de Candia Giuseppe

Prof. Scioscia Floriano

***Formal Languages & Compilers***

C2Py

Il lavoro presentato in questo documento, consiste nella realizzazione di traduttore unidirezionale capace di convertire un subset di funzionalità C nell’equivalente codice Python.

Il tool è stato realizzato in C con l’ausilio dei software open source FLEX e BISON.

**Indice**

1. Introduzione

1.1 Linguaggio C

1.2 Linguaggio Python

1.3 Tool di Supporto

2. Traduttore C2Py

2.1 lexer.l

2.2 parser.y

2.3 AST.h

2.4 compiler.c

3. Casi di Test

3.1 Test1

3.2 Test2

3.3 Test + errore

**1 Introduzione**

La comunicazione fra un essere umano ed un calcolatore necessita di una traduzione delle istruzioni a causa del divario fra i linguaggi compresi delle due entità. La parlata comunemente dagli uomini non può esser direttamente compresa da un computer che richiede un linguaggio macchina al fine di eseguire determinate operazioni. Da questa necessità sono stati sviluppati i compilatori, strumenti capaci di rendere questa traduzione automatica ed invisibile all’utente che si interfaccia con il calcolatore.

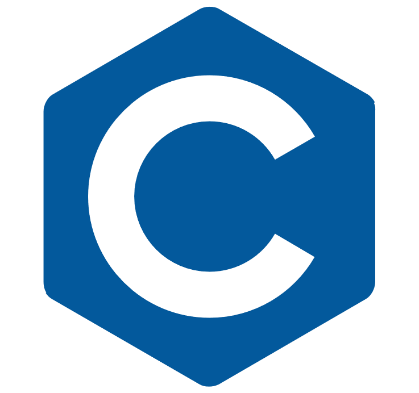
L’obiettivo del lavoro svolto è quello di effettuare una traduzione unidirezionale fra due linguaggi di programmazione ad alto livello per simulare il funzionamento di un compilatore.

I linguaggi scelti a tal proposito sono:

**source** : **C**

**target**  : **Python**

* 1. **Linguaggio C**

Il linguaggio **C** è un linguaggio procedurale nato nel 1972 e può esser considerato il latino dei linguaggi di programmazione. Esso rientra nella categoria dei LdP di alto livello ma presenta caratteristiche dei linguaggi a basso livello come ad esempio la possibilità di riferirsi ad indirizzi di memoria. Il C è rinomato per la sua efficienza e si è diffuso per la realizzazione di software di sistema su molte piattaforme hardware.

La semantica del C sfrutta un insieme ristretto di concetti a stretto contatto con l’hardware dei calcolatori tanto che molte delle sue funzioni sono direttamente traducibili con una singola istruzione di linguaggio macchina. Il concetto di puntatore assume un ruolo centrale nel linguaggio poiché tale entità coincide con l’indirizzamento indiretto in modo da accedere alla memoria hardware.

Un programma scritto in C può esser composto da più file detti ***header file*** con estensione *.h*, caricate attraverso la direttiva *include* del preprocessore (la prima componente incaricata di leggere il codice) ed includere un vasto numero di librerie per importare funzioni che l’utente non dovrà nuovamente implementare.

Il linguaggio C prevede l’utlizzo dei classici tipi di dato come *int, char* e *float*; accanto a questi tipi elementari esso mette a disposizione una funzionalità d’aggragazione di tali tipi elementari in “nuovi” tipi definiti dall’utente dette *struct*.

* 1. **Linguaggio Python**

Python è un linguaggio d’alto livello nato nel 1991 per diversi scopi; Esso è un linguaggio multi-paradigma e pertanto altamente versatile: esso è *orientato agli oggetti*, viene utilizzato per *programmazione strutturata e funzionale*, sfruttato per lo *scripting* ecc. L’esecuzione di codice Python richiede un *Python Interpreter*,il quale legge lo script, lo compila in bytecode e quindi esegue quest’ultimo. Come appena detto, a differenza del C, in questo caso il codice non viene compilato e pertanto ciascuna istruzione deve essere di volta in volta tradotta dall’interprete, il quale deve essere costantemente in esecuzione durante il processamento del programm.

Fra le principali caratteristiche di Python vi è la *non tipizzazione delle variabili* e l’utilizzo dell’*indentazione* per delimitare i blocchi di programma. Nonostante la non tipizzazione delle variabili, il ***type check*** viene effettuato a runtime. La gestione della memoria è automatica e sfrutta un *garbage collector*.

