Exercise 1

Codify in *Yacc* the generator of the abstract trees of the language defined by the following grammar:

program → *decl-list*
decl-list → *decl-list decl* | *decl*
decl → *type init-list* **;**
type → **int** | **string** | **boolean**
init-list → *init* **,** *init-list* | *init*
init → **id =** *const*
const → **intconst** | **strconst** | **boolconst**

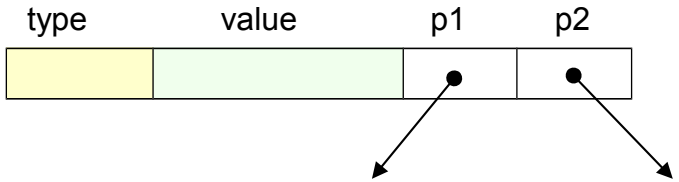
```
int x = 3, num = 100;
string A = "alpha", B = "beta";
boolean ok = true, end = false;
```

Exercise 1

Codify in *Yacc* the generator of the abstract trees of the language defined by the following grammar:

program → *decl-list*
decl-list → *decl-list decl* | *decl*
decl → *type init-list* **;**
type → **int** | **string** | **boolean**
init-list → *init* **,** *init-list* | *init*
init → **id =** *const*
const → **intconst** | **strconst** | **boolconst**

```
int x = 3, num = 100;
string A = "alpha", B = "beta";
boolean ok = true, end = false;
```




```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE *root = NULL;
Lexval lexval;
%}
%token INT STRING BOOLEAN INTCONST STRCONST BOOLCONST ID ERROR
%%
program     : decl-list {root = $$ = ntn(NPROGRAM); $$->p1 = $1;}
            ;
decl-list   : decl-list decl {$$ = ntn(NDECL_LIST); $$->p1 = $1; $$->p2 = $2;}
            | decl {$$ = ntn(NDECL_LIST); $$->p1 = $1}
            ;
decl        : type init_list ';' {$$ = ntn(NDECL); $$->p1 = $1; $$->p2 = $2;}
            ;
type        : INT {$$ =ntn(NTYPE); $$->p1 = keynode(T_INT);}
            | STRING {$$ =ntn(NTYPE); $$->p1 = keynode(T_STRING);}
            | BOOLEAN {$$ =ntn(NTYPE); $$->p1 = keynode(T_BOOLEAN);}
            ;
init_list   : init ',' init_list {$$ = ntn(NINIT_LIST); $$->p1 = $1; $$->p2 = $3;}
            | init {$$ = ntn(NINIT_LIST); $$->p1 = $1;}
            ;
init        : ID {$$ = idnode();}'=' const {$$ = ntn(NINIT); $$->p1 = $2; $$->p2 = $4;}
const       : INTCONST {$$ = ntn(NCONST): $$->p1 = intconstnode();}
            | STRCONST {$$ = ntn(NCONST): $$->p1 = strconstnode();}
            | BOOLCONST {$$ = ntn(NCONST): $$->p1 = boolconstnode();}
            ;
%%
main(){yyparse();}
```

Exercise 2

Using *Lex* and *Yacc*, codify the parser of the language defined by the following grammar:

program → *decl-list*
decl-list → *decl* **;** *decl-list* **|** *decl*
decl → *var-list* **:** *type*
var-list → **id ,** *var-list* **| id**
type → **integer** | **string** | **boolean**

```
a, b, c: integer;
x, y: string
```

Exercise 2

Using *Lex* and *Yacc*, codify the parser of the language defined by the following grammar:

program → *decl-list*
decl-list → *decl* **;** *decl-list* | *decl*
decl → *var-list* **:** *type*
var-list → **id** **,** *var-list* | **id**
type → **integer** | **string** | **boolean**

```
a, b, c: integer;
x, y: string
```

```
%{
#include "parser.tab.h"
#include <stdio.h>
char *lexval;
%}
delimiter    [ \t\n]
spacing      {delimiter}+
letter       [A-Za-z]
digit        [0-9]
id           {letter}({letter}|{digit})*
sugar        [:,;]

%%

{spacing}        ;
integer           {return(INTEGER);}
string            {return(STRING);}
boolean           {return(BOOLEAN);}
{id}              {lexval = newstring(yytext);
                   return(ID);}
{sugar}           {return(yytext[0]);}
.                 {return(ERROR);}

%%
char *newstring(char *s)
{
  char *p;

  p = malloc(sizeof(strlen(s)+1));
  strcpy(p, s);
  return(p);
}
```

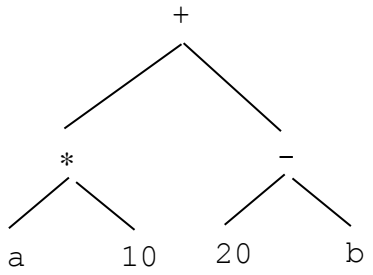
```
%{
#include <stdio.h>
%}
%token ID INTEGER STRING BOOLEAN ERROR
%%
program      :   decl_list
             ;

decl_list    :   decl ';' decl_list
             |   decl
             ;

decl         :   var_list ':' type
             ;

var_list     :   ID ',' var_list
             |   ID
             ;

type         :   INTEGER
             |   STRING
             |   BOOLEAN
             ;
%%

main()
{yyparse();}

yyerror()
{printf("Syntax error\n"); exit(1);}
```

Exercise 3

Using *Yacc* and *Lex*, codify a generator of the abstract syntax trees relevant to the following grammar:

program → *expr*
expr → *expr* **+** *term* | *expr* **–** *term* | *term*
term → *term* ***** *factor* | *term* **/** *factor* | *factor*
factor → **(** *expr* **)** | **id** | **num**

Here is an example of mapping: `a * 10 + (20 – b)` ⇨

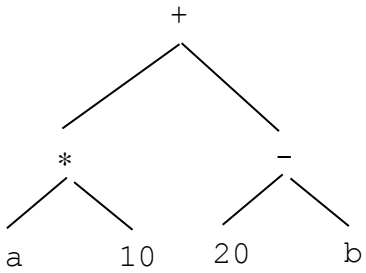


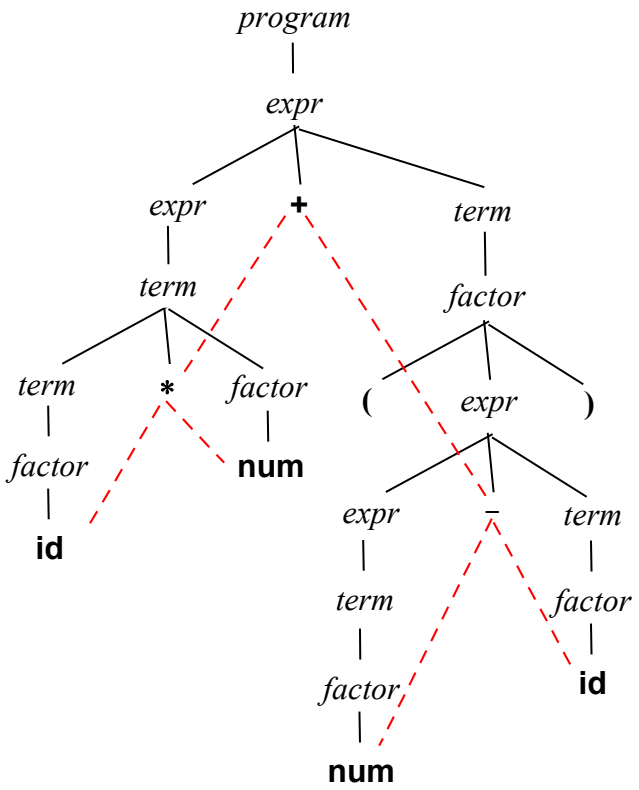
Exercise 3

Using *Yacc* and *Lex*, codify a generator of the abstract syntax trees relevant to the following grammar:

program → *expr*
expr → *expr* **+** *term* | *expr* **–** *term* | *term*
term → *term* ***** *factor* | *term* **/** *factor* | *factor*
factor → **(** *expr* **)** | **id** | **num**

Here is an example of mapping: `a * 10 + (20 – b)`



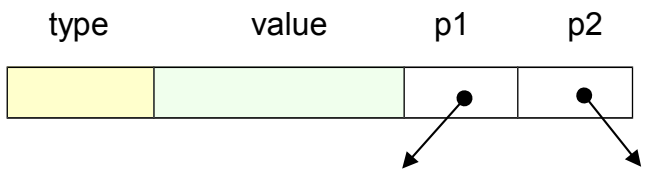


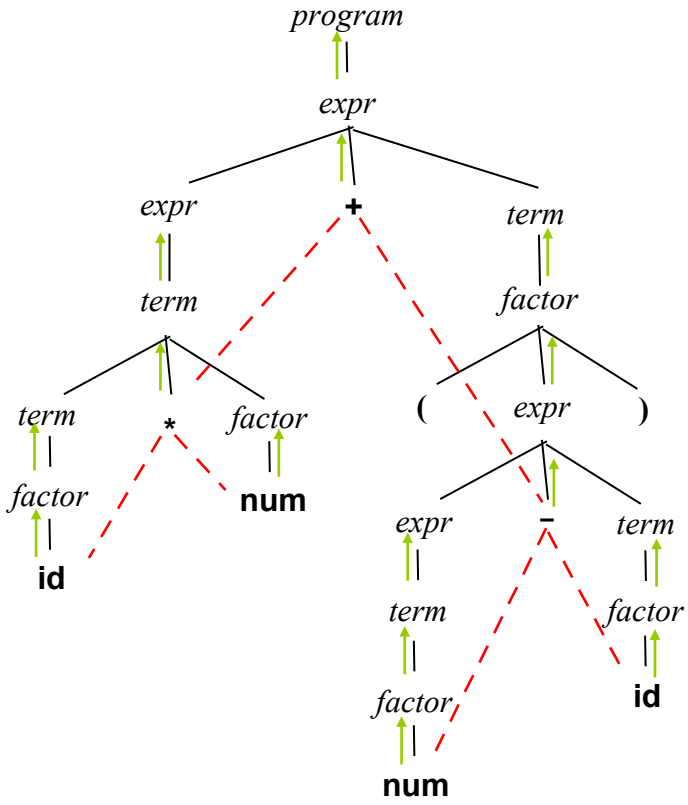
Exercise 3 (ii)

```
%{
#include "parser.h"
#include <stdlib.h>
Value lexval;
%}

delimiter   [ \t\n]
spacing     {delimiter}+
letter      [A-Za-z]
id          {letter}({letter}|{digit})*
digit       [0-9]
num         {digit}+
other       [+\-*/()]

%%

{spacing}    ;
{id}         {lexval.sval = newstring(yytext); return(ID);}
{num}        {lexval.ival = atoi(yytext); return(NUM);}
{other}      {return(yytext[0]);}
.            {return(ERROR);}

%%

char *newstring(char *s)
{
  char *p;

  p = malloc(strlen(s)+1);
  strcpy(p, s);
  return(p);
}
```

Exercise 3 (iii)

```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
extern Value lexval;
%}
%token ID NUM ERROR
%%
program  :   expr {root = $1;}
         ;
expr     :   expr '+' term {$$ = opnode(PLUS);
                            $$->p1 = $1;
                            $$->p2 = $3;}
         |   expr '-' term {$$ = opnode(MINUS);
                            $$->p1 = $1;
                            $$->p2 = $3;}
         |   term
         ;
term     :   term '*' factor {$$ = opnode(TIMES);
                              $$->p1 = $1;
                              $$->p2 = $3;}
         |   term '/' factor {$$ = opnode(SLASH);
                              $$->p1 = $1;
                              $$->p2 = $3;}
         |   factor
         ;
factor   :   '(' expr ')' {$$ = $2;}
         |   ID {$$ = idnode();}
         |   NUM {$$ = numnode();}
         ;
%%
...
main(){yyparse();}
```





Exercise 4

Codify in *Yacc* the generator of the concrete syntax trees relevant to the sentences of the language defined by the following grammar:

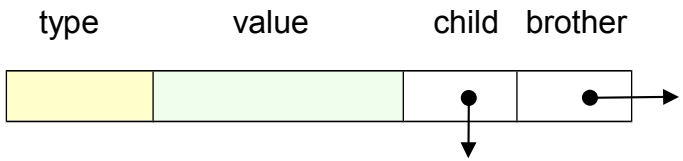
program → *decl-list*
decl-list → *decl* **;** *decl-list* | *decl* **;**
decl → *type var-list*
type → **int** | **string**
var-list → **id** **,** *var-list* | **id**

(example of sentence)

```
int a, b, c;
string x, y;
int z;
```

Exercise 4

```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE *root = NULL;
char *lexval;
%}
%token INT STRING ID ERROR
%%
program    : decl_list {root = $$ = ntn(NPROGRAM);
                        $$->child = $1;}
           ;
decl_list  : decl ';' decl-list {$$ = ntn(NDECL_LIST);
                                 $$->child = $1;
                                 $1->brother = $2 = tn(NSEMICOLON);
                                 $2->brother = $3;}
           | decl ';' {$$ = ntn(NDECL_LIST);
                       $$->child = $1;
                       $1->brother = tn(NSEMICOLON);}
           ;
decl       : type var_list {$$ = ntn(NDECL);
                            $$->child = $1;
                            $1->brother = $2;}
           ;
type       : INT {$$ =ntn(NTYPE);
                  $$->child = keynode(T_INT);}
           | STRING {$$ =ntn(NTYPE);
                     $$->child = keynode(T_STRING);}
           ;
var_list   : ID {$$ = idnode();} ',' var_list {$$ = ntn(NVAR_LIST);
                                               $$->child = $2;
                                               $2->child = $3 = tn(NCOMMA);
                                               $3->brother = $4;}
           | ID {$$ = ntn(NVAR_LIST);
                 $$->child = idnode();}
           ;
%%
main(){yyparse();}
```



Exercise 5

Define an unambiguous grammar G for the language specified by the following ambiguous grammar

$L \rightarrow E$ **eol**
$E \rightarrow E$ **+** $E \mid E$ **-** $E \mid E$ ***** $E \mid E$ **/** $E \mid E$ **^** $E \mid$ **(** E **)** $\mid$ **rconst**

based on the following rules: sum (**+**) and difference (**-**) have minimum precedence and right associativity; multiplication (*****) and division (**/**) have intermediate precedence and left associativity; exponentiation (**^**) has maximum precedence and right associativity; **rconst** is a real constant.
Then, codify in *Yacc* the calculator of the expressions of **G**.

