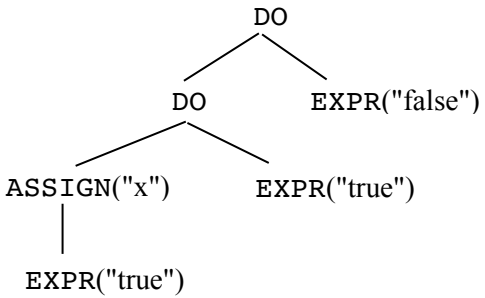
Exercise 1

Given the language defined by the following BNF,

stat → assign-stat | do-stat
assign-stat → **id :=** expr
expr → **true** | **false**
do-stat → **do** stat **while** expr




```
do
  do
    x := true
  while true
while false
```

codify procedure `gencode(PNODE p)`, for intermediate code generation, based on the following requirements:

- The intermediate code is the language of a P-machine with the following set of instructions:

`LDV <id>`	(load value of `<id>`)
`LDA <id>`	(load address of `<id>`)
`LDC <const>`	(load constant `<const>`)
`GOFALSE <label>`	(jump to `<label>` if false)
`GOTRUE <label>`	(jump to `<label>` if true)
`STO`	(store)
`LABEL <label>`	(define label `<label>`)

- Each node of the abstract tree is so structured:

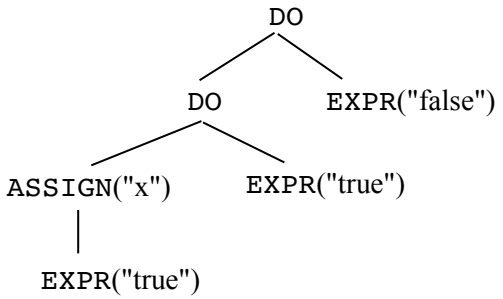
symbol	**val**	**p1**	**p2**

(`Symbol`)	`symbol` { `DO`, `ASSIGN`, `EXPR` }
(`char *`)	`val`: for `ASSIGN` (name of identifier) and `EXPR` (boolean constant)
(`PNODE`)	`p1`, `p2`: pointers to children

- For printing each instruction, a function `pcode(operator, argument)` is used (not to be codified), with two strings of characters as input, where the second string may be empty.

Exercise 1

```
void gencode(PNODE p)
{
    char *lab;

    switch(p->symbol)
    {
    case EXPR: pcode("LDC", p->val);
               break;
    case ASSIGN: pcode("LDA", p->val);
                 gencode(p->p1);
                 pcode("STO", "");
                 break;
    case DO: lab = newlabel();
             pcode("LABEL", lab);
             gencode(p->p1);
             gencode(p->p2);
             pcode("GOTRUE", lab);
             break;
    }
}
```



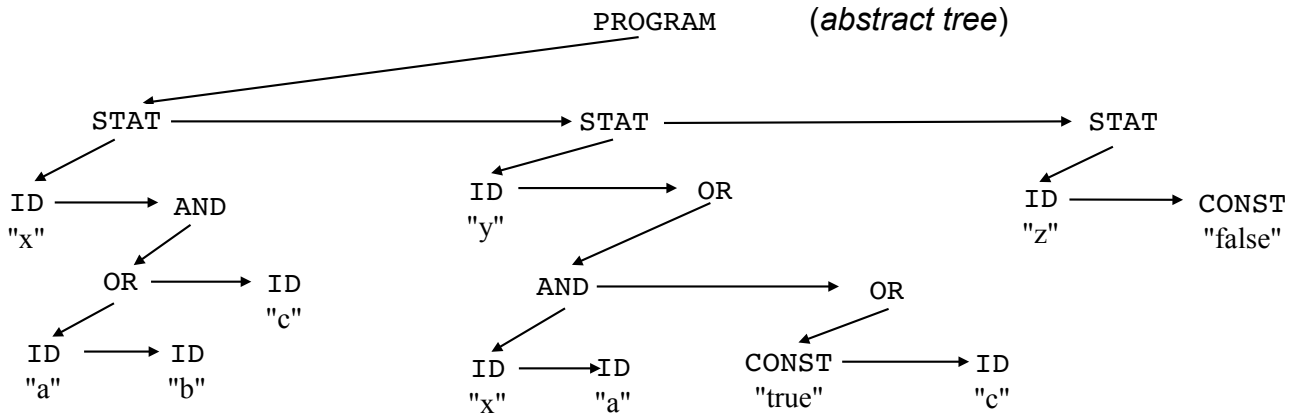

```
LABEL L
⟨stat⟩
⟨expr⟩
GOTRUE L
```

Exercise 2

A language **L** is defined by the following BNF:

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat*
stat → **id :=** *expr*
expr → *expr* **or** *term* | *term*
term → *term* **and** *factor* | *factor*
factor → **id** | **true** | **false** | **(** *expr* **)**

```
x := (a or b) and c;
y := (x and a) or (true or c);
z := false;
```



Exercise 2 (ii)

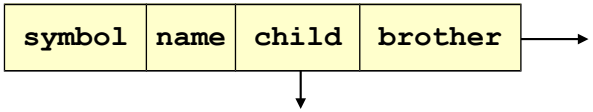
Codify procedure `gencode(PNODE p)`, for p-code intermediate code generation, based on the following requirements:

- In language **L**, evaluation of **`and`** is in short circuit, while **`or`** is fully evaluated.

- The language of the P-machine includes the following set of instructions:

`LOD <id>`	(load value of `<id>` on stack)
`LDA <id>`	(load address of `<id>` on stack)
`LDC <const>`	(load `<const>` on stack, where `<const>` ∈ { `TRUE`, `FALSE` })
`GOFALSE <label>`	(jump to `<label>` if on top of stack is `FALSE`)
`GOTRUE <label>`	(jump to `<label>` if on top of stack is `TRUE`)
`GOTO <label>`	(jump to `<label>`)
`STO`	(store)
`AND`	(logical conjunction)
`OR`	(logical disjunction)
`PLUS`	(addition)
`MINUS`	(difference)
`LAB <label>`	(define label `<label>`)

- Each node of the abstract tree is so structured:

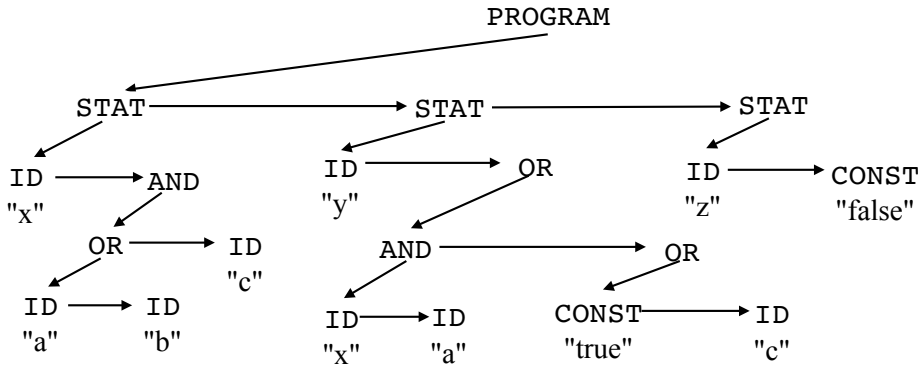


(`Symbol`)	`symbol` { `PROGRAM`, `STAT`, `ID`, `AND`, `OR`, `CONST` }
(`char *`)	`name`: lexical string in case of `ID` or `CONST`
(`PNODE`)	`child`: pointer to first child
(`PNODE`)	`brother`: pointer to right brother

- To print each instruction, a function `print(operator, argument)` (not to be codified), with two strings of characters as input, where `argument` may be empty (when `operator` has no explicit argument).

Exercise 2

```
void gencode(PNODE p)
{
    char *lab1, *lab2;

    switch(p->symbol)
    {
    case PROGRAM: p = p->child;
                  do{
                       gencode(p);
                       p = p->brother;
                  } while(p!= NULL);
                  break;
    case STAT: pcode("LDA", p->child->name);
               gencode(p->child->brother);
               print("STO", "");
               break;
    case OR: gencode(p->child);
             gencode(p->child->brother);
             print("OR","");
             break;
    case AND: lab1 = newlab(); lab2 = newlab();
              gencode(p->child);
              print("GOFALSE", lab1);
              gencode(p->child->brother);
              print("GOTO", lab2);
              print("LAB", lab1);
              print("LDC", "FALSE");
              print("LAB", lab2);
              break;
    case ID: print("LOD", p->name);
             break;
    case CONST: print("LDC", (p->name[0] == 'f' ? "FALSE" : "TRUE"));
                break;
    }
}
```



$\langle expr_1 \rangle$
GOFALSE L1
$\langle expr_2 \rangle$
GOTO L2
LAB L1
LDC FALSE
LAB L2

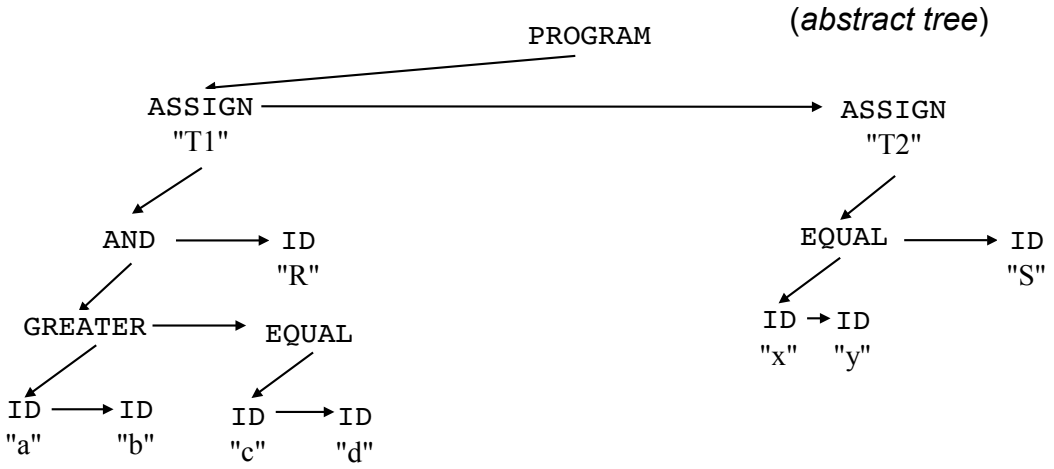
Exercise 3

A language **L** is defined by the following BNF:

program → *assign-list*
assign-list → *assign* **;** *assign-list* | *assign*
assign → **id :=** **select [** *predicate* **]** **id**
predicate → *cond* **and** *predicate* | *cond*
cond → **id** *relop* **id**
relop → = | > | <

```
T1 := select [ a>b and c=d ] R;
T2 := select [ x=y ] S;
```

