# Exercise 1

Specify the attribute grammar relevant to the following BNF:

*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;
```

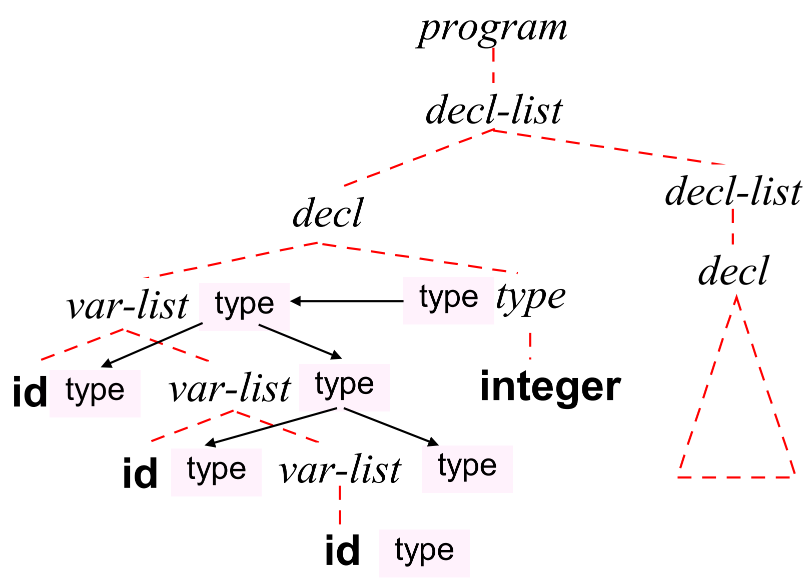
# Exercise 1

Specify the attribute grammar relevant to the following BNF:

*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;
```

| Production | Semantic rules |
|---|---|
| *program* → *decl-list* | |
| $decl\text{-}list_1$ → *decl* **;** $decl\text{-}list_2$ | |
| *decl-list* → *decl* | |
| *decl* → *var-list* **:** *type* | *var-list*.type = *type*.type |
| $var\text{-}list_1$ → **id ,** $var\text{-}list_2$ | **id**.type = $var\text{-}list_1$.type<br>$var\text{-}list_2$.type = $var\text{-}list_1$.type |
| *var-list* → **id** | **id**.type = *var-list*.type |
| *type* → **integer** | *type*.type = INTEGER |
| *type* → **string** | *type*.type = STRING |
| *type* → **boolean** | *type*.type = BOOLEAN |




A = { type }

# Exercise 2

Specify the attribute grammar, whose equations represent assignments, for the language of table declarations relevant to the following BNF:

*def* → **id : (** *attr-list* **)**
*attr-list* → *decl* **,** *attr-list* | *decl*
*decl* → **id :** *type*
*type* → **int** | **string** | **bool**

```
def R: (a: int, b: string, c: bool)
```

Note: Homonymous fields are not allowed within the table.

# Exercise 2

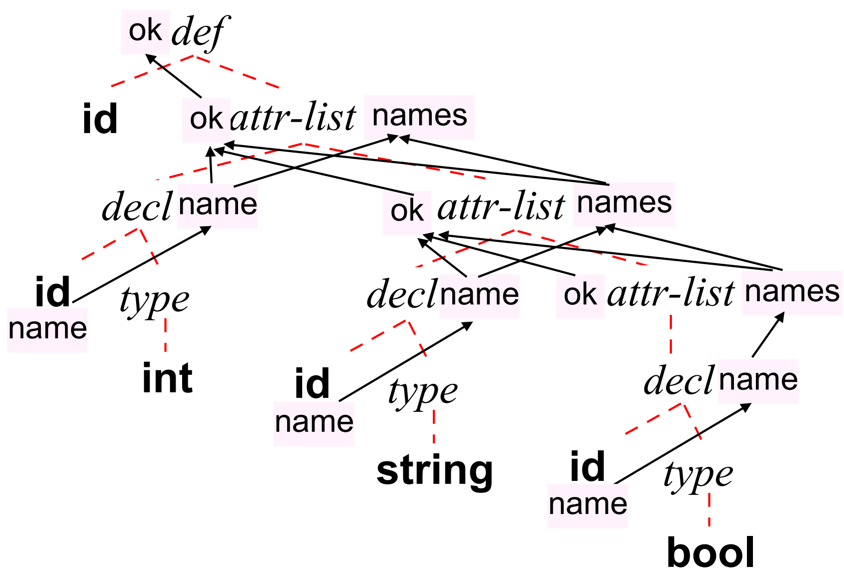
Specify the attribute grammar, whose equations represent assignments, for the language of table declarations relevant to the following BNF:

$def \rightarrow$ **id : (** $attr\text{-}list$ **)**
$attr\text{-}list \rightarrow decl$ **,** $attr\text{-}list \mid decl$
$decl \rightarrow$ **id :** $type$
$type \rightarrow$ **int** | **string** | **bool**

```
def R: (a: int, b: string, c: bool)
```

<u>Note</u>: Homonymous fields are not allowed within the table.

| Production | Semantic rules |
|---|---|
| $def \rightarrow$ **id : (** $attr\text{-}list$ **)** | $def$.ok = $attr\text{-}list$.ok |
| $attr\text{-}list_1 \rightarrow decl$ **,** $attr\text{-}list_2$ | $attr\text{-}list_1$.ok = $attr\text{-}list_2$.ok **and** $decl$.name $\notin$ $attr\text{-}list_2$.names<br>$attr\text{-}list_1$.names = $attr\text{-}list_2$.names $\cup$ { $decl$.name } |
| $attr\text{-}list \rightarrow decl$ | $attr\text{-}list$.ok = **true**<br>$attr\text{-}list$.names = { $decl$.name } |
| $decl \rightarrow$ **id :** $type$ | $decl$.name = **id**.name |
| $type \rightarrow$ **int** | |
| $type \rightarrow$ **string** | |
| $type \rightarrow$ **bool** | |




A = { ok, name, names }

# Exercise 3

Given the language **L** defined by the following BNF:

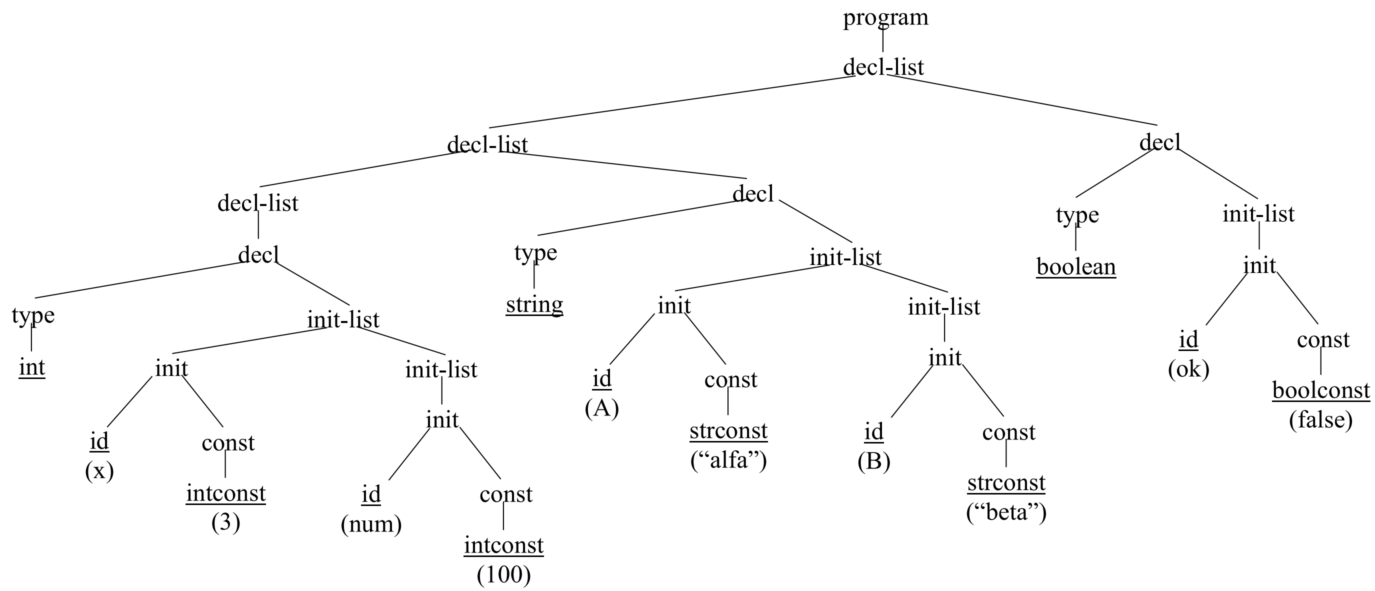
*program* → *decl-list*
*decl-list* → *decl-list decl* | *decl*
*decl* → *type init-list* **;**
*type* → **int** | **string** | **boolean**
*init-list* → *init* **,** *init-list* | *init*
*init* → **id** = *const*
*const* → **intconst** | **strconst** | **boolconst**

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

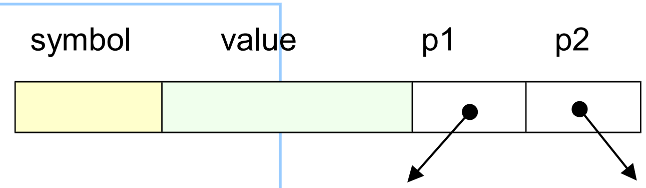
a) Outline the abstract syntax tree relevant to the given phrase;
b) Codify the semantic procedure in order to:
- Check the initializations of the phrases of **L**;
- For each defined variable, call function `insert` (whose coding is not required), which is assumed to insert in a symbol table, for each variable identifier, relevant type and (initialization) value.

In case of semantic error, after an error message, the semantic analysis ends immediately.

# Exercise 3




```
int type;
sem(Node *n)
{
    switch(n->symbol)
    {
    case PROGRAM: sem(n->p1); break;
    case DECL-LIST: sem(n->p1);
                    if (n->p2 != NULL) sem(n->p2); break;
    case DECL: sem(n->p1); sem(n->p2); break;
    case INIT_LIST: sem(n->p1);
                    if (n->p2 != NULL) sem(n->p2); break;
    case TYPE: switch (n->p1->symbol)
               {
               case INT: type = INTEGER; break;
               case STRING: type = STRING; break;
               case BOOLEAN: type = BOOLEAN; break;
               }
    case INIT: if !compatible(type, n->p2->p1->symbol) semerror("Incompatible initialization");
               else insert(n->p1->value.sval, type, n->p2->p1->value);
               break;
    }
}
```




| *name* | *type* | *value* |
|---|---|---|
| x | INTEGER | 3 |
| num | INTEGER | 100 |
| A | STRING | "alpha" |
| B | STRING | "beta" |
| ok | BOOLEAN | false |
| | | |
| | | |

# Exercise 4

Specify the attribute grammar relevant to the following BNF:

*program* → *proc-decl* *proc-call*
*proc-decl* → **procedure id (** *formal-list* **)**
*formal-list* → *formal* **,** *formal-list* | *formal*
*formal* → **id :** *domain*
*domain* → **int** | **string** | **bool**
*proc-call* → **id (** *actual-list* **)**
*actual-list* → *actual* **,** *actual-list* | *actual*
*actual* → **intconst** | **strconst** | **boolconst**

```
procedure P (a: int, b: string)
P(3, "alpha")
```

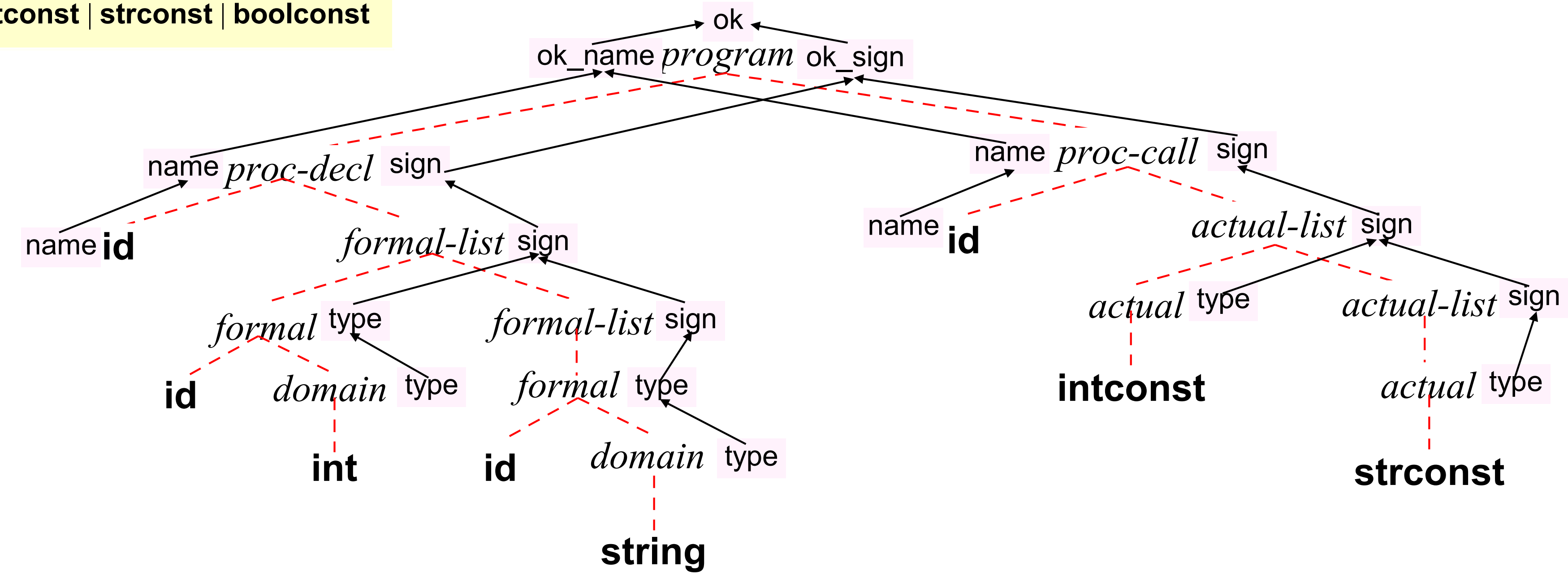
based on the following semantic constraints:

a) The name of the called procedure shall be equal to the name of the declared procedure.
b) The number of actual parameters shall be equal to that of formal parameters.
c) Each actual parameter shall be compatible with corresponding formal parameter.

# Exercise 4

*program* → *proc-decl* *proc-call*
*proc-decl* → **procedure id (** *formal-list* **)**
*formal-list* → *formal* **,** *formal-list* | *formal*
*formal* → **id :** *domain*
*domain* → **int** | **string** | **bool**
*proc-call* → **id (** *actual-list* **)**
*actual-list* → *actual* **,** *actual-list* | *actual*
*actual* → **intconst** | **strconst** | **boolconst**

```
procedure P (a: int, b: string)
P(3, "alpha")
```




A = { ok, ok_name, ok_sign, name, type, sign }

# Exercise 4 (ii)

| *Production* | *Semantic rules* |
|---|---|
| $program \rightarrow proc\text{-}decl\ \ proc\text{-}call$ | $program$.ok_name := $proc\text{-}decl$.name = $proc\text{-}call$.name<br>$program$.ok_sign := $proc\text{-}decl$.sign = $proc\text{-}call$.sign<br>$program$.ok := $program$.ok_name **`and`** $program$.ok_sign |
| $proc\text{-}decl \rightarrow$ **procedure id (** $formal\text{-}list$ **)** | $proc\text{-}decl$.name := **id**.name<br>$proc\text{-}decl$.sign := $formal\text{-}list$.sign |
| $formal\text{-}list_1 \rightarrow formal$ **,** $formal\text{-}list_2$ | $formal\text{-}list_1$.sign := [$formal$.type] $\cup$ $formal\text{-}list_2$.sign |
| $formal\text{-}list \rightarrow formal$ | $formal\text{-}list$.sign := [$formal$.type] |
| $formal \rightarrow$ **id :** $domain$ | $formal$.type := $domain$.type |
| $domain \rightarrow$ **int** | $domain$.type := INT |
| $domain \rightarrow$ **string** | $domain$.type := STRING |
| $domain \rightarrow$ **bool** | $domain$.type := BOOL |
| $proc\text{-}call \rightarrow$ **id (** $actual\text{-}list$ **)** | $proc\text{-}call$.sign := $actual\text{-}list$.sign<br>$proc\text{-}call$.name := **id**.name |
| $actual\text{-}list_1 \rightarrow actual$ **,** $actual\text{-}list_2$ | $actual\text{-}list_1$.sign := [$actual$.type] $\cup$ $actual\text{-}list_2$.sign |
| $actual\text{-}list \rightarrow actual$ | $actual\text{-}list$.sign := [$actual$.type] |
| $actual \rightarrow$ **intconst** | $actual$.type := INT |
| $actual \rightarrow$ **strconst** | $actual$.type := STRING |
| $actual \rightarrow$ **boolconst** | $actual$.type := BOOL |

# Exercise 5

Specify the attribute grammar, whose equations represent assignments, for the language of table declarations relevant to the following BNF:

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

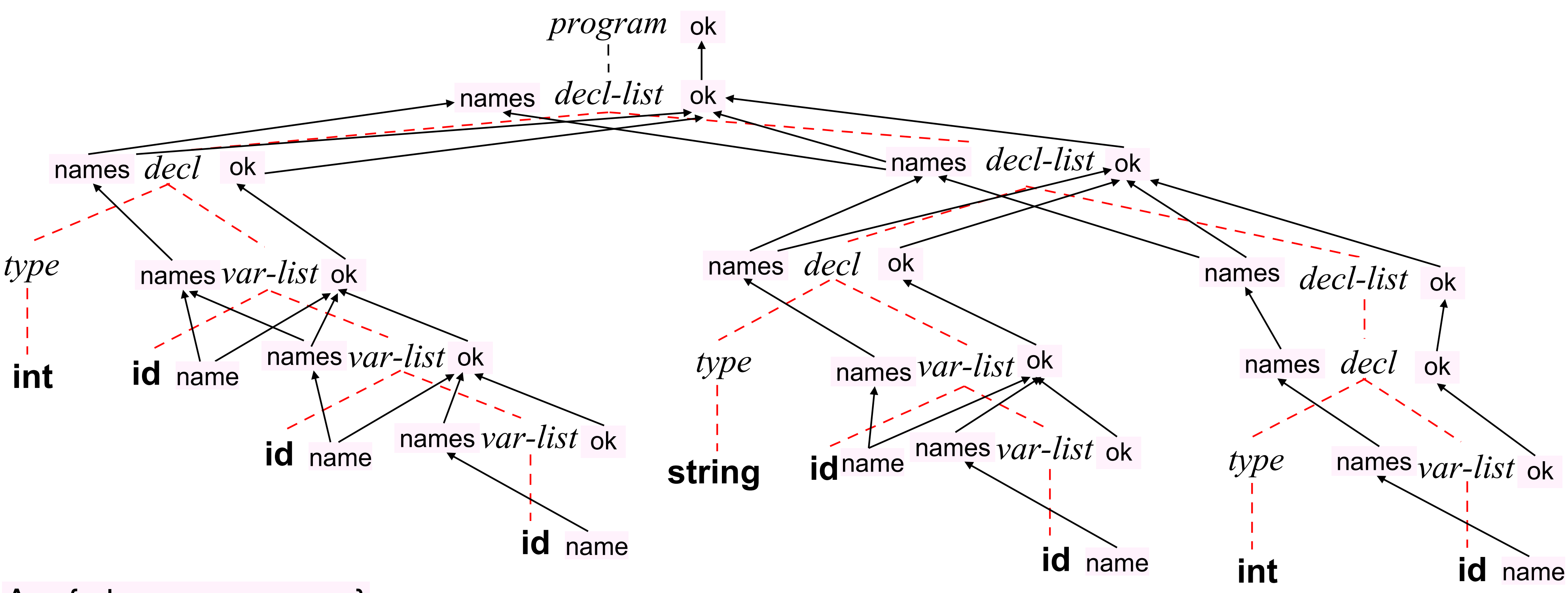
```
int a, b, c;
string x, y;
int z;
```

such that, within each phrase, names of variables are unique.

