

UNIVERSITÀ DEGLI STUDI DI BERGAMO
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA
INFORMATICA
SCUOLA DI INGEGNERIA

Parsing in C e traduzione in Assembler MIPS

Creazione di un compilatore semplificato per C (lessico, sintassi,
semantica) usando ANTLR e traduzione semplificata in
Assembler MIPS

Autori

Samuele FERRI

Simone SUDATI

31 Gennaio 2021

Indice

| | | |
|----------|---|-----------|
| 1 | Introduzione | 1 |
| 1.1 | Obiettivi | 1 |
| 1.2 | Tools e codice sorgente | 1 |
| 2 | Analisi lessicale | 3 |
| 2.1 | Token riconosciuti | 3 |
| 2.2 | Esempio Lexer | 4 |
| 3 | Analisi sintattica | 7 |
| 3.1 | Struttura sintattica della grammatica | 7 |
| 3.1.1 | Struttura iniziale | 7 |
| 3.1.2 | Scope globale | 7 |
| 3.1.3 | Scope locale | 8 |
| 3.1.4 | Assegnamenti | 8 |
| 3.1.5 | Funzioni | 9 |
| 3.1.6 | Puntatori | 9 |
| 3.1.7 | Vettori | 10 |
| 3.1.8 | Espressioni matematiche | 10 |
| 3.1.9 | Statements | 11 |
| 3.1.10 | Condizioni | 12 |
| 4 | Analisi semantica | 14 |
| 4.1 | Semantica del linguaggio | 14 |
| 4.2 | Symbol tables | 14 |
| 4.3 | Debug della semantica | 15 |
| 5 | Gestione degli errori | 19 |
| 5.1 | Controlli | 19 |
| 5.2 | Errori individuati | 19 |
| 5.3 | Esempio di errori catturati | 20 |
| 5.3.1 | Errore sintattico | 20 |

| | | |
|----------|--|-----------|
| 5.3.2 | Errore sintattico e semantico in contemporanea nella stessa istruzione | 21 |
| 5.3.3 | Errore doppia dichiarazione | 22 |
| 5.3.4 | Errore uso variabile senza inizializzazione | 22 |
| 5.3.5 | Errore tipo ritornato dalla funzione diverso da quello dichiarato . . | 23 |
| 6 | Traduzione in Assembler MIPS | 24 |
| 6.1 | Operazioni tra variabili intere | 24 |
| 6.2 | Statements | 25 |
| 6.2.1 | If-ElseIf-Else | 26 |
| 6.2.2 | While | 28 |
| 6.2.3 | For | 29 |
| 6.3 | Chiamata di funzione | 30 |

Capitolo 1

Introduzione

1.1 Obiettivi

L'obiettivo del progetto è generare un compilatore semplificato per il linguaggio C che svolga rispettivamente i seguenti compiti:

1. **Analisi lessicale (lexer)**: il compilatore deve analizzare i token appartenenti al linguaggio e riuscire a individuare gli eventuali token errati non appartenenti all'alfabeto della grammatica.
2. **Analisi sintattica (parser)**: il compilatore deve analizzare la struttura sintattica del linguaggio e individuare se corrisponde a una sequenza corretta o meno.
3. **Analisi semantica**: ricavare il significato associato alla struttura sintattica e verificare che le regole di impiego del linguaggio siano soddisfatte.
4. **Gestione degli errori**: riconoscere gli errori e gestire gli stessi indicandoli in maniera esplicativa.
5. **Traduzione**: infine procedere nella traduzione del linguaggio C in un linguaggio di più basso livello quale l'Assembler MIPS (versione semplificata).

Il parser ha prospezione $LL(1)$, quindi con $k=1$.

1.2 Tools e codice sorgente

Per la generazione della grammatica del linguaggio C è stato utilizzato il programma **ANTLR**¹ acronimo di ANother Tool for Language Recognition, un generatore di parser

¹ANTLR (<https://www.antlr.org/>)

che fa uso del sistema di parsing LL. Per gestire la semantica e la traduzione sono stati scritti metodi in Java usando **IntelliJ**². Per il versioning si è usato **GitHub**³.

²JetBrains IntelliJ (<https://www.jetbrains.com/idea/>)

³GitHub (<https://github.com/>)

Capitolo 2

Analisi lessicale

In questa sezione trattiamo l'analisi lessicale.

2.1 Token riconosciuti

Di seguito elenco tutti i token riconosciuti da linguaggio ovvero tutti i simboli appartenenti al linguaggio e riconosciuti dal parser:

| TOKEN | CODICE | TOKEN | CODICE |
|---------------|--------|---------|--------|
| EOF | 1 | K_CHAR | 28 |
| ADD | 4 | K_FLOAT | 29 |
| AMP | 5 | K_INT | 30 |
| ARROW | 6 | LBRACK | 31 |
| ASS | 7 | LCURL | 32 |
| BACKSLASH | 8 | LE | 33 |
| CHAR | 9 | LPAREN | 34 |
| CHAR_QUOTE | 10 | LT | 35 |
| COMMA | 11 | MULT | 36 |
| COMMENT | 12 | NEQ | 37 |
| DIGIT | 13 | NEWL | 38 |
| DIGIT_NO_ZERO | 14 | OR | 39 |
| COMMENT | 12 | PERC | 40 |
| DIV | 15 | RBRACK | 41 |
| DOT | 16 | RCURL | 42 |
| D_QUOTE | 17 | RETURN | 43 |

| TOKEN | CODICE | TOKEN | CODICE |
|---------|--------|-------------|--------|
| ELSE | 18 | RPAREN | 44 |
| EQ | 19 | SEMICOL | 45 |
| FLOAT | 20 | SLASHR | 46 |
| FOR | 21 | SPACE | 47 |
| GE | 22 | SUB | 48 |
| GT | 23 | S_QUOTE | 49 |
| HASHTAG | 24 | TAB | 50 |
| IF | 25 | TOKEN_ERROR | 51 |
| INCLUDE | 26 | UNDRSCR | 52 |
| INT | 27 | VOID | 53 |
| WHILE | 54 | WORD | 55 |
| WS | 56 | | |

Se un token non rientra in una di queste 56 categorie verrà riconosciuto un errore lessicale in quanto è stato usato un token non appartenente al linguaggio.