La sua semplicità e flessibilità rispetto ad altri linguaggi stanno rendendo Python il linguaggio di programmazione più diffuso al mondo, sia per semplici script che per elaborati progetti di Artificial Intelligence e Machine Learning.

* 1. **Tool di Supporto**

Il traduttore C/Python è stato realizzato sfruttando i Tool di supporto **Flex** e **Bison**, i quali lavorano in coppia per effettuare la traduzione.

* + 1. **Flex**

Flex è un noto generatore di scanner Unix/Linux progettato per essere utilizzato insieme al parser generator Bison. L’analizzatore lessicale prodotto da Flex è una funzione C detta **yylex()** che è quella che si aspetta Bison.

Esso è scritto in C ed accetta in input dei file con estensione “**.l**”. Tali file hanno una struttura ben definita e composta da 3 sezioni:

1. Definizione di costanti, inclusione di librerie e file esterni
2. Definizione di token ed azioni
3. Procedure C di supporto

Tali sezioni vengono poi trasformate da Flex in un codice C compilabile che assume il nome di **lex.yy.c**.

Il lexer generato può essere eseguito autonomamente ed accettare in input un flusso di stringhe per cui è stato progettato. In output esso darà una sequenza ti *token* individuati e restituiti.

* + 1. **Bison**

Bison è uno strumento open source UNIX/LINUX utile a generare il parser. Esso legge file strutturati in una maniera ben definita con estensione “**.y**” al fine di generare automaticamente l’analizzatore sintattico che lavora in tandem col lexer.

Esso è composto da 3 sezioni come il file lexer.l:

1. Inclusione di librerie e dichiarazioni
2. Definizione di regole grammaticali e costrutti sintattici
3. Routine di supporto

I tre punti verranno successivamente discussi

**2 Traduttore C2Py**

Di seguito verranno descritti i file che compongono il traduttore prodotto e come i tool utilizzati descritti si interfacciano con esso.

I file generati sono i seguenti:

* **lexer.l**
* **bison.y**
* **AST.h**
* **compiler.c**

**2.1 lexer.l**

Il file lexer.l è quello che viene utilizzato da Flex per generare lo scanner che sarà in grado di riconoscere i principali *simboli* del linguaggio C e pertanto effettuare il controllo *lessicale* del codice in ingresso. Come detto precedentemente esso è composto da 3 parti:

1. Sezione Dichiarativa

In questa porzione vengono incluse le principali librerie, i file esterni con cui il target file **lex.yy.c** dovrà interfacciarsi. In questa sezione infatti si possono notare le direttive:

#include "**AST.h**"

#include "**parser.tab.h**"

Oltre alle inclusioni di librerie e file esterni, vengono poi dichiarate funzioni e strutture dati necessarie al corretto funzionamento del traduttore:

struct symbol\* **lookUp**(char\* sym);

static unsigned **symHash**(char \*sym);

struct symbol **symtab**[NHASH];

Qui possiamo notare due funzioni ed un vettore di simboli. Essi servono a realizzare le funzioni d’accesso e gestione di una *symbol table* necessaria a memorizzare le variabili incontrate nel codice in input. Tale symbol table è accessibile da ogni componente del traduttore e viene realizzata mediante un vettore di *struct symbol*, a loro volta composte da *nome, tipo* e *valore*.

Di seguito vengono dichiarati gli “atomi” che verranno successivamente utilizzati per la generazione di regole di riconoscimento di simboli e token.

1. Sezione delle Regole

A questo punto è possibile definire le regole che il riconoscitore dovrà rispettare, infatti vengono indicati simboli necessari, parole chiave del codice C, e sequenze di “simboli atomici” utili al riconoscimento di stringhe, interi e così via.

Tali regole sono delle coppie in cui a sinistra si ha il simboli, sequenze di simboli o espressioni regolari da riconoscere mentre a destra le relative azioni che il riconoscitore deve svolgere una volta riconosciuto un pattern.

e.g.

**"+" {return TOKEN\_ADD;}**

e.g.

**int {return INT\_TOKEN;}**

e.g.

({letter}|\\_)({letter}|{digit}|\\_)\* {yylval.symb=lookUp(yytext); return NAME;}

Quest’ultimo esempio è utile per capire come il riconoscimento dell’espressione regolare a sinistra invochi la funzione “lookUp()” atta a verificare l’esistenza della variabile, con il nome riconosciuto, all’interno della symbola table: in caso di non esistenza di tale variabile, la funzione scrive nella SymTab, altrimenti restituisce alla variabile *yylval.symb* il simbolo corrispondente. Tale meccanismo (utilizzo della variabile interna *yylval*) permette la diretta comunicazione fra *parser* e *lexer*. Infine, il riconoscimento dell’espressione regolare, termina con la restituzione di un *token* detto **NAME**, il quale verrà utilizzato poi dal parser per riconoscere le produzioni.