(*abstract tree*)



Codify procedure `gen(PNODE p)` , for generation of intermediate code for a (high-level) p-machine for the manipulation of tables, based on the following requirements:

Exercise 3 (ii)

- In language **L**, evaluation of **`and`** is <u>not</u> in short circuit.
- The abstract p-machine includes the following set of instructions:

`LDA <id>`	(load address of table `<id>` on stack)
`LDV <id>`	(load instance of table `<id>` on stack)
`LAT <id>`	(load attribute `<id>` on stack when the attribute is referenced within selection predicate)
`STO`	(store)
`AND`	(logical conjunction)
`EQ`	(equality)
`LT`	(less than)
`GT`	(greater than)
`LAB <label>`	(define label `<label>`)
`GOTO <label>`	(jump to `<label>`)

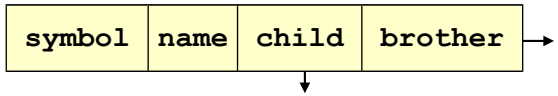
- We assume the following translation mapping for assignment by selection:

R `:=` **`select`** `[` *predicate* `]` *S*

⇩

```
LDA R
LAB label
LDV S
⟨predicate⟩
GOTO label
STO
```

- Each node of the abstract tree is so structured:

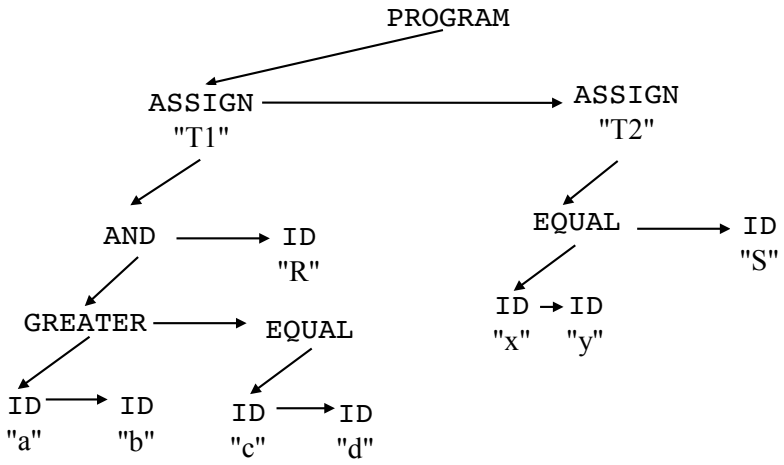


(`Symbol`)	`symbol` { `PROGRAM`, `ASSIGN`, `AND`, `EQUAL`, `GREATER`, `LESS`, `ID` }
(`char *`)	`name`: lexical string in case of `ID` and `ASSIGN` (name of assigned table)
(`PNODE`)	`child`: pointer to first child
(`PNODE`)	`brother`: pointer to right brother

- To print each instruction, a function `print(operator, argument)` (not to be codified), with two strings of characters as input, where `argument` may be empty (when `operator` has no explicit argument) .

Exercise 3

```
void gen(PNODE p)
{
    char *lab;

    switch(p->symbol)
    {
    case PROGRAM: p = p->child;
                  do{
                      gen(p);
                      p = p->brother;
                  } while(p!= NULL);
                  break;
    case ASSIGN: print("LDA", p->name);
                 lab = newlab();
                 print("LAB", lab);
                 print("LDV", p->child->brother->name);
                 gen(p->child);
                 print("GOTO", lab);
                 print("STO", "");
                 break;
    case AND: gen(p->child);
              gen(p->child->brother);
              print("AND","");
              break;
    case EQUAL:
    case GRETER:
    case LESS: gen(p->child);
               gen(p->child->brother);
               print((p->symbol == EQUAL ? "EQ" :
                     (p->symbol == GREATER ? "GT" : "LT")),"");
               break;
    case ID: gen("LAT", p->name); break;
    }
}
```




```
LDA  R
LAB  label
LDV  S
⟨predicate⟩
GOTO label
STO
```

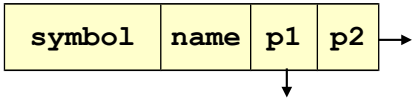
Exercise 4

A language **L** is defined by the following BNF:

program → *stat-list*
stat-list → *stat* *stat-list* | *stat*
stat → **id** := *expr* **;**
expr → *if-expr* | **id** | **boolconst**
if-expr → **(** *expr* **?** *expr* **:** *expr* **)**

```
a := b;
b := false;
c := (a ? b : false);
d := (c ? d : (e ? true : false));
```

a) Outline the abstract tree of the example assuming each node so structured:



(Symbol) `symbol` { `PROGRAM`, `STAT`, `ID`, `IF-EXPR`, `BOOLCONST` }
(char *) `name`: lexical string in case of `ID` and `BOOLCONST`
(PNODE) `p1`: pointer to first child
(PNODE) `p2`: pointer to right brother

b) Codify procedure `gen(PNODE p)`, for intermediate p-code generation, based on the following requirements

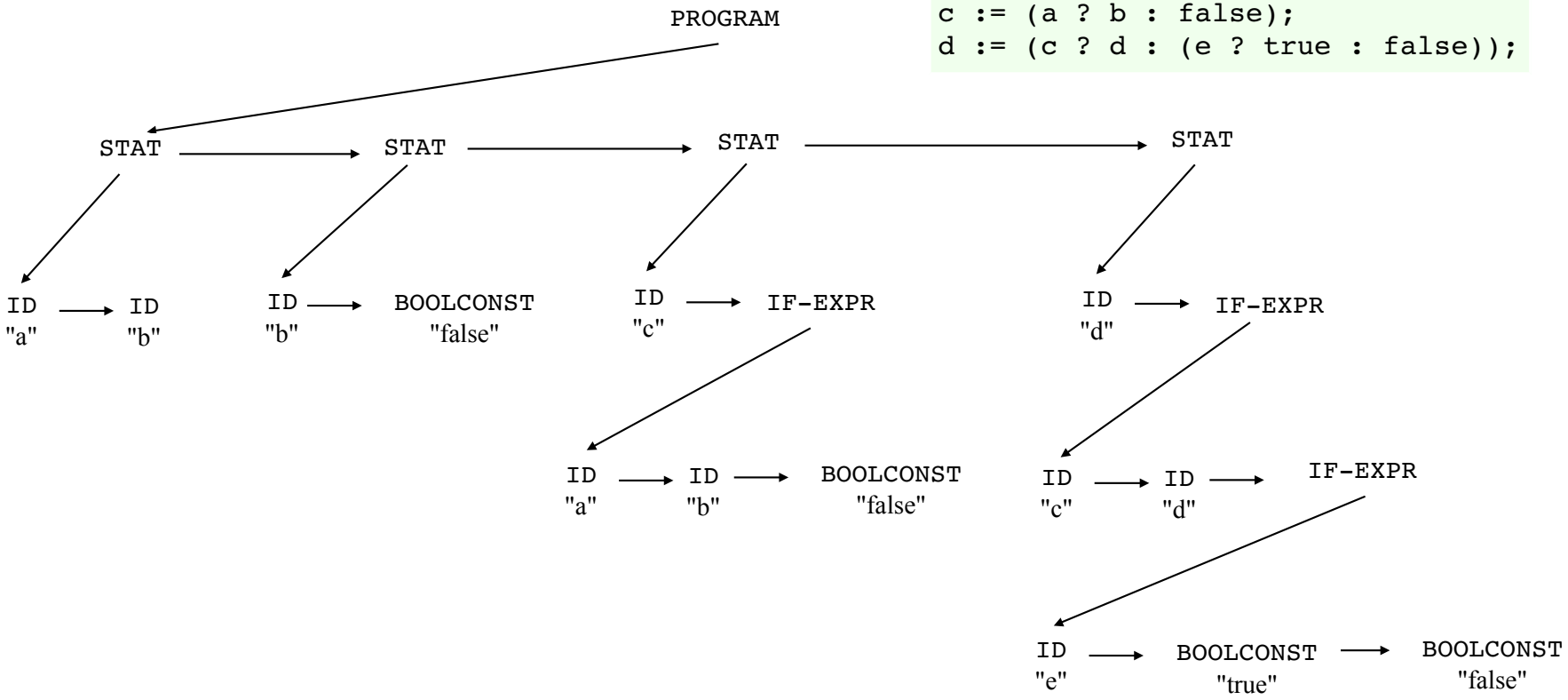
- Evaluation of expressions in *if-expr* is from left to right.
- The language of the P-machine includes the following set of instructions

`LDA <id>`	(load address of `<id>` on stack)
`LDV <id>`	(load value of `<id>` on stack)
`LDC <const>`	(load `<const>` on stack, where `<const>` ∈ { `TRUE`, `FALSE` })
`GOFALSE <label>`	(jump to `<label>` if `FALSE` is on top of stack)
`GOTO <label>`	(jump to `<label>`)
`STO`	(store)
`LAB <label>`	(define label `<label>`)

- To print the code, a function `emit(operator, argument)` is used (not to be codified), with two strings of characters as input, where `argument` may be empty (when `operator` has no explicit argument).