2.2 Esempio Lexer

Usando il lexer per analizzare i token contenenti un codice di input d'esempio e si può notare che tutti i token corretti rientrano in una categoria delle 56 precedentemente elencate. I token non appartenenti al linguaggio rientrano nella categoria del token 51 che raccoglie tutti gli errori.

Qui di seguito abbiamo riportato l'esempio e il controesempio, in particolare per evitare di dilungarci troppo abbiamo riportato solo la parte iniziale. Per effettuare il controesempio abbiamo supposto che i caratteri ? e @ non esistessero nella grammatica.

```

1 ---Input---
2 #include<stdio.h>
3 #include<stdlib.h>
4
5 int a;
6 float b = 2.2;
7 char c;
8 ...
9
10 ---Test ANTLR lexer---
11 Token 1: (1,0) TokenType:26: #include
12 Token 2: (1,9) TokenType:35: <
13 Token 3: (1,10) TokenType:55: stdio

```

```

14 Token 4: (1,15) TokenType:16: .
15 Token 5: (1,16) TokenType:55: h
16 Token 6: (1,17) TokenType:23: >
17 Token 7: (2,0) TokenType:26: #include
18 Token 8: (2,9) TokenType:35: <
19 Token 9: (2,10) TokenType:55: stdlib
20 Token 10: (2,16) TokenType:16: .
21 Token 11: (2,17) TokenType:55: h
22 Token 12: (2,18) TokenType:23: >
23 Token 13: (10,0) TokenType:30: int
24 Token 14: (10,4) TokenType:55: a
25 Token 15: (10,5) TokenType:45: ;
26 Token 16: (11,0) TokenType:29: float
27 Token 17: (11,6) TokenType:55: b
28 Token 18: (11,8) TokenType:7: =
29 Token 19: (11,10) TokenType:20: 2.2
30 Token 20: (11,13) TokenType:45: ;
31 Token 21: (12,0) TokenType:28: char
32 Token 22: (12,5) TokenType:55: c
33 Token 23: (12,6) TokenType:45: ;
34 ...

```

Listing 2.1: Esempio con token corretti categorizzati

```

1 ---Input---
2 #include<stdio.h>
3 #include<stdlib.h>
4
5 @
6 func(){?}
7 ...
8
9 ---Test ANTLR lexer---
10 Token 1: (1,0) TokenType:26: #include
11 Token 2: (1,9) TokenType:35: <
12 Token 3: (1,10) TokenType:55: stdio
13 Token 4: (1,15) TokenType:16: .
14 Token 5: (1,16) TokenType:55: h
15 Token 6: (1,17) TokenType:23: >
16 Token 7: (2,0) TokenType:26: #include
17 Token 8: (2,9) TokenType:35: <
18 Token 9: (2,10) TokenType:55: stdlib
19 Token 10: (2,16) TokenType:16: .
20 Token 11: (2,17) TokenType:55: h
21 Token 12: (2,18) TokenType:23: >
22 Token 13: (4,0) TokenType:51: @ # Errore
23 Token 14: (6,4) TokenType:55: funct

```



```

24 Token 15: (6,9) TokenType:34: (
25 Token 16: (6,10) TokenType:44: )
26 Token 17: (6,11) TokenType:32: {
27 Token 18: (7,4) TokenType:51: ?          # Errore
28 Token 19: (8,0) TokenType:42: }
29 ...
30 *****
31 ***** Lexing completato con 2 errori *****
32 *****

```

Listing 2.2: Controesempio con token non appartenenti al linguaggio

Capitolo 3

Analisi sintattica

In questa sezione trattiamo l'analisi sintattica.

3.1 Struttura sintattica della grammatica

In questa sezione verranno illustrati tutti i diagrammi forniti in output da ANTLR per ogni regola sintattica prodotta; sono raggruppate per contesto simile.

3.1.1 Struttura iniziale

La grammatica presenta la seguente struttura sintattica iniziale con la presenza di librerie oppure si passa alla parte globale.

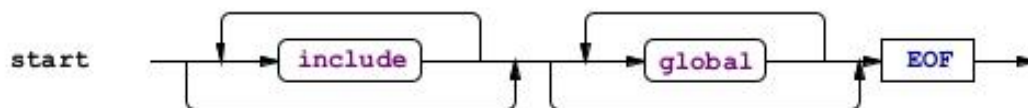


Figura 3.1: Struttura principale della grammatica



Figura 3.2: Struttura degli include

3.1.2 Scope globale

Lo scope globale della sintassi che consente di definire funzioni e variabili con visibilità globale.

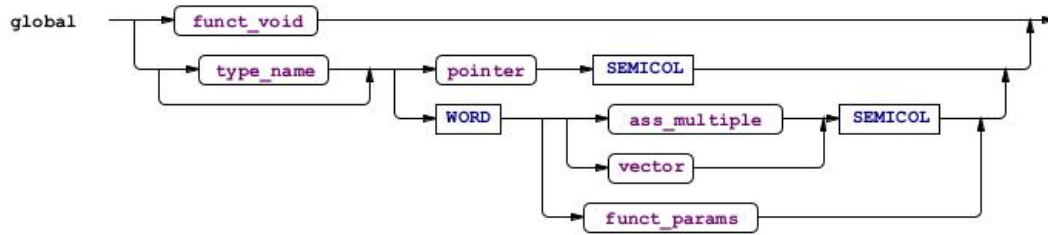


Figura 3.3: Struttura delle funzioni e delle variabili globali

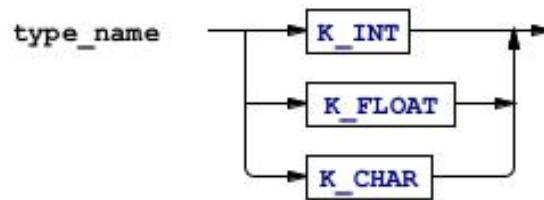


Figura 3.4: I diversi tipi delle variabili

3.1.3 Scope locale

Lo scope locale della sintassi che consente di definire funzioni e variabili con visibilità locale all'interno di ogni singola funzione.