1. La sezione del codice aggiuntivo rimane vuota

**2.2 parser.y**

Il file parser.y contiene le informazioni necessarie a Bison per generare il l’analizzatore sintattico capace di riconoscere le regole grammaticali del C. Una volta riconosciute tali regole, vengono realizzare gli alberi sintattici. Come il file del lexer esso è composto da 3 parti:

1. Sezione Dichiarativa

Come nel caso precedente è necessaria una sezione di dichiarazioni di funzioni e variabili, nonché file esterni e librerie. Anche in questo caso viene dichiarata la symbol table come vettore di simboli. Inoltre viene dichiarata una funzione necessaria alla comunicazione con il lexer:

int yylex();

Dopo la dichiarazione classica di variabili e funzioni è presente una fondamentale componente necessaria al passaggio dei valori dal lexer al parser; tale sezione è indicata con **%union** e serve ad indicare sia i tipi dei token che verranno dichiarati successivamente che le variabili e strutture incluse a loro volta all’interno della precedentemente discussa *yylval* (variabile interna).

e.g.

%union{ struct symbol\* symb; }

Segue una lista di token (quelli restituiti dal lexer) per cui alcuni sono tipizzati ed altri no.

e.g.

%token <symb> NAME

Tale riga sta ad indicare che il token NAME passato dal lexer dovrà essere di tipo *symbol*.

Per concludere la sezione dichiarativa abbiamo poi una parte di codice dove viene indicato il tipo dei simboli *non – terminali* che verranno poi aggiunti all’interno della grammatica. Questi vengono identificati dalla direttiva **%type**.

e.g.

%type <express> expression

Tale riga sta a significare che il simbolo non terminale “expression” è di tipo “expression”.

1. Grammatica

Terminata la sezione dichiarativa, la seconda sezione è quella dedicata alla grammatica del parser. Qui vengono indicate le regole grammaticali sottoforma di produzioni al fine di avere una struttura ad albero fra *non – terminali* e *terminali* (questi ultimi fungono da foglie sulla parte più esterna dell’albero ).

Le regole implementate nel parser sono solo un subset di quelle che sono in realtà le “frasi” possibili in C. Le funzionalità implementate sono:

* **Espressioni** (espressioni matematich, assegnazioni a variabili etc.)
* **Iterazione** (ciclo *for*)
* **Diramazione** (*if – if elseif – if else*)
* **Input** (funzione *scanf*)
* **Output** (funzione *printf*)
* **Strutture dati** (*array*)
* **Casting** (*int/float – float/int*)

Ogni regola grammaticale è stata realizzata associando a ciascun pattern un’azione da eseguire una volta riconosciuto; tali azioni sono assimilabili a degli **innesti** di sottoalberi nell’albero principale controllati da istruzioni di diramazione atti ad eseguire il controllo *semantico*.

Le azioni appena citate vengono realizzate da delle funzioni C che allocano memoria per delle struct, precedentemente definite nel file incluso nella prima sezione “AST.h”, ed assegnano ad esse i valori dei simboli terminali e non – terminali.

e.g.

**init: INT\_TOKEN NAME TOKEN\_ASS expression TOKEN\_SEM**

**{if($4->type==SYMBOL\_INT)**

**$$=expression\_create(EXPRESSION\_ASS,expression\_name($4->type,**

**lookUp($2->name)),$4, $4->type);}**

In questo caso “*init*” è un simbolo non – terminale e la sua *produzione* è “INT\_TOKEN NAME TOKEN\_ASS expression TOKEN\_SEM”; i simboli in caps lock sono simboli terminali mentre “expression” è ancora un non – terminale definito in un altro punto della sezione 2.

A seguito del riconoscimento di tale regola ne deriva un controllo sul tipo del 4° simbolo (expression); il ramo del vero di tale controllo assegna al simbolo “init”, il valore restituito dalla funzione “expression\_create” definita nella sezione 3 che verrà a breve descritta; si noti che il passaggio del valore viene effettuato ad una variabile *$$* che permette di riferirsi al non – terminale da cui deriva la regola grammaticale.