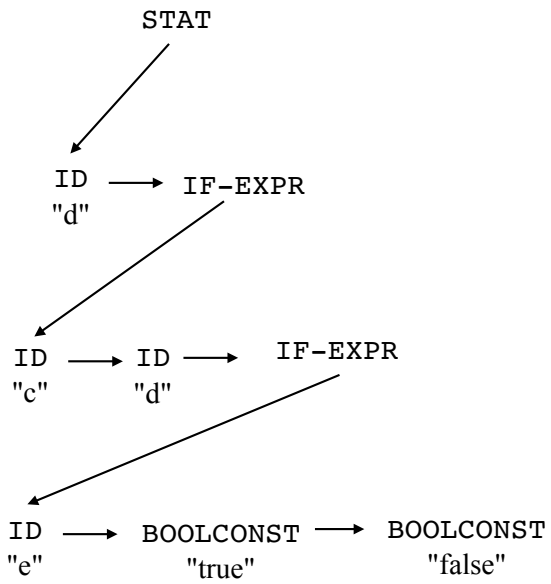
Exercise 4

```
a := b;
b := false;
c := (a ? b : false);
d := (c ? d : (e ? true : false));
```



Exercise 4 (ii)

```
void gen(PNODE p)
{
    char *lab1, *lab2;

    switch(p->symbol)
    {
    case PROGRAM: p = p->p1;
                  do{
                      gen(p);
                      p = p->p2;
                  } while(p != NULL);
                  break;

    case STAT: emit("LDA", p->p1->name);
               gen(p->p1->p2);
               emit("STO", "");
               break;

    case IF-EXPR: lab1 = newlab(); lab2 = newlab();
              gen(p->p1);
              emit("GOFALSE", lab1);
              gen(p->p1->p2);
              emit("GOTO", lab2);
              emit("LAB", lab1);
              gen(p->p1->p2->p2);
              emit("LAB", lab2);
              break;

    case ID: emit("LDV", p->name);
             break;

    case BOOLCONST: emit("LDC", p->name); break;
    }
}
```



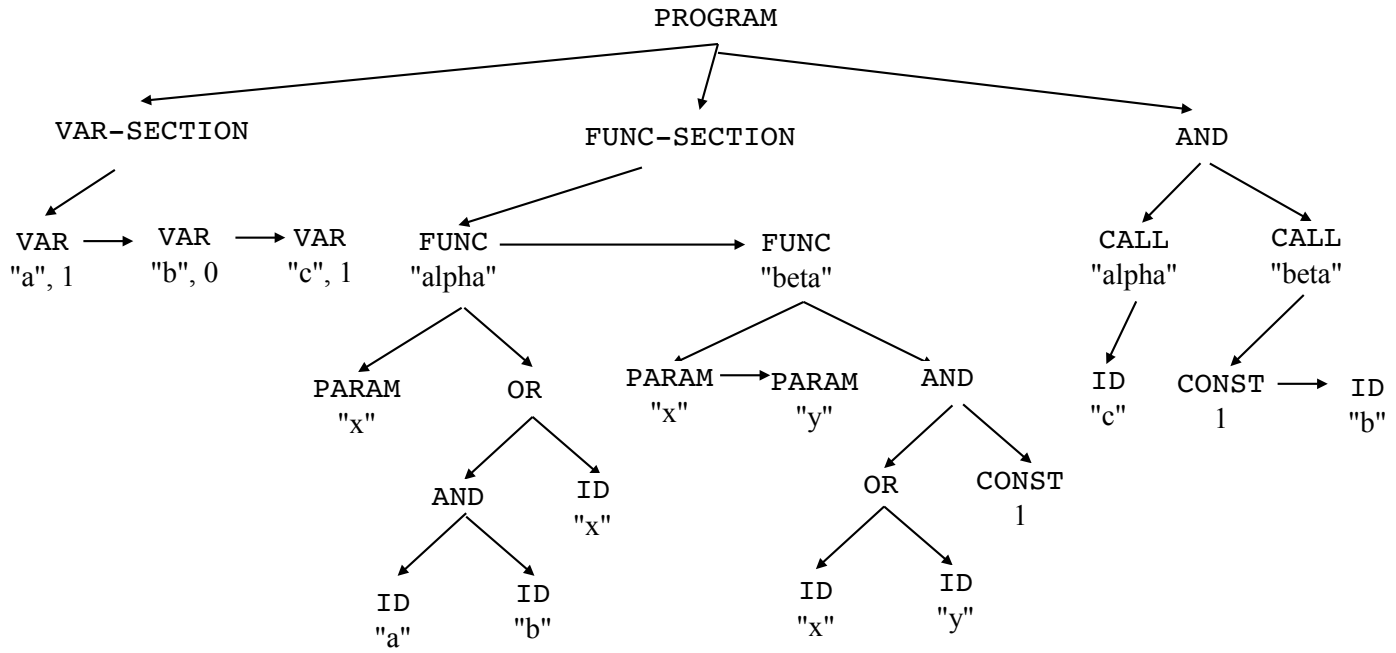
$\langle expr_1 \rangle$
GOFALSE L1
$\langle expr_2 \rangle$
GOTO L2
LAB L1
$\langle expr_3 \rangle$
LAB L2

Exercise 5

A language **L** is defined by the following EBNF:

program → *var-section func-section expr* **;**
var-section → { **id = boolconst** }
func-section → { **id (id** { **, id** } **) =** *expr* **;** }
expr → *expr* **or** *expr* | *expr* **and** *expr* | **(** *expr* **)** | *call* | **id** | **boolconst**
call → **id (** *expr* { **,** *expr* } **)**

```
a=true b=false c=true
function alpha(x) = (a and b) or x;
function beta(x,y) = (x or y) and true;
alpha(c) and beta(true,b);
```



Exercise 5 (ii)

Each node of the abstract tree is so structured:

symbol	name	num	p1	p2	p3

(Symbol) `symbol` ∈ { `PROGRAM`, `VAR-SECTION`, `VAR`, `FUNC-SECTION`, `FUNC`, `PARAM`, `CALL`, `AND`, `OR`, `CONST`, `ID` }.
(`char *`) `name`: lexical string in case of `VAR`, `FUNC`, `PARAM`, `CALL` and `ID`.
(`int`) `num` ∈ { `0`, `1`}: integer value (boolean) in case of `VAR` and `CONST`.
(PNODE) `p1`, `p2`, `p3`: pointers to other nodes.

Assuming that names of local variables (parameters) differ from those of global variables, we ask to codify procedure `gen(PNODE p)`, for intermediate p-code generation, based on the following requirements:

- Evaluation of operators (and actual parameters) is from left to right.
- Operators `AND` and `OR` are evaluated in short circuit.
- The body of a function can reference either local variables (formal parameters) or global variables (variables defined and instantiated in `VAR-SECTION`).
- A symbol table is used, cataloging global variables only, by means of the following function (not to be implemented):

  `bool lookup(char* name)`: indicates whether `name` is in the symbol table (if so, it is a global variable, otherwise it is a parameter).

Exercise 5 (iii)

- The language of the P-machine includes the following set of instructions:

`NEW <id>`	(allocate global variable `<id>` in data memory)
`LDA <id>`	(load address of `<id>` on stack)
`LLD <id>`	(load value of local variable (parameter) `<id>` on stack)
`GLD <id>`	(load value of global variable `<id>` on stack)
`LDC <const>`	(load `<const>` on stack, where `<const>` $\in \{ 0, 1 \}$)
`GOFALSE <label>`	(jump to `<label>` if 0 is on top of stack)
`GOTO <label>`	(jump to `<label>`)
`STO`	(store)
`LAB <label>`	(define label `<label>`)
`ENT <fun>`	(start definition of function `<fun>`)
`RET`	(return of current function)
`MST`	(mark-stack relevant to function call)
`CAL <fun>`	(call function `<fun>`)

- To print each instruction, a function `emit(operator, argument)` is used (not to be codified), with two strings of characters as input, where `argument` may be empty (when `operator` has no explicit argument).

Exercise 5

```
void gen(PNODE p)
{
    char *lab1, *lab2;
    PNODE pt;

    switch(p->symbol)
    {
    case PROGRAM: gen(p->p1); gen(p->p2); gen(p->p3);
                  break;

    case VAR-SECTION:
         FUNC-SECTION: for(pt=p->p1; pt!=NULL; pt=pt->p3)
                         gen(pt);
                       break;

    case VAR: emit("NEW", p->name);
              emit("LDA", p->name);
              emit("LDC", itoa(p->num));
              emit("STO", "");
              break;

    case FUNC: emit("ENT", p->name);
               gen(p->p2);
               emit("RET", "");
               break;

    case AND: lab1 = newlab(); lab2 = newlab();
              gen(p->p1);
              emit("GOFALSE", lab1);
              gen(p->p2);
              emit("GOTO", lab2);
              emit("LAB", lab1);
              emit("LDC", "0");
              emit("LAB", lab2);
              break;
```

⟨*expr*₁⟩
GOFALSE L1
⟨*expr*₂⟩
GOTO L2
LAB L1
LDC 0
LAB L2

```
    case OR: lab1 = newlab(); lab2 = newlab();
             gen(p->p1);
             emit("GOFALSE", lab1);
             emit("LDC", "1");
             emit("GOTO", lab2);
             emit("LAB", lab1);
             gen(p->p2);
             emit("LAB", lab2);
             break;

    case CALL: emit("MST", "");
               for(pt=p->p1; pt!=NULL; pt=pt->p3)
                 gen(pt);
               emit("CAL", p->name);
               break;

    case ID: emit((lookup(p->name) ? "GLD" : "LLD"),
                  p->name);
             break;

    case CONST: emit("LDC", itoa(p->num));
                break;

    }
}
```

⟨*expr*₁⟩
GOFALSE L1
LDC 1
GOTO L2
LAB L1
⟨*expr*₂⟩
LAB L2

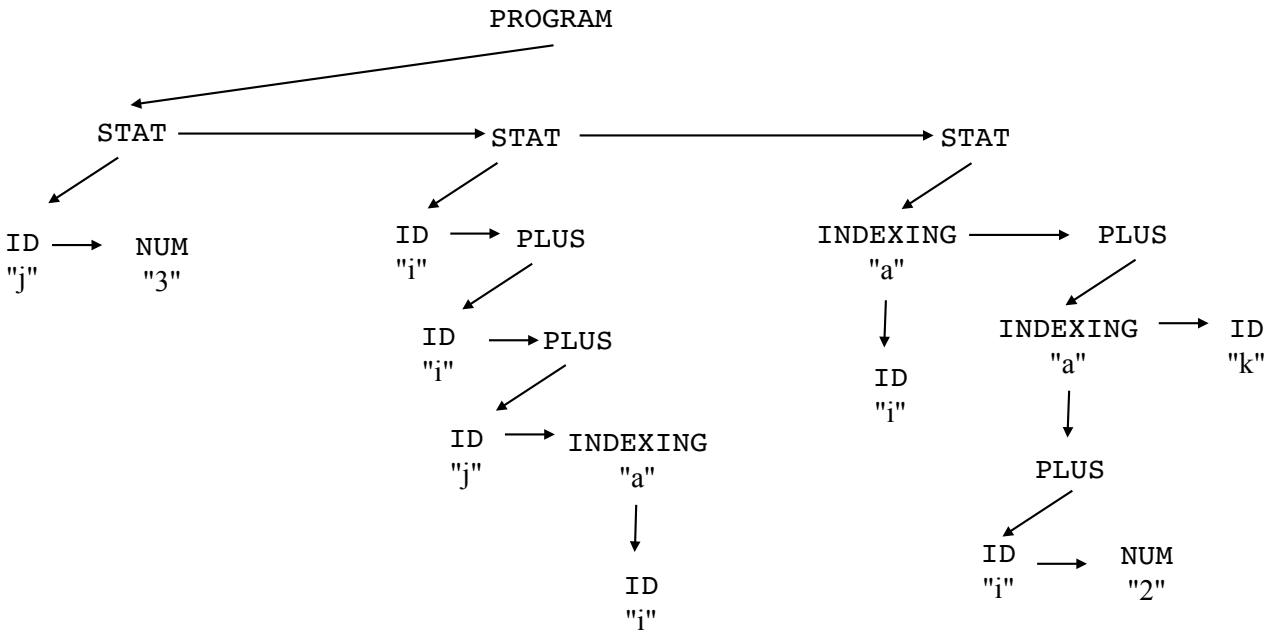
Exercise 6

A lanuguage **L** is defined by the following EBNF:

program → *stat-list*
stat-list → { *target* **=** *expr* **;** }$^{+}$
target → **id** | *indexing*
indexing → **id [** *expr* **]**
expr → *expr* **+** *expr* | *indexing* | **num** | **id**

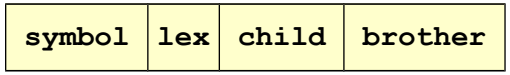
```
j = 3;
i = i + j + a[i];
a[i] = a[i+2] + k;
```

Here is the abstract tree relevant to the example phrase:



Exercise 6 (ii)

Each node of the abstract tree is so structured:



(Symbol) `symbol` ∈ { `PROGRAM`, `STAT`, `ID`, `INDEXING`, `PLUS`, `NUM` }.
(`char *`) `lex`: lexical string in case of `ID`, `INDEXING` and `NUM`.
(`PNODE`) `child`: pointer to first child.
(`PNODE`) `brother`: pointer to right brother.