Exercise 5

Define an unambiguous grammar G for the language specified by the following ambiguous grammar

$L \rightarrow E$ **eol**
$E \rightarrow E$ **+** $E \mid E$ **-** $E \mid E$ ***** $E \mid E$ **/** $E \mid E$ **^** $E \mid$ **(** E **)** $\mid$ **rconst**

based on the following rules: sum (**+**) and difference (**-**) have minimum precedence and right associativity; multiplication (*****) and division (**/**) have intermediate precedence and left associativity; exponentiation (**^**) has maximum precedence and right associativity; **rconst** is a real constant.
Then, codify in *Yacc* the calculator of the expressions of **G**.

Operators	Associativity	Nonterminal
+, -	right	*expr*
*, /	left	*term*
^	right	*sup*
		factor

line → *expr* **eol**
expr → *term* **+** *expr* | *term* **-** *expr* | *term*
term → *term* ***** *sup* | *term* **/** *sup* | *sup*
sup → *factor* **^** *sup* | *factor*
factor → **(** *expr* **)** | **rconst**

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE float
%}
%token RCONST
%%
line   :  expr '\n' {printf("%f\n", $1);}
expr   :  term '+' expr {$$ = $1 + $3;}
       |  term '-' expr {$$ = $1 - $3;}
       |  term
       ;
term   :  term '*' sup {$$ = $1 * $3;}
       |  term '/' sup {$$ = $1 / $3;}
       |  sup
       ;
sup    :  factor '^' sup {$$ = power($1,$3);}
       |  factor
       ;
factor :  '(' expr ')'{$$ = $2;}
       |  RCONST
       ;
%%
```

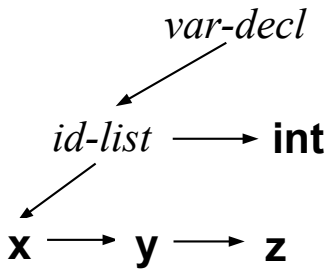
```
yylex()
{
  int c;

  while((c=getchar()) == ' ' || c == '\t')
     ;
  if (isdigit(c))
  {
    ungetc(c, stdin);
    scanf("%f", &yylval);
    return(RCONST);
  }
  return(c);
}

yyerror()
{ fprintf(stderr, "Syntax error\n") };

main()
{ yyparse(); }
```

Exercise 6

Codify in *Yacc* the generator of the abstract syntax trees relevant to the following grammar (Note: it is not required the specification of the lexical analyzer):

program → *var-decl*
var-decl → *id-list* **:** *type*
id-list → **id ,** *id-list* | **id**
type → **int** | **real** | **string**

Here is an example of mapping: `x, y, z: int` ⇨

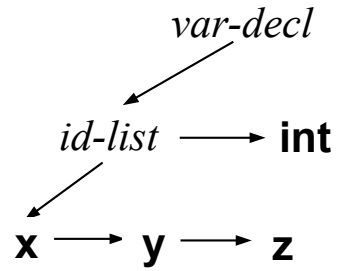


Exercise 6

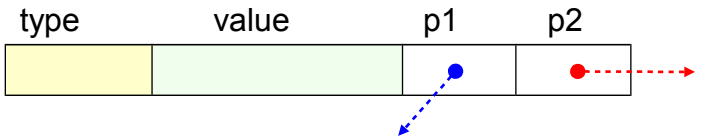
Codify in *Yacc* the generator of the abstract syntax trees relevant to the following grammar (Note: it is not required the specification of the lexical analyzer):

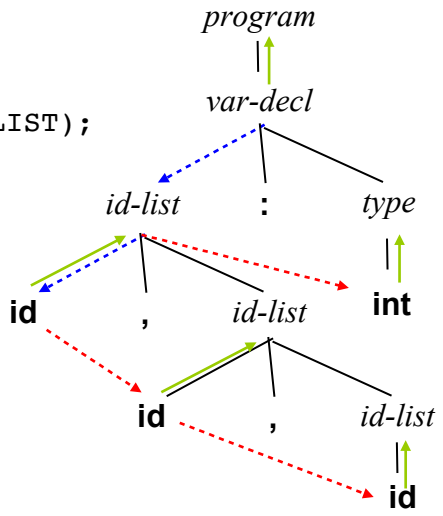
program → *var-decl*
var-decl → *id-list* **:** *type*
id-list → **id** **,** *id-list* | **id**
type → **int** | **real** | **string**

Here is an example of mapping: `x, y, z: int` ⇒




```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
char *lexval;
%}
%token INT REAL STRING ID ERROR
%%
program  :  var_decl {root = $1;}
         ;
var_decl :  id_list ':' type {$$ = nontermnode(VAR_DECL); $$->p1 = nontermnode(ID_LIST);
                              $$->p1->p1 = $1; $$->p1->p2 = $3;}
         ;
id_list  :  ID {$$ = idnode();} ',' id_list {$$ = $2; $2->p2 = $4;}
         |  ID {$$ = idnode();}
         ;
type     :  INT {$$ = intnode();}
         |  REAL {$$ = realnode();}
         |  STRING {$$ = stringnode();}
         ;
%%
...
```





Exercise 7

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

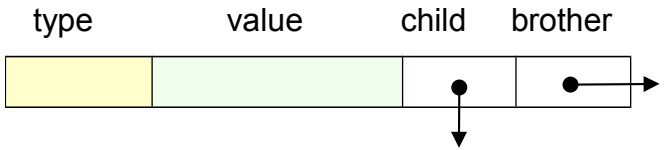
program → *def-table select-op*
def-table→ **table id (** *type-list* **)**
type-list → *type-list* **,** *type* | *type*
type → **string** | **bool**
select-op→ **select id where numattr =** *const*
const → **strconst** | **boolconst**

(example of sentence)

```
table T (string, bool)
select T where 1 = "alpha"
```

Exercise 7

```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
Value lexval;
%}
%token TABLE ID STRING BOOL SELECT WHERE NUMATTR STRCONST BOOLCONST ERROR
%%
program     : def_table select_op {root = $$ = ntn(NPROGRAM);
                                   $$->child = $1;
                                   $1->brother = $2;}
            ;
def_table   : TABLE ID {$$=idnode();}
              '(' type_list ')' {$$ = ntn(NDEF_TABLE);
                                 $$->child = $3;
                                 $3->brother = $5;}
            ;
type_list   : type_list ',' type {$$ = ntn(NTYPE_LIST);
                                  $$->child = $1;
                                  $1->brother = $3;}
            | type {$$ = ntn(NTYPE_LIST);
                    $$->child = $1;}
            ;
type        : STRING {$$ =ntn(NTYPE);
                      $$->child = keynode(T_STRING);}
            | BOOL {$$ =ntn(NTYPE);
                    $$->child = keynode(T_BOOL);}
            ;
select_op   : SELECT ID {$$ = idnode();}
              WHERE NUMATTR {$$ = numnode();} '=' const
              {$$ = ntn(NSELECT_OP);
               $$->child = $3;
               $3->brother = $6;
               $6->brother = $8;}
            ;
const       : STRCONST {$$ =ntn(NCONST);
                        $$->child = strconstnode();}
            | BOOLCONST {$$ =ntn(NCONST);
                         $$->child = boolconstnode();}
            ;
%%
main(){yyparse();}
```



program → *def-table select-op*
def-table → **table id (** *type-list* **)**
type-list → *type-list* **,** *type* | *type*
type → **string | bool**
select-op → **select id where numattr =** *const*
const → **strconst | boolconst**

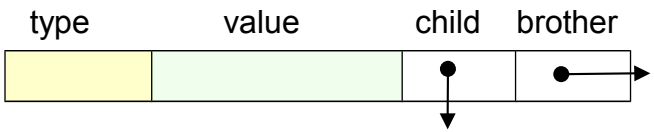
Exercise 8

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

program → *stat-list*
stat-list → *stat* **;** *stat-list* | **ε**
stat → *def-stat* | *if-stat* | **display**
def-stat → **id :** *type*
type → **int** | **string**
if-stat → **if** *expr* **then** *stat* **else** *stat*
expr → **boolconst**

Exercise 8

```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
Value lexval;
%}
%token DISPLAY ID INT STRING IF THEN ELSE BOOLCONST ERROR
%%
program    : stat_list {root = $$ = ntn(NPROGRAM); $$->child = $1;}
           ;
stat_list  : stat ';' stat_list {$$ = ntn(NSTAT_LIST);
                                 $$->child = $1;
                                 $1->brother = $3;}
           | {$$ = ntn(NSTAT_LIST);}
           ;
stat        : def_stat {$$ = ntn(NSTAT); $$->child = $1;}
            | if_stat {$$ = ntn(NSTAT); $$->child = $1;}
            | DISPLAY {$$ = ntn(NSTAT); $$->child = keynode(DISPLAY);}
           ;
def_stat    : ID {$$ = idnode();} ':' type {$$ =ntn(NDEF_STAT);
                                            $$->child = $2;
                                            $2->brother = $4;}
           ;
type        : INT {$$ =ntn(NTYPE); $$->child = keynode(INT);}
            : STRING {$$ =ntn(NTYPE); $$->child = keynode(STRING);}
            ;
if_stat    : IF expr THEN stat ELSE stat {$$ = ntn(NIF_STAT);
                                          $$->child = $2;
                                          $2->brother = $4;
                                          $4->brother = $6;}
           ;
expr        : BOOLCONST {$$ =ntn(NEXPR); $$->child = boolconstnode();}
            ;
%%
main(){yyparse();
```



program → *stat-list*
stat-list → *stat* **;** *stat-list* | **ε**
stat → *def-stat* | *if-stat* | **display**
def-stat → **id :** *type*
type → **int** | **string**
if-stat → **if** *expr* **then** *stat* **else** *stat*
expr → **boolconst**

Exercise 9

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

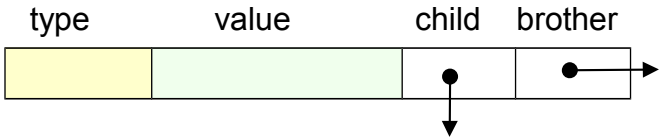
program → *vector-decl display-stat*
vector-decl→ **id : vector [intconst] of** *type*
type → **int** | **string**
display-stat→ **display (** *type*, **id [intconst])**

(example of sentence)

```
v: vector [10] of string
display(string, v[7])
```

Exercise 9

```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
Value lexval;
%}
%token ID VECTOR INTCONST OF INT STRING DISPLAY ERROR
%%
program : vector_decl display_stat
          {root = $$ = ntn(NPROGRAM);
          $$->child = $1;
          $1->brother = $2}
          ;

vector_decl : ID {$$ = idnode();} ':' VECTOR '[' INTCONST {$$ = intconstnode();} ']' OF type
              {$$ = ntn(NVECTOR_DECL);
              $$->child = $2;
              $2->brother = $7;
              $6->brother = $10;}
              ;

type : INT {$$ =ntn(NTYPE); $$->child = keynode(INT);}
     : STRING {$$ =ntn(NTYPE); $$->child = keynode(STRING);}
     ;

display_stat : DISPLAY '(' type ',' ID {$$ = idnode();} '[' INTCONST {$$ = intconstnode();} ']' ')'
               {$$ = ntn(DISPLAY_STAT);
                SS->child = $3;
                $3->brother = $6;
                $6->brother = $9;}
%%
main(){yyparse();}
```



program → *vector-decl display-stat*
vector-decl→ **id : vector [intconst] of** *type*
type → **int** | **string**
display-stat→ **display (** *type*, **id [intconst])**

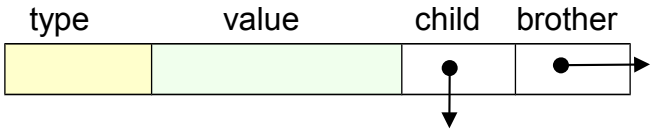
Exercise 10

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

program → *def-list*
def-list → *def* **;** *def-list* | **ε**
def → *type-def* | *function-def*
type-def → **type id =** *domain*
domain → **int** | **string** | **[** *domain* **]**
function-def → **function id (** *param-list* **) :** *domain*
param-list → *param* **,** *param-list* | **ε**
param → **id :** *domain*

Exercise 10

