Esercizio 1

Specificare la grammatica EBNF di un linguaggio in cui ogni frase corrisponde ad una o più dichiarazioni di protocollo di funzione. Ogni protocollo è composto dalla keyword **function**, seguita dal nome della funzione, dalla definizione di uno o più parametri di ingresso e dal tipo del valore di ritorno, come nella seguente frase:

```
function alfa(A: integer, B: array [2..5] of string): boolean;
function beta(V: array [1..10] of array [1..5] of integer): integer;
function gamma(X: string): array [5..20] of boolean;
```

Il dominio (tipo) dei parametri (sia di ingresso che di uscita) può essere atomico (**integer**, **string**, **boolean**) o strutturato (**array** di domini atomici o strutturati). Il range di un array è definito da due costanti intere separate da due punti orizzontali.

Esercizio 1

Specificare la grammatica EBNF di un linguaggio in cui ogni frase corrisponde ad una o più dichiarazioni di protocollo di funzione. Ogni protocollo è composto dalla keyword **function**, seguita dal nome della funzione, dalla definizione di uno o più parametri di ingresso e dal tipo del valore di ritorno, come nella seguente frase:

```
function alfa(A: integer, B: array [2..5] of string): boolean;
function beta(V: array [1..10] of array [1..5] of integer): integer;
function gamma(X: string): array [5..20] of boolean;
```

Il dominio (tipo) dei parametri (sia di ingresso che di uscita) può essere atomico (**integer**, **string**, **boolean**) o strutturato (**array** di domini atomici o strutturati). Il range di un array è definito da due costanti intere separate da due punti orizzontali.

program → { *funct-def* }$^{+}$
funct-def → **function id** "**(**" *param-list* "**)**" **:** *domain* **;**
param-list → *param-decl* { **,** *param-decl* }
param-decl → **id :** *domain*
domain → *atomic-domain* | *structured-domain*
atomic-domain → **integer** | **string** | **boolean**
structured-domain → **array [num .. num] of** *domain*

Esercizio 2

Specificare la grammatica EBNF del linguaggio **L**, in cui ogni frase è una sequenza non vuota di assegnamenti terminati dal separatore ';'. Un assegnamento è composto da un identificatore alfanumerico (a sinistra), il simbolo ':=' (nel mezzo) ed una espressione insiemistica (a destra). Gli operatori insiemistici sono infissi (in mezzo ai loro due operandi) e comprendono **`union`**, **`diff`**, e **`intersect`**. I loro operandi possono essere altre espressioni insiemistiche o semplici identificatori. In particolare, una espressione può essere un identificatore. Ogni applicazione di un operatore insiemistico ai suoi operandi è sempre racchiusa tra parentesi. Ecco un esempio di frase di **L**:

```
alfa := beta;
beta := (A union B);
gamma := ((alfa union beta) diff (A intersect B));
```

Esercizio 2

Specificare la grammatica EBNF del linguaggio **L**, in cui ogni frase è una sequenza non vuota di assegnamenti terminati dal separatore ';'. Un assegnamento è composto da un identificatore alfanumerico (a sinistra), il simbolo ':=' (nel mezzo) ed una espressione insiemistica (a destra). Gli operatori insiemistici sono infissi (in mezzo ai loro due operandi) e comprendono **`union`**, **`diff`**, e **`intersect`**. I loro operandi possono essere altre espressioni insiemistiche o semplici identificatori. In particolare, una espressione può essere un identificatore. Ogni applicazione di un operatore insiemistico ai suoi operandi è sempre racchiusa tra parentesi. Ecco un esempio di frase di **L**:

```
alfa := beta;
beta := (A union B);
gamma := ((alfa union beta) diff (A intersect B));
```

$$
\begin{aligned}
&\textit{program} \rightarrow \{\textit{stat};\}^{+}\\
&\textit{stat} \rightarrow \textbf{id} := \textit{expr}\\
&\textit{expr} \rightarrow (\textit{expr}\ \textit{op}\ \textit{expr}) \mid \textbf{id}\\
&\textit{op} \rightarrow \textbf{union} \mid \textbf{diff} \mid \textbf{intersect}
\end{aligned}
$$

Esercizio 3

Specificare la grammatica EBNF di un linguaggio in cui ogni frase corrisponde ad una o più dichiarazioni di variabili. Ogni dichiarazione è definita dal nome della variabile e dal suo tipo. Possibili tipi sono **int**, **string**, **bool** e **table**. A differenza dei primi tre che sono semplici, il tipo **table** definisce una tabella le cui colonne sono a loro volta caratterizzate da identificatori e relativi tipi, incluso **table**. In questo modo, il linguaggio permette la definizione di tabelle complesse (tabelle che incorporano tabelle, senza limiti di profondità). Ecco un esempio di frase (si noti che la descrizione di una tabella è racchiusi tra parentesi graffe):

```
alfa: int
beta: string
T: table {A: string, B: table {C: bool, D: int}, E: string}
```

Esercizio 3

Specificare la grammatica EBNF di un linguaggio in cui ogni frase corrisponde ad una o più dichiarazioni di variabili. Ogni dichiarazione è definita dal nome della variabile e dal suo tipo. Possibili tipi sono **int**, **string**, **bool** e **table**. A differenza dei primi tre che sono semplici, il tipo **table** definisce una tabella le cui colonne sono a loro volta caratterizzate da identificatori e relativi tipi, incluso **table**. In questo modo, il linguaggio permette la definizione di tabelle complesse (tabelle che incorporano tabelle, senza limiti di profondità). Ecco un esempio di frase (si noti che la descrizione di una tabella è racchiusi tra parentesi graffe):

```
alfa: int
beta: string
T: table {A: string, B: table {C: bool, D: int}, E: string}
```

program → { *def* }$^{+}$
def → **id :** *type*
type → *simple-type* | *table-type*
simple-type → **bool** | **string** | **int**
table-type → **table** "**{**" *attr-list* "**}**"
attr-list → *attr-decl* { **,** *attr-decl* }
attr-decl → **id :** *type*

Esercizio 4

Specificare la grammatica EBNF di un linguaggio in cui ogni frase corrisponde ad una o più dichiarazioni di classi. Ogni dichiarazione di classe è costituita da una intestazione (comprendente il nome della classe), dalla dichiarazione delle variabili di istanza (almeno una), dalla dichiarazione dei metodi (almeno uno), e da una coda (comprendente il nome della classe). Ecco un esempio di frase che definisce due classi Alfa e Beta:

```
class Alfa is
  variables
    a, b, c: integer;
    d, e: real;
    f: record (x: integer, y: record (v: string, w: real));
  methods
    gamma(a: string, d: real), delta(f: string): real;
    epsilon(c: integer): integer;
    omicron(d: real), tau(r: integer): record(a: string, b: real);
end Alfa;

class Beta is
  variables
    m, n: integer;
    r1, r2: record (x: integer, y: string);
  methods
    zeta(a: string, d: real), sigma(f: string): string;
    omega(c: integer): integer;
end Beta;
```

Le variabili (introdotte dalla keyword **variables**) sono definite da sequenze di identificatori terminate dal tipo corrispondente. Possibili tipi sono **integer**, **real**, **string** e **record**. I primi tre tipi sono atomici. Il tipo **record** definisce una struttura (racchiusa tra parentesi tonde) i cui campi (almeno uno) sono a loro volta caratterizzati da identificatori e relativi tipi. I tipi sono ortogonali tra loro. La dichiarazione dei metodi (introdotti dalla keyword **methods**) è simile a quella delle variabili, con l'aggiunta della dichiarazione dei parametri formali in ingresso (almeno uno) .

