Università degli Studi di Salerno
Dipartimento di Informatica

Progetto di Compilatori

MyFun

Docente

Prof. Gennaro Costagliola

Gruppo

Elio Testa
Matr. 05225 01301

Anno Accademico 2021/2022

Sommario

Struttura del documento

Il presente documento riporta, per ognuna delle fasi del compilatore realizzato, le scelte progettuali adottate e le modifiche effettuate rispetto alle indicazioni fornite dalle tracce delle esercitazioni.

1. Project SDK e Project Language Level

Project SDK: 17 version (17.0.1)

Project Language Level: SDK Default (17 Sealed types, always-strict floating-point semantics)

2. Analisi lessicale

Implementazione aderente alle indicazioni fornite.

3. Analisi sintattica

3.1 Risoluzione dei conflitti della grammatica in Cup

1. Una volta inserite le regole della grammatica data, ho provato a generare il parser con Cup ed ho analizzato il file "dump.txt" prodotto in output: sono stati individuati dei conflitti.
2. Ho aggiunto le **regole di precedenza e associatività**, seguendo le specifiche date, ed il numero dei conflitti è sceso ma non si è azzerato; nello specifico ho notato che la totalità dei conflitti è causata dalle produzioni *StatList -> Stat* e *Stat -> /*empty*/*.
3. Ho risolto il conflitto causato da ***StatList -> Stat*** e ***Stat -> /* empty */*** rimuovendo quest'ultima produzione e cambiando la prima con ***StatList -> /*empty*/***, in questo modo invece di arrivare a Stat per andare in **/*empty*/** è possibile anticipare restituendo la lista vuota di Stat. A seguito di ciò, il numero dei conflitti si è **azzerato**.
4. Un'altra modifica apportata alla grammatica riguarda la produzione ***Const -> BOOL_CONST*** che ho modificato con ***Const -> Bool_Const*** e ho aggiunto 2 produzioni: ***Bool_Const -> TRUE*** e ***Bool_Const -> FALSE*** in modo da poter realizzare anche l'inizializzazione di una variabile con true o false.

Di seguito, vengono riportate le regole di precedenza impiegate:

```
precedence left OR;
precedence right AND;
precedence left NOT;
```

```
precedence left AND;
precedence left EQ, NE, LT, LE, GT, GE;
precedence left STR_CONCAT;
precedence left PLUS, MINUS;
precedence left TIMES, DIV, DIVINT;
precedence right POW;
precedence left LPAR, RPAR;
```

3.2 Ulteriori scelte effettuate per la realizzazione dell'Abstract Syntax Tree

- Utilizzo della classe "CoppiaIdExprNode" per la memorizzazione di coppie (identificatore, valore); questa verrà, a sua volta, impiegata nella classe "IdListInitNode" per mantenere una lista di "CoppiaIdExprNode", in modo tale che durante l'inizializzazione sarà possibile sapere se a una variabile dichiarata è stato assegnato o meno un valore iniziale.
- Utilizzo della classe "CoppiaIdConstNode" per la memorizzazione di coppie (identificatore, valore); questa verrà, a sua volta, impiegata nella classe "IdListInitObblNode" per mantenere una lista di " CoppiaIdConstNode ", in modo tale che durante l'inizializzazione sarà possibile sapere se a una variabile dichiarata è stato assegnato o meno un valore iniziale, la differenza fra i due sta che IdListInitObbl è utilizzato per le variabili di tipo var in modo da distinguere le due produzioni della grammatica.
- Realizzazione della superclasse Expr e dell'interfaccia Stat, in quanto in Java una classe non può estendere due classi e ci saremmo ritrovati CallFun che avrebbe dovuto estendere sia Expr che Stat.

4. Analisi semantica

4.1 Regole di type checking implementate nel formato regole di inferenza

$\forall \tau_i \in \{\text{bool, integer, real, string}\}$, con $i \in \mathbb{N}$, sono definite le seguenti regole di type checking:

$$\frac{\Gamma(id) = \tau}{\Gamma \vdash id : \tau}$$

$$\Gamma \vdash \text{true} : \text{boolean}$$

$$\Gamma \vdash \text{false} : \text{boolean}$$

$$\Gamma \vdash \text{integer\_const} : \text{integer}$$

$$\Gamma \vdash \text{real\_const} : \text{real}$$

$$\Gamma \vdash \text{string\_const} : \text{string}$$

$$\frac{\Gamma \vdash stmt_1 : notype \quad \Gamma \vdash stmt_2 : notype}{\Gamma \vdash stmt_1 ; stmt_2 : notype}$$

$$\frac{\Gamma \vdash f : \tau_1, \tau_2, \dots, \tau_n \rightarrow \tau \quad \Gamma \vdash e_i : \tau_i^{\,i \in 1 \dots n}}{\Gamma \vdash f (e_1, e_2, \dots, e_n) : \tau}$$

$$\frac{\Gamma \vdash f : \tau_1, \tau_2, \dots, \tau_n \rightarrow notype \quad \Gamma \vdash e_i : \tau_i^{\,i \in 1 \dots n}}{\Gamma \vdash f (e_1, e_2, \dots, e_n) : notype}$$

$$\frac{\Gamma(id) = \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash id := e : notype}$$

$$\frac{\Gamma(id) = \tau \quad stmt : notype}{\Gamma \vdash id; stmt : \ notype}$$

$$\frac{\Gamma \vdash e : \ boolean \quad \Gamma \vdash block : notype}{\Gamma \vdash \textbf{while} \ e \ \textbf{loop} \ block \ \textbf{end loop} : notype}$$

