Esercizio 1

Specificare la grammatica BNF del linguaggio L, in cui ogni frase corrisponde ad una o più dichiarazioni. Ogni dichiarazione è composta da un tipo (**int**, **string** o **boolean**) e da una lista di identificatori di variabili con inizializzazione. Ogni variabile nella lista è immediatamente inizializzata con una costante appartenente a quel tipo, come nella seguente frase:

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

Esercizio 1

Specificare la grammatica BNF del linguaggio L, in cui ogni frase corrisponde ad una o più dichiarazioni. Ogni dichiarazione è composta da un tipo (**int**, **string** o **boolean**) e da una lista di identificatori di variabili con inizializzazione. Ogni variabile nella lista è immediatamente inizializzata con una costante appartenente a quel tipo, come nella seguente frase:

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

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

Esercizio 2

Specificare la grammatica BNF di un linguaggio in cui ogni frase corrisponde ad una sequenza non vuota di istruzioni di dichiarazione di variabili e/o di assegnamenti. Le variabili possono essere intere o reali. Le espressioni di assegnamento coinvolgono gli operatori somma e differenza, con possibilità di parentesi. Ecco un esempio di frase:

```
int alfa, beta, gamma;
real x, y, z;
beta := 10;
alfa := (beta + gamma) - (alfa +2);
x := y - (z + x - (alfa - 5));
```

Esercizio 2

Specificare la grammatica BNF di un linguaggio in cui ogni frase corrisponde ad una sequenza non vuota di istruzioni di dichiarazione di variabili e/o di assegnamenti. Le variabili possono essere intere o reali. Le espressioni di assegnamento coinvolgono gli operatori somma e differenza, con possibilità di parentesi. Ecco un esempio di frase:

```
int alfa, beta, gamma;
real x, y, z;
beta := 10;
alfa := (beta + gamma) - (alfa +2);
x := y - (z + x - (alfa - 5));
```

program → *stat-list*
stat-list → *stat-list* *stat* | *stat*
stat → *decl-stat* | *assign-stat*
decl-stat → *type decl-list* **;**
decl-list → *decl-list* **, id** | **id**
type → **int** | **real**
assign-stat → **id :=** *expr* **;**
expr → *expr* **+** *expr* | *expr* **–** *expr* | **(** *expr* **)** | **id** | **num**

Esercizio 3

Specificare la grammatica BNF di un linguaggio in cui ogni frase corrisponde ad una o più dichiarazioni di variabili. Ogni dichiarazione (che, ad eccezione dell'ultima, è separata dalla successiva mediante una virgola) è definita dal nome della variabile e dal suo tipo. Possibili tipi sono **int**, **real**, **string**, **record** e **function**. I primi tre tipi sono semplici. Il tipo **record** (terminato dalla keyword **end**) definisce una struttura i cui campi sono a loro volta caratterizzati da identificatori e relativi tipi. Il tipo function esprime il prototipo di una funzione avente uno o più parametri in ingresso ed un tipo in uscita preceduto dalla keyword **return**. Il linguaggio è perfettamente ortogonale, nel senso che le dichiarazioni sia di record che di funzioni possono mutuamente innestarsi senza limiti di profondità, come nella seguente frase:

```
alfa: int,
R: record A: string, B: record C: real, D: int end, E: string end,
F: function (X: int, Y: string) return string,
G: function (A: real, B: function (C: string) return int) return int,
H: record
    N: string,
    M: function (C: int) return function (D: string) return int,
    L: real
   end
```

Esercizio 3

Specificare la grammatica BNF di un linguaggio in cui ogni frase corrisponde ad una o più dichiarazioni di variabili. Ogni dichiarazione (che, ad eccezione dell'ultima, è separata dalla successiva mediante una virgola) è definita dal nome della variabile e dal suo tipo. Possibili tipi sono **int**, **real**, **string**, **record** e **function**. I primi tre tipi sono semplici. Il tipo **record** (terminato dalla keyword **end**) definisce una struttura i cui campi sono a loro volta caratterizzati da identificatori e relativi tipi. Il tipo function esprime il prototipo di una funzione avente uno o più parametri in ingresso ed un tipo in uscita preceduto dalla keyword **return**. Il linguaggio è perfettamente ortogonale, nel senso che le dichiarazioni sia di record che di funzioni possono mutuamente innestarsi senza limiti di profondità, come nella seguente frase:

```
alfa: int,
R: record A: string, B: record C: real, D: int end, E: string end,
F: function (X: int, Y: string) return string,
G: function (A: real, B: function (C: string) return int) return int,
H: record
     N: string,
     M: function (C: int) return function (D: string) return int,
     L: real
   end
```

program → *decl-list*
decl-list → *decl-list* **,** *decl* | *decl*
decl → **id :** *type*
type → **int** | **real** | **string** | *rec-type* | *func-type*
rec-type → **record** *decl-list* **end**
func-type → **function (** *decl-list* **) return** *type*

Esercizio 4

Specificare la grammatica BNF di un linguaggio logico L basato su clausole di Horn. Una frase di L è costituita da una lista non vuota di clausole (fatti o regole), come nel seguente esempio:

```
padre(guido, luisa).
padre(guido, elena).
genitore(X, Y) :- padre(X, Y).
madre(luisa, andrea).
madre(luisa, dario).
genitore(X, Y) :- madre(X, Y).
nonno(X, Y) :- padre(X, Z), genitore(Z, Y).
nonna(X, Y) :- madre(X, Z), genitore(Z, Y).
```

Gli argomenti dei predicati coinvolti nelle clausole sono semplicemente simboli alfanumerici. La parte destra di ogni regola coinvolge unicamente predicati in forma prefissa.

Esercizio 4

Specificare la grammatica BNF di un linguaggio logico L basato su clausole di Horn. Una frase di L è costituita da una lista non vuota di clausole (fatti o regole), come nel seguente esempio:

```
padre(guido, luisa).
padre(guido, elena).
genitore(X, Y) :- padre(X, Y).
madre(luisa, andrea).
madre(luisa, dario).
genitore(X, Y) :- madre(X, Y).
nonno(X, Y) :- padre(X, Z), genitore(Z, Y).
nonna(X, Y) :- madre(X, Z), genitore(Z, Y).
```

Gli argomenti dei predicati coinvolti nelle clausole sono semplicemente simboli alfanumerici. La parte destra di ogni regola coinvolge unicamente predicati in forma prefissa.

program → *clause-list*
clause-list → *clause-list clause* | *clause*
clause → *fact* **.** | *rule* **.**
fact → *predicate*
predicate → **id (** *param-list* **)**
param-list → *param-list* **,** **id** | **id**
rule → *predicate* **:-** *pred-list*
pred-list → *pred-list* **,** *predicate* | *predicate*

Esercizio 5

Specificare la grammatica BNF di un linguaggio in cui ogni frase corrisponde alla specifica di un package che incapsula la definizione di tipi e variabili, come nel seguente esempio:

```
package P is
  type
    T1, T2: record(a: int, b: real);
    T3: array [1..10] of string;
    T4, T5, T6: T1;
    T7: array [2..20] of array [3..6] of T3;
  var
    i, j: int;
    x, y, z: T2;
    v, w : record(s: string, v: array [1..100] of T7);
 end P
```

Il package ha un nome e due sezioni, una per i tipi e l'altra per le variabili. Le sezioni non possono essere vuote. I tipi semplici primitivi sono **int**, **real** e **string**. Esistono due costruttori (ortogonali) di tipo: il **record** e l'**array**. Per quest'ultimo, è necessario specificare il range dell'indice mediante due costanti intere. La sezione dei tipi permette di associare uno o più nomi ad una certa struttura dati. Tali nomi possono quindi comparire in dichiarazioni di tipi e/o variabili.