Esercizio 4

```
class Alfa is
  variables
    a, b, c: integer;
    d, e: real;
    f: record (x: integer, y: record (v: string, w: real));
  methods
    gamma(a: string, d: real), delta(f: string): real;
    epsilon(c: integer): integer;
    omicron(d: real), tau(r: integer): record(a: string, b: real);
end Alfa;

class Beta is
  variables
    m, n: integer;
    r1, r2: record (x: integer, y: string);
  methods
    zeta(a: string, d: real), sigma(f: string): string;
    omega(c: integer): integer;
end Beta;
```

program → { *class-decl* }$^{+}$
class-decl → **class id is** *var-decl meth-decl* **end id;**
var-decl → **variables** { *var-list* }$^{+}$
var-list → **id** {**, id**}**:** *domain*;
domain → *atomic-domain* | *record-domain*
atomic-domain → **integer** | **real** | **string**
record-domain → **record (** *pair-list* **)**
pair-list → **id:** *domain* {**, id:** *domain* }
meth-decl → **methods** { *meth-list* }$^{+}$
meth-list → *method* {, *method* }**:** *domain***;**
method →**id (** *pair-list* **)**

Esercizio 5

Specificare la grammatica EBNF di un linguaggio per la manipolazione di tabelle, in cui ogni frase comprende una o più sezioni. Esistono due tipi di sezioni: **data** e **query**. Nella sezione **data** vengono definiti gli schemi di una o più tabelle. Nella sezione **query** vengono formulate una o più interrogazioni sulle tabelle utilizzando (in modo ortogonale) le operazioni relazionali di proiezione (**project**) e selezione (**select**). Lo schema di una tabella è specificato dal nome della tabella e dalla lista di attributi definiti su corrispondenti domini (**int**, **string**, **bool**). Nella operazione **project** si specifica la lista degli attributi di proiezione. Nella operazione **select** si specifica un predicato di selezione composto dalla congiunzione di una o più confronti tra un attributo e una costante. Ecco una frase contenente due sezioni:

```
data
  R: (a: int, b: string, c: bool, d: int);
  S: (alfa: bool, beta24: string);
  T: (X: string);
end

query
  project [a, b ] R;
  select [ alfa = false ] S;
  select [ X = "luna" ] T;
  project [ b ] select [ a = 5 ] R;
  project [a, d ]
    select [ a = 3 and b = "stella" and c = false ]
      select [ d = 125 ] R;
end
```

Esercizio 5

Specificare la grammatica EBNF di un linguaggio per la manipolazione di tabelle, in cui ogni frase comprende una o più sezioni. Esistono due tipi di sezioni: **data** e **query**. Nella sezione **data** vengono definiti gli schemi di una o più tabelle. Nella sezione **query** vengono formulate una o più interrogazioni sulle tabelle utilizzando (in modo ortogonale) le operazioni relazionali di proiezione (**project**) e selezione (**select**). Lo schema di una tabella è specificato dal nome della tabella e dalla lista di attributi definiti su corrispondenti domini (**int**, **string**, **bool**). Nella operazione **project** si specifica la lista degli attributi di proiezione. Nella operazione **select** si specifica un predicato di selezione composto dalla congiunzione di una o più confronti tra un attributo e una costante. Ecco una frase contenente due sezioni:

```
data
  R: (a: int, b: string, c: bool, d: int);
  S: (alfa: bool, beta24: string);
  T: (X: string);
end

query
  project [a, b ] R;
  select [ alfa = false ] S;
  select [ X = "luna" ] T;
  project [ b ] select [ a = 5 ] R;
  project [a, d ]
    select [ a = 3 and b = "stella" and c = false ]
      select [ d = 125 ] R;
end
```

program → { *section* }$^+$
section → *data-section* | *query-section*
data-section → **data** { *schema-def* }$^+$ **end**
schema-def → **id : (** *attr-def* { **,** *attr-def* } **) ;**
attr-def → **id :** *domain*
domain → **int** | **string** | **bool**
query-section → **query** { *query-def* }$^+$ **end**
query-def → { *operation* }$^+$ **id ;**
operation → *proj-op* | *sel-op*
proj-op → **project [id** { **, id** } **]**
sel-op → **select [id =** *const* { **and id =** *const* } **]**
const → **intconst** | **strconst** | **boolconst**

Esercizio 6

Specificare la grammatica EBNF di un linguaggio in cui ogni frase corrisponde ad un modulo SQL-like. Tale modulo contiene una lista (non vuota) di istruzioni. Ogni istruzione può essere una creazione di tabella o una interrogazione, come nel seguente esempio:

```
create table R
(
    A    character(20),
    B    numeric(5),
    C    character(15)
);

create table S
(
    D    character(20),
    E    numeric(6)
);

select A, E
from R, S
where A = D and B > 100 or E < 50;

select *
from R, S
where A = 'alfa' and B in (select E
                           from S
                           where D = A);
```

I domini degli attributi delle tabelle sono **character** e **numeric**, entrambi qualificati con una dimensione. Ogni interrogazione ha la forma *select-from-where*, in cui le clausole **select** e **from** sono obbligatorie. La clausola **select** specifica una lista di attributi o il metacarattere '*'. La clausola **from** specifica una lista di tabelle. La clausola **where** specifica il predicato di selezione che coinvolge gli operatori logici **and** ed **or**, i quali sono applicati ad operazioni di confronto (**=**, **<>**, **>**, **>=**, **<** , **<=**) o di appartenenza (**in**). Il secondo operando dell'operatore di appartenenza (necessariamente fra parentesi) è a sua volta (ricorsivamente) una espressione *select-from-where*.