# Exercise 5

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

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




A = { ok, name, names }

# Exercise 5 (ii)

| Production | Semantic rules |
|---|---|
| $program \rightarrow decl\text{-}list$ | $program$.ok = $decl\text{-}list$.ok |
| $decl\text{-}list_1 \rightarrow decl$ **;** $decl\text{-}list_2$ | $decl\text{-}list_1$.ok = $decl$.ok **and** $decl\text{-}list_2$.ok **and** ($decl$.names $\cap$ $decl\text{-}list_2$.names = $\emptyset$)<br>$decl\text{-}list_1$.names = $decl$.names $\cup$ $decl\text{-}list_2$.names |
| $decl\text{-}list \rightarrow decl$ **;** | $decl\text{-}list$.ok = $decl$.ok<br>$decl\text{-}list$.names = $decl$.names |
| $decl \rightarrow type\ var\text{-}list$ | $decl$.ok = $var\text{-}list$.ok<br>$decl$.names = $var\text{-}list$.names |
| $type \rightarrow$ **int** | |
| $type \rightarrow$ **string** | |
| $var\text{-}list_1 \rightarrow$ **id ,** $var\text{-}list_2$ | $var\text{-}list_1$.ok = $var\text{-}list_2$.ok **and** (**id**.name $\notin$ $var\text{-}list_2$.names)<br>$var\text{-}list_1$.names = $var\text{-}list_2$.names $\cup$ { **id**.name} |
| $var\text{-}list \rightarrow$ **id** | $var\text{-}list$.ok = **true**<br>$var\text{-}list$.names = { **id**.name} |

# Exercise 6

Specify the attribute grammar relevant to the following BNF:

*program* → *decl-list*
*decl-list* → *decl* **,** *decl-list* | *decl*
*decl* → **class id** *inheritance* **;**
*inheritance*→ **inherits id** | **ε**

```
class A;
class B;
class C inherits B;
```

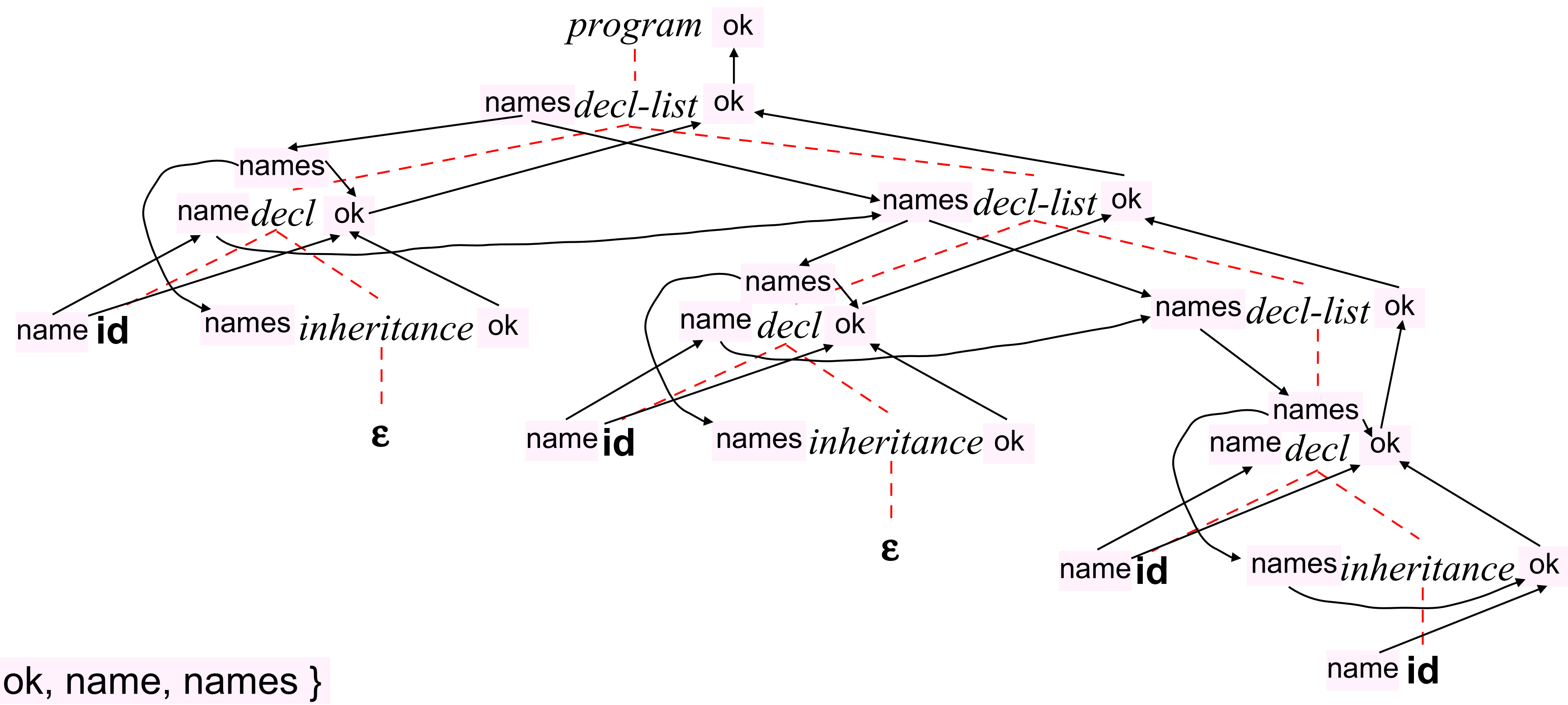
based on the following semantic constraints:

a) Names of classes are unique.
b) In inheritance, the superclass shall be defined (previously).

# Exercise 6

*program* → *decl-list*
*decl-list* → *decl* **,** *decl-list* | *decl*
*decl* → **class id** *inheritance* **;**
*inheritance*→ **inherits id** | ε

```
class A;
class B;
class C inherits B;
```




A = { ok, name, names }

# Exercise 6 (ii)

| *Production* | *Semantic rules* |
|---|---|
| *program* → *decl-list* | *decl-list*.names := ∅ <br> *program*.ok := *decl-list*.ok |
| $decl\text{-}list_1$ → *decl* **,** $decl\text{-}list_2$ | $decl\text{-}list_2$.names := $decl\text{-}list_1$.names ∪ {*decl*.name} <br> *decl*.names := $decl\text{-}list_1$.names <br> $decl\text{-}list_1$.ok := *decl*.ok **and** $decl\text{-}list_2$.ok |
| *decl-list* → *decl* | *decl-list*.ok := *decl*.ok <br> *decl*.names := *decl-list*.names |
| *decl* → **class id** *inheritance* **;** | *inheritance*.names := *decl*.names <br> *decl*.name := **id**.name <br> *decl*.ok := (**id**.name ∉ *decl*.names) **and** *inheritance*.ok |
| *inheritance*→ **inherits id** | *inheritance*.ok := **id**.name ∈ *inheritance*.names |
| *inheritance*→ **ε** | *inheritance*.ok := **true** |

# Exercise 7

Specify the attribute grammar relevant to the following BNF:

*program* → *def-stat project-stat*
*def-stat* → **def id (** *id-list* **)**
*id-list* → **id ,** *id-list* | **id**
*project-stat* → **project (** *id-list* **) id**

```
def R (a, b, c)
project (a, c) R
```

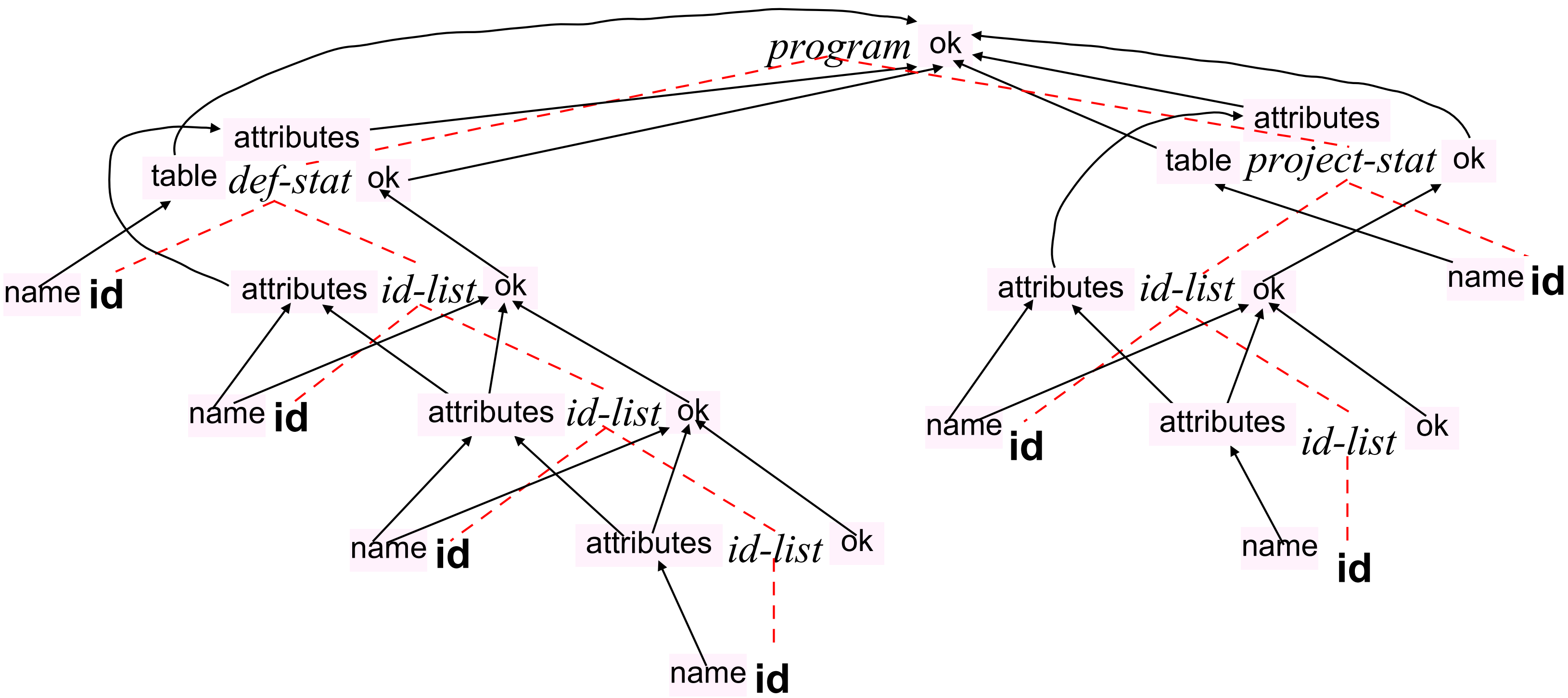
based on the following semantic constraints:

a) Attribute names are unique within a table.
b) The name of operand table in the projection shall be equal to the name of the defined table.
c) Names of projection attributes shall be unique.
d) Projection attributes shall be strictly contained within the schema of the operand table.

# Exercise 7

*program* → *def-stat project-stat*
*def-stat* → **def id (** *id-list* **)**
*id-list* → **id ,** *id-list* | **id**
*project-stat* → **project (** *id-list* **) id**

```
def R (a, b, c)
project (a, c) R
```




A = { ok, table, name, attributes }

# Exercise 7 (ii)

| *Production* | *Semantic rules* |
|---|---|
| $program \rightarrow def\text{-}stat\ project\text{-}stat$ | $program$.ok := $def\text{-}stat$.ok **and** $def\text{-}stat$.table = $project\text{-}stat$.table **and** $project\text{-}stat$.ok **and** $project\text{-}stat$.attributes $\subset$ $def\text{-}stat$.attributes |
| $def\text{-}stat \rightarrow$ **def id (** $id\text{-}list$ **)** | $def\text{-}stat$.table := **id**.name<br>$def\text{-}stat$.ok := $id\text{-}list$.ok<br>$def\text{-}stat$.attributes := $id\text{-}list$.attributes |
| $id\text{-}list_1 \rightarrow$ **id ,** $id\text{-}list_2$ | $id\text{-}list_1$.ok := $id\text{-}list_2$.ok **and id**.name $\not\in$ $id\text{-}list_2$.attributes<br>$id\text{-}list_1$.attributes := $id\text{-}list_2$.attributes $\cup$ { **id**.name } |
| $id\text{-}list \rightarrow$ **id** | $id\text{-}list$.ok := **true**<br>$id\text{-}list$.attributes := { **id**.name } |
| $project\text{-}stat \rightarrow$ **project (** $id\text{-}list$ **) id** | $project\text{-}stat$.table := **id**.name<br>$project\text{-}stat$.ok := $id\text{-}list$.ok<br>$project\text{-}stat$.attributes := $id\text{-}list$.attributes |

# Exercise 8

Specify the attribute grammar relevant to 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**

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

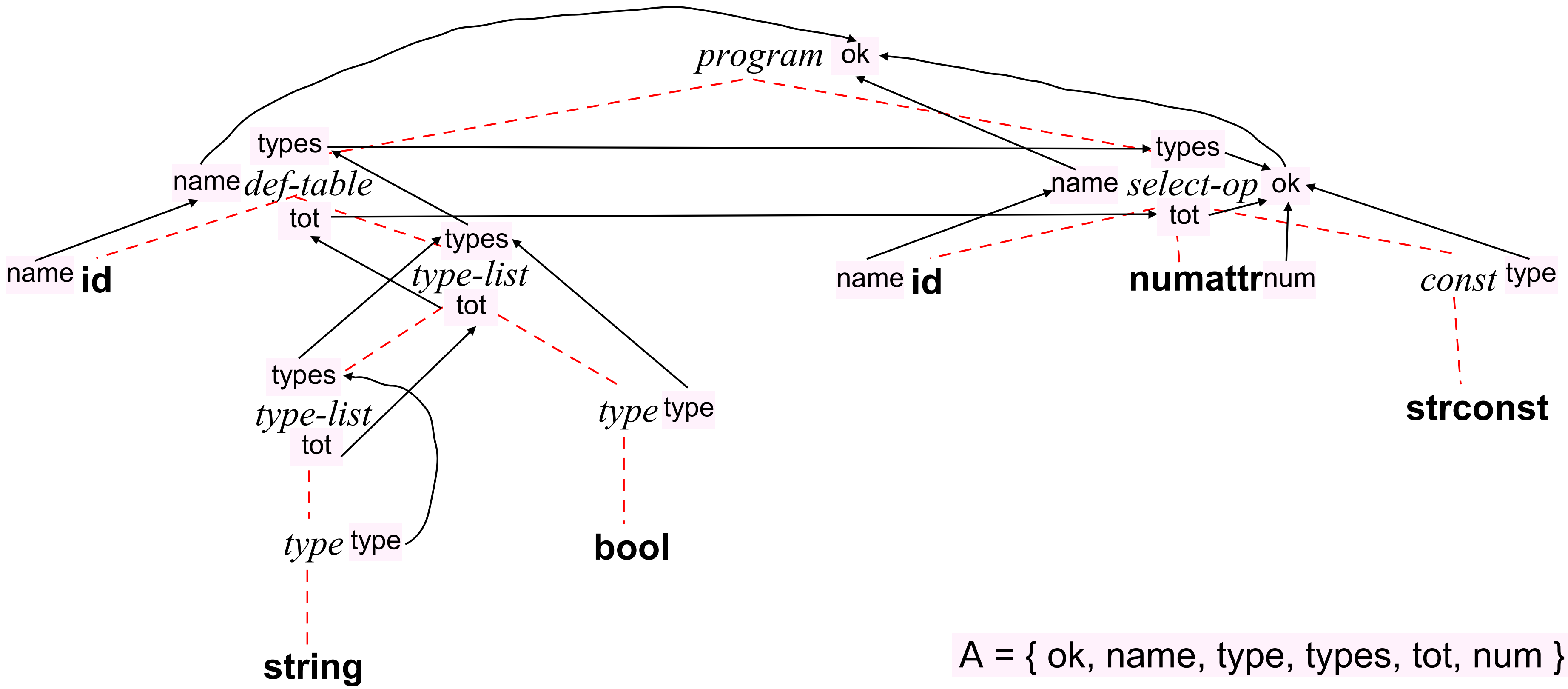
based on the following semantic constraints:

1) The name of the operand table in selection shall be equal to the name of the defined table.
2) The number **numattr** (identifying positionally an attribute) shall be between 1 and the number of table attributes.
3) Within **where** clause, attribute identified by **numattr** shall be of the same type of the constant involved in the comparison.

# Exercise 8

*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**

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




A = { ok, name, type, types, tot, num }

# Exercise 8 (ii)

| *Production* | *Semantic rules* |
|---|---|
| *program* → *def-table* *select-op* | *program*.ok := *select-op*.ok **and** *def-table*.name = *select-op*.name<br>*select-op*.types := *def-table*.types<br>*select-op*.tot := *def-table*.tot |
| *def-table*→ **table id (** *type-list* **)** | *def-table*.name := **id**.name<br>*def-table*.types := *type-list*.types<br>*def-table*.tot := *type-list*.tot |
| $type\text{-}list_1$ → $type\text{-}list_2$ **,** *type* | $type\text{-}list_1$.types := $type\text{-}list_2$.types ∪ [ *type*.type ]<br>$type\text{-}list_1$.tot := $type\text{-}list_2$.tot **+** 1 |
| *type-list* → *type* | *type-list*.types := [ *type*.type ]<br>*type-list*.tot := 1 |
| *type*→ **string** | *type*.type := STRING |
| *type*→ **bool** | *type*.type := BOOL |
| *select-op*→ **select id where numattr =** *const* | *select-op*.name := **id**.name<br>*select-op*.ok := **numattr**.num ≥1 **and** **numattr**.num ≤ *select-op*.tot **and** **select-op**.types[**numattr**.num] = *const*.type |
| *const* → **strconst** | *const*.type := STRING |
| *const* → **boolconst** | *const*.type := BOOL |

# Exercise 9

Specify the attribute grammar relevant to the following BNF:

*program* → *proc-decl* *proc-call*
*proc-decl*→ **procedure id (** *form-params* **) ;**
*form-params* → *param-decl* **,** *form-params* | *param-decl*
*param-decl* → **id :** *type*
*type* → **int** | **real** | **string**
*proc-call*→ **call id with** *act-params* **;**
*act-params* → *param-set* **,** *act-params* | *param-set*
*param-set* → **id =** *const*
*const* → **intconst** | **realconst** | **strconst**

```
procedure P (a: int, b: real, c: string);
call P with b = 4.12, a = 24, c = "beta";
```

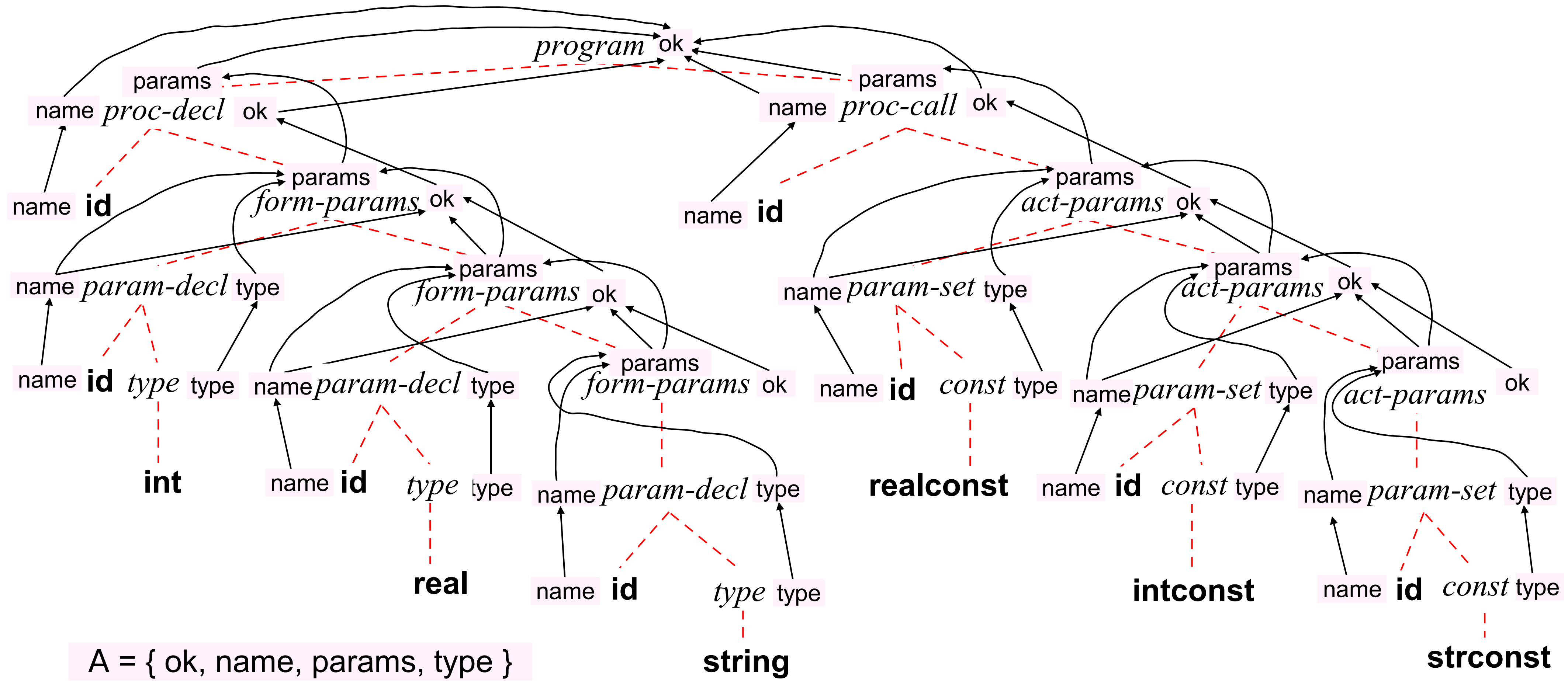
based on the following semantic constraints:

- The name of the called procedure shall be equal to the name of the defined procedure;
- Names of formal parameters are unique;
- Within call, all parameters shall be instantiated once based on their types;
- The correspondence between formal and actual parameters is explicit.

