POLITECNICO DI BARI DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

Formal Languages and Compilers

C++ to Python Transpiler

A.A.	2021/2022
COURSE	Formal Languages and Compilers (Prof. Floriano Scioscia)
STUDENTS	Vincenzo Ancona, Danilo Danese

Introduzione

Il progetto realizzato consiste nella realizzazione di un source to source compiler (o transpiler) da una restrizione del linguaggio di programmazione orientato agli oggetti C++ al linguaggio Python.

Tale compilatore è stato realizzato in linguaggio C utilizzando il generatore di scanner automatico Flex (Fast Lexical Analyzer Generator) per l'analisi del lessico e il generatore di parser GNU Bison per l'analisi sintattica, i controlli semantici sono effettuati all'interno del parser.

È stata inoltre utilizzata la libreria uthash per l'implementazione della symbol table, gestita come hash table.

La restrizione del linguaggio sorgente C++ prevede le seguenti istruzioni da tradurre nel linguaggio di arrivo:

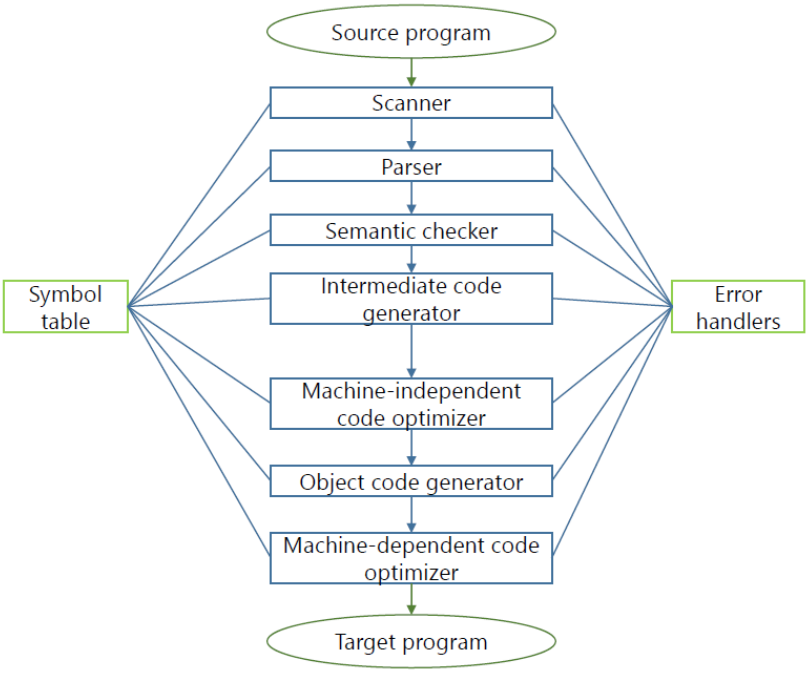
- Quattro data types: variabili aventi come valori numeri decimali floating point a singola precisione (float), numeri interi (int), stringhe alfanumeriche (string) e variabili booleane (bool); la gestione dei numeri interi è stata aggiunta alle specifiche concordate in quanto in linguaggio C++ la funzione main() può restituire solo valori interi;
- Operatori aritmetici per somma (+), sottrazione (-), moltiplicazione (*), divisione (/);
- Operatori di confronto quali <, <=, >, >=;
- Operatori logici AND e OR;
- L'istruzione di diramazione if else;
- L'istruzione di iterazione for;
- L'istruzione per la gestione degli input da tastiera cin;
- L'istruzione per la gestione degli output da tastiera cout;

Oltre alle istruzioni di cui sopra, il transpiler oggetto di questo progetto gestisce le seguenti funzionalità:

- Dichiarazione di variabili;
- Operazioni di assegnazione su variabili;
- Definizione e chiamata di funzioni;
- Espressioni tra variabili o chiamate di funzioni;

- Definizione e istanziazione di classi con relativa chiamata di attributi e metodi;
- Ereditarietà singola tra classi.

Struttura di un compilatore



Fasi di un compilatore

Un compilatore può essere scomposto in due macro-fasi: una detta di analisi (front-end) e una di sintesi (back-end).

La prima fase si divide a sua volta in:

- **analisi lessicale**: chiamata anche fase di scanning, in cui viene letto il flusso di caratteri del programma sorgente e, dopo averlo diviso in lessemi, viene restituita una sequenza di token <nome, valore>;
- **analisi sintattica**: chiamata anche fase di parsing, in cui, data una sequenza di token ricevuti, si controlla se essa rispetta la grammatica del linguaggio definita. Può lavorare con approccio top-down o bottom-up e produce l'albero sintattico o Abstract Syntax Tree;

- **analisi semantica**: in tale fase si utilizza l'Abstract Syntax Tree e le informazioni salvate nella symbol table per effettuare i controlli semantici sul linguaggio sorgente;
- **generazione di codice intermedio**: qui viene generata una rappresentazione intermedia del codice sorgente al fine di facilitarne la traduzione.