Esercizio 6

```
create table R
(
    A    character(20),
    B    numeric(5),
    C    character(15)
);

create table S
(
    D    character(20),
    E    numeric(6)
);

select A, E
from R, S
where A = D and B > 100 or E < 50;

select *
from R, S
where A = 'alfa' and B in (select E
                           from S
                           where D = A);
```

module → { *stat* **;** }$^{+}$
stat → *creation* | *query*
creation → **create table id (** *attr-decl* { **,** *attr-decl* } **)**
attr-decl → **id** (**character** | **numeric**) **(num)**
query → **select** (*id-list* | *****) **from** *id-list* [**where** *predicate*]
id-list → **id** { **,** **id** }
predicate → *condition* { (**and** | **or**) *condition* }
condition → *comparison* | *membership*
comparison → *elem comp-op elem*
elem → **id** | **strconst** | **intconst**
comp-op → **=** | **<>** | **>** | **<** | **>=** | **<=**
membership → *elem* **in (** *query* **)**

Esercizio 7

Specificare la EBNF di un linguaggio per definire e manipolare tabelle complesse, come nel seguente esempio:

```
def T1:(a: int, b: string, c: bool, d: int);
def T2:(alfa: int, beta: int, gamma:(x: int, y: string));
def T3:(m: int, r:(v: int, w: string, s:(p: string, q: string), n: int), t: bool);
…
retrieve [a=d] T1;
retrieve [a=d and b="sole"] T1;
retrieve [a=d and b="sole" or c=false] T1;
retrieve [alfa=15 and retrieve [x=4 or y="stella"] gamma = T5] T2;
retrieve [beta=10 and retrieve [x=20] gamma = retrieve [y="luna"] gamma] T2;
```

Un programma è una lista non vuota di istruzioni di definizione e/o di manipolazione. Ogni istruzione è terminata dal simbolo '**;**'. I due tipi di istruzioni possono essere specificati in qualsiasi ordine reciproco. Una istruzione di definizione specifica il nome di una tabella e la lista (non vuota) di attributi con i relativi domini. I domini atomici sono **`int`**, **`string`** e **`bool`**. Un attributo può essere a sua volta una tabella (anche complessa). Ogni istruzione di manipolazione è introdotta dalla keyword **`retrieve`**, seguita da un predicato racchiuso tra parentesi quadre e dal nome della tabella operando. Il predicato è una lista di condizioni logiche (**`and`**, **`or`**) in cui ogni condizione è un confronto di uguaglianza. Possono essere confrontati attributi semplici con altri attributi semplici o con costanti. Possono essere confrontati anche attributi complessi con altri attributi complessi (o tabelle) o con il risultato di una manipolazione di attributi complessi (o tabelle), senza limiti di innestamento delle **`retrieve`** nei predicati.

Esercizio 7

Specificare la EBNF di un linguaggio per definire e manipolare tabelle complesse, come nel seguente esempio:

```
def T1:(a: int, b: string, c: bool, d: int);
def T2:(alfa: int, beta: int, gamma:(x: int, y: string));
def T3:(m: int, r:(v: int, w: string, s:(p: string, q: string), n: int), t: bool);
…
retrieve [a=d] T1;
retrieve [a=d and b="sole"] T1;
retrieve [a=d and b="sole" or c=false] T1;
retrieve [alfa=15 and retrieve [x=4 or y="stella"] gamma = T5] T2;
retrieve [beta=10 and retrieve [x=20] gamma = retrieve [y="luna"] gamma] T2;
```

program → {*stat* **;**}$^{+}$
stat → (*def-stat* | *retrieve-op*)
def-stat → **def id : (** *attr-list* **)**
attr-list → *attr-decl* {**,** *attr-decl*}
attr-decl → **id :** *domain*
domain → **int** | **string** | **bool** | **(** *attr-list* **)**
retrieve-op → **retrieve [** *predicate* **] id**
predicate → *cond* {(**and** | **or**) *cond*}
cond → (**id** | *retrieve-op*) **=** (**id** | *retrieve-op* | *const*)
const → **intconst** | **stringconst** | **boolconst**

Esercizio 8

Specificare la EBNF di un linguaggio per definire *ambienti computazionali*, come nel seguente esempio:

```
env alfa is
    a, b: integer;
    c: string;
    function beta(x: real, y: matrix[2,3] of string): real;
    function gamma(z: string, w: matrix[2] of matrix[5,6,7] of integer): matrix[10] of real;
    d, e, f: matrix[2,20] of integer;
    env delta is
        function epsilon(t: string): integer;
        g: real;
        h: string;
    end;
    k: integer;
    function zeta(): string;
    env lambda is
        m: matrix[10,20] of integer;
    end;
    function omega(s: string): string;
end;
```

Un ambiente è definito da un nome (racchiuso dalle keyword **`env`** ed **`is`**) e termina con la keyword **`end`**. Il corpo dell'ambiente è costituito da una lista non vuota di dichiarazioni che possono essere liste di variabili, funzioni o ambienti. Non esiste un ordine predefinito per tali dichiarazioni. Una lista di variabili è definita dal nome delle variabili seguite dal tipo. Un tipo può essere semplice (**`integer`**, **`real`**, **`string`**) o matriciale (**`matrix`**). Un tipo matriciale specifica una matrice di $n \geq 1$ dimensioni, in cui ogni dimensione è definita da un intero positivo. Una funzione è definita da un nome, dalla lista (eventualmente vuota) di parametri e dal tipo del valore di ritorno.

Esercizio 8

Specificare la EBNF di un linguaggio per definire *ambienti computazionali*, come nel seguente esempio:

```
env alfa is
    a, b: integer;
    c: string;
    function beta(x: real, y: matrix[2,3] of string): real;
    function gamma(z: string, w: matrix[2] of matrix[5,6,7] of integer): matrix[10] of real;
    d, e, f: matrix[2,20] of integer;
    env delta is
        function epsilon(t: string): integer;
        g: real;
        h: string;
    end;
    k: integer;
    function zeta(): string;
    env lambda is
        m: matrix[10,20] of integer;
    end;
    function omega(s: string): string;
end;
```

program → *env-decl*
env-decl → **env id is** { *var-list-decl* | *function-decl* | *env-decl* }$^{+}$ **end** ;
var-list-decl → *id-list* **:** *type* **;**
id-list → **id** { **,** **id** }
type → **integer** | **real** | **string** | *matrix-type*
matrix-type → **matrix [** *num-list* **]** **of** *type*
num-list → **num** { **,** **num** }
function-decl → **function id (** [*param-list*] **) :** *type* **;**
param-list → *param-decl* { **,** *param-decl* }
param-decl → **id :** *type*

Esercizio 9

Specificare la EBNF di un linguaggio imperativo in cui ogni programma è composto da una intestazione, da una sezione (opzionale) di dichiarazione di variabili e da un corpo, come nel seguente esempio:

```
program primavera
  var a, b1, beta, i: int;
      c, z: real;
      d, e, omega25: string;
  begin
    a := 10;
    b1 := a;
    if a > 3 then
      c := 3.0;
      z := 12.24;
    elsif a < b then
      z := a;
    elsif a = b then
      d := "alfa";
      for i := 1 to 25 do
        if c > z then
          omega25 := e;
        endif;
      endfor;
    otherwise
      while a != i do
        a := b1;
        b1 := 3;
      endwhile;
    endif;
  end.
```

Le variabili possono essere di tipo **int**, **real** o **string**. Il corpo del programma è costituito da una sequenza non vuota di istruzioni racchiusa tra **begin** ed **end**. Ogni istruzione può essere un assegnamento, una istruzione condizionale a più vie, un ciclo a condizione iniziale o un ciclo a conteggio. Possono essere assegnate variabili con altri variabili o valori. Una istruzione condizionale **if** … **endif** coinvolge zero o più rami **elsif** e, opzionalemte, il ramo **otherwise**. Una condizione è il confronto (**=**, **!=**, **>**, **<**, **>=**, **<=**) tra una variabile e un valore o un'altra variabile. Ad ogni iterazione del ciclo a conteggio **for … endfor**, la variabile di conteggio può incrementare (**to**) o decrementare (**downto**). Il corpo dei cicli (**for** e **while**), del **then** e dell'**otherwise** è costituito da una lista non vuota di istruzioni.