Codify the (recursive) function for the generation of intermediate p-code, based on the following requirements:

- Evaluation of addition is from right to left.
- The array index is an integer.
- The size of integers in the P-machine is 4 bytes.
- The language of the P-machine includes the following set of instructions:

`LDA <id>`	(load address of variable `<id>` on stack)
`LOD <id>`	(load value of variable `<id>` on stack)
`LDC <const>`	(load `<const>` on stack)
`IXA <scale>`	(indexed address, where `<scale>` is the scale factor)
`IND <offset>`	(indirect load, where `<offset>` is the offset)
`ADI`	(addition)
`STO`	(store)

- To print each instruction, a function `emit(operator, argument)` is used (not to be codified), with two strings of characters as input, where `argument` may be empty (when `operator` has no explicit argument).

Exercise 6

```
void gen(PNODE p, Bool isAddr)
{   PNODE pt;

    switch(p->symbol)
    {
    case PROGRAM: pt = p->child;
                  do{gen(pt, isAddr); pt=pt->brother;}
                  while(pt!=NULL);
                  break;

    case STAT: gen(p->child, TRUE);
               gen(p->child->brother, FALSE);
               emit("STO", "");
               break;

    case PLUS: gen(p->child->brother, FALSE);
               gen(p->child, FALSE);
               emit("ADI", "");
               break;

    case NUM: emit("LDC", p->lex);
              break;

    case ID: emit((isAddr ? "LDA" : "LOD"), p->lex);
             break;

    case INDEXING: emit("LDA", p->lex);
                   gen(p->child, FALSE);
                   emit("IXA", "4");
                   if(!isAddr)
                     emit("IND", "0");
                   break;
    }
}
```

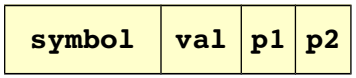
Exercise 7

A language for arithmetic expressions is given, based on the following BNF:

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

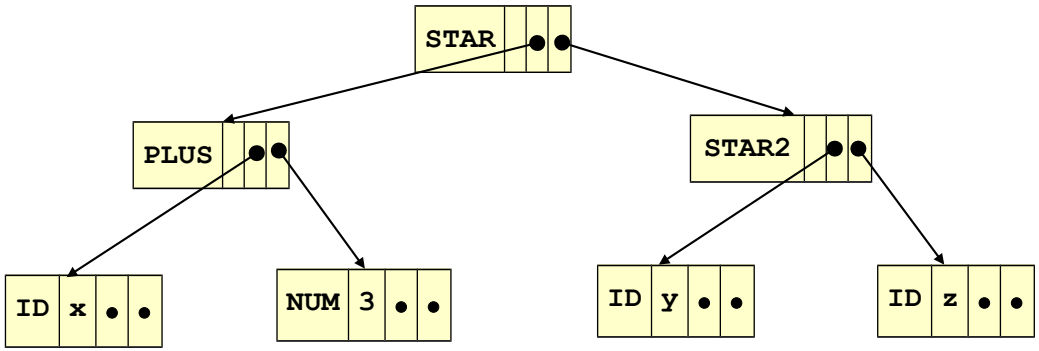
```
(x + 3) * (y ** z)
```

Each node of the abstract tree is so structured:



(Symbol) `symbol` ∈ { `PLUS`, `MINUS`, `STAR`, `SLASH`, `STAR2`, `ID`, `NUM` }.
(char *) `val`: lexical string in case of `ID` and `NUM`.
(PNODE) `p1`: pointer to first child.
(PNODE) `p2`: pointer to second child.

Here is the abstract tree relevant to the example phrase:



Exercise 7 (ii)

Codify the (recursive) function for the generation of intermediate P-code, based on the following requirements:

- Evaluation of addition, difference, product and division is from left to right.
- Evaluation of exponentiation (**) is from right to left.
- Evaluation of product is in short circuit: if the first argument is 0 then the result is 0.
- Evaluation of exponentiation is in short circuit: if the exponent is 0 then the result is 1.
- The language of the P-machine includes the following set of instructions:

`LOD <id>`	(load value of variable `<id>` on stack)
`LDC <const>`	(load `<const>` on stack)
`GOZERO <label>`	(jump to `<label>` if 0 is on top of stack; delete top of stack only if it is 0);
`GOTO <label>`	(jump to `<label>`);
`ADD`	(addition)
`SUB`	(difference)
`MUL`	(product)
`DIV`	(division)
`POWER`	(exponentiation)
`LAB <label>`	(define label `<label>`)

- To print each instruction, a function `emit(operator, argument)` is used (not to be codified), with two strings of characters as input, where `argument` may be empty (when `operator` has no explicit argument).

Exercise 7

```
void gen(PNODE p)
{
    char *lab1, *lab2;

    switch(p->symbol)
    {
    case STAR: lab1 = newlab(); lab2 = newlab();
               gen(p->p1);
               emit("GOZERO", lab1);
               gen(p->p2);
               emit("MUL", "");
               emit("GOTO", lab2);
               emit("LAB", lab1);
               emit("LDC", "0");
               emit("LAB", lab2);
               break;
```

⟨*expr*$_1$⟩
GOZERO L1
⟨*expr*$_2$⟩
MUL
GOTO L2
LAB L1
LDC 0
LAB L2

```
    case STAR2: lab1 = newlab(); lab2 = newlab();
                gen(p->p2);
                emit("GOZERO", lab1);
                gen(p->p1);
                emit("POWER", "");
                emit("GOTO", lab2);
                emit("LAB", lab1);
                emit("LDC", "1");
                emit("LAB", lab2);
                break;
```

⟨*expr*$_2$⟩
GOZERO L1
⟨*expr*$_1$⟩
POWER
GOTO L2
LAB L1
LDC 1
LAB L2

```
    case PLUS: gen(p->p1); gen(p->p2);
               emit("ADD", "");
               break;

    case MINUS: gen(p->p1); gen(p->p2);
                emit("SUB", "");
                break;

    case SLASH: gen(p->p1); gen(p->p2);
                emit("DIV", "");
                break;

    case ID: emit("LOD", p->val);
             break;

    case NUM: emit("LDC", p->val);
              break;

    }
}
```

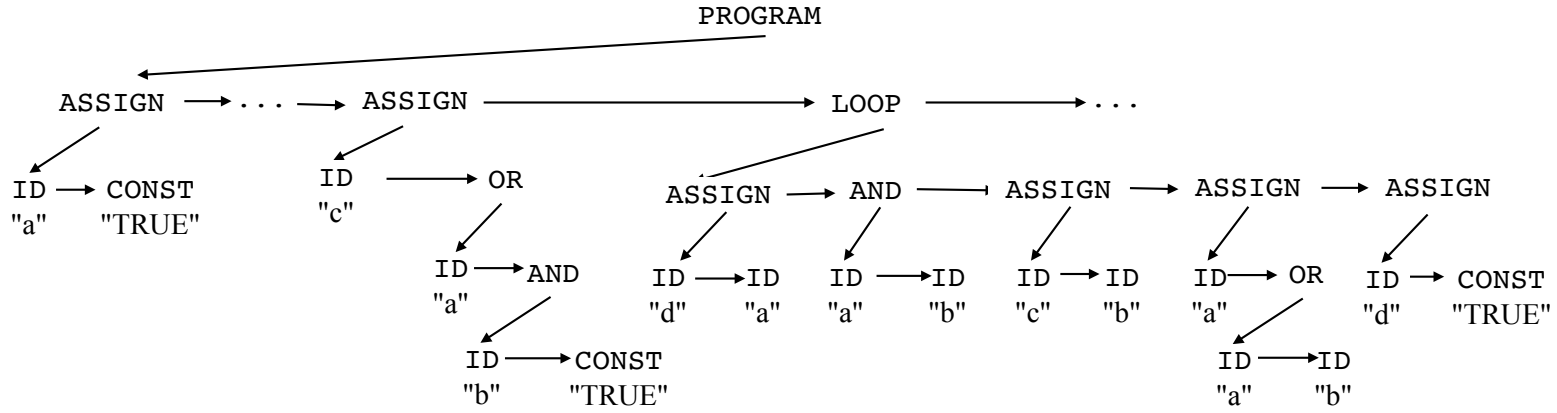
Exercise 8

A language for the manipulation of booleans is given, defined by the following EBNF:

program → { *stat* **;**}$^+$
stat → *assign* | *loop*
assign → **id** **=** *expr*
expr → *expr* **or** *expr* |
expr **and** *expr* |
(*expr* **)** | **id** | **const**
loop → **for (** *stat* **;** *expr* **;** *stat* **)** { *stat* **;**}$^+$ **end**

```
a = true;
b = false;
c = a or (b and true);
for(d = a; a and b; c = b)
  a = a or b;
  d = true;
end;
b = a and b;
```

Here is a fragment of the abstract tree relevant to the example:



Each node of the abstract tree is so structured:

symbol	**lexval**	**child**	**brother**

(`Symbol`) `symbol` ∈ { `PROGRAM`, `ASSIGN`, `LOOP`, `AND`, `OR`, `ID`, `CONST` }.
(`char *`) `lexval`: lexical string in case of `ID` and `CONST`.
(`PNODE`) `child`: pointer to first child.
(`PNODE`) `brother`: pointer to right brother.