La finalità di questa prima fase è la generazione di un codice intermedio, il quale costituisce l'input per la fase successiva di sintesi. Essa si divide in:

- **ottimizzatore di codice machine-dependent/independent**: questa sottofase è dedicata al miglioramento del codice intermedio in base a metriche come ad esempio la velocità di compilazione;
- **generatore di codice oggetto**: questa sottofase riceve il codice intermedio ed effettua la vera e propria traduzione nel linguaggio finale, il quale può essere il codice macchina o un altro linguaggio ad alto livello come nel nostro caso.

Altri aspetti importanti nella realizzazione di un compilatore riguardano la gestione della symbol table e la gestione degli errori, entrambe vengono infatti effettuate in ogni fase del processo di compilazione.

La symbol table è una tabella che contiene le informazioni su tutti gli elementi simbolici del linguaggio, come ad esempio i nomi delle variabili del codice da tradurre e altri attributi utili. La scelta migliore per l'implementazione di tale tabella è utilizzare una struttura dati di tipo hash table.

La gestione degli errori è infine un passaggio importante in quanto è necessario definire il comportamento del compilatore in presenza di errori nel codice sorgente.

Struttura del progetto

Di seguito viene descritta la struttura del progetto fornendo prima una breve descrizione di ciascun file presente nella cartella per poi analizzare ciascuna fase del lavoro svolto.

Componenti

I files che compongono il progetto sono elencati di seguito:

- **scanner.l**: questo file è utilizzato per fornire a Flex le istruzioni per la generazione dello scanner;
- **parser.y**: questo file contiene le istruzioni per la generazione del parser da fornire a Bison quali la grammatica del linguaggio C++, le istruzioni per la gestione degli scope e per la gestione di tutti i controlli semantici;
- **ast.h**: in questo file vi è la definizione di tutti i nodi che compongono l'Abstract Syntax Tree e altre strutture necessarie alla gestione dei simboli e dei tipi di dato;
- **symtab.h**: in questo file si definisce la struttura di ciascun simbolo, della symbol table e tutte le funzioni di gestione ad essi relativi (funzione per aggiungere/rimuovere un nuovo simbolo, ecc);
- **uthash.h**: questo header file contiene la libreria con tutte le funzioni necessarie per gestire la symbol table come struttura dati di tipo hash;
- **translation.h**: in questo file vengono definite tutte le funzioni per percorrere e tradurre i nodi dell' Abstract Syntax Tree in linguaggio Python.

Per ciascuna tipologia di istruzione da tradurre sono inoltre presenti due file: il primo contiene codice in linguaggio C++ scritto correttamente mentre il secondo contiene codice volutamente errato in modo da valutare la gestione degli errori da parte del transpiler.

Il codice in linguaggio Python di output viene stampato nel file "python_transpiled.py" mentre la stampa degli errori sintattici e semantici viene effettuata sul terminale in cui è stato avviato il file eseguibile.

Analisi lessicale (fase di scanning)

La prima operazione caratteristica della macro-fase di analisi consiste nello scanning, in cui il flusso di caratteri del programma sorgente viene analizzato per produrre una sequenza di token su ogni lessema riconosciuto.

Di seguito analizziamo la sua struttura:

Contenuto del file scanner.l

Come si può notare dalle figure di cui sopra, nel nostro file scanner.y andiamo innanzitutto a definire le espressioni regolari che devono essere riconosciute in input.

Mediante l'aggiunta, oltre allo stato init presente di default, degli stati comment, commentsl, library e namespacestd è stato implementato il riconoscimento delle linee contenenti commenti, commenti multilinea, import di librerie e la linea che segnala l'utilizzo del namespace std. Poiché sostanzialmente inutili al fine di generare codice correttamente eseguibile in linguaggio Python, tali righe vengono semplicemente ignorate in fase di traduzione.

In seguito vengono definiti i token che devono essere riconosciuti in fase di scanning, i quali vengono ritornati dopo aver stampato su terminale il relativo avviso quando identificati.

La gestione della numerazione delle righe di codice viene effettuata utilizzando l'opzione yylineno.

Nel caso in cui vi siano errori di tipo lessicale, ovvero venga rilevata un'espressione regolare nel file di input non ammessa dal linguaggio, essi saranno sempre di tipo bloccante e impediranno la generazione del file tradotto.

Analisi sintattica (fase di parsing)

Successivamente è stata effettuata l'analisi sintattica, il cui scopo è, come detto precedentemente, la validazione della sequenza di token ricavata in fase di scanning rispetto ad una grammatica relativa al linguaggio di partenza.