Esercizio 5

Specificare la grammatica BNF di un linguaggio in cui ogni frase corrisponde alla specifica di un package che incapsula la definizione di tipi e variabili, come nel seguente esempio:

```
package P is
  type
    T1, T2: record(a: int, b: real);
    T3: array [1..10] of string;
    T4, T5, T6: T1;
    T7: array [2..20] of array [3..6] of T3;
  var
    i, j: int;
    x, y, z: T2;
    v, w : record(s: string, v: array [1..100] of T7);
 end P
```

Il package ha un nome e due sezioni, una per i tipi e l'altra per le variabili. Le sezioni non possono essere vuote. I tipi semplici primitivi sono **int**, **real** e **string**. Esistono due costruttori (ortogonali) di tipo: il **record** e l'**array**. Per quest'ultimo, è necessario specificare il range dell'indice mediante due costanti intere. La sezione dei tipi permette di associare uno o più nomi ad una certa struttura dati. Tali nomi possono quindi comparire in dichiarazioni di tipi e/o variabili.

program → **package id is** *type-section var-section* **end id**
type-section → **type** *decl-list*
decl-list → *decl-list decl* | *decl*
decl → *id-list* **:** *domain* **;**
id-list → *id-list* **,** **id** | **id**
domain → *atomic-domain* | *record-domain* | *array-domain* | **id**
atomic-domain → **int** | **real** | **string**
record-domain → **record (** *pair-list* **)**
pair-list → **id :** *domain* **,** *pair-list* | **id :** *domain*
array-domain → **array [num .. num] of** *domain*
var-section → **var** *decl-list*

Esercizio 6

Specificare la grammatica BNF di un linguaggio ad eventi in cui ogni programma è costituito da una lista (non vuota) di trigger come nel seguente esempio:

```
define trigger propaga
  event alfa, beta, gamma
  condition alfa = beta, gamma > alfa
  action f1(x, y, z), f2(n), f3(m, alfa)
end propaga.

define trigger controlla
  event delta, epsilon
  condition delta != epsilon
  action g(a, b)
end controlla.
```

Ogni trigger ha un nome ed è definito in termini di regole ECA (*event-condition-action*), in cui le clausole *event* ed *action* sono obbligatorie, mentre la clausola *condition* è opzionale. La clausola *event* specifica una lista (non vuota) di eventi espressi come identificatori. La clusola *condition* specifica uno o più confronti semplici tra due eventi. Possibili operatori di confronto sono **=**, **!=**, **>**, **<**, **>=**, **<=**. La clausola *action* specifica una sequenza (non vuota) di chiamate di funzioni, in cui ogni funzione è applicata ad una lista (non vuota) di argomenti espressi come identificatori. La specifica del trigger si chiude con il nome del trigger definito nell'intestazione.

Esercizio 6

Specificare la grammatica BNF di un linguaggio ad eventi in cui ogni programma è costituito da una lista (non vuota) di trigger come nel seguente esempio:

```
define trigger propaga
  event alfa, beta, gamma
  condition alfa = beta, gamma > alfa
  action f1(x, y, z), f2(n), f3(m, alfa)
end propaga.

define trigger controlla
  event delta, epsilon
  condition delta != epsilon
  action g(a, b)
end controlla.
```

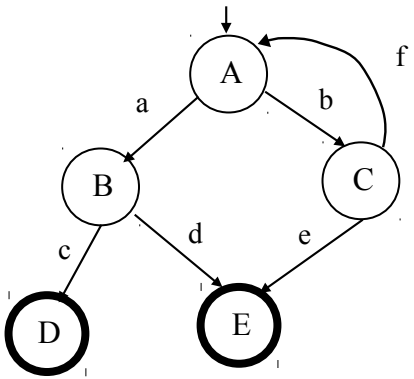
program → *trigger-list*
trigger-list → *trigger-def* *trigger-list* | *trigger-def*
trigger-def → **define trigger id** *event-clause condition-clause action-clause* **end id .**
event-clause → **event** *event-list*
event-list → **id ,** *event-list* | **id**
condition-clause → **condition** *comp-list* | **ε**
comp-list → *comparison* **,** *comp-list* | *comparison*
comparison → **id** *comp-op* **id**
comp-op → = | **!=** | > | < | >= | <=
action-clause → **action** *call-list*
call-list → *call* **,** *call-list* | *call*
call → **id (** *param-list* **)**
param-list → **id ,** *param-list* | **id**

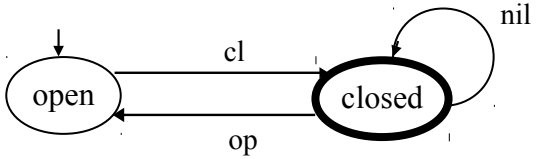
Esercizio 7

Specificare la grammatica BNF di un linguaggio per definire automi, in cui ogni programma è costituito da una lista (non vuota) di definizioni, come nel seguente esempio:

```
automaton alfa is
  states A, B, C, D, E;
  transitions
      from A to B on a,
      from A to C on b,
      from B to D on c,
      from B to E on d,
      from C to E on e,
      from C to A on f;
  initial A;
  final D, E;
end alfa.

automaton breaker is
  states open, closed;
  transitions
      from open to closed on cl,
      from closed to open on op,
      from closed to closed on nil;
  initial open;
  final closed;
end breaker.
```





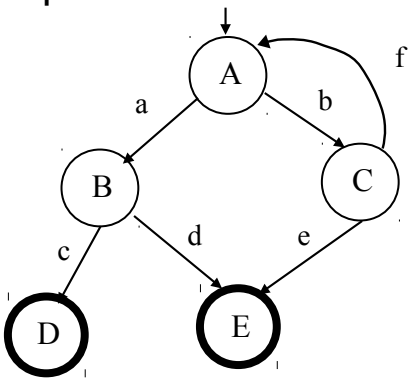
Ogni definizione include una intestazione, l'insieme (non vuoto) degli stati, l'insieme (non vuoto) delle transizioni, lo stato iniziale (uno ed uno solo), l'insieme (non vuoto) degli stati finali e una coda. La coda ripete il nome dell'automa nell'intestazione. Ogni transizione indica lo stato di partenza, lo stato di arrivo e l'evento da cui è attivata. La sezione **`initial`** è opzionale (in tal caso lo stato iniziale è implicitamente il primo stato di **`states`**).

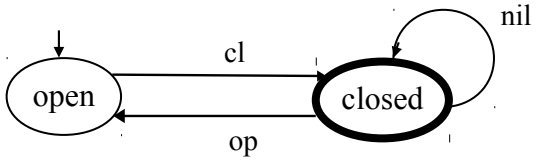
Esercizio 7

Specificare la grammatica BNF di un linguaggio per definire automi, in cui ogni programma è costituito da una lista (non vuota) di definizioni, come nel seguente esempio:

```
automaton alfa is
  states A, B, C, D, E;
  transitions
      from A to B on a,
      from A to C on b,
      from B to D on c,
      from B to E on d,
      from C to E on e,
      from C to A on f;
  initial A;
  final D, E;
end alfa.

automaton breaker is
  states open, closed;
  transitions
      from open to closed on cl,
      from closed to open on op,
      from closed to closed on nil;
  initial open;
  final closed;
end breaker.
```





program → *automaton-list*
automaton-list → *automaton-def automaton-list* | *automaton-def*
automaton-def → **automaton id is** *states-decl transitions-decl initial-decl final-decl* **end id .**
states-decl → **states** *id-list* **;**
id-list → **id ,** *id-list* | **id**
transitions-decl → **transitions** *trans-list* **;**
trans-list → *trans-decl* **,** *trans-list* | *trans-decl*
trans-decl → **from id to id on id**
initial-decl → **initial id ;** | **ε**
final-decl → **final** *id-list* **;**

Esercizio 8

Specificare la BNF di un linguaggio per definire ed istanziare relazioni complesse (non in prima forma normale), come nel seguente esempio:

```
relation R: [a: int, b: string, c: [d: int, e: real]];

relation S: [num: int, r: [t: [delta: int]]];

R := [(3, "alfa", [(10, 23.12)(12, 1.44)])
      (4, "beta", [(16, 3.56)])
      (5, "gamma", [])];

S := [(2, [([(10)(20)(30)])])
      (8, [([(124)(25)])])
      (28, [([])])
      (45, [])];

relation Delta: [x: int, y: int];

Delta := [];
```

Ogni programma contiene almeno una istruzione. Le istruzioni di definizione ed assegnamento delle relazioni possono essere specificate in qualsiasi ordine. Gli attributi atomici sono **int**, **real** e **string**. Non esiste limite di innestamento delle relazioni. Una relazione (o un attributo di tipo relazione) può essere istanziato con la relazione vuota **[ ]**.

Esercizio 8

Specificare la BNF di un linguaggio per definire ed istanziare relazioni complesse (non in prima forma normale), come nel seguente esempio:

```
relation R: [a: int, b: string, c: [d: int, e: real]];

relation S: [num: int, r: [t: [delta: int]]];

R := [(3, "alfa", [(10, 23.12)(12, 1.44)])
      (4, "beta", [(16, 3.56)])
      (5, "gamma", [])];

S := [(2, [([(10)(20)(30)])])
      (8, [([(124)(25)])])
      (28, [([])])
      (45, [])];

relation Delta: [x: int, y: int];

Delta := [];
```

program → *stat-list*
stat-list → *stat* *stat-list* | *stat*
stat → *def-stat* **;** | *assign-stat* **;**
def-stat → **relation id :** *rel-type*
rel-type → **[** *attr-list* **]**
attr-list → *attr-def* **,** *attr-list* | *attr-def*
attr-def → **id :** *type*
type → *atomic-type* | *rel-type*
atomic-type → **int** | **real** | **string**
assign-stat → **id :=** *rel-inst*
rel-inst → **[** *tuple-list* **]**
tuple-list → *tuple tuple-list* | **ε**
tuple → **(** *attr-inst-list* **)**
attr-inst-list → *attr-inst attr-inst-list* | *attr-inst*
attr-inst → *atomic-inst* | *rel-inst*
atomic-inst → **intconst** | **realconst** | **stringconst**

Esercizio 9

Specificare la BNF di un linguaggio imperativo in cui ogni programma, identificato da un nome, contiene zero o più dichiarazioni di variabili ed un corpo, come nel seguente esempio:

```
program esempio
  int a, alfa, beta4;
  float x, y, z;
  string s1, s2;
  int i;
begin
  alfa := 10;
  beta4 := alfa;
  if alfa > 3 then
    x := 12.1;
    y := 1.98;
  else
    z := x;
  endif;
  s1 := "alfa";
  repeat
    a := alfa;
    x := a;
  until x = alfa;
end.
```

Le variabili possono essere di tipo **int**, **float** 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 una o due vie) o un ciclo a condizione finale. Possono essere assegnate variabili con altri variabili o valori. Una condizione è il confronto (**=**, **!=**, **>**, **<**) tra una variabile e un valore o un'altra variabile.