```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
Value lexval;
%}
%token ID TYPE INT STRING FUNCTION INTCONST STRCONST ERROR
%%
program : def_list
          {root = $$ = ntn(NPROGRAM);
          $$->child = $1; }
        ;

def_list : def ';' def_list
           {$$ = ntn(NDEF_LIST);
            $$->child = $1;
            $1->brother = $3;}
         | {$$ = ntn(NDEF_LIST);}
         ;

def : type_def
      {$$ = ntn(NDEF}; $$->child = $1;}
    | function_def
      {$$ = ntn(NDEF}; $$->child = $1;}
    ;

type_def : TYPE ID {$$ = idnode();} '=' domain
           { $$ = ntn{NTYPE_DEF};
             $$->child = $3;
             $3->brother = $5;
           }
         ;
```



program → *def-list*
def-list → *def* **;** *def-list* | **ε**
def → *type-def* | *function-def*
type-def → **type id** = *domain*

Exercise 10 (ii)

```
domain : INT {$$ =ntn(NDOMAIN); $$->child = keynode(INT);}
       | STRING {$$ =ntn(NDOMAIN); $$->child = keynode(STRING);}
       | '[' domain ']' {$$ =ntn(NDOMAIN); $$->child = $2;}
       ;

function_def : FUNCTION ID {$$ = idnode();} '(' param_list ')' ':' domain
               { $$ = ntn(NFUNCTION_DEF);
                 $$->child = $3;
                 $3->brother = $5;
                 $5->brother = $8;}

param_list : param ',' param_list
             { $$ = ntn(NPARAM_LIST);
               $$-> child = $1;
               $1->brother = $3;}
           | { $$ = ntn(NPARAM_LIST) }
           ;

param : ID {$$ = idnode();} ':' domain
        { $$ = ntn(NPARAM);
          $$->child = $2;
          $2->brother = $4;}
%%
main()
{
  yyparse();
}
```

domain → **int** | **string** | **[** *domain* **]**
function-def → **function id (** *param-list* **) :** *domain*
param-list → *param* **,** *param-list* | **ε**
param → **id :** *domain*

Exercise 11

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF, based on the following information:

program → *type-def*
type-def → **id** **:** *type*
type → **int** | **string** | *ptr-type* | *rec-type* | *vect-type*
ptr-type → **^** *type*
rec-type → **record** **(** *type-def-list* **)**
type-def-list → *type-def* *type-def-list* | **ε**
vect-type → **vector** **[** **intconst** **]** **of** *type*

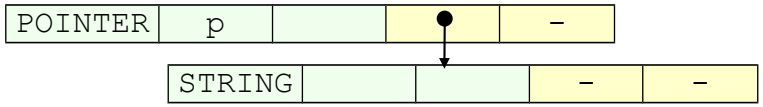
- Each node of the abstract tree is structured by the following fields:

  `type` : type of node { `INT`, `STRING`, `POINTER`, `RECORD`, `VECTOR` }
  `name` : name of variable or record field
  `num` : vector size
  `p1`: pointer to first child (if structured type)
  `p2`: pointer to right brother (if fields of record)

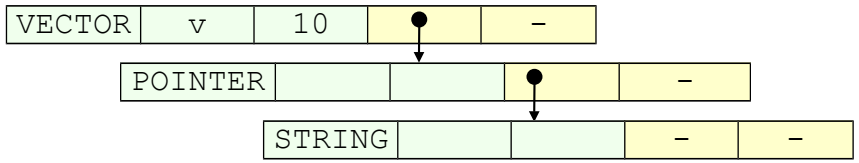
- Here are some examples of mapping:

```
alpha : int
```

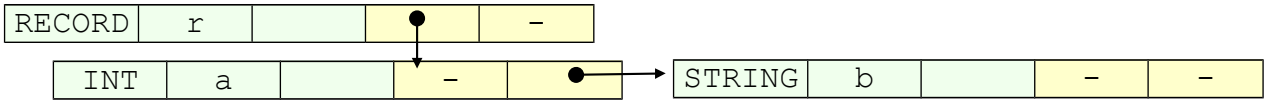

```
p : ^ string
```




```
v : vector [10] of ^ string
```



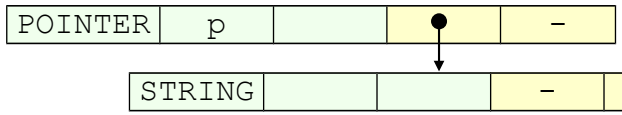

```
r : record (a: int b: string)
```

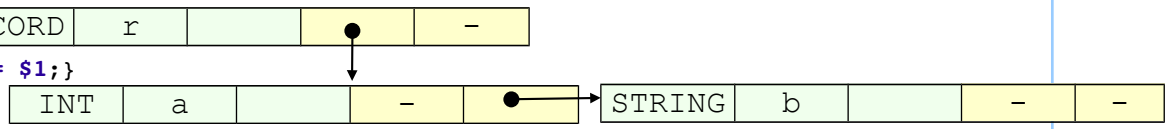


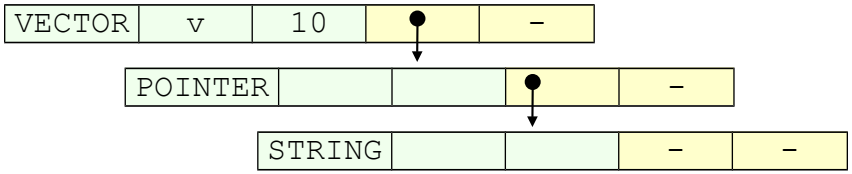
Exercise 11

```
%{
#include "stdlib.h"
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
Value lexval; /* union: int ival, char *sval */
%}
%token ID INT STRING RECORD VECTOR INTCONST OF
%%
program : type_def {root = $1;}
        ;

type_def : ID {$$ = (PNODE) lexval.sval;} ':' type
           {$$ = $4; $$->name = (char *) $2;}
         ;

type : INT {$$ = newnode_simple(INT);}
     | STRING {$$ = newnode_simple(STRING);}
     | ptr_type {$$ = $1;}
     | rec_type {$$ = $1;}
     | vect_type {$$ = $1;}
     ;

ptr_type : '^' type {$$ = newnode_complex(POINTER, $2);}
         ;

rec_type : RECORD '(' type_def_list ')' {$$ = newnode_complex(RECORD, $3);}
         ;

type_def_list : type_def type_def_list {$1->p2 = $2; $$ = $1;}
              | {$$ = NULL;}
              ;

vect_type : VECTOR '[' INTCONST {$$ = (PNODE) lexval.ival;} ']' OF type
           {$$ = newnode_complex(VECTOR, $7); $$->num = (int) $4;}
          ;
%%
PNODE *newnode_simple(int type)
{  PNODE *p = malloc(sizeof(PNODE));
   p->type = type; p->p1 = p->p2 = NULL; return(p);
}

PNODE *newnode_complex(int constructor, PNODE elem_type)
{  PNODE *p = newnode_simple(constructor);
   p->p1 = elem_type; return(p);
}
main(){yyparse();}
```

program → *type-def*
type-def → **id** : *type*
type → **int** | **string** | *ptr-type* | *rec-type* | *vect-type*
ptr-type → ^ *type*
rec-type → **record** (*type-def-list*)
type-def-list → *type-def* *type-def-list* | **ε**
vect-type → **vector** [**intconst**] **of** *type*







Exercise 12

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

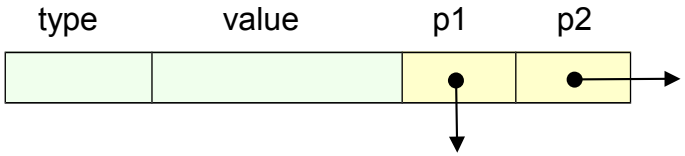
program → *def assign*
def → **id : matrix [num, num] of** *type*
type → **integer** | **string**
assign → **id :=** **[** *vector-list* **]**
vector-list → *vector* **,** *vector-list* | *vector*
vector→ **[** *const-list* **]**
const-list → *const*, *const-list* | *const*
const → **intconst** | **stringconst**

```
alfa: matrix[3,4] of integer
alfa := [ [10, 15, 20, 25],
          [30, 40, 50, 60],
          [12, 13, 14, 15] ]
```

Exercise 12

```
%{
#include "stdlib.h"
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
Lexval val;
%}
%token ID MATRIX NUM OF INTEGER STRING ID ASSIGN INTCONST STRCONST
%%
program : def assign
          {root = $$ = ntn(NPROGRAM}; $$->p1 = $1; $1->p2 = $2;}
        ;

def : ID {$$ = idnode();} ':' MATRIX
      '[' NUM {$$ = numnode();} ',' NUM {$$ = numnode();} OF type
      {$$ = ntn(NDEF); $$->p1 = $2; $2->p2 = $7; $7->p2 = $10; $10->p2 = $13;}
      ;

type : INTEGER {$$ = ntn(NTYPE); $$->p1 = keynode(INTEGER);}
     | STRING {$$ = ntn(NTYPE); $$->p1 = keynode(STRING);}
     ;

assign : ID {$$ = idnode();} ASSIGN '[' vector_list ']'
         {$$ = ntn(NASSIGN); $$->p1 = $2; $2->p2 = $5;}
       ;

vector_list : vector ',' vector_list
              {$$ = ntn(NVECTOR_LIST}; $$->p1 = $1; $1->p2 = $3;}
            | vector {$$ = ntn(NVECTOR_LIST}; $$->p1 = $1;}
            ;

vector : '[' const_list ']' {$$ = ntn(NVECTOR); $$->p1 = $2;}
       ;

const_list : const ',' const_list
             {$$ = ntn(NCONST_LIST); $$->p1 = $1; $1->p2 = $3;}
             | const {$$ = ntn(NCONST_LIST); $$->p1 = $1;}
             ;

const : INTCONST {$$ = ntn(NCONST); $$->p1 = intconstnode();}
      | STRCONST {$$ = ntn(NCONST); $$->p1 = strconstnode();}
      ;
```

program → *def assign*
def → **id : matrix [num, num] of** *type*
type → **integer** | **string**
assign → **id :=** **[** *vector-list* **]**
vector-list → *vector* **,** *vector-list* | *vector*
vector → **[** *const-list* **]**
const-list → *const*, *const-list* | *const*
const → **intconst** | **stringconst**



Exercise 13

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

program → *stat*
stat → *def-stat* | *assign-stat*
def-stat → **relation id :** *rel-type*
rel-type → **[** *attr-list* **]**
attr-list → *attr-def attr-list* | **ε**
attr-def → **id :** *type*
type → *atomic-type* | *rel-type*
atomic-type → **int** | **string**
assign-stat → **id :=** *const*
const → **intconst** | **strconst**

based on the following abstract EBNF:

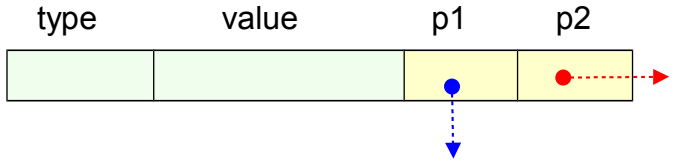
program → *stat*
stat → *def-stat* | *assign-stat*
def-stat → **id** *rel-type*
rel-type → { *attr-def* }
attr-def → **id** *type*
type → *atomic-type* | *rel-type*
atomic-type → **int** | **string**
assign-stat → **id** *const*
const → **intconst** | **strconst**

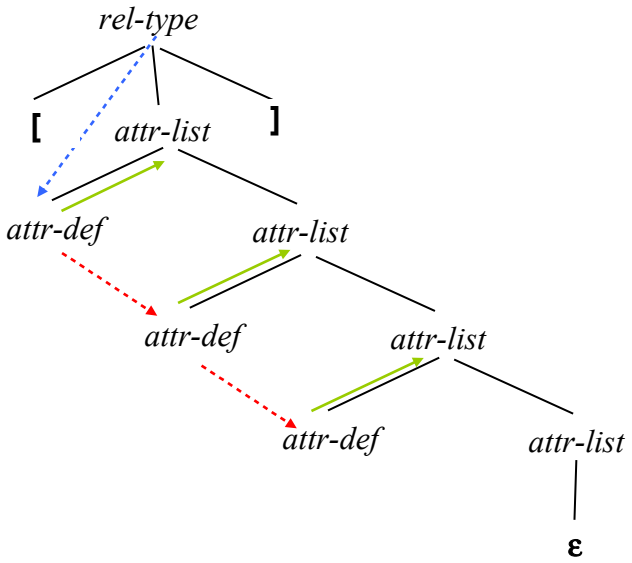
Exercise 13