Per tale scopo è stato quindi utilizzato il file parser.y in input al generatore di parser Bison.

In tale file sono state definite tutte le produzioni relative alla grammatica della restrizione di C++ in Backus-Naur Form.

Esempi di non terminal definiti nella grammatica del linguaggio sono mostrati nelle figure seguenti:

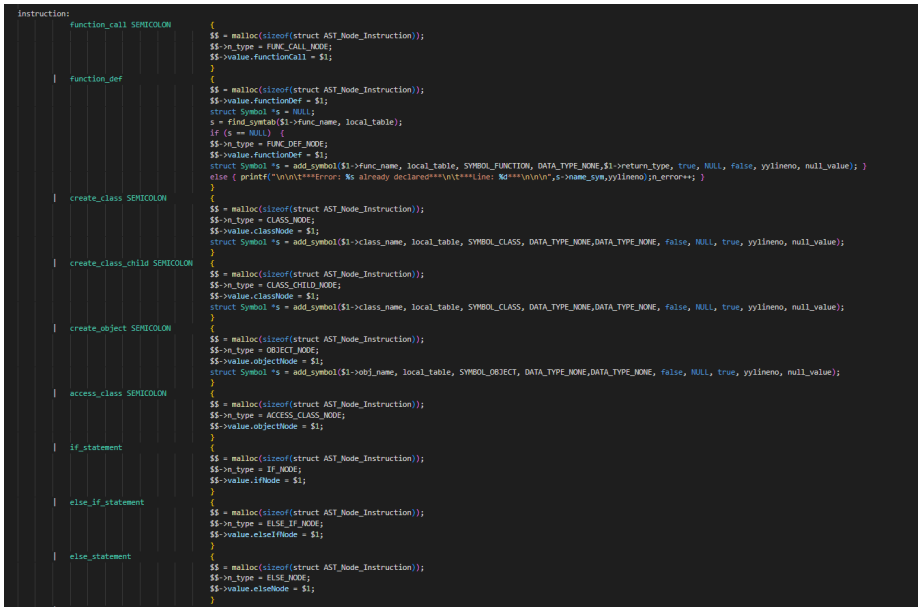
```
%start program
%%

program:   { scope_enter(); }   statements   { root = $2; scope_exit(); };


statements:
          instruction statements            {
                                                $$ = malloc(sizeof(struct AST_Node_Statements));
                                                $$->n_type = STATEMENT_NODE;
                                                $$->left = $1;
                                                $$->right = $2;
                                            }
        | instruction                       {
                                                $$ = malloc(sizeof(struct AST_Node_Statements));
                                                $$->n_type = STATEMENT_NODE;
                                                $$->left = $1;
                                                $$->right = NULL;
                                            };
```

Non-terminal "program" e "statements"

Il primo non-terminal, “program”, è stato definito per gestire l’avvio dell’intero programma di input. Una volta in “program”, verrà chiamata la funzione “scope_enter()” definita in basso tramite una mid-rule action al fine di definire un nuovo scope per il programma.

Una volta riconosciuto il non-terminal “statements”, contenente le produzioni per la gestione di uno o più “instruction”, lo si imposta come nodo radice dell’ AST assegnandolo alla variabile root (puntatore al nodo dedicato agli statements). Terminata la lettura di tutte le istruzioni del programma, viene chiamata la funzione “scope_exit()” per uscire dallo scope. Poiché per scelta progettuale ad ogni scope viene assegnata una symbol table dedicata, all’uscita di ogni scope la relativa symbol table viene eliminata.

In generale, si è scelto di denominare i non-terminal dedicati alla gestione di istruzioni ripetute con il prefisso “multi_”.