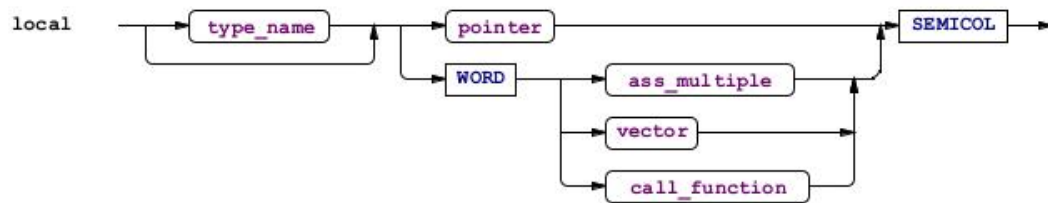


Figura 3.5: Struttura dello scope locale

3.1.4 Assegnamenti

Struttura sintattica che gestisce gli assegnamenti (anche multipli) per le variabili (sia singoli identificatori che vettori).

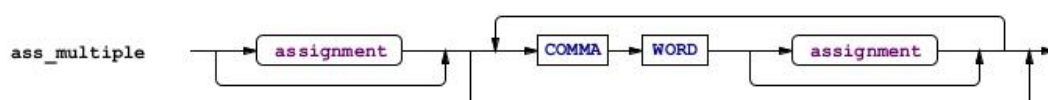


Figura 3.6: Struttura degli assegnamenti multipli

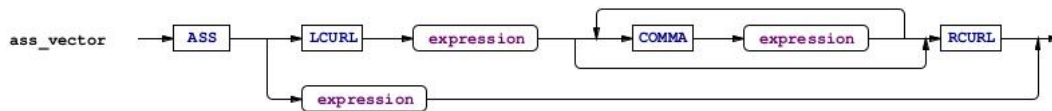


Figura 3.7: Struttura degli assegnamenti nei vettori

3.1.5 Funzioni

Struttura sintattica della funzione per gestire sia le chiamate di funzione che la definizione della stessa.

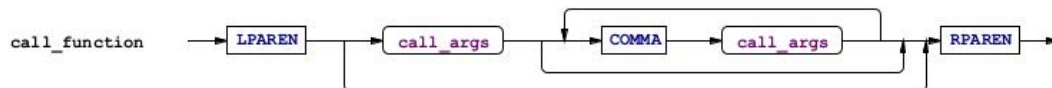


Figura 3.8: Struttura della chiamata di funzione

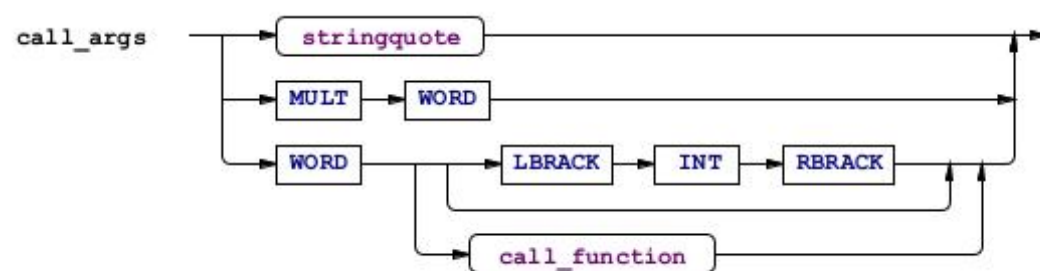


Figura 3.9: Struttura degli argomenti funzioni nella chiamata di funzione



Figura 3.10: Struttura di una funzione void

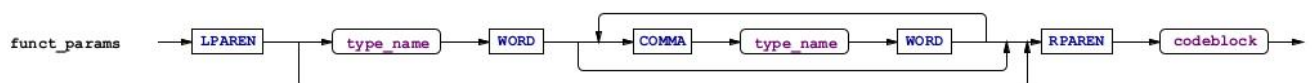


Figura 3.11: Struttura di una funzione

3.1.6 Puntatori

Struttura sintattica dei puntatori con eventuale inizializzazione seguente alla creazione.

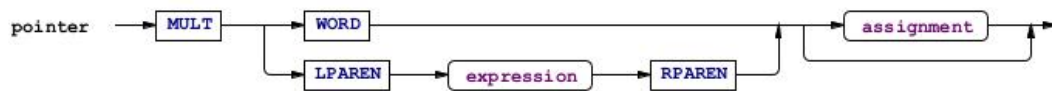


Figura 3.12: Struttura dei puntatori

3.1.7 Vettori

Struttura sintattica dei vettori con eventuale inizializzazione multipla delle posizioni del vettore.

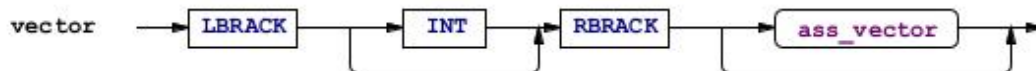


Figura 3.13: Struttura dei vettori

3.1.8 Espressioni matematiche

Struttura delle espressioni matematiche che rispetti l'ordine degli operatori matematici dando nell'ordine priorità alla moltiplicazione e divisione, seguite da somma e sottrazione.