# Exercise 9

*program* → *proc-decl* *proc-call*
*proc-decl* → **procedure id (** *form-params* **) ;**
*form-params* → *param-decl* **,** *form-params* | *param-decl*
*param-decl* → **id :** *type*
*type* → **int** | **real** | **string**
*proc-call* → **call id with** *act-params* **;**
*act-params* → *param-set* **,** *act-params* | *param-set*
*param-set* → **id =** *const*
*const* → **intconst** | **realconst** | **strconst**

```
procedure P (a: int, b: real, c: string);
call P with b = 4.12, a = 24, c = "beta";
```




A = { ok, name, params, type }

# Exercise 9 (ii)

| ***Production*** | ***Semantic rules*** |
|---|---|
| *program* → *proc-decl* *proc-call* | *program*.ok := *proc-decl*.ok **and** *proc-call*.ok **and**<br>*proc-decl*.name = *proc-call*.name **and**<br>*proc-decl*.params = *proc-call*. params |
| *proc-decl*→ **procedure id (** *form-params* **) ;** | *proc-decl*.name := **id**.name<br>*proc-decl*.params := *form-params*.params<br>*proc-decl*.ok := *form-params*.ok |
| $form\text{-}params_1$ → *param-decl* **,** $form\text{-}params_2$ | $form\text{-}params_1$.ok := $form\text{-}params_2$.ok **and**<br>missing(*param-decl*.name, $form\text{-}params_2$.params)<br>$form\text{-}params_1$.params :=<br>$form\text{-}params_2$.params ∪ { ( *param-decl*.name, *param-decl*.type) } |
| *form-params* → *param-decl* | *form-params*.ok := **true**<br>*form-params*.params := { ( *param-decl*.name, *param-decl*.type) } |
| *param-decl* → **id :** *type* | *param-decl*.name := **id**.name<br>*param-decl*.type := *type*.type |
| *type*→ **int** | *type*.type := INT |
| *type*→ **real** | *type*.type := REAL |
| *type*→ **string** | *type*.type := STRING |
| *proc-call*→ **call id with** *act-params* **;** | *proc-call*.name := **id**.name<br>*proc-call*.params := *act-params*.params<br>*proc-decl*.ok := *act-params*.ok |
| $act\text{-}params_1$ → *param-set* **,** $act\text{-}params_2$ | $act\text{-}params_1$.ok := $act\text{-}params_2$.ok **and**<br>missing(*param-set*.name, $act\text{-}params_2$.params)<br>$act\text{-}params_1$.params :=<br>$act\text{-}params_2$.params ∪ { ( *param-set*.name, *param-set*.type) } |
| *act-params* → *param-set* | *act-params*.ok := **true**<br>*act-params*.params := { ( *param-set*.name, *param-set*.type) } |
| *param-set* → **id =** *const* | *param-set*.name := **id**.name<br>*param-set*.type := *const*.type |
| *const* → **intconst** | *const*.type := INT |
| *const* → **realconst** | *const*.type := REAL |
| *const* → **strconst** | *const*.type := STRING |

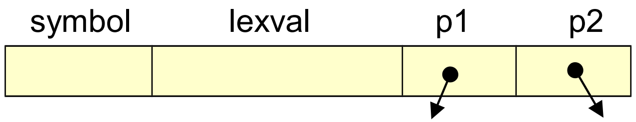
# Exercise 9 (iii)

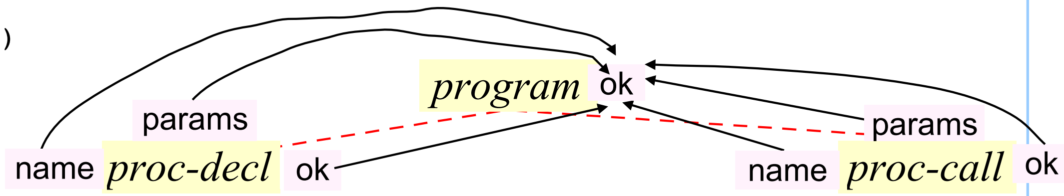
| symbol | lexval | p1 | p2 | name | type | params | ok |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

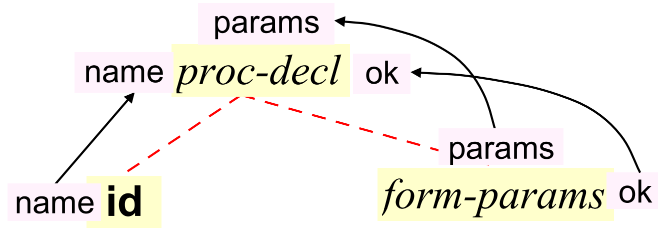
```
sem(Node *p)
{
    switch(p->symbol)
    {
    case PROGRAM: sem(p->p1); sem(p->p2);
                  p->ok = p->p1->ok && p->p2->ok &&
                  p->p1->name == p->p2->name &&
                  p->p1->params == p->p2->params;
                  break;
    case PROC-DECL:
    case PROC-CALL: sem(p->p1); sem(p->p2);
                    p->name = p->p1->name; p->params = p->p2->params; p->ok = p->p2->ok;
                    break;
    case FORM-PARAMS:
    case ACT-PARAMS: sem(p->p1);
                     if(p->p2){
                         sem(p->p2);
                         p->ok = p->p2->ok && missing(p->p1->name, p->p2->params);
                         p->params = union(p->p2->params, singleton(p->p1->name, p->p1->type));
                     else {p->ok = TRUE; p->params = singleton(p->p1->name, p->p1->type);}
                     break;
    case PARAM-DECL:
    case PARAM-SET: sem(p->p1); sem(p->p2);
                    p->name = p->p1->name; p->type = p->p2->type;
                    break;
    case TYPE:
         CONST: p->type = (p->p1->symbol == INT || p->p1->symbol == INTCONST ? INT :
                          (p->p1->symbol == REAL || p->p1->symbol == REALCONST ? REAL : STRING));
                break;
    case ID: p->name = p->lexval.sval;
             break;
     }
}
```

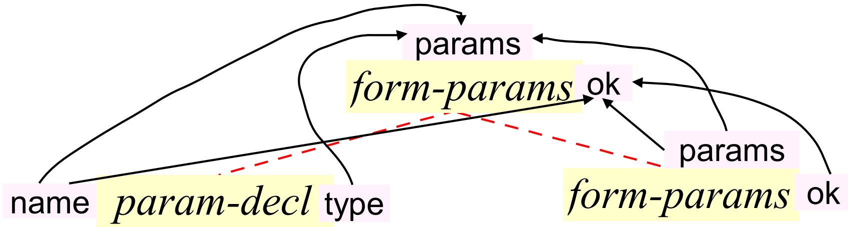
# Exercise 9 (iv)

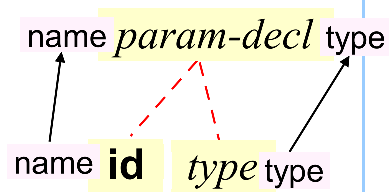
```
Bool program(Node *p)
{  Bool ok1, ok2; char *name1, *name2; Table params1, params2;

   ok1 = proc_decl_call(p->p1, &name1, &params1);
   ok2 = proc_decl_call (p->p2, &name2, &params2);
   return(ok1 && ok2 && name1 == name2 && params1 == params2);
}

Bool proc_decl_call(Node *p, char **name, Table *params)
{
   *name = id(p->p1);
   return(form_act_params(p->p2, params));
}

Bool form_act_params(Node *p, Table *params)
{  char *name; int type; Table params2; Bool ok2;

   name = param_decl_set(p->p1, &type);
   if(p->p2){
      ok2 = form_act_params(p->p2, &params2);
      *params = union(params2, singleton(name, type));
      return(ok2 && missing(name, params2));
   }
   else {*params = singleton(name, type); return(TRUE);}
}

char *param_decl_set(Node *p, int *type)
{
   *type = type_const(p->p2);
   return(id(p->p1));
}

int type_const(Node *p)
{  return(p->p1->symbol == INT || p->p1->symbol == INTCONST ? INT :
          (p->p1->symbol == REAL || p->p1->symbol == REALCONST ? REAL : STRING));
}

char *id(Node *p){return(p->lexval.sval);}
```



# Exercise 10

Given a language defined by the following BNF, where each phrase defines a vector and prints an element of it:

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

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

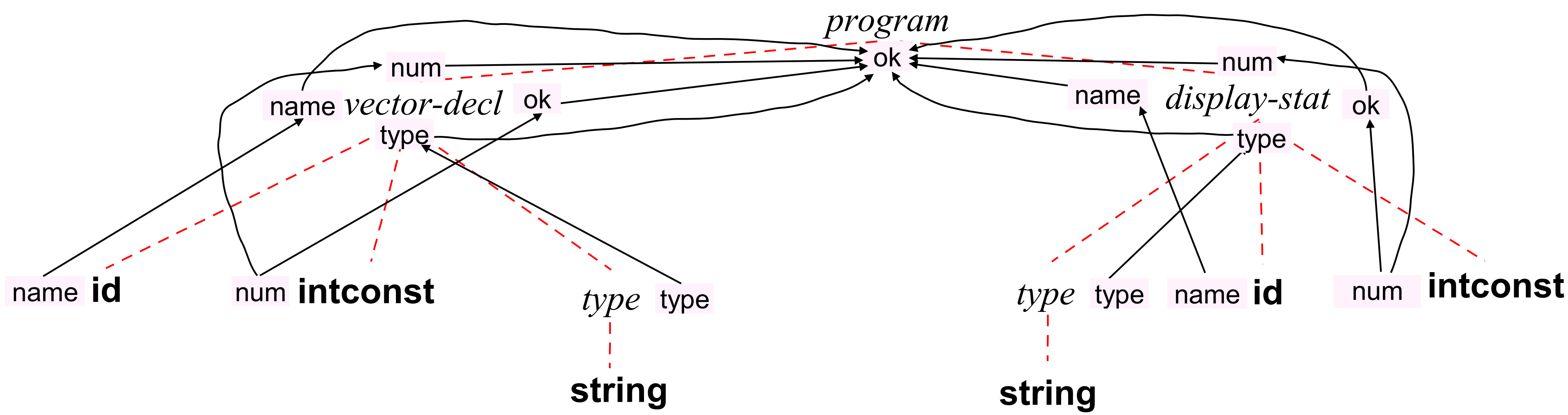
Specify the relevant attribute grammar based on the following semantic constraints:

- The integer constant within the definition is a natural number $n \geq 1$ (denoting range 1..$n$);
- The first argument of **display** shall equal the type of the vector's elements;
- The second argument of **display** shall fulfill the following requirements:
  - The name of the referenced vector equals the name of the defined vector;
  - The integer constant (index) shall be contained in the defined ramge.

# Exercise 10

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

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




A = { ok, name, num, type }

# Exercise 10 (ii)

| *Production* | *Semantic rules* |
|---|---|
| *program* → *vector-decl display-stat* | *program*.ok := *vector-decl*.ok **and** *display_stat*.ok **and** *vector-decl*.name = *display_stat*.name **and** *vector-decl*.type = *display_stat*.type **and** *vector-decl*.num ≥ *display_stat*. num |
| *vector-decl*→ **id : vector [ intconst ] of** *type* | *vector-decl*.name := **id**.name<br>*vector-decl*.type := *type*.type<br>*vector-decl*.num := **intconst**.num<br>*vector-decl*.ok := **intconst**.num ≥ 1 |
| *type*→ **int** | *type*.type := INT |
| *type*→ **string** | *type*.type := STRING |
| *display-stat*→ **display (** *type*, **id [ intconst ] )** | *display-stat*.name := **id**.name<br>*display-stat*.type := *type*.type<br>*display-stat*.num := **intconst**.num<br>*display-stat*.ok := **intconst**.num ≥ 1 |

# Exercise 11

Given a language defined by the following BNF, where each phrase defines a set that is assigned a value:

$program \rightarrow set\text{-}def\ \ set\text{-}assign$
$set\text{-}def \rightarrow$ **def id : set of** $domain$
$domain \rightarrow$ **int** | **string**
$set\text{-}assign \rightarrow$ **id := {** $const\text{-}list$ **}**
$const\text{-}list \rightarrow const\ \ const\text{-}list \mid \varepsilon$
$const \rightarrow$ **intconst** | **strconst**

(examples of phrases)

```
def alfa : set of int
alfa := {3 5 8}
```

```
def beta : set of int
beta := {}
```

```
def S : set of string
S := {"flower" "sun"}
```

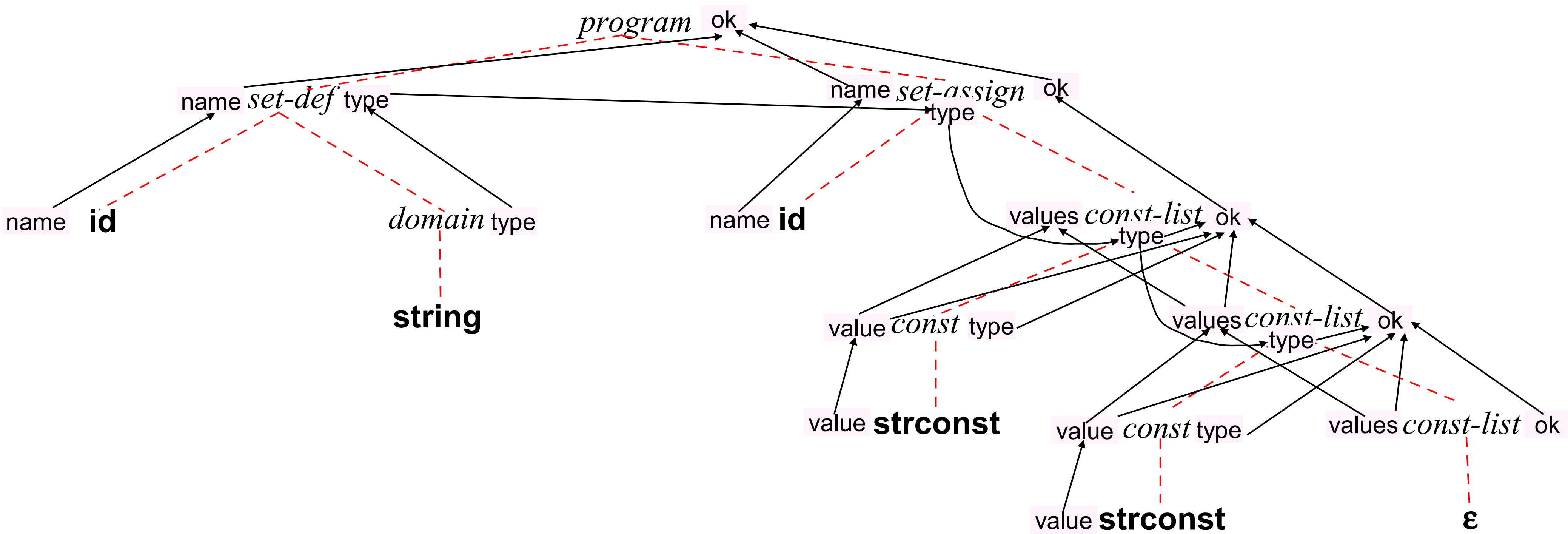
Specify the relevant attribute grammar based on the following semantic constraints:

- The name of the defined set equals the name of the assigned set;
- The type of each atomic constant within the assignment equals the type of the elements of the set;
- Within assignment, no duplicates are allowed.

# Exercise 11

*program* → *set-def* *set-assign*
*set-def* → **def id : set of** *domain*
*domain* → **int** | **string**
*set-assign* → **id := {** *const-list* **}**
*const-list* → *const* *const-list* | **ε**
*const* → **intconst** | **strconst**

```
def S : set of string
S := {"flower" "sun"}
```




A = { ok, name, type, value, values }

# Exercise 11 (ii)

| *Production* | *Semantic rules* |
|---|---|
| *program* → *set-def* *set-assign* | *program*.ok := *set-def*.name = *set-assign*.name **and** *set-assign*.ok<br>*set-assign*.type := *set-def*.type |
| *set-def* → **def id : set of** *domain* | *set-def*.name := **id**.name<br>*set-def*.type := *domain*.type |
| *domain* → **integer** | *domain*.type := INT |
| *domain* → **string** | *domain*.type := STR |
| *set-assign* → **id := {** *const-list* **}** | *set-assign*.name := **id**.name<br>*set-assign*.ok := *const-list*.ok<br>*const-list*.type := *set-assign*.type |
| $const\text{-}list_1$ → *const* **,** $const\text{-}list_2$ | $const\text{-}list_1$.ok := $const\text{-}list_2$.ok **and**<br>*const*.type = $const\text{-}list_1$.type **and**<br>*const*.value $\notin$ $const\text{-}list_2$.values<br>$const\text{-}list_2$.type := $const\text{-}list_1$.type<br>$const\text{-}list_1$.values := $const\text{-}list_2$.values $\cup$ { *const*.value } |
| *const-list* → **ε** | *const-list*.values := {}<br>*const-list*.ok := **true** |
| *const* → **intconst** | *const*.type := INT<br>*const*.value := **intconst**.value |
| *const* → **strconst** | *const*.type := STRING<br>*const*.value := **strconst**.value |

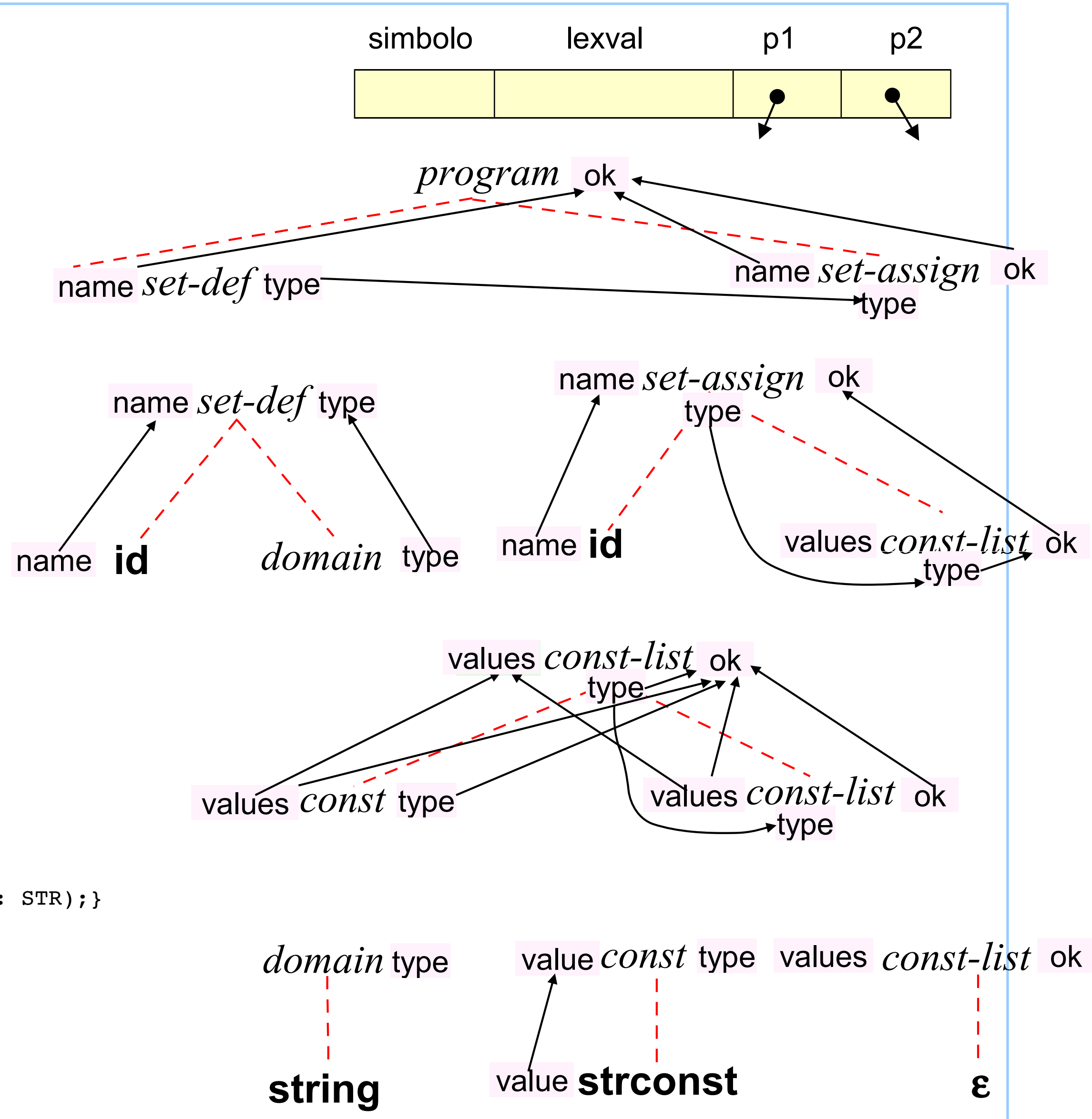
# Exercise 11 (iii)

| symbol | lexval | p1 | p2 | name | type | value | values | ok |
|---|---|---|---|---|---|---|---|---|
| | | • | • | | | | | |

```
sem(Node *p)
{
    switch(p->symbol)
    {
    case PROGRAM: sem(p->p1); p->p2->type = p->p1->type; sem(p->p2);
                  p->ok = p->p1->name == p->p2->name && p->p2->ok;
                  break;
    case SET-DEF: sem(p->p1); sem(p->p2);
                  p->name = p->p1->name; p->type = p->p2->type;
                  break;
    case SET-ASSIGN: sem(p->p1); p->p2->type = p->type; sem(p->p2);
                     p->name = p->p1->name; p->ok = p->p2->ok;
                     break;
    case CONST-LIST: if(p->p1){
                         sem(p->p1); p->p2->type = p->type; sem(p->p2);
                         p->ok = p->p2->ok && p->p1->type == p->type && !member(p->p1->value, p->p2->values);
                         p->values = union(p->p2->values, singleton(p->p1->value));
                     }
                     else {p->values = emptyset(); p->ok = TRUE;}
                     break;
    case DOMAIN: p->type = (p->p1->symbol == INTEGER ? INT : STR);
                 break;
    case CONST: p->type = (p->p1->symbol == INTCONST ? INT : STR);
                p->value = p->p1->value;
                break;
    case INTCONST:
    case STRCONST: p->value = p->p1->lexval;
                   break;
    case ID: p->name = p->lexval.sval;
             break;
    }
}
```

