Università degli Studi di Salerno

Dipartimento di Informatica



Progetto di Compilatori

MyFun

|  |  |
| --- | --- |
| **Docente** | **Gruppo** |
| Prof. Gennaro Costagliola | Elio Testa  *Matr. 05225 01301* |

Anno Accademico 2021/2022

Sommario

[Struttura del documento 1](#_Toc94030964)

[1. Project SDK e Project Language Level 1](#_Toc94030965)

[2. Analisi lessicale 1](#_Toc94030966)

[3. Analisi sintattica 1](#_Toc94030967)

[3.1 Risoluzione dei conflitti della grammatica in Cup 1](#_Toc94030968)

[3.2 Ulteriori scelte effettuate per la realizzazione dell'Abstract Syntax Tree 2](#_Toc94030969)

[4. Analisi semantica 3](#_Toc94030970)

[4.1 Regole di type checking implementate nel formato regole di inferenza 3](#_Toc94030971)

[4.2 Typing relations 5](#_Toc94030972)

[5. Generazione del codice C 7](#_Toc94030973)

# 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.

# 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)

# Analisi lessicale

Implementazione aderente alle indicazioni fornite.

# Analisi sintattica

### 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;

### 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.

# Analisi semantica

### Regole di type checking implementate nel formato regole di inferenza

sono definite le seguenti regole di type checking:

### Typing relations

Optype1(Op1, Operando)

|  |  |  |
| --- | --- | --- |
| Op1 | Operando | Risultato |
| - | integer | integer |
| - | real | real |
| ! | boolean | boolean |

Optype2(Op1, Operando1, Operando2)

|  |  |  |  |
| --- | --- | --- | --- |
| Op2 | Operando1 | Operando2 | 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 | |

# 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;  
}