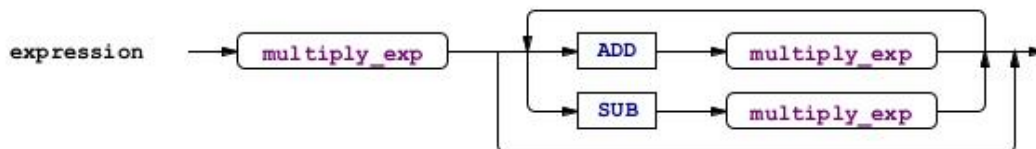


Figura 3.14: Struttura delle espressioni matematiche

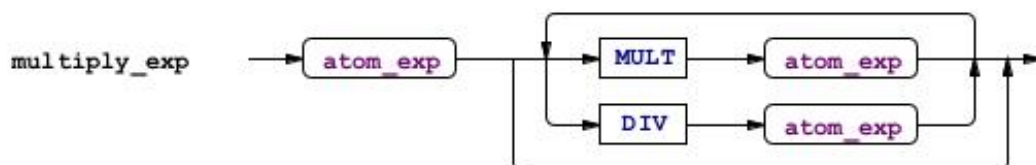


Figura 3.15: Struttura delle sotto-espressioni matematiche

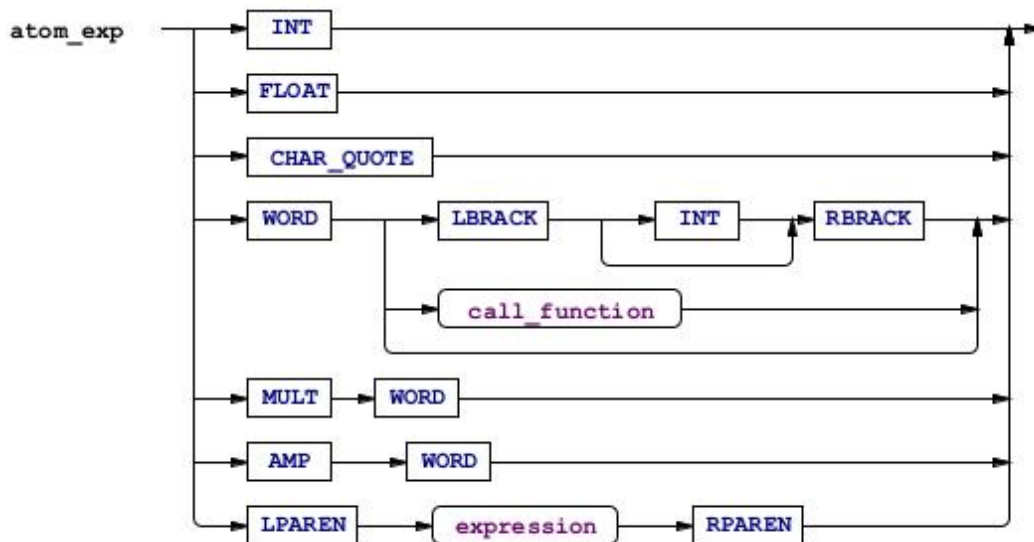


Figura 3.16: Tipologia dei singoli elementi atomici presenti in un'espressione

3.1.9 Statements

Struttura sintattica degli elementi che si possono utilizzare localmente all'interno di una funzione. Tra gli altri si possono inserire le condizioni (If-ElseIf-Else), i cicli (While, For), il return per restituire un valore e dei blocchi di codice (codeblock).

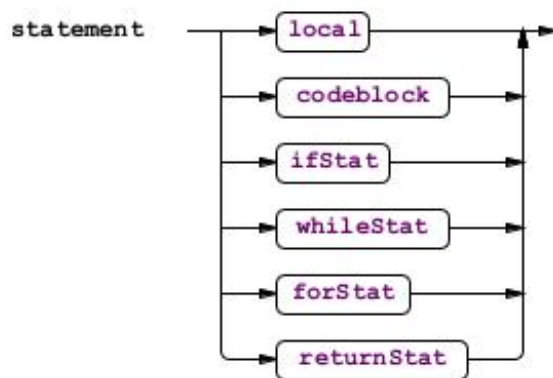


Figura 3.17: Struttura delle variabili e delle funzioni locali

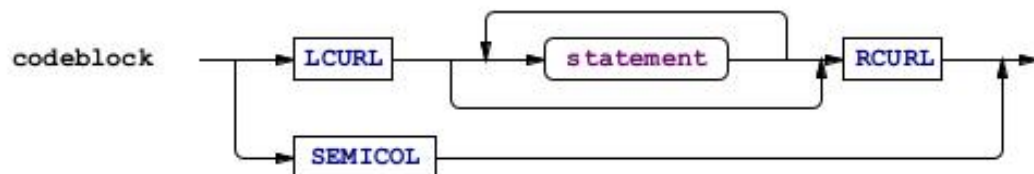


Figura 3.18: Possibili strutture di codice all'interno di un codeblock

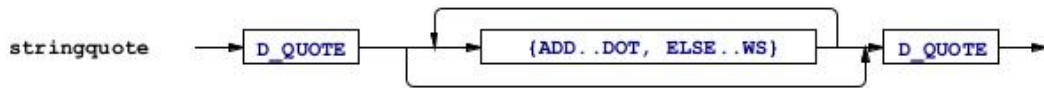


Figura 3.19: Struttura delle stringhe tra le virgolette

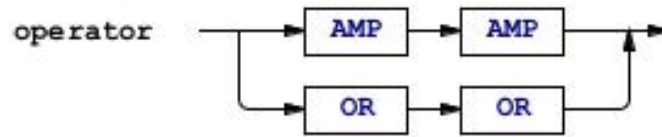


Figura 3.20: Struttura degli operatori logici

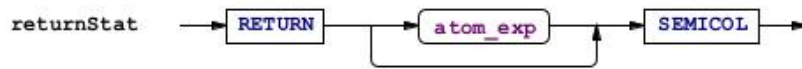


Figura 3.21: Struttura del costrutto return



Figura 3.22: Struttura del costrutto If-ElseIf-Else

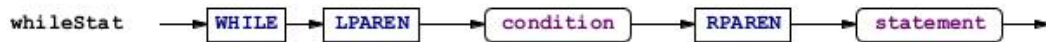


Figura 3.23: Struttura del costrutto While



Figura 3.24: Struttura del costrutto For

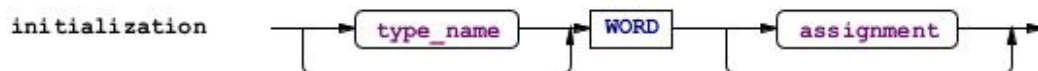


Figura 3.25: Struttura dell'inizializzazione usata negli statements



Figura 3.26: Struttura dell'incremento di variabile usato nei For

3.1.10 Condizioni

Struttura sintattica delle condizioni presenti nei costrutti If-ElseIf-Else, While e For.



Figura 3.27: Diversi tipologie di comparatori logici

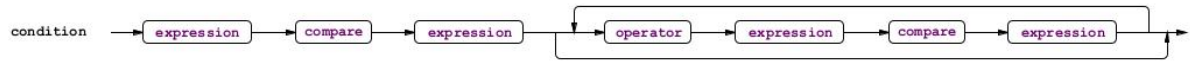


Figura 3.28: Struttura delle condizioni

Capitolo 4

Analisi semantica

In questa sezione trattiamo l'analisi semantica.

4.1 Semantica del linguaggio

Oltre alle symbol tables descritte nella prossima sezione, sono state gestite le operazioni matematiche tra espressioni (costanti, variabili, vettori, puntatori), assegnamenti, statements e numerosi altri controlli.

Inoltre è stata usata una classe **Value** per la creazione e la gestione delle variabili nella symbol table.

