

# Progetto Compilatori

A.A. 2020/2021

Gaetano Antonucci

Alessio Romano

29 Gennaio 2021

## Indice

<b>1</b>	<b>Scelte progettuali</b>	<b>3</b>
1.1	Analizzatore Lessicale - JFlex . . . . .	3
1.2	Analizzatore Sintattico - CUP . . . . .	4
1.2.1	Grammatica Utilizzata . . . . .	4
1.2.2	Parser . . . . .	6
1.2.3	Generazione Abstract Syntax Tree . . . . .	6
1.3	Analizzatore Semantico . . . . .	7
1.4	Generazione del codice intermedio . . . . .	8
1.4.1	Gestione dei valori di ritorno multipli per una procedura . . . . .	8
1.4.2	Gestione di assegnazioni multiple . . . . .	8
1.4.3	Gestione della procedura main . . . . .	9
1.4.4	Gestione del costrutto while . . . . .	9
<b>2</b>	<b>Regole di Type Checking implementate</b>	<b>10</b>
2.1	Tipi Primitivi . . . . .	10
2.2	Dichiarazioni di Variabili . . . . .	10
2.3	Operazioni Unarie . . . . .	10
2.4	Operazioni Binarie . . . . .	10
2.5	Chiamata a Procedura . . . . .	10
2.6	Statement . . . . .	10
2.6.1	if-then . . . . .	10
2.6.2	if-then-else . . . . .	10
2.6.3	if-then-elif-else . . . . .	10
2.6.4	while . . . . .	10
2.6.5	while-return . . . . .	11
2.6.6	readln . . . . .	11
2.6.7	write . . . . .	11

2.6.8	simple-assign . . . . .	11
2.6.9	multiple-assign . . . . .	11
2.6.10	return . . . . .	11
2.7	Tabelle di Compatibilità . . . . .	12

# 1 Scelte progettuali

## 1.1 Analizzatore Lessicale - JFlex

Per gestire gli errori "Commento non chiuso" e "Stringa costante non completata", sono stati utilizzati gli stati in JFlex: **COMMENT** e **COMMENT2** per i commenti, **STRING** per le stringhe costanti per rendere più agevole la rilevazione degli errori.

Il procedimento per i commenti è il seguente:

- Si accede allo stato **COMMENT** all'inizio di un commento (`/*`).
- Se nello stato **COMMENT** si legge un asterisco (`*`) si passa nello stato **COMMENT2** poichè potrebbe corrispondere all'asterisco che precede il simbolo di fine commento
- Se in **COMMENT2** si legge il simbolo di fine commento (`/`) si ritorna nello stato iniziale **YYINITIAL**. In tutti gli altri casi si torna in **COMMENT**
- Se si raggiunge la fine del file (**EOF**) mentre ci si trova in **COMMENT** o **COMMENT2** viene generato l'errore "Commento non chiuso"

Il procedimento per le stringhe è il seguente:

- Si accede allo stato **STRING** quando si leggono i doppi apici (`"`)
- All'interno dello stato, tutti i caratteri letti vengono aggiunti in una variabile di tipo `StringBuffer` che rappresenta la stringa costante in questione
- Sono stati gestiti in maniera specifica tutti i caratteri di escape (e.g. `\n`, `\t`, `\r`) in modo da inserirli letteralmente nella variabile precedentemente menzionata
- A causa della naturale presenza dei caratteri di escape nel sorgente analizzato, si sono dovuti distinguere i caratteri di escape espliciti, inseriti in una stringa costante, da quelli impliciti nel sorgente i quali dovranno essere ignorati

Nelle precedenti esercitazioni (es3 e es4) si era deciso di non permettere che i numeri in virgola mobile potessero avere solo zeri dopo la virgola (e.g. `34.0`).

In un secondo momento ci si è accorti che, in molti linguaggi di programmazione, quali il C, è possibile utilizzare numeri nella precedente forma in maniera tale da usare una rappresentazione in virgola mobile di un numero intero per evitare troncamenti nelle operazioni. Di conseguenza, l'espressione regolare per il riconoscimento dei numeri in virgola mobile è stata semplificata.

## 1.2 Analizzatore Sintattico - CUP

### 1.2.1 Grammatica Utilizzata

La grammatica fornita nelle specifiche del linguaggio Toy è stata modificata accertandosi di non modificare il linguaggio generato. Sono stati introdotti due *nuovi* non-terminali, `ProcBody` e `ParIdList` in modo da semplificare le produzioni di `Proc`, in particolare:

- `ProcBody` è stato aggiunto per semplificare lo sviluppo del parser tramite CUP
- `ParIdList` (le cui produzioni sono equivalenti a quelle di `IdList`) è stato aggiunto perché è stato deciso che gli elementi della tabella dei simboli venissero creati dal parser. Conseguentemente, è necessario distinguere il tipo di lookup da effettuare nella tabella dei simboli nel caso dei parametri di una procedura (lookup fatta solo nello scope corrente creato dalla procedura e non in tutti gli scope).