```
%{
#include "stdlib.h"
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
Lexval val;
%}
%token RELATION ID INT STRING ASSIGN INTCONST STRCONST
%%
program : stat
          {root = $$ = ntn(NPROGRAM}; $$->p1 = $1;}
        ;

stat : def_stat {$$ = ntn(NSTAT}; $$->p1 = $1;}
     | assign_stat {$$ = ntn(NSTAT}; $$->p1 = $1;}
     ;

def_stat : RELATION ID {$$ = idnode();} ':' rel_type
           {$$ = ntn(NDEF_STAT); $$->p1 = $3; $3->p2 = $5;}
         ;

rel_type : '[' attr_list ']' {$$ = ntn(NREL_TYPE); SS->p1 = $2;}
         ;

attr_list : attr_def attr_list {$$ = $1; $1->p2 = $2;}
          | {$$ = NULL;}
          ;

attr_def : ID {$$ = idnode();} ':' type {$$ = ntn(NATTR_DEF}; $$->p1 = $2; $2->p2 = $4;}
         ;

type : atomic_type {$$ = ntn(N_TYPE); $$->p1 = $1;}
     | rel_type {$$ = ntn(N_TYPE); $$->p1 = $1;}
     ;

atomic_type : INT {$$ = ntn(NATOMIC_TYPE); $$->p1 = keynode(INT);}
            | STRING {$$ = ntn(NATOMIC_TYPE); $$->p1 = keynode(STRING);}
            ;

assign_stat : ID {$$ = idnode();} ASSIGN const
         {$$ = ntn(NASSIGN); $$->p1 = $2; $2->p2 = $4;}
       ;

const : INTCONST {$$ = ntn(NCONST); $$->p1 = intconstnode();}
      | STRCONST {$$ = ntn(NCONST); $$->p1 = strconstnode();}
      ;
%%
```

program → *stat*
stat → *def-stat* | *assign-stat*
def-stat → **relation id :** *rel-type*
rel-type → **[** *attr-list* **]**
attr-list → *attr-def attr-list* | **ε**
attr-def → **id :** *type*
type → *atomic-type* | *rel-type*
atomic-type → **int** | **string**
assign-stat → **id :=** *const*
const → **intconst** | **strconst**

program → *stat*
stat → *def-stat* | *assign-stat*
def-stat → **id** *rel-type*
rel-type → { *attr-def* }
attr-def → **id** *type*
type → *atomic-type* | *rel-type*
atomic-type → **int** | **string**
assign-stat → **id** *const*
const → **intconst** | **strconst**





Exercise 14

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

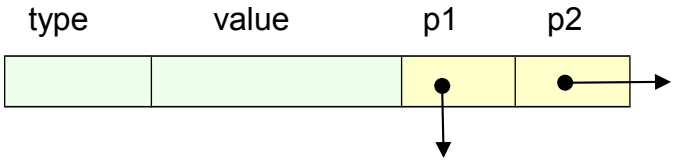
program → *def-table* *update-op*
def-table → **table id (** *attr-list* **)**
attr-list → *attr* **,** *attr-list* **|** *attr*
attr → **id :** *type*
type → **int | real**
update-op → **update [id =** *expr* **] id**
expr → *expr* **+** *term* **|** *term*
term → **id | intconst | realconst**

```
table T (a: int, b: real, c: int)
update [ a = a + c + 2 ] T
```

Exercise 14

```
%{
#include "stdlib.h"
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
Lexval val;
%}
%token TABLE ID INT REAL UPDATE INTCONST REALCONST
%%
program : def_table update_op
          {root = $$ = ntn(NPROGRAM}; $$->p1 = $1; $1->p2 = $2}
        ;

def_table : TABLE ID {$$ = idnode();} '(' attr_list ')'
            {$$ = ntn(NDEF_TABLE}; $$->p1 = $3; $3->p2 = $5;}
          ;

attr_list : attr ',' attr_list
           {$$ = ntn(NATTR_LIST); $$->p1 = $1; $1->p2 = $3;}
          | attr {$$ = ntn(NATTR_LIST); $$->p1 = $1;}
          ;

attr : ID {$$ = idnode();}':' type
       {$$ = ntn(NATTR); SS->p1 = $2; $2->p2 = $4;}
     ;

type : INT {$$ = ntn(NTYPE); $$->p1 = keynode(INT);}
     | REAL {$$ = ntn(NTYPE); $$->p1 = keynode(REAL);}
     ;

update_op : UPDATE '[' ID {$$ = idnode();}'=' expr ']' ID
            {$$ = ntn(NUPDATE_OP); $$->p1 = $4; $4->p2 = $6; $6->p2 = idnode();}
          ;

expr : expr '+' term {$$ = ntn(NEXPR); $$->p1 = $1; $1->p2 = $3;}
     | term {$$ = ntn(NEXPR); $$->p1 = $1;}
     ;

term : ID {$$ = ntn(NTERM): $$->p1 = idnode();}
     | INTCONST {$$ = ntn(NTERM); $$->p1 = intconstnode();}
     | REALCONST {$$ = ntn(NTERM); $$->p1 = realconstnode();}
     ;
%%
```

program → *def-table update-op*
def-table → **table id (** *attr-list* **)**
attr-list → *attr* **,** *attr-list* **|** *attr*
attr → **id :** *type*
type → **int | real**
update-op → **update [id =** *expr* **] id**
expr → *expr* **+** *term* **|** *term*
term → **id | intconst | realconst**



Exercise 15

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

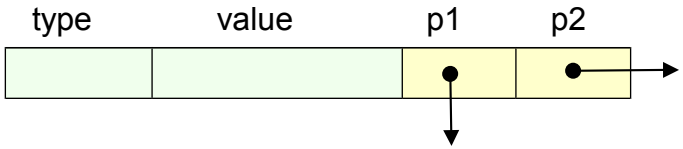
program → **automaton id is states** *id-list*; **initial id; finals** *id-list*; **transitions** *trans-list*; **end id.**
id-list → **id** *id-list* | **id**
trans-list → *trans* *trans-list* | *trans*
trans → **(id, id, id)**

```
automaton A is
  states a, b, c;
  initial a;
  finals b, c;
  transitions (a,x,b), (b,y,c);
end A.
```

Exercise 15

program → **automaton id is states** *id-list*; **initial id; finals** *id-list*; **transitions** *trans-list*; **end id.**
id-list → **id** *id-list* | **id**
trans-list → *trans* *trans-list* | *trans*
trans → **(id, id, id)**

```
%{
#include "stdlib.h"
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
Lexval val;
%}
%token AUTOMATON ID IS STATES INITIAL FINALS TRANSITIONS END
%%
program : AUTOMATON ID {$$ = idnode();} IS
          STATES id_list ';'
          INITIAL ID {$$ = idnode();} ';'
          FINALS id_list ';'
          TRANSITIONS trans_list ';'
          END ID {$$ = idnode();}';'
          {root = $$ = ntn(NPROGRAM};
           $$->p1 = $3; $3->p2 = $6; $6->p2 = $10; $10->p2 = $13; $13->p2 = $16; $16->p2 = $20;}
        ;

id_list : ID {$$ = idnode();} id_list  {$$ = ntn(NID_LIST); $$->p1 = $2; $2->p2 = $3;}
        | ID                           {$$ = ntn(NID_LIST); $$->p1 = idnode();}
        ;

trans_list : trans trans-list {$$ = ntn(NTRANS_LIST); $$->p1 = $1; $1->p2 = $2;}
           | trans {$$ = ntn(NTRANS_LIST); $$->p1 = $1;}
           ;
trans : '(' ID {$$ = idnode();} ',' ID {$$ = idnode();} ',' ID {$$ = idnode();} ')'
        {$$ = ntn(NTRANS); $$->p1 = $3, $3->p2 = $6, $6->p2 = $9;}
%%
```



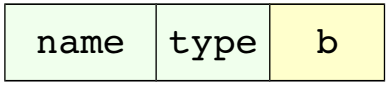
Exercise 16

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF,

program → def-list
def-list → def , def-list | def
def → **id :** type
type → **int** | **string** | set
set → **{** type **}**

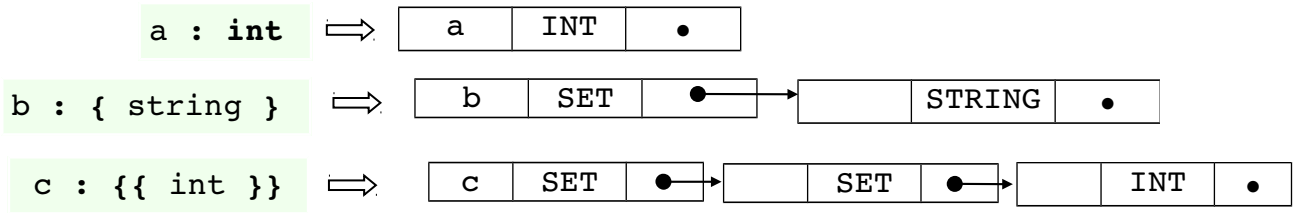
based on the following information:

- Each node of the abstract tree is structured by the following fields:



(`char *`)	`name` : name of the identifier
(`Typenode`)	`type` : type of node { `INT`, `STRING`, `SET` }
(`PNODE`)	`b`: pointer to brother node (when `type` = `SET`)

- Here are some mapping examples:



- To create a node qualified with its `type`, the following auxiliary function is available:
  `PNODE newnode(Typenode type)`
- Each defined variable is stored in a symbol table by means of the procedure `insert(name, root)`, where `root` is the root of the structure of the type of the variable identified by `name`.

Exercise 16

```
%{
#include "def.h"
#define YYSTYPE PNODE
char *lexval;
%}
%token ID INT STRING ERROR
%%
program : def_list
        ;

def_list : def ',' def_list {insert($1->name, $1);}
         | def {insert($1->name, $1);}
         ;

def : ID {$$ = (PNODE) lexval;} ':' type {$$ = $4; $$->name = (char *) $2;}
    ;

type : INT {$$ = newnode(INT);}
     | STRING {$$ = newnode(STRING);}
     | set {$$ = $1;}
     ;

set : '{' type '}' {$$ = newnode(SET); $$->b = $2;}
    ;

%%
```

name	type	b

program → def-list
def-list → def , def-list | def
def → **id** : type
type → **int** | **string** | set
set → **{** type **}**

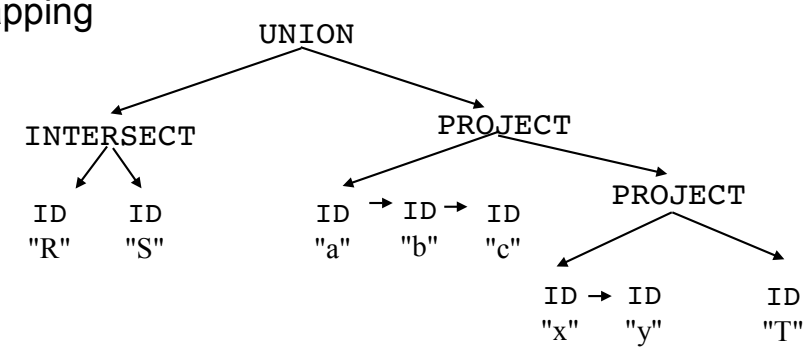
Exercise 17

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

program → *expr*
expr → *expr* **union** *term* | *expr* **intersect** *term* | *term*
term → **project [** *id-list* **]** *term* | *factor*
id-list → **id ,** *id-list* | **id**
factor → **id** | **(** *expr* **)**

example of mapping

```
(R intersect S) union project [a,b,c] project [x,y] T
```



- Each node of the abstract tree is structured by the following fields:

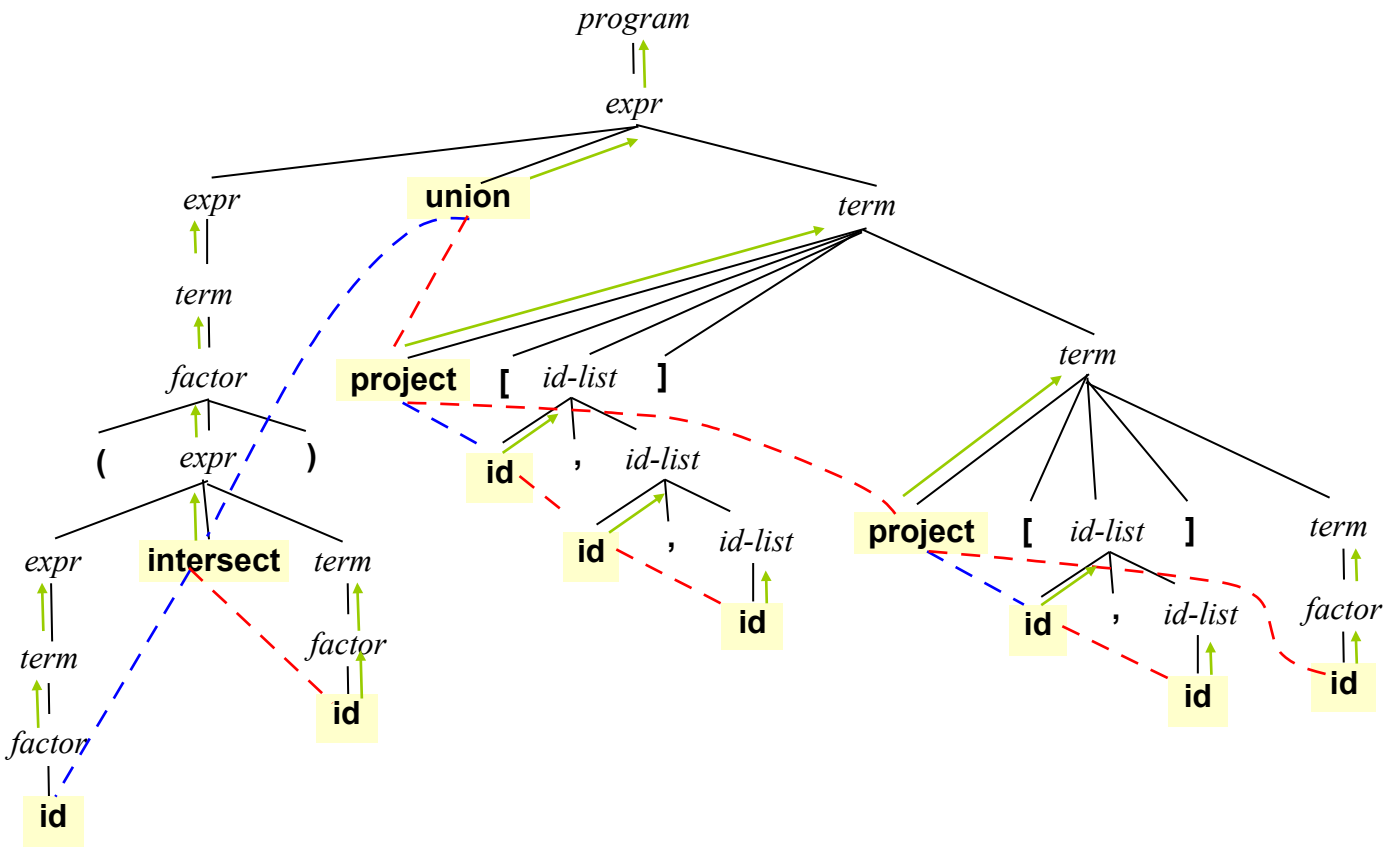
(`Typenode`)	`type` : type of node { `UNION`, `INTERSECT`, `PROJECT`, `ID` }
(`char *`)	`name` : name of identifier when `type=ID`
(`PNODE`)	`p1`, `p2`: pointers to next nodes

- For union and intersection, `p1` and `p2` point to operands;
- For projection, `p1` points to the first attribute, `p2` points to the operand of projection;
- Projection attributes are linked (in linear list) by means of pointer `p2`.
- The following auxiliary functions can be used:

  `PNODE opnode(Typenode op)`: to create an internal node,

  `PNODE idnode()`: to create a leaf node

Exercise 17


program
expr
expr
union
term
term
factor
project
[
id-list
]
term
(
expr
)
id
,
id-list
expr
intersect
term
id
,
id-list
project
[
id-list
]
term
term
factor
id
id
id
,
id-list
factor
factor
id
id
id

Exercise 17 (ii)