# Exercise 11 (iv)

```
Bool program(Node *p)
{  char *name1, *name2; int type; Bool ok2;

   name1 = set_def(p->p1, &type);
   ok2 = set_assign(p->p2, type, &name2);
   return(name1 == name2 && ok2);
}

char *set_def(Node *p, int *type)
{  *type = domain(p->p2);
   return(id(p->p1));
}

Bool set_assign(Node *p, int type, char **name)
{  Lexval values[];

   *name = id(p->p1);
   return(const_list(p->p2, type, &values));
}

Bool const_list(Node *p, int type, Lexval *values[])
{  Lexval value, values2[]; int type1; Bool ok, ok2;

   if(p->p1){
      type1 = const(p->p1, &value);
      ok2 = const_list(p->p2, type, &values2);
      ok = ok2 && !member(value, values2);
      *values = union(values2, singleton(value));
      return(ok);
   }
   else {*values = emptyset(); return(TRUE);
}

int domain(Node *p){return(p->p1->symbol == INTEGER ? INT : STR);}

int const(Node *p, Lexval *value)
{
   *value = p->p1->lexval;
   return(p->p1->symbol == INTCONST ? INT : STR);
}

char *id(Node *p){return(p->lexval.sval);}
```



# Exercise 12

Specify the (extended) attribute grammar relevant to the following BNF:

*program* → *stat-list*
*stat-list* → *stat* *stat-list* | *stat*
*stat* → *def-stat* | *assign-stat*
*def-stat* → *type* **id**
*type* → **int** | **string** | **bool**
*assign-stat* → **id :=** *cond-expr*
*cond-expr* → **(** *predicate* **? id : id )**
*predicate* → **id = id**

```
int a
int b
int c
c := (a = b ? a : b)
string s
```

based on the following semantic constraints:

- Each variable can be declared only once,
- Each referenced variable shall have been defined,
- Within conditional expression, the last two variables share the same type,
- The assigned variable has the same type of the conditional expression,
- Within predicate, the two compared variables share the same type,

and the following requirements:

- In case of semantic error, function `error()` is called, which terminates the analysis,
- A symbol table is used, for inserting and looking for variables, by means of the following functions:
  `void insert(name, type)`
  `Type lookup(name)`
- Function `lookup(name)` returns either the variable's type or `nil` (if the variable is not cataloged).

# Exercise 12

*program* → *stat-list*
*stat-list* → *stat*  *stat-list* | *stat*
*stat* → *def-stat* | *assign-stat*
*def-stat* → *type* **id**
*type* → **int** | **string** | **bool**
*assign-stat* → **id :=** *cond-expr*
*cond-expr* → **(** *predicate* **? id : id )**
*predicate* → **id = id**

```
int a
int b
int c
c := (a = b ? a : b)
string s
```

| Production | Semantic rules |
|---|---|
| *def-stat* → *type* **id** | **if** lookup(**id**.name) == **nil then** insert(**id**.name, *type*.type) **else** error(); |
| *type* → **int** | *type*.type := INT |
| *type* → **string** | *type*.type := STRING |
| *type* → **bool** | *type*.type := BOOL |
| *assign-stat* → **id :=** *cond-expr* | **if** (type = lookup(**id**.name)) == **nil or** type != *cond-expr*.type **then** error(); |
| *cond-expr* → **(** *predicate* **?** $\mathbf{id}_1$ **:** $\mathbf{id}_2$ **)** | **if** (t1 = lookup($\mathbf{id}_1$.name)) == **nil or** (t2 = lookup($\mathbf{id}_2$.name)) == **nil or** (t1 != t2) **then** error() **else** *cond-expr*.type = t1; |
| *predicate* → $\mathbf{id}_1$ **=** $\mathbf{id}_2$ | **if** (t1 = lookup($\mathbf{id}_1$.name)) == **nil or** (t2 = lookup($\mathbf{id}_2$.name)) == **nil or** (t1 != t2) **then** error(); |

A = { name, type } + ST

# Exercise 13

Specify the attribute grammar relevant to 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**

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

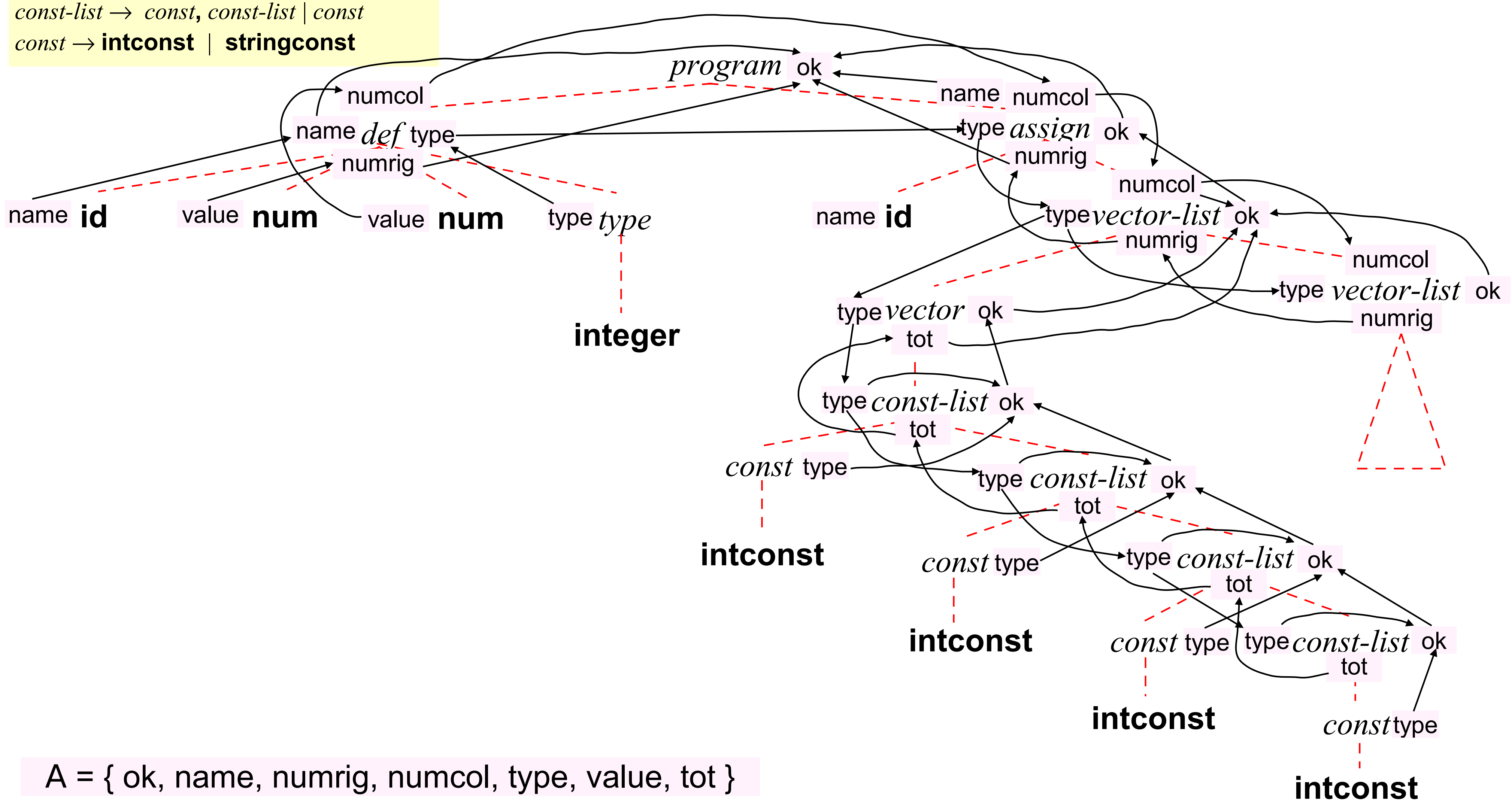
based on the following semantic constraints:

- The name of the defined matrix equals the name of the assigned matrix;
- The RHS of the assignment shall conform to size and type of the defined matrix.

# Exercise 13

*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**

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




A = { ok, name, numrig, numcol, type, value, tot }

# Exercise 13 (ii)

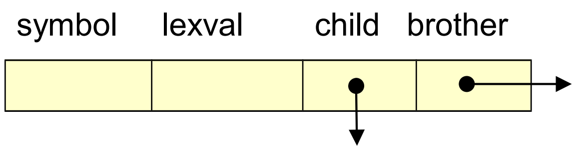
| Production | Semantic rules |
|---|---|
| *program* → *def assign* | *program*.ok := *def*.name = *assign*.name **and** *def*.numrig = *assign*.numrig **and** *assign*.ok<br>*assign*.numcol := *def*.numcol<br>*assign*.type := *def*.type |
| *def* → **id** : **matrix** [ $\mathbf{num}_1$, $\mathbf{num}_2$ ] **of** *type* | *def*.name = **id**.name<br>*def*.numrig = $\mathbf{num}_1$.value<br>*def*.numcol = $\mathbf{num}_2$.value<br>*def*.type := *type*.type |
| *type* → **integer** | *type*.type := INT |
| *type* → **string** | *type*.type := STR |
| *assign* → **id** := [ *vector-list* ] | *assign*.name := **id**.name<br>*assign*.ok := *vector-list*.ok<br>*assign*.numrig := *vector-list*.numrig<br>*vector-list*.numcol := *assign*.numcol<br>*vector-list*.type := *assign*.type |
| $vector\text{-}list_1$ → *vector* , $vector\text{-}list_2$ | $vector\text{-}list_1$.ok := $vector\text{-}list_2$. ok **and** *vector*.ok **and** *vector*.tot = $vector\text{-}list_1$.numcol<br>$vector\text{-}list_1$.numrig := $vector\text{-}list_2$.numrig + 1<br>$vector\text{-}list_2$.numcol := $vector\text{-}list_1$.numcol<br>$vector\text{-}list_2$.type := $vector\text{-}list_1$.type<br>*vector*.type := $vector\text{-}list_1$.type |
| *vector-list* → *vector* | *vector-list*.ok := *vector*.ok **and** *vector*.tot = *vector-list*.numcol<br>*vector-list*.numrig := 1<br>*vector*.type := *vector-list*.type |
| *vector* → [ *const-list* ] | *vector*.ok := *const-list*.ok<br>*vector*.tot := *const-list*.tot<br>*const-list*.type := *vector*.type |
| $const\text{-}list_1$ → *const*, $const\text{-}list_2$ | $const\text{-}list_1$.ok := $const\text{-}list_2$.ok **and** *const*.type = $const\text{-}list_1$.type<br>$const\text{-}list_1$.tot := $const\text{-}list_2$.tot + 1<br>$const\text{-}list_2$.type := $const\text{-}list_1$.type |
| *const-list* → *const* | *const-list*.ok := *const*.type = *const-list*.type<br>*const-list*.tot := 1 |
| *const* → **intconst** | *const*.type := INT |
| *const* → **strconst** | *const*.type := STR |

# Exercise 13 (iii)

| symbol | lexval | child | brother | name | type | value | numrig | numcol | tot | ok |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | |

```
sem(Node *p)
{   switch(p->symbol)
    {
    case PROGRAM: p1 = p->child; p2 = p1->brother;
                  sem(p1); p2->numcol = p1->numcol; p2->type = p1->type; sem(p2);
                  p->ok = p1->name == p2->name && p1->numrig == p2->numrig && p2->ok;
                  break;
    case DEF: p1 = p->child; p2 = p1->brother; p3 = p2->brother; p4 = p3->brother;
              sem(p1); sem(p2); sem(p3); sem(p4);
              p->name = p1->name; p->numrig = p2->value; p->numcol = p3->value; p->type = p4->type;
              break;
    case ASSIGN: p1 = p->child; p2 = p1->brother;
                 sem(p1); p2->numcol = p->numcol; p2->type = p->type; sem(p2);
                 p->name = p1->name; p->ok = p2->ok; p->numrig = p2->numrig;
                 break;
    case VECTOR-LIST: p1 = p->child; p2 = p1->brother;
                      p1->type = p->type; sem(p1);
                      if(p2){
                         p2->numcol = p->numcol; p2->type = p->type; sem(p2);
                         p->ok = p2->ok && p1->ok && p1->tot == p->numcol;
                         p->numrig = p2->numrig + 1;
                      }
                      else {p->ok = p1->ok && p1->tot == p->numcol; p->numrig = 1;}
                      break;
    case VECTOR: p->child->type = p->type; sem(p->child);
                 p->ok = p->child->ok; p->tot = p->child->tot;
                 break;
    case CONST-LIST: p1 = p->child; p2 = p1->brother; sem(p1);
                     if(p2){
                         p2->type = p->type; sem(p2);
                         p->ok = p2->ok && p1->type == p->type;
                         p->tot = p2->tot + 1;
                     }
                     else {p->ok = p1->type == p->type; p->tot = 1;}
                     break;

    case TYPE: p->type = (p->child->symbol == INTEGER ? INT : STR); break;
    case CONST: p->type = (p->child->symbol == INTCONST ? INT : STR); break;
    case NUM: p->value = p->lexval.ival; break;
    case ID: p->name = p->lexval.sval; break;
    }
}
```

# Exercise 13 (iv)




```
Bool program(Node *p)
{  char *name1, *name2; int numcol, numrig1, numrig2, type; Bool ok2;
   Node *p1 = p->child, *p2 = p1->brother;

   name1 = def(p1, &numrig1, &numcol, &type);
   ok2 = assign(p2, type, numcol, &numrig2, &name2);
   return(name1 == name2 && numrig1 == numrig2 && ok2);
}

char *def(Node *p, int *numrig, int *numcol, int *type)
{  p1 = p->child; p2 = p1->brother; p3 = p2->brother; p4 = p3->brother;
   *numrig = p2->lexval.ival; *numcol = f3->lexval.ival; *type = get_type(p4);
   return(p1->lexval.sval);
}

Bool assign(Node *p, int type, int numcol, int *numrig, char **name)
{  *name = id(p->child);
   return(vector_list(p->child->brother, type, numcol, numrig));
}

Bool vector_list(Node *p, int type, int numcol, int *numrig)
{  int tot, numrig2; Bool ok1, ok2; Node *p1 = p->child, *p2 = p1->brother;

   ok1 = vector(p1, type, &tot);
   if(p2){ok2 = vector_list(p2, type, numcol, &numrig2); *numrig = numrig2 + 1; return(ok1 && ok2 && tot == numcol);}
   else {*numrig = 1; return(ok1 && tot == numcol);}
}

Bool vector(Node *p, int type, int *tot){return(const_list(p->child, type, tot);}

Bool const_list(Node *p, int type, int *tot)
{  int type1, tot2; Bool ok2; Node *p1 = p->child, *p2 = p1->brother;

   type1 = const(p1);
   if(p2) {const_list(p2, type, &tot2); *tot = tot2 = 1; return(ok2 && type1 == type);}
   else {*tot = 1; return(type1 == type);}
}

int const(Node *p){return(p->child->symbol == INTCONST ? INT : STR);}
int get_type(Node *p){return(p->p1->symbol == INTEGER ? INT : STR);}
char *id(Node *p){return(p->lexval->sval);}
```

# Exercise 14

Specify the (extended) attribute grammar relevant to the following BNF:

*program* → *class-def-list*
*class-def-list* → *class-def* *class-def-list* | *class-def*
*class-def* → **class id** *inheritance*
*inheritance* → **inherits** *id-list* | ε
*id-list* → **id ,** *id-list* **| id**

```
class A
class B
class C inherits A, B
class D
```

based on the following semantic constraints:

- Each class can be defined only once,
- Within inheritance, superclasses shall have been defined,
- Within inheritance, names of superclasses are unique,

and the following requirements:

- In case of semantic error, function `error()` is called, which terminates the analysis;
- A symbol table is used, to insert and look for classes, by means of the following functions:
  `void insert(class, superclasses)`
  `names lookup(class)`
- Function `lookup(class)` returns either the (possibly empty) set of superclasses of `class` or `nil` (if `class` is not cataloged).

# Exercise 14

*program* → *class-def-list*
*class-def-list* → *class-def class-def-list* | *class-def*
*class-def* → **class id** *inheritance*
*inheritance* → **inherits** *id-list* | **ε**
*id-list* → **id ,** *id-list* **| id**

```
class A
class B
class C inherits A, B
class D
```

| *Production* | *Semantic rules* |
|---|---|
| *class-def* → **class id** *inheritance* | **if** lookup(**id**.name) == **nil then**<br>insert(**id**.name, *inheritance*.names)<br>**else** error(); |
| *inheritance* → **inherits** *id-list* | *inheritance*.names := *id-list*.names |
| *inheritance* → **ε** | *inheritance*.names := ∅ |
| $id\text{-}list_1$ → **id ,** $id\text{-}list_2$ | **if** lookup(**id**.name) ≠ **nil and id**.name ∉ $id\text{-}list_2$.names **then**<br>$id\text{-}list_1$.names := $id\text{-}list_2$.names ∪ { **id**.name}<br>**else** error(); |
| *id-list* → **id** | **if** lookup(**id**.name) ≠ **nil then**<br>*id-list*.names := { **id**.name}<br>**else** error(); |