Esercizio 9

Specificare la BNF di un linguaggio imperativo in cui ogni programma, identificato da un nome, contiene zero o più dichiarazioni di variabili ed un corpo, come nel seguente esempio:

```
program esempio
  int a, alfa, beta4;
  float x, y, z;
  string s1, s2;
  int i;
begin
  alfa := 10;
  beta4 := alfa;
  if alfa > 3 then
    x := 12.1;
    y := 1.98;
  else
    z := x;
  endif;
  s1 := "alfa";
  repeat
    a := alfa;
    x := a;
  until x = alfa;
end.
```

program → **program id** *var-decl-list body* **.**
var-decl-list → *var-decl var-decl-list* | **ε**
var-decl → *type id-list* **;**
type → **int** | **float** | **string**
id-list → **id ,** *id-list* | **id**
body → **begin** *stat-list* **end**
stat-list → *stat stat-list* | *stat*
stat → *assign-stat* | *if-stat* | *repeat-stat*
assign-stat → **id :=** *rhs* **;**
rhs → *const* | **id**
const → **intconst** | **floatconst** | **stringconst**
if-stat → **if** *cond* **then** *stat-list else-part* **endif ;**
else-part → **else** *stat-list* | **ε**
cond → **id** *comp-op rhs*
comp-op → = | **!=** | >| <
repeat-stat → **repeat** *stat-list* **until** *cond* **;**

Le variabili possono essere di tipo **int**, **float** 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 una o due vie) o un ciclo a condizione finale. Possono essere assegnate variabili con altri variabili o valori. Una condizione è il confronto (**=**, **!=**, **>**, **<**) tra una variabile e un valore o un'altra variabile.

Esercizio 10

Specificare la grammatica BNF del linguaggio **L** in cui ogni frase è una grammatica (non vuota) BNF, assumendo per **L** i seguenti terminali: **nonterminal** (per identificare un nonterminale), **terminal** (per identificare un terminale), **epsilon** (per identificare il simbolo ε).

Esercizio 10

Specificare la grammatica BNF del linguaggio **L** in cui ogni frase è una grammatica (non vuota) BNF, assumendo per **L** i seguenti terminali: **nonterminal** (per identificare un nonterminale), **terminal** (per identificare un terminale), **epsilon** (per identificare il simbolo **ε**).

grammar → *production-list*
production-list → *production production-list* | *production*
production → *lhs* **->** *rhs*
lhs → **nonterminal**
rhs → *symbol-list* | **epsilon**
symbol-list → *symbol symbol-list* | *symbol*
symbol → **nonterminal | terminal**

Esercizio 11

Specificare la grammatica BNF del linguaggio **L** in cui ogni frase è una grammatica (non vuota) EBNF in forma non fattorizzata (un'unica alternativa per ogni nonterminale), utilizzando per **L** (tra gli altri) i seguenti terminali: **nonterminal** (per identificare un nonterminale), **terminal** (per identificare un terminale), **epsilon** (per identificare il simbolo **ε**). Si assumono per le frasi di **L** unicamente i seguenti operatori estesi:

- { … } = ripetizione zero o più volte;
- […] = opzionalità.

(Si noti che non si considera l'operatore alternativa).

Esercizio 11

Specificare la grammatica BNF del linguaggio **L** in cui ogni frase è una grammatica (non vuota) EBNF in forma non fattorizzata (un'unica alternativa per ogni nonterminale), utilizzando per **L** (tra gli altri) i seguenti terminali: **nonterminal** (per identificare un nonterminale), **terminal** (per identificare un terminale), **epsilon** (per identificare il simbolo **ε**). Si assumono per le frasi di **L** unicamente i seguenti operatori estesi:

- { … } = ripetizione zero o più volte;
- […] = opzionalità.

(Si noti che non si considera l'operatore alternativa).

grammar → *production-list*
production-list → *production production-list* | *production*
production → *lhs* **->** *rhs*
lhs → **nonterminal**
rhs → *expr-list* | **epsilon**
expr-list → *expr expr-list* | *expr*
expr → **nonterminal** | **terminal** | **[** *expr-list* **]** | **{** *expr-list* **}**

Esercizio 12

Specificare la grammatica BNF di un linguaggio per la definizione di moduli di programma, in cui ogni frase contiene una specifica di modulo, come nel seguente esempio:

```
module M is
  var a, b: integer;
      c, d: vector [10] of string;
      r: record (a: integer, b: string);
     x, y, alfa22: vector [5] of record (a: integer, b: vector [20] of string);
  body
    a := 1;
    b := 2;
    c[2] := "alfa";
    r.a := 3;
    x[2].b[3] = "beta";
  end
```

Un modulo ha un identificatore e contiene due sezioni. La prima sezione (introdotta dalla keyword **`var`**) specifica una serie di dichiarazioni di variabili con il loro tipo. I costruttori di tipo (ortogonali tra loro) sono **`vector`** e **`record`**. I tipi semplici sono **`integer`** e **`string`**. Nel caso di vettore, si indica la dimensione, mentre per il record si elencano gli attributi (almeno uno). La seconda sezione (introdotta dalla keyword **`body`**) specifica una lista di assegnamenti, in cui la parte sinistra è una espressione che rappresenta simbolicamente un indirizzo, mentre la parte destra può essere solo una costante semplice.

Esercizio 12

```
module M is
  var a, b: integer;
      c, d: vector [10] of string;
      r: record (a: integer, b: string);
     x, y, alfa22: vector [5] of record (a: integer, b: vector [20] of string);
  body
    a := 1;
    b := 2;
    c[2] := "alfa";
    r.a := 3;
    x[2].b[3] = "beta";
  end
```

program → **module id is** *var-decl* **body** *body-decl* **end**
var-decl → *var-list var-decl* | *var-list*
var-list → *id-list* **:** *type* **;**
id-list → **id ,** *id-list* | **id**
type → **integer** | **string** | *rec-type* | *vec-type*
rec-type → **record (** *attr-list* **)**
attr-list → *attr-decl* **,** *attr-list* | *attr-decl*
attr-decl → **id :** *type*
vec-type → **vector [intconst] of** *type*
body-decl → *assign body-decl* | *assign*
assign → *lhs* **:=** *rhs* **;**
lhs → **id** | *lhs* **[intconst]** | *lhs* **. id**
rhs → **intconst** | **strconst**

Esercizio 13

Specificare la grammatica BNF di un linguaggio per la specifica di definizioni di protocolli di funzioni Haskell-like. Ecco un esempio di frase (ogni frase contiene almeno una definizione):

```
alfa :: Int -> Int
beta :: [Int] -> Bool
gamma :: Char -> (Int, Bool)
zeta :: [[(Char, (Bool, Int))]] -> Int
f10 :: (Int, Bool) -> [Char]
g20 :: Int -> (Int -> Bool -> Char) -> Bool
omega :: (Int -> (Int -> Bool)) -> [[Int]] -> (Bool, Char)
```

I tipi atomici sono `Int`, `Bool` e `Char`. I costruttori di tupla e di lista (ortogonali tra loro) sono indicati rispettivamente dalle parentesi tonde e dalle parentesi quadre. Un parametro di tipo funzione è specificato dal relativo protocollo tra parentesi tonde. Ogni funzione ha almeno un parametro di ingresso. Ogni frase contiene almeno una definizione.