La grammatica risultante è la seguente:

```
Program := VarDeclList ProcList
VarDeclList := VarDecl VarDeclList
            | /* empty */
VarDecl := Type IdListInit SEMI
Type := INT
      | BOOL
      | FLOAT
      | STRING
IdListInit := ID
            | IdListInit COMMA ID
            | ID ASSIGN Expr
            | IdListInit COMMA ID ASSIGN Expr
ProcList := Proc
          | Proc ProcList
Proc := PROC ID LPAR ParamDeclList RPAR ResultTypeList COLON ProcBody
      | PROC ID LPAR RPAR ResultTypeList COLON ProcBody
ProcBody := VarDeclList StatList RETURN ReturnExprs CORP SEMI
          | VarDeclList RETURN ReturnExprs CORP SEMI
ParamDeclList := ParDecl
              | ParamDeclList SEMI ParDecl
ParDecl := Type ParIdList
ParIdList := ID
           | ParIdList COMMA ID
```

```

ResultTypeList := ResultType
                | ResultType COMMA ResultTypeList
ResultType := Type
            | VOID
StatList := Stat SEMI
          | Stat SEMI StatList
Stat := IfStat
      | WhileStat
      | ReadlnStat
      | AssignStat
      | CallProc
IfStat := IF Expr THEN StatList ElifList Else FI
ElifList := Elif ElifList
          | /* empty */
Elif := ELIF Expr THEN StatList
Else := ELSE StatList
      | /* empty */
WhileStat := WHILE StatList RETURN Expr DO StatList OD
          | WHILE Expr DO StatList OD
ReadlnStat := READ LPAR IdList RPAR
IdList := ID
        | IdList COMMA ID
WriteStat := WRITE LPAR ExprList RPAR
AssignStat := IdList ASSIGN ExprList
CallProc := ID LPAR ExprList RPAR
          | ID LPAR RPAR
ReturnExprs := ExprList
            | /* empty */
ExprList := Expr
          | Expr COMMA ExprList

```

```

Expr :=  NULL
        |  TRUE
        |  FALSE
        |  INT_CONST
        |  FLOAT_CONST
        |  STRING_CONST
        |  ID
        |  MINUS Expr
        |  Expr PLUS Expr
        |  Expr MINUS Expr
        |  Expr TIMES Expr
        |  Expr DIV Expr
        |  NOT Expr
        |  Expr AND Expr
        |  Expr OR Expr
        |  Expr GT Expr
        |  Expr GE Expr
        |  Expr LT Expr
        |  Expr LE Expr
        |  Expr EQ Expr
        |  Expr NE Expr
        |  CallProc

```

### 1.2.2 Parser

In fase di progettazione del compilatore, già a partire dall'analizzatore sintattico, si è deciso di gestire la gerarchia di tabelle dei simboli tramite uno stack. Sarà quindi il parser a creare le singole tabelle e a collegarle tra di loro ogni volta che un costrutto del linguaggio definisce un nuovo scope. Di conseguenza, nelle azioni semantiche delle produzioni dei non terminali che definiscono un nuovo scope, si è inserito il codice per la gestione dello stack di tabelle.

### 1.2.3 Generazione Abstract Syntax Tree

Per la gestione delle componenti dell'albero sintattico, si è optato per l'utilizzo di una classe padre `Node.java` contenente l'attributo nome in quanto comune a tutte le componenti. Ogni classe rappresentante una componente diversa dell'albero, quindi, estende la classe `Node.java`.

Ragionamento analogo al punto precedente, è stato fatto per le classi che rappresentano gli Statement del linguaggio di programmazione Toy. In questo caso, la classe padre è `StatOp.java`.

Siccome ci sono alcune componenti dell'albero sintattico che sono comuni a più produzioni della grammatica, si è scelto di avere, `Expr` e `IdListInit` come interfacce. Questo perché,

ad esempio, `ID` (che corrisponde ad una foglia dell'albero) può essere sia visto come `Expr` sia come un elemento di `IdListInit`. Ciò è stato fatto per ottenere, ad esempio, una lista di `Expr` i cui elementi sono eterogenei (e anche perché in Java non esiste l'ereditarietà multipla). Per applicare il pattern Visitor, si è utilizzata l'interfaccia `Visitor.java` che, per nostra scelta, contiene le firme distinte dei vari metodi `visit()`, ognuno dei quali è specifico per una componente dell'albero sintattico.