A = { name, names } + ST

# Exercise 15

Specify the (extended) attribute grammar relevant to 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
```

based on the following semantic constraints:

- Names of attributes are unique,
- The operand of the `update` is the defined table,
- The update attribute belongs to the table,
- Each identifier within the expression is an attribute of the table,
- The type of the update attribute equals the type of the update expression,

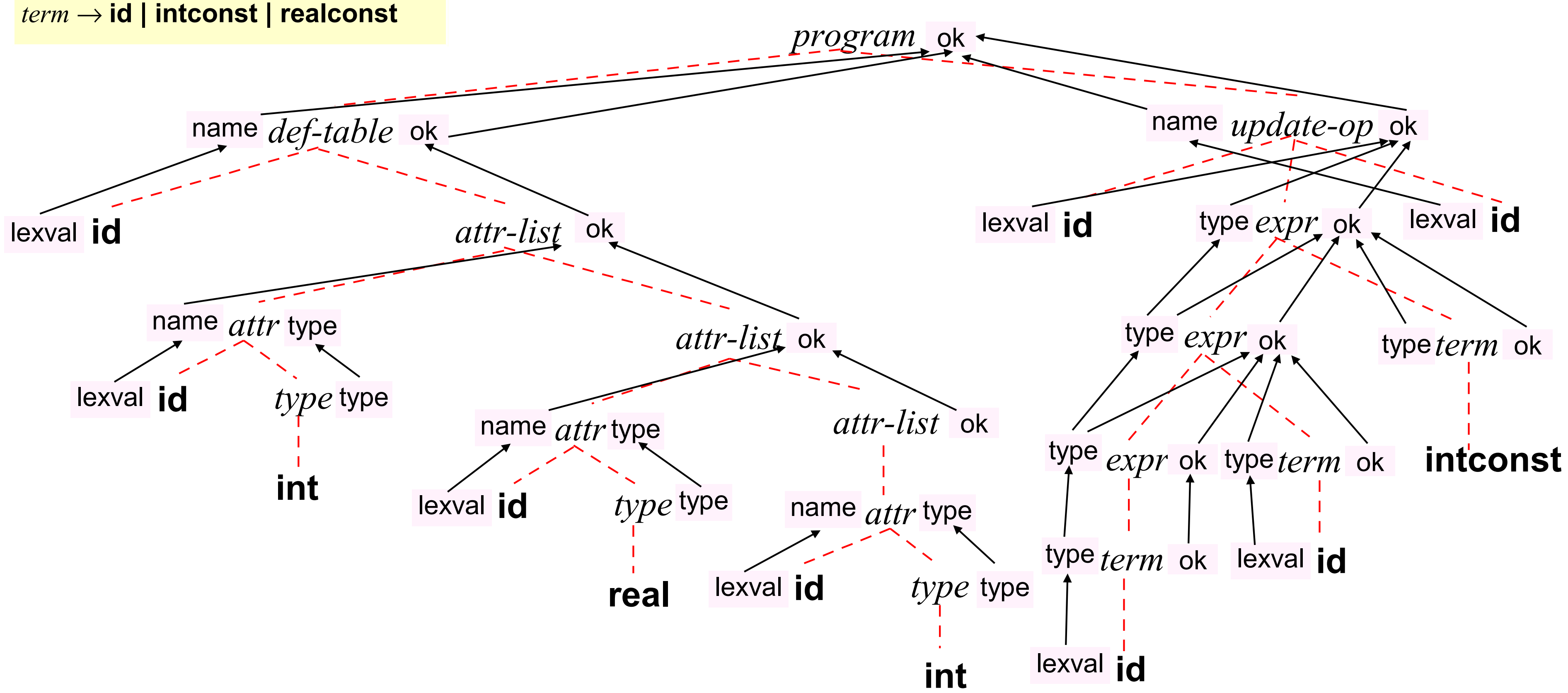
and the following requirements:

- The set of semantic attributes is { ok, name, type },
- A symbol table is used to catalog table attributes by means of the following functions:
```
void insert(name, type)
Type lookup(name)
```
- Function `lookup(name)` returns the attribute type (INT, REAL) if the attribute is cataloged, otherwise it returns NIL (if the attribute is not cataloged),
- A possible intermediate semantic error does <u>not</u> end the semantic analysis.

# Exercise 15

*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
```




A = { ok, name, type }

# Exercise 15 (ii)

| *Production* | *Semantic rules* |
|---|---|
| $program \rightarrow def\text{-}table\ \ update\text{-}op$ | $program$.ok := $def\text{-}table$.ok **and** $update\text{-}op$.ok **and** $def\text{-}table$.name = $update\text{-}op$.name |
| $def\text{-}table \rightarrow$ **table id (** $attr\text{-}list$ **)** | $def\text{-}table$.name := **id**.lexval; $def\text{-}table$.ok := $attr\text{-}list$.ok |
| $attr\text{-}list_1 \rightarrow attr$ **,** $attr\text{-}list_2$ | $attr\text{-}list_1$.ok := $attr\text{-}list_2$.ok **and** `lookup`($attr$.name) = NIL; `insert`($attr$.name, $attr$.type) |
| $attr\text{-}list \rightarrow\ attr$ | $attr\text{-}list$.ok := **true**; `insert`($attr$.name, $attr$.type) |
| $attr \rightarrow$ **id :** $type$ | $attr$.name := **id**.lexval; $attr$.type := $type$.type |
| $type \rightarrow$ **int** | $type$.type = INT |
| $type \rightarrow$ **real** | $type$.type = REAL |
| $update\text{-}op \rightarrow$ **update [ id**$_1$ **=** $expr$ **] id**$_2$ | $update\text{-}op$.name := **id**$_2$.lexval; $update\text{-}op$.ok := $expr$.ok **and** `lookup`(**id**$_1$.lexval) = $expr$.type |
| $expr_1 \rightarrow expr_2$ **+** $term$ | $expr_1$.ok := $expr_2$.ok **and** $term$.ok **and** $expr_2$.type = $term$.type; $expr_1$.type := **if** $expr_1$.ok **then** $expr_2$.type **else** ERROR |
| $expr \rightarrow\ term$ | $expr$.ok := $term$.ok; $expr$.type := $term$.type |
| $term \rightarrow$ **id** | $term$.type := `lookup`(**id**.lexval); $term$.ok := $term$.type $\neq$ NIL |
| $term \rightarrow$ **intconst** | $term$.ok := **true**; $term$.type := INT |
| $term \rightarrow$ **realconst** | $term$.ok := **true**; $term$.type := REAL |

# Exercise 16

specify the attribute grammar relevant to 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.
```

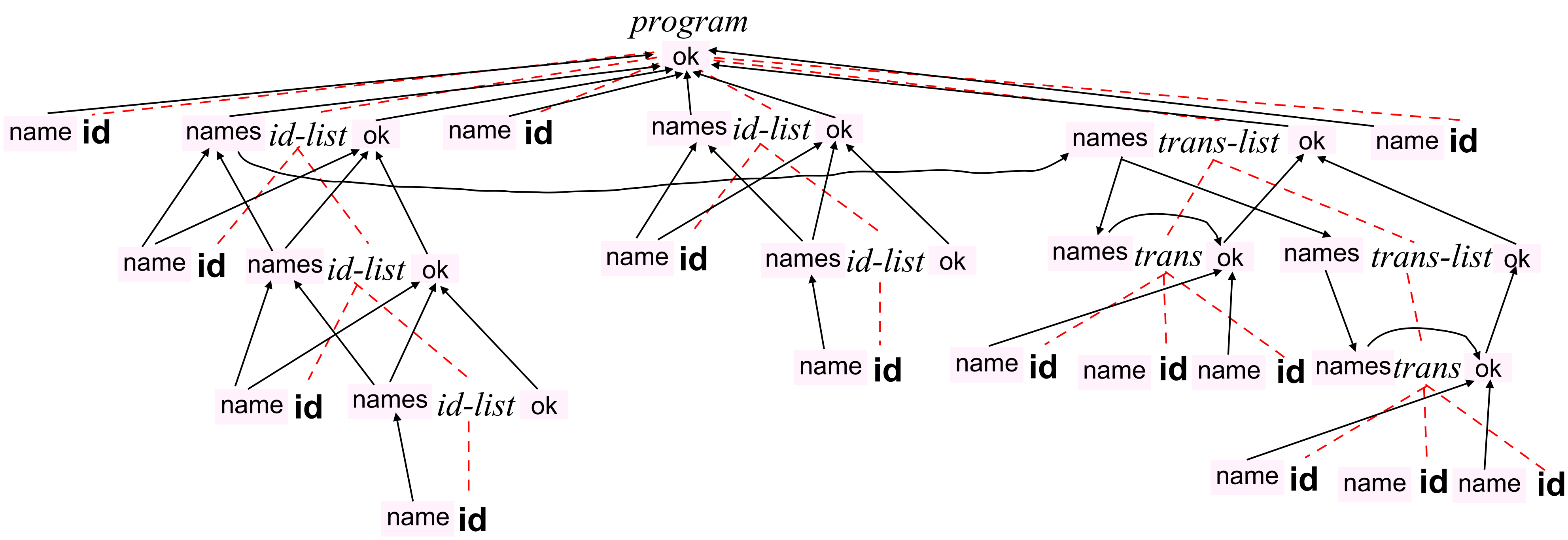
based on the following semantic constraints:

- The automaton name ending the specification equals the name declared at the beginning;
- State names are unique;
- The initial state belongs to the automaton states;
- Each final state belongs to the automaton states;
- For each transition, both states belong to the automaton states.

# Exercise 16

*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.
```




A = { ok, name, names }

# Exercise 16 (ii)

| *Production* | *Semantic rules* |
|---|---|
| program $\rightarrow$ **automaton id**$_1$ **is** **states** id-list$_1$ **;** **initial id**$_2$ **;** **finals** id-list$_2$ **;** **transitions** trans-list **;** **end id**$_3$ **.** | program.ok := **id**$_1$.name = **id**$_3$.name **and** id-list$_1$.ok **and** **id**$_2$.name $\in$ id-list$_1$.names **and** id-list$_2$.ok **and** id-list$_2$.names $\subseteq$ id-list$_1$.names **and** trans-list.ok;<br>trans-list.names := id-list$_1$.names |
| id-list$_1$ $\rightarrow$ **id** id-list$_2$ | id-list$_1$.ok := id-list$_2$.ok **and** **id**.name $\notin$ id-list$_2$.names;<br>id-list$_1$.names := id-list$_2$.names $\cup$ { **id**.name } |
| id-list $\rightarrow$ **id** | id-list.ok := TRUE;<br>id-list.names := { **id**.name } |
| trans-list$_1$ $\rightarrow$ trans trans-list$_2$ | trans.names := trans-list$_1$.names;<br>trans-list$_2$.names := trans-list$_1$.names;<br>trans-list$_1$.ok := trans.ok **and** trans-list$_2$.ok |
| trans-list $\rightarrow$ trans | trans.names := trans-list.names;<br>trans-list.ok := trans.ok |
| trans $\rightarrow$ **( id**$_1$**, id**$_2$**, id**$_3$ **)** | trans.ok := **id**$_1$.name $\in$ trans.names **and** **id**$_3$.name $\in$ trans.names |

# Exercise 17

Specify the (extended) attribute grammar relevant to the following BNF (where symbol '**⊗**' denotes the join operation):

program → stat-list
stat-list → stat stat-list | stat
stat → def | assign
def → **id : (** attr-list **)**
attr-list → attr **,** attr-list | attr
attr → **id :** type
type → **int** | **real** | **string**
assign → **id := id ⊗ id**

```
R: (a: int, b: string, c: real)
S: (x: real, y: int)
T := R ⊗ S
```

based on the following semantic constraints:

- Names of tables are unique,
- For each table, attribute names are unique,
- In assignment, the assigned table shall neither have been defined nor assigned previously,
- The two tables of join shall have been cataloged and cannot share attribute names,

and the following requirements:

- The set of semantic attributes is { name, type, schema },
- Table schema is a list of pairs (name, type), each defining an attribute,
- A symbol table is used, where cataloging table schemas by means of the following functions:
  `void insert(tabname, schema)`
  `Schema lookup(tabname)`
- Function `lookup(tabname)` returns the schema of table `tabname` if the table is cataloged, otherwise it returns the empty list (if the table is not cataloged),
- In assignment, the assigned table is cataloged, whose schema is (by definition) the concatenation of the two schemas of the operand tables,
- In case of semantic error, function `error()` is called, which terminates the analysis.

# Exercise 17

| *Production* | *Semantic rules* |
|---|---|
| *def* → **id : (** *attr-list* **)** | `if lookup(`**id**.name) == [ ] `then insert(`**id**.name, *attr-list*.schema) `else error`(); |
| $attr\text{-}list_1$ → *attr* **,** $attr\text{-}list_2$ | `if` *attr*.name ∈ `extract_names`($attr\text{-}list_2$.schema) `then error`()<br>**else** $attr\text{-}list_1$.schema = [(*attr*.name, *attr*.type)] ∪ $attr\text{-}list_2$.schema; |
| *attr-list* → *attr* | *attr-list*.schema = [(*attr*.name, *attr*.type)]; |
| *attr* → **id :** *type* | *attr*.name = **id**.name;<br>*attr*.type = *type*.type; |
| *type* → **int** | *type*.type = INT; |
| *type* → **real** | *type*.type = REAL; |
| *type* → **string** | *type*.type = STRING; |
| *assign* → $\mathbf{id}_1$ **:=** $\mathbf{id}_2$ ⊗ $\mathbf{id}_3$ | `if lookup(`$\mathbf{id}_1$.name) ≠ [ ] `or`<br>(`s2` = `lookup`($\mathbf{id}_2$.name)) == [ ] `or` (`s3` = `lookup`($\mathbf{id}_3$.name)) == [ ] `or`<br>(`extract_names(s2)` ∩ `extract_names(s3)`) ≠ [ ] `then error`()<br>`else insert`($\mathbf{id}_1$.name, `s2` ∪ `s3`); |

# Exercise 18

A language is given, where each phrase declares two tables and a natural join, defined by the following BNF:

*program* → *def* *def* *natjoin*
*def* → **id :** **(** *attr-list* **)**
*attr-list* → *attr* **,** *attr-list* | *attr*
*attr* → **id :** *type*
*type* → **int** | **real** | **string**
*natjoin* → **id** **njoin** **id**

```
alpha: (a: int, b: real, c: string)
beta: (x: string, a: int, y: real, b: real)
beta njoin alpha
```

a) Specify the attribute grammar based on the following semantic constraints:

- Within each table, attribute names are unique,
- The two tables have different names,
- The two operand tables of the natural join are those defined (possibly in different order),
- If the two tables share homonymous attributes, each pair of homonymous attributes share the same type,

and the following requirements:

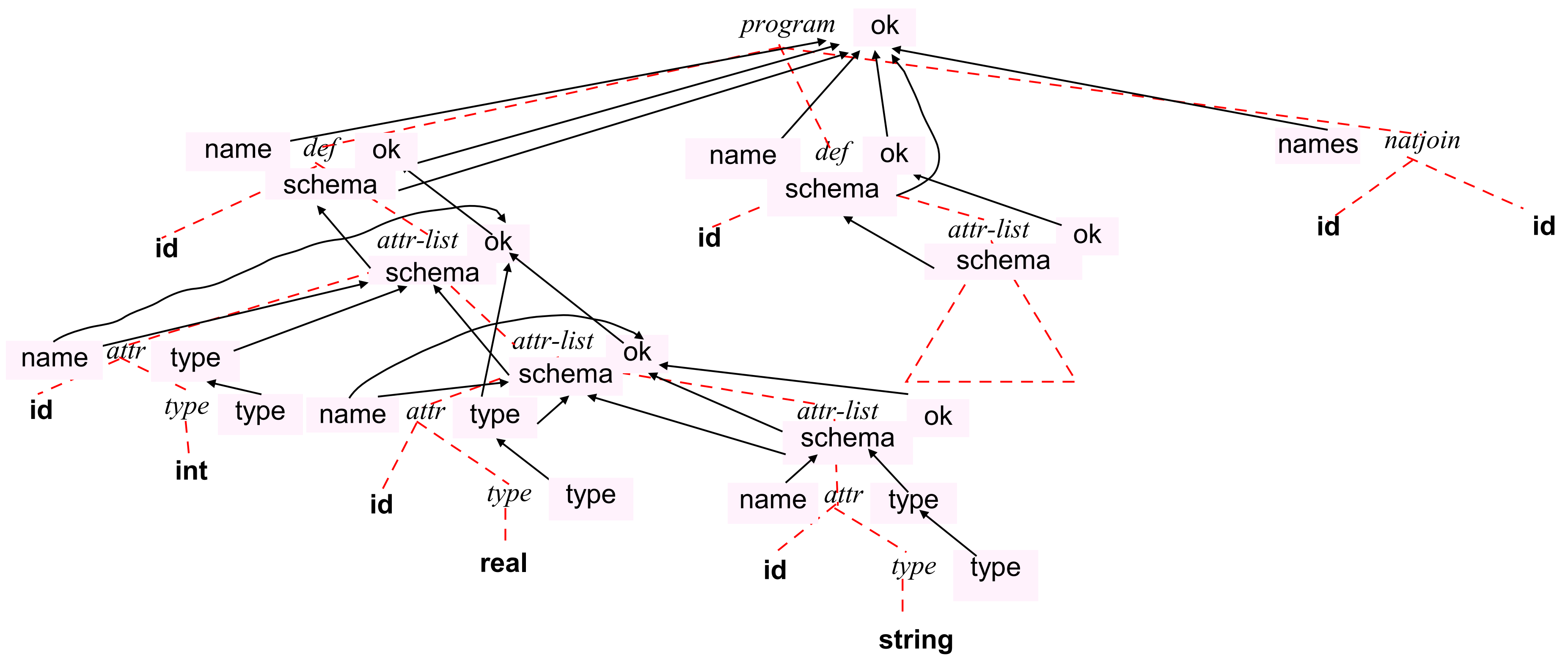
- The set of semantic attributes is { ok, schema, type, name, names },
- Attribute schema is a list of pairs (name, type), each defining an attribute.

b) Assuming the semantic attributes not stored within the abstract tree, codify the semantic procedure `Bool program(PNODE p)` associated with the root, assuming nodes with the following structure:

| type | lexval | p1 | p2 | p3 |
|---|---|---|---|---|

where `type` ∈ { `PROGRAM`, `DEF`, `NATJOIN`, `ID`, `ATTR-LIST`, `ATTR`, `TYPE`, `INT`, `REAL`, `STRING` }

# Exercise 18


program ok
name def ok
schema
id
attr-list ok
schema
name attr type
id
type type
int
attr-list ok
schema
name attr type
id
type type
real
attr-list ok
schema
name attr type
id
type type
string
name def ok
schema
id
attr-list ok
schema
names natjoin
id
id

# Esercizio 18 (ii)

| *Production* | *Semantic rules* |
|---|---|
| $program \rightarrow def_1 \; def_2 \; natjoin$ | $program$.ok = $def_1$.ok **and** $def_2$.ok **and**<br>$def_1$.name ≠ $def_2$.name **and**<br>{ $def_1$.name, $def_2$.name} == $natjoin$.names **and**<br>∀(n1,t1) ∈ $def_1$.schema, ∀(n2,t2)∈ $def_2$.schema, n1 == n2 (t1 == t2); |
| $def \rightarrow$ **id :** **(** $attr\text{-}list$ **)** | $def$.name = **id**.lexval;<br>$def$.schema = $attr\text{-}list$.schema;<br>$def$.ok = $attr\text{-}list$.ok; |
| $attr\text{-}list_1 \rightarrow attr$ **,** $attr\text{-}list_2$ | $attr\text{-}list_1$.ok = $attr\text{-}list_2$.ok **and** $attr$.name ∉ `extract_names`($attr\text{-}list_2$.schema);<br>$attr\text{-}list_1$.schema = [($attr$.name, $attr$.type)] ∪ $attr\text{-}list_2$.schema; |
| $attr\text{-}list \rightarrow attr$ | $attr\text{-}list_1$.ok = **true**;<br>$attr\text{-}list$.schema = [($attr$.name, $attr$.type)]; |
| $attr \rightarrow$ **id :** $type$ | $attr$.name = **id**.lexval;<br>$attr$.type = $type$.type; |
| $type \rightarrow$ **int** | $type$.type = INT; |
| $type \rightarrow$ **real** | $type$.type = REAL; |
| $type \rightarrow$ **string** | $type$.type = STRING; |
| $natjoin \rightarrow$ **id**$_1$ **njoin** **id**$_2$ | $natjoin$.names = { **id**$_1$.lexval, **id**$_2$.lexval }; |

# Exercise 18 (iii)

```
Bool program(PNODE p)
{
    Bool ok1, ok2, ok_homonymous;
    list(char *name, Type type) schema1, schema2;
    set(char*) names;

    ok1 = def(p->p1, &name1, &schema1);
    ok2 = def(p->p2, &name2, &schema2);
    names = natjoin(p->p3);
    ok_homonymous = TRUE;
    for(i1=0; i1<length(schema1); i1++)
        for(i2=0; i2<length(schema2); i2++)
            if(schema1[i1].name == schema2[i2].name &&
               schema1[i1].type != schema2[i2].type)
                   ok_homonymous = FALSE;
    return(ok1 && ok2 &&
           name1 != name2 &&
           set(name1, name2) == names &&
           ok_homonymous);
}
```

# Exercise 19

A language is given, where each phrase specifies a list of statements on tables. Each statement either defines or assigns a table. The assignment expression involves a relational operator and two operand tables. Operators include **`union`** (set-theoretic union) , **`inter`** (set-theoretic intersection), and **`join`** (Cartesian product).

*program* → *stat-list*
*stat-list* → *stat* *stat-list* | *stat*
*stat* → *def-stat* | *assign-stat*
*def-stat* → **id :** **(** *attr-list* )
*attr-list* → *attr* **,** *attr-list* | *attr*
*attr* → **id :** *type*
*type* → **int** | **real** | **string**
*assign-stat* → **id := id** *operator* **id**
*operator* → **union** | **inter** | **join**

```
T1: (a: int, b: real)
T2: (a: int, b: real)
R := T1 union T2
S: (x: int, y: string)
T := R join S
```

Specify the attribute grammar based on <u>the following semantic constraints</u>:

- Table names are unique,
- Within a table, attribute names are unique,
- In assignment, the assigned table cannot have been defined or assigned previously;
- Operand tables in the RHS of assignment shall have been either defined or assigned previously,
- In **`union`**, the two tables share an identical schema (in terms of names and types of attributes), and the assigned table is defined by that common schema,
- In **`inter`**, the two tables share the same signature (attribute types), and the assigned table is defined with the schema of the first operand,
- In **`join`**, the two tables do not share homonymous attributes, and the assigned table is defined with the schema resulting from the concatenation of the schemas of the two operands,

<u>and the following requirements</u>:

- The set of semantic attributes is { name, type, schema, operator },
- Attribute schema is a list of pairs (name, type), each defining an attribute,
- A symbol table is used, where table schemas are cataloged by means of the following functions:
```
void insert(tabname, schema)
Schema lookup(tabname)
```
- Function `lookup(tabname)` returns the schema of table `tabname` if the table is cataloged,otherwise it returns the empty list (if the table is not cataloged),
- In assignment, the assigned table shall be cataloged,
- In case of semantic error, function `error()` is called, which terminates the analysis.