Esercizio 9

Specificare la EBNF di un linguaggio imperativo in cui ogni programma è composto da una intestazione, da una sezione (opzionale) di dichiarazione di variabili e da un corpo, come nel seguente esempio:

```
program primavera
  var a, b1, beta, i: int;
      c, z: real;
      d, e, omega25: string;
  begin
    a := 10;
    b1 := a;
    if a > 3 then
      c := 3.0;
      z := 12.24;
    elsif a < b then
      z := a;
    elsif a = b then
      d := "alfa";
      for i := 1 to 25 do
        if c > z then
          omega25 := e;
        endif;
      endfor;
    otherwise
      while a != i do
        a := b1;
        b1 := 3;
      endwhile;
    endif;
  end.
```

program → **program id** [*var-section*] *body* **.**
var-section → **var** { *decl-list* **;** }$^{+}$
decl-list → **id** { **,** **id** } **:** *type*
type → **int** | **real** | **string**
body → **begin** *block* **end**
block → { *stat* **;** }$^{+}$
stat → *assign-stat* | *if-stat* | *while-stat* | *for-stat*
assign-stat → **id** **:=** (*const* | **id**)
const → **intconst** | **realconst** | **stringconst**
if-stat → **if** *cond* **then** *block* { **elsif** *cond* **then** *block* } [**otherwise** *block*] **endif**
cond → **id** (**=** | **!=** | **>** | **<** | **>=** | **<=**) (*const* | **id**)
while-stat → **while** *cond* **do** *block* **endwhile**
for-stat → **for id := intconst (to | downto) intconst do** *block* **endfor**

Esercizio 10

Specificare la grammatica EBNF dei pattern di un linguaggio *Haskell*-like. Ecco alcuni esempi di pattern:

```
0
12
'a'
0.123
104.27896
True
x
num2

_
(x, y2, False,'w')
[1, 2, _, n, m]
testa:coda
x:y:[]
(2, True, ([1,2,3], x:[_, y], 'q'))
[(_,12), ('a', 0), ('b', 246), (_,n), _]
```

Ogni frase del linguaggio contiene un solo pattern. Semplici pattern possono essere costanti carattere (ad esempio `'a'`), costanti booleane (`True`, `False`) , costanti intere (senza zeri prefissi non significativi, ad esempio `102` ma non `0012`), costanti reali (ad esempio `12.34`, `0.025`, ma non `002.23`), nomi di variabili (necessariamente alfanumeriche, ad esempio `x`, `num3`, `area`, ma non `3num`), ed il carattere underscore `_`. Pattern più complessi possono essere specificati mediante i simboli di tupla (ad esempio `(3,x)`) o di lista (ad esempio `[1,2,n]`). Infine, un pattern può essere specificato mediante il costruttore di lista (ad esempio `n:[1,2,3]`). Sussiste piena ortogonalità tra tuple e liste. Si assumono i seguenti simboli lessicali (di cui non è richiesta la specifica): **true**, **false** (costanti boolenae), **char** (un qualsiasi carattere), **alpha** (un carattere alfabetico), **digit** (una qualsiasi cifra), **nonzero** (una cifra diversa da zero).

Esercizio 10

pattern → *simple-pattern* | *complex-pattern*
simple-pattern → *char-pattern* | *bool-pattern* | *int-pattern* | *real-pattern* | *var-pattern* | **_**
char-pattern → **'char'**
bool-pattern → **true** | **false**
int-pattern → **digit** | (**nonzero** { **digit** }$^{+}$)
real-pattern → *int-pattern* **.** { **digit** }$^{+}$
var-pattern → **alpha** { (**alpha** | **digit**) }
complex-pattern → *tuple-pattern* | *list-pattern* | *cons-pattern*
tuple-pattern → **(** *pattern-sequence* **)**
pattern-sequence → *pattern* { **,** *pattern* }
list-pattern → **[** [*pattern-sequence*] **]**
cons-pattern → *pattern* **:** (*var-pattern* | **_** | *list-pattern* | *cons-pattern*)

```
0
12
'a'
0.123
104.27896
True
x
num2

_
(x, y2, False,'w')
[1, 2, _, n, m]
testa:coda
x:y:[]
(2, True, ([1,2,3], x:[_, y], 'q'))
[(_,12), ('a', 0), ('b', 246), (_,n), _]
```

Esercizio 11

Specificare la grammatica EBNF di un linguaggio imperativo, in cui ogni frase definisce una lista (non vuota) di procedure. Ecco un esempio di definizione di procedura:

```
proc alfa(in a,b: int, out s: string, out r: record(x,y: real, i: int), inout j: int)
  v, w: real;
  m,n: int;
  dispari: set of int;
  t: record(e,f: int, s: set of real);
  s1, s2: set of record(a: int, b: set of record (c: string, d: set of int));
  nome, cognome: string;
begin
  r.x := 2.34;
  r.y := r.x;
  m := 10;
  n := m;
  dispari := [1,3,5,7,9];
  t.e := n;
  t.s := [1.0, 3.14, 12.375, 0.01];
  s1 := [(2, [("sole", [4, 6, 10]), ("luna", []),("terra", [1,100])]),
         (5, [("giove", [16]), ("marte", [2,7,9])]),
         (3, [])];
  s2 := s1;
  s1 := [];
  nome := "luigi";
  cognome := "rossi";
end;
```

Ogni procedura ha una lista (non vuota) di parametri. Ogni parametro è qualificato mediante una keyword che ne specifica la modalità di passaggio: **`in`**, **`out`**, **`inout`**. Dopo l'intestazione, è possibile (ma non necessario) definire un insieme di variabili locali. I tipi atomici sono **`int`**, **`real`** e **`string`**. I costruttori di tipo (perfettamente ortogonali tra loro) sono **`record`** e **`set`** (insieme). Il corpo della procedura è costituito da una lista non vuota di assegnamenti, in cui la parte sinistra è un identificatore (di variabile o parametro) o un campo (eventualmente annidato) di record. La parte destra dell'assegnamento può essere una costante, un identificatore o un campo (eventualmente annidato) di record. Una costante di tipo **`record`** è racchiusa tra parentesi tonde. Una costante di tipo **`set`** è racchiusa tra parentesi quadre.