```
%{
#include "def.h"
#define YYSTYPE PNODE
%}
%token ID UNION INTERSECT PROJECT ERROR
%%
program : expr {root = $1;}
        ;

expr : expr UNION term {$$ = opnode(UNION); $$->p1 = $1; $$->p2 = $3;}
     | expr INTERSECT term {$$ = opnode(INTERSECT); $$->p1 = $1; $$->p2 = $3;}
     | term
     ;

term : PROJECT '[' id-list ']' term {$$ = opnode(PROJECT); $$->p1 = $3; $$->p2 = $5;}
     | factor
     ;

id-list : ID {$$ = idnode();} ',' id-list {$$ = $2; $2->p2 = $4;}
        | ID {$$ = idnode();}
        ;

factor : ID {$$ = idnode();}
       | '(' expr ')' {$$ = $2;}
       ;
%%
...
```

program → *expr*
expr → *expr* **union** *term* | *expr* **intersect** *term* | *term*
term → **project [** *id-list* **]** *term* | *factor*
id-list → **id ,** *id-list* | **id**
factor → **id** | **(** *expr* **)**

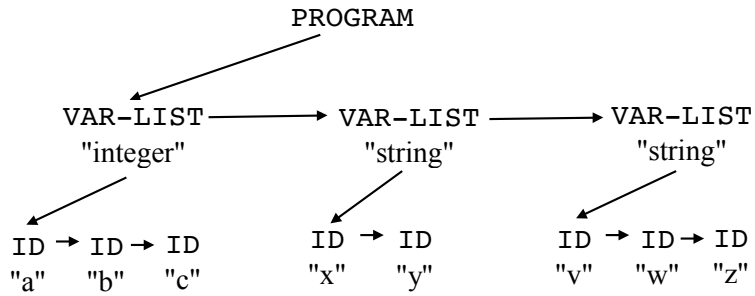
Exercise 18

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

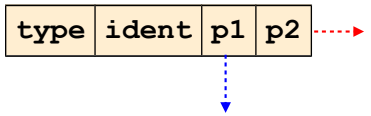
program → *var-decl*
var-decl → *var-list var-decl* | *var-list*
var-list → *id-list* **:** *type* **;**
id-list → **id ,** *id-list* | **id**
type → **integer** | **string**

example of mapping:

```
a, b, c: integer;
x, y: string;
v, w, z: string;
```



- Each node of the abstract tree is structured by the following fields:



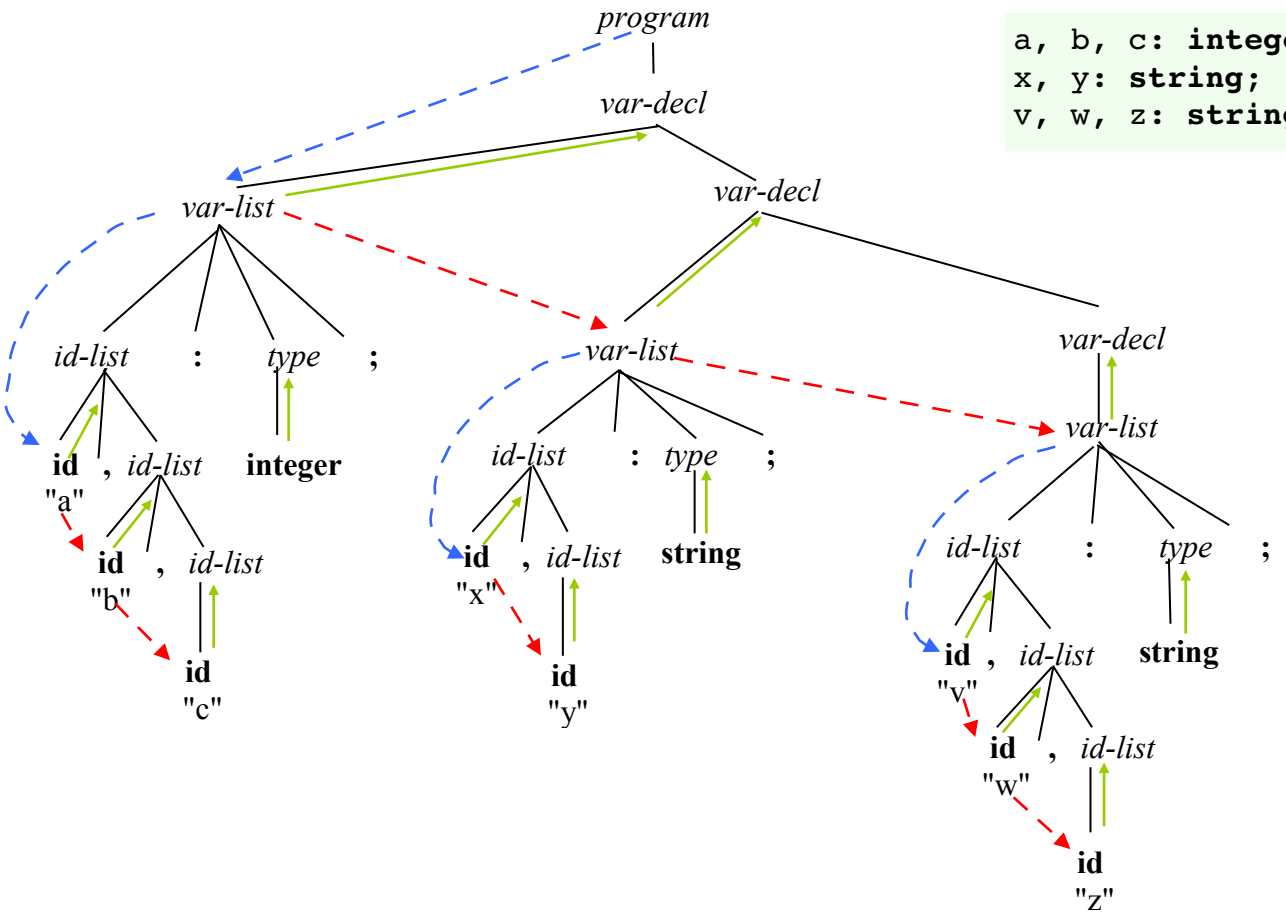
(Typenode) `type` : type of node { `PROGRAM`, `VAR-LIST`, `ID` }
(`char *`) `ident` : when `type` =`ID`, name of identifier,
when `type` =`VAR-LIST`, name of (simple) type
(`PNODE`) `p1`, `p2`: pointer to first child and pointer to right brother, respectively

- The following auxiliary functions can be used:

`PNODE newnode(Typenode)`: to create a node with `ident` = `NULL`,

`PNODE newnodeid(Typenode, char*)`: to create a node with `ident` instantiated.

Exercise 18




```
a, b, c: integer;
x, y: string;
v, w, z: string;
```

Exercise 18 (ii)

```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE root;
%}
%token ID INTEGER STRING ERROR
%%
program : var-decl {root = newnode(PROGRAM); root->p1 = $1;}
        ;

var-decl : var-list var-decl {$$ = $1; $$->p2 = $2;}
         | var-list {$$ = $1;}
         ;

var-list : id-list ':' type ';' {$$ = newnodeid(VAR-LIST, (char *) $3); $$->p1 = $1;}
         ;

id-list : ID {$$ = newnodeid(ID, lexval);} ',' id-list {$$ = $2; $2->p2 = $4;}
        | ID {$$ = newnodeid(ID, lexval);}
        ;

type : INTEGER {$$ = (PNODE) "integer";}
     | STRING {$$ = (PNODE) "string";)}
     ;
%%
...
```

program → *var-decl*
var-decl → *var-list var-decl* | *var-list*
var-list → *id-list* **:** *type* **;**
id-list → **id** **,** *id-list* | **id**
type → **integer** | **string**

Exercise 19

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF,

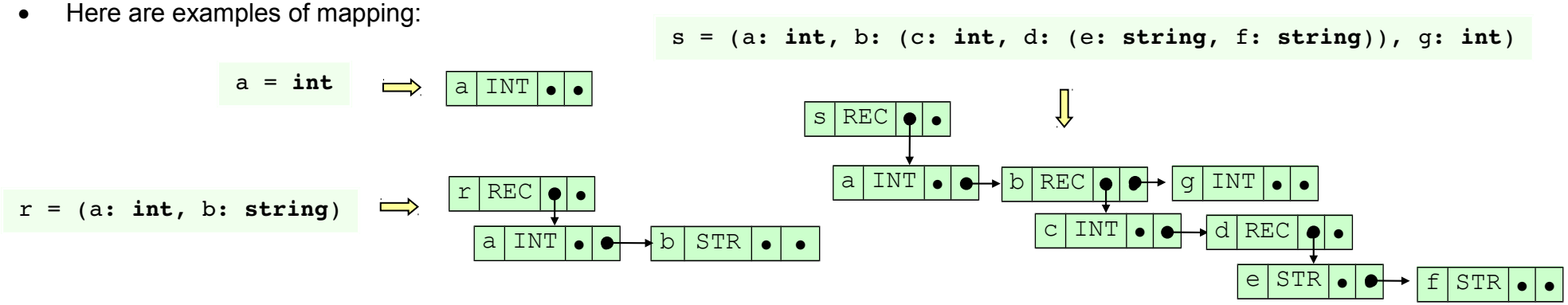
program → *decl-list*
decl-list → *decl* *decl-list* | *decl*
decl → **id** = *domain* **;**
domain → **int** | **string** | *record*
record → **(** *attr-list* **)**
attr-list → *attr* **,** *attr-list* | *attr*
attr → **id :** *domain*

based on the following information:

- Each node of the abstract tree is structured by the following fields:

`(char *)` `name` : name of identifier
`(Typenode)` `type` : type of node { `INT`, `STR`, `REC` }
`(PNODE)` `p1`: pointer to first child (when type `REC`)
`(PNODE)` `p2`: pointer to right brother (next record attribute)

- Here are examples of mapping:



- To create a node qualified with its type, the following auxiliary function can be used: `PNODE create_node(Typenode type)`

Each defined variable is stored into a symbol table by procedure `insert(name, root)`, where `root` is the root of the structure of the type of the variable identified by `name`.

Exercise 19

```
%{
#include "def.h"
#define YYSTYPE PNODE
char *lexval;
%}
%token ID INT STRING ERROR
%%
program : decl-list
        ;
decl-list : decl decl-list {insert($1->name, $1);}
          | decl {insert($1->name, $1);}
          ;
decl : ID {$$ = (PNODE) lexval;} '=' domain ';' {$$ = $4; $$->name = (char *) $2;}
     ;
domain : INT {$$ = create_node(INT);}
       | STRING {$$ = create_node(STR);}
       | record {$$ = $1;}
       ;
record : '(' attr-list ')' {$$ = create_node(REC); $$->p1 = $2;}
       ;
attr-list : attr ',' attr-list {$$ = $1; $1->p2 = $3;}
          | attr {$$ = $1}
          ;
attr : ID {$$ = (PNODE) lexval;} ':' domain {$$ = $4; $$->name = (char *) $2;}
     ;
%%
```

program → *decl-list*
decl-list → *decl* *decl-list* | *decl*
decl → **id** = *domain* ;
domain → **int** | **string** | *record*
record → **(** *attr-list* **)**
attr-list → *attr* , *attr-list* | *attr*
attr → **id** : *domain*

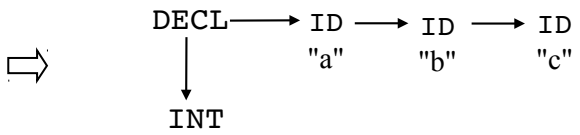
Exercise 20

Codify in *Yacc* the generator of the abstract syntax trees relevant to the language defined by the following BNF:

decl → *type id-list* **;**
type → **int** | **string** | **bool**
id-list → **id** **,** *id-list* | **id**

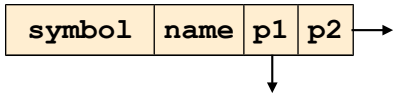
example of mapping:

```
int a, b, c;
```



Note:

- Each node of the abstract tree is structured by the following fields:



`(Symbol)` `symbol` ∈ { `DECL`, `INT`, `STRING`, `BOOL`, `ID` }: symbol relevant to node
`(char *)` `name`: variable name (when `symbol` = `ID`)
`(PNODE)` `p1`, `p2`: pointer to first child and pointer to right brother, respectively

- To create any node, the following function is used:

```
PNODE node(Symbol)
```

where fields `name`, `p1`, and `p2` of the created node are initialized to `NULL`.

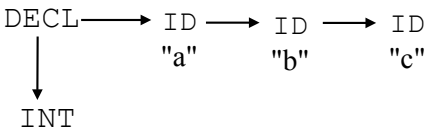
- The pointer of the name of an identifier is inserted by the lexical analyzer (not to be codified) in the following variable of *Yacc*:

```
char *lexval
```

Exercise 20

decl → *type id-list* **;**
type → **int** | **string** | **bool**
id-list → **id** **,** *id-list* | **id**

```
int a, b, c;
```




```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE root=NULL;
char *lexval;
%}
%token INT STRING BOOL ID ERROR
%%
decl : type id-list ';' {root = $$ = node(DECL); $$->p1 = $1; $$->p2 = $2;}
     ;
type : INT {$$ = node(INT);}
     | STRING {$$ = node(STRING);}
     | BOOL {$$ = node(BOOL);}
     ;
id-list : ID {$$ = node(ID); $$->name = lexval;}
          ',' id-list {$$ = $2; $$->p2 = $4;}
        | ID {$$ = node(ID); $$->name = lexval;}
        ;
%%
```

Exercise 21

Given the following BNF, relevant to a language for arithmetic expressions,

expr → *expr* **+** *term* | *expr* **-** *term* | *term*
term → *term* ***** *factor* | *term* **/** *factor* | *factor*
factor → **(** *expr* **)** | **id** | **num**

codify in *Yacc* the code generator of expressions for a P-machine. The p-code shall be generated as strings of characters (define **`YYSTYPE *char`**). The following auxiliary functions are provided:

- `char* stat(char* operator, char* argument)`, generating the string containing the single p-code statement "`operator argument`", where `argument` may be empty;
- `char* append(char* pcode1, char* pcode2)`, generating the string containing the catenation of the two strings of code (separated by a newline) "`pcode1 \n pcode2`".

Here is an example of mapping:

```
a + (b – 3)
```

⇨

```
LOD a
LOD b
LDC 3
SUB
ADI
```

The lexical string relevant to **id** or **num** is assumed to be assigned by the lexical analyzer (not to be codified) in the *Yacc* variable `char* lexval`.

Exercise 21

$expr \rightarrow expr$ **+** $term \mid expr$ **-** $term \mid term$
$term \rightarrow term$ ***** $factor \mid term$ **/** $factor \mid factor$
$factor \rightarrow$ **(** $expr$ **)** | **id** | **num**

```
%{
#include "def.h"
#define YYSTYPE char*
char *lexval;
%}
%token ID NUM ERROR
%%
expr : expr '+' term {$$ = append(append($1, $3), "ADI");}
     | expr '-' term {$$ = append(append($1, $3), "SUB");}
     | term
     ;

term : term '*' factor {$$ = append(append($1, $3), "MUL");}
     | term '/' factor {$$ = append(append($1, $3), "DIV");}
     | factor
     ;

factor : '(' expr ')' {$$ = $2;}
       | ID {$$ = stat("LOD", lexval);}
       | NUM {$$ = stat("LDC", lexval);}
       ;

%%
...
```

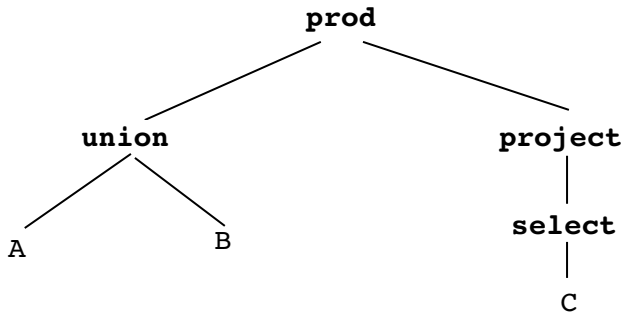
Exercise 22

Given the following BNF, relevant to a Relational Algebra,

program → *expr*
expr → *expr* **union** *term* | *expr* **diff** *term* | *term*
term → *term* **prod** *factor* | *term* **inter** *factor* | *factor*
factor → **project** *factor* | **select** *factor* | **(** *expr* **)** | **id**

codify in *Yacc* the generator of the abstract syntax trees. Here is an example of mapping:

```
(A union B) prod project select C
```



Note:

- Each node of the abstract tree is structured by the following fields:

  `(Typenode)` `type`: type of node { `UNION`, `DIFF`, `PROD`, `INTER`, `PROJECT`, `SELECT`, `ID` }
  `(char *)` `name`: name of identifier when `type=ID`
  `(PNODE)` `p1, p2`: pointer to children

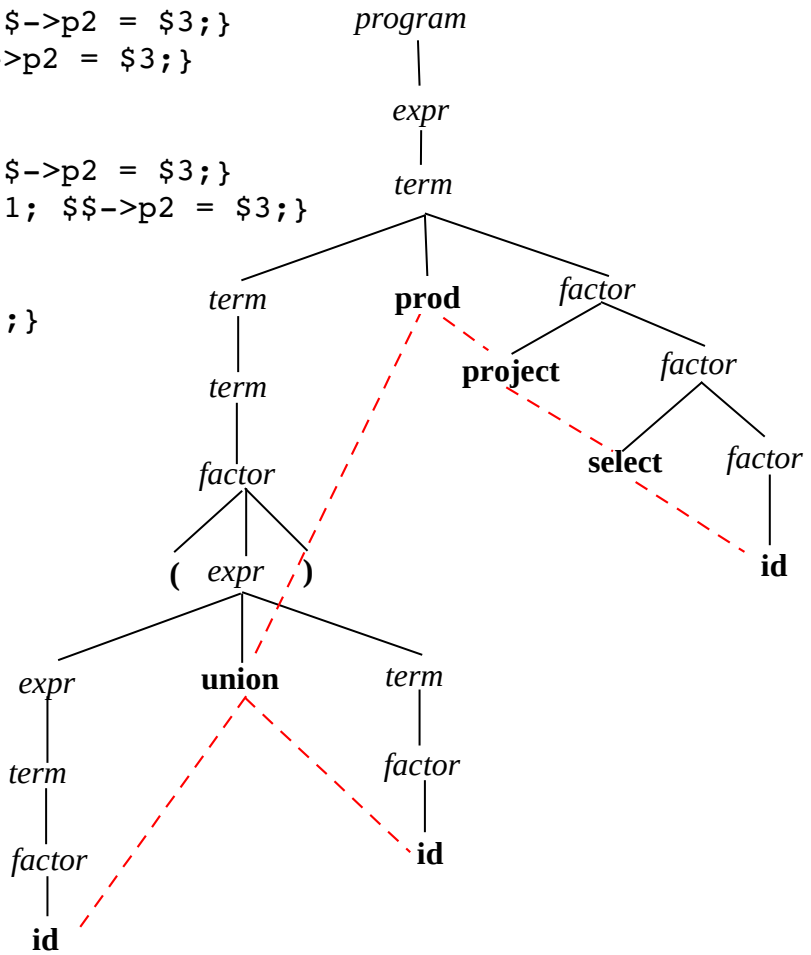
- The following auxiliary functions can be used:

  `PNODE opnode(Typenode op)`: to create an internal node,
  `PNODE idnode()`: to create a leaf node.

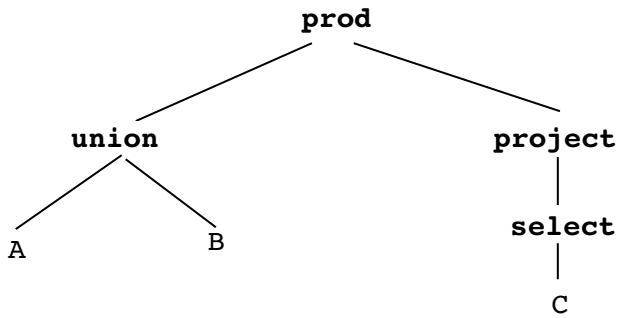
Exercise 22

```
%{
#include "def.h"
#define YYSTYPE PNODE
PNODE root;
%}
%token UNION DIFF PROD INTER PROJECT SELECT ID ERROR
%%
program : expr {root = $1;}

expr : expr UNION term {$$ = opnode(UNION); $$->p1 = $1; $$->p2 = $3;}
     | expr DIFF term {$$ = opnode(DIFF); $$->p1 = $1; $$->p2 = $3;}
     | term
     ;
term : term PROD factor {$$ = opnode(PROD); $$->p1 = $1; $$->p2 = $3;}
     | term SELECT factor {$$ = opnode(SELECT); $$->p1 = $1; $$->p2 = $3;}
     | factor
     ;
factor : PROJECT factor {$$ = opnode(PROJECT); $$->p1 = $2;}
       | SELECT factor {$$ = opnode(SELECT); $$->p1 = $2;}
       | '(' expr ')' {$$ = $2;}
       | ID {$$ = idnode();}
       ;
%%
```

program → *expr*
expr → *expr* **union** *term* | *expr* **diff** *term* | *term*
term → *term* **prod** *factor* | *term* **inter** *factor* | *factor*
factor → **project** *factor* | **select** *factor* | **(** *expr* **)** | **id**



(A union B) prod project select C



Exercise 23

The following BNF defines the **L** procedural language to manipulate integer values by means of arithmetic expressions and conditionals:

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *assign-stat* | *write-stat*
assign-stat → **id** = *expr*
expr → *expr* **+** *term* | *term*
term → *term* ***** *factor* | *factor*
factor → **(** *expr* **)** | *if-expr* | **id** | **num**
if-expr → **if** *expr* **then** *expr* **else** *expr* **end**
write-stat → **write** *expr*

```
a = 0;
b = a + 1;
c = 2;
d = a + (b *c);
write d + 5;
c = if c then a + b else d end;
write if a * b then
        if c + 1 then
          a + b
        else
          c + 12
        end
      else
        d * 2
      end;
a = b + (c * d) + 100;
```

(*example of phrase*)

All variables are of integer type. In conditions, a zero value represents **`false`**, otherwise (if different from zero) represents **`true`**. The **`write`** statement prints its argument on the standard output.
We ask to codify in *Yacc* the interpreter of language **L** based on the following requirements:

- A symbol table is used, where variables and their values are stored, of which the following functions are given:

  `int lookup(char* name)`, returning the value of var `name` if this is stored, otherwise it returns `NIL`;
  `void assign(char* name, int val)`, associating to variable `name` the value `val` (possibly inserting the variable if the latter is not stored).

- *Yacc* symbol `YYSTYPE` is defined by: `struct{char *name; int val}`.
- The lexical value is inserted by the lexical analyzer (not to be codified) into variable `lexval` of type:
  `union{char *name; int val}`.

In case of error, the `error()` function is called, which terminates the interpretation.

Exercise 23

```
%{
#include <stdio.h>
#define YYSTYPE struct{char *name; int val;}
extern union{char *name; int val;} lexval;
int val;
%}
%token ID NUM IF THEN ELSE END WRITE
%%
program : stat-list
        ;

stat-list : stat ';' stat-list
          | stat
          ;

stat : assign-stat
     | write-stat
     ;

assign-stat : ID {$$.name = lexval.name;} '=' expr {assign($2.name, $4.val);}

expr : expr '+' term {$$.val = $1.val + $3.val;}
     | term {$$.val = $1.val;}
     ;

term : term '*' factor {$$.val = $1.val * $3.val;}
     | factor {$$.val = $1.val;}
     ;

factor : '(' expr ')' {$$.val = $2.val;}
       | if-expr {$$.val = $1.val;}
       | ID {if((val = lookup(lexval.name)) == NIL) error(); else $$.val = val;}
       | NUM {$$.val = lexval.val;}
       ;
if-expr : IF expr THEN expr ELSE expr END {$$.val = ($2.val ? $4.val : $6.val);}
        |
        ;
write-stat : WRITE expr {printf("%d\n", $2.val);}
```

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *assign-stat* | *write-stat*
assign-stat → **id** = *expr*
expr → *expr* **+** *term* | *term*
term → *term* ***** *factor* | *factor*
factor → **(** *expr* **)** | *if-expr* | **id** | **num**
if-expr → **if** *expr* **then** *expr* **else** *expr* **end**
write-stat → **write** *expr*

Exercise 24

Codify in *Yacc* the generator of abstract binary trees relevant to the language defined by the following BNF,

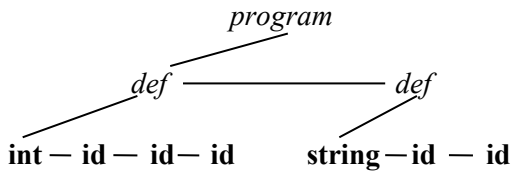
program → *def-list*
def-list → *def* **;** *def-list* | *def* **;**
def → *id-list* **:** *type*
id-list → **id ,** *id-list* | **id**
type → **int** | **string**

