analyse mit lex fuer PL/0
*/
#include "y.tab.h"
#include "list.h"
#include "debug.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

FILE * pIF;
int Line;
%}
%%
```

Funktionenteil

```
%%

int Lex()
{
  return yylex();;
}

int yywrap(){ return 1;}

int initLex(char* fname)
{
  char vName[128+1];

  strcpy(vName,fname);
  if (!strstr(vName,".pl0")) strcat(vName,".pl0");

  pIF=fopen(vName,"rt");
  if (pIF) {yyin=pIF; return OK;}
  return FAIL;
}
```

Regelteil

```
%%
 /* rules division */

 /*************************************************/
 /* return beendet die lexikalische Analyse nach dem*/
 /* Erkennen eines Morphems.                      */
 /*************************************************/

 /************************/
 /* Leer- und Trennzeichen */
 /************************/
[ \t]+

 /******************/
 /* Zeilenwechsel */
 /******************/
[\n]      {Line++;}
```

```
 /*********************************/
 /* Schluesselwoerter (Wortsymbole)   */
 /* werden wie Sonderzeichen behandlt */
 /*********************************/
"begin"   {return T_BEGIN;}
call      {return T_CALL;}
const     {return T_CONST;}
do        {return T_DO;}
else      {return T_ELSE;}
end       {return T_END;}
if        {return T_IF;}
odd       {return T_ODD;}
procedure {return T_PROCEDURE;}
then      {return T_THEN;}
"var"     {return T_VAR;}
while     {return T_WHILE;}
```

```
 /**********/
 /* Zahlen */
 /**********/
[0-9]+        {
                 yylval.num=atol(yytext);
                 return T_Num;
              }
 /***************/
 /* Bezeichner, */
 /***************/

 /* muessen hinter Schluesselwoertern aufgefuehrt werden */
[A-Za-z]([A-Za-z0-9])* {
                  ylval.idnt=malloc(strlen(yytext)+1);
                  strcpy(yylval.idnt,yytext);/*yytext;*/
                  printf("<%s>\n",yylval.idnt);
                  return T_Ident;
                  }
```

```
 /******************/
 /* Sonderzeichen */
 /******************/
("?")       {return '?';}
("!")       {return '!';}
("+")       {return '+';}
("-")       {return '-';}
("*")       {return '*';}
("/")       {return '/';}
("=")       {return '=';}
(">")       {return '>';}
("<")       {return '<';}
(":=")      {return T_ERG;}
("<=")      {return T_LEQ;}
(">=")      {return T_GEQ;}
(";")       {return ';';}
(".")       {return '.';}
(",")       {return ',';}
(\".*\")    {
          yylval.strg=yytext;
          return T_String;
        }
```

Die Funktionen

```
Factor: T_Num    {AfacNum($1);};
      | T_Ident  {AfacId ($1);};
      | '(' expression ')';
*************************************/

int AfacId (char* pName)
{
  tBez *pB;
  if ((pB=searchBezGlobal(pName))==NULL) Error(EBezNtFnd);
  if (((tVar*)pB->pObj)->Kz==KzVar)
  {
    if (pB->IdxProc==0)
       code(puValVrMain,((tVar*)pB->pObj)->Dspl); else
    if (pB->IdxProc==pCurrPr->IdxProc)
       code(puValVrLocl,((tVar*)pB->pObj)->Dspl); else
       code(puValVrGlob,((tVar*)pB->pObj)->Dspl,pB->IdxProc);
       return OK;
  }else
  if (((tConst*)pB->pObj)->Kz==KzConst)
  {
    code(puConst,((tConst*)pB->pObj)->Idx);
    return OK;
  }
  else Error(ENoNumVal);
  return 0;
}
```

```
int AfacNum(long Numb)
{
  tConst *pCnst;
  if ((pCnst=searchConst(Numb))!=NULL) ;
  else
  {
    pCnst=createConst(Numb);
    if (pCnst==NULL) return FAIL;
  }
  code(puConst,pCnst->Idx);
  return OK;
}
```