Esercizio 11

```
proc alfa(in a,b: int, out s: string, out r: record(x,y: real, i: int), inout j: int)
  v, w: real;
  m,n: int;
  dispari: set of int;
  t: record(e,f: int, s: set of real);
  s1, s2: set of record(a: int, b: set of record (c: string, d: set of int));
  nome, cognome: string;
begin
  r.x := 2.34;
  r.y := r.x;
  m := 10;
  n := m;
  dispari := [1,3,5,7,9];
  t.e := n;
  t.s := [1.0, 3.14, 12.375, 0.01];
  s1 := [(2, [("sole", [4, 6, 10]), ("luna", []),("terra", [1,100])]),
         (5, [("giove", [16]), ("marte", [2,7,9])]),
         (3, [])];
  s2 := s1;
  s1 := [];
  nome := "luigi";
  cognome := "rossi";
end;
```

program → { *proc-decl* }$^{+}$
proc-decl → **proc (** *param-list* **)** [*var-section*] **begin** { *assign* }$^{+}$ **end ;**
param-list → *params-decl* { **,** *params-decl* }
params-decl → *qualifier id-list* **:** *type*
qualifier → **in** | **out** | **inout**
id-list → **id** { **,** **id** }
type → **int** | **real** | **string** | *rec-type* | *set-type*
rec-type → **record (** *id-list* **:** *type* { **,** *id-list* **:** *type* } **)**
set-type → **set of** *type*
var-section → { *id-list* **:** *type* **;** }$^{+}$
assign → *lhs* **:=** *rhs* **;**
lhs → **id** { **.** **id** }
rhs → *lhs* | *const*
const → **intconst** | **realconst** | **strconst** | *rec-const* | *set-const*
rec-const → **(** *const* { **,** *const* } **)**
set-const → **[** [*const* { **,** *const* }] **]**

Esercizio 12

Specificare la grammatica EBNF di un linguaggio per la specifica di un programma basato sul paradigma funzionale. Un programma si compone di tre sezioni, di cui le prime due sono opzionali, mentre la terza è obbligatoria. Nella prima sezione vengono dichiarate le variabili mediante un tipo (intero o booleano) ed una costante corrispondente. Nella seconda sezione vengono definite le funzioni in termini di protocollo (lista dei parametri formali) e corpo (espressione). La terza sezione specifica l'espressione del programma. Ogni espressione può coinvolgere operatori aritmetici (**+**, **-**, *****, */*) o logici (**and**, **or**), nomi di variabili e/o parametri formali, costanti (intere o booleane) e chiamate di funzione (nome della funzione seguita da una lista di espressioni, i parametri attuali). Ecco un esempio di programma:

```
int a = 3, b = 5;
bool c = true, d = false;

function alfa(x) = (x and c) or d;
function beta(x, y) = ((a + x) * (b - y)) / 2;

(beta(a, b+1) + beta(b, a-b)) - 10;
```

Esercizio 12

Specificare la grammatica EBNF di un linguaggio per la specifica di un programma basato sul paradigma funzionale. Un programma si compone di tre sezioni, di cui le prime due sono opzionali, mentre la terza è obbligatoria. Nella prima sezione vengono dichiarate le variabili mediante un tipo (intero o booleano) ed una costante corrispondente. Nella seconda sezione vengono definite le funzioni in termini di protocollo (lista dei parametri formali) e corpo (espressione). La terza sezione specifica l'espressione del programma. Ogni espressione può coinvolgere operatori aritmetici (**+**, **-**, *****, */*) o logici (**and**, **or**), nomi di variabili e/o parametri formali, costanti (intere o booleane) e chiamate di funzione (nome della funzione seguita da una lista di espressioni, i parametri attuali). Ecco un esempio di programma:

```
int a = 3, b = 5;
bool c = true, d = false;

function alfa(x) = (x and c) or d;
function beta(x, y) = ((a + x) * (b - y)) / 2;

(beta(a, b+1) + beta(b, a-b)) - 10;
```

program → { *var-decl* } { *function-decl* } *expr* **;**
var-decl → *type var* { **,** *var* } **;**
type → **int** | **bool**
var → **id** = *const*
const → **intconst** | **boolconst**
function-decl → **function id (** [*id-list*] **)** = *expr* **;**
id-list → **id** { **,** **id** }
expr → *expr* + *expr* | *expr* – *expr* | *expr* * *expr* | *expr* / *expr* |
expr **and** *expr* | *expr* **or** *expr* |
(*expr* **)** | **id** | *const* | *call*
call → **id (** [*expr-list*] **)**
expr-list → *expr* { **,** *expr* }

Esercizio 13

Specificare la grammatica EBNF di un linguaggio di interrogazione SQL-like, in cui ogni frase comprende una o più interrogazioni, come nel seguente esempio:

```
SELECT voto
FROM Esami;

SELECT nome, cognome
FROM Persone
WHERE citta = 'milano';

SELECT AVG(voto) AS media
FROM Esami
WHERE matricola = 1246;

SELECT Studenti.matricola AS codice, AVG(Esami.voto)
FROM Studenti, Esami
WHERE Studenti.matricola = Esami.matricola AND
      Studenti.matricola > 1000 AND Studenti.matricola < 2000
GROUP BY Studenti.matricola;

SELECT nome, cognome, MIN(voto), AVG(voto), MAX(voto)
FROM Studenti, Esami
WHERE Studenti.matricola = Esami.matricola AND
      Studenti.matricola > 1000 AND Studenti.matricola < 2000
GROUP BY Studenti.matricola
HAVING COUNT(*) > 10 AND MIN(voto) > 20
ORDER BY cognome, nome;
```

In ogni interrogazione, le clausole **SELECT** e **FROM** sono obbligatorie, mentre le clausole **WHERE**, **GROUP BY**, **HAVING** e **ORDER BY** sono facoltative. Inoltre, la clausola **HAVING** può essere specificata solo se è stata specificata la clausola **GROUP BY**. La clausola **SELECT** specifica una lista di attributi di relazione e/o funzioni aggregate su attributi. Possibili funzioni aggregate sono **COUNT**, **MIN**, **MAX** ed **AVG**. La funzione **COUNT** ha sempre come argomento un asterisco, mentre le altre funzioni hanno come argomento un attributo di relazione. Il nome di un attributo può essere preceduto dal nome della relazione di cui fa parte. Ogni elemento listato nella clausola **SELECT** può essere ridenominato mediante la keyword **AS** seguita da un identificatore. La clausola **FROM** specifica una lista di nomi di relazioni. La clausola **WHERE** specifica il predicato di selezione, composto in generale da una catena di congiunzioni di condizioni. Una condizione è il confronto tra un attributo ed una costante (intera o stringa) o un altro attributo. Possibili confronti sono **=**, **<** , **>**. La clausola **GROUP BY** specifica una lista di attributi di raggruppamento. La clausola **HAVING** specifica un predicato sui gruppi di tuple stabiliti dalla clausola **GROUP BY**. Le condizioni espresse dalla clausola **HAVING** sono rappresentate da un confronto tra una funzione aggregata ed una costante intera o un'altra funzione aggregata. Infine, la clausola **ORDER BY** specifica una lista di attributi sui quali ordinare le tuple risultanti. Ogni interrogazione termina con un punto e virgola.