Esercizio 13

Specificare la grammatica BNF di un linguaggio per la specifica di definizioni di protocolli di funzioni Haskell-like. Ecco un esempio di frase (ogni frase contiene almeno una definizione):

```
alfa :: Int -> Int
beta :: [Int] -> Bool
gamma :: Char -> (Int, Bool)
zeta :: [[(Char, (Bool, Int))]] -> Int
f10 :: (Int, Bool) -> [Char]
g20 :: Int -> (Int -> Bool -> Char) -> Bool
omega :: (Int -> (Int -> Bool)) -> [[Int]] -> (Bool, Char)
```

I tipi atomici sono `Int`, `Bool` e `Char`. I costruttori di tupla e di lista (ortogonali tra loro) sono indicati rispettivamente dalle parentesi tonde e dalle parentesi quadre. Un parametro di tipo funzione è specificato dal relativo protocollo tra parentesi tonde. Ogni funzione ha almeno un parametro di ingresso. Ogni frase contiene almeno una definizione.

program → *def-list*
def-list → *def def-list* | *def*
def → **id ::** *type-map*
type-map → *type* **->** *type-map* | *type* **->** *type*
type → *atomic-type* | *tuple-type* | *list-type* | *function-type*
atomic-type → **Int** | **Bool** | **Char**
tuple-type → **(** *type-list* **)**
type-list → *type* **,** *type-list* | *type*
list-type → **[** *type* **]**
function-type → **(** *type-map* **)**

Esercizio 14

Specificare la grammatica BNF di un linguaggio di markup (tipo *HTML*) per la specifica di tabelle. Ogni frase del linguaggio definisce una o più tabelle. Ogni tabella ha almeno una colonna. Inoltre, ogni tabella ha necessariamente una prima riga, che specifica l'intestazione, e una serie (eventualmente vuota) di righe successive che specificano il contenuto. In ogni cella relativa a ciascuna riga è inserito un dato. Un dato può essere un identificatore, un numero o un'altra tabella. Nel caso di intestazione, il dato di ogni cella è necessariamente un identificatore. Tuttavia, un dato può essere omesso (in tal caso, la cella risulta vuota). Le celle relative all'intestazione, invece, non possono essere vuote. Ecco un esempio di tabella e relativa specifica:

Studente	Matricola	Anno	Esami
carlo	46124		
anna	42567	2	corso / voto algebra / 26 geometria / 25

```
<table>
    <tr>
        <th> Studente </th>
        <th> Matricola </th>
        <th> Anno </th>
        <th> Esami </th>
    </tr>
    <tr>
        <td> carlo </td>
        <td >46124 </td>
        <td> </td>
        <td> </td>
    </tr>
    <tr>
        <td> anna </td>
        <td> 42567 </td>
        <td> 2 </td>
        <td>
          <table>
              <tr>
                  <th> corso </th>
                  <th> voto </th>
              </tr>
              <tr>
                  <td> algebra </td>
                  <td> 26 </td>
              </tr>
              <tr>
                  <td> geometria </td>
                  <td> 25 </td>
              </tr>
          </table>
       </td>
    </tr>
</table>
```

Ogni tabella è racchiusa dai tag `<table>` e `</table>`. Ogni riga è racchiusa dai tag `<tr>` e `</tr>` (*table row*). Ogni cella della prima riga è racchiusa dai tag `<th>` e `</th>` (*table heading*), mentre le celle delle righe successive sono racchiuse dai tag `<td>` e `</td>` (*table data*).

Esercizio 14

Specificare la grammatica BNF di un linguaggio di markup (tipo *HTML*) per la specifica di tabelle. Ogni frase del linguaggio definisce una o più tabelle. Ogni tabella ha almeno una colonna. Inoltre, ogni tabella ha necessariamente una prima riga, che specifica l'intestazione, e una serie (eventualmente vuota) di righe successive che specificano il contenuto. In ogni cella relativa a ciascuna riga è inserito un dato. Un dato può essere un identificatore, un numero o un'altra tabella. Nel caso di intestazione, il dato di ogni cella è necessariamente un identificatore. Tuttavia, un dato può essere omesso (in tal caso, la cella risulta vuota). Le celle relative all'intestazione, invece, non possono essere vuote. Ecco un esempio di tabella e relativa specifica:

Studente	Matricola	Anno	Esami
carlo	46124		
anna	42567	2	corso / voto: algebra 26; geometria 25

Ogni tabella è racchiusa dai tag `<table>` e `</table>`. Ogni riga è racchiusa dai tag `<tr>` e `</tr>` (*table row*). Ogni cella della prima riga è racchiusa dai tag `<th>` e `</th>` (*table heading*), mentre le celle delle righe successive sono racchiuse dai tag `<td>` e `</td>` (*table data*).

```
<table>
    <tr>
        <th> Studente </th>
        <th> Matricola </th>
        <th> Anno </th>
        <th> Esami </th>
    </tr>
    <tr>
        <td> carlo </td>
        <td >46124 </td>
        <td> </td>
        <td> </td>
    </tr>
    <tr>
        <td> anna </td>
        <td> 42567 </td>
        <td> 2 </td>
        <td>
          <table>
              <tr>
                  <th> corso </th>
                  <th> voto </th>
              </tr>
              <tr>
                  <td> algebra </td>
                  <td> 26 </td>
              </tr>
              <tr>
                  <td> geometria </td>
                  <td> 25 </td>
              </tr>
          </table>
        </td>
    </tr>
</table>
```

program → *table-list*
table-list → *table table-list* | table
table → **<table>** *head row-list* **</table>**
head → **<tr>** *th-list* **</tr>**
th-list → *th th-list* | *th*
th → **<th> id </th>**
row-list → *row row-list* | **ε**
row → **<tr>** *td-list* **</tr>**
td-list → *td td-list* | *td*
td → **<td>** *data* **</td>**
data → **id** | **num** | *table* | **ε**

Esercizio 15

Specificare la grammatica BNF del sotto-linguaggio di *Haskell* relativo alla specifica di classi di tipi. Ogni frase è composta da una lista non vuota di classi, come nel seguente esempio:

```
class Visible a where
  tostring :: a -> String
  size     :: a -> Int

class Eq a where
  (==), (/=) :: a -> a -> Bool

class Eq b => Ord b where
  (<), (<=), (>), (>=) :: b -> b -> Bool
  max, min             :: b -> b -> b
```

Ogni classe di tipo contiene almeno una funzione. Ogni funzione ha almeno un operando. Per semplicità, oltre alle variabili di tipo, si assumono unicamente i tipi `Int`, `Bool` e `String` (senza costruttori). Gli operatori binari (definiti tra parentesi) sono rappresentati dal simbolo terminale **operator** (che non include le parentesi). Si assume ereditarietà singola (se presente). Le variabili di tipo sono rappresentate dal terminale **var**.

Esercizio 15

Specificare la grammatica BNF del sotto-linguaggio di *Haskell* relativo alla specifica di classi di tipi. Ogni frase è composta da una lista non vuota di classi, come nel seguente esempio:

```
class Visible a where
  tostring :: a -> String
  size     :: a -> Int

class Eq a where
  (==), (/=) :: a -> a -> Bool

class Eq b => Ord b where
  (<), (<=), (>), (>=) :: b -> b -> Bool
  max, min             :: b -> b -> b
```

program → *class-spec-list*
class-spec-list → *class-spec-list class-spec* | *class-spec*
class-spec → **class id var** *inheritance* **where** *protocol-list*
inheritance → **=> id var** | **ε**
protocol-list → *protocol-list protocol* | *protocol*
protocol → *func-name-list* **::** *mapping*
func-name-list → *func-name-list* **,** *func-name* | *func-name*
func-name → **(operator)** | **id**
mapping → *mapping* **->** *type* | *type* **->** *type*
type → **Int** | **Bool** | **String** | **var**

Esercizio 16

Specificare la grammatica BNF di un linguaggio in cui ogni frase è composta da una lista (anche vuota) di classi, come nel seguente esempio:

```
class alfa is
  attributes a: int;
             b: string;
             c: bool;
             d: record(x: bool, y: record(z1: string, num: int));
             t: table(n: int, descrizione: string);
  methods lookup(p1: int, p2: table(a: string, b: int)): int;
          remdup(a1: string, b2: int): table(a: int, b: string);
end alfa;

...

class omega inherits alfa, beta, gamma23 is
  methods ...

end omega;

...
```

Le sezioni degli attributi e dei metodi sono opzionali (ad esempio, la calsse `omega` non contiene la sezione degli attributi). Ogni metodo ha almeno un parametro formale e restituisce un valore. I possibili tipi sono **`int`**, **`string`**, **`bool`**, **`record`** e **`table`**. A differenza di **`record`** (i cui campi possono essere di qualsiasi tipo), il tipo **`table`** può contenere solo campi di tipo **`int`**, **`string`**, o **`bool`**. Una classe può opzionalmente ereditare da altre classi.