# Exercise 19

| *Production* | *Semantic rules* |
|---|---|
| *def-stat* → **id** **:** **(** *attr-list* **)** | **if** lookup(**id**.name) == [] **then**<br>insert(**id**.name, *attr-list*.schema)<br>**else**<br>error()<br>**end-if**; |
| $\textit{attr-list}_1$ → *attr* **,** $\textit{attr-list}_2$ | **if** *attr*.name ∈ get_names($\textit{attr-list}_2$.schema) **then**<br>error()<br>**else**<br>$\textit{attr-list}_1$.schema = [(*attr*.name, *attr*.type)] ∪ $\textit{attr-list}_2$.schema<br>**end-if**; |
| *attr-list* → *attr* | *attr-list*.schema = [(*attr*.name, *attr*.type)]; |
| *attr* → **id** **:** *type* | *attr*.name = **id**.name;<br>*attr*.type = *type*.type; |
| *type* → **int** | *type*.type = INT; |
| *type* → **real** | *type*.type = REAL; |
| *type* → **string** | *type*.type = STRING; |
| *assign-stat* → $\mathbf{id}_1$ **:=** $\mathbf{id}_2$ *operator* $\mathbf{id}_3$ | **if** lookup($\mathbf{id}_1$.name) ≠ [ ] **or**<br>(s2 = lookup($\mathbf{id}_2$.name)) == [ ] **or**<br>(s3 = lookup($\mathbf{id}_3$.name)) == [ ] **or**<br>(*operator*.operator == UNION **and** s2.schema ≠ s3.schema) **or**<br>(*operator*.operator == INTER **and** get_sign(s2) ≠ get_sign(s3)) **or**<br>(*operator*.operator == JOIN **and**<br>get_names(s2) ∩ get_names(s3) ≠ ∅ ) **then**<br>error()<br>**else**<br>**if** *operator*.operator = UNION **or** *operator*.operator = INTER **then**<br>insert($\mathbf{id}_1$.name, s2)<br>**else**<br>insert($\mathbf{id}_1$.name, s2 ∪ s3)<br>**end-if**<br>**end-if**; |
| *operator* → **union** | *operator*.operator = UNION; |
| *operator* → **inter** | *operator*.operator = INTER; |
| *operator* → **join** | *operator*.operator = JOIN; |

# Exercise 20

Consider the following fragment of BNF:

*assign-list* → *assign* *assign-list* | *assign*
*assign* → *pathname* **:=** *pathname* **;**
*pathname* → *pathname* **. id** | **id**

```
r.a := a;
r.b := s.b.d.f;
a := s.b.c;
```

Specify the (extended) attribute grammar for the productions relevant to the given BNF fragment, based on the following semantic constraints:

- Each pathname (LHS or RHS of assignment) shall reference either a variable or an attribute (possibly nested) of a record;
- The two involved pathnames within an assignment shall reference elements with same structure

and the following requirements:

- The set of semantic attributes is { name, root };
- Attribute root is the root of the type tree;
- A symbol table is used, providing the following functions (not to be implemented):

  `PNODE get_tree(char* varname)`: returns the root of the structure tree for variable `varname` if included in the symbol table, otherwise it returns `NULL`.

  `PNODE get_subtree(PNODE *root, char* attrname)`: returns the root of the sub-tree of the structure of attribute (at first level) `attrname` if defined within the record tree identified by `root`, otherwise it returns `NULL`.

  `Boolean equal(PNODE *root1, PNODO *root2)`: returns `TRUE` if the two trees identified by `root1` and `root2` share the same structure, otherwise it returns `FALSE`.
- In case of semantic error, function `error()` is called, which terminates the analysis.

# Exercise 20

| *Production* | *Semantic rules* |
|---|---|
| $assign\text{-}list_1 \rightarrow assign\ \ assign\text{-}list_2$ | |
| $assign\text{-}list \rightarrow assign$ | |
| $assign \rightarrow pathname_1$ **:=** $pathname_2$ **;** | **if not** `equal`($pathname_1$.root, $pathname_2$.root) **then** `error`(); |
| $pathname_1 \rightarrow pathname_2$ **. id** | **if** (`s` = `get_subtree`($pathname_2$.root, **id**.name)) == `NULL`<br>**then** `error`()<br>**else**<br>$pathname_1$.root = `s`; |
| $pathname \rightarrow$ **id** | **if** (`t` = `get_tree`(**id**.name)) == `NULL`<br>**then** `error`()<br>**else**<br>$pathname$.root = `t`; |

# Exercise 21

The following BNF is given:

*program* → *stat-list*
*stat-list* → *stat* **;** *stat-list* | *stat* **;**
*stat* → *def-stat* | *assign-stat*
*def-stat* → *type id-list*
*type* → **int** | **bool**
*id-list* → **id , ** *id-list* | **id**
*assign-stat* → **id** = *expr*
*expr* → *expr* **+** *expr* | *expr* **and** *expr* | **id**

```
int i, j, k;
bool a, b, c;
i = i + j + k;
a = b and c and a;
```

Specify the (extended) attribute grammar based on the following semantic constraints:

- Variable names are unique,
- Each referenced variable shall have been defined previously,
- Operators (**+** , **and**) shall be applied to correct types

and the following requirements:

- A symbol table is used, where variables and their types are cataloged, by means of the following functions (not to be codified):

  `void insert(char* name, Type type)`: insert variable and its type.

  `Type lookup(char* name)`: returns variable type, if variable exists, otherwise `NULL`.

- In case of semantic error, function `error()` is called, which terminates the analysis.

# Exercise 21

| *Production* | *Semantic rules* |
|---|---|
| *def-stat* → *type id-list* | *id-list*.type = *type*.type |
| *type* → **int** | *type*.type = INT |
| *type* → **bool** | *type*.type = BOOL |
| $\textit{id-list}_1$ → **id** , $\textit{id-list}_2$ | **if** `lookup`(**id**.name) == `NULL` **then** `insert`(**id**.name, $\textit{id-list}_1$.type) **else** `error`();<br>$\textit{id-list}_2$.type = $\textit{id-list}_1$.type |
| *id-list* → **id** | **if** `lookup`(**id**.name) == `NULL` **then** `insert`(**id**.name, *id-list*.type) **else** `error`(); |
| *assign-stat* → **id** = *expr* | **if** (`t` = `lookup`(**id**.name)) == `NULL` **or** `t` != *expr*.type **then** `error`(); |
| $\textit{expr}_1$ → $\textit{expr}_2$ **+** $\textit{expr}_3$ | **if** $\textit{expr}_2$.type != INT **or** $\textit{expr}_3$.type != INT **then** `error`();<br>$\textit{expr}_1$.type = INT |
| $\textit{expr}_1$ → $\textit{expr}_2$ **and** $\textit{expr}_3$ | **if** $\textit{expr}_2$.type != BOOL **or** $\textit{expr}_3$.type != BOOL **then** `error`();<br>$\textit{expr}_1$.type = BOOL |
| *expr* → **id** | **if** (`t` = `lookup`(**id**.name)) == `NULL` **then** `error`();<br>*expr*.type = `t` |

# Exercise 22

A BNF is given, where each phrase of the language defines, initializes, and indexes an associative array, where access keys are strings of characters:

*program* → *def* **;** *init* **;** *ref* **;**
*def* → **id : array of** *type*
*type* → **int** | **bool** | **string**
*ref* → **id [ strconst ]**
*init* → **id = (** *elem-list* **)**
*elem-list* → *elem* **,** *elem-list* | *elem*
*elem* → **strconst =>** *const*
*const* → **intconst** | **boolconst** | **strconst**

```
a: array of int;

a = ("alpha" => 5,
     "beta" => 7,
     "gamma" => 8);

a["beta"];
```

a) Define (with a brief explanation) the set of semantic attributes;

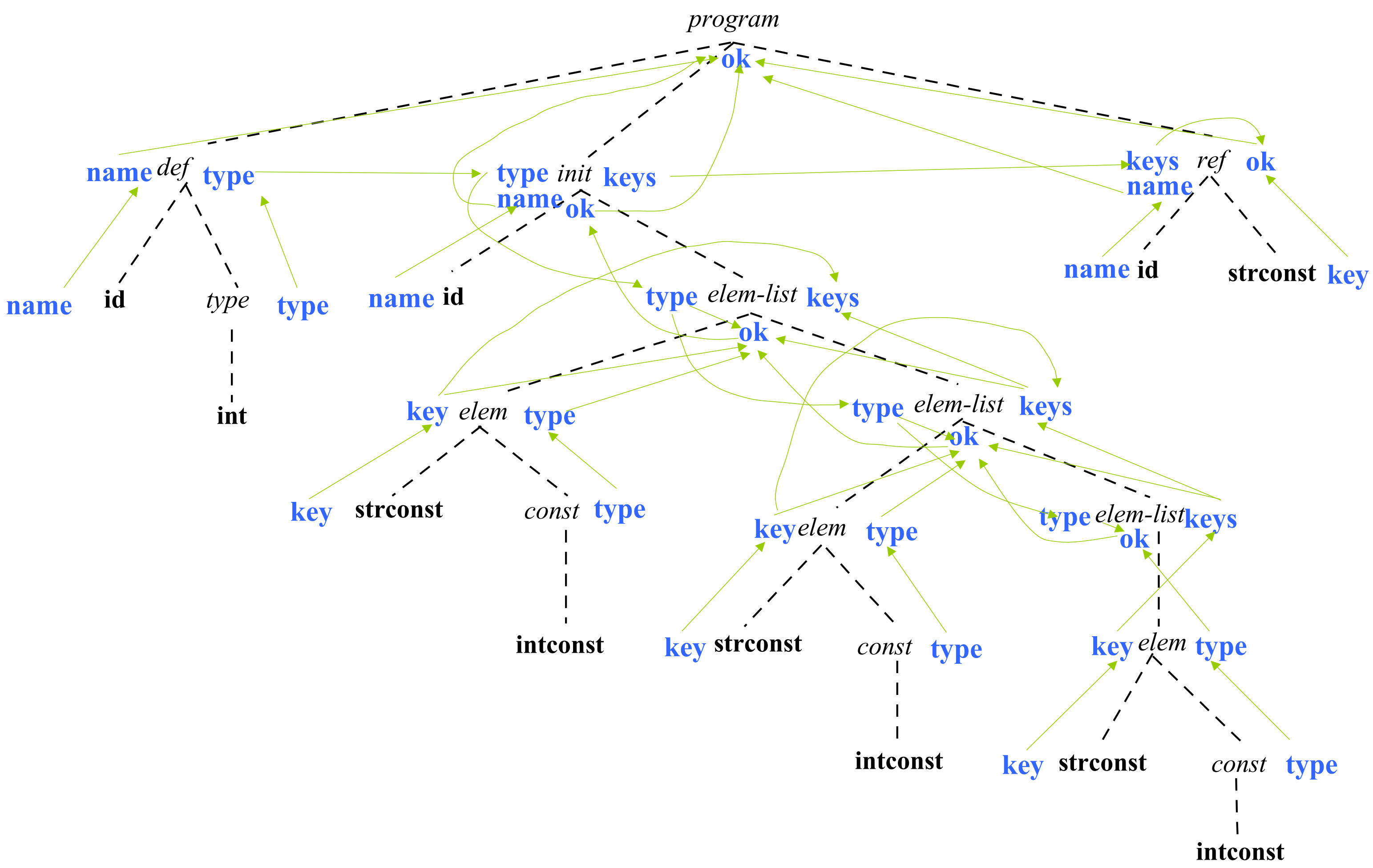
b) represent the decorated abstract syntax tree relevant to the example phrase;

c) Specify the attribute grammar based on the following semantic constraints:

- The initialized (and indexed) array coincides with that declared;
- Within initialization, access keys are unique;
- Within initialization, the type array elements shall be consistent with the declaration;
- Within indexing, the access key shall be one of those defined in the initialization.

# Exercise 22

**Attributes** = {ok, name, type, key, keys}



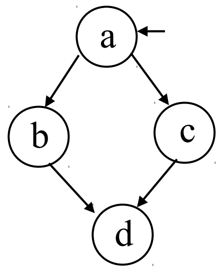
# Exercise 22

| Production | Semantic rules |
|---|---|
| *program* → *def* **;** *init* **;** *ref* **;** | *program*.ok := *def*.name = *init*.name **and** *def*.name = *ref*.name **and** *init*.ok **and** *ref*.ok;<br>*init*.type := *def*.type;<br>*ref*.keys := *init*.keys; |
| *def* → **id : array of** *type* | *def*.name := **id**.name;<br>*def*.type := *type*.type; |
| *type* → **int** | *type*.type := INT; |
| *type* → **bool** | *type*.type := BOOL; |
| *type* → **string** | *type*.type := STRING; |
| *ref* → **id [ strconst ]** | *ref*.name := **id**.name;<br>*ref*.ok := **strconst**.key $\in$ *ref*.keys; |
| *init* → **id = (** *elem-list* **)** | *init*.name := **id**.name;<br>*elem-list*.type := *init*.type;<br>*init*.ok := *elem-list*.ok; |
| $elem\text{-}list_1$ → *elem*, $elem\text{-}list_2$ | $elem\text{-}list_1$.ok := $elem\text{-}list_2$.ok **and** *elem*.key $\notin$ $elem\text{-}list_2$.keys **and** *elem*.type = $elem\text{-}list_1$.type;<br>$elem\text{-}list_1$.keys := { *elem*.key } $\cup$ $elem\text{-}list_2$.keys;<br>$elem\text{-}list_2$.type := $elem\text{-}list_1$.type; |
| *elem-list* → *elem* | *elem-list*.ok := *elem*.type = *elem-list*.type;<br>*elem-list*.keys := { *elem*.key }; |
| *elem* → **strconst =>** *const* | *elem*.type := **const**.type;<br>*elem*.key := **strconst**.key; |
| *const* → **intconst** | *const*.type := INT; |
| *const* → **boolconst** | *const*.type := BOOL; |
| *const* → **strconst** | *const*.type := STRING; |

# Exercise 23

The following BNF is given, relevant to a directed graph:

*graph* → **initial id ;**
**nodes** *id-list* **;**
**arcs** *pair-list* **;**
*id-list* → **id,** *id-list* | **id**
*pair-list* → *pair***,** *pair-list* | *pair*
*pair* → **( id, id )**




```
initial a;
nodes a,b,c,d;
arcs (a,b),(a,c),(b,d),(c,d);
```

a) Define (with a short explanation) the set of semantic attributes;

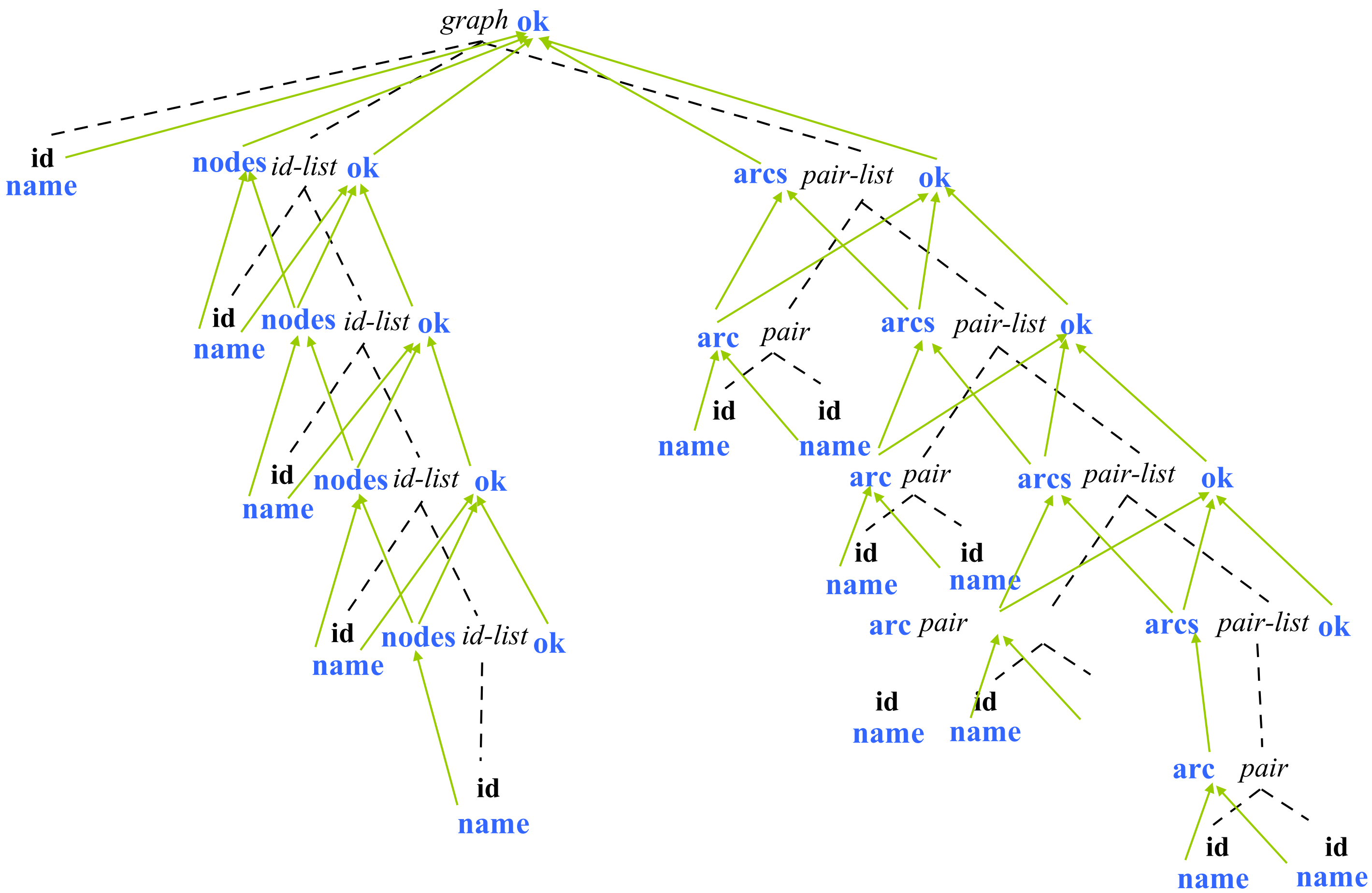
b) Outline the decorated abstract syntax tree relevant to the example phrase;

c) Specify the attribute grammar based on the following semantic constraints:

- Node **initial** belongs to the set of **nodes**;
- Nodes within **nodes** are unique**;**
- Arcs **arcs** are unique;
- All nodes in **nodes** are involved in at least one arc of **arcs**.
- All nodes involved in **arcs** belong to **nodes**.
- The graph is binary (each node is exited by at most two arcs).