Esercizio 13

Specificare la grammatica EBNF di un linguaggio di interrogazione SQL-like, in cui ogni frase comprende una o più interrogazioni, come nel seguente esempio:

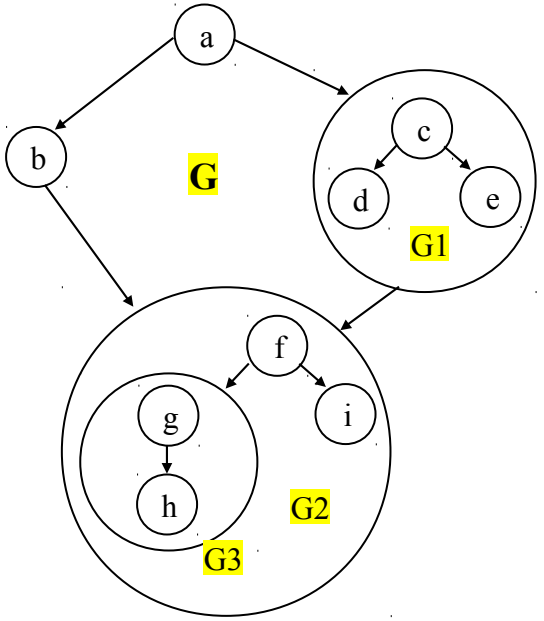
```
SELECT voto
FROM Esami;

SELECT nome, cognome
FROM Persone
WHERE citta = 'milano';

SELECT AVG(voto) AS media
FROM Esami
WHERE matricola = 1246;

SELECT Studenti.matricola AS codice, AVG(Esami.voto)
FROM Studenti, Esami
WHERE Studenti.matricola = Esami.matricola AND
      Studenti.matricola > 1000 AND Studenti.matricola < 2000
GROUP BY Studenti.matricola;

SELECT nome, cognome, MIN(voto), AVG(voto), MAX(voto)
FROM Studenti, Esami
WHERE Studenti.matricola = Esami.matricola AND
      Studenti.matricola > 1000 AND Studenti.matricola < 2000
GROUP BY Studenti.matricola
HAVING COUNT(*) > 10 AND MIN(voto) > 20
ORDER BY cognome, nome;
```

program → { *query* **;** }$^{+}$
query → select-clause
from-clause
[*where-clause*]
[*group-by-clause* [*having-clause*]]
[*order-by-clause*]
select-clause → **SELECT** *target-id* { **,** *target-id* }
target-id → (*attr-id* | *aggr-func*) [**AS id**]
attr-id → [**id.**] **id**
aggr-func → **COUNT(*)** | (**MIN** | **MAX** | **AVG**)(*attr-id*)
from-clause → **FROM** *id-list*
id-list → **id** { **,** **id** }
where-clause → **WHERE** *cond* { **AND** *cond* }
cond → *attr-id comp-op* (*attr-id* | **intconst** | **strconst**)
comp-op → = | < | >
group-by-clause → **GROUP BY** *attr-id-list*
attr-id-list → *attr-id* { **,** *attr-id* }
having-clause → **HAVING** *aggr-cond* { **AND** *aggr-cond* }
aggr-cond → *aggr-func comp-op* (*aggr-func* | **intconst**)
order-by-clause → **ORDER BY** *attr-id-list*

Esercizio 14

Specificare la grammatica EBNF di un linguaggio in cui ogni frase specifica un grafo, come nel seguente esempio:

```
graph G is
  nodes a,
        b,
        graph G1 is
          nodes c, d, e
          arcs (c,d), (c,e)
        end G1,
        graph G2 is
          nodes f,
                i,
                graph G3 is
                  nodes g, h
                  arcs (g,h)
                end G3,
          arcs (f,G3), (f,i)
        end G2
  arcs (a,b), (a,G1), (b,G2), (G1,G2)
end G
```

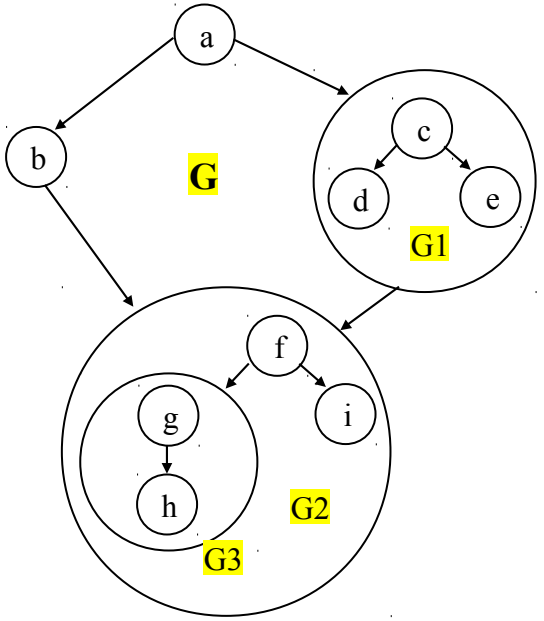


Un grafo è definito da un nome, una lista di nodi ed una lista di archi. Un nodo può essere a sua volta un grafo, senza limiti di profondità. Ogni arco è definito dalla coppia di identificatori dei nodi congiunti dall'arco.
(Si faccia attenzione alle regole di punteggiatura indicate nell'esempio.)

Esercizio 14

Specificare la grammatica EBNF di un linguaggio in cui ogni frase specifica un grafo, come nel seguente esempio:

```
graph G is
  nodes a,
        b,
        graph G1 is
          nodes c, d, e
          arcs (c,d), (c,e)
        end G1,
        graph G2 is
          nodes f,
                i,
                graph G3 is
                  nodes g, h
                  arcs (g,h)
                end G3,
          arcs (f,G3), (f,i)
        end G2
  arcs (a,b), (a,G1), (b,G2), (G1,G2)
end G
```



Un grafo è definito da un nome, una lista di nodi ed una lista di archi. Un nodo può essere a sua volta un grafo, senza limiti di profondità. Ogni arco è definito dalla coppia di identificatori dei nodi congiunti dall'arco.
(Si faccia attenzione alle regole di punteggiatura indicate nell'esempio.)

program → *graph-def*
graph-def → **graph id is nodes** *node-list* **arcs** *arc-list* **end id**
node-list → *node-def* { **,** *node-def* }
node-def → **id** | *graph-def*
arc-list → *arc-def* { **,** *arc-def* }
arc-def → **(id, id)**