Una visualizzazione in XML dell'AST è stata ottenuta tramite un visitor (`XmlVisitor.java`) Per la generazione del file xml si è utilizzata la libreria `jdom` (v 2.0.6) inserita come external library

### 1.3 Analizzatore Semantico

L'analisi semantica viene effettuata utilizzando due visite dell'AST:

- La prima serve ad aggiungere informazioni alla symbol table necessarie per il type checking. In particolare, vengono aggiunti:
  - I tipi delle variabili all'interno dello specifico scope;
  - I tipi di ritorno delle procedure (eventualmente multipli) nello scope del chiamante;
  - I tipi dei parametri delle procedure all'interno dello scope della procedura;
- La seconda esegue il type checking, e altri controlli semantici, quali i seguenti:
  - Controllo della dichiarazione di una singola procedura main;
  - Controllo del tipo di ritorno della procedura main (deve avere forma `proc main() void`);
  - Controllo degli utilizzi di procedure non dichiarate all'interno del codice;
  - Controllo degli utilizzi di variabili non dichiarate all'interno del codice;
  - Controllo dei parametri della procedura `write` (non accetta valori null);

Nota: A differenza di quanto indicato nelle specifiche, è stata utilizzata la forward reference (fref) per le procedure. Di conseguenza, non è strettamente necessario dichiarare ogni procedura prima del suo uso

## 1.4 Generazione del codice intermedio

### 1.4.1 Gestione dei valori di ritorno multipli per una procedura

In fase di progettazione del compilatore si è deciso di gestire i valori di ritorno multipli per una procedura nel seguente modo:

1. Il tipo di ritorno nella firma della funzione in C corrisponderà al primo valore di ritorno nella firma della procedura in Toy. Questa scelta è dovuta al fatto che il linguaggio C non permette l'utilizzo di valori di ritorno multipli per una funzione.
2. Per i valori di ritorno successivi al primo, vengono definite delle variabili globali i cui nomi sono nella forma "procedureName\_number" dove number parte da 1.
3. Si assegnano i valori di ritorno successivi al primo alle variabili definite al punto precedente.
4. Si aggiunge la clausola **return** seguita dal primo valore di ritorno della corrispondente procedura in Toy.

### 1.4.2 Gestione di assegnazioni multiple

In fase di progettazione del compilatore si è deciso di gestire le assegnazioni multiple secondo il seguente procedimento: Si distinguono due possibili casi:

1. Nel lato destro dell'assegnazione multipla non sono presenti chiamate a procedure con valori di ritorno multipli
2. Nel lato destro dell'assegnazione multipla sono presenti chiamate a procedure con valori di ritorno multipli

In entrambi i casi, l'assegnazione multipla in Toy è la seguente:  $var_1, \dots, var_n := val_1, \dots, val_n$ ; Il codice C generato, è ottenuto rispettivamente come segue:

1. Nel primo caso, il codice generato, è  $var_i = val_i; \forall i \in \{1, \dots, n\}$
2. Nel secondo caso, in Toy, è possibile che i valori  $val_i$  nella parte destra dell'assegnazione multipla siano in numero minore rispetto a quelli della parte sinistra. Di conseguenza, all'intervallo  $var_i \dots var_k$  corrisponde un solo valore  $val_i$  corrispondente ad una chiamata a procedura con  $k$  valori di ritorno (denotata come **procName()**). In questo caso, il codice generato è  $var_i = procName(); var_j = procName\_j \forall j \in \{i + 1, \dots, k\}$   
Il motivo di questo approccio è descritto nella sezione 1.4.1



### 1.4.3 Gestione della procedura main

In fase di progettazione del compilatore, per la gestione della procedura main, si è deciso di permetterne una firma unica poiché il linguaggio C, per la funzione main, non prevede valore di ritorno diverso da int. In altri termini, la procedura main in Toy sarà sempre nella forma:

```
proc main() void vardecl1 ... vardecln; stat1, ... statn; ->corp;
```

Il corrispondente codice C generato per la funzione main (a causa della limitazione introdotta dal linguaggio) sarà sempre nella forma:

```
int main(){stat1, ..., statn; return 0;}
```

### 1.4.4 Gestione del costrutto while

Le specifiche sintattiche del linguaggio Toy prevedono le due seguenti produzioni per il costrutto while:

1. while statlist1 return expr do statlist2 od
2. while expr do statlist od

In C, il costrutto while corrispondente sarà, rispettivamente:

1. statlist1 while(expr){statlist2; statlist1;}
2. while(expr){statlist;}

## 2 Regole di Type Checking implementate

### 2.1 Tipi Primitivi

$$\begin{array}{l} \Gamma \vdash \text{null} : \text{null} \quad \Gamma \vdash \text{true} : \text{boolean} \quad \Gamma \vdash \text{false} : \text{boolean} \\ \Gamma \vdash \text{int} : \text{int} \quad \Gamma \vdash \text{float} : \text{float} \quad \Gamma \vdash \text{string} : \text{string} \quad \Gamma \vdash \text{bool} : \text{boolean} \end{array}$$