Exercise 8 (ii)

Codify the (recursive) function of intermediate p-code generation, based on the following requirements:

- Evaluation of logical operators is in short circuit.
- The semantics of loop is like that of **`for`** statement in the C language.
- The language of the P-machine includes the following set of instructions:

`LDA <id>`	(load address of boolean variable `<id>` on stack)
`LOD <id>`	(load value of boolean variable `<id>` on stack)
`LDC <const>`	(load boolean constant `<const>`, `TRUE` or `FALSE`, on stack)
`AND`	(logical conjunction)
`OR`	(logical disjunction)
`GOFALSE <label>`	(conditional jump)
`GOTO <label>`	(unconditional jump)
`LABEL <label>`	(define label)
`STO`	(store)

Exercise 8

```
void gen(Pnode p)
{
    Pnode pt;
    char *lab1, *lab2;

    switch(p->symbol)
    {
    case PROGRAM: pt = p->child;
                  do{gen(pt); pt = pt->brother;}
                  while(pt != NULL);
                  break;

    case ASSIGN: emit("LDA", p->child->lexval);
                 gen(p->child->brother);
                 emit("STO", "");
                 break;

    case LOOP: gen(p->child);
               lab1 = newlab(); lab2 = newlab();
               emit("LABEL", lab1);
               gen(p->child->brother);
               emit("GOFALSE", lab2);
               pt = p->child->brother->brother->brother;
               do
               { gen(pt);
                 pt = pt->brother;
               } while(pt != NULL);
               gen(p->child->brother->brother);
               emit("GOTO", lab1);
               emit("LABEL", lab2);
               break;
```

```
for(stat1; expr; stat2)
  body
end
```

```
⟨stat1⟩
LABEL L1
⟨expr⟩
GOFALSE L2
⟨body⟩
⟨stat2⟩
GOTO L1
LABEL L2
```

Exercise 8 (ii)

```
    case AND: lab1 = newlab(); lab2 = newlab();
              gen(p->child);
              emit("GOFALSE", lab1);
              gen(p->child->brother);
              emit("GOTO", lab2);
              emit("LABEL", lab1);
              emit("LDC", "FALSE");
              emit("LABEL", lab2);
              break;

    case OR:  lab1 = newlab(); lab2 = newlab();
              gen(p->child);
              emit("GOFALSE", lab1);
              emit("LDC", "TRUE");
              emit("GOTO", lab2);
              emit("LABEL", lab1);
              gen(p->child->brother);
              emit("LABEL", lab2);
              break;

    case CONST: emit("LDC", p->lexval);
                break;

    case ID:  emit("LOD", p->lexval);
              break;
    }
}
```

$\langle expr_1 \rangle$
GOFALSE L1
$\langle expr_2 \rangle$
GOTO L2
LABEL L1
LDC FALSE
LABEL L2

$\langle expr_1 \rangle$
GOFALSE L1
LDC TRUE
GOTO L2
LABEL L1
$\langle expr_2 \rangle$
LABEL L2

Exercise 9

A language is given, based on the following EBNF:

program → { stat **;**}$^+$
stat → *assign-stat* | *if-stat* | *while-stat*
assign-stat → **id =** *expr*
expr → *expr* (**+** | **–** | ***** | **/**) *expr* | **id** | **intconst**
if-stat → **if** *predicate* **then** { stat **;**}$^+$ **end**
while-stat → **while** *predicate* **do** { stat **;**}$^+$ **end**
predicate → *predicate* (**and** | **or**) *predicate* | **id** | **boolconst**

```
a = b + c - d;
if x and y then
  while z do
    d = e + 10;
    f = d * c;
  end;
end;
```

We ask to:

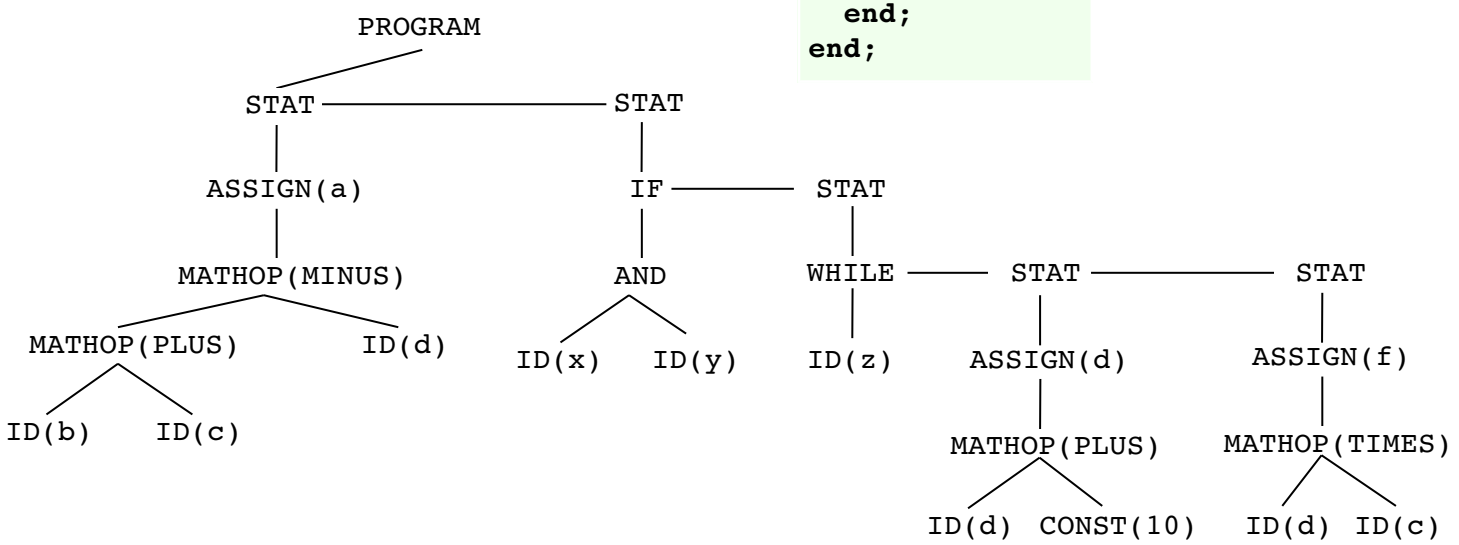
a) Define the node structure for an abstract syntax tree;
b) Based on the previous point, outline the abstract tree relevant to the example phrase;
c) Codify a function for intermediate p-code generation, based on the following requirements:
   - Operands are evaluated from left to right;
   - Evaluation of logical operators is in short circuit;
   - The language of the P-machine includes the following set of instructions:

`LDA <id>`	(load address of variable `<id>` on stack)
`LOD <id>`	(load value of variable `<id>` on stack)
`LDC <const>`	(load constant `<const>` on stack)
`PLUS`	(addition)
`MINUS`	(difference)
`TIMES`	(product)
`DIV`	(division)
`GOFALSE <label>`	(jump conditioned to value `FALSE` on stack)
`GOTO <label>`	(unconditional jump)
`LABEL <label>`	(define label)
`STO`	(store)

Exercise 9

```
a = b + c - d;
if x and y then
  while z do
    d = e + 10;
    f = d * c;
  end;
end;
```

symbol	value	p1	p2



Exercise 9 (ii)

```
void gen(Pnode p)
{
    Pnode pt;
    char *lab1, *lab2;

    switch(p->symbol)
    {
    case PROGRAM: pt = p->p1;
                  do{gen(pt->p1); pt = pt->p2;}
                  while(pt != NULL);
                  break;

    case ASSIGN: emit("LDA", p->value.sval);
                 gen(p->p1);
                 emit("STO", "");
                 break;

    case IF: gen(p->p1);
             lab1 = newlab();
             emit("GOFALSE", lab1);
             for(pt = p->p2; pt !=NULL; pt = pt->p2)
               gen(pt->p1);
             emit("LABEL", lab1);
             break;

    case WHILE: lab1 = newlab(); lab2 = newlab();
                emit("LABEL", lab1);
                gen(p->p1);
                emit("GOFALSE", lab2);
                for(pt = p->p2; pt !=NULL; pt = pt->p2)
                  gen(pt->p1);
                emit("GOTO", lab1);
                emit("LABEL", lab2);
                break;
```

⟨*predicate*⟩
GOFALSE L1
⟨*statements*⟩
LABEL L1

LABEL L1
⟨*predicate*⟩
GOFALSE L2
⟨*statements*⟩
GOTO L1
LABEL L2

Exercise 9 (iii)

```
    case AND: lab1 = newlab(); lab2 = newlab();
              gen(p->p1);
              emit("GOFALSE", lab1);
              gen(p->p2);
              emit("GOTO", lab2);
              emit("LABEL", lab1);
              emit("LDC", "0");
              emit("LABEL", lab2);
              break;

    case OR: lab1 = newlab(); lab2 = newlab();
             gen(p->p1);
             emit("GOFALSE", lab1);
             emit("LDC", "1");
             emit("GOTO", lab2);
             emit("LABEL", lab1);
             gen(p->p2);
             emit("LABEL", lab2);
             break;

    case MATHOP: gen(p->p1); gen(p->p2);
                 emit(p->value.sval);   /* operator stored in value.sval */
                 break;

    case CONST: emit("LDC", p->value.ival);
                break;

    case ID: emit("LOD", p->value.sval);
             break;
    }
}
```

⟨*predicate*$_1$⟩
GOFALSE L1
⟨*predicate*$_2$⟩
GOTO L2
LABEL L1
LDC 0
LABEL L2

⟨*predicate*$_1$⟩
GOFALSE L1
LDC 1
GOTO L2
LABEL L1
⟨*predicate*$_2$⟩
LABEL L2

Exercise 10

Given the language defined by the following BNF,

program → *stat-list*
stat-list → *stat* *stat-list* | *stat*
stat → *declaration* | *assignment* | *loop*
declaration → *type id-list*
type → **int** | **real** | **bool**
id-list → **id** , *id-list* | **id**
assignment → **id** = *expr*
expr → *expr* + *expr* | *expr* == *expr* | **id** | **intconst** | **realconst** | **boolconst**
loop → **while** *expr* **do** *stat*

specify the synthesis of P-code by means of an attribute grammar, where the only attribute is code, based on the following assumptions:

- Within nodes of the syntax tree, terminals **id**, **intconst**, **realconst**, and **boolconst** are associated with the corresponding lexical string `lexeme`;
- To generate the code, the following auxiliary functions are used:
  `Code makecode(String operation)`: builds the instruction operation (without arguments);
  `Code makecode2(String operation, String argument)`: builds the instruction operation applied to `argument`;
  `Code catcode(Code code1, Code code2, ...)`: builds the concatenation of `code1`, `code2`, ...;
  `String getoid(String name)`: returns the object identifier of variable `name`;
  `String newlab()`: returns a new label;
- The P-machine includes the following set of instructions:
  - `ENT`: enter program (the first instruction of the generated code);
  - `NEW` *oid*: create variable identified by object identifier *oid*;
  - `LDA` *oid*: load address of variable identified by object identifier *oid*;
  - `LOD` *oid*: load value of variable identified by object identifier *oid*;
  - `LDI` *value*: load integer *value*; `LDR` *value*: load real *value*;
  - `LDB` *value*: load boolean *value*;
  - `ADD`: addition;
  - `EQU`: equality;
  - `STO`: store;
  - `LAB` *label*: create *label*;
  - `GOF` *label*: conditional jump (to false); `GOT` *label*: unconditional jump;
  - `HLT`: halts program (the last instruction of the generated code).