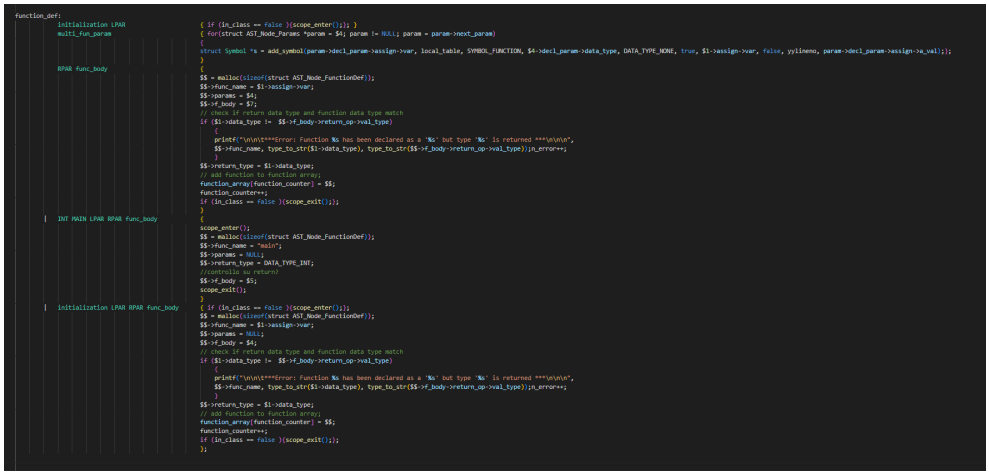


```
instruction:
        function_call SEMICOLON
                {
                $$ = malloc(sizeof(struct AST_Node_Instruction));
                $$->n_type = FUNC_CALL_NODE;
                $$->value.functionCall = $1;
                }
    |   function_def
                {
                $$ = malloc(sizeof(struct AST_Node_Instruction));
                $$->value.functionDef = $1;
                struct Symbol *s = NULL;
                s = find_symtab($1->func_name, local_table);
                if (s == NULL)  {
                $$->n_type = FUNC_DEF_NODE;
                $$->value.functionDef = $1;
                struct Symbol *s = add_symbol($1->func_name, local_table, SYMBOL_FUNCTION, DATA_TYPE_NONE,$1->return_type, true, NULL, false, yylineno, null_value); }
                else { printf("\n\n\t***Error: %s already declared***\n\t***Line: %d***\n\n\n",s->name_sym,yylineno);n_error++; }
                }
    |   create_class SEMICOLON
                {
                $$ = malloc(sizeof(struct AST_Node_Instruction));
                $$->n_type = CLASS_NODE;
                $$->value.classNode = $1;
                struct Symbol *s = add_symbol($1->class_name, local_table, SYMBOL_CLASS, DATA_TYPE_NONE,DATA_TYPE_NONE, false, NULL, true, yylineno, null_value);
                }
    |   create_class_child SEMICOLON
                {
                $$ = malloc(sizeof(struct AST_Node_Instruction));
                $$->n_type = CLASS_CHILD_NODE;
                $$->value.classNode = $1;
                struct Symbol *s = add_symbol($1->class_name, local_table, SYMBOL_CLASS, DATA_TYPE_NONE,DATA_TYPE_NONE, false, NULL, true, yylineno, null_value);
                }
    |   create_object SEMICOLON
                {
                $$ = malloc(sizeof(struct AST_Node_Instruction));
                $$->n_type = OBJECT_NODE;
                $$->value.objectNode = $1;
                struct Symbol *s = add_symbol($1->obj_name, local_table, SYMBOL_OBJECT, DATA_TYPE_NONE,DATA_TYPE_NONE, false, NULL, true, yylineno, null_value);
                }
    |   access_class SEMICOLON
                {
                $$ = malloc(sizeof(struct AST_Node_Instruction));
                $$->n_type = ACCESS_CLASS_NODE;
                $$->value.objectNode = $1;
                }
    |   if_statement
                {
                $$ = malloc(sizeof(struct AST_Node_Instruction));
                $$->n_type = IF_NODE;
                $$->value.ifNode = $1;
                }
    |   else_if_statement
                {
                $$ = malloc(sizeof(struct AST_Node_Instruction));
                $$->n_type = ELSE_IF_NODE;
                $$->value.elseIfNode = $1;
                }
    |   else_statement
                {
                $$ = malloc(sizeof(struct AST_Node_Instruction));
                $$->n_type = ELSE_NODE;
                $$->value.elseNode = $1;
                }
```

Non-terminal “instruction”

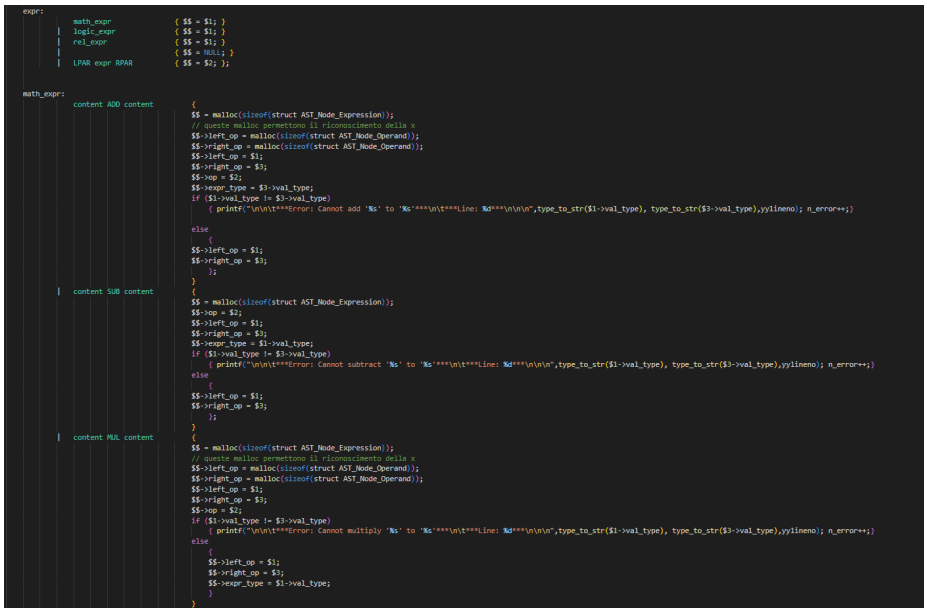
Come si può notare, ogni volta che viene individuata una nuova tipologia di istruzione, il relativo simbolo viene aggiunto alla symbol table tramite la funzione add_symbol() e, nel caso di funzioni e inizializzazioni, viene effettuato un controllo sulla symbol table per verificare che la variabile non sia stata già dichiarata tramite la funzione find_symbol()

