Descrizione della sintassi di Grammo

Di seguito una descrizione discorsiva della sintassi del linguaggio Grammo, con espliciti rinvii a ciò che sarà trattato solo a livello semantico.

Struttura generale del programma

Un programma Grammo è costituito da una sequenza di dichiarazioni top-level (*TopDecl*), che possono essere:

- definizioni di funzione/procedura (*FuncDef*),
- dichiarazioni di variabili (*VarDecl*).

Formalmente:

- *Program ::= TopDeclList*
- *TopDeclList ::= TopDecl TopDeclList | /* empty */*
- *TopDecl ::= FuncDef | VarDecl*

Non viene imposto alcun ordine tra dichiarazioni di funzioni, procedure e variabili globali. È quindi possibile dichiarare funzioni dopo la loro prima chiamata: la correttezza di tali chiamate (in termini di esistenza e compatibilità dei tipi) sarà verificata dall'analisi semantica.

A livello semantico, si richiederà l'esistenza di una funzione di ingresso *main* con una firma predefinita (ad es. *func void -> main() { … }*), ma ciò non è codificato nella grammatica.

Tipi di dato e costanti

Grammo supporta quattro tipi primitivi:

- int (token *INT_TYPE*): numeri interi,
- real (*REAL_TYPE*): numeri reali in virgola mobile,
- bool (*BOOL_TYPE*): booleani,
- string (*STRING_TYPE*): stringhe.

Il simbolo void (*VOID_TYPE*) indica l'assenza di valore di ritorno per le funzioni/procedure.

Produzioni:

- *Type ::= INT_TYPE | REAL_TYPE | BOOL_TYPE | STRING_TYPE*
- *ReturnType ::= Type | VOID_TYPE*

Le costanti (*Const*) comprendono:

- interi (*INT_CONST*),
- reali (*REAL_CONST*),
- stringhe (*STRING_CONST*),
- booleani (*TRUE / FALSE*).

La forma lessicale delle costanti è verificata dal lexer; la gestione dei tipi (ad es. promozione *int→real*, compatibilità tra operandi) è rimandata al type checking semantico.

Dichiarazioni di funzione e procedura

Funzioni e procedure sono unificate sotto il costrutto *FuncDef*. La differenza è data dal tipo di ritorno:

- una **funzione** ha *ReturnType* diverso da void,
- una **procedura** ha *ReturnType* uguale a void.

Sintassi: *func <ReturnType> -> <nome>(<lista_parametri_opzionale>) { <corpo> }*

Produzione: *FuncDef ::= FUNC ReturnType ARROW ID LPAR ParamListOpt RPAR Block*

Esempi:

```
func int -> sum(int: a, int: b) {
  var int: res;
  res = a + b;
  return res;
}

func void -> print_hello() {
  << "Hello";
}
```

I parametri di funzione sono sempre dichiarati singolarmente e separati da virgole, nella forma: <*Type*> *:* <*identificatore*>

Produzioni:

- *ParamListOpt ::= ParamList | /* empty */*
- *ParamList ::= Param | Param COMMA ParamList*
- *Param ::= Type COLON ID*

Non è previsto (per ora) il passaggio per riferimento; tutti i parametri sono concettualmente passati "per valore". Eventuali regole su modificabilità e aliasing saranno definite semanticamente.

Vincoli delegati all'analisi semantica:

- Una funzione con *ReturnType* non void deve:
  - garantire la restituzione di un valore in tutti i possibili percorsi di esecuzione,
  - restituire un valore di tipo compatibile col *ReturnType*.
- Una funzione con *ReturnType = void* può:
  - non avere return (o avere return; senza espressione),
  - non può avere return <*Expr*>.

Questi vincoli non sono espressi nella grammatica, ma saranno verificati sull'AST.

Dichiarazioni di variabili

Le variabili possono essere dichiarate in due modi:

1. **Dichiarazione tipata con più variabili**:
   var int: a;
   var real: x, y;

   Produzione:

   - *VarDecl ::= VAR Type COLON IdList SEMI*
   - *IdList ::= ID | ID COMMA IdList*

2. **Dichiarazione con inizializzazione a costante, senza tipo esplicito**:

   var x = 10;
   var msg = "ciao";

   Produzione:

   - *VarDecl ::= VAR ID ASSIGN Const SEMI*

In quest'ultimo caso, il tipo della variabile è dedotto dalla costante (inferenza di tipo), ma ciò è responsabilità dell'analisi semantica. La grammatica consente qualsiasi combinazione *ID ASSIGN Const*, indipendentemente dalla compatibilità o dal tipo sottostante.

Le dichiarazioni di variabile sono ammesse sia a livello globale (*TopDecl*) sia all'interno dei blocchi delle funzioni e dei costrutti di controllo, tramite la produzione:

- *Stmt ::= VarDecl | ...*

Grammo, per l'analisi semantica, adotta un unico livello di scope per le variabili; questo aspetto è però trasparente a livello sintattico.