Exercise 10

Production	*Semantic rules*
program → *stat-list*	*program*.code = `catcode`(`makecode`("ENT"), *stat-list*.code, `makecode`("HLT"));
stat-list$_1$ → *stat* *stat-list*$_2$	*stat-list*$_1$.code = `catcode`(*stat*.code, *stat-list*$_2$.code);
stat-list → *stat*	*stat-list*.code = *stat*.code;
stat → *declaration*	*stat*.code = *declaration*.code;
stat → *assignment*	*stat*.code = *assignment*.code;
stat → *loop*	*stat*.code = *loop*.code;
declaration → *type id-list*	*declaration*.code = *id-list*.code;
type → **int**	
type → **real**	
type → **bool**	
id-list$_1$ → **id** , *id-list*$_2$	*id-list*$_1$.code = `catcode`(`makecode2`("NEW", `getoid`(**id**.lexeme)), *id-list*$_2$.code);
id-list → **id**	*id-list*.code = `makecode2`("NEW", `getoid`(**id**.lexeme));
assignment → **id** = *expr*	*assignment*.code = `catcode`(`makecode2`("LDA", `getoid`(**id**.lexeme)), *expr*.code, `makecode`("STO"));
expr$_1$ → *expr*$_2$ + *expr*$_3$	*expr*$_1$.code = `catcode`(*expr*$_2$.code, *expr*$_3$.code, `makecode`("ADD"));
expr$_1$ → *expr*$_2$ == *expr*$_3$	*expr*$_1$.code = `catcode`(*expr*$_2$.code, *expr*$_3$.code, `makecode`("EQU"));
expr → **id**	*expr*.code = `makecode2`("LOD", `getoid`(**id**.lexeme));
expr → **intconst**	*expr*.code = `makecode2`("LDI", **intconst**.lexeme);
expr → **realconst**	*expr*.code = `makecode2`("LDR", **realconst**.lexeme);
expr → **boolconst**	*expr*.code = `makecode2`("LDB", **boolconst**.lexeme);
loop → **while** *expr* **do** *stat*	`lab1` = `newlab`(); `lab2` = `newlab`(); *loop*.code = `catcode`(`makecode2`("LAB", `lab1`), *expr*.code, `makecode2`("GOF", `lab2`), *stat*.code, `makecode2`("GOT", `lab1`), `makecode2`("LAB", `lab2`));

Exercise 11

A language is defined by the following EBNF:

program → { *stat* }$^+$
stat → *assign* | *loop* | **break**
assign → (*indexpr* | **id**) = *expr*
indexpr → **id [** *expr* **]**
expr → *indexpr* | **num** | **id**
loop → **while** *expr* **do** { *stat* }$^+$

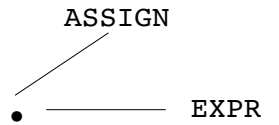
Assuming that the abstract tree is binary (pointers: `child`, `brother`) and irrelevant syntax sugar is not stored, we ask to codify a procedure for P-code generation based on the following requirements:

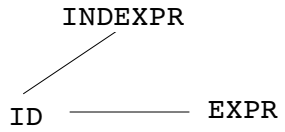
a) the **break** statement breaks the loop in which it appears;
b) the language of the P-machine includes the following set of instructions:
   - `LDA` *id*: load address of variable *id*;
   - `LOD` *id*: load value of variable *id*;
   - `LDI` *value*: load integer *value*;
   - `LAB` *label*: create *label*;
   - `GOF` *label:* conditional jump (to false);
   - `GOT` *label*: unconditional jump;
   - `IND` *offset*: indirect load;
   - `IXA` *scale*: indexed address;
   - `STO`: store.
c) the size of array elements is 4;
d) the (overloaded) auxiliary function `emit(string operator [, string operand])` is used to print an instruction of the P-machine.

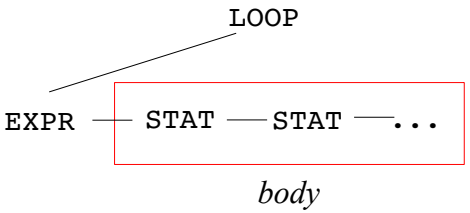
Exercise 11

```
void gencode(PNODE root, int isAddr, char *exitlab)
{
  PNODE p;

  switch(root->symbol)
  {
    case PROGRAM:
      for(p = root->child; p; p = p->brother)
        gencode(p, FALSE, NULL); break;
    case STAT:
       gencode(p->child, FALSE, exitlab); break;
    case ASSIGN:
      gencode(root->child, TRUE, NULL);
      gencode(root->child->brother, FALSE, NULL);
      emit("STO"); break;
    case ID:
      if(isAddr) emit("LDA", root->lexval);
      else emit("LOD", root->lexval); break;
    case INDEXPR:
      emit("LDA", root->child->lexval);
      gencode(root->child->brother, FALSE, NULL);
      emit("IXA", "4");
      if(!isAddr) emit("IND", "0"); break;
    case NUM:
      emit("LDI", root->lexval); break;
    case LOOP:
      lab1 = newlab(); lab2 = newlab();
      emit("LAB", lab1);
      gen(root->child, FALSE, NULL);
      emit("GOF", lab2);
      for(p = root->child->brother; p; p = p->brother)
        gencode(p, FALSE, lab2);
      emit("GOT", lab1);
      emit("LAB", lab2);
      break;
    case BREAK: emit("GOT", exitlab); break;
   }
}
```

```
LAB L1
⟨expr⟩
GOF L2
⟨body⟩
GOT L1
LAB L2
```







Exercise 12

A language is defined by the following BNF:

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat* | *if-stat* | *for-stat*
def-stat → **id :** *type*
type → **int** | **bool**
assign-stat → **id :=** *expr*
expr → *expr* + *expr* | *expr* * *expr* | *expr* **or** *expr* | *expr* **and** *expr* | **not** *expr* | **id** | **intconst** | **boolconst**
if-stat → **if** *expr* **then** *stat-list* **else** *stat-list* **endif**
for-stat → **for id** = *expr* **to** *expr* **do** *stat-list* **endfor**

assuming that the concrete syntax tree of a phrase is binary (pointers: `child`, `brother`), we ask to codify a procedure for P-code generation based on the following requirements:

- A symbol table is used to catalog variables by means of the following functions:
  `void insert(name, type)`: inserts variable name with `type`;
  `Type lookup(name)`: returns the type of variable `name` (`INT`, `BOOL`) if cataloged, otherwise `NULL`;
- Within the concrete syntax tree, each lexical value is stored as a string in field `lexval`;
- The language of the P-machine includes the following set of instructions:
  - `NEI` *id*: allocates integer variable *id* in data memory;
  - `NEB` *id*: allocates boolean variable *id* in data memory;
  - `LDA` *id*: load address of variable *id*;
  - `LOD` *id*: load value of variable *id*;
  - `LDC` *string*: load constant *string*;
  - `LAB` *label*: create *label*;
  - `GOF` *label*: conditional jump (to false);
  - `JMP` *label*: unconditional jump;
  - `PLUS`, `TIMES`: addition; multiplication,
  - `AND`, `OR`: `NOT`: conjunction, disjunction; negation,
  - `LTE`: less than or equal (≤),
  - `STO`: store;
- The auxiliary function `emit(string [, string ])` is used to print an instruction of the P-machine.

Exercise 12

```
void gencode(PNODE root)
{ PNODE p; Type optype; char *op, *name, *lab1, *lab2;
  switch(root->symbol)
  {
    case PROGRAM: gencode(root->child); break;
    case STAT_LIST: gencode(root->child); if(p=root->child->brother->brother) gencode(p); break;
    case STAT: gencode(root->child); break;
    case DEF_STAT: name = root->child->lexval; op = lookup(name) == INT ? "NEI" : "NEB";
                   emit(op, name); break;
    case ASSIGN: emit("LDA", root->child->lexval)
                 gencode(root->child->brother->brother);
                 emit("STO"); break;
    case ID: emit("LOD", root->lexval); break;
    case INTCONST:
    case BOOLCONST: emit("LDC", root->lexval); break;
    case EXPR: if(root->child->brother == NULL)
                 gencode(root->child);
               else if(root->child->type == NOT){
                 gencode(root->child->brother);
                 emit("NOT");
               }
               else {
                 optype = root->child->brother->symbol;
                 gencode(root->child);
                 gencode(root->child->brother->brother);
                 emit(tostring(optype));
               }
               break;
    case IF_STAT: lab1 = newlab(); lab2 = newlab();
                  gencode(root->child->brother);
                  emit("GOF", lab1);
                  gencode(root->child->brother->brother->brother);
                  emit("JMP", lab2);
                  emit("LAB", lab1);
                  gencode(root->child->brother->brother->brother->brother->brother);
                  emit("LAB", lab2); break;
    case FOR_STAT:
          lab1 = newlab(); lab2 = newlab();
          emit("LDA", root->child->brother->lexval);
          gecode(root->child->brother->brother->brother);
          emit("STO");
          emit("LAB", lab1);
          emit("LOD", root->child->brother->lexval);
          gencode(root->child->brother->brother->brother->brother->brother);
          emit("LTE");
          emit("GOF", lab2);
          gencode(root->child->brother->brother->brother->brother->brother->brother->brother);
          emit("LDA", root->child->brother->lexval);
          emit("LOD", root->child->brother->lexval);
          emit("LDC", "1");
          emit("PLUS");
          emit("STO");
          emit("JMP", lab1);
          emit("LAB", lab2); break;
  }
```