Esercizio 16

Specificare la grammatica BNF di un linguaggio in cui ogni frase è composta da una lista (anche vuota) di classi, come nel seguente esempio:

```
class alfa is
  attributes a: int;
             b: string;
             c: bool;
             d: record(x: bool, y: record(z1: string, num: int));
             t: table(n: int, descrizione: string);
  methods lookup(p1: int, p2: table(a: string, b: int)): int;
          remdup(a1: string, b2: int): table(a: int, b: string);
end alfa;

...

class omega inherits alfa, beta, gamma23 is
  methods ...

end omega;

...
```

program → *class-list*
class-list → *class-def* **;** *class-list* | **ε**
class-def → **class id** *inheritance* **is** *attr-sect meth-sect* **end id**
inheritance → **inherits** *id-list* | **ε**
id-list → **id ,** *id-list* | **id**
attr-sect → **attributes** *attr-list* | **ε**
attr-list → *attr-def* **;** *attr-list* | *attr-def* **;**
attr-def → **id :** *type*
type → *simple-type* | *record-type* | *table-type*
simple-type → **int** | **string** | **bool**
record-type → **record (** *field-list* **)**
field-list → *attr-def* **,** *field-list* | *attr-def*
table-type → **table (** *simple-list* **)**
simple-list → *simple-attr* **,** *simple-list* | *simple-attr*
simple-attr → **id :** *simple-type*
meth-sect → **methods** *meth-list* | **ε**
meth-list → *meth-def meth-list* | *meth-def*
meth-def → **id (** *field-list* **) :** *type* **;**

Esercizio 17

Specificare la grammatica BNF di un linguaggio SQL-like per l'interrogazione di database relazionali, come nel seguente esempio di programma:

```
use anagrafe;

SELECT *
FROM Persone
WHERE citta = 'firenze';

SELECT nome, cognome, professione
FROM Persone;

use statistiche;

SELECT a+b, c*(d+e-4), f/g
FROM Parametri, Distribuzioni
WHERE a = b+1 AND b > 10 OR (a > 0 AND f < 100+c-d);

SELECT a, b, c
FROM (SELECT c, d FROM Sondaggi WHERE d > 0) AS Sond, Medie
WHERE c > 0 AND d = h;
```

L'istruzione use apre un database al quale fanno riferimento le interrogazioni successive. L'apertura di un nuovo database rimuove l'accesso al database corrente ed abilita l'accesso a quello nuovo. Ogni interrogazione è composta dal pattern **SELECT-FROM-WHERE**, in cui la clausola **WHERE** è opzionale. La clausola **SELECT** specifica una lista di espressioni aritmetiche su nomi di attributi e costanti intere, oppure il metacarattere '*****' per indicare tutti gli attributi in gioco. La clausola **FROM** specifica una lista di tabelle, in cui ogni tabella è identificata da un nome eventualmente preceduto da una interrogazione tra parentesi e dalla keyword **AS**. La clausola **WHERE** specifica una espressione booleana che coinvolge gli operatori logici **AND** e **OR** applicati a operazioni di confronto (mediante gli operatori **=**, **<**, **>**) tra espressioni aritmetiche.

Esercizio 17

```
use anagrafe;

SELECT *
FROM Persone
WHERE citta = 'firenze';

SELECT nome, cognome, professione
FROM Persone;

use statistiche;

SELECT a+b, c*(d+e-4), f/g
FROM Parametri, Distribuzioni
WHERE a = b+1 AND b > 10 OR (a > 0 AND f < 100+c-d);

SELECT a, b, c
FROM (SELECT c, d FROM Sondaggi WHERE d > 0) AS Sond, Medie
WHERE c > 0 AND d = h;
```

program → *statements*
statements → *statement* **;** *statements* | *statement* **;**
statement → *use-stat* | *query*
use-stat → **use id**
query → **SELECT** *target* **FROM** *table-list where-clause*
target → *expr-list* | ***
expr-list → *expr* **,** *expr-list* | *expr*
expr → *expr math-op expr* | **(** *expr* **)** | **id** | **num**
math-op → **+** | **–** | ***** | **/**
table-list → *table* **,** *table-list* | *table*
table → **id** | **(** *query* **) AS id**
where-clause → **WHERE** *predicate* | **ε**
predicate → *predicate logic-op predicate* | *expr comp-op expr* | **(** *predicate* **)**
logic-op → **AND** | **OR**
comp-op → **=** | **<** | **>**

Esercizio 18

Specificare la grammatica BNF di un linguaggio *PHP*-like, per la specifica di prototipi di funzioni, come nella seguente frase:

```
function alfa($a, $b, &$c, $min=1, $max=20);
function beta(&$sole, $luna);
function gamma();
function delta($nome='stella', $cognome='cometa', $anni=21);
```

Ogni frase specifica una lista (non vuota) di prototipi di funzioni. La lista dei parametri formali (eventualmente vuota) elenca una serie di nomi di variabili (preceduti dal simbolo speciale $). Se un parametro è passato per referenza (invece che per valore), viene prefisso dal simbolo &. I parametri in coda alla lista possono avere un valore di default (intero o stringa). I parametri con valori di default non possono essere passati per referenza.

Esercizio 18

Specificare la grammatica BNF di un linguaggio *PHP*-like, per la specifica di prototipi di funzioni, come nella seguente frase:

```
function alfa($a, $b, &$c, $min=1, $max=20);
function beta(&$sole, $luna);
function gamma();
function delta($nome='stella', $cognome='cometa', $anni=21);
```

Ogni frase specifica una lista (non vuota) di prototipi di funzioni. La lista dei parametri formali (eventualmente vuota) elenca una serie di nomi di variabili (preceduti dal simbolo speciale $). Se un parametro è passato per referenza (invece che per valore), viene prefisso dal simbolo &. I parametri in coda alla lista possono avere un valore di default (intero o stringa). I parametri con valori di default non possono essere passati per referenza.

program → *funct-list*
funct-list → *funct funct-list* | *funct*
funct → **function id (** *param-list* **) ;**
param-list → *prefix-list* | *prefix-list* **,** *postfix-list* | *postfix-list* | **ε**
prefix-list → *prefix-param* **,** *prefix-list* | *prefix-param*
prefix-param → *optional-ref var*
optional-ref → **&** | **ε**
var → **$id**
postfix-list → *postfix-param* **,** *postfix-list* | *postfix-param*
postfix-param → *var* = *const*
const → **intconst** | **strconst**

Esercizio 19

Specificare la grammatica BNF di un linguaggio per la dichiarazione di variabili, come nel seguente esempio:

```
a, x1, gamma: integer;
s: string;
b1, b2: boolean;
v: vector(3,5,10) of string;
s: structure(a: integer, b: vector(2,5) of string);
f, g: function() -> vector(10) of integer;
omega: function(integer, structure(codice: string, prezzo: integer)) -> boolean;
```

Ogni frase specifica una lista (non vuota) di dichiarazioni. Partendo dai tipi elementari **integer**, **string** e **boolean**, si possono specificare espressioni di tipo mediante i costruttori **vector**, **structure** e **function**. Un vettore viene qualificato da una lista (non vuota) di dimensioni. Una struttura è definita da una lista (non vuota) di attributi. Una funzione è definita dal suo protocollo (parametri anonimi). Il linguaggio non è ortogonale poichè il tipo di un vettore è sempre atomico e una funzione non è una forma funzionale (non può ricevere o restituire funzioni).

Esercizio 19

Specificare la grammatica BNF di un linguaggio per la dichiarazione di variabili, come nel seguente esempio:

```
a, x1, gamma: integer;
s: string;
b1, b2: boolean;
v: vector(3,5,10) of string;
s: structure(a: integer, b: vector(2,5) of string);
f, g: function() -> vector(10) of integer;
omega: function(integer, structure(codice: string, prezzo: integer)) -> boolean;
```

Ogni frase specifica una lista (non vuota) di dichiarazioni. Partendo dai tipi elementari **integer**, **string** e **boolean**, si possono specificare espressioni di tipo mediante i costruttori **vector**, **structure** e **function**. Un vettore viene qualificato da una lista (non vuota) di dimensioni. Una struttura è definita da una lista (non vuota) di attributi. Una funzione è definita dal suo protocollo (parametri anonimi). Il linguaggio non è ortogonale poichè il tipo di un vettore è sempre atomico e una funzione non è una forma funzionale (non può ricevere o restituire funzioni).

program → *def-list*
def-list → *def def-list* | *def*
def → *id-list* **:** *type* **;**
id-list → **id ,** *id-list* | **id**
type → *non-func-type* | *func-type*
non-func-type → *simple-type* | *vect-type* | *struct-type*
simple-type → **integer** | **string** | **boolean**
vect-type → **vector (** *num-list* **) of** *simple-type*
num-list → **num ,** *num-list* | **num**
struct-type → **structure (** *attr-list* **)**
attr-list → *attr attr-list* | *attr*
attr → **id :** *type*
func-type → **function (** *optional-param-list* **) ->** *non-func-type*
optional-param-list → *param-list* | **ε**
param-list → *non-func-type* **,** *param-list* | *non-func-type*

Esercizio 20

Specificare la BNF di un linguaggio per la dichiarazione e istanziazione di variabili, come nel seguente esempio:

```
i, j, k: integer;
j = 10;
i = j;
x, y: real;
y = 3.14;
s: string;
r1, r2: record a: integer, b: string end;
r1 = rec(3, "alfa");
r2 = r1;
v1, v2: vector [3] of record nome: string, cognome: string end;
v2 = vec(rec("luigi", "rossi"), rec("anna", "verdi"), rec("mino", "viola"));
```

Si può solo dichiarare e assegnare variabili con costanti o altre variabili. I costruttori di tipo sono ortogonali. Una costante strutturata (record o vettore) viene indicata dalla relativa keyword (rispettivamente rec, vec) e dalla lista dei suoi elementi.