Subsection Blocchi e lista di istruzioni

Il corpo di una funzione/procedura, e in generale i blocchi di controllo (if, while, for), sono espressi dal non terminale *Block*:

- *Block ::= LBRACE StmtListOpt RBRACE*
- *StmtListOpt ::= StmtList | /* empty */*
- *StmtList ::= Stmt | Stmt StmtList*

Un blocco è quindi una sequenza (eventualmente vuota) di statement (*Stmt*).

Gli statement ammessi sono:

- dichiarazioni di variabili,
- assegnamenti,
- chiamate a procedura,
- I/O (output con/senza newline, input),
- return,
- strutture di controllo (if, while, for),
- blocchi annidati.

Produzione:

Stmt ::= VarDecl
| *AssignStmt SEMI*
| *ProcCall SEMI*
| *OutputStmt SEMI*
| *OutputLnStmt SEMI*
| *InputStmt SEMI*
| *ReturnStmt SEMI*
| *IfStmt*
| *WhileStmt*
| *ForStmt*
| *Block*

Istruzioni di assegnamento e chiamate

Assegnamento

L'assegnamento avviene tramite = *(token ASSIGN)*:

- *AssignStmt ::= ID ASSIGN Expr*

Esempio:

```
x = 10;
y = x + 3;
flag = x < y;
```

È consentita solo l'assegnazione di una singola variabile a destra; non vi è multi-assegnazione. Il controllo di tipo (compatibilità tra tipo della variabile e tipo dell'espressione) è semantico.

Chiamata a procedura/funzione come statement

La forma generica di chiamata è:

- *ProcCall ::= ID LPAR ArgListOpt RPAR*

con:

- *ArgListOpt ::= ArgList | /* empty */*
- *ArgList ::= Expr | Expr COMMA ArgList*

Esempi:

```
print_hello();
log_int(x);
update_state(x, y, z);
```

La distinzione tra:

- chiamata usata come statement (procedura → void),
- chiamata usata come espressione (funzione → tipo non void)

sarà imposta a livello semantico, verificando il *ReturnType* della *FuncDef* corrispondente.

Istruzioni di I/O

- *<< (OUT)* per output senza newline,
- *<<! (OUTLN)* per output con newline,
- *>> (IN)* per input,
- *#(Expr)* per interpolare un'espressione nei flussi di I/O.

Produzioni:

OutputStmt ::= OUT IOArgs
OutputLnStmt ::= OUTLN IOArgs
InputStmt ::= IN IOArgs

IOArgs ::= Expr IOArgs
| *HASH LPAR Expr RPAR IOArgs*
| */* empty */*

Esempi sintatticamente corretti:

```
<< "Somma: " #(a + b);
<<! "Fine";

>> "Inserisci n: " #(n);
>> #(name);
```

Scelte semantiche (non riflesse nella grammatica, ma previste):

- In **output** (*OutputStmt*, *OutputLnStmt*):
  - *Expr* può essere di qualunque tipo; sarà la semantica a definire le conversioni in stringa o a sollevare errori.
- In **input** (*InputStmt*):
  - gli argomenti *HASH LPAR Expr RPAR* dovranno essere identificatori (variabili assegnabili), non espressioni arbitrarie,
  - eventuali *Expr* libere (non interpolate con #) devono essere di tipo stringa; anche in questo caso la verifica è demandata alla semantica.

Istruzioni di ritorno

La produzione:

- *ReturnStmt ::= RETURN ExprOpt*
- *ExprOpt ::= Expr | /* empty */*

permette due forme:

- *return;*
- *return expr;*

Il vincolo tra forma del *return* e *ReturnType* della funzione sarà definito in analisi semantica:

- funzioni non-void → devono avere return *Expr* con tipo compatibile,
- funzioni void → possono avere *return*; o nessun *return*, ma non *return Expr*.

Strutture di controllo

If / elif / else

Il costrutto condizionale ha la forma:

```
if (condizione) {
   ...
} elif (altra_cond) {
   ...
} else {
   ...
}
```

Produzioni:

IfStmt ::= IF LPAR Expr RPAR Block ElifListOpt ElseOpt

ElifListOpt ::= ElifList
| */* empty */*

ElifList ::= Elif
| *Elif ElifList*

Elif ::= ELIF LPAR Expr RPAR Block

ElseOpt ::= ELSE Block
| */* empty */*

La condizione (*Expr*) in *if* ed *elif* deve essere di tipo bool; questo vincolo è demandato al type checker.

While

La struttura di ciclo while è:

```
while (condizione) {
   ...
}
```

Produzione:

- *WhileStmt ::= WHILE LPAR Expr RPAR Block*

La condizione deve essere bool (vincolo semantico).