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat* | *if-stat* | *for-stat*
def-stat → **id :** *type*
type → **int** | **bool**
assign-stat → **id :=** *expr*
expr → *expr* + *expr* | *expr* * *expr* | *expr* **or** *expr* | *expr* **and** *expr* | **not** *expr* | **id** | **intconst** | **boolconst**
if-stat → **if** *expr* **then** *stat-list* **else** *stat-list* **endif**
for-stat → **for id =** *expr* **to** *expr* **do** *stat-list* **endfor**

⟨*expr*⟩
GOF L1
⟨*stat-list*$_1$⟩
JMP L2
LAB L1
⟨*stat-list*$_2$⟩
LAB L2

LDA **id**
⟨*expr*$_1$⟩
STO
LAB L1
LOD **id**
⟨*expr*$_2$⟩
LTE
GOF L2
⟨*stat-list*⟩
LDA **id**
LOD **id**
LDC 1
PLUS
STO
JMP L1
LAB L2

Exercise 13

A language is defined by the following BNF:

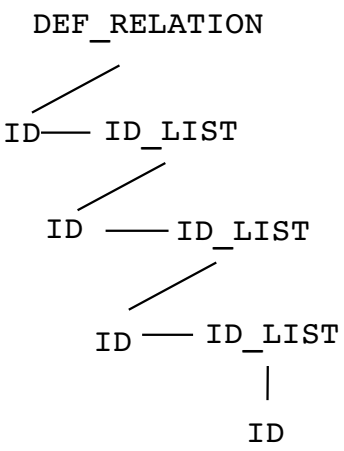
program → *def-relation* *extend-relation*
def-relation → **relation id (** *id-list* **)**
id-list → **id ,** *id-list* | **id**
extend-relation → **extend id by id =** *expr*
expr → *expr* + *term* | *expr* **-** *term* | *term*
term → **id** | **num**

```
relation R (a, b, c)
extend R by n = a + c - 25
```

assuming that the syntax tree of the phrase is semi-abstract (only irrelevant lexical sugar is removed from the concrete tree) and binary (pointers: `child`, `brother`), we first ask to outline the semi-abstract syntax tree relevant to the example phrase, and then to codify a procedure for P-code generation based on the following requirements:

- Within the syntax tree, each lexical value is stored as a string in field `lexval`;
- The translation scheme of the definition of relation *relname* is composed by a first instruction `NEW` *relname* followed by several instructions `ATTR` *attrname*, one for each attribute *attrname* in the relation, terminated by the instruction `END`;
- The translation scheme of the extension of relation *relname* with new attribute *attrname* is composed by a first instruction `EXT` *relname attrname*, followed by the translation of the attribute-value expression, terminated by the instruction `END`;
- The set of instructions for the P-machine also includes `ADD`, `SUB`, `LDC`, and `LOD`, with the usual meaning.
- The auxiliary function `emit(string, ...)`, with one or more string operands, is used to print an instruction of the P-machine.

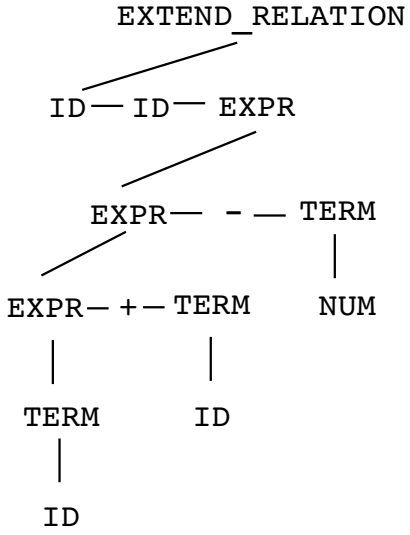
Exercise 13




```
void gen(PNODE root)
{
  gen_def(root->child);
  gen_ext(root->child->brother);
}

void gen_def(PNODE root)
{
  PNODE p;

  emit("NEW", root->child->lexval);
  for(p = root->child->brother; p != NULL; p = p->child->brother)
    emit("ATTR", p->child->lexval);
  emit("END");
}

void gen_ext(PNODE root)
{
  emit("EXT", root->child->lexval, root->child->brother->lexval);
  gen_expr(root->child->brother->brother);
  emit("END");
}

void gen_expr(PNODE root)
{
  PNODE p;

  switch(root->symbol)
  {
    case EXPR: gen_expr(root->child);
               if((p = root->child->brother) != NULL)
               {
                  gen_expr(p->brother);
                  emit(p->symbol == PLUS ? "ADD" : "SUB");
               }
               break;
    case TERM: gen_expr(root->child); break;
    case ID: emit("LOD", root->lexval); break;
    case NUM: emit("LDC", root->lexval); break;
}
```

program → *def-relation* *extend-relation*
def-relation → **id** *id-list*
id-list → **id** *id-list* | **id**
extend-relation → **id** **id** *expr*
expr → *expr* + *term* | *expr* - *term* | *term*
term → **id** | **num**

```
relation R (a, b, c)
extend R by n = a + c - 25
```



Exercise 14

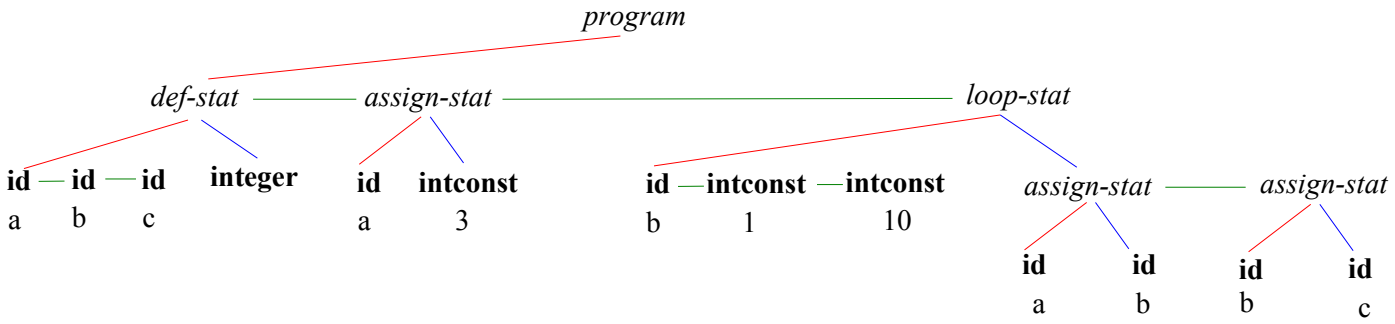
A language is defined by the following BNF:

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 = 4;
  b = 5;
end;
```

kind	lexval	p1	p2	p3

where the abstract tree of the given phrase is so structured:



we ask to codify a procedure of P-code generation for a virtual machine involving the following set of instructions:

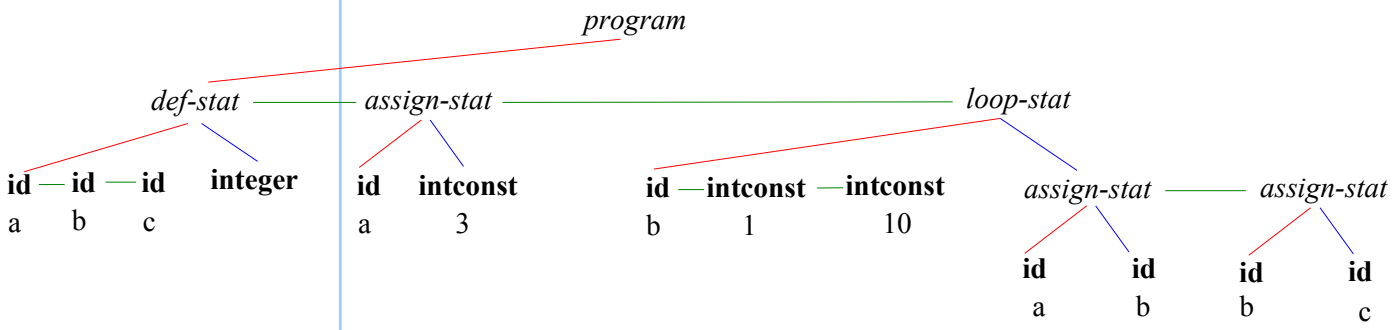
`NEW` *id*: allocate variable named *id*;
`LDA` *id*: load address of variable named *id*;
`LOD` *id*: load value of variable named *id*;
`LDI` *value*: load integer *value*;
`LDS` *value*: load string *value*;
`ADD`: addition;
`SUB`: subtraction;
`MUL`: multiplication;
`DIV`: division;
`STO`: store;
`LAB` *label*: create *label*;
`EQU`: equality;
`LTH`: less than;
`GTH`: greater than;
`JMF` *label*: conditional (to false) jump;
`JMP` *label*: unconditional jump;
`HLT`: halt program (the last instruction of the generated code).

Note: Assume that the counting variable cannot be assigned within the body of the loop.

Exercise 14

```
void gencode(PNODE root)
{ PNODE p;
  switch(root->kind)
  {
    case PROGRAM
      for(p = root->p1; p; p = p->p3)
        gencode(p); break;
      emit("HLT"); break;
    case DEF_STAT
      for(p = root->p1; p; p = p->p3)
        gencode(p); break;
    case ID:
        emit("NEW", root->lexval); break;
    case ASSIGN_STAT:
      emit("LDA", root->p1->lexval);
      gencode(root->p2);
      emit("STO"); break;
    case INTCONST:
      emit("LDI", root->lexval); break;
    case STRCONST:
      emit("LDS", root->lexval); break;
    case LOOP:
      lab = newlab();
      emit("LDA", root->p1->lexval);
      emit("LDI", root->p1->p3->lexval;
      emit("STO");
      emit("LAB", lab);
      for(p = root->p2; p; p = p->p3)
        gencode(p); break;
      emit("LDA", root->p1->lexval);
      emit("LOD", root->p1->lexval);
      emit("LDI", "1");
      emit("ADD");
      emit("STO");
      emit("LOD", root->p1->lexval);
      emit("LDI", root->p1->p3->p3->lexval);
      emit("EQU");
      emit("GOF", lab);
      break;
   }
}
```