Nelle immagini seguenti si riportano alcuni esempi di non terminal e relative azioni per la gestione di alcune funzionalità del linguaggio, quali dichiarazioni e assegnazioni di variabili, definizione e chiamata di funzioni e classi, gestione degli if, dei cicli for e delle funzioni di input/output.



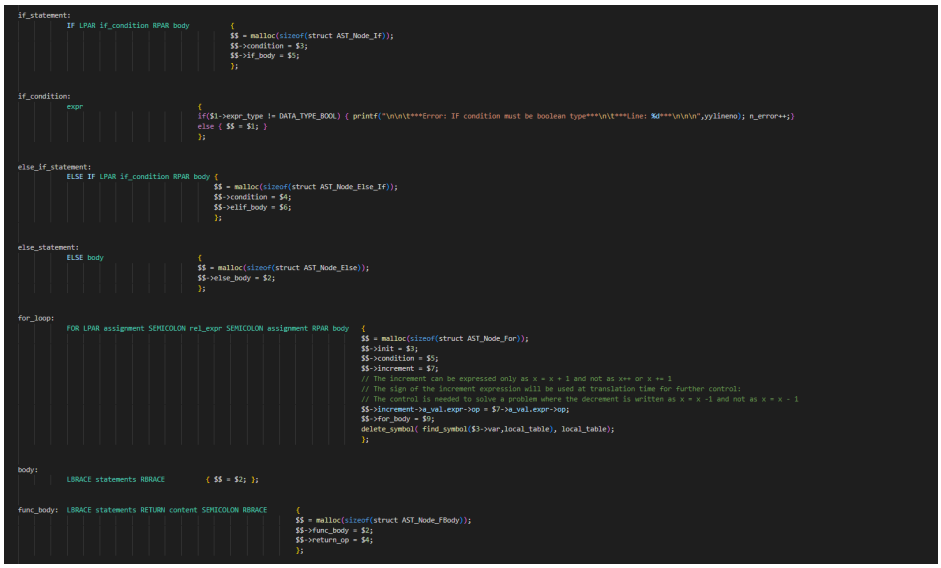
```
function_def:
        initialization LPAR                 { if (in_class == false ){scope_enter();}; }
        multi_fun_param                     { for(struct AST_Node_Params *param = $4; param != NULL; param = param->next_param)
                                            {
                                            struct Symbol *s = add_symbol(param->decl_param->assign->var, local_table, SYMBOL_FUNCTION, $4->decl_param->data_type, DATA_TYPE_NONE, true, $1->assign->var, false, yylineno, param->decl_param->assign->a_val);};
                                            }
        RPAR func_body                      {
                                            $$ = malloc(sizeof(struct AST_Node_FunctionDef));
                                            $$->func_name = $1->assign->var;
                                            $$->params = $4;
                                            $$->f_body = $7;
                                            // check if return data type and function data type match
                                            if ($1->data_type !=  $$->f_body->return_op->val_type)
                                                {
                                                printf("\n\n\t***Error: Function %s has been declared as a '%s' but type '%s' is returned ***\n\n\n",
                                                $$->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] = $$;
                                            function_counter++;
                                            if (in_class == false ){scope_exit();};
                                            }
    |   INT MAIN LPAR RPAR func_body        {
                                            scope_enter();
                                            $$ = malloc(sizeof(struct AST_Node_FunctionDef));
                                            $$->func_name = "main";
                                            $$->params = NULL;
                                            $$->return_type = DATA_TYPE_INT;
                                            //controllo su return?
                                            $$->f_body = $5;
                                            scope_exit();
                                            }
    |   initialization LPAR RPAR func_body  { if (in_class == false ){scope_enter();};
                                            $$ = malloc(sizeof(struct AST_Node_FunctionDef));
                                            $$->func_name = $1->assign->var;
                                            $$->params = NULL;
                                            $$->f_body = $4;
                                            // check if return data type and function data type match
                                            if ($1->data_type !=  $$->f_body->return_op->val_type)
                                                {
                                                printf("\n\n\t***Error: Function %s has been declared as a '%s' but type '%s' is returned ***\n\n\n",
                                                $$->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();};
                                            };
```