For

Grammo prevede anche un ciclo for in stile C semplificato:

```
for (inizializzazione; condizione; aggiornamento) {
   ...
}
```

Produzioni:

ForStmt ::= FOR LPAR ForInitOpt SEMI ExprOpt SEMI ForUpdateOpt RPAR Block

ForInitOpt ::= AssignStmt
| */* empty */*

ForUpdateOpt ::= AssignStmt
| */* empty */*

ExprOpt ::= Expr
| */* empty */*

Esempi:

```
for (i = 0; i < n; i = i + 1) {
  sum = sum + i;
}

for (;;){
  ...
}
```

Scelte progettuali:

- Nel for l'inizializzazione e l'aggiornamento sono limitati ad assegnamenti (*AssignStmt*); non è possibile dichiarare nuove variabili (*var* …) all'interno dell'intestazione del for.
- La condizione (*ExprOpt*) può essere omessa: in tal caso, viene semanticamente interpretata come true (ciclo infinito).

Espressioni e precedenze

Le espressioni di Grammo sono completamente stratificate per codificare precedenza e associatività "matematica":

- livello più basso: || (OR),
- poi && (AND),
- poi confronti (==, <>, <, <=, >, >=),
- poi somma e sottrazione (+, -),
- poi moltiplicazione e divisione (*, /),
- poi operatori unari (!, -),
- infine literal, identificatori, chiamate e parentesi.

Produzioni:

Expr ::= OrExpr

OrExpr ::= AndExpr
| *OrExpr OR AndExpr*

AndExpr ::= EqualityExpr
| *AndExpr AND EqualityExpr*

EqualityExpr ::= RelExpr
| *EqualityExpr EQ RelExpr*
| *EqualityExpr NE RelExpr*

RelExpr ::= AddExpr
| *RelExpr LT AddExpr*
| *RelExpr LE AddExpr*
| *RelExpr GT AddExpr*
| *RelExpr GE AddExpr*

AddExpr ::= MulExpr
| *AddExpr PLUS MulExpr*
| *AddExpr MINUS MulExpr*

MulExpr ::= UnaryExpr
| *MulExpr TIMES UnaryExpr*
| *MulExpr DIV UnaryExpr*

UnaryExpr ::= NOT UnaryExpr
| *MINUS UnaryExpr*
| *Primary*

Primary ::= ID PrimaryTail
| *INT_CONST*
| *REAL_CONST*
| *STRING_CONST*
| *TRUE*
| *FALSE*
| *LPAR Expr RPAR*

PrimaryTail ::= LPAR ArgListOpt RPAR
| */* empty */*

In questo modo:

- *a + b * c == d* è interpretata come *((a) + (b * c)) == d*,
- *!a && b* come *((!a) && b)*.

Il fatto che **tutti gli operatori** possano, sintatticamente, applicarsi a qualunque combinazione di costanti, identificatori o chiamate (es. *3 + true*, *"ciao" < 5*, ecc.) è voluto: queste costruzioni sono accettate dalla grammatica, ma saranno scartate dal type checker in analisi semantica.

Aspetti rinviati esplicitamente all'analisi semantica

Per chiarezza, di seguito sono elencati i principali punti che non sono vincolati dalla grammatica ma saranno gestiti nella fase di analisi semantica e/o type checking:

1. **Identificazione di main**
    - Deve esistere una funzione di ingresso main con una firma prestabilita (ad es. *func void -> main() { ... }*).
    - Deve essere unica.
2. **Scope e simboli**
    - Gestione di un unico livello di scope per variabili e funzioni, con controllo su:
        - dichiarazioni multiple dello stesso identificatore,
        - uso di identificatori non dichiarati.
    - Vige il divieto di shadowing: non è possibile dichiarare una variabile locale (o parametro) con lo stesso nome di una variabile globale o di una funzione già definita. L'identificatore deve essere unico nel programma.
3. **Inferenza di tipo nelle dichiarazioni**
    - Per *var ID = Const;*, determinazione del tipo della variabile a partire dal tipo della costante.
4. **Type checking delle espressioni**
    - Compatibilità dei tipi per:
        - operatori aritmetici (+, -, *, /),
        - operatori logici (&&, ||, !),
        - operatori relazionali (==, <>, <, <=, >, >=),
        - assegnamenti (*ID = Expr*),
        - argomenti e parametri nelle chiamate di funzione.
5. **Funzioni e return**
    - Verifica che:
        - funzioni non-void restituiscano sempre un valore di tipo corretto,
        - funzioni void non restituiscano valori,
        - il numero e il tipo degli argomenti nelle chiamate (*ID(...)*) coincidano con la dichiarazione (*FuncDef*).
6. **Uso delle chiamate funzione/procedura**
    - Distinzione tra:
        - chiamate usate come espressioni (richiedono *ReturnType* non void),
        - chiamate usate come statement (*ProcCall*) (richiedono *ReturnType* void).
7. **Condizioni booleane nei controlli di flusso**
    - if, elif, while e il secondo elemento del for devono avere espressione di tipo bool.
8. **Vincoli sugli I/O**
    - In *InputStmt* (>>):
        - nelle occorrenze di *#(Expr) l'Expr* deve essere un *ID*,
        - gli eventuali altri argomenti (non *#(...)*) dovrebbero essere stringhe di prompt.

- In *OutputStmt/OutputLnStmt*:
  - definizione delle conversioni automatiche dei tipi non-stringa in stringa.