1. Sezione di Definizione delle Funzioni

In quest’area vengono **definite le funzioni** utilizzate dal parser per l’allocazione di memoria e la creazione del *parse tree*. Inoltre vi sono presenti altre funzioni utili all’accesso e alla manipolazione della symbol table.

e.g.

struct expression\* expression\_create(expression\_type kind, struct expression\* left, struct expression\* right, symbol\_type type){

struct expression\* e = calloc(4,sizeof(struct expression));

if(!e){

printf("Out of space!");

exit(1);

}

e->kind = kind;

e->left = left;

e->right = right;

e->type = type;

return e;

}

La funzione appena rappresenta permette di allocare, mediante la funzione *calloc()*, una *struct expression* (definita nel file AST.h) che sta a rappresentare una porzione dell’intero parse tree. Tale funzione quindi passa la tipologia di espressione al valore *kind* (EXP\_ASS, utile per effettuare una assegnazione), i componenti sinistro e destro dell’albero (*left* e *right*) ed infine il tipo di dato (e.g. *int, float*).

**2.3 AST.h**

Il file AST.h è un file noto come “header file” e funge da **supporto** per l’intero progetto scritto in linguaggio C. Esso infatti si occupa di dichiarare tutte le funzioni, le strutture e valori.

e.g.

typedef enum{

SYMBOL\_INT

} symbol\_type;

Questa definizione permette di generare un nuovo tipo (*symbol\_*type) che poi verrà incluso all’interno della struct symbol. Essa serve ad identificare un set finito di che una variabile all’interno della symbol table può assumere.

e.g.

typedef struct symbol{

char\* name;

symbol\_type type;

} symbol;

La definizione della struct symbol, necessaria alla generazione del vettore symTab, prevede, come detto in precedenza, il nome, tipo una variabile.

e.g.

**struct expression\* expression\_create(expression\_type kind, struct expression\* L, struct expression\* R, symbol\_type type);**

Tale riga di codice è necessaria alla dichiarazione della funzione poi definita all’interno della sezione 3 del parser.

**2.4 compiler.c**

Il quarto ed ultimo file è quello che effettua l’ultimo task necessario alla traduzione del codice in input al tool. Esso contiene una funzione principale detta *evaluate\_state()*, richiamata dalla regola grammaticale in prossimità della *root* che assume in input una struct di tipo statement e mediante un costrutto switch – case effettua la traduzione finale.

La diramazione switch – case serve infatti a “stampare” nel file di destinazione l’equivalente codice Python relativo ad una data porzione di parse tree. Tale operazione viene effettuata a seconda della tipologia di statement e ciascun *case* può o meno richiamare altre funzioni definite all’interno dello stesso file compiler.c

e.g.

void **eval\_statement**(struct statement\* statement){

if(!statement){

return;

}

**switch**(statement->kind){

**case STATEMENT\_EXP**:

if(!(statement->expression->left) || !( statement ->expression->right)){

yyerror("Invalid statement");

return;

}

**eval\_expression(**statement**->exp);**

fprintf(fp,"\n");

break;

In questo caso, uno statement di tipo espressione (STATEMENT\_EXP) richiama la funzone *eval\_expression(struct expression\* e)* atta ad valutare l’espressione passata.

e.g.

void **eval\_expression**(struct expression\* e){

if(!e){

return;

}

**eval\_expression(e->left);**

**switch**(e->kind){

**case EXPRESSION\_NAME:**

**fprintf(fp,"%s",e->s->name);**

break;

**case EXPRESSION\_ASS:**

f**printf(fp,"=");**

break;

**case EXPRESSION\_INT:**

**fprintf(fp,"%d",e->int\_lit);**

break;

}

**eval\_expression(e->right);**}

La funzione appena definita permette l’analisi di una struct di tipo expression e, in base alla tipologia di *expression*, il costrutto di diramazione richiama una diversa funzione f*printf()* utile alla stampa del corretto equivalente Python del codice C.

**3 Casi di Test**

Una volta terminata la scrittura del traduttore è stato possibile effettuare dei test con lo scopo di ottenere innanzitutto delle corrette traduzioni e successivamente dei messaggi di errori, volutamente immessi nel codice di test, per verificare la risposta del software in presenza di incongruenze.

I test sono stati eseguiti a seguito della generazione finale del tool, ottenuta mediante i seguenti comandi:

**bison -d parser.y**

**flex -l lexer.l**

**gcc -o a.out compiler.c lex.yy.c parser.tab.c -lfl**

**3.1 Test 1**

Il primo test verifica la correttezza dei seguenti costrutti:

- definizione e relativa assegnazione di valori a variabili dei tre tipi inclusi: *int, char e float*.