```
LDA id
LDI intconst1
STO
LAB L
⟨stat-list⟩
LDA id
LOD id
LDI 1
ADD
STO
LOD id
LDI intconst2
EQU
GOF L
```



Exercise 15

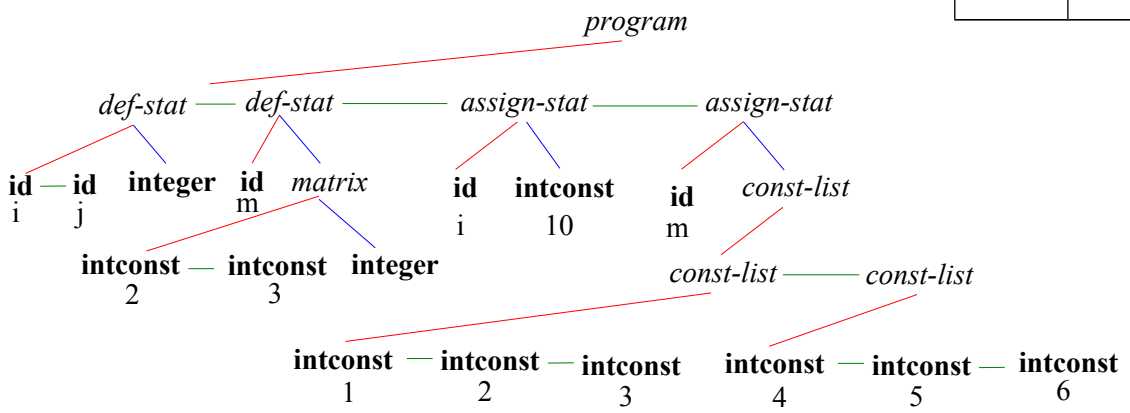
A language is defined by the following BNF:

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-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*

```
var i, j is integer;
var m is matrix(2,3) of integer;
i = 10;
m = [[1,2,3],[4,5,6]];
```

where the abstract tree of the given phrase is so structured:

kind	lexval	p1	p2	p3



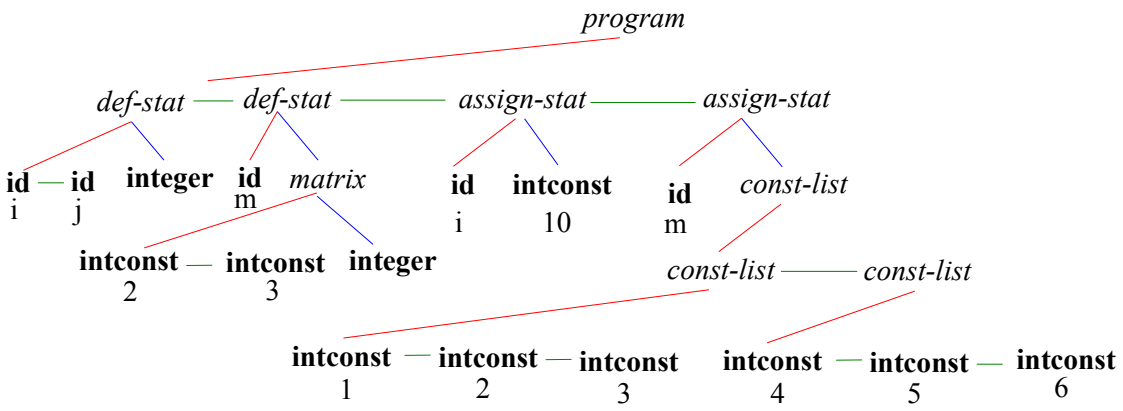
we ask to codify a procedure of P-code generation for a virtual machine involving the following set of instructions:

- **DEF** *id*: allocate variable named *id*;
- **LDA** *id*: load address of variable named *id*;
- **LDI** *constant*: load integer *constant*;
- **LDS** *constant*: load string *constant*;
- **HLT**: halt program (the last instruction of the generated code).

Note: To load a matrix constant in assignment, all involved atomic constants (left to right) must be loaded before the (unique) final store.

Exercise 15

```
void gencode(PNODE root)
{
  PNODE p;

  switch(root->symbol)
  {
    case PROGRAM
      for(p = root->p1; p; p = p->p3)
        gencode(p); break;
      emit("HLT"); break;
    case DEF_STAT
      for(p = root->p1; p; p = p->p3)
        gencode(p); break;
    case ID:
        emit("DEF", root->lexval); break;
    case ASSIGN:
      emit("LDA", root->p1->lexval);
      gencode(root->p2);
      emit("STO"); break;
    case INTCONST:
      emit("LDI", root->lexval); break;
    case STRCONST:
      emit("LDS", root->lexval); break;
    case CONST_LIST:
      for(p = root->p1; p; p = p->p3)
        gencode(p); break;
   }
}
```



Exercise 16

Given the language for the manipulation of integers, defined by the following BNF,

program → *stat-list*
stat-list → *stat* *stat-list* | *stat*
stat → **id** := *expr*
expr → *expr* + *expr* | *expr* * *expr* | *expr* **and** *expr* | *expr* **or** *expr* | **not** *expr* | **(** *expr* **)** | **id** | **num**

assuming a concrete syntax tree where nodes are structured by the following fields:

- `Symbol symbol`: the grammar symbol,
- `char *lexval`: lexical value (for both identifiers and numbers),
- `child`: pointer to first child,
- `brother`: pointer to right brother,

we ask to specify a procedure of P-code generation based on the following requirements:

1. Logical operators are based on the same rules of the C programming language (0 stands for **false**, while a number different from 0 stands for **true**);
2. Operands are evaluated from left to right;
3. Logical **and** is evaluated in short circuit, (while logical **or** is fully evaluated);
4. The language of the P-machine includes the following set of instructions:

LDA <id>	(loading of address of variable **<id>** on stack)
LOD <id>	(loading of value of variable **<id>** on stack)
LDC <const>	(loading of integer constant **<const>** on stack)
PLUS	(arithmetic addition)
TIMES	(arithmetic multiplication)
AND	(conjunction)
OR	(disjunction)
NOT	(negation)
GOFALSE <label>	(conditional jump)
GOTO <label>	(unconditional jump)
LABEL <label>	(implicit address)
STO	(store)
HALT	(program termination: to be generated as the final instruction)

Exercise 16

```
void gen(Pnode p)
{  char *lab1, *lab2;
   switch(p->symbol){
   case PROGRAM: gen(p->child);
                 emit("HALT"); break;

   case STAT-LIST: gen(p->child);
                   if(p->child->brother)
                     gen(p->child->brother); break;

   case STAT: emit("LDA", p->child->lexval);
              gen(p->child->brother->brother);
              emit("STO"); break;

   case EXPR: if(p->child->symbol == ID)
                emit("LOD", p->child->lexval);
              elsif(p->child->symbol == NUM)
                emit("LDC", p->child->lexval);
              elsif(p->child->symbol == NOT)
                {gen(p->child->brother); emit("NOT");}
              elsif(p->child->symbol == '(')
                gen(p->child->brother);
              elsif(p->child->brother->symbol == '+')
                {gen(p->child); gen(p->child->brother->brother; emit("PLUS");}
              elsif(p->child->brother->symbol == '*')
                {gen(p->child); gen(p->child->brother->brother; emit("TIMES");}
              elsif(p->child->brother->symbol == OR)
                {gen(p->child); gen(p->child->brother->brother; emit("OR");}
              elsif(p->child->brother->symbol == AND){
                lab1 = newlab(); lab2 = newlab();
                gen(p->child);
                emit("GOFALSE", lab1);
                gen(p->child->brother->brother);
                emit("GOTO", lab2);
                emit("LABEL", lab1);
                emit("LDC", "0");
                emit("LABEL", lab2);
              }
              break;
    }
}
```

program → *stat-list*
stat-list → *stat* *stat-list* | *stat*
stat → **id** := *expr*
expr → *expr* + *expr* | *expr* * *expr* | *expr* **and** *expr* | *expr* **or** *expr* | **not** *expr* | (*expr*) | **id** | **num**

⟨$cond_1$⟩
GOFALSE L1
⟨$cond_2$⟩
GOTO L2
LABEL L1
LDC 0
LABEL L2

Exercise 17

Given the language for the manipulation of integers, defined by the following BNF,

program → *stat-list*
stat-list → *stat* *stat-list* | *stat*
stat → **id** := *expr*
expr → *expr* **+** *expr* | *expr* ***** *expr* | **id** | **num**

assuming a concrete syntax tree where nodes are qualified by fields `symbol` (the grammar symbol), `lexeme` (lexical string), `p1` (pointer to first child), `p2` (pointer to second child), and `p3` (pointer to third child), we ask to specify a procedure of P-code generation based on the following requirements:

a. Operands are evaluated from left to right;
b. Unlike the addition, the multiplication is evaluated in short circuit, based on the following rule: if the first operand is 0 then the result is 0 (otherwise, fully evaluation of multiplication is required);
c. The language of the P-machine includes the following set of instructions:

LDA <id>	(loading of address of variable **<id>** on stack)
LOD <id>	(loading of value of variable **<id>** on stack)
LDC <const>	(loading of integer constant **<const>** on stack)
ADD	(addition)
MUL	(multiplication)
EQU	(equality, pushing either `TRUE` or `FALSE` on the stack)
JMF <label>	(jumps if `FALSE`)
JMP <label>	(unconditional jump)
LAB <label>	(implicit address)
STO	(store)

Note: In the P-machine, integers cannot be treated as booleans.

Exercise 17

```
void gen(Pnode p)
{  char *lab1, *lab2;
   switch(p->symbol){
   case PROGRAM: gen(p->p1); break;

   case STAT-LIST: gen(p->p1);
                   if(p->p2)
                     gen(p->p2);
                   break;

   case STAT: emit("LDA", p->p1->lexeme);
              gen(p->p3);
              emit("STO"); break;

   case EXPR: if(p->p1->symbol == ID)
                emit("LOD", p->p1->lexeme);
              elsif(p->p1->symbol == NUM)
                emit("LDC", p->p1->lexeme);
              elsif(p->p2->symbol == '+'){
                gen(p->p1);
                gen(p->p3);
                emit("ADD");
              }
              elsif(p->p2->symbol == '*'){
                lab1 = newlab(); lab2 = newlab();
                gen(p->p1);
                emit("LDC", "0");
                emit("EQU");
                emit("JMF", lab1);
                emit("LDC", "0");
                emit("JMF", lab2);
                emit("LAB", lab1);
                gen(p->p1);
                gen(p->p3);
                emit("MUL");
                emit("LAB", lab2);
              }
              break;
    }
}
```

program → *stat-list*
stat-list → *stat* *stat-list* | *stat*
stat → **id** := *expr*
expr → *expr* **+** *expr* | *expr* ***** *expr* | **id** | **num**

⟨*expr*$_1$⟩
LDC 0
EQU
JMF L1
LDC 0
JMP L2
LAB L1
⟨*expr*$_1$⟩
⟨*expr*$_2$⟩
MUL
LAB L2