### 2.2 Dichiarazioni di Variabili

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

### 2.3 Operazioni Unarie

$$\frac{\Gamma \vdash e : \tau_1 \quad \text{optype1}(op, \tau_1) = \tau}{\Gamma \vdash (op \ e) : \tau}$$

### 2.4 Operazioni Binarie

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \text{optype2}(op, \tau_1, \tau_2) = \tau}{\Gamma \vdash (e_1 \ op \ e_2) : \tau}$$

### 2.5 Chiamata a Procedura

$$\frac{\Gamma \vdash f : \tau_i^{i \in 1 \dots n} \rightarrow \tau_j^{j \in 1 \dots m} \quad \Gamma \vdash e_i : \tau_i^{i \in 1 \dots n}}{\Gamma \vdash f(e_i^{i \in 1 \dots n}) : \tau_j^{j \in 1 \dots m}}$$

### 2.6 Statement

#### 2.6.1 if-then

$$\frac{\Gamma \vdash e : \text{boolean} \quad \Gamma \vdash stmt}{\Gamma \vdash \text{if } e \text{ then } stmt \text{ fi}}$$

#### 2.6.2 if-then-else

$$\frac{\Gamma \vdash e : \text{boolean} \quad \Gamma \vdash stmt_1 \quad \Gamma \vdash stmt_2}{\Gamma \vdash \text{if } e \text{ then } stmt_1 \text{ else } stmt_2 \text{ fi}}$$

#### 2.6.3 if-then-elif-else

$$\frac{\Gamma \vdash e_j^{j \in 1 \dots m} : \text{boolean} \quad \Gamma \vdash stmt_i^{i \in 1 \dots 3}}{\Gamma \vdash \text{if } e_1 \text{ then } stmt_1 \text{ (elif } e_j^{j \in 2 \dots m} \text{ then } stmt_2)_t^{t \in 1 \dots k} \text{ else } stmt_3 \text{ fi}}$$

#### 2.6.4 while

$$\frac{\Gamma \vdash e : \text{boolean} \quad \Gamma \vdash stmt}{\Gamma \vdash \text{while } e \text{ do } stmt \text{ od}}$$

### 2.6.5 while-return

$$\frac{\Gamma \vdash e : \text{boolean} \quad \Gamma \vdash stmt_1 \quad \Gamma \vdash stmt_2}{\Gamma \vdash \text{while } stmt_1 \rightarrow e \text{ do } stmt_2 \text{ od}}$$

### 2.6.6 readln

$$\frac{(x_i^{i \in 1 \dots n} : \tau_i^{i \in 1 \dots n}) \in \Gamma}{\Gamma \vdash \text{readln}(x_i^{i \in 1 \dots n})}$$

### 2.6.7 write

$$\frac{\Gamma \vdash e : \tau \quad (\tau \neq \text{void})^1}{\Gamma \vdash \text{write}(e : \tau)}$$

### 2.6.8 simple-assign

$$\frac{(x : \tau) \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e}$$

### 2.6.9 multiple-assign

$$\frac{(x_i^{i \in 1 \dots n} : \tau_i^{i \in 1 \dots n}) \in \Gamma \quad \Gamma \vdash e_j^{j \in 1 \dots n} : \tau_j^{j \in 1 \dots n}}{\Gamma \vdash x_i^{i \in 1 \dots n} := e_j^{j \in 1 \dots n}}$$

### 2.6.10 return

$$\frac{(\$ret : \tau) \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash \rightarrow e}$$

---

<sup>1</sup>Si vuole intendere che il tipo di  $e$  deve essere diverso da `void`. Per ulteriori informazioni consultare le scelte di sviluppo del compilatore.

## 2.7 Tabelle di Compatibilità

op	operand	result
-	integer	integer
-	float	float
!	boolean	boolean

(a) optype1

op	first operand	second operand	result
+ - * /	integer	integer	integer
+ - * /	integer	float	float
+ - * /	float	integer	float
+ - * /	float	float	float
&&	boolean	boolean	boolean
< = > <= >= <>	integer	integer	boolean
< = > <= >= <>	integer	float	boolean
< = > <= >= <>	float	integer	boolean
< = > <= >= <>	float	float	boolean
< = > <= >= <>	string	string	boolean

(b) optype2

Figura 1: Relazioni di tipo per gli operatori primitivi. Gli operatori aritmetici lavorano sia su numeri interi sia su numeri in virgola mobile. Gli operatori logici ! && || (not, and e or) lavorano su boolean. Gli operatori di comparazione lavorano su tipi primitivi diversi da boolean.