4.2 Symbol tables

La **symbol table** (tabella dei simboli) è una tabella che mantiene le corrispondenze tra i nomi delle etichette e gli indirizzi delle parole di memoria (usati successivamente anche nella traduzione). In questo modo ogni variabile creata viene salvata nella symbol table con il relativo valore nel caso sia inizializzata, con il proprio address di memoria e eventuali campi aggiuntivi che la caratterizzano. In particolare abbiamo utilizzato due tipologie di symbol table: una *symbol table globale* per tenere traccia di tutte le variabili globali e delle funzioni, e una *symbol table locale* per ogni funzione con le proprie variabili locali (che possono oscurare variabili globali avente stesso nome).

La symbol table permette di effettuare le seguenti funzioni:

- Distinguere le variabili, i puntatori, i vettori (con i relativi valori salvati) e le funzioni (ognuna delle quali avrà una symbol table locale) attraverso campi specifici.
- La separazione tra local e global symbol table consente di tenere separati i diversi scope delle variabili usate.

- Tenere collegati il nome, il tipo, il valore e l'indirizzo di memoria di una variabile.
- Recuperare il valore, il tipo o l'indirizzo di una variabile creata e inizializzata in precedenza.
- In caso di assegnamento, aggiornare il valore della variabile/vettore o l'indirizzo nel caso di modifica di un puntatore.
- Gestire i valori di return delle funzioni.

Grazie alla symbol table sono possibili diversi controlli semantici; in caso di violazioni, il tutto sarà catturato dalla gestione degli errori (vedi [Capitolo 5](#)).

La **symbol table globale** ha la seguente struttura:

| NAME | TYPE | VALUE | ADDRESS | isVAR | VECT |
|------|------|-------|---------|-------|--------|
| b | int | 5 | 3996 | True | X |
| main | int | null | 3992 | False | X |
| c | int | null | 3888 | True | 2,10,7 |

Il seguente esempio rappresenta il funzionamento della symbol table globale e contiene rispettivamente: nella prima riga la dichiarazione di una variabile intera globale, la seconda la dichiarazione della funzione *main* e infine la dichiarazione di un vettore di interi globale. Infatti ogni funzione creata (compreso il *main*) sarà presente nella symbol table globale.

La **symbol table locale** ha la seguente struttura:

| NAME | TYPE | VALUE | ADDRESS | isVAR | VECT |
|------|------|-------|---------|-------|--------|
| f | int | 5 | 3996 | True | X |
| d | int | null | 3888 | True | 2,10,7 |

Il seguente esempio rappresenta il funzionamento della symbol table locale e contiene rispettivamente: nella prima riga la dichiarazione di una variabile intera locale ad una funzione, la seconda riga la dichiarazione di un vettore di interi locale.

4.3 Debug della semantica

Si è effettuato il debug della semantica, in particolare ogni symbol table locale di ogni funzione, prima di essere eliminata veniva stampati in output. Infine anche la symbol table locale veniva stampata con gli ultimi valori aggiornati.

File di input: `trad_func.t`.

Il contenuto è presente anche nel seguente listato:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 2;
5 int b = 3;
6
7 void funct1() {
8     int d = 3;
9
10    if (d == 2) {
11        a = 3;
12    } else {
13        a = 4;
14    }
15 }
16
17 void funct2() {
18     int e = 3;
19
20     while (e < 5) {
21         e += 2;
22     }
23 }
24
25 int main() {
26     int c = 4;
27     a = 5;
28
29     funct1();
30
31     return 0;
32 }

```

Listing 4.1: File di input per il debug della semantica

File di output: FILE_DEBUG.

In quel file e nella console vengono stampate tutte le symbol tables, salvate con gli ultimi valori rimasti.

```

1 -----
2 ***** Local Symbol Table: funct1 *****
3 -----
4 1:  d=Value{name='d', type='int', value='3', address='3988', isVar=true,
      isVarPassed=false, isVect=false, vect=[]}
5
6 -----

```

```

7 ***** Local Symbol Table: funct2 *****
8 -----
9 1:  e=Value{name='e', type='int', value='5', address='3980', isVar=true,
      isVarPassed=false, isVect=false, vect=[]}
10
11 -----
12 ***** Local Symbol Table: main *****
13 -----
14 1:  c=Value{name='c', type='int', value='4', address='3972', isVar=true,
      isVarPassed=false, isVect=false, vect=[]}
15
16 -----
17 ***** Global Symbol Table *****
18 -----
19 1:  b=Value{name='b', type='int', value='3', address='3996', isVar=true,
      isVarPassed=false, isVect=false, vect=[]}
20 2:  a=Value{name='a', type='int', value='5', address='4000', isVar=true,
      isVarPassed=false, isVect=false, vect=[]}
21 3:  main=Value{name='main', type='int', value='0', address='3976', isVar=
      false, isVarPassed=false, isVect=false, vect=[]}
22 4:  funct2=Value{name='funct2', type='void', value='null', address='3984',
      isVar=false, isVarPassed=false, isVect=false, vect=[]}
23 5:  funct1=Value{name='funct1', type='void', value='null', address='3992',
      isVar=false, isVarPassed=false, isVect=false, vect=[]}

```

Listing 4.2: Log delle symbol tables

Inoltre per aiutare la comprensione di ciò che veniva fatto man mano si è stampato delle linee di debug (discorsive) nella console di output. Qui di seguito è fornito un esempio:

```

1 -----
2 ***** Debug *****
3 -----
4 Ho dichiarato la variabile 'a' come 'int'
5 Ho assegnato alla variabile 'a' il valore '2'
6 Ho dichiarato la variabile 'b' come 'int'
7 Ho assegnato alla variabile 'b' il valore '3'
8 Ho dichiarato la funzione 'funct1' con tipo 'void'
9 Ho dichiarato la variabile locale 'd' come 'int'
10 Ho assegnato alla variabile 'd' il valore '3'
11 Ho assegnato alla variabile 'a' il valore '3'
12 Ho assegnato alla variabile 'a' il valore '4'
13 Ho dichiarato la funzione 'funct2' con tipo 'void'
14 Ho dichiarato la variabile locale 'e' come 'int'
15 Ho assegnato alla variabile 'e' il valore '3'
16 Ho assegnato alla variabile 'e' il valore '5'

```

```
17 Ho dichiarato la funzione 'main' con tipo 'int'
18 Ho dichiarato la variabile locale 'c' come 'int'
19 Ho assegnato alla variabile 'c' il valore '4'
20 Ho assegnato alla variabile 'a' il valore '5'
```

Listing 4.3: Log discorsivo della semantica

Capitolo 5

Gestione degli errori

In questa sezione trattiamo la gestione degli errori.

5.1 Controlli

Sono stati effettuati numerosi controlli, tra cui quelli relativi alla symbol table sono:

- Controllo sulla variabile creata che non fosse già stata creata in precedenza (in base allo scope).
- Controllo sulla variabile usata che sia stata creata e inizializzata in precedenza.
- Controllo sulla compatibilità di tipo tra variabile e valore.
- Controllo che sia presente uno e un solo main.
- Controllo sulle chiamate di funzione.
- Controllo sui valori ritornati dalle funzioni e le compatibilità tra i vari tipi.
- Controllo sugli operatori usati, se sono definiti tra i vari operandi e se entrambi gli operandi hanno valore diverso da null.
- Controllo su divisioni per zero.

La classe `ValueTypes` è stata di supporto per i controlli sulla compatibilità tra tipi (usando matrici di compatibilità tra i vari tipi possibili).

File di output: `FILE_ERRORI`.

5.2 Errori individuati

Sono gestite le seguenti tipologie di errore con il relativo output a console:

1. `ERR_ALREADY_DECLARED`: "La variabile X è già stata dichiarata";
2. `ERR_UNDECLARED`: "La variabile X non è stata dichiarata";
3. `ERR_FUNC_ALREADY_DECLARED`: "La funzione X è già stata dichiarata";
4. `ERR_FUNC_UNDECLARED`: "La funzione X non è stata dichiarata";
5. `ERR_TYPE_CALL_FUNCT`: "La chiamata di funzione X non deve avere il type";
6. `ERR_TYPE_FUNCT_RETURN`: "Il tipo ritornato non ha lo stesso type del tipo di funzione atteso";
7. `ERR_TYPE_CALL_FUNCT_RETURN`: "La chiamata di funzione non ha lo stesso type dell'espressione attesa";
8. `ERR_CALL_FUNC_IN_GLOBAL`: "La chiamata di funzione X non può essere fatta globalmente";
9. `ERR_MAIN_NOT_FOUND`: "La funzione `int main()` è richiesta ma non è stata trovata";
10. `ERR_MAIN_TYPE_NOT_INT`: "La funzione `int main()` è richiesta ma non è stata trovata";
11. `ERR_TYPE_MISMATCH`: "Valore di tipo non compatibile (null)";
12. `ERR_UNDEFINED_OP`: "L'operatore X non è definito per i due operandi";
13. `ERR_DIV_BY_0`: "Divisione per 0";
14. `ERR_NO_VALUE`: "La variabile X non è stata inizializzata";
15. `ERR_NO_VALUES`: "L'operazione X non può essere eseguita perché almeno uno dei due operandi non ha valore";
16. In caso non faccia match con alcun caso precedente: "Errore non definito sul token X ".

5.3 Esempio di errori catturati

Alcuni esempi di errori catturati (sia sintattici che semantici), riportiamo solo la parte del codice interessata.

5.3.1 Errore sintattico

File di input: `test_C_1` e decommentare riga 112.

Una porzione del file di input è mostrata nel seguente listing:

```

1 ...
2 int statements(int t) {
3 ...
4 # Parentesi graffa non chiusa

```

Listing 5.1: Porzione del file di input: `test_C_1` modificato

In questo caso il parser non ha trovato una parentesi graffa chiusa.

```

1 *****
2 ***** Parsing completato con 1 errori *****
3 *****
4
5 1: Errore sintattico [1] in (112, 3) - Found ('<EOF>') - mismatched input
   '<EOF>' expecting RCURL

```

Listing 5.2: Esempio di errore sintattico

5.3.2 Errore sintattico e semantico in contemporanea nella stessa istruzione

File di input: `test_C_1` e decommentare riga 13 e 112.

Una porzione del file di input è mostrata nel seguente listing:

```

1 int a;
2 int a = 1 # Doppia dichiarazione e punto virgola mancante
3 ...

```

Listing 5.3: Porzione del file di input: `test_C_1` modificato

In questo caso il parser ha trovato due errori nella stessa istruzione, sintattico e semantico.

```

1 *****
2 ***** Parsing completato con 2 errori *****
3 *****
4
5 1: Errore semantico [10] in (13,4) - La variabile <a> è già stata
   dichiarata
6 2: Errore sintattico [1] in (14, 0) - Found K_INT ('int') - missing SEMICOL
   at 'int'

```

Listing 5.4: Esempio di errore sintattico

5.3.3 Errore doppia dichiarazione

File di input: `test_C_1` e decommentare riga 49.

Una porzione del file di input è mostrata nel seguente listing:

```
1 ...
2 int statements(int t) {
3 int a;
4 int a = 1;
5 ...
6 }
```

Listing 5.5: Porzione del file di input: `test_C_1` modificato

```
1 *****
2 ***** Parsing completato con 1 errori *****
3 *****
4
5 1: Errore semantico [10] in (49,8) - La variabile <a> è già stata
   dichiarata
```

Listing 5.6: Esempio di errore per doppia dichiarazione

5.3.4 Errore uso variabile senza inizializzazione

File di input: `test_C_2` e commentare riga 24.

Una porzione del file di input è mostrata nel seguente listing:

```
1 int *p1;
2 // *p1 = 4; // Commentarla per catturare l'errore alla riga sotto
3 int e = 5 + *p1;
4 ...
```

Listing 5.7: Porzione del file di input: `test_C_2` modificato

Gli errori ulteriori sono dovuti alla propagazione dell'errore nelle righe di codice successive.

```
1 *****
2 ***** Parsing completato con 3 errori *****
3 *****
4
5 1: Errore semantico [12] in (25,13) - La variabile <p1> non è stata
   inizializzata
6 2: Errore semantico [13] in (25,10) - L'operazione <+> non può essere
   eseguita perché almeno uno dei due operandi non ha valore
```

```
7 3: Errore semantico [16] in (25,4) - Valore di tipo non compatibile (null)
```

Listing 5.8: Esempio di errore per uso di variabile senza inizializzazione

5.3.5 Errore tipo ritornato dalla funzione diverso da quello dichiarato

File di input: `test_C_2` e decommentare riga 51.

Una porzione del file di input è mostrata nel seguente listing:

```
1 int func_test(int j, char k) {  
2 ...  
3 return;  
4 }
```

Listing 5.9: Porzione del file di input: `test_C_2` modificato

```
1 *****  
2 ***** Parsing completato con 1 errori *****  
3 *****  
4  
5 1: Errore semantico [113] in (51,4) - Il tipo ritornato non ha lo stesso  
   type del tipo di funzione atteso
```

Listing 5.10: Esempio di errore per diverso tipo restituito dalla funzione

Capitolo 6

Traduzione in Assembler MIPS

Si è proceduti con la traduzione del codice in una versione semplificata dell'Assembler MIPS. In particolare si sono fatte le seguenti assunzioni:

- Ogni variabile e costante intera (int) occupa 4 Byte, quindi una word nella memoria del MIPS.
- La memoria parte da 0x4000 e va a decrescere man mano che vengono salvate variabili.
- Sono state scritte istruzioni semplificate e simboliche per gestire i valori ritornati (\$v0), il return address (\$ra)...

La classe `Translation.java` contiene i vari metodi che gestiscono la parte di traduzione ed è istanziata da `ParserEnvironment.java`.

Abbiamo fatto la traduzione della definizione di variabili intere, delle varie operazioni tra interi, degli statements (if-iffelse-else, while, for) e delle chiamate di funzioni.

File di output: `FILE_TRADUZIONI`.

6.1 Operazioni tra variabili intere

File di input: `trad_operations`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 2;
5 int b = 35;
6
7 int main() {
```

```

8     int c = 4;
9
10    b = (a + b) * c;
11
12    return 0;
13 }

```

Listing 6.1: File di input `trad_operations`

Le variabili globali sono state messe in memoria usando `.word`. Le operazioni matematiche sono state fatte da: `add`, `sub`, `mul`, `div`.

Quando viene chiamata la funzione viene creata un'area nello stack pointer per salvare il return address e il valore di uscita. Alla chiusura di una funzione si procederà con il ripristino del return address, l'eliminazione della memoria dello stack pointer precedentemente usata e si salta alla prossima istruzione definita dal return address.

```

1 A: .word 2
2 B: .word 35
3 MAIN:
4     addiu $sp, $sp, -8      #Crea area nello stack pointer
5     sw $ra, 4($sp)         #Salva return address
6     lw $t1, 4
7     sw $t1, 0x3988         #[c=4]
8     lw $t2, 0x4000         #[a=2]
9     lw $t1, 0x3996         #[b=35]
10    add $t2, $t1, $t2
11    lw $t1, 0x3988         #[c=4]
12    mul $t1, $t1, $t2
13    sw $t1, 0x3996         #[b=148]
14    sw 0, 0($sp)           #Memorizza il valore del return
15    lw $v0, 0($sp)         #Salva valore d'uscita
16    lw $ra, 4($sp)         #Ripristina return address
17    addiu $sp, $sp, 8      #Elimina area nello stack pointer
18    jr $ra                 #Salta al return address

```

Listing 6.2: Traduzioni delle operazioni matematiche

6.2 Statements

Sono state usate delle label per gestire i vari statement e i salti tra i vari rami in base alle condizioni presenti. In particolare in base al comparatore sono state usate una combinazione di istruzioni tra le seguenti:

- `beq $t1, $t2, LABEL`: se `$t1` è uguale a `$t2` si salta alla label.

- `bne $t1, $t2, LABEL`: se `$t1` è diverso da `$t2` si salta alla label.
- `slt $t8, $t1, $t2`: se `$t1` è minore di `$t2` si setta `$t8` a 1.

```

1 switch (comp.getText()) {
2     case "==":
3         emit("bne $t1, $t2, " + label);
4         break;
5     case "!=":
6         emit("beq $t1, $t2, " + label);
7         break;
8     case "<":
9         emit("slt $t8, $t1, $t2");
10        emit("bne $t8, 1, " + label);
11        break;
12    case ">":
13        emit("slt $t8, $t2, $t1"); // Registri invertiti
14        emit("bne $t8, 1, " + label);
15        break;
16    case "<=":
17        emit("slt $t8, $t2, $t1"); // Registri invertiti
18        emit("beq $t8, 1, " + label);
19        break;
20    case ">=":
21        emit("slt $t8, $t1, $t2");
22        emit("beq $t8, 1, " + label);
23        break;
24 }

```

Listing 6.3: Compare

6.2.1 If-ElseIf-Else

File di input: trad_if.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 2;
5 int b = 3;
6
7 int main() {
8     int c = 4;
9
10    if (a == 3) {
11        b += 5;
12    } else if (b == 3) {

```

```

13     c = 6;
14 } else {
15     c = 7;
16 }
17
18     return 0;
19 }

```

Listing 6.4: File di input trad_if

Di seguito la traduzione:

```

1 A: .word 2
2 B: .word 3
3 MAIN:
4     addiu $sp, $sp, -8      #Crea area nello stack pointer
5     sw $ra, 4($sp)         #Salva return address
6     lw $t1, 4
7     sw $t1, 0x3988         #[c=4]
8     IF1:
9         lw $t2, 0x4000      #[a=2]
10        lw $t1, 3
11        bne $t1, $t2, ELSE1
12        lw $t2, 0x3996      #[b=3]
13        lw $t1, 5
14        add $t2, $t1, $t2
15        sw $t2, 0x3996      #[b=8]
16        j ENDIF1
17    ELSE1:
18        IF2:
19            lw $t1, 0x3996    #[b=8]
20            lw $t2, 3
21            bne $t1, $t2, ELSE2
22            lw $t1, 6
23            sw $t1, 0x3988    #[c=6]
24            j ENDIF2
25        ELSE2:
26            lw $t2, 7
27            sw $t2, 0x3988    #[c=7]
28        ENDIF2:
29    ENDIF1:
30    sw 0, 0($sp)            #Memorizza il valore del return
31    lw $v0, 0($sp)         #Salva valore d'uscita
32    lw $ra, 4($sp)         #Ripristina return address
33    addiu $sp, $sp, 8      #Elimina area nello stack pointer
34    jr $ra                 #Salta al return address

```

Listing 6.5: Traduzioni dell'if-elseif-else statement

6.2.2 While

File di input: trad_while.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 2;
5 int b = 3;
6
7 int main() {
8     while (b < 10) {
9         b += a;
10    }
11
12    return 0;
13 }
```

Listing 6.6: File di input trad_while

Di seguito la traduzione:

```
1 A: .word 2
2 B: .word 3
3 MAIN:
4     addiu $sp, $sp, -8    #Crea area nello stack pointer
5     sw $ra, 4($sp)       #Salva return address
6     WHILE1:
7         lw $t1, 0x3996    #[b=3]
8         lw $t2, 10
9         slt $t8, $t1, $t2
10        bne $t8, 1, ENDWHILE1
11        lw $t1, 0x3996    #[b=3]
12        lw $t2, 0x4000    #[a=2]
13        add $t1, $t1, $t2
14        sw $t1, 0x3996    #[b=5]
15        j WHILE1
16    ENDWHILE1:
17        sw 0, 0($sp)      #Memorizza il valore del return
18        lw $v0, 0($sp)    #Salva valore d'uscita
19        lw $ra, 4($sp)    #Ripristina return address
20        addiu $sp, $sp, 8  #Elimina area nello stack pointer
21        jr $ra            #Salta al return address
```

Listing 6.7: Traduzioni del while statement

6.2.3 For

File di input: trad_for.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 2;
5 int b = 3;
6
7 int main() {
8     int c = 4;
9
10    for (b = 2; b < 5; b += 2) {
11        c += 4;
12    }
13
14    return 0;
15 }
```

Listing 6.8: File di input trad_for

Di seguito la traduzione (notare la label per gli incrementi):

```
1 A: .word 2
2 B: .word 3
3 MAIN:
4     addiu $sp, $sp, -8      #Crea area nello stack pointer
5     sw $ra, 4($sp)         #Salva return address
6     lw $t1, 4
7     sw $t1, 0x3988         #[c=4]
8     FOR1:
9         lw $t2, 2
10        sw $t2, 0x3996       #[b=2]
11        lw $t1, 0x3996       #[b=2]
12        lw $t2, 5
13        slt $t8, $t1, $t2
14        bne $t8, 1, ENDFOR1
15        j ENDINCRFOR1
16    INCRFOR1:
17        lw $t1, 0x3996       #[b=2]
18        lw $t2, 2
19        add $t1, $t1, $t2
20        sw $t1, 0x3996       #[b=4]
21        j FOR1
22    ENDFOR1:
23    lw $t2, 0x3988          #[c=4]
```



```

24     lw $t1, 4
25     add $t2, $t1, $t2
26     sw $t2, 0x3988      #[c=8]
27     j INCRFOR1
28 ENDFOR1:
29     sw 0, 0($sp)        #Memorizza il valore del return
30     lw $v0, 0($sp)      #Salva valore d'uscita
31     lw $ra, 4($sp)      #Ripristina return address
32     addiu $sp, $sp, 8    #Elimina area nello stack pointer
33     jr $ra              #Salta al return address

```

Listing 6.9: Traduzioni del for statement

6.3 Chiamata di funzione

File di input: trad_funcnt.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a = 2;
5 int b = 3;
6
7 void funct1() {
8     int d = 3;
9
10    if (d == 2) {
11        a = 3;
12    } else {
13        a = 4;
14    }
15 }
16
17 void funct2() {
18     int e = 3;
19
20     while (e < 5) {
21         e += 2;
22     }
23 }
24
25 int main() {
26     int c = 4;
27     a = 5;
28
29     funct1();

```

```

30
31     return 0;
32 }

```

Listing 6.10: File di input trad_func.t

Di seguito la traduzione:

```

1 A: .word 2
2 B: .word 3
3 FUNCT1:
4     addiu $sp, $sp, -8      #Crea area nello stack pointer
5     sw $ra, 4($sp)         #Salva return address
6     lw $t1, 3
7     sw $t1, 0x3988         #[d=3]
8     IF1:
9         lw $t2, 0x3988     #[d=3]
10        lw $t1, 2
11        bne $t1, $t2, ELSE1
12        lw $t2, 3
13        sw $t2, 0x4000     #[a=3]
14        j ENDIF1
15    ELSE1:
16        lw $t1, 4
17        sw $t1, 0x4000     #[a=4]
18    ENDIF1:
19    sw 0, 0($sp)           #Memorizza il valore del return
20    lw $v0, 0($sp)         #Salva valore d'uscita
21    lw $ra, 4($sp)         #Ripristina return address
22    addiu $sp, $sp, 8      #Elimina area nello stack pointer
23    jr $ra                 #Salta al return address
24 FUNCT2:
25    addiu $sp, $sp, -8      #Crea area nello stack pointer
26    sw $ra, 4($sp)         #Salva return address
27    lw $t2, 3
28    sw $t2, 0x3980         #[e=3]
29    WHILE1:
30        lw $t1, 0x3980     #[e=3]
31        lw $t2, 5
32        slt $t8, $t1, $t2
33        bne $t8, 1, ENDWHILE1
34        lw $t1, 0x3980     #[e=3]
35        lw $t2, 2
36        add $t1, $t1, $t2
37        sw $t1, 0x3980     #[e=5]
38        j WHILE1
39    ENDWHILE1:

```

```

40  sw 0, 0($sp)      #Memorizza il valore del return
41  lw $v0, 0($sp)    #Salva valore d'uscita
42  lw $ra, 4($sp)     #Ripristina return address
43  addiu $sp, $sp, 8  #Elimina area nello stack pointer
44  jr $ra             #Salta al return address
45 MAIN:
46  addiu $sp, $sp, -8  #Crea area nello stack pointer
47  sw $ra, 4($sp)     #Salva return address
48  lw $t2, 4
49  sw $t2, 0x3972     #[c=4]
50  lw $t1, 5
51  sw $t1, 0x4000     #[a=5]
52  addiu $sp, $sp, -4  #Crea area nello stack pointer (per la chiamata di
    funzione)
53  sw $ra, 0($sp)     #Salva return address (per la chiamata di funzione)
54  j FUNCT1           #Chiamata di funzione
55  lw $ra, 0($sp)     #Ripristina return address (dopo la chiamata di
    funzione)
56  addiu $sp, $sp, 4  #Elimina area nello stack pointer (dopo la chiamata
    di funzione)
57  sw 0, 0($sp)      #Memorizza il valore del return
58  lw $v0, 0($sp)    #Salva valore d'uscita
59  lw $ra, 4($sp)     #Ripristina return address
60  addiu $sp, $sp, 8  #Elimina area nello stack pointer
61  jr $ra             #Salta al return address

```

Listing 6.11: Traduzioni delle chiamate di funzioni