Esercizio 15

Specificare la grammatica EBNF di un linguaggio logico *Prolog*-like, come nel seguente esempio di frase:

```
alfa(lio2,3).
beta(teo,rio(X,Y)).
gamma(_,Z,[1,2,meo(25,X,beta(Y,rio(a,b))),legge(rino,libro),10]).
zeta(X,Y) :- beta(X,Z1), alfa(Y,Z1).
omega([a,b,X|Y]) :- tilde([X|[alfa(X,Y),beta(Z,W)]]), beta(_,W).
epsilon(geo(neo(X,Y),Z,[])) :- omega(X), alfa(Y,Z).
```

Una frase è costituita da una lista (non vuota) di fatti e/o regole. Gli argomenti di ogni predicato nei fatti e nelle regole possono essere semplici o complessi. Nel primo caso si tratta di atomi (stringhe alfanumeriche), numeri, variabili con nome o variabili anonime (rappresentate dal simbolo underscore '_'). Nel secondo caso, si tratta di strutture o liste. I costruttori struttura e lista sono perfettamente ortogonali fra loro. Inoltre, una lista può essere espressa mediante il pattern [*prefisso* | *coda*] (come nell'esempio). Nella EBNF, si richiede l'uso (fra gli altri) dei seguenti terminali:

- **atom**: stringa alfanumerica (con lettera iniziale minuscola)
- **num**: numero (stringa di cifre decimali)
- **var**: nome di variabile (con lettera iniziale maiuscola)
- **underscore**: simbolo di underscore (variabile anonima)
- **id**: nome di predicato.

La parte destra di ogni regola è rappresentata da una lista di predicati separati da una virgola e terminata da un punto. Tutti i predicati, strutture e fatti hanno almeno un argomento.

Esercizio 15

Specificare la grammatica EBNF di un linguaggio logico *Prolog*-like, come nel seguente esempio di frase:

```
alfa(lio2,3).
beta(teo,rio(X,Y)).
gamma(_,Z,[1,2,meo(25,X,beta(Y,rio(a,b))),legge(rino,libro),10]).
zeta(X,Y) :- beta(X,Z1), alfa(Y,Z1).
omega([a,b,X|Y]) :- tilde([X|[alfa(X,Y),beta(Z,W)]]), beta(_,W).
epsilon(geo(neo(X,Y),Z,[])) :- omega(X), alfa(Y,Z).
```

program → { *fatto* | *regola* }$^{+}$
fatto → *predicato* **.**
regola → *predicato* **:-** *predicato* { **,** *predicato* } **.**
predicato → **id** **(** *arg* { **,** *arg* } **)**
arg → *semplice* | *complesso*
semplice → **atom** | **num** | **var** | **underscore**
complesso → *struttura* | *lista*
struttura → *predicato*
lista → **[** [*arg* { **,** *arg* } [| (*var* | *underscore* | *lista*)]] **]**

Esercizio 16

Specificare la grammatica EBNF di un linguaggio per la definizione di tipi di dati, come nel seguente esempio:

```
type Giorni is (Lun, Mar, Mer, Gio, Ven, Sab, Dom);

subtype Lavorativi is Giorni range {Lun..Ven};

subtype Indice is Integer range {1..100};

type Temperatura: Real;

type Temperature is array [1..10] of Temperatura;

type Tabella is array [1..7] of Temperature;

type Studente is record
  nome: String;
  cognome: String;
  matricola: Integer;
end record;
```

Ogni frase contiene almeno una definizione. I tipi primitivi sono **`Integer`**, **`Real`**, **`Bool`** e **`String`**. Il tipo enumerativo elenca i suoi elementi in una lista di identificatori. Si possono definire sottotipi di interi o enumerativi specificando il range. Esistono due costruttori (ortogonali) di tipo: array e record, i quali possono essere applicati unicamente a nomi di tipi, sottotipi o tipi primitivi. Si richiede di massimizzare i vincoli sintattici.

Esercizio 16

```
type Giorni is (Lun, Mar, Mer, Gio, Ven, Sab, Dom);

subtype Lavorativi is Giorni range {Lun..Ven};

subtype Indice is Integer range {1..100};

type Temperatura: Real;

type Temperature is array [1..10] of Temperatura;

type Tabella is array [1..7] of Temperature;

type Studente is record
  nome: String;
  cognome: String;
  matricola: Integer;
end record;
```

module → { *def* **;** }$^{+}$
def → *type-def* | *subtype-def*
type-def → **type id is** (*primitive-type* | *enum-type* | *array-type* | *record-type*)
primitive-type → **Integer** | **Real** | **Bool** | **String**
enum-type → **(id** { **, id** } **)**
subtype-def → **subtype id is** (*integer-subtype* | *enum-subtype*)
integer-subtype → **Integer range { num .. num }**
enum-subtype → **id range { id .. id }**
array-type → **array [num .. num] of** *type-name*
type-name → (*primitive-type* | **id**)
record-type → **record (** { **id :** *type-name* **;** }$^{+}$ **) end record**

Esercizio 17

Specificare la EBNF di un linguaggio per la dichiarazione e assegnamento di variabili, come nel seguente esempio:

```
a, b: integer;
s: string;
alfa: record (x: integer, y: string, beta: record (z: integer, w: integer));
v1: vector [5] of integer;
v2: vector [10] of vector [4] of integer;
a = 1;
s = "buongiorno";
b = a + 10;
alfa.x = b + 2 - a;
v1[a - v1[3]] = alfa.beta.z + (4 - v1[a-b]);
v2[7][3] = a + b - 2;
```

Tutte le dichiarazioni (se esistono) devono precedere gli assegnamenti (se esistono). Le variabili possono essere di tipo intero, stringa, record o vettore. I costruttori di tipo sono ortogonali fra loro. Negli assegnamenti, la parte sinistra può anche essere il campo di un record o l'elemento di un vettore. Le espressioni (di indicizzazione e di assegnamento) possono coinvolgere le operazioni aritmetiche di somma e differenza.