Esercizio 20

```
i, j, k: integer;
j = 10;
i = j;
x, y: real;
y = 3.14;
s: string;
r1, r2: record a: integer, b: string end;
r1 = rec(3, "alfa");
r2 = r1;
v1, v2: vector [3] of record nome: string, cognome: string end;
v2 = vec(rec("luigi", "rossi"), rec("anna", "verdi"), rec("mino", "viola"));
```

Si può solo dichiarare e assegnare variabili con costanti o altre variabili. I costruttori di tipo sono ortogonali. Una costante strutturata (record o vettore) viene indicata dalla relativa keyword (rispettivamente rec, vec) e dalla lista dei suoi elementi.

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 → **integer** | **string** | **real** | *record-type* | *vector-type*
record-type → **record** *attr-list* **end**
simple-type → **integer** | **string** | **boolean**
attr-list → *attr* **,** *attr-list* | *attr*
attr → **id :** *type*
vector-type → **vector [intconst] of** *type*
assign-stat → **id =** *value*
value → **id** | *const*
const → **intconst** | **strconst** | **realconst** | *complex-const*
complex-const → *constructor* **(** *const-list* **)**
constructor → **rec** | **vec**
const-list → *const* **,** *const-list* | *const*

Esercizio 21

Specificare la grammatica BNF del linguaggio **R** per descrivere espressioni regolari in formato testuale, nel quale ogni frase è una definizione regolare, utilizzando per **R** (tra gli altri) i seguenti simboli: **id** (identificatore), **char** (carattere dell'alfabeto), **epsilon** (simbolo ε). Oltre alla concatenazione, si assumono per **R** i seguenti operatori:

*****	(ripetizione zero o più volte);
+	(ripetizione una o più volte);
|	(alternativa);
[... **]**	(un range di caratteri).

Un range di caratteri può essere espresso mediante enumerazione (ad esempio, **[abc])** oppure mediante gli estremi del range, eventualmente multipli (ad esempio, **[a-zA-Z]**).
Infine, è possibile usare le parentesi tonde per raggruppare sottoespressioni regolari.

Esercizio 21

Specificare la grammatica BNF del linguaggio **R** per descrivere espressioni regolari in formato testuale, nel quale ogni frase è una definizione regolare, utilizzando per **R** (tra gli altri) i seguenti simboli: **id** (identificatore), **char** (carattere dell'alfabeto), **epsilon** (simbolo ε). Oltre alla concatenazione, si assumono per **R** i seguenti operatori:

*****	(ripetizione zero o più volte);
+	(ripetizione una o più volte);
|	(alternativa);
[...]	(un range di caratteri).

Un range di caratteri può essere espresso mediante enumerazione (ad esempio, **[abc])** oppure mediante gli estremi del range, eventualmente multipli (ad esempio, **[a-zA-Z]**).
Infine, è possibile usare le parentesi tonde per raggruppare sottoespressioni regolari.

regdef → *def-list*
def-list → *def def-list* | *def*
def → **id** '**->**' *expr*
expr → **epsilon** | **char** | **id** | **[** *range* **]** | *expr expr* | *expr* ***** | *expr* + | *expr* '|' *expr* | **(** *expr* **)**
range → *char-list* | *segment-list*
char-list → **char** *char-list* | **char**
segment-list → *segment segment-list* | *segment*
segment → **char - char**

Esercizio 22

Specificare la grammatica BNF di un linguaggio in cui ogni frase è una lista (anche vuota) di dichiarazioni di variabili, come nel seguente esempio:

```
n, m: int;
a, b, c: record(x1, x2: real, y, z, w: string, i: int);
m: vector [10,30,20] of real;
lista1, lista2: sequence of string;
```

Oltre ai tipi semplici **int**, **real** e **string**, le espressioni di tipo coinvolgono i costruttori **record** (struttura), **vector** (vettore multidimensionale) e **sequence** (sequenza). Le dimensioni di un vettore sono rappresentate da costanti intere. I costruttori di tipo sono ortogonali tra loro ad eccezione del fatto che gli elementi di un vettore non possono essere né record, né sequenze, né vettori.

Esercizio 22

Specificare la grammatica BNF di un linguaggio in cui ogni frase è una lista (anche vuota) di dichiarazioni di variabili, come nel seguente esempio:

```
n, m: int;
a, b, c: record(x1, x2: real, y, z, w: string, i: int);
m: vector [10,30,20] of real;
lista1, lista2: sequence of string;
```

program → *def-list* | **ε**
def-list → *def* **;** *def-list* | *def*
def → *id-list* **:** *type*
id-list → **id ,** *id-list* | **id**
type → *simple-type* | *complex-type*
simple-type → **int** | **real** | **string**
complex-type → *record-type* | *vector-type* | *sequence-type*
record-type → **record (** *attr-list* **)**
attr-list → *def* **,** *attr-list-list* | *def*
vector-type → **vector [** *num-list* **] of** *atomic-type*
num-list → **num ,** *num-list* | **num**
sequence-type → **sequence of** *type*

Esercizio 23

Specificare la grammatica BNF di un generatore *Haskell*-like di liste, come nel seguente esempio:

```
[ a+(b*c) | (a,b) <- alfa, (c,d,e) <- beta, a > b-2, b == d*(c/e) ]
```

La parte sinistra (target) del generatore è una espressione aritmetica. La parte destra del generatore è suddivisa in due sezioni. La prima sezione è una lista (non vuota) di condizioni di appartenenza espresse mediante il pattern tupla. La seconda sezione è una lista (non vuota) di semplici confronti fra espressioni aritmetiche (mediante gli operatori di confronto ==, !=, > , >=, <, <=).

Esercizio 23

Specificare la grammatica BNF di un generatore *Haskell*-like di liste, come nel seguente esempio:

```
[ a+(b*c) | (a,b) <- alfa, (c,d,e) <- beta, a > b-2, b == d*(c/e) ]
```

La parte sinistra (target) del generatore è una espressione aritmetica. La parte destra del generatore è suddivisa in due sezioni. La prima sezione è una lista (non vuota) di condizioni di appartenenza espresse mediante il pattern tupla. La seconda sezione è una lista (non vuota) di semplici confronti fra espressioni aritmetiche (mediante gli operatori di confronto ==, !=, > , >=, <, <=).

generator → **[** *expr* '|' *membership-list* **,** *comparison-list* **]**
expr → **id** | **intconst** | **(** *expr* **)** | *expr* + *expr* | *expr* - *expr* | *expr* * *expr* | *expr* / *expr*
membership-list → *membership* **,** *membership-list* | *membership*
membership → **(** *id-list* **)** **<-** **id**
id-list → **id** **,** *id-list* | **id**
comparison-list → *comparison* **,** *comparison-list* | *comparison*
comparison → *expr* *relop* *expr*
relop → == | **!=** | > | >= | < | <=

Esercizio 24

Specificare la grammatica BNF di un linguaggio per la manipolazione di tabelle, in cui ogni frase è una lista (anche vuota) di istruzioni, come nel seguente esempio:

```
def R (a: int, b: string)
R = [(1, "alfa"), (2, "beta")]
def T (x: string, y: string, z: int)
def V (elem: int)
V = []
T = [("sole", "luna", 25)]
select [elem >= 0 ] V
select [a > 1 and (b == "beta" or b != "gamma")] select [a != 2 ] R
```

Esistono tre tipi di istruzioni: definizione di tabella, istanziazione di tabella e interrogazione. Ogni definizione coinvolge almeno un attributo (di tipo **int** o **string**). Nella istanziazione, la parte destra (istanza della tabella) è una lista (anche vuota) di tuple. Una interrogazione può semplicemente essere una tabella o, più in generale, la selezione di una tabella (o di un'altra selezione, senza limiti di innestamento). Il predicato di selezione (racchiuso tra parentesi quadre) può essere una comparazione (che coinvolge gli operatori ==, !=, <, <=, >, >=) o, più in generale, una espressione logica che coinvolge gli operatori (ortogonali tra loro) **and** ed **or**. Si possono comparare due attributi o un attributo ed un valore, ma non due valori.

Esercizio 24

```
def R (a: int, b: string)
R = [(1, "alfa"), (2, "beta")]
def T (x: string, y: string, z: int)
def V (elem: int)
V = []
T = [("sole", "luna", 25)]
select [elem >= 0 ] V
select [a > 1 and (b == "beta" or b != "gam
```

program → *stat-list*
stat-list → *stat stat-list* | **ε**
stat → *definition* | *instantiation* | *query*
definition → **def id (** *attr-list* **)**
attr-list → *attr* **,** *attr-list* | *attr*
attr→ **id :** *type*
type → **int** | **string**
instantiation → **id = [** *opt-tuple-list* **]**
opt-tuple-list → *tuple-list* | **ε**
tuple-list → *tuple* **,** *tuple-list* | *tuple*
tuple → **(** *const-list* **)**
const-list → *const* , *const-list* | *const*
const → **intconst** | **strconst**
query → **select** [*pred*] *query* | **id**
pred → *comp* | **(** *pred* **)** | *pred* **and** *pred* | *pred* **or** *pred*
comp → **id** *relop const* | *const relop* **id** | **id** *relop* **id**
relop → **==** | **!=** | **>** | **>=** | **<** | **<=**

Esercizio 25

Specificare la grammatica BNF di un linguaggio per definire ed assegnare tabelle, come nel seguente esempio:

```
table (int a, string b, table(int d, real e) c) tab1;

table (int lung, table (table (int delta) t) r) tab2;

tab1 = [(3, "sole", [(10, 23.12)(12, 1.44)])
        (4, "mare", [(16, 3.56)])
        (5, "stella", [])];

tab2 = [(1, [([(12)(33)(37)])])
        (3, [([(256)(1)])])
        (24, [([])])
        (46, [])];

table (int x, int y) Zeta;

Zeta = [];
```

Ogni frase contiene almeno una istruzione. Le istruzioni di definizione ed assegnamento delle tabelle possono essere specificate in qualsiasi ordine. Gli attributi atomici sono **int**, **real** e **string**. Non esiste limite di innestamento delle tabelle. Una tabella (o un attributo di tipo tabella) può essere assegnato con la tabella vuota **[ ]**.

