# Progetto Compilatori

A.A. 2020/2021

Gaetano Antonucci Alessio Romano

29 Gennaio 2021

# Indice

# 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` poiché 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, tutto 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 (es.3 e es.4) si era deciso di **non permettere** che i numeri in virgola mobile potessero avere zeri come ultima cifra decimale 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 aritmetiche. 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 si è deciso che gli elementi della tabella dei simboli venissero creati dal parser. Conseguentemente, si è reso necessario distinguere il tipo di lookup da effettuare nella tabella dei simboli nel caso dei parametri di una procedura. Cioè, la lookup viene effettuata 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
```

*continua nella pagina successiva ...*

```
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 le loro righe) e a collegarle tra di loro ogni volta che effettua una riduzione per i non terminali `Program` e `Proc`. Di conseguenza, nelle loro azioni semantiche, si è inserito il codice per la gestione dello stack di tabelle.

### 1.2.3 Generazione dell'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 comune a tutte le componenti, dunque, ogni classe rappresentante una componente diversa dell'albero 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 alcune componenti dell'AST svolgono ruoli diversi a seconda della produzione della grammatica dalla quale derivano, 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 un metodo `visit()` distinto per ogni componente dell'AST.

Una visualizzazione in XML dell'AST è stata ottenuta tramite il 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 necessarie per il type checking alla symbol table. 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 così come definito dalle regole di inferenza nella sezione 2, e altri controlli semantici, quali i seguenti:
  - Controllo della dichiarazione di una singola procedura main;
  - Controllo del tipo di ritorno della procedura main (che per scelta progettuale deve avere la 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 $k$ variabili globali, con $k$ pari al numero di valori di ritorno totale meno uno. Tali variabili globali saranno identificate da *procedureName*_$i$ dove $i \in \{1, \ldots, k\}$.

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 procedimento descritto di seguito.

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:

$$var_1, \ldots, var_n := val_1, \ldots, val_n;$$

Il codice C generato, è ottenuto rispettivamente come segue:

1. Nel primo caso, il codice generato, è

$$var_i = val_i; \quad \forall i \in \{1, \ldots, n\}$$

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 $var_i \ldots var_k$ corrisponde un solo valore $val_i$ corrispondente ad una chiamata a procedura con $k$ valori di ritorno (denotata nel seguito come `procName()`). In questo caso, il codice generato è

$$\begin{aligned} &var_i = procName(); \\ &var_j = procName\_j; \quad \forall j \in \{i+1, \ldots, k\} \end{aligned}$$

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 permettere un'unica firma 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`
$vardecl_1$
$\vdots$
$vardecl_n$;
$stat_1$;
$\vdots$
$stat_n$;
`->`
`corp;`

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

`int main(){`
$stat_1$;
$\vdots$
$stat_n$;
`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` $\texttt{StatList}_1$ `RETURN Expr DO` $\texttt{StatList}_2$ `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

$$\Gamma \vdash null : null \qquad \Gamma \vdash true : boolean \qquad \Gamma \vdash false : boolean$$

$$\Gamma \vdash int : int \qquad \Gamma \vdash float : float \qquad \Gamma \vdash string : string \qquad \Gamma \vdash bool : boolean$$

## 2.2 Dichiarazioni di Variabili

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

## 2.3 Operazioni Unarie

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

## 2.4 Operazioni Binarie

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

## 2.5 Chiamata a Procedura

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

## 2.6 Statement

### 2.6.1 if-then

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

### 2.6.2 if-then-else

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

### 2.6.3 if-then-elif-else

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

### 2.6.4 while

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

### 2.6.5 while-return

$$\frac{\Gamma \vdash e\colon \mathit{boolean} \quad \Gamma \vdash \mathit{stmt}_1 \quad \Gamma \vdash \mathit{stmt}_2}{\Gamma \vdash \texttt{while}\ \mathit{stmt}_1 \texttt{->}\ e\ \texttt{do}\ \mathit{stmt}_2\ \texttt{od}}$$

### 2.6.6 readln

$$\frac{(x_i^{\,i\in 1\dots n}\colon \tau_i^{\,i\in 1\dots n}) \in \Gamma}{\Gamma \vdash \texttt{readln}(x_i^{\,i\in 1\dots n})}$$

### 2.6.7 write

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

### 2.6.8 simple-assign

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

### 2.6.9 multiple-assign

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

### 2.6.10 return

$$\frac{(\$ret_i^{\,i\in 0\dots n}\colon \tau_i^{\,i\in 0\dots n}) \in \Gamma \quad \Gamma \vdash e_i^{\,i\in 0\dots n}\colon \tau_i^{\,i\in 0\dots n}}{\Gamma \vdash \texttt{->} e_i^{\,i\in 0\dots n}}$$

$\$ret$ nell'environment $\Gamma$ rappresenta i tipi dei valori di ritorno nella firma della procedura e non è un identificatore valido del linguaggio Toy e pertanto compare solo in queste regole di inferenza. Per $i = 0$ si intende che il tipo di ritorno della procedura è `void`.

[1] 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.