# Exercise 23

**Attributes** = {ok, name, nodes, arc, arcs}



# Exercise 23 (ii)

| *Production* | *Semantic rules* |
|---|---|
| $graph \rightarrow$ **initial id ; nodes** $id\text{-}list$ **; arcs** $pair\text{-}list$ **;** | $graph.\mathrm{ok} := id\text{-}list.\mathrm{ok}$ **and** $pair\text{-}list.\mathrm{ok}$ **and** $\mathbf{id}.\mathrm{name} \in id\text{-}list.\mathrm{nodes}$ **and** $\forall N \in id\text{-}list.\mathrm{nodes}\ (\ (N_1, N_2) \in pair\text{-}list.\mathrm{archi},\ N = N_1$ **or** $N = N_2\ )$ **and** $\forall (N_1,N_2) \in pair\text{-}list.\mathrm{arcs}\ (\{N_1,N_2\} \subseteq id\text{-}list.\mathrm{nodes})$ **and** $\forall N \in id\text{-}list.\mathrm{nodes}\ (\lvert\{(N_1,N_2) \mid (N_1,N_2) \in pair\text{-}list.\mathrm{arcs},\ N = N1\}\rvert \leq 2);$ |
| $id\text{-}list_1 \rightarrow$ **id ,** $id\text{-}list_2$ | $id\text{-}list_1.\mathrm{ok} := id\text{-}list_2.\mathrm{ok}$ **and** $id.\mathrm{name} \notin id\text{-}list_2.\mathrm{nodes};$ $id\text{-}list_1.\mathrm{nodes} := \{\ \mathbf{id}.\mathrm{name}\ \} \cup id\text{-}list_2.\mathrm{nodes};$ |
| $id\text{-}list \rightarrow$ **id** | $id\text{-}list.\mathrm{ok} = \mathtt{TRUE};$ $id\text{-}list.\mathrm{nodes} = \{\ \mathbf{id}.\mathrm{name}\ \};$ |
| $pair\text{-}list_1 \rightarrow pair$ **,** $pair\text{-}list_2$ | $pair\text{-}list_1.\mathrm{ok} := pair\text{-}list_2.\mathrm{ok}$ **and** $pair.\mathrm{arc} \notin pair\text{-}list_2.\mathrm{arcs};$ $pair\text{-}list_1.\mathrm{arcs} := \{\ pair.\mathrm{arc}\ \} \cup pair\text{-}list_2.\mathrm{arcs};$ |
| $pair\text{-}list \rightarrow pair$ | $pair\text{-}list.\mathrm{ok} = \mathtt{TRUE};$ $pair\text{-}list.\mathrm{arcs} = \{\ pair.\mathrm{arcs}\ \};$ |
| $pair \rightarrow$ **(** $\mathbf{id}_1$ **,** $\mathbf{id}_2$ **)** | $pair.\mathrm{arc} := (\mathbf{id}_1.\mathrm{name},\ \mathbf{id}_2.\mathrm{name});$ |

# Exercise 24

A BNF is given, relevant to a language of *n*-dimensional matrices, $n \geq 1$:

*program* → *statements*
*statements* → *stat* ; *statements* | *stat*
*stat* → *definition* | *assignment*
*definition* → **id : matrix (** *numbers* **) of** *type*
*numbers* → **number ,** *numbers* | **number**
*type* → **int** | **string**
*assignment* → **id (** *numbers* **) =** *const*
*const* → **intconst** | **strconst**

```
m: matrix(3,4,10) of integer;
s: matrix(20,40) of string;
m(2,1,7) = 12;
s(12,35) = "star";
```

Specify the (extended) attribute grammar based on the following semantic constraints:

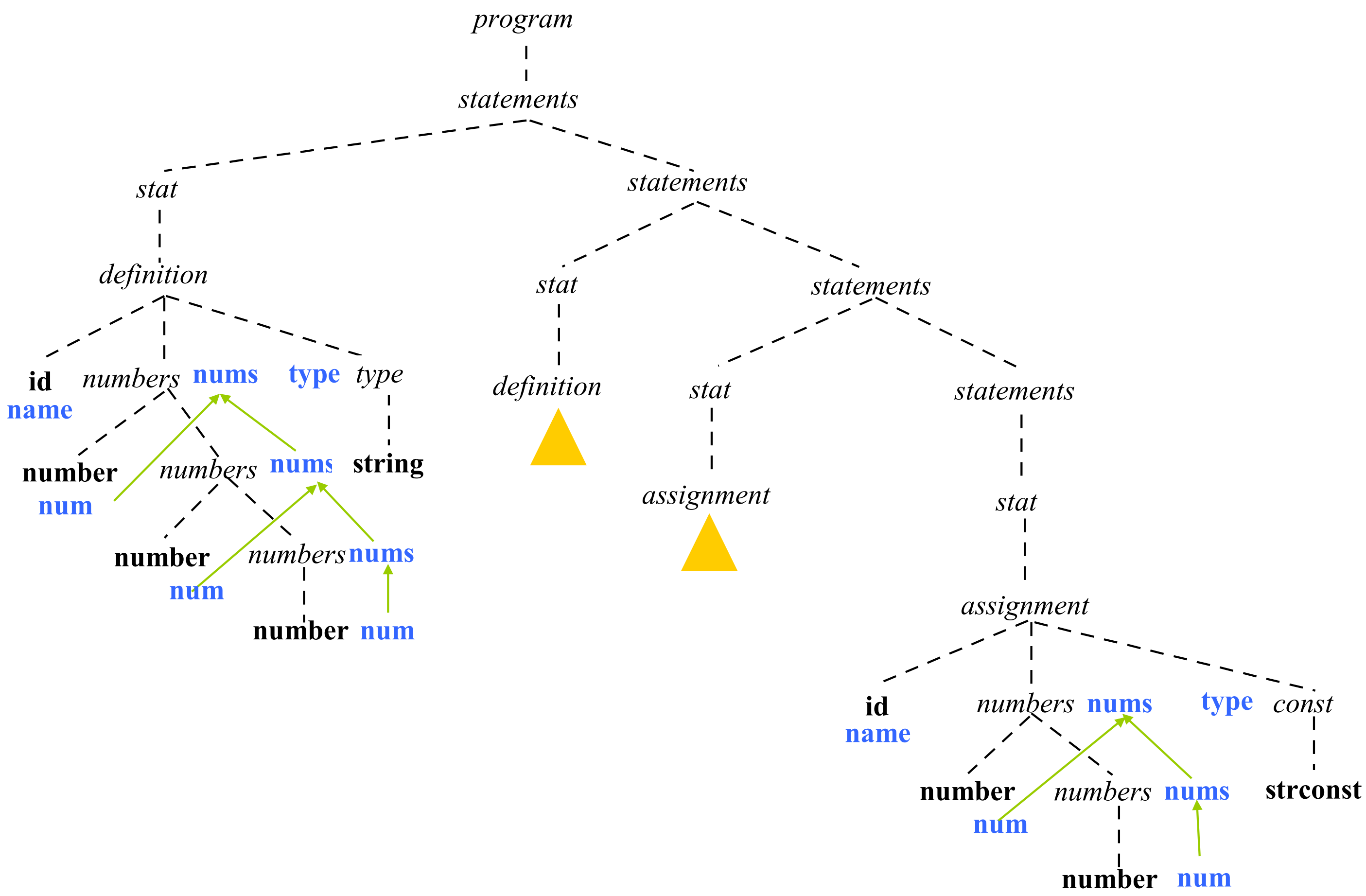
- Within definition, each matrix dimension shall be an integer ≥ 1;
- Matrices cannot be redefined;
- Within assignment, the number of indexes equals the number of dimensions of the matrix;
- Within assignment, each index is between 1 and n, where n is the dimension indexed by the index;
- Within assignment, the RHS type equals the LHS type.

To this end, we assume that:

- There exists a symbol table cataloging matrices, with each row so structured:
  `char* name`: name of matrix;
  `[int] dim`: sequence of matrix dimensions;
  `enum Type {INT,STRING} type`: type of elements of matrix.
- The symbol table is accessed by the following functions:
  `Row* lookup(char* name)`, returning the pointer to the row where matrix `name` is stored, if this exists, otherwise it returns `NULL`;
  `void insert(char* name, [int] dim, Type type)`, cataloging matrix `name`, associating the sequence of dimensions `dim` and the `type` of its elements.
- In case of semantic error, function `semerror()` is called, which terminates the analysis.

# Exercise 24

**Attributes** = { name, nums, type, num }



# Exercise 24 (ii)

| *Production* | *Semantic rules* |
|---|---|
| *program* $\rightarrow$ *statements* | |
| *statements* $\rightarrow$ *stat* ; *statements* | |
| *statements* $\rightarrow$ *stat* | |
| *stat* $\rightarrow$ *definition* | |
| *stat* $\rightarrow$ *assignment* | |
| *definition* $\rightarrow$ **id : matrix (** *numbers* **) of** *type* | **if** `lookup`(**id**.name) != `NULL` **then** `semerror()`<br>**else** `insert`(**id**.name, *numbers*.nums, *type*.type) ; |
| $numbers_1 \rightarrow$ **number** , $numbers_2$ | **if number**.num $\leq 0$ **then** `semerror()`<br>**else** $numbers_1$.nums := [**number**.num] $\cup$ $numbers_2$.nums; |
| *numbers* $\rightarrow$ **number** | **if** *number*.num $\leq 0$ **then** `semerror()`<br>**else** *numbers*.nums := [**number**.num]; |
| *type* $\rightarrow$ **int** | *type*.type := INT; |
| *type* $\rightarrow$ **string** | *type*.type := STRING; |
| *assignment* $\rightarrow$ **id (** *numbers* **) =** *const* | **if** (p = `lookup`(**id**.name)) == `NULL` **or**<br>`p->type` $\neq$ *const*.type **or**<br>`length(p->dim)` $\neq$ `length`(*numbers*.nums) **or**<br>$\exists$ i $\in$ [1 .. `length`(*numbers*.nums)] (*numbers*.nums[i] > `p->dim`[i])<br>**then** `semerror();` |
| *const* $\rightarrow$ **intconst** | *const*.type := INT; |
| *const* $\rightarrow$ **strconst** | *const*.type := STRING; |

# Exercise 25

The following BNF is given:

*program* → *stat-list*
*stat-list* → *stat* **;** *stat-list* | *stat* **;**
*stat* → *def* | *query*
*def* → **table id (** *id-list* **)**
*id-list* → **id ,** *id-list* | **id**
*query* → **select** *id-list* **from** *id-list*

```
table R(a, b, c);

table S(x, y, z, w);

select a, c, y, z
from R, S;
```

Each table is defined by a list of attribute names. Within a query, clause **`select`** specifies a set of attributes relevant to tables specified in the **`from`** clause.

Specify the attribute grammar based on the following semantic constraints:

- Table names are unique,
- Attribute names are unique within a table,
- Within the list in **`select`** clause, attribute names are unique,
- Within the list in **`from`** clause, table names are unique,
- Each table in **`from`** clause shall have been defined,
- Each attribute in **`select`** clause shall belong to one and only one table of **`from`** clause,

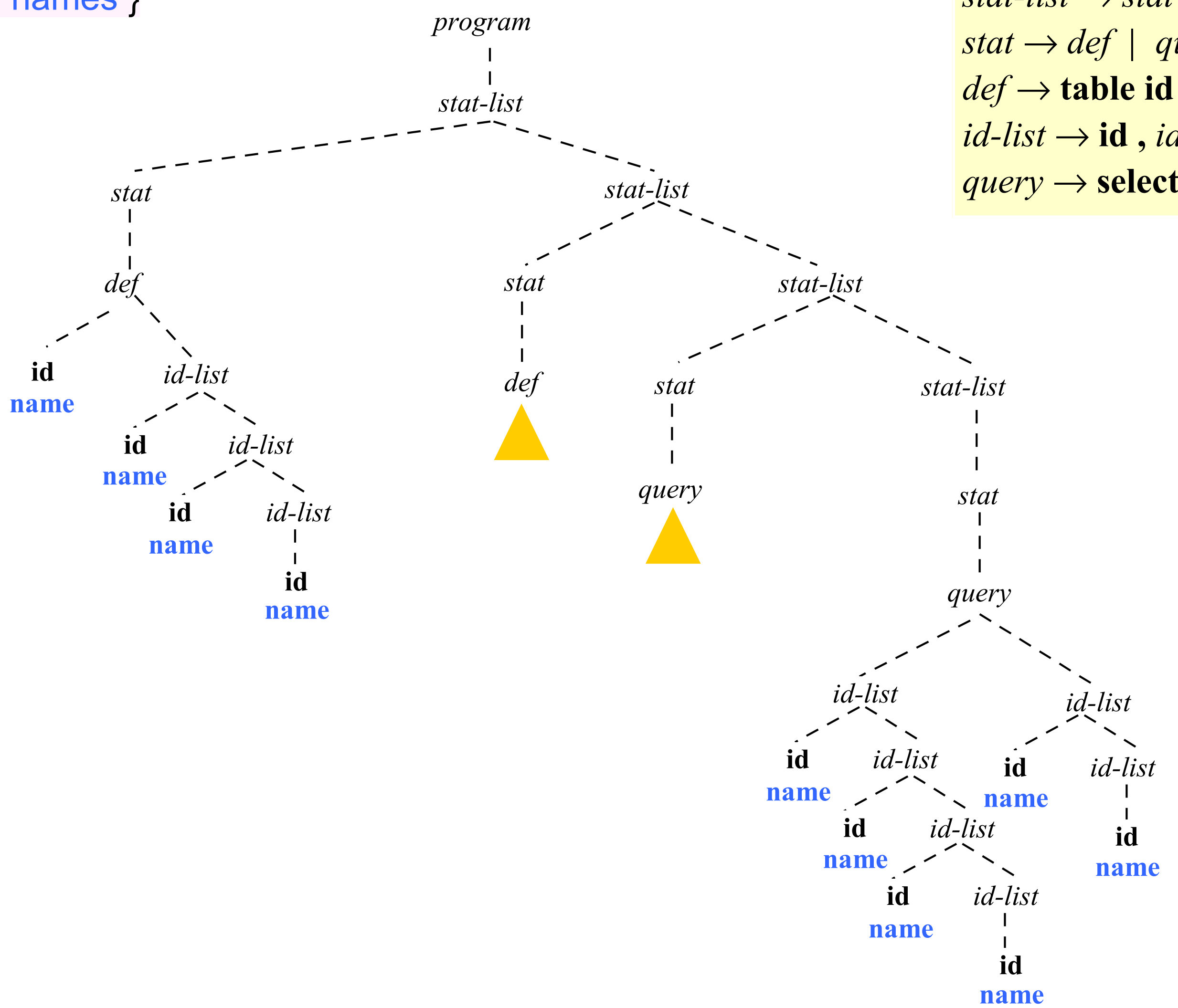
and the following requirements:

- A symbol table is used, for cataloging attributes by means of the following functions:
  ```
  void insert(tabname, attributes)
  Attributes lookup(tabname)
  ```
- Function `insert(tabname, attributes)` catalogs tables with their attributes,
- Function `lookup(tabname)` returns the list of attribute names if the table is cataloged, otherwise it returns `NULL` (if the table is not cataloged),
- In case of semantic error, the analysis terminates by calling `error()`.

# Exercise 25

**Attributes** = { name, names }

*program* → *stat-list*
*stat-list* → *stat* **;** *stat-list* | *stat* **;**
*stat* → *def* | *query*
*def* → **table id (** *id-list* **)**
*id-list* → **id ,** *id-list* | **id**
*query* → **select** *id-list* **from** *id-list*



# Exercise 25 (ii)

| *Production* | *Semantic rules* |
|---|---|
| *def* → **table id (** *id-list* **)** | **if** `lookup`(**id**.name) != `NULL` **then** `error()`;<br>`insert`(**id**.name, *id-list*.names) ; |
| $id\text{-}list_1$ → **id ,** $id\text{-}list_2$ | **if** *id*.name ∈ $id\text{-}list_2$.names **then** `error()`;<br>$id\text{-}list_1$.names = [**id**.name] ∪ $id\text{-}list_2$.names; |
| *id-list* → **id** | *id-list*.names = [**id**.name]; |
| *query* → **select** $id\text{-}list_1$ **from** $id\text{-}list_2$ | **foreach** `tabname` ∈ $id\text{-}list_2$.names **do**<br> **if** `lookup`(`tabname`) == `NULL` **then** `error`() **endif**<br>**endfor**;<br>**forach** `attrname` ∈ $id\text{-}list_1$.names **do**<br> `found` = `FALSE`;<br> **foreach** `tabname` ∈ $id\text{-}list_2$.names **do**<br>  `attributes` = `lookup`(`tabname`);<br>  **if** `attrname` ∈ `attributes` **then**<br>   **if** `found` **then** `error()` **else** `found` = `TRUE` **endif**:<br>  **endif**<br> **endfor**;<br> **if not** `found` **then** `error()` **endif**;<br>**endfor**; |

# Exercise 26

Specify the (extended) attribute grammar relevant to 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*

based on the following semantic constraints:

- Variable names are unique;
- Mixed expressions are not allowed.

Notes:

- A symbol table is used to catalog variables by means of the following functions:
  `void insert(name, type)`
  `Type lookup(name)`: returns the type of variable name (`INT`, `REAL`, `BOOL`) if cataloged, otherwise `NULL`;
- In case of semantic error, function `semerror()` is called, which terminates the analysis.

# Exercise 26

| Production | Semantic rules |
|---|---|
| *declaration* → *type id-list* | *id-list*.type = *type*.type |
| *type* → **int** | type.type = INT |
| *type* → **real** | type.type = REAL |
| *type* → **bool** | type.type = BOOL |
| $\textit{id-list}_1$ → **id** , $\textit{id-list}_2$ | **if** lookup(**id**.lexval) == NULL **then**<br>insert(**id**.lexval, $\textit{id-list}_1$.type)<br>**else** semerror();<br>$\textit{id-list}_2$.type = $\textit{id-list}_1$.type |
| *id-list* → **id** | **if** lookup(**id**.lexval) == NULL **then**<br>insert(**id**.lexval, *id-list*.type)<br>**else** semerror(); |
| *assignment* → **id** = *expr* | **if** ((t = lookup(**id**.lexval)) == NULL **or** *expr*.type != t **then**<br>semerror(); |
| $expr_1$ → $expr_2$ + $expr_3$ | **if** $expr_2$.type == $expr_3$.type **and** $expr_2$.type != BOOL **then**<br>$expr_1$.type = $expr_2$.type<br>**else** semerror(); |
| $expr_1$ → $expr_2$ == $expr_3$ | **if** $expr_2$.type == $expr_3$.type **then**<br>$expr_1$.type = BOOL<br>**else** semerror(); |
| *expr* → **id** | **if** ((t = lookup(**id**.lexval)) != NULL **then**<br>*expr*.type = t<br>**else** semerror(); |
| *expr* → **intconst** | *expr*.type = INT |
| *expr* → **realconst** | *expr*.type = REAL |
| *expr* → **boolconst** | *expr*.type = BOOL |
| *loop* → **while** *expr* **do** *stat* | **if** *expr*.type != BOOL **then** semerror(); |

# Exercise 27

Specify the (extended) attribute grammar relevant to 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 semantic constraints:

- all definitions shall precede all assignments;
- variable names are unique;
- the two variables involved in assignment shall exist and be of same type;
- a variable cannot be assigned with itself.

Notes:

- the lexical value of identifiers is stored in the `lexval` field of the tree node;
- a symbol table is used to catalog variables by means of the following functions:
  `void insert(name, type)`
  `Type lookup(name)`: returns the type of variable `name` (`INT`, `STRING`, `BOOL`) if cataloged, otherwise `NULL`;
- no other global variables can be used;
- in case of semantic error, function `error(string message)` is called, which prints the relevant error `message` before terminating the analysis.

# Exercise 27

| *Production* | *Semantic rules* |
|---|---|
| *program* → *stat-list* | *stat-list*.assigned = **false** |
| $stat\text{-}list_1$ → *stat* **;** $stat\text{-}list_2$ | **if** $stat\text{-}list_1$.assigned **and** *stat*.defined **then**<br>`error`("*Definition after assignment*");<br>$stat\text{-}list_2$.assigned = $stat\text{-}list_1$.assigned **or** *stat*.assigned; |
| *stat-list* → *stat* **;** | **if** *stat-list*.assigned **and** *stat*.defined **then**<br>`error`("*Definition after assignment*"); |
| *stat* → *def-stat* | *stat*.defined = **true**; |
| *stat* → *assign-stat* | *stat*.assigned = **true**; |
| *def-stat* → *id-list type* | *id-list*.type = type.type; |
| *type* → **int** | type.type = `INT` |
| *type* → **string** | type.type = `STRING` |
| *type* → **bool** | type.type = `BOOL` |
| $id\text{-}list_1$ → **id** **,** $id\text{-}list_2$ | **if** `lookup`(**id**.`lexval`) == `NULL` **then**<br>`insert`(**id**.`lexval`, $id\text{-}list_1$.`type`)<br>**else** `error`("*Variable redeclaration*");<br>$id\text{-}list_2$.`type` = $id\text{-}list_1$.`type` |
| *id-list* → **id** | **if** `lookup`(**id**.`lexval`) == `NULL` **then**<br>`insert`(**id**.`lexval`, *id-list*.`type`)<br>**else** `error`("*Variable redeclaration*"); |
| *assign-stat* → $\mathbf{id}_1$ = $\mathbf{id}_2$ | **if** ((`t1` = `lookup`($\mathbf{id}_1$.`lexval`)) == `NULL` **or**<br>(`t2` = `lookup`(**id**2.`lexval`)) == `NULL` **then**<br>`error`("*Undefined variable*")<br>**elsif** `t1` ≠ `t2` **then**<br>`error`("*Different variable types in assignment*")<br>**elsif** $\mathbf{id}_1$.`lexval` == $\mathbf{id}_2$.`lexval` **then**<br>`error`("*Variable assigned with itself*"); |

# Exercise 28

Specify the (extended) attribute grammar relevant to 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**

based on the following semantic constraints:

- Variable names are unique;
- Referenced variables shall exist;
- Arithmetic and logical operators are applied to integers and booleans, respectively;
- Conditions are of type boolean;
- Within the **for** statement, the counting variable is of type integer;
- No mixed expressions are allowed.
- The lexical value of identifiers is stored in the `lexval` field of the tree node;
- 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`;
- In case of semantic error, function `error(string message)` is called, which prints the relevant error `message` before terminating the analysis.