Esercizio 25

Specificare la grammatica BNF di un linguaggio per definire ed assegnare tabelle, come nel seguente esempio:

```
table (int a, string b, table(int d, real e) c) tab1;

table (int lung, table (table (int delta) t) r) tab2;

tab1 = [(3, "sole", [(10, 23.12)(12, 1.44)])
        (4, "mare", [(16, 3.56)])
        (5, "stella", [])];

tab2 = [(1, [([(12)(33)(37)])])
        (3, [([(256)(1)])])
        (24, [([])])
        (46, [])];

table (int x, int y) Zeta;

Zeta = [];
```

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat*
def-stat → *table-type* **id**
table-type → **table (** *attr-list* **)**
attr-list → *attr-def* **,** *attr-list* | *attr-def*
attr-def → *type* **id**
type → *atomic-type* | *table-type*
atomic-type → **int** | **real** | **string**
assign-stat → **id =** *table-inst*
table-inst → **[** *tuple-list* **]**
tuple-list → *tuple tuple-list* | **ε**
tuple → **(** *attr-inst-list* **)**
attr-inst-list → *attr-inst attr-inst-list* | *attr-inst*
attr-inst → *atomic-inst* | *table-inst*
atomic-inst → **intconst** | **realconst** | **stringconst**

Esercizio 26

Specificare la grammatica BNF di un linguaggio per la dichiarazione di variabili, come nel seguente esempio:

```
num, i, j, k: integer;
name, surname: string;
flag: boolean;
a: array [1..100] of integer;
m: array [10..20] of array ['a'..'z'] of boolean;
epsilon: set of integer;
omega: set of set of string;
```

Ogni frase contiene almeno una dichiarazione. I tipi atomici sono **integer**, **string** e **boolean**. I costruttori di tipo sono **array** e **set**. L'indice di un array può essere un intero o un carattere, il cui range è specificato nella dichiarazione. I costruttori di tipo sono ortogonali solo a se stessi.

Esercizio 26

Specificare la grammatica BNF di un linguaggio per la dichiarazione di variabili, come nel seguente esempio:

```
num, i, j, k: integer;
name, surname: string;
flag: boolean;
a: array [1..100] of integer;
m: array [10..20] of array ['a'..'z'] of boolean;
epsilon: set of integer;
omega: set of set of string;
```

program → *decl-list*
decl-list → *decl decl-list* | *decl*
decl → *id-list* **:** *type* **;**
id-list → **id ,** *id-list* | **id**
type → *atomic-type* | *array-type* | *set-type*
atomic-type → **integer** | **string** | **boolean**
array-type → **array [** *range* **] of** *array-element*
range → **intconst .. intconst** | **charconst .. charconst**
array-element → *atomic-type* | *array-type*
set-type→ **set of** *set-element*
set-element → *atomic-type* | *set-type*

Esercizio 27

Specificare la grammatica BNF di un linguaggio per la manipolazione di variabili, come nel seguente esempio:

```
program
  int a, b, c;
  string x, y;
begin
  a = 12;
  x = "alfa";
  if a == 10 then
    y = "beta";
  elsif "beta" == y then
    x = "gamma";
    y = "omega";
  else
    b = a;
  endif;
  while a == b do
    a = 10;
  endwhile;
end.
```

La sezione di dichiarazione delle variabili (che è opzionale) precede il corpo del programma. Quest'ultimo è composto da una lista non vuota di istruzioni (assegnamenti, selezioni a più vie, cicli). L'espressione di assegnamento è una costante o una variabile. Le condizioni sono espresse dal confronto di uguaglianza tra una variabile ed una costante o tra due variabili. Nella selezione a più vie, i rami elsif (in numero illimitato) ed else sono opzionali.

Esercizio 27

Specificare la grammatica BNF di un linguaggio per la manipolazione di variabili, come nel seguente esempio:

```
program
  int a, b, c;
  string x, y;
begin
  a = 12;
  x = "alfa";
  if a == 10 then
    y = "beta";
  elsif "beta" == y then
    x = "gamma";
    y = "omega";
  else
    b = a;
  endif;
  while a == b do
    a = 10;
  endwhile;
end.
```

program → **program** *decl-section* **begin** *stat-list* **end .**
decl-section → *decl-list* | **ε**
decl-list → *decl decl-list* | *decl*
decl→ *type id-list* **;**
type → **integer** | **string**
id-list → **id ,** *id-list* | **id**
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat→ *assign-stat* | *if-stat* | *while-stat*
assign-stat → **id** = *const* | **id** = **id**
const → **intconst** | **strconst**
if-stat→ **if** *cond* **then** *stat-list elsif-part else-part* **endif**
cond → **id** == *const* | **id** == **id**
elsif-part→ **elsif** *cond* **then** *stat-list elsif-part* | **ε**
else-part→ **else** *cond* **then** *stat-list* | **ε**
while-stat→ **while** *cond* **do** *stat-list* **endwhile**

La sezione di dichiarazione delle variabili (che è opzionale) precede il corpo del programma. Quest'ultimo è composto da una lista non vuota di istruzioni (assegnamenti, selezioni a più vie, cicli). L'espressione di assegnamento è una costante o una variabile. Le condizioni sono espresse dal confronto di uguaglianza tra una variabile ed una costante o tra due variabili. Nella selezione a più vie, i rami elsif (in numero illimitato) ed else sono opzionali.

Esercizio 28

Specificare la grammatica BNF relativa ad un linguaggio in cui ogni frase è una lista (anche vuota) di numeri complessi, come nel seguente esempio:

```
[ (1, 24.66),(0.12, +3),(-1.845, -1.23E20 ) , (33E4, 26.80E-3)    ]
```

sulla base dei seguenti vincoli:

- Un numero complesso è rappresentato da una coppia (parte reale, parte immaginaria);
- Ognuna delle due parti è rappresentata da un numero, eventualmente con segno, avente una parte intera, opzionalmente una parte decimale e, infine, opzionalmente una parte esponenziale (rappresentata da E seguita da un intero, eventualmente con segno);
- La parte intera non contiene zeri non significativi.

Si assume che gli elementi lessicali siano: **0 1 2 3 4 5 6 7 8 9 [ ] ( ) , + - E .**

Esercizio 28

Specificare la grammatica BNF relativa ad un linguaggio in cui ogni frase è una lista (anche vuota) di numeri complessi, come nel seguente esempio:

```
[ (1, 24.66),(0.12, +3),(-1.845, -1.23E20 ) , (33E4, 26.80E-3)    ]
```

program → **[** *opt-complex-list* **]**
opt-complex-list → *complex-list* | **ε**
complex-list → *complex* **,** *complex-list* | *complex*
complex → (*number* , *number*)
number → *signed-integer* *opt-decimal* *opt-exponential*
signed-integer → *opt-sign integer*
opt-sign → **+** | **–** | **ε**
integer → **0** | *nonzero opt-digits*
nonzero → **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
opt-digits → *digits* | **ε**
digits → *digit digits* | *digit*
digit → **0** | *nonzero*
opt-decimal → **.** *digits* | **ε**
opt-exponential → **E** *signed-integer* | **ε**

Esercizio 29

Specificare la grammatica BNF di un linguaggio per la dichiarazione di variabili, come nel seguente esempio:

```
int alfa, beta, i, j23;
string nome, cognome, paese;
bool b1, ok;
int vector [100] v1, v2;
string vector [100] vector [50] matrice, m2;
int list lista;
int list list list spazio, complex_list;
```

Una frase può essere vuota. I tipi atomici sono `int`, `string` e `bool`. I costruttori di tipo sono `vector` e `list`. La dimensione di un vettore è definita da una costante intera. I costruttori di tipo sono ortogonali solo a se stessi.