```
a, b, c: int;
x, y: string;
```

(*example of phrase*)

based on the following abstract EBNF:

program → { *def* }$^{+}$
def → (**int** | **string**) { **id** }$^{+}$

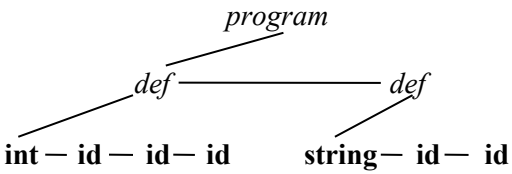


(*abstract tree*)

Exercise 24

program → *def-list*
def-list → *def* **;** *def-list* | *def* **;**
def → *id-list* **:** *type*
id-list → **id ,** *id-list* | **id**
type → **int** | **string**

program → { *def* }$^{+}$
def → (**int** | **string**) { **id** }$^{+}$




```
%{
#include <stdio.h>
#define YYSTYPE PNODE
extern char *lexval;
PNODE root;
%}
%token ID INT STRING
%%
program : def-list {root = $$ = ntn(NPROGRAM) ; $$->p1 = $1;}
        ;

def-list : def ';' def-list {$$ = $1; $1->p2 = $3;}
         | def {$$ = $1;}
         ;

def : id-list ':' type {$$ = ntn(NDEF); $$->p1 = $3; $3->p2 = $1;}
    ;

id-list : ID {$$ = idnode();} ',' id-list {$$ = $2; $2->p2 = $4;}
        | ID {$$ = idnode();}
        ;

type : INT {$$ = keynode(INT);}
     | STRING {$$ = keynode(STRING);}
     ;
%%
```

Exercise 25

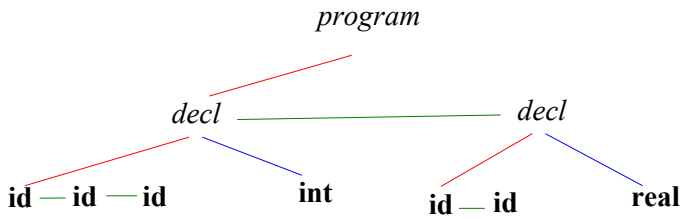
Codify in *Yacc* the generator of the ternary abstract trees of the language defined by the following BNF,

program → *decl-list*
decl-list → *decl* *decl-list* | *decl*
decl → *type id-list*
type → **int** | **real**
id-list → **id** , *id-list* | **id**

based on the following abstract EBNF:

program → { *decl* }$^{+}$
decl → { **id** }$^{+}$ (**int** | **real**)

```
int a, b, c;
real x, y;
```



Exercise 25

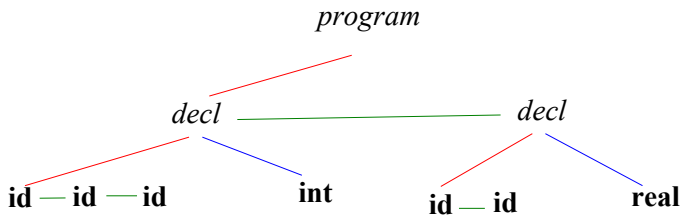
Codify in *Yacc* the generator of the ternary abstract trees of the language defined by the following BNF,

program → *decl-list*
decl-list → *decl* *decl-list* | *decl*
decl → *type id-list*
type → **int** | **real**
id-list → **id** , *id-list* | **id**

based on the following abstract EBNF:

program → { *decl* }$^+$
decl → { **id** }$^+$ (**int** | **real**)

```
int a, b, c;
real x, y;
```




```
%{
#include <stdio.h>
#define YYSTYPE Pnode
extern char *lexval;
Pnode root;
%}
%token ID INT STRING ERROR
%%
program : decl-list {root = $$ = ntn(NPROGRAM); $$->p1 = $1;}
        ;
decl-list : decl decl-list {$$ = $1; $1->p3 = $2;}
          | decl
          ;
decl : type id-list ';' {$$ = ntn(NDECL); $$->p1 = $2; $$->p2 = $1;}
     ;
type : INT {$$ = keynode(INT);}
type : REAL {$$ = keynode(REAL);}
id-list : ID {$$ = idnode()} ',' id-list {$$ = $2; $2->p3 = $4;}
        | ID {$$ = idnode();}
```

type	lexval	p1	p2	p3

Exercise 26

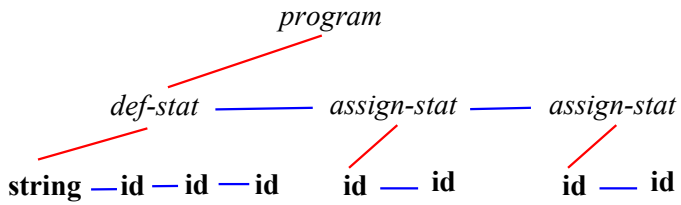
Codify in Yacc the generator of the binary abstract trees relevant to the language defined by the following BNF:

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat*
def-stat → *id-list* **:** *type*
id-list → **id ,** *id-list* | **id**
type → **int** | **string** | **bool**
assign-stat → **id = id**

based on the following abstract EBNF:

program → { *def-stat* | *assign-stat* }$^{+}$
def-stat → (**int** | **string** | **bool**) { **id** }$^{+}$
assign-stat → **id id**

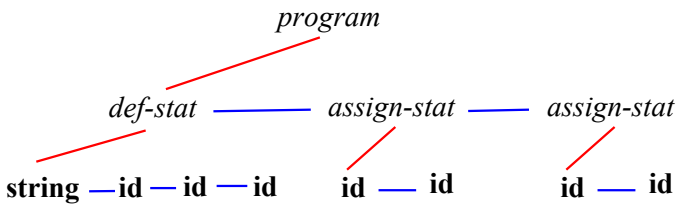
```
x, y, z: string;
x = y;
y = z;
```



Exercise 26

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat*
def-stat → *id-list* **:** *type*
id-list → **id ,** *id-list* | **id**
type → **int** | **string** | **bool**
assign-stat → **id = id**

program → { *def-stat* | *assign-stat* }$^+$
def-stat → **(int | string | bool)** { **id** }$^+$
assign-stat → **id id**




```
%{
#include <stdio.h>
#define YYSTYPE Pnode
extern char *lexval;
Pnode root;
%}
%token ID INT STRING BOOL ERROR
%%
program : stat-list {root = $$ = ntn(NPROGRAM); $$->child = $1;}
        ;
stat-list : stat ';' stat-list {$$ = $1; $1->brother = $3;}
          | stat ';' {$$ = $1;}
          ;
stat : def-stat {$$ = $1;}
     ;
stat : assign-stat {$$ = $1;}
     ;
def-stat : id-list ':' type {$$ = ntn(NDEF_STAT);
                             $$->child = $3;
                             $3->brother = $1;}
     ;
id-list : ID {$$ = idnode()} ',' id-list {$$ = $2; $2->brother = $4;}
        | ID {$$ = idnode();}
type : INT {$$ = keynode(INT);}
type : STRING {$$ = keynode(STRING);}
type : BOOL {$$ = keynode(BOOL);}
assign-stat : id {$$ = idnode()} '=' id {$$ = ntn(NASSIGN_STAT);
                                         $$->child = $2;
                                         $2->brother = idnode();}
```

type	lexval	child	brother

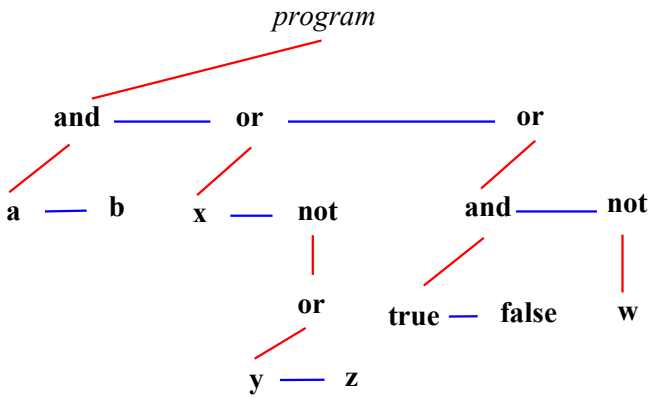
Exercise 27

Codify in Yacc the generator of the binary abstract trees relevant to the language defined by the following BNF:

program → *expr-list*
expr-list → *expr* **;** *expr-list* | *expr* **;**
expr → *expr* **or** *term* | *term*
term → *term* **and** *factor* | *factor*
factor → **not** *factor* | **(** *expr* **)** | **id** | **boolconst**

based on the following example:

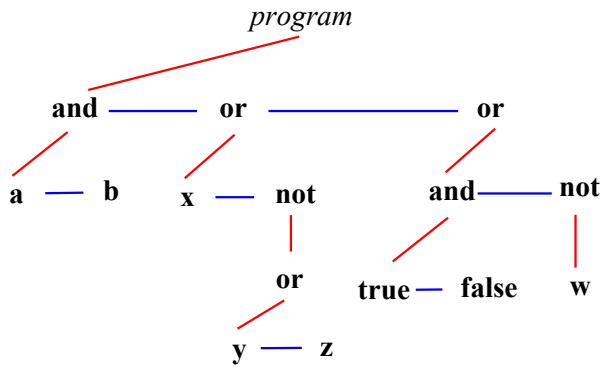
```
a and b;
x or not (y or z);
true and false or not w;
```



Exercise 27

program → *expr-list*
expr-list → *expr* **;** *expr-list* | *expr* **;**
expr → *expr* **or** *term* | *term*
term → *term* **and** *factor* | *factor*
factor → **not** *factor* | **(** *expr* **)** | **id** | **boolconst**

```
a and b;
x or not (y or z);
true and false or not w;
```




```
%{
#include <stdio.h>
#define YYSTYPE Pnode
extern char *lexval;
Pnode root;
%}
%token AND OR NOT ID BOOLCONST ERROR
%%
program : expr-list {root = $$ = ntn(NPROGRAM); $$->child = $1;}
        ;
expr-list : expr ';' expr-list {$$ = $1; $1->brother = $3;}
          | expr ';'
          ;
expr : expr OR term {$$ = opnode(OR); $$->child = $1; $1->brother = $3;}
     | term
     ;
term : term AND factor {$$ = opnode(AND); $$->child = $1; $1->brother = $3;}
     | factor
     ;
factor : NOT factor {$$ = opnode(NOT); $$->child = $2;}
       | '(' expr ')' {$$ = $2;}
       | ID {$$ = idnode();}
       | BOOLCONST {$$ = boolconstnode();}
       ;
%%
```

lexval	child	brother

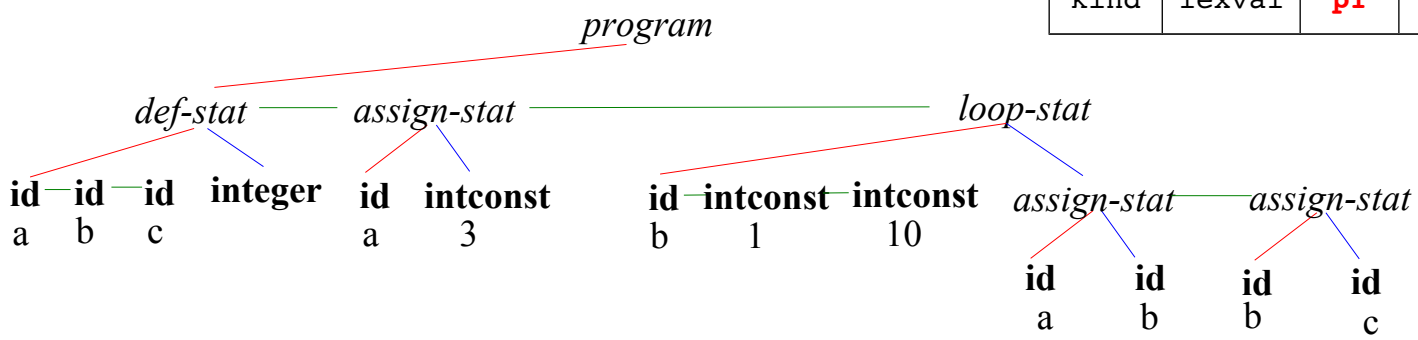
Exercise 28

Codify in *Yacc* the generator of the ternary abstract trees based on the following BNF and structures:

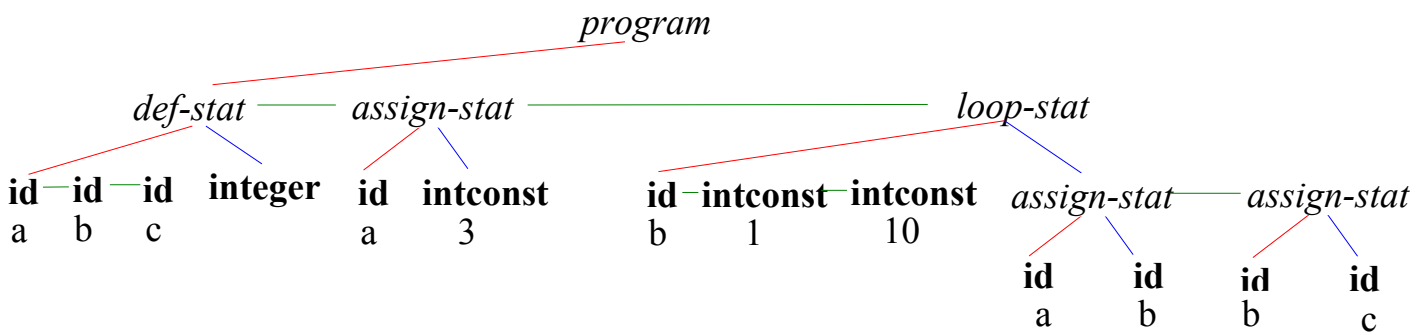
program → *stat-list* | **ε**
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat* | *loop-stat*
def-stat → **def** *id-list* **as** *type*
id-list → **id,** *id-list* | **id**
type → **integer** | **string**
assign-stat → **id** = *const*
const → **intconst** | **strconst**
loop-stat → **for id from intconst to intconst do** *stat-list* **end**