# Exercise 28

| *Production* | *Semantic rules* |
|---|---|
| *def-stat* → **id:** *type* | **if** lookup(**id**.lexval) == NULL **then** insert(**id**.lexval, *type*.type) **else** error("*Variable redeclaration*") **endif**; |
| *type* → **int** | *type*.type = INT |
| *type* → **bool** | *type*.type = BOOL |
| *assign-stat* → **id** := *expr* | **if** ((t = lookup(**id** .lexval)) == NULL **then** error("*Undefined variable*") **elsif** t ≠ *expr*.type **then** error("*Type mismatch in assignment*") **endif**; |
| $expr_1 \rightarrow expr_2 + expr_3$ | **if** $expr_2$.type ≠ INT or $expr_3$.type ≠ INT **then** error("*Wrong type in addition*") **else** $expr_1$.type = INT **endif**; |
| $expr_1 \rightarrow expr_2 * expr_3$ | **if** $expr_2$.type ≠ INT or $expr_3$.type ≠ INT **then** error("*Wrong type in multiplication*") **else** $expr_1$.type = INT **endif**; |
| $expr_1$ → $expr_2$ **or** $expr_3$ | **if** $expr_2$.type ≠ BOOL or $expr_3$.type ≠ BOOL **then** error("*Wrong type in disjunction*") **else** $expr_1$.type = BOOL **endif**; |
| $expr_1$ → $expr_2$ **and** $expr_3$ | **if** $expr_2$.type ≠ BOOL or $expr_3$.type ≠ BOOL **then** error("*Wrong type in conjunction*") **else** $expr_1$.type = BOOL **endif**; |
| $expr_1$ → **not** $expr_2$ | **if** $expr_2$.type ≠ BOOL **then** error("*Wrong type in negation*") **else** $expr_1$.type = BOOL **endif**; |
| *expr* → **id** | **if** (t = lookup(**id**.lexval)) == NULL **then** error("*Unknown variable*") **else** *expr*.type = t **endif**; |
| *expr* → **intconst** | *expr*.type = INT; |
| *expr* → **boolconst** | *expr*.type = BOOL; |
| *if-stat* → **if** *expr* **then** $stat\text{-}list_1$ **else** $stat\text{-}list_2$ **endif** | **if** *expr*.type ≠ BOOL **then** error("*Wrong type in condition*") **endif**; |
| *for-stat* → **for id** = $expr_1$ **to** $expr_2$ **do** *stat-list* **endfor** | **if** (t = lookup(**id**.lexval)) == NULL **then** error("*Undefined counting variable*") **elsif** t ≠ INT **then** error("*Wrong type of counting variable*") **endif**; **if** $expr_1$.type ≠ INT **or** $expr_2$.type ≠ INT **then** error("*Wrong type of range expression*") **endif**; |

# Exercise 29

Specify the (extended) attribute grammar relevant to 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
```

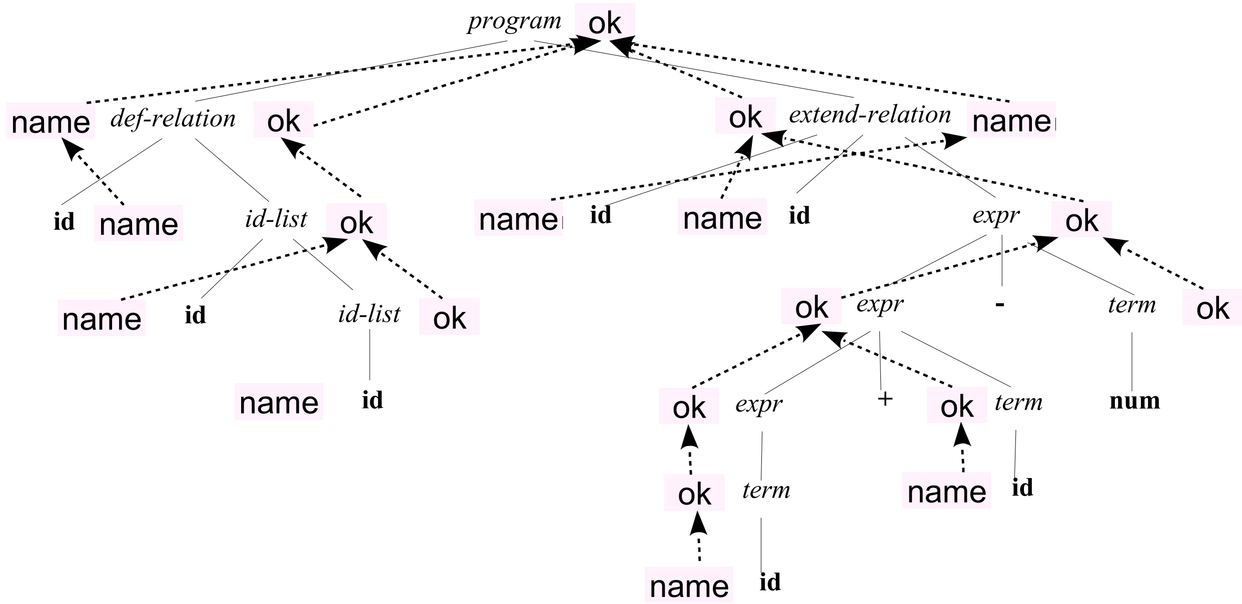
based on the following semantic constraints:

- Attributes are implicitly of integer type;
- Names of attributes are unique,
- The operand of the extend is the defined relation,
- The new attribute does not belong to the relation,
- Each identifier within the expression is an attribute of the relation,

and the following requirements:

- The set of semantic attributes is { ok, name },
- A symbol table is used to catalog table attributes by means of the following functions:
  `void insert(attr)`
  `bool lookup(attr)`
- Function `lookup(name)` returns `true` if the attribute is cataloged, otherwise it returns `false`,
- A possible intermediate semantic error does not terminate the semantic analysis.

# Exercise 29



program
ok
name
def-relation
ok
ok
extend-relation
name
id
name
id-list
ok
name
id
name
id
expr
ok
name
id
id-list
ok
ok
expr
-
term
ok
name
id
ok
expr
+
ok
term
num
ok
term
name
id
name
id

# Exercise 29 (ii)

| *Production* | *Semantic rules* |
|---|---|
| *program* → *def-relation extend-relation* | *program*.ok := *def-relation*.ok **and** *extend-relation*.ok **and** *def-relation*.name = *extend-relation*.name |
| *def-relation* → **relation id (** *id-list* **)** | *def-relation*.name := **id**.name;<br>*def-relation*.ok := *id-list*.ok |
| $id\text{-}list_1$ → **id ,** $id\text{-}list_2$ | $id\text{-}list_1$.ok := $id\text{-}list_2$.ok **and not** lookup(**id**.name) ;<br>insert(**id**.name) |
| *id-list* → **id** | *id-list*.ok := **not** lookup(**id**.name) ;<br>insert(**id**.name) |
| *extend-relation* → **extend** $\mathbf{id}_1$ **by** $\mathbf{id}_2$ = *expr* | *extend-relation*.name := $\mathbf{id}_1$.name;<br>*extend-relation*.ok := **not** lookup($\mathbf{id}_2$.name) **and** *expr*.ok |
| $expr_1$ → $expr_2$ + *term* | $expr_1$.ok := $expr_2$.ok **and** *term*.ok; |
| $expr_1$ → $expr_2$ - *term* | $expr_1$.ok := $expr_2$.ok **and** *term*.ok; |
| *expr* → *term* | *expr*.ok := *term*.*ok*; |
| *term* → **id** | *term*.*ok* := lookup(**id**); |
| *term* → **num** | *term*.*ok* := **true**; |

# Exercise 30

Specify the (extended) attribute grammar relevant to the following BNF,

*program* → *stat-list* | **ε**
*stat-list* → *stat* **;** *stat-list* | *stat* **;**
*stat* → *def-stat* | *assign-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: integer;
a = 3;
for b from 1 to 10 do
  a = b;
  b = c;
end;
```

based on the following semantic constraints:

- Variable names are unique;
- Referenced variables shall exist;
- In loop, the counting variable is of type integer;
- In the range $[n .. m]$ of a loop, $m > n$;
- Variables are assigned with constants of the same type.

and the following requirements:

- Lexical values of terminals are `ival` (integer) and `sval` (string);
- A symbol table is used to catalog variables by means of the following functions:
- `void insert(name, type)`: insert variable name with `type`;
  `Type lookup(name)`: returns the type of variable name (`INT`, `STR`) if cataloged, otherwise `NULL`;
- In case of semantic error, function `semerror(string msg)` is called, which prints a <u>pertinent</u> error message `msg`, and then terminates the analysis.

# Exercise 30

| Production | Semantic rules |
|---|---|
| *def-stat* → **def** *id-list* **as** *type* | *id-list*.type = *type*.type |
| *type* → **integer** | type.type = `INT` |
| *type* → **string** | type.type = `STR` |
| $id\text{-}list_1$ → **id** **,** $id\text{-}list_2$ | **if** `lookup`(**id**.`sval`) == `NULL` **then**<br>`insert`(**id**.`sval`, $id\text{-}list_1$.`type`)<br>**else** `semerror`("Redelcared variable");<br>$id\text{-}list_2$.`type` = $id\text{-}list_1$.`type` |
| *id-list* → **id** | **if** `lookup`(**id**.`sval`) == `NULL` **then**<br>`insert`(**id**.`sval`, $id\text{-}list_1$.`type`)<br>**else** `semerror`("Redeclared variable"); |
| *assign-stat* → ***id*** **=** *const* | **if** ((`t` = `lookup`(**id**.`sval`)) == `NULL`) **then**<br>`semerror`("Undeclared variable")<br>**elsif** *const*.`type` != `t` **then**<br>`semerror`("Type mismatch in assignment"); |
| *const* → **intconst** | *const*.`type` = `INT` |
| *const* → **strconst** | *const*.`type` = `STR` |
| *loop-stat* → **for id from** $\textbf{intconst}_1$ **to** $\textbf{intconst}_2$ **do** *stat-list* **end** | **if** ((`t` = `lookup`(**id**.`sval`)) == `NULL` **then** `semerror`("Undeclared variable");<br>**elsif** `t` ≠ `INT` **then**<br>`semerror`("Counting variable must be of integer type");<br>**elsif** $\textbf{intconst}_2$.`ival` - $\textbf{intconst}_1$.`ival` < 1 **then**<br>`semerror`("Wrong loop range"); |

# Exercise 31

Specify the (extended) attribute grammar relevant to the following BNF,

*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;
```

based on the following semantic constraints <u>only</u>:

- Variable names are unique;
- Referenced variables shall exist;
- Each dimension in matrix definition is greater than zero;
- In case statement, the case variable (**id**) is of simple type (either integer or string);
- In case statement, each case constant has the same type of the case variable;
- Variables are assigned with constants of the same type (<u>in case of matrix, no type-checking of the deep structure of the matrix is required</u>).

and the following requirements:

- Lexical values of terminals are accessed through `lexval`;
- A symbol table is used to catalog variables by means of the following functions:
- `void insert(name, type)`: insert variable name with `type`;
  `Type lookup(name)`: returns the type of variable name (`INT`, `STR`, `MAT`) if cataloged, otherwise `NULL`;
- In case of semantic error, function `semerror(string msg)` is called, which prints a <u>pertinent</u> error message `msg`, and then terminates the analysis.

# Exercise 31

| Production | Semantic rules |
|---|---|
| *def-stat* → **var** *id-list* **is** *type* | *id-list*.type = *type*.type |
| *type* → **integer** | *type*.type = `INT` |
| *type* → **string** | *type*.type = `STR` |
| $type_1$ → **matrix (** *intconst-list* **) of** $type_2$ | $type_1$.type = `MAT` |
| $id\text{-}list_1$ → **id ,** $id\text{-}list_2$ | **if** `lookup`(**id**.`lexval`) == `NULL` **then** `insert`(**id**.`lexval`, $id\text{-}list_1$.`type`)<br>**else** `semerror`("Redelcared variable");<br>$id\text{-}list_2$.`type` = $id\text{-}list_1$.`type` |
| *id-list* → **id** | **if** `lookup`(**id**.`lexval`) == `NULL` **then** `insert`(**id**.`lexval`, $id\text{-}list_1$.`type`)<br>**else** `semerror`("Redeclared variable"); |
| $intconst\text{-}list_1$ → **intconst ,** $intconst\text{-}list_2$ | **if** (**intconst**.`lexval`) ≤ `0` **then** `semerror`("Negative dimension"); |
| $intconst\text{-}list_1$ → **intconst** | **if** (**intconst**.`lexval`) ≤ `0` **then** `semerror`("Negative dimension"); |
| *assign-stat* → **id =** *const* | **if** ((`t` = `lookup`(**id**.`lexval`)) == `NULL` **then** `semerror`("Undeclared variable")<br>**elsif** `t` ≠ *const*.`type` **then** `semerror`("Type mismatch in assignment"); |
| *const* → **intconst** | *const*.`type` = `INT` |
| *const* → **strconst** | *const*.`type` = `STR` |
| *const* → *matconst* | *const*.`type` = `MAT` |
| *case-stat* → **case id of** *branch-list opt-default* **end** | **if** (`t` = `lookup`(**id**.`lexval`)) == `NULL` **then** `semerror`("Undeclared variable")<br>**elsif** `t` == `MAT` **then** `semerror`("Case variable cannot be a matrix");<br>*branch-list*.`type` = `t`; |
| $branch\text{-}list_1$ → *branch* **,** $branch\text{-}list_2$ | *branch*.`type` = $branch\text{-}list_1$.`type`;<br>$branch\text{-}list_2$.`type` = $branch\text{-}list_1$.`type`; |
| *branch-list* → *branch* | *branch*.`type` = $branch\text{-}list_1$.`type`; |
| *branch* → *const* **:** *stat* | **if** *branch*.`type` ≠ *const*.`type` **then**<br>`semerror`("Type mismatch in case constant"); |

# Exercise 32

Specify the attribute grammar relevant to the following BNF,

*program* → *rec-def*   *rec-assign*
*rec-def* → **def id : record (** *attr-list* **)**
*attr-list* → *attr* **,** *attr-list* | *attr*
*attr* → **id :** *type*
*type* → **int** | **string** | **bool**
*rec-assign* → **id := record (** *const-list* **)**
*const-list* → *const* **,** *const-list* | *const*
*const* → **intconst** | **strconst** | **boolconst**

```
def r: record (a: int, b: string, c: bool)
r := record (12, "omega", true)
```

based on the following semantic constraints:

- The name of the defined record shall equal the name of the assigned record;
- Attribute names shall be unique;
- The attribute values in the assignment shall be consistent with the attribute types in the definition.

# Exercise 32

A = { ok, name, names, type, sign }

| *Production* | *Semantic rules* |
|---|---|
| *program* → *rec-def* *rec-assign* | *program*.ok := *rec-def*.ok **and** *rec-def*.name = *rec-assign*.name **and** *rec-def*.sign = *rec-assign*.sign |
| *rec-def* → **def id : record (** *attr-list* **)** | *rec-def*.name := **id**.name<br>*rec-def*.sign := *attr-list*.sign<br>*rec-def*.ok := *attr-list*.ok |
| $attr\text{-}list_1$ → *attr* **,** $attr\text{-}list_2$ | $attr\text{-}list_1$.ok := *attr*.name ∉ $attr\text{-}list_2$.names<br>$attr\text{-}list_1$.names := [*attr*.name] ∪ $attr\text{-}list_2$.names<br>$attr\text{-}list_1$.sign := [*attr*.type] ∪ $attr\text{-}list_2$.sign |
| *attr-list* → *attr* | *attr-list*.ok := **true**<br>*attr-list*.names := [*attr*.name]<br>*attr-list*.sign := [*attr*.type] |
| *attr* → **id :** *type* | *attr*.name := **id**.name<br>*attr*.type := *type*.type |
| *type* → **int** | *type*.type := INT |
| *type* → **string** | *type*.type := STRING |
| *type* → **bool** | *type*.type := BOOL |
| *rec-assign* → **id := record (** *const-list* **)** | *rec-assign*.name := **id**.name<br>*rec-assign*.sign := *const-list*.sign |
| $const\text{-}list_1$ → *const* **,** $const\text{-}list_2$ | $const\text{-}list_1$.sign := [*const*.type] ∪ $const\text{-}list_2$.sign |
| *const-list* → *const* | *const-list*.sign := [*const*.type] |
| *const* → **intconst** | *const*.type := INT |
| *const* → **strconst** | *const*.type := STRING |
| *const* → **boolconst** | *const*.type := BOOL |

# Exercise 33

Based on all reasonable semantic constraints of a strongly typed language, specify the attribute grammar relevant to the following BNF (in particular, in **foreach** loop, *expr* shall be an array with element type equal to the type of variable **id**):

*program* → *stat-list*
*stat-list* → *stat* **;** *stat-list* | *stat* **;**
*stat* → *def-stat* | *assign-stat* | *if-stat* | *foreach-stat*
*def-stat* → *id-list* **:** *type*
*id-list* → **id** **,** *id-list* | **id**
*type* → **int** | **bool** | *array-type*
*array-type* → **array [ intconst ] of** *type*
*assign-stat* → **id :=** *expr*
*expr* → *expr* + *expr* | *expr* **and** *expr* | **-** *expr* | **not** *expr* | **(***expr* **)** | **id** | **intconst** | **boolconst**
*if-stat* → **if** *expr* **then** *stat-list* **else** *stat-list* **endif**
*foreach-stat* → **foreach id in** *expr* **do** *stat-list* **endfor**

assuming each node of the type tree being qualified by fields `domain` ∈ {`INT`, `BOOL`, `ARRAY`}, `size` (array dimension), and `child` (pointer to array element type), and the availability of the following auxiliary functions:

- `insert(name,type)`: inserts variable name and its type into the symbol table;
- `lookup(name)`: returns type of variable name (if cataloged) or **nil**;
- `typeEqual(t1,t2)`: checks the equality of types `t1` and `t2`;
- `simpleNode(domain)`: creates a type node for `domain` ∈ {`INT`, `BOOL`};
- `arrayNode(size,type)`: creates an array type node with dimension `size` and child type `type`;
- `error(message)`: prints relevant error `message` and terminates the analysis.

# Exercise 33

A = { name, type, val }

| Production | Semantic rules |
|---|---|
| *def-stat* → *id-list* **:** *type* | *id-list*.type = *type*.type |
| $id\text{-}list_1$ → **id ,** $id\text{-}list_2$ | **if** lookup(**id**.name) = **nil then** insert(**id**.name, $id\text{-}list_1$.type) **else** error("Redeclared variable") **endif;** <br> $id\text{-}list_2$.type = $id\text{-}list_1$.type |
| *id-list* → **id** | **if** lookup(**id**.name) = **nil then** insert(**id**.name, *id-list*.type) **else** error("Redeclared variable") **endif** |
| *type* → **int** | *type*.type = simpleNode(INT) |
| *type* → **bool** | *type*.type = simpleNode(BOOL) |
| *type* → *array-type* | *type*.type = *array-type*.type |
| *array-type* → **array [ intconst ] of** *type* | **if intconst**.val <= 0 **then** error("Wrong array size") **endif;** <br> *array-type*.type = arrayNode(**intconst**.val, *type*.type) |
| *assign-stat* → **id :=** *expr* | **if** (t = lookup(**id**.name)) == **nil**) **then** error("Undeclared varianle") <br> **elsif not** typeEqual(t, *expr*.type) **then** error("Type mismatch") **endif** |
| $expr_1$ → $expr_2$ + $expr_3$ | **if** $expr_2$.type->domain != INT **or** $expr_3$.type->domain != INT **then** error("Type must be integer") <br> **else** $expr_1$.type = $expr_2$.type **endif** |
| $expr_1$ → $expr_2$ **and** $expr_3$ | **if** $expr_2$.type->domain != BOOL **or** $expr_3$.type->domain != BOOL **then** error("Type must be boolean") <br> **else** $expr_1$.type = $expr_2$.type **endif** |
| $expr_1$ → **-** $expr_2$ | **if** $expr_2$.type->domain != INT **then** error("Type must be integer") <br> **else** $expr_1$.type = $expr_2$.type **endif** |
| $expr_1$ → **not** $expr_2$ | **if** $expr_2$.type->domain != BOOL **then** error("Type must be boolean") <br> **else** $expr_1$.type = $expr_2$.type **endif** |
| $expr_1$ → **(** $expr_2$ **)** | $expr_1$.type = $expr_2$.type |
| *expr* → **id** | **if** (t = lookup(**id**.name)) == **nil**) **then** error("Undeclared variable") <br> **else** *expr*.type = t **endif** |
| *expr* → **intconst** | *expr*.type = simpleNode(INT) |
| *expr* → **boolconst** | *expr*.type = simpleNode(BOOL) |
| *if-stat* → **if** *expr* **then** *stat-list* **else** *stat-list* **endif** | **if** *expr*.type->domain != BOOL **then** error("Expected boolean type") **endif** |
| *foreach-stat* → **foreach id in** *expr* **do** *stat-list* **endfor** | **if** (t = lookup(id.name)) == **nil**) **then** error("Undeclared variable") <br> **elsif** *expr*.type->domain != ARRAY **then** error("Expected array type") <br> **elsif not not** typeEqual(t, expr.type->child) **then** error("Type mismatch") **endif** |