Esercizio 29

Specificare la grammatica BNF di un linguaggio per la dichiarazione di variabili, come nel seguente esempio:

```
int alfa, beta, i, j23;
string nome, cognome, paese;
bool b1, ok;
int vector [100] v1, v2;
string vector [100] vector [50] matrice, m2;
int list lista;
int list list list spazio, complex_list;
```

Una frase può essere vuota. I tipi atomici sono `int`, `string` e `bool`. I costruttori di tipo sono `vector` e `list`. La dimensione di un vettore è definita da una costante intera. I costruttori di tipo sono ortogonali solo a se stessi.

program → *decl-list* | **ε**
decl-list→ *decl decl-list* | *decl*
decl → *type id-list* **;**
type → *simple-type constructor-list*
id-list → **id ,** *id-list* | **id**
simple-type → **int** | **string** | **bool**
constructor-list → *vector-list* | *list-list* | **ε**
vector-list → *vector-decl vector-list* | *vector-decl*
vector-decl → **vector [intconst]**
list-list → **list** *list-list* | **list**

Esercizio 30

Specificare la grammatica BNF di un linguaggio di espressioni relazionali su tabelle complesse (non in prima forma normale), come nel seguente esempio:

```
select [ A == select [ B != C and C > select [ D == E ] F ] H ] R;

S;

select [ A < B or select [ C == D ] E >= F ]
  select [ G <= select [ H != L ] select [ K > N ] W ] T;
```

Ogni frase si compone di almeno una espressione su tabella, il cui effetto è la visualizzazione del risultato. È possibile visualizzare una intera tabella o una selezione (eventualmente multipla) di una tabella. Il predicato di selezione (racchiuso tra parentesi quadre) si compone di una serie di confronti fra espressioni, collegati dagli operatori logici **and** ed **or** (senza uso di parentesi).

Esercizio 30

Specificare la grammatica BNF di un linguaggio di espressioni relazionali su tabelle complesse (non in prima forma normale), come nel seguente esempio:

```
select [ A == select [ B != C and C > select [ D == E ] F ] H ] R;

S;

select [ A < B or select [ C == D ] E >= F ]
  select [ G <= select [ H != L ] select [ K > N ] W ] T;
```

Ogni frase si compone di almeno una espressione su tabella, il cui effetto è la visualizzazione del risultato. È possibile visualizzare una intera tabella o una selezione (eventualmente multipla) di una tabella. Il predicato di selezione (racchiuso tra parentesi quadre) si compone di una serie di confronti fra espressioni, collegati dagli operatori logici **and** ed **or** (senza uso di parentesi).

program → *expr-list*
expr-list → *expr* **;** *expr-list* | *expr* **;**
expr → **select [** *predicate* **]** *expr* | **id**
predicate → *predicate logical-op comparison* | *comparison*
comparison → *expr comp-op expr*
logical-op → **and** | **or**
comp-op → == | **!=** | > | < | >= | <=

Esercizio 31

Specificare la grammatica BNF di un linguaggio per la dichiarazione di variabili, come nel seguente esempio:

```
lung, z1, omega: int;
descrizione: string;
b1, b2, isnew: bool;
m: matrix(15,20,8,120) of int;
s: struct (a: int, b: matrix(10,3) of string);
alfa: func() : matrix(100) of int;
beta: func(int, struct (prodotto: string, peso: int)) : int;
```

Ogni frase specifica una lista (anche vuota) di dichiarazioni. Partendo dai tipi elementari `int`, `string` e `bool`, si possono specificare espressioni di tipo mediante i costruttori `matrix`, `struct` e `func`. Una matrice viene qualificata da una lista (non vuota) di dimensioni. Una struttura è definita da una lista (non vuota) di attributi. Una funzione è definita dalla lista dei tipi dei parametri. Il linguaggio non è ortogonale poichè il tipo di una matrice è sempre atomico e una funzione non può ricevere o restituire funzioni.

Esercizio 31

Specificare la grammatica BNF di un linguaggio per la dichiarazione di variabili, come nel seguente esempio:

```
lung, z1, omega: int;
descrizione: string;
b1, b2, isnew: bool;
m: matrix(15,20,8,120) of int;
s: struct (a: int, b: matrix(10,3) of string);
alfa: func() : matrix(100) of int;
beta: func(int, struct (prodotto: string, peso: int)) : int;
```

Ogni frase specifica una lista (anche vuota) di dichiarazioni. Partendo dai tipi elementari `int`, `string` e `bool`, si possono specificare espressioni di tipo mediante i costruttori `matrix`, `struct` e `func`. Una matrice viene qualificata da una lista (non vuota) di dimensioni. Una struttura è definita da una lista (non vuota) di attributi. Una funzione è definita dalla lista dei tipi dei parametri. Il linguaggio non è ortogonale poichè il tipo di una matrice è sempre atomico e una funzione non può ricevere o restituire funzioni.

program → *def-list* | **ε**
def-list → *def def-list* | *def*
def → *id-list* **:** *type* **;**
id-list → **id ,** *id-list* | **id**
type → *non-func-type* | *func-type*
non-func-type → *simple-type* | *matrix-type* | *struct-type*
simple-type → **int** | **string** | **bool**
matrix-type → **matrix (** *num-list* **) of** *simple-type*

num-list → **num ,** *num-list* | **num**
struct-type → **struct (** *attr-list* **)**
attr-list → *attr attr-list* | *attr*
attr → **id :** *type*
func-type → **func (** *optional-param-list* **) :** *non-func-type*
optional-param-list → *param-list* | **ε**
param-list → *non-func-type* **,** *param-list* | *non-func-type*

Esercizio 32

Specificare la grammatica BNF di un linguaggio in cui ogni frase è una sequenza (anche vuota) di tuple di costanti numeriche, come nel seguente esempio:

```
[(1, 12.35),(-2.034, +10, 20.0001),(+0.138),(0,1,2,3,-44.999103,124,-2)]
```

sulla base sei seguenti vincoli:

- Ogni tupla include almeno una costante numerica.
- Una costante numerica (con segno opzionale) è composta da una parte intera ed opzionalmente da una parte decimale.
- La parte intera e la parte decimale (composta almeno da una cifra) non contengono zeri non significativi.
- Si assumono (unicamente) i seguenti elementi lessicali: **`0 1 2 3 4 5 6 7 8 9 0 [ ] ( ) , + - .`**

Esercizio 32

Specificare la grammatica BNF di un linguaggio in cui ogni frase è una sequenza (anche vuota) di tuple di costanti numeriche, come nel seguente esempio:

```
[(1, 12.35),(-2.034, +10, 20.0001),(+0.138),(0,1,2,3,-44.999103,124,-2)]
```

sequence → **[** *opt-tuple-list* **]**
opt-tuple-list → *tuple-list* | **ε**
tuple-list → *tuple tuple-list* | *tuple*
tuple → **(** *const-list* **)**
const-list → *const* **,** *const-list* | *const*
const → *signed-integer opt-decimal*
signed-integer → *opt-sign integer*
opt-sign → **+** | **-** | **ε**
integer → **0** | *nonzero opt-digits*
nonzero → **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
opt-digits → *digit opt-digit* | **ε**
digit → **0** | *nonzero*
opt-decimal → **.** *opt-digits nonzero* | **ε**

Esercizio 33

Specificare la grammatica BNF di un linguaggio in cui ogni frase è un programma specificato dal un linguaggio funzionale per la manipolazione di interi, come nel seguente esempio:

```
a=3, b=5, c=12;

function alfa(n, m) = a + b - 3,
function beta(x, y, z) = alfa(x + y - (a + 3), 5) - z + 1,
function gamma(w) = beta(w, a - 1, c * w) - (a + b) * w;

(a + b) * alfa(a, 3) - gamma(beta(a,b,c)).
```

Il programma è composto da tre sezioni, di cui solo la terza è obbligatoria. La prima sezione specifica un insieme di costanti. La seconda sezione specifica una serie di funzioni definite da una lista (anche vuota) di parametri formali ed una espressione (corpo della funzione). La terza sezione specifica l'espressione del programma. Una espressione coinvolge le quattro operazioni aritmetiche (con possibilità di parentesi) e chiamate di funzione.

Esercizio 33

Specificare la grammatica BNF di un linguaggio in cui ogni frase è un programma specificato dal un linguaggio funzionale per la manipolazione di interi, come nel seguente esempio:

```
a=3, b=5, c=12;

function alfa(n, m) = a + b - 3,
function beta(x, y, z) = alfa(x + y - (a + 3), 5) - z + 1,
function gamma(w) = beta(w, a - 1, c * w) - (a + b) * w;

(a + b) * alfa(a, 3) - gamma(beta(a,b,c)).
```

program → *opt-const-sect* *opt-func-sect* *expr* **.**
opt-const-sect → *const-def-list* **;** | **ε**
const-def-list → *const-def* **,** *const-def-list* | *const-def*
const-def → **id = num**
opt-func-sect → *func-def-list* **;** | **ε**
func-def-list → *func-def* **,** *func-def-list* | *func-def*
func-def → **function id (** *opt-id-list* **)** = *expr*
opt-id-list → *id-list* | **ε**
id-list → **id ,** *id-list* | **id**
expr → *epr op expr* | **(** *expr* **)** | *func-call* | **id** | **num**
op → + | **-** | ***** | /
func-call → **id (** *opt-expr-list* **)**
opt-expr-list → *expr-list* | **ε**
expr-list → *expr* **,** *expr-list* | *expr*