Esercizio 17

Specificare la EBNF di un linguaggio per la dichiarazione e assegnamento di variabili, come nel seguente esempio:

```
a, b: integer;
s: string;
alfa: record (x: integer, y: string, beta: record (z: integer, w: integer));
v1: vector [5] of integer;
v2: vector [10] of vector [4] of integer;
a = 1;
s = "buongiorno";
b = a + 10;
alfa.x = b + 2 - a;
v1[a - v1[3]] = alfa.beta.z + (4 - v1[a-b]);
v2[7][3] = a + b - 2;
```

Tutte le dichiarazioni (se esistono) devono precedere gli assegnamenti (se esistono). Le variabili possono essere di tipo intero, stringa, record o vettore. I costruttori di tipo sono ortogonali fra loro. Negli assegnamenti, la parte sinistra può anche essere il campo di un record o l'elemento di un vettore. Le espressioni (di indicizzazione e di assegnamento) possono coinvolgere le operazioni aritmetiche di somma e differenza.

program → { *def* } { *assign* }
def → **id** { **,** **id** } **:** *type* **;**
type → **integer** | **string** | *record-type* | *vector-type*
record-type → **record (id :** *type* { **,** **id :** *type* }**)**
vector-type → **vector [intconst] of** *type*
assign → *lhs* **=** *expr* **;**
lhs → **id** | *lhs* **.** **id** | *lhs* **[***expr***]**
expr → *lhs* | **intconst** | **strconst** | *expr* **+** *expr* | *expr* **–** *expr* | **(** *expr* **)**

Esercizio 18

Specificare la grammatica EBNF di un sottoinsieme **P** del linguaggio *Prolog*, in cui ogni frase è una sequenza non vuota di clausole, come nel seguente esempio (non esaustivo):

```
alfa(a,X,_).
beta([1,2,[H|T]]).
gamma(opt([X,Y,Z|W],p(5))).
r(X,Y) :- alfa(_,X,2), beta([1,2,Y]), gamma(_).
r(_,[]) :- alfa(a,b,c).
```

Il linguaggio **P** comprende strutture (con uno o più argomenti), liste (eventualmente rappresentate mediante pattern), atomi (numeri e stringhe), variabili e il carattere '_' (variabile anonima). Strutture e liste sono ortogonali fra loro. Nella specifica della EBNF si fa uso (tra gli altri) dei simboli lessicali **atom** (atomo) e **var** (variabile).

Esercizio 18

Specificare la grammatica EBNF di un sottoinsieme **P** del linguaggio *Prolog*, in cui ogni frase è una sequenza non vuota di clausole, come nel seguente esempio (non esaustivo):

```
alfa(a,X,_).
beta([1,2,[H|T]]).
gamma(opt([X,Y,Z|W],p(5))).
r(X,Y) :- alfa(_,X,2), beta([1,2,Y]), gamma(_).
r(_,[]) :- alfa(a,b,c).
```

Il linguaggio **P** comprende strutture (con uno o più argomenti), liste (eventualmente rappresentate mediante pattern), atomi (numeri e stringhe), variabili e il carattere '_' (variabile anonima). Strutture e liste sono ortogonali fra loro. Nella specifica della EBNF si fa uso (tra gli altri) dei simboli lessicali **atom** (atomo) e **var** (variabile).

program → {*clause* **.**}$^+$
clause → *struct* | *rule*
struct → **id (** *param*, {**,** *param*} **)**
param → **atom** | **var** | '_ '| *list* | *struct*
list → **[** [*param* {**,** *param*} ['|' (**var** | _ | *list*)]] **]**
rule → *struct* **:-** *struct* {**,** *struct* }

Esercizio 19

Specificare la grammatica EBNF di un linguaggio per la dichiarazione e assegnamento di variabili, come nel seguente esempio:

```
i, j: scalar;
alfa: record (x: scalar, y: array [3,7,10] of scalar);
vector: array[100] of scalar;
complex: array[2,5] of array[3] of record(a: scalar, b: scalar);
i = j + 1;
j = (i + j) * (i - j / 25);
alfa.x = alfa.y[2,4,6];
complex[1,3] = complex [1,4] - 15;
alfa.y[1,2,j+2] = vector[1 + j - vector[20] / alfa.x];
```

Ogni frase si compone di almeno una istruzione. Una variabile può essere scalare (cioè, intera) o strutturata. Esistono due costruttori di tipo (ortogonali fra loro): il record e l'array multidimensionale (in cui ogni dimensione viene specificata da un intero positivo). Le espressioni di assegnamento coinvolgono le quattro operazioni aritmetiche, le espressioni di indicizzazione degli array e l'estrazione di campi dei record, senza limiti di ortogonalità.

Esercizio 19

Specificare la grammatica EBNF di un linguaggio per la dichiarazione e assegnamento di variabili, come nel seguente esempio:

```
i, j: scalar;
alfa: record (x: scalar, y: array [3,7,10] of scalar);
vector: array[100] of scalar;
complex: array[2,5] of array[3] of record(a: scalar, b: scalar);
i = j + 1;
j = (i + j) * (i - j / 25);
alfa.x = alfa.y[2,4,6];
complex[1,3] = complex [1,4] - 15;
alfa.y[1,2,j+2] = vector[1 + j - vector[20] / alfa.x];
```

program → {*stat*;}$^{+}$
stat → (*def-stat* | *assign-stat*)
def-stat → **id** {, **id**} : *type*
type → **scalar** | *record-type* | *array-type*
record-type → **record (id:** *type* {, **id:** *type*}**)**
array-type → **array [num** {, **num**} **] of** *type*
assign-stat → *lhs* = *expr*
lhs → **id** | *lhs*.**id** | *lhs* **[** *expr* {, *expr*} **]**
expr → *lhs* | **num** | *expr* (+ | - | * | /) *expr* | **(***expr***)**

Esercizio 20

Specificare la grammatica EBNF di un linguaggio per la manipolazione di record, come nel seguente esempio:

```
r1, r2: (a: int, b: bool, c: string, d: (x: int, y: int));
r1.a = (r2.d.x - r2.d.y) / 10 + r1.a;
alfa: (n: int, s: string, w: bool);
r1 = r2;
alfa.s = r1.c;
r1.b = r2.b and (alfa.w or r2.b) or r1.b;
r2.b = true;
```

La frase può essere vuota. Esistono solo due tipi di istruzioni: dichiarazione di record e assegnamento. Ogni attributo di un record può essere un intero, un booleano, una stringa o un record (senza limiti di profondità). L'espressione di assegnamento non può mischiare elementi aritmetici (operatori e costanti numeriche) con elementi booleani (operatori e costanti booleane).

Esercizio 20

Specificare la grammatica EBNF di un linguaggio per la manipolazione di record, come nel seguente esempio:

```
r1, r2: (a: int, b: bool, c: string, d: (x: int, y: int));
r1.a = (r2.d.x - r2.d.y) / 10 + r1.a;
alfa: (n: int, s: string, w: bool);
r1 = r2;
alfa.s = r1.c;
r1.b = r2.b and (alfa.w or r2.b) or r1.b;
r2.b = true;
```

program → {*stat* **;**}
stat → *def-stat* | *assign-stat*
def-stat → **id** {**, id** } **:** *record*
record → **(***attr-def* { **,** *attr-def* }**)**
attr-def→ **id :** *type*
type → **int** | **string** | **bool** | *record*
assign-stat→ *lhs* **=** *expr*
lhs → **id** { **. id** }
expr → *int-expr* | *bool-expr*
int-expr → *int-expr int-operator int-expr* | **(** *int-expr* **)** | *lhs* | **intconst**
int-operator → **+** | **-** | ***** | **/**
bool-expr → *bool-expr bool-operator bool-expr* | **(** *bool-expr* **)** | *lhs* | **boolconst**
bool-operator → **and** | **or**