```
def a, b, c as integer;
a = 3;
for b from 1 to 10 do
  a = b;
  b = c;
end;
```

kind	lexval	p1	p2	p3



Exercise 28




```
%{
#define YYSTYPE Pnode
extern Lexval lexval;
Pnode root;
%}
%token DEF AS ID INTEGER STRING INTCONST STRCONST FOR FROM TO DO END
%%
program : stat-list {root = ntn(NPROGRAM); $$->p1 = $1;}
        | {root = ntn(NPROGRAM);}
        ;
stat-list : stat ';' stat-list {$$ = $1; $1->p3 = $3;}
          | stat ';'
          ;
stat : def-stat
     | assign-stat
     | loop-stat
     ;
def-stat : DEF id-list AS type {$$ = ntn(NDEF_STAT); $$->p1 = $2; $$->p2 = $4;}
         ;
id-list : ID {$$ = idnode();} ',' id-list {$$ = $2; $2->p3 = $4;}
        | ID {$$ = idnode();}
        ;
type : INTEGER {$$ = keynode(NINTEGER);}
type : STRING {$$ = keynode(NSTRING);}
assign-stat : ID {$$ = idnode();} '=' const
              {$$ = ntn(NASSIGN_STAT); $$->p1 = $2; $$->p2 = $4;}
            ;
const : INTCONST {$$ = intconstnode();}
      | STRCONST {$$ = strconstnode();}
      ;
loop-stat : FOR ID {$$ = idnode();} FROM INTCONST {$$ = intconstnode();}
            TO INTCONST {$$ = intconstnode();} DO stat-list END
            {$$ = ntn(LOOP_STAT); $$->p1 = $3; $3->p3 = $6; $6->p3 = $9; $$->p2 = $11;}
          ;
```

program → *stat-list* | **ε**
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat* | *loop-stat*
def-stat → **def** *id-list* **as** *type*
id-list → **id,** *id-list* | **id**
type → **integer** | **string**
assign-stat → **id** = *const*
const → **intconst** | **strconst**
loop-stat → **for id from intconst to intconst do** *stat-list* **end**

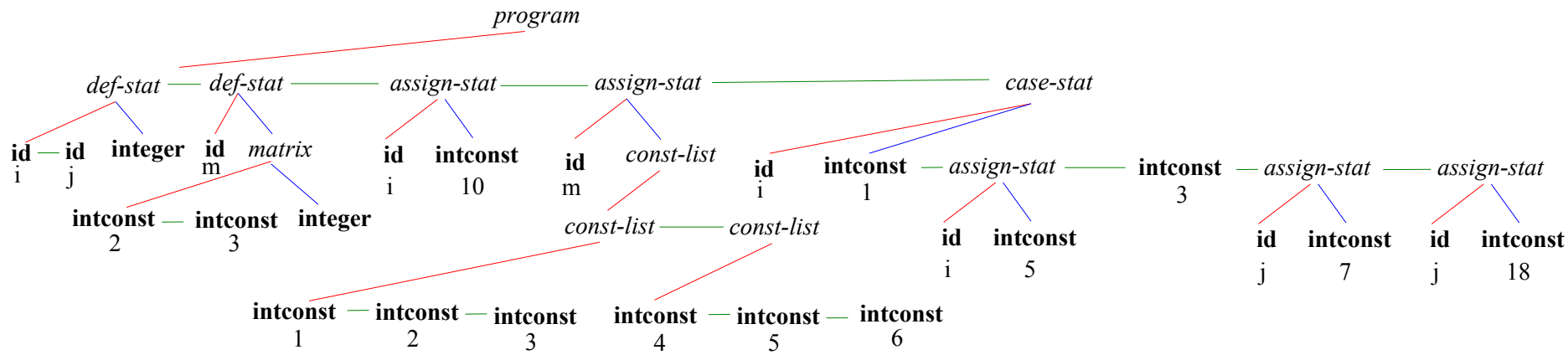
Exercise 29

Codify in *Yacc* the generator of the ternary abstract trees based on the following BNF and structures:

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat* | *case-stat*
def-stat → **var** *id-list* **is** *type*
id-list → **id ,** *id-list* | **id**
type → **integer** | **string** | **matrix (** *intconst-list* **) of** *type*
intconst-list → **intconst ,** *intconst-list* | **intconst**
assign-stat → **id =** *const*
const → **intconst** | **strconst** | *matconst*
matconst → **[** *const-list* **]**
const-list → *const* **,** *const-list* | *const*
case-stat → **case id of** *branch-list opt-default* **end**
branch-list → *branch* **,** *branch-list* | *branch*
branch → *const* **:** *stat* **;**
opt-default → **default :** *stat* **;** | **ε**

```
var i, j is integer;
var m is matrix(2,3) of integer;
i = 10;
m = [[1,2,3],[4,5,6]];
case i of
  1: i = 5;
  3: j = 7;
  default: j = 18;
end;
```

kind	lexval	p1	p2	p3



Exercise 29

```
%{
#define YYSTYPE Pnode
extern Lexval lexval;
Pnode root;
%}
%token VAR IS ID INTEGER STRING MATRIX OF INTCONST STRCONST CASE DEFAULT END
%%
program : stat-list {root = ntn(PROGRAM); $$->p1 = $1;}
        ;
stat-list : stat ';' stat-list {$$ = $1; $1->p3 = $3;}
          | stat ';' {$$ = $1;}
          ;
stat : def-stat
     | assign-stat
     | case-stat
     ;
def-stat : VAR id-list IS type {$$ = ntn(DEF_STAT); $$->p1 = $2; $$->p2 = $4;}
         ;
id-list : ID {$$ = idnode();} ',' id-list {$$ = $2; $2->p3 = $4;}
        | ID {$$ = idnode();}
        ;
type : INTEGER {$$ = keynode(INTEGER);}
     | STRING {$$ = keynode(STRING);}
     | MATRIX '(' intconst-list ')' OF type
       {$$ = ntn(MATRIX); $$->p1 = $3; $$->p2 = $6;}
     ;
intconst-list : INTCONST {$$ = intconstnode();} ',' intconst-list
                {$$ = $2; $2->p3 = $4;}
              | INTCONST {$$ = intconstnode();}
              ;
```

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat* | *case-stat*
def-stat → **var** *id-list* **is** *type*
id-list → **id ,** *id-list* | **id**
type → **integer** | **string** | **matrix (** *intconst-list* **) of** *type*
intconst-list → **intconst ,** *intconst-list* | **intconst**

Exercise 29 (ii)

```
assign-stat : ID {$$ = idnode();} '=' const
              {$$ = ntn(ASSIGN_STAT); $$->p1 = $2; $$->p2 = $4;}

const : INTCONST {$$ = intconstnode();}
      | STRCONST {$$ = strconstnode();}
      | matconst {$$ = ntn(CONST_LIST); $$->p1 = $1;}
      ;
matconst : '[' const-list ']' {$$ = $2;}
         ;
const-list : const ',' const-list {$$ = $1; $1->p3 = $3;}
           | const {$$ = $1;}
           ;
case-stat : CASE ID {$$ = idnode();} OF branch-list opt-default END
            {$$ = ntn(CASE_STAT); $$->p1 = $3; $$->p2 = $5; last($5)->p3 = $6;}
          ;
branch-list : branch ',' branch-list {$$ = $1; $1->p3->p3 = $3;}
        | branch {$$ = $1;}
        ;
branch : const ':' stat {$$ = $1; $1->p3 = $3;}
       ;
opt-default : DEFAULT ':' stat {$$ = $3;}
            | {$$ = NULL;}
            ;
```

assign-stat → **id** = *const*
const → **intconst** | **strconst** | *matconst*
matconst → **[** *const-list* **]**
const-list → *const* **,** *const-list* | *const*
case-stat → **case id of** *branch-list opt-default* **end**
branch-list → *branch* **,** *branch-list* | *branch*
branch → *const* **:** *stat* **;**
opt-default → **default :** *stat* **;** | **ε**

Exercise 30

A language of boolean expressions is defined by the following ambiguous BNF:

program → *expr*
expr → *expr* **and** *expr* | *expr* **or** *expr* | **not** *expr* | **(** *expr* **)** | **true** | **false**

Assuming that **not** has highest precedence and right associativity, **and** has intermediate precedence and left associativity, while **or** has lowest precedence and left associativity, we ask to specify in *Yacc* the interpreter of the language, which is required to print the result of the expression (either "`true`" or "`false`").
Note: The grammar specified in the translation rules of *Yacc* must be equal to the given BNF.

Exercise 30

A language of boolean expressions is defined by the following ambiguous BNF:

program → *expr*
expr → *expr* **and** *expr* | *expr* **or** *expr* | **not** *expr* | **(** *expr* **)** | **true** | **false**

Assuming that **not** has highest precedence and right associativity, **and** has intermediate precedence and left associativity, while **or** has lowest precedence and left associativity, we ask to specify in *Yacc* the interpreter of the language, which is required to print the result of the expression (either "`true`" or "`false`").

```
%{
#include <stdio.h>
%}
%token AND OR NOT TRUE FALSE
%left OR
%left AND
%right NOT
%%
program : expr {printf("%s\n", ($1 ? "true" : "false");}
        ;
expr : expr AND expr {$$ = $1 && $3;}
     | expr OR expr {$$ = $1 || $3;}
     | NOT expr {$$ = !$2;}
     | '(' expr ')' {$$ = $2;}
     | TRUE {SS = 1;}
     | FALSE {$$ = 0;}
     ;
%%
```

Exercise 31

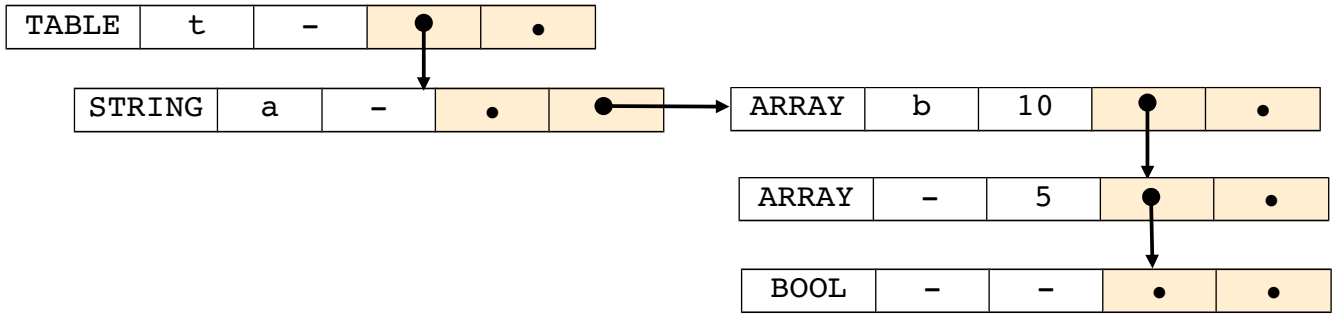
A language for type definitions is specified by the following BNF:

def → **id :** *type*
type → **int** | **string** | **bool** | *table-type* | *array-type*
table-type → **table (** *attr-list* **)**
attr-list → *def* **,** *attr-list* | *def*
array-type → **array [num] of** *type*

We ask to codify in *Yacc* the generator of the corresponding type trees, assuming that each node is qualified by a `domain` ∈ {`INT`, `STRING`, `BOOL`, `TABLE`, `ARRAY`}, a `name` (for variables and attributes), a `dimension` (size of the array), and pointers `child` (first child) and `brother` (right brother). For instance, the following declaration:

```
t: table(a: string, b: array [10] of array [5] of bool)
```

shall generate the following type tree:



Note: If the type of the variable is simple, it shall be represented by a single node.

Exercise 31

```
%{
#include "stdlib.h"
#include "def.h"
#define YYSTYPE PNODE
PNODE root = NULL;
Value lexval; /* union: int ival, char *sval */
%}
%token ID INT STRING BOOL TABLE ARRAY NUM OF
%%

def : ID {$$ = (PNODE) lexval.sval;} ':' type
         {$$ = $4; $$->name = (char *) $2;}
    ;
type : INT {$$ = atomic_node(INT);}
     | STRING {$$ = atomic_node(STRING);}
     | BOOL {$$ = atomic_node(BOOL);}
     | table_type {$$ = $1;}
     | array_type {$$ = $1;}
     ;
table_type : TABLE '(' attr_list ')' {$$ = structured_node(TABLE, $3);}
           ;
attr_list : def ',' attr_list {$1->brother = $3; $$ = $1;}
          | {$$ = $1;}
          ;
array_type : ARRAY '[' NUM {$$ = (PNODE) lexval.ival;} ']' OF type
             {$$ = structured_node(ARRAY, $7); $$->dimension = (int) $4;}
          ;
%%
PNODE *atomic_node(int type)
{  PNODE *p = malloc(sizeof(PNODE));
   p->type = type; p->child = p->brother = NULL; return(p);
}
PNODE *structured_node(int constructor, PNODE elem_type)
{  PNODE *p = atomic_node(constructor);
   p->child = elem_type; return(p);
}
main(){yyparse();}
```