- applicazione del *casting* nel caso di *int e float*

- funzione di standard output relativa ad una stringa (*printf*)

- funzioni di standard output per la stampa di variabili numeriche

- 2 cicli *for* annidati

Il codice C relativo a tale test è il seguente:

**#include <stdio.h>**

**void main(){**

**int i;**

**int j = 25;**

**char\* str = "123 this is a string 123 ";**

**float f;**

**float t = 8.0;**

**float m = 3.21;**

**i = (int)t;**

**f = (float)j;**

**for(i=1;i<5;i++){**

**for(j=4;j<7;j++){**

**printf("nested FOR statements ");**

**}**

**}**

**f = f\*m;**

**printf("%s",str);**

**printf("%f",f);**

**}**

La traduzione di tale programma è stata realizzata considerando solo la sezione interna alla funzione main() e sfruttando un file testuale (test1.txt) dato in pasto al traduttore con il seguente comando linux:

**./a.out < test1.txt**

ottenendo il seguente output:

**i=0**

**j=25**

**str="123 this is a string 123 "**

**f=0.000000**

**t=8.000000**

**m=3.210000**

**i=int(t)**

**f=float(j)**

**for i in range(1,5):**

**for j in range(4,7):**

**print("nested FOR statements ")**

**f=f\*m**

**print(str)**

**print(f)**

**3.2 Test 2**

Analogamente al primo test, nel secondo vengono analizzati i seguenti punti:

- definizione di interi e di un *vettore*

- ciclo for per *caricamento vettore* da tastiera

- ciclo for per *stampa* valori caricati

- costrutto di diramazione di tipo *if - else if* con controllo su elemento del vettore

- espressione algebrica

- *stampa*

Il codice C realizzato per implementare tali funzionalità è:

**#include <stdio.h>**

**void main(){**

**int k;**

**int i;**

**int x[5];**

**int m = 10;**

**for(i = 0;i<5;i++){**

**scanf("%d",&x[i]);**

**}**

**for(i = 0;i<5;i++){**

**printf("%d",x[i]);**

**}**

**if(x[2]>7){**

**k=1234;**

**}**

**else if(x[2]<=7){**

**k = 5678;**

**}**

**k = k+m;**

**printf("%d",k);**

**}**

Allo stesso modo del test precedente è stato lanciato il comando **./a.out < test2.txt** per ottenere il file test2.py:

**k=0**

**i=0**

**x = []**

**m=10**

**for i in range(0,5):**

**x.append(int(input()))**

**for i in range(0,5):**

**print(x[i])**

**if x[2]>7:**

**k=1234**

**elif x[2]<=7:**

**k=5678**

**k=k+m**

**print(k)**

**3.3 Test con Errore**

Una volta verificato il giusto comportamento del traduttore in presenza di codice in input corretto, si è pensato di utilizzare il codice dei precedenti casi di test con immissione di errori al fine di generare messaggi d’errore.

**3.3.1 Test1 + errore**

Come prima caso di test d’errore si è pensato di:

* utilizzare una variabile di incremento differente da quelle utilizzate per il controllo delle iterazioni
* effettuare la stampa di una stringa utilizzando "%d" anzichè "%s"

**int i;**

**int j = 25;**

**char\* str = "123 this is a string 123 ";**

**float f;**

**float t = 8.0;**

**i = (int)t;**

**f = (float)j;**

**for(i=1;i<5;j++){**

**printf("nested FOR statements ");**

**}**

**printf("%d",str);**

**printf("%f",f);**

Il relativo output, generato dalla funzione interna yyerror(), è il seguente:

**<Line 16> For arguments have different variables**

**<Line 22> Variable is not an integer**

**3.3.2 Test2 + errore**

Nel secondo caso viene utilizzato il codice del Test2 con la differenza che prima dell’output finale viene effettuato un prodotto fra un intero ed un float a seguito di un casting.

**int k;**

**int i;**

**int x[5];**

**int m = 10;**

**for(i = 0;i<5;i++){**

**scanf("%d",&x[i]);**

**}**

**for(i = 0;i<5;i++){**

**printf("%d",x[i]);**

**}**

**if(x[2]>7){**

**k=1234;**

**}**

**else if(x[2]<=7){**

**k = 5678;**

**}**

**m = (float)m;**

**k = k+m;**

**printf("%d",k);**

In questo caso, l’output ottenuto è:

**<Line 27> Type of terms is not the same**