$$\frac{\Gamma \vdash e : \ boolean \quad \Gamma \vdash block : \ notype \ \Gamma \vdash block_2 : \ notype}{\Gamma \vdash \textbf{if} \ e \ \textbf{then} \ block \ \textbf{else} \ block_2 \ \textbf{end if} : notype}$$

$$\frac{\Gamma \vdash e : \ boolean \quad \Gamma \vdash block : \ notype}{\Gamma \vdash \textbf{if} \ e \ \textbf{then} \ block \ \textbf{end if} : notype}$$

$$\frac{\Gamma \vdash e : \ \tau_1 \qquad optype1(op_1 , \ \tau_1) = \ \tau}{\Gamma \vdash \ op_1 \ \mathrm{e} \ : \tau}$$

$$\frac{\Gamma \vdash e_1 : \ \tau_1 \qquad \Gamma \vdash e_2 : \ \tau_2 \qquad optype2(op_2 , \ \tau_1 , \ \tau_2 \ ) = \ \tau}{\Gamma \vdash \ \mathrm{e}_1 \ op_2 \ \mathrm{e}_2 \ : \tau}$$

$$\frac{(x_1 : \tau_1) \in \Gamma \quad (x_2 : \tau_2) \in \Gamma \quad \dots \quad (x_n : \tau_n) \in \Gamma \quad \mathrm{n} \in \mathrm{N}}{\Gamma \vdash \textbf{readln} \ (x_1 , \ x_2 , \dots , \ x_n)}$$

$$\frac{\Gamma \vdash e_1 : \ \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \ \dots \ \Gamma \vdash e_n : \ \tau_n \quad \mathrm{n} \in \mathrm{N}}{\Gamma \vdash \textbf{write} \ (e_1 , \ e_2 , \dots , \ e_n)}$$

$$\frac{\Gamma[id \to \tau] \vdash stmt : \ notype}{\Gamma \vdash \tau \ id; stmt : notype}$$

4.2 Typing relations

Optype1(Op$_1$, Operando)

Op$_1$	Operando	Risultato
-	integer	integer
-	real	real
!	boolean	boolean

Optype2(Op$_1$, Operando$_1$, Operando$_2$)

Op$_2$	Operando$_1$	Operando$_2$	Risultato
+ - * / ^	integer	integer	integer
+ - * / ^	integer	real	real
+ - * / ^	real	integer	real
+ - * / ^	real	real	real
div int	integer	integer	integer
div int	integer	real	integer
div int	real	integer	integer
div int	real	real	integer
&& ||	bool	bool	boolean
= <> < <= > >=	bool	bool	boolean
= <> < <= > >=	integer	integer	boolean
= <> < <= > >=	integer	real	boolean
= <> < <= > >=	real	integer	boolean
= <> < <= > >=	real	real	boolean
= <> < <= > >=	string	string	boolean
String Concat	string	string	string
String Concat	string	real	string
String Concat	real	string	string
String Concat	string	integer	string
String Concat	integer	string	string
String Concat	string	boolean	string
String Concat	boolean	string	string

Tabella di compatibilità tra tipi

Tipo	Tipi compatibili	
integer	integer	
real	real	integer
boolean	boolean	
string	string	

5. Generazione del codice C

Attenendomi alle specifiche del linguaggio MyFun, ho deciso che i valori booleani saranno stampati come "true" e "false" (quindi non come "1" oppure "0"); se saranno richiesti in input, ho deciso di fare in modo che l'utente debba inserire 1 per denotare una variabile booleana vera e qualsiasi altro numero per denotare una variabile booleana falsa (in generale, è suggerito inserire 0).

Essendo stato deciso, nell'analisi semantica, che il tipo "real" è compatibile con il tipo "int", sarà possibile assegnare un intero ad una variabile dichiarata "real", in modo coerente anche con quanto è possibile fare nel linguaggio C.

Per quanto riguarda la concatenazione fra tipi, ho utilizzato una funzione C che mi permette la concatenazione sia di stringhe che di interi che di real che verrà generata solo all'utilizzo della concatenazione corretta. (Esempio ho la concatenazione fra una stringa e un intero, allora genera la funzione che permetterà la concatenazione fra stringa e intero).

Qui riporto le 3 funzioni in base ai casi:

```
char* concatRealToString(char *s1, float i) {
    char* s = malloc(256);
    sprintf(s, "%s%.2f", s1, i);
    return s;
}

char* concatIntegerToString(char *s1, int i) {
    char* s = malloc(256);
    sprintf(s, "%s%d", s1, i);
    return s;
}

char* concatStringToString(char *s1, char* i) {
    char* s = malloc(256);
    sprintf(s, "%s%s", s1, i);
    return s;
}
```

Per quanto riguarda le variabili var, in particolare nel caso in cui ho una inizializzazione con una costante stringa, in C non è possibile assegnare ad una variabile char* una stringa in questo modo: char* str = "elio" , quindi ho optato per creare anche in questo caso una funzione che mi permette di passare la parte costante e creare quella che è la stringa in C.

Qui riporto la funzione per la creazione della stringa

```
char* creaString(char* string){
   char* s = malloc(256);
   sprintf(s, "%s",string);
   return s;
}
```

Infine per quanto riguarda la variabile out in una funzione, ho optato per utilizzare il doppio asterisco di C permettendomi così di poter lavorare sull'indirizzo di memoria della variabile passata in input alla funzione.

Qui riporto uno snap del codice dato come esempio

```
float sommac(int a, float b, char* *size) {

 float result;

 result = a + b + c;

 if (result> 100) {

  char* valore = creaString( "grande");
  *size = valore;

 }else {

  char* valore = creaString( "piccola");
  *size = valore;

 }
 return result;
}
```