Non-terminal “function_def”



```
expr:
        math_expr               { $$ = $1; }
    |   logic_expr              { $$ = $1; }
    |   rel_expr                { $$ = $1; }
    |                           { $$ = NULL; }
    |   LPAR expr RPAR          { $$ = $2; };


math_expr:
        content ADD content         {
                                    $$ = malloc(sizeof(struct AST_Node_Expression));
                                    // questo malloc permettono il riconoscimento della x
                                    $$->left_op = malloc(sizeof(struct AST_Node_Operand));
                                    $$->right_op = malloc(sizeof(struct AST_Node_Operand));
                                    $$->left_op = $1;
                                    $$->right_op = $3;
                                    $$->op = $2;
                                    $$->expr_type = $3->val_type;
                                    if ($1->val_type != $3->val_type)
                                        { printf("\n\n\t***Error: Cannot add '%s' to '%s'***\n\t***Line: %d***\n\n\n",type_to_str($1->val_type), type_to_str($3->val_type),yylineno); n_error++;}

                                    else
                                        {
                                    $$->left_op = $1;
                                    $$->right_op = $3;
                                        };
                                    }
    |   content SUB content         {
                                    $$ = malloc(sizeof(struct AST_Node_Expression));
                                    $$->op = $2;
                                    $$->left_op = $1;
                                    $$->right_op = $3;
                                    $$->expr_type = $1->val_type;
                                    if ($1->val_type != $3->val_type)
                                        { printf("\n\n\t***Error: Cannot subtract '%s' to '%s'***\n\t***Line: %d***\n\n\n",type_to_str($1->val_type), type_to_str($3->val_type),yylineno); n_error++;}
                                    else
                                        {
                                    $$->left_op = $1;
                                    $$->right_op = $3;
                                        };
                                    }
    |   content MUL content         {
                                    $$ = malloc(sizeof(struct AST_Node_Expression));
                                    // questo malloc permettono il riconoscimento della x
                                    $$->left_op = malloc(sizeof(struct AST_Node_Operand));
                                    $$->right_op = malloc(sizeof(struct AST_Node_Operand));
                                    $$->left_op = $1;
                                    $$->right_op = $3;
                                    $$->op = $2;
                                    if ($1->val_type != $3->val_type)
                                        { printf("\n\n\t***Error: Cannot multiply '%s' to '%s'***\n\t***Line: %d***\n\n\n",type_to_str($1->val_type), type_to_str($3->val_type),yylineno); n_error++;}
                                    else
                                        {
                                        $$->left_op = $1;
                                        $$->right_op = $3;
                                        $$->expr_type = $1->val_type;
                                        }
                                    }
```

Non-terminal “expr”



```
if_statement:
        IF LPAR if_condition RPAR body          {
                                                $$ = malloc(sizeof(struct AST_Node_If));
                                                $$->condition = $3;
                                                $$->if_body = $5;
                                                };


if_condition:
        expr                                    {
                                                if($1->expr_type != DATA_TYPE_BOOL) { printf("\n\n\t***Error: IF condition must be boolean type***\n\t***Line: %d***\n\n\n",yylineno); n_error++;}
                                                else { $$ = $1; }
                                                };


else_if_statement:
        ELSE IF LPAR if_condition RPAR body {
                                                $$ = malloc(sizeof(struct AST_Node_Else_If));
                                                $$->condition = $4;
                                                $$->elif_body = $6;
                                                };


else_statement:
        ELSE body                               {
                                                $$ = malloc(sizeof(struct AST_Node_Else));
                                                $$->else_body = $2;
                                                };


for_loop:
        FOR LPAR assignment SEMICOLON rel_expr SEMICOLON assignment RPAR body   {
                                                                                $$ = malloc(sizeof(struct AST_Node_For));
                                                                                $$->init = $3;
                                                                                $$->condition = $5;
                                                                                $$->increment = $7;
                                                                                // The increment can be expressed only as x = x + 1 and not as x++ or x += 1
                                                                                // The sign of the increment expression will be used at translation time for further control:
                                                                                // The control is needed to solve a problem where the decrement is written as x = x -1 and not as x = x - 1
                                                                                $$->increment->a_val.expr->op = $7->a_val.expr->op;
                                                                                $$->for_body = $9;
                                                                                delete_symbol( find_symbol($3->var,local_table), local_table);
                                                                                };


body:
        LBRACE statements RBRACE        { $$ = $2; };


func_body:  LBRACE statements RETURN content SEMICOLON RBRACE       {
                                                                    $$ = malloc(sizeof(struct AST_Node_FBody));
                                                                    $$->func_body = $2;
                                                                    $$->return_op = $4;
                                                                    };
```

Altri esempi di non-terminal

In generale, quando la sequenza di token identificata da ciascun non-terminal viene riconosciuta all'interno delle azioni associate alle produzioni, si alloca dapprima la struttura dati relativa al nodo corrispondente per poi andare ad effettuare le assegnazioni su ciascun puntatore della struttura in maniera tale da riempire lo spazio allocato.

Per la gestione di classi, oggetti e funzioni si è scelto di operare nel modo seguente: per ognuno di essi viene inizializzato un array puntatore al relativo nodo dell' AST con il relativo contatore e, dopo aver riempito ciascuna struttura dati allocata, la si inserisce nell'array di appartenenza per poi incrementarne il contatore. Tale operazione è necessaria successivamente sia per la gestione dell'ereditarietà singola durante la chiamata di metodi e attributi di classi e oggetti sia in fase di traduzione.

All'interno del file parser.y sono state inoltre inserite alcune funzioni di supporto oltre quelle per la gestione degli scope, tra cui le funzioni type_to_str() e str_to_type(), utilizzate per convertire i nomi dei data type in stringa e viceversa;

Nel momento in cui il compilatore rileverà errori di sintassi, esso li stamperà su terminale indicando il numero della rispettiva riga tramite le opzioni yyerror e yylineno. Tali tipi di errori saranno sempre di tipo bloccante.

Abstract Syntax Tree

All'interno del file "ast.h" sono definiti i nodi dell' Abstract Syntax Tree. Essa è una rappresentazione ad albero della struttura grammaticale del linguaggio ed è necessaria al fine di effettuare la traduzione del linguaggio sorgente e la fase di analisi semantica.

L'albero che si è realizzato possiede il nodo AST_Node_Statements come nodo radice. Esso rappresenta un nodo generico e comprende:

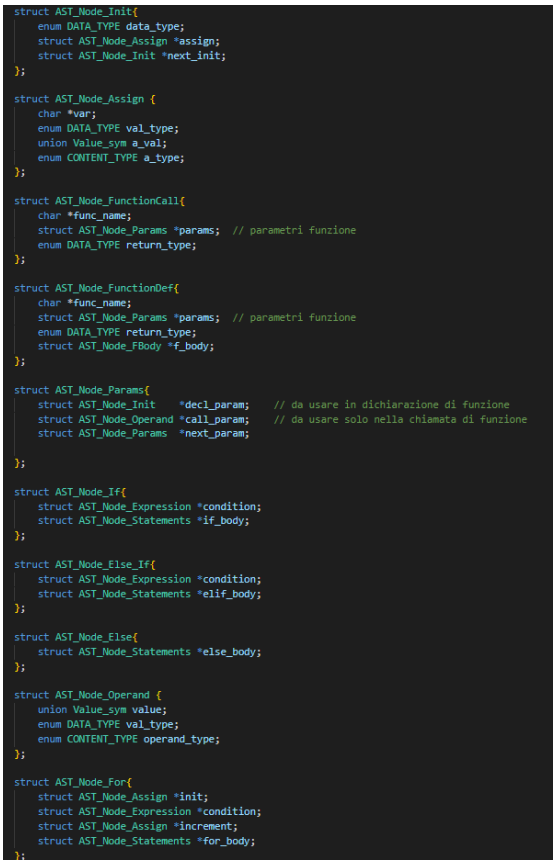
- **n_type**: unione che indica il tipo di nodo allocato;
- ***left**: puntatore al nodo AST_Node_Instruction che rappresenta il nodo corrente da esaminare (allocare in memoria o tradurre);
- ***right**: puntatore dello stesso tipo del nodo radice che si riferisce al prossimo nodo da esaminare.

Il nodo AST_Node_Instruction è invece strutturato nel modo seguente:

- **n_type**;
- **value**: unione a yystype dei nodi da noi definiti e puntatore a char per definire il tipo dei terminal token.

Per ogni istruzione, la combinazione tra n_type e value viene fornita nel non-terminal instruction al fine di identificare il nodo corretto da allocare.

Nell' immagine seguente si fornisce qualche esempio di nodi dell'AST per le varie tipologie di istruzioni da implementare:



```c
struct AST_Node_Init{
    enum DATA_TYPE data_type;
    struct AST_Node_Assign *assign;
    struct AST_Node_Init *next_init;
};

struct AST_Node_Assign {
    char *var;
    enum DATA_TYPE val_type;
    union Value_sym a_val;
    enum CONTENT_TYPE a_type;
};

struct AST_Node_FunctionCall{
    char *func_name;
    struct AST_Node_Params *params;  // parametri funzione
    enum DATA_TYPE return_type;
};

struct AST_Node_FunctionDef{
    char *func_name;
    struct AST_Node_Params *params;  // parametri funzione
    enum DATA_TYPE return_type;
    struct AST_Node_FBody *f_body;
};

struct AST_Node_Params{
    struct AST_Node_Init    *decl_param;    // da usare in dichiarazione di funzione
    struct AST_Node_Operand *call_param;    // da usare solo nella chiamata di funzione
    struct AST_Node_Params  *next_param;

};

struct AST_Node_If{
    struct AST_Node_Expression *condition;
    struct AST_Node_Statements *if_body;
};

struct AST_Node_Else_If{
    struct AST_Node_Expression *condition;
    struct AST_Node_Statements *elif_body;
};

struct AST_Node_Else{
    struct AST_Node_Statements *else_body;
};

struct AST_Node_Operand {
    union Value_sym value;
    enum DATA_TYPE val_type;
    enum CONTENT_TYPE operand_type;
};

struct AST_Node_For{
    struct AST_Node_Assign *init;
    struct AST_Node_Expression *condition;
    struct AST_Node_Assign *increment;
    struct AST_Node_Statements *for_body;
};
```

Nodi dell'Abstract Syntax Tree

Symbol Table

La symbol table è, come detto precedentemente, una struttura dati il cui compito è quello di salvare tutti i simboli di interesse del codice in input con le relative informazioni associate.

Il modo migliore per realizzarla dal punto di vista implementativo è quello di costruire una hash table in quanto la durata temporale della fase di ricerca dei simboli su una struttura dati di questo tipo è di molto inferiore rispetto ad altri tipi di strutture dati e pertanto evita il verificarsi di un collo di bottiglia tra la tabella e gli elementi che ne richiedono l'accesso.

L'implementazione di tale tabella per questo progetto è stata definita nel file symtab.h ed ha richiesto l'utilizzo della libreria "uthash" come supporto per la gestione degli hash.

In particolare, si sono definite due strutture, rispettivamente una dedicata alla gestione degli scope (avendo una tabella separata per ciascuno scope) e una dedicata al simbolo con le relative informazioni.

La prima struttura è composta dai seguenti campi:

- ***symtab**: puntatore alla tabella attuale;
- **scope**: identificativo intero per lo scope;
- ***next**: puntatore alla tabella relativa allo scope più esterno.

Ciascuna tabella tiene traccia del simbolo conservando informazioni relative ai seguenti campi in ciascun record:

- ***name_sym**: puntatore a char che indica il nome del simbolo, utilizzato come chiave della tabella;
- **symbol_type**: indica il tipo di simbolo (variabile, contenuto, funzione, classe, oggetto o parametro di funzione);
- **data_type**: indica il tipo di dato associato al simbolo (float, int, string, bool o indefinito);
- **ret_type**: solo nel caso in cui il tipo di simbolo sia una funzione, indica il tipo ritornato dalla stessa; in caso contrario, esso è impostato a indefinito (DATA_TYPE_NONE);
- **is_function**: flag che indica se il simbolo rappresenta una funzione;
- **is_class**: flag che indica se il simbolo rappresenta una classe;
- **param_function_name**: char che, quando non nullo, è usato per segnalare se una variabile è un parametro di una funzione e il nome della funzione di cui tale variabile è parametro;
- **line_num**: intero che indica il numero di riga del codice sorgente in cui il simbolo compare;
- **value_sym**: unione che indica il valore assunto dal simbolo. Può essere un semplice puntatore a char, un puntatore al nodo espressione, chiamata di funzione o assegnazione;

Sono state create inoltre alcune funzioni per la gestione dei simboli e delle tabelle, come find_symbol() e add_symbol() già citate in precedenza. Altre funzioni per la gestione delle tabelle sono new_symtab() e delete_symtab(), essenziali per l'entrata e uscita da un nuovo scope.

Analisi Semantica

Dopo la fase di scanning e di parsing, la fase successiva è quella di analisi semantica. Essa consiste nell'effettuare i controlli sulla correttezza semantica del programma ricevuto in input dal transpiler, ovvero nel verificare che non vi siano errori di significato.

Di seguito sono indicati alcuni dei controlli semantici che sono stati effettuati all'interno del file parser.y.

Nel caso in cui siano presenti degli errori semantici, il transpiler stamperà sul terminale una spiegazione del tipