



**POLITECNICO DI BARI DIPARTIMENTO DI INGEGNERIA
ELETTRICA E DELL'INFORMAZIONE**

FORMAL LANGUAGES AND COMPILERS

C to Python Transpiler

A.A.	2022/2023
Docente	Floriano Scioscia
Studenti	Mauro Giordano – Francesco Palumbo

1. INTRODUZIONE

Lo scopo di questo progetto è quello di realizzare un transpiler la cui funzione è quella di ricevere in input un subset di istruzioni scritte in linguaggio C e tradurlo nell'equivalente set di istruzioni in linguaggio Python.

Tale progetto è stato realizzato in linguaggio C mediante l'ausilio del generatore di scanner automatico Flex e del generatore automatico di parser Bison, unitamente a uthash, una libreria esterna per la gestione della symbol table.

1.1 Restrizione sul linguaggio sorgente

Poiché il design e l'implementazione di un compilatore per un intero linguaggio di programmazione costituisce un lavoro di una certa entità, il nostro progetto opera su un sottoinsieme del linguaggio C di partenza, concordato con il professore ad inizio progetto. In particolare sono state previste le seguenti istruzioni e funzionalità:

- **Tipi di dato:** valori numerici interi con segno (int), valori in virgola mobile a singola precisione (float), caratteri di tipo alfanumerico (char), tipi di dato relativi a funzioni che non restituiscono alcun dato al loro chiamante (void).
- **Operatori aritmetici:** somma (+), sottrazione (-), moltiplicazione (*), divisione (/)
- **Operatori logici:** and, or, not
- **Operatori di confronto:** <, <=, >, >=
- **Espressioni:** fanno uso degli operatori logici, aritmetici e di confronto. Prevedono la possibilità di utilizzare variabili, elementi di vettori e chiamate a funzioni.
- **Istruzioni condizionali:** If, Else, Else if, While
- **Istruzioni per I/O:** printf, scanf
- **Dichiarazione e assegnazione di variabili:** è prevista la possibilità di effettuare, oltre alle normali assegnazioni di valori, anche assegnazioni complesse.
- **Dichiarazione e assegnazione di vettori**
- **Dichiarazione e chiamata di funzioni:** con parametri locali e senza parametri. In entrambi i casi viene gestito lo scope.
- **Commenti:** singola e multi linea

1.2 Struttura del progetto

Il progetto si compone di diversi file, elencati di seguito:

- **lexer.l** file che contiene le specifiche per la generazione dello scanner mediante il tool Flex.

- **parser.y** file che contiene le specifiche per la generazione del parser mediante il tool Bison. Qui viene definita la grammatica del linguaggio C, le istruzioni per la gestione di tutti i controlli semantici con relativi errori e le istruzioni per la gestione degli scope.
- **ast.h** file contenente la struttura di tutti i nodi che compongono l'Abstract Syntax Tree, insieme ad altre strutture per la gestione del tipo di simbolo e di informazioni necessarie alla traduzione.
- **symboltable.h** file contenente la struttura della symbol table, una struttura necessaria a rappresentare tutte le informazioni relative ai vari simboli e tutte le funzioni di gestione relative agli stessi (aggiunta, ricerca e rimozione simboli). Contiene, inoltre, una struttura del tipo linked list, necessaria alla gestione degli scope.
- **uthash.h** libreria necessaria ad implementare una hash table in C. Viene utilizzata per supportare la gestione della symbol table.
- **c2py.h** file che permette la traduzione nel linguaggio destinazione. Contiene le procedure necessarie all'attraversamento dell'Abstract Syntax Tree.

2. DESCRIZIONE DEL PROGETTO

In questa fase verranno ripercorse brevemente le fasi di un compilatore, ponendo particolare attenzione al progetto in essere.

2.1 Analisi Lessicale

È la prima fase di un compilatore nota anche come scanning. Qui, viene analizzato il testo in input e viene prodotta una sequenza di token per ciascun lessema riconosciuto.

Per generare lo scanner è stato utilizzato il tool Flex (Fast Lexical Analyzer generator). Questo prende in input il file *lexer.l* in cui sono definite le espressioni regolari che devono essere riconosciute.

Nel file creato, vengono innanzitutto definite le espressioni regolari che devono essere riconosciute in input. Vengono poi dichiarati gli stati custom relativi all'analisi lessicale. Infine vengono definiti i token da riconoscere in fase di scanning insieme al rispettivo codice da eseguire al momento del riconoscimento.

2.1.1 Stati aggiuntivi

Il funzionamento di Flex è assimilabile a quello di una macchina a stati finiti, al cui stato di default INITIAL, è possibile aggiungere ulteriori stati custom.

In aggiunta allo stato di default sono stati definiti tre stati custom:

MLCOMMENT: permette il riconoscimento dei commenti multi linea

SLCOMMENT: permette il riconoscimento dei commenti a singola linea
LIBRARY: permette il riconoscimento delle direttive di tipo #include

```
"/**"                   BEGIN(MLCOMMENT);   printf("found MULTI-LINE comment\n");  
<MLCOMMENT>[^*\n]*  
<MLCOMMENT>"**"+[^\n]*  
<MLCOMMENT>\n  
<MLCOMMENT>"**"+"/"       BEGIN(INITIAL);   printf("end of MULTI-LINE comment\n");  
  
"//"  
<SLCOMMENT>[^*\n]*  
<SLCOMMENT>\n                   BEGIN(INITIAL);   printf("end of SINGLE-LINE comment\n");  
  
"#include"  
<LIBRARY>^.*  
<LIBRARY>[^\n]*  
<LIBRARY>\n                   BEGIN(INITIAL);   printf("end of #include<library>\n");
```

lexer.l stati per il riconoscimento di particolari lessemi

In tutti e tre i casi, i lessemi riconosciuti, non sono utili al fine di generare codice correttamente eseguibile in Python. Di conseguenza, tali elementi, vengono ignorati in fase di traduzione.

2.2 Analisi Sintattica

Analisi lessicale e sintattica avvengono in maniera congiunta. Nello specifico il lexer analizza l'input dell'utente e, in caso di esito positivo, restituisce al parser i token nello stesso ordine in cui essi sono stati riconosciuti.

Lo scopo del parser è quello di stabilire se la sequenza di token ricevuta in ingresso rappresenta una frase ammessa dalla sintassi del linguaggio. Nel nostro caso, abbiamo deciso di avvalerci del parser generator Bison.

Dopo la dichiarazione di alcuni elementi di supporto al funzionamento del parser, sono state definite le dichiarazioni (%token, %left, %right) utilizzate per definire i token e specificare l'associazione e la precedenza degli operatori nel linguaggio di programmazione sorgente, in modo che il parser sappia come interpretare correttamente le espressioni matematiche o logiche durante l'analisi sintattica. Questa parte è stata perfezionata in seguito alla scoperta di un errore descritto nella sezione 4.5.

È stato definito un type "union" per conservare i valori associati ai nodi non-terminal riconosciuti dallo scanner. Nel caso in esame, la union è stata chiamata "yystype".

Sono poi stati definiti i tipi di dato relativi alle varie classi definite in seguito. Questo consente di definire una struttura di dati per conservare i valori dei token riconosciuti dallo scanner e di specificare la relazione tra i diversi tipi di dati utilizzati nel parser.

2.2.1 Produzione Iniziale

Il primo non-terminal "program" è stato definito per gestire l'analisi dell'intero programma in input.

Questo, per mezzo della funzione `beginScope()`, permette di definire uno scope globale contenente le produzioni per la gestione di una o più "instruction".

A sua volta, il non-terminal "program", contiene la produzione "statements", che viene impostata come nodo dell'abstract syntax tree e che contiene al suo interno i vari tipi di "instruction".

Se la lettura di tutti gli statements termina correttamente, viene invocata la funzione `endScope()`, la quale permette di uscire dallo scope attuale, eliminando la symbol table corrente. La gestione degli scope verrà approfondita nella sezione 2.4.3.

2.2.2 Produzioni principali

Durante l'individuazione di una nuova tipologia di istruzione, viene eseguito il seguente processo: ogni volta che viene riconosciuto un nuovo simbolo, questo viene aggiunto alla symbol table utilizzando la funzione "createSym()". In aggiunta, per le funzioni e le inizializzazioni, viene eseguito un controllo sulla symbol table per verificare se la variabile è già stata dichiarata, utilizzando la funzione "findSym()" e "findSymtab()".

In generale, la sequenza di operazioni attuate nel momento in cui la sequenza di token identificata da ciascun non-terminal viene riconosciuta correttamente è la seguente: prima viene allocata la struttura dati relativa al nodo corrispondente e successivamente vengono effettuate le assegnazioni agli elementi della struttura in modo da riempire lo spazio allocato.

Nel caso di nuove inizializzazioni (variabili, funzioni e array), l'aggiunta alla symbol table avviene facendo riferimento proprio alle informazioni appena inserite nella struttura dati.

2.2.3 Ulteriori produzioni

Tutte le produzioni presenti nel linguaggio sono racchiuse nel non-terminal "instruction". Da qui partono le diramazioni per i casi specifici. Vi sono, poi, delle produzioni che non hanno un diretto corrispettivo nel linguaggio di partenza, ma svolgono un ruolo di supporto per il corretto riconoscimento:

- **arrayDecl** e **arrayCall**, relativamente alle produzioni principali **arrayInit** e **arrayAssign**, consentono di gestire i diversi tipi di inizializzazione e assegnazione degli array. In particolare, entrambi permettono di distinguere la casistica in cui è specificato il numero di elementi del vettore (`array[3]`) da quella in cui non è specificato (`array[]`). **arrayDecl** contiene informazioni sulla lunghezza dell'array mentre **arrayCall** contiene informazioni sull'indice dell'elemento chiamato.
- **outputElements** e **inputElements**, relativamente alle funzioni principali **outputFunction** e **inputFunction**, consentono di distinguere la casistica in cui, le funzioni `printf` e `scanf` del linguaggio di partenza, presentino uno o più argomenti.
- Il non-terminal **content**, oltre ad avere le produzioni relative ai vari tipi di dato (`int`, `float`, `char`), contiene al suo interno le produzioni relative a **id**, **expression**, **functionCall**, **arrayCall**. Questo permette di gestire la casistica in cui ad una variabile viene assegnato, oltre ad un valore diretto, anche un id, un'espressione, una funzione o un elemento di un array.

```

arrayDecl:
ID LSBRA RSBRA
| ID LSBRA INT_VALUE RSBRA
arrayCall:
ID LSBRA RSBRA
| ID LSBRA content RSBRA

```

```

{
    $$=malloc(sizeof(struct AstNodeArrayDecl));
    printf("AstNodeArrayDecl allocated for 'ID LSBRA RSBRA'\n");
    $$->arrayName = $1;
    $$->arrayLength = NULL;
}
{
    $$=malloc(sizeof(struct AstNodeArrayDecl));
    printf("AstNodeArrayDecl allocated for 'ID LSBRA INT_VALUE RSBRA'\n");
    $$->arrayName = $1;
    $$->arrayLength = $3;
};

```

```

{
    $$=malloc(sizeof(struct AstNodeArrayCall));
    printf("AstNodeArrayCall allocated for 'ID LSBRA RSBRA'\n");
    $$->arrayName = $1;
    $$->elementIndex = NULL;
}
{
    $$=malloc(sizeof(struct AstNodeArrayCall));
    printf("AstNodeArrayCall allocated for 'ID LSBRA content RSBRA'\n");
    $$->arrayName = $1;
    $$->elementIndex = $3;
    if($$->elementIndex->contentType != CONTENT_TYPE_ID || $$->elementIndex->contentType != CONTENT_TYPE_INT_NUMBER) {
        printf("Error: invalid array index type\n");
    }
}

```

Produzione di supporto per inizializzazione e assegnazione di un array

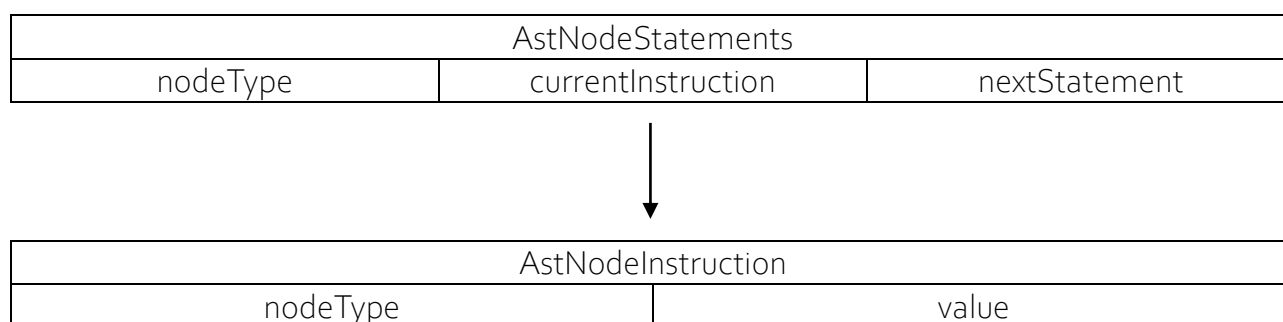
2.3 Abstract Syntax Tree

All'interno del file "ast.h" sono definiti i nodi dell'abstract syntax tree. Essi rappresentano i costrutti della grammatica sotto forma di albero. Questa struttura dati è indispensabile sia per l'analisi semantica, sia per la successiva fase di traduzione.

Nel caso in esame si è scelto di utilizzare il costrutto "struct" del linguaggio C per rappresentare i nodi.

Il nodo principale è "AstNodeStatements", che racchiude al suo interno tutti i nodi di tipo "instruction". In esso è definito il tipo di nodo (STATEMENT_NODE), il puntatore alla struct dell'istruzione corrente e il puntatore alla struct dell'istruzione successiva.

A sua volta, nel nodo "instruction", è definito il tipo di nodo e il puntatore alla struct, entrambi relativi alla specifica instruction riconosciuta.



Rappresentazione dei principali nodi di ast.h

Onde evitare di appesantire inutilmente la relazione, si rimanda al codice per la struttura degli altri nodi.

2.3.1 Strutture a supporto

A supporto dei costrutti rappresentanti i nodi, sono state definite le seguenti strutture dati:

SymbolType	usato per definire il tipo di simbolo nel momento in cui avviene la memorizzazione nella symbol table
DataType	usato per definire il tipo del token restituito dal lexer, relativamente ad una variabile, ad un array, ad una funzione o ad un valore restituito dalla stessa
ContentType	usato per specificare il tipo dell'elemento che si trova a destra di una assegnazione
NodeType	specifica il tipo di nodo. Necessario per la fase di traduzione
yystype	tipo union che raccoglie tutte le tipologie di nodi definiti nell'ast.h

2.4 Symbol Table

Il compito della symbol table è quello di tenere traccia della struttura semantica degli identificatori. La struttura scelta per la memorizzazione è quella della hash table, sia per la sua semplicità di gestione, sia per la velocità ed efficienza nella memorizzazione/ricerca dei simboli. Per l'implementazione della tabella è stata utilizzata la libreria "uthash" come supporto per la gestione degli hash, insieme al codice specifico implementato nel file symboltable.h

2.4.1 Struttura dei record

Ogni symbol table contiene dei record per ogni identificatore, con dei campi che vengono valorizzati solo in alcuni casi. Nello specifico, i record, vengono creati durante la fase di parsing a patto che siano verificate determinate condizioni: uthash memorizza i simboli usando come chiave il loro nome; tuttavia, al momento dell'inserimento, non controlla se è stato già memorizzato un simbolo con lo stesso nome, lasciando all'utente tale verifica. Pertanto, nel parser, prima dell'inserimento di un simbolo, viene effettuata un'operazione di lettura nella hash table al fine di evitare l'inserimento di record duplicati.

Nella tabella sottostante, è mostrata la struttura dedicata alla gestione del simbolo con le relative informazioni:

NOME	FUNZIONE
<i>symbolName</i>	Nome del simbolo. Funge anche da chiave per la memorizzazione.
<i>symbolType</i>	Tipo di simbolo memorizzato (variable, content, function, ecc.)
<i>dataType</i>	Tipo di dato memorizzato (void, int, float, char, none)
<i>returnType</i>	Tipo di dato restituito dalla funzione (negli altri casi NULL)
<i>funcName</i>	Nome della funzione a cui appartiene un parametro
<i>funcParameters</i>	Vettore per la memorizzazione dei parametri di una funzione
<i>arrayLength</i>	Lunghezza del vettore, numero degli elementi che esso contiene.
<i>valueOper</i>	Tipo dell'elemento che si trova a destra dell'assegnazione (valore, funzione, variabile, array)

Come già accennato, alcuni campi vengono valorizzati in casi specifici (es. quelli relativi alle funzioni), diversamente vengono settati a NULL.

Inoltre, sempre all'interno della stessa struttura, è definito il seguente comando:

```
UT_hash_handle hh;
```

questo serve a rendere la struct hashable. Il fatto di valorizzarlo come "hh" è suggerito dalla documentazione di `uthash`.

2.4.2 Gestione degli scope

Tenendo presente la gestione degli scope in C, per implementare la stessa funzionalità relativamente alla gestione e alla visibilità delle variabili, si è optato per la strada più intuitiva: gestire ogni scope con una symbol table separata.

Nello specifico, `uthash`, in fase di memorizzazione, fa' uso di una funzione del tipo:

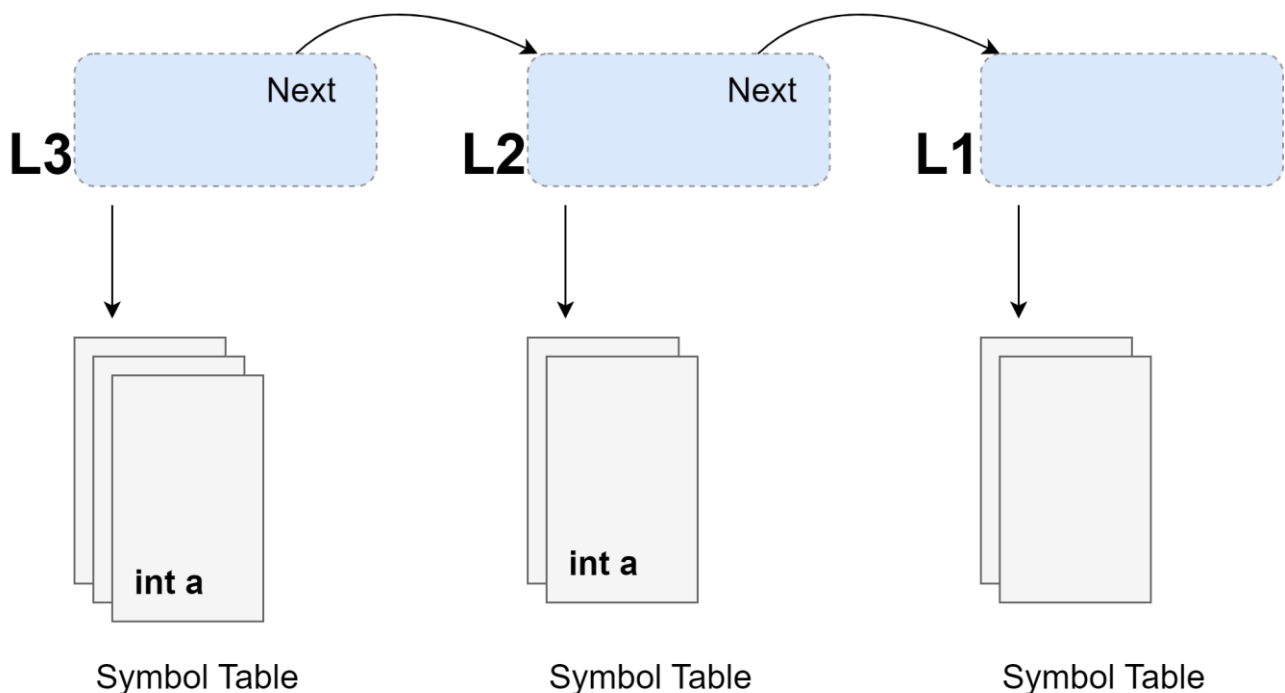
```
HASH_ADD_STR(pointer, id, s);
```

dove "pointer" rappresenta il puntatore alla symbol table nella quale si vuole memorizzare il simbolo, "id" rappresenta il nome del simbolo, "s" la struttura dati che si vuole memorizzare.

Pertanto, per gestire molteplici scope, è necessario utilizzare diversi puntatori. In aggiunta, per poter implementare una gerarchia tra gli stessi, occorre collegarli a partire da quello più generale.

Per gestire tale logica, si è optato per la seguente configurazione:

viene creata una linked list ogni volta che si cambia scope. Ogni elemento contiene al suo interno un puntatore (pointer) ad una struct (s) che rappresenta il simbolo e un puntatore all'elemento precedente della lista (next).



Funzionamento della symbol table

Alla prima iterazione, nel momento in cui viene identificato il non-terminal "program", viene creato l'elemento della lista L1, il cui parametro "next" è NULL. Nel momento in cui si vanno ad aggiungere nuovi simboli nello stesso scope, viene usato come puntatore il parametro "pointer" di L1, associando così, ogni simbolo, alla medesima symbol table.

Quando si entra in uno scope più interno, viene creato un nuovo elemento L2 della lista. Il parametro "next" di L2 verrà impostato in modo che punti ad L1. A questo punto, tutti gli elementi dello scope più interno verranno associati ad una nuova symbol table il cui puntatore non è più il valore "pointer" memorizzato in L1, ma quello memorizzato in L2.

Le stesse operazioni vengono svolte man mano che si entra in scope più interni.

Nel momento in cui si esce da uno scope per ritornare a quello più esterno, viene eliminato l'elemento della lista corrente L2 insieme a tutti i simboli ad esso appartenenti e si imposta come elemento corrente della lista il suo predecessore L1.

Nel file parser.y si tiene traccia dell'elemento attuale Li, mentre l'implementazione delle funzioni per la creazione e distruzione degli elementi Li e dei simboli ad essi associati avviene nel file symboltable.h.

Nella tabella sottostante è mostrato lo schema della struct per la gestione della linked list.

NOME	FUNZIONE
<i>scope</i>	Variabile di tipo intero per tracciare il numero di scope
<i>next</i>	Puntatore all'elemento precedente della lista (scope più esterno)
<i>symTab</i>	Puntatore alla struct symTab. Utilizzato per la creazione di nuovi simboli

2.4.3 Operazioni sulla Symbol Table

Nello stesso file *symboltable.h* sono definite le funzioni che permettono la manipolazione della symbol table. Queste, vengono poi richiamate durante la fase di parsing.

createList	Permette la creazione di un nuovo elemento della linked list. Richiede, come parametri il numero relativo allo scope e l'indirizzo dell'elemento precedente.
deleteList	Permette la cancellazione dell'elemento corrente della linked list, insieme a tutti i simboli ad esso collegati.
createSym	Funzione per la creazione del simbolo. Oltre ai parametri relativi al simbolo stesso, richiede il puntatore all'elemento corrente della linked list.
findSym	Ricerca di un simbolo unicamente all'interno dello scope corrente.
findSymtab	Funzione per la ricerca di un simbolo precedentemente memorizzato. A differenza della precedente, se non trova il simbolo nello scope corrente, prosegue la ricerca negli scope via via più esterni. Restituirà NULL solo se il simbolo non è stato dichiarato in nessuno degli scope più esterni.

2.5 Analisi Semantica

Dopo la fase di scanning e di parsing, lo step successivo è quello dell'analisi semantica. Essa consiste nell'effettuare controlli sulla semantica delle istruzioni. In altre parole, consiste nel controllare se, nonostante la fase di parsing sia andata a buon fine, sono presenti errori di significato nel codice.

Nel caso in questione, i controlli semantici vengono effettuati in concomitanza con la fase di parsing e in particolare nel momento in cui viene riconosciuta una nuova produzione. Ciò permette di sfruttare le informazioni presenti sia nel nodo dell'ast relativo all'istruzione corrente, sia nella symbol table.

2.5.1 Controlli su Inizializzazioni e Assegnazioni

- Durante l'inizializzazione di una variabile viene verificato che non sia stata precedentemente inizializzata una variabile con lo stesso nome.
- Durante l'inizializzazione e l'assegnazione di una variabile, si verifica che questa non sia stata dichiarata come void.
- Durante l'assegnazione di un elemento ad una variabile, si verifica che il tipo dell'elemento (*DataType*) coincida con il tipo della variabile. Nel caso quest'ultimo sia una costante, la verifica è immediata. Nel caso di una funzione si fa' riferimento al tipo di dato restituito dalla funzione. Nel caso di un'espressione si fa' riferimento al tipo di dato memorizzato nel nodo expression. Nel caso di un array si fa' riferimento al tipo di dato relativo all'array stesso.

2.5.2 Controlli sulle espressioni

- Relativamente alle espressioni, viene verificato che non vengano effettuate operazioni di somma, sottrazione, moltiplicazione e divisione tra membri aventi un *DataType* differente.
- Per la divisione viene verificato che il divisore sia diverso da 0. Diversamente viene restituito un errore.
- Le espressioni contenenti numeri negativi vengono gestite, a patto che tra gli operatori e l'operando sia presente uno spazio vuoto.

2.5.3 Controlli sugli array

- Nel caso di inizializzazione, viene verificato che non esista un array con lo stesso nome.
- Nel caso di inizializzazione del tipo *int myArray[]*; viene verificato che all'interno delle parentesi sia effettivamente presente un valore.
- Nel caso di assegnazione, viene fatto un controllo circa il valore assegnato. Se questo è del tipo vuoto {}, viene restituito un errore.
- Nel caso di assegnazione di singolo valore, del tipo *myArray[1] = 24*; viene verificato che all'interno delle parentesi sia presente un valore.
- Un'assegnazione del tipo *myArray[3] = {24, 27, 29}* restituisce errore a prescindere, in quanto è possibile assegnare un set di elementi all'array solo in fase di inizializzazione.
-

2.5.4 Controlli sulle dichiarazioni di funzioni

- Nel momento dell'inizializzazione di una funzione, viene verificato che nella symbol table non sia presente una funzione/variabile con il medesimo nome.

2.5.5 Controlli sulle chiamate a Funzioni

- Per tutte le funzioni viene verificato che la funzione chiamata sia effettivamente presente all'interno della symbol table.
- Nel caso di dichiarazione di funzioni con parametri, viene verificato, al momento della chiamata, che il numero dei parametri coincida con quello dei parametri attesi e che il tipo degli stessi coincida.

2.5.6 Controlli relativi agli scope

Questo tipo di controllo è necessario tutte le volte che viene effettuata la ricerca di una variabile/funzione in un preciso scope.

Occorre ricordare che nel caso del linguaggio C, per le assegnazioni, da uno scope esterno non è possibile accedere a variabili dichiarate in scope più interni. Al contrario, a partire da uno scope interno, è possibile accedere a variabili precedentemente dichiarate negli scope più esterni. Pertanto, nel momento in cui viene ricercata una variabile (es. "a"), questa verrà cercata prima nello scope corrente e, nel caso questa non venga trovata, si andrà a ricercarla nello scope via via più esterno.

Nel caso delle inizializzazioni, in ogni scope è possibile inizializzare nuovamente le variabili, anche se queste sono state precedentemente inizializzate in uno scope più esterno. È possibile, ad esempio, dichiarare *int a* e, nel momento in cui si entra nel corpo di un ciclo if, è possibile nuovamente dichiarare *int a*, senza che venga restituito un messaggio di errore.

2.5.7 Controlli su printf e scanf

- Nel caso delle funzioni di input e di output, viene verificato che all'interno delle parentesi, sia effettivamente presente una stringa. Di conseguenza, una produzione del tipo *printf()* o *scanf()* produrrà un errore.

2.6 Generazione del codice

Per generare il codice oggetto è stato utilizzato l'abstract syntax tree creato durante la fase di parsing. Esso contiene la rappresentazione intermedia del programma di partenza.

Nello specifico, il codice per la traduzione è presente nel file *c2py.h*. Qui è stata definita una funzione per ogni nodo dell'AST che genera il codice oggetto in base agli specifici valori ivi memorizzati.

Tutte queste funzioni, vengono chiamate da una funzione generica "*translate*" che, leggendo il nodo principale (root), esegue la traduzione dall'alto verso il basso.

Anche in questo file vi sono funzioni di supporto come quella per l'identificazione del node type. Poiché a differenza del C, in python la funzione main non è obbligatoria, è stato possibile scegliere se eliminarla totalmente in fase di traduzione oppure tradurla e richiamarla in coda a tutto il codice tradotto. Si è optato per la seconda opzione.

3. TEST

Al fine di verificare il corretto funzionamento del transpiler, sono stati eseguiti alcuni file di test organizzati in diverse sottocartelle. Ciascuna contiene due versioni di una specifica istruzione o struttura da testare: una corretta con la relativa traduzione e una volutamente errata al fine di verificare il corretto riconoscimento degli errori sintattici e semantici.

Va' specificato che, nonostante gli errori vengano riconosciuti, essi sono considerati tutti bloccanti: nel momento in cui si verifica anche un solo errore, anche se in molti casi la fase di analisi continua, il file con la relativa traduzione non viene generato.

4. PROBLEMATICHE RISCONTRATE

In questo capitolo verranno elencate le varie problematiche incontrate durante la fase di progettazione del transpiler e le relative soluzioni implementate.

4.1 Problematica espressioni

Una delle prime problematiche riscontrate è stata quella relativa alla gestione delle espressioni del tipo: `int a = ELEMENT;`

L'idea iniziale è stata quella di gestire il tutto con un unico nodo dell'ast, un non-terminal di tipo **Assignment** e diverse produzioni ad esso collegate che tenessero conto delle varie tipologie di assegnazione, quindi:

<code>int a = 5;</code>	<code>TYPE ID EQ VALUE</code>
<code>int a = function;</code>	<code>TYPE ID EQ FUNCTION</code>
<code>int a = expression;</code>	<code>TYPE ID EQ EXPRESSION</code>

Se per la prima produzione non ci sono stati problemi, quando si è passati alla seconda, si è posto il problema di gestire il contenuto di **EXPRESSION** in un'unica variabile. Poiché il nodo **EXPRESSION** veniva direttamente decomposto con un'altra produzione e gestito, si perdeva il valore utile, ovvero la stringa contenente tutta l'espressione.

Dopo vari e fantasiosi tentativi per risolvere il problema, cercando di modificare la minor quantità possibile di codice, mantenendo inalterate le produzioni, è stata esclusa anche l'ipotesi di modificare la struct "AstNodeExpression" inserendo un elemento che contenesse il valore di ritorno.

Si è così realizzato che, senza una ulteriore diramazione, non era possibile gestire l'elemento dopo l'uguale che, oltre al valore in sé per sé, necessita almeno del tipo nel caso di un valore o di altre informazioni nel caso di una funzione o di un'assegnazione.

Pertanto, al fine di evitare di creare un unico nodo con troppe informazioni da settare a NULL a seconda dei casi, sono stati definiti 3 nuovi nodi nell'ast per tenere conto dei possibili elementi che possono comparire dopo l'uguale, in particolare un elemento unico (`int float char id`), una funzione oppure un'assegnazione.

```

assignment:
| types ID EQ content      {
    $$ = malloc(sizeof(struct AST_NODE_ASSIGN)); printf("AST_NODE_ASSIGN allocated");
    $$->variable_name = $2;
    $$->variable_type = str_to_type($1);
    $$->assign_value.val = $4; //valore vero dopo uguale, oppure FunctionCall oppure Expression
    $$->assign_type = $4
    $$->operand = malloc(sizeof(struct AST_NODE_OPERAND)); //puntatore struct tipo operand
    $$->operand->variable_type //tipo operando restituito
    $$->operand->value //valore restituito
    $$->operand->content_type //int expression function
};
/* | types ID EQ functionCall

```

Come è possibile vedere nell'immagine in alto, il nuovo non-terminal **assignment** ha una produzione del tipo **TYPES ID EQ CONTENT** con "CONTENT" ulteriore non-terminal che tiene conto dei possibili elementi che possono esserci dopo l'uguale. Questo permette la memorizzazione sotto forma di stringa, dell'intero elemento assegnato.

In seguito a questa problematica si è appreso che bisogna trovare il giusto bilanciamento tra grandezza dei nodi dell'AST e numero di diramazioni dell'albero. È possibile avere un albero con molte diramazioni e nodi molto semplici, oppure un albero con poche diramazioni e nodi molto corposi in termini di informazioni.

4.2 Problematica parser-lexer

Questa problematica si è verificata nel momento in cui sono stati separati i 2 non-terminal relativi ad **initialization** e **assignment**.

Nello specifico, è stato notato in fase di test che, relativamente alla **initialization**, in alcuni casi il parsing andava a buon fine mentre in altri casi restituiva un errore.

Poiché l'errore visualizzato era "Syntax error", si continuava a cercare l'errore nel parser e nell'AST.

Ci si è poi resi conto che nel caso di assignment con un elemento a 2 cifre (int a = 54), il parsing andava a buon fine. Nel caso di assignment con un elemento a singola cifra (int a = 5), veniva restituito un errore.

Dopo qualche ora di tentativi si è scoperto che, nonostante il messaggio fuorviante, l'errore era dovuto al modo in cui era stato strutturato il lexer: essendoci una priorità in fase di definizione dei lessemi, era stato definito prima "digit" e poi "integer number". Pertanto con numeri ad una cifra veniva riconosciuto il lessema **digit** che, non essendo definito nelle nostre produzioni, restituiva "syntax error", mentre con numeri a 2+ cifre veniva restituito il lessema **int_value** andando a buon fine con il parsing.

Questa problematica è servita a comprendere che i messaggi di errore possono essere fuorvianti e che le cause vanno ricercate dal principio.

4.3 Problematica produzione id eq id

Per la produzione **ID EQ ID** è stata inizialmente attuata un'accortezza particolare. La produzione **ID EQ CONTENT** teneva conto dell'assegnazione di un valore ad una variabile, ma anche dell'assegnazione di una variabile ad un'altra variabile. Questo perché il non-terminal **content**, aveva tra le sue possibili produzioni anche **ID**.

Tuttavia, poiché l'assegnazione di **ID EQ ID** veniva gestita diversamente, era stata scritta una produzione a parte per quest'ultima e la si era messa prima di **ID EQ CONTENT**. In questo modo, per l'assegnazione di una variabile ad un'altra veniva utilizzata la produzione **ID EQ ID**, mentre per l'assegnazione di un valore ad una variabile veniva utilizzata una produzione del tipo **ID EQ CONTENT** con content valore int, float o char.

Ciò restituiva, in fase di compilazione, un warning relativo ad un conflitto di tipo Reduce – Reduce.

In seguito, poiché ci si è resi conto che il token **ID** sarebbe servito anche nelle produzioni di tipo **expression**, si è uniformato tutto nel non-terminal di tipo **content**.

In generale, durante tutta la fase di progettazione/implementazione del transpiler, le produzioni relative ad initialization e assignment, sono state più volte riviste.

4.4 Errore di segmentazione (core dump creato)

Questo errore è stato uno dei più ostici da risolvere e si è verificato nel momento in cui è stata implementata la symbol table.

In generale, esso si verifica quando il programma tenta di accedere ad un'area di memoria a cui non ha diritto oppure che non esiste (come nel nostro caso). Un'altra possibilità è quella in cui si va' out of memory per qualche motivo. Inoltre, questo errore, non viene generato in fase di compilazione ma di esecuzione, esattamente nel momento in cui si andavano ad utilizzare produzioni in cui veniva richiamata la symbol table.

Dopo un attento riesame della stessa e del parser, sono state effettuate alcune correzioni marginali, tuttavia l'errore continuava a ripresentarsi. Rimuovendo la symbol table questo spariva e le produzioni venivano riconosciute senza errori.

È stata così inserita una printf per ogni riga di codice per scoprire che l'errore veniva generato nel momento in cui veniva eseguito il comando a riga 95 dell'immagine sotto.

```
91
92 struct SymTab *findSym(char *symbolName, struct List *symList) {
93     printf("Entro in findSym\n");
94     struct SymTab *s;
95     HASH_FIND_STR(symList->symTab, symbolName, s); //hash table list
96     return s; //puntatore alla symbol table cercata
97     if (s==NULL) {
98         printf("\n Error: Symbol %s not found \n", symbolName);
99         return NULL;
```

Si è scoperto che l'errore era dovuto nuovamente al lexer.

L'errore è stato risolto aggiungendo la seguente stringa al codice corrispondente al riconoscimento di ogni lessema: **yylval.string=strdup(yytext)**.

In generale, quando flex identifica un lessema, questo viene messo in cima ad un input buffer e viene chiamato yytext(). Questo punta all'elemento in cima al buffer, quindi all'ultimo lessema identificato.

Nel momento in cui viene identificato un nuovo lessema, il valore a cui punta yytext sarà sovrascritto.

In altre parole, yytext è valido solo fino a quando non viene riconosciuto un nuovo lessema.

Come si risolve se questo valore ci serve successivamente per il processing con un parser che ha un look ahead diverso da zero?

Per preservare il valore che ci serve, lo si copia con strdup() o una funzione simile.

Flex esegue una routine chiamata yylex() per trovare il token successivo da dare in input a bison. Oltre a restituire il tipo di token trovato, mette il valore adesso associato nella variabile globale yylval.

Yylval è comunemente utilizzata nel contesto dell'analisi sintattica (parser) per associare informazioni aggiuntive ai token riconosciuti dallo scanner.

In definitiva, l'espressione definita per correggere l'errore, è utilizzata per passare le informazioni relative ai token dallo scanner al parser, in modo che il parser possa utilizzare tali informazioni durante l'analisi sintattica per eseguire le azioni appropriate.

L'errore riscontrato, era dovuto al fatto che il parser non era in grado di accedere al valore vero e proprio del token letto. Questo è stato passato al parser sotto forma di stringa risolvendo il problema.

4.5 Conflitti shift-reduce e reduce-reduce

Durante i primi test, nel momento in cui si lanciava il comando **bison -d parser.y**, comparivano degli errori di questo tipo:

```
parser.y: attenzione: 156 salta/riduce i conflitti [-Wconflicts-sr]
```

```
parser.y: attenzione: 1 ridurre/ridurre il conflitto [-Wconflicts-rr]
```

In questo caso, essendo già noti i 2 tipi di errori e le relative soluzioni, non è stato difficoltoso risolverli.

RISOLUZIONE DEI CONFLITTI SHIFT-REDUCE: si attuano dei meccanismi di priorità relativi ai token.

In particolare i conflitti si avevano sugli operatori aritmetici, logici e di comparazione. Modificando questi token, mettendo la priorità %left per tutti tranne che per la sottrazione (%right), la cosa si è risolta.

RISOLUZIONE DEI CONFLITTI REDUCE-REDUCE: si verificava su ID EQ ID e ID EQ CONTENT, 2 produzioni equivalenti (content a sua volta può generare ID).

In generale, questo tipo di errore viene automaticamente risolto da Bison scegliendo la produzione definita prima.

Al fine di evitare completamente il verificarsi dell'errore si è scelto di mantenere una sola produzione.

4.6 Problema assegnazione con expression

Come già accennato sopra, nel corso del progetto sono state ridefinite più volte le produzioni per la gestione dell'inizializzazione e l'assegnazione di una variabile.

Alla fine si è deciso di utilizzare un nodo "initialization" per gestire la sola inizializzazione di una variabile (senza assegnazione): **INT A**.

L'assegnazione, invece, copre 2 casistiche: **A = 5**, (controllando che la variabile A sia stata inizializzata) e **INT A = 5** (controllando opportunamente che il tipo assegnato coincida).

Va' precisato che la produzione che viene gestita è del tipo **TYPE ID EQ CONTENT**. Dove CONTENT rappresenta non solo un numero, ma anche un'espressione o una funzione. Nel testare un'assegnazione del tipo **INT A = 5 + 5**; veniva riscontrato il seguente errore:

```
int h = 6+6;
keyword 'INT' detected
Defined 'type: INT'
Found DELIM
Found ID
Found DELIM
keyword '=' detected
Found DELIM
Found INTEGER NUMBER
AstNodeOperand allocated for 'INT_VALUE'
Found INTEGER NUMBER
'types ID EQ content': the variable h has not already been declared and then I create the symbol table for this variable
AstNodeAssign allocated for 'types ID EQ content'
syntax error
Errore di segmentazione (core dump creato)
francesco@francesco:~/Documenti/GitHub/C2Py-transpiler/transpiler$ ./a.out
int a;
```

Dopo qualche test, si è scoperto che nell'inserire l'espressione in quel modo, veniva letto "6+6" come un unico elemento e, non trovando questo riscontro nel lexer, veniva restituito un syntax error.

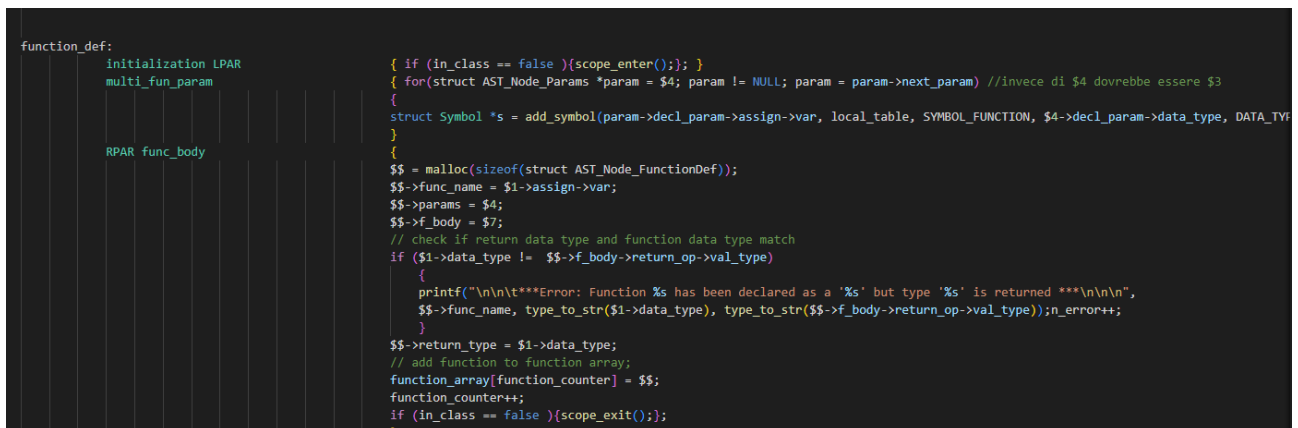
Pertanto, al fine di poter interpretare correttamente l'assegnazione di una espressione, è necessario inserire gli spazi tra gli operandi. La forma corretta è del tipo: **int a = 5 + 5**. Come si evince dallo screenshot sottostante, la modalità suddetta viene correttamente riconosciuta.

```
a = 5 + 5;
Found ID
Found DELIM
keyword '=' detected
Found DELIM
Found INTEGER NUMBER
AstNodeOperand allocated for 'INT_VALUE'
Found DELIM
keyword '+' detected
Found DELIM
Found INTEGER NUMBER
AstNodeOperand allocated for 'INT_VALUE'
Found SEMICOLON
AstNodeExpression allocated for 'content ADD content'
Expression of type Sum
AstNodeOperand allocated for 'expression'
AstNodeAssign allocated for 'ID EQ content'
AstNodeInstruction allocated for 'assignment SEMICOL'
```


4.7 Problema scope delle funzioni

Si è rappresentato nel momento in cui si stavano effettuando i test finali. Questa problematica è stata del tutto inattesa.

Nello specifico, traducendo `int funzione (int a, int b { a=5; b=4; }` venivano dichiarati i parametri tutti all'interno dello stesso scope. Dopo una ricerca approfondita ci si è resi conto che un modo efficiente per risolvere la questione era relativo all'ordine in cui venivano letti i token.



```
function_def:
    initialization LPAR
    multi_fun_param
    RPAR func_body

    { if (in_class == false ){scope_enter();} }
    { for(struct AST_Node_Params *param = $4; param != NULL; param = param->next_param) //invece di $4 dovrebbe essere $3
      {
        struct Symbol *s = add_symbol(param->decl_param->assign->var, local_table, SYMBOL_FUNCTION, $4->decl_param->data_type, DATA_TYPE);
      }
    }
    $$ = malloc(sizeof(struct AST_Node_FunctionDef));
    $$->func_name = $1->assign->var;
    $$->params = $4;
    $$->f_body = $7;
    // check if return data type and function data type match
    if ($1->data_type != $$->f_body->return_op->val_type)
    {
      printf("\n\n***Error: Function %s has been declared as a '%s' but type '%s' is returned ***\n\n",
        $$->func_name, type_to_str($1->data_type), type_to_str($$->f_body->return_op->val_type)); n_error++;
    }
    $$->return_type = $1->data_type;
    // add function to function array;
    function_array[function_counter] = $$;
    function_counter++;
    if (in_class == false ){scope_exit();}
```

Relativamente allo screenshot di sopra, i token verdi vengono letti da sinistra verso destra e man mano che avviene il match con una produzione che ha una diramazione, questa viene espansa. Il problema stava nel codice relativo a `function_def`. Questo veniva eseguito solo dopo aver letto tutte le produzioni in verde. Ciò significa che la funzione `beginScope()` veniva eseguita solo dopo aver inizializzato i parametri locali della funzione e aver eseguito il corpo. Ci serviva un modo per eseguire `beginScope()` prima che venisse eseguito il codice relativo a tutti i token in verde.

Si è così scoperto che è possibile dividere i token in verde in più righe ed eseguire del codice nel momento in cui avviene il match con un dato numero di token. Pertanto, dopo `initialization LPAR` viene eseguita una parte di codice, stessa cosa dopo `functionParams`.

In questo modo `beginScope()` viene eseguito prima di passare ai parametri locali della funzione in modo che quegli stessi parametri e il body vengano creati all'interno dello scope più interno. Altra accortezza è quella di considerare quel codice come un ulteriore elemento della produzione, quindi dopo LPAR c'è `$3` che equivale al codice eseguito in quel momento. È stato necessario ridefinire i puntamenti agli elementi della produzione.

Si è reso necessario fare la stessa cosa per la produzione relativa al main e per tutti i blocchi if else while in quanto nel linguaggio C ogni blocco di parentesi graffe comporta l'entrata in uno scope più interno.

4.8 Problema parametri “function decl” e “function call”

Dai test fatti in precedenza sembrava funzionare tutto correttamente. In realtà, ad un certo punto, si si è resi conto che dichiarando una funzione e memorizzando il tipo dei suoi parametri in un vettore, questi se erano maggiori di 8 non venivano memorizzati.

```
prova(5);
Found ID
keyword '(' detected
Found INTEGER NUMBER
AstNodeOperand allocated for 'INT_VALUE'
keyword ')' detected
AstNodeFunctionParams allocated for 'content'
AstNodeFunctionCall allocated for 'ID LPAR functionParams RPAR'
Parameters of the declared function: floatflo
Parameters of the called function: int
The parameters in the function call are incorrect
Found SEMICOLON
AstNodeInstruction allocated for 'functionCall SEMICOL'
```

```
francesco@francesco:~/Documenti/GitHub/C2Py-transpiler/transpiler$ ./a.out
int prova(float a, float b) { char c;}
keyword 'INT' detected
Defined 'type: INT'
Found DELIM
Found ID
keyword '(' detected
AstNodeInit allocated for 'types ID'
keyword 'FLOAT' detected
scope aperto

Defined 'type: FLOAT'
Found DELIM
Found ID
Found COMMA
Found DELIM
keyword 'FLOAT' detected
Defined 'type: FLOAT'
Found DELIM
Found ID
keyword ')' detected
AstNodeFunctionParams allocated for 'types ID'
AstNodeFunctionParams allocated for 'types ID COMMA functionParams'
Sono entrato nel ciclo for

La sizeof vale 8
```

Dopo varie ricerche si è collegato il problema ad un codice errato nella symbol table.

```
//aggiunge una symbol table con simbolo alla lista
struct SymTab *createSym(char *symbolName, struct List *list, enum SymbolType symbolT
    struct SymTab *s;
    s = findSym(symbolName, list);
    if(s == NULL) {
        s = (struct SymTab *)malloc(sizeof(struct SymTab));
        s->symbolName = symbolName;
        s->symbolType = symbolType;
        s->dataType = dataType;
        s->returnType = returnType;
        s->funcName = funcName;
        printf("La sizeof vale %ld \n \n",sizeof(funcParameters));
        if (funcParameters != NULL) {
            for(int i=0;i<(sizeof(funcParameters));i++) {
                s->funcParameters[i] = funcParameters[i];
            }
        }
        s->arraylength = arraylength;
```

In particolare nel ciclo for, momento in cui si capiva il numero di elementi del vettore funcParameters, usavamo un sizeof. Questo andava a restituire sempre 8 in quanto non conta il numero di elementi presenti nel vettore ma il numero di byte del tipo dello stesso, quindi int 8 byte.

Sostituendo il codice errato con un ciclo che itera su tutti gli elementi del vettore abbiamo risolto il problema e ottenuto il risultato desiderato.

```
//aggiunge una symbol table con simbolo alla lista
struct SymTab *createSym(char *symbolName, struct List *list, enum Sy
    struct SymTab *s;
    s = findSym(symbolName, list);
    if(s == NULL) {
        s = (struct SymTab *)malloc(sizeof(struct SymTab));
        s->symbolName = symbolName;
        s->symbolType = symbolType;
        s->dataType = dataType;
        s->returnType = returnType;
        s->funcName = funcName;
        if (funcParameters != NULL) {
            for(int i=0;i<100;i++) {
                s->funcParameters[i] = funcParameters[i];
            }
        }
    }
}
```

```
francesco@francesco:~/Documenti/GitHub/C2Py-transpiler/transpiler$ ./a.out
int prova(float a, float b, float c, int e) { char r; }
keyword 'INT' detected
Defined 'type: INT'
Found DELIM
Found ID
keyword '(' detected
AstNodeInit allocated for 'types ID'
keyword 'FLOAT' detected
scope aperto
```

```
prova(6,5.5);
Found ID
keyword '(' detected
Found INTEGER NUMBER
AstNodeOperand allocated for 'INT_VALUE'
Found COMMA
Found FLOAT NUMBER
AstNodeOperand allocated for 'FLOAT_VALUE'
keyword ')' detected
AstNodeFunctionParams allocated for 'content'
AstNodeFunctionParams allocated for 'content COMMA functionParams'
AstNodeFunctionCall allocated for 'ID LPAR functionParams RPAR'
Parameters of the declared function: floatfloatfloatint
Parameters of the called function: intfloat
The parameters in the function call are incorrect
Found SEMICOLON
AstNodeInstruction allocated for 'functionCall SEMICOL'
```

5. CONCLUSIONI

Il progetto svolto ci ha permesso di acquisire una comprensione più approfondita del funzionamento e delle dinamiche legate al mondo dei compilatori. Attraverso l'esplorazione del linguaggio C, abbiamo avuto l'opportunità di ampliare le nostre competenze, comprendendo meglio le sfumature del linguaggio e imparando ad utilizzare gli strumenti necessari per identificare ed affrontare gli errori che possono sorgere durante la fase di progettazione e implementazione di un compilatore. Questa esperienza ci ha fornito una base solida per affrontare futuri progetti legati alla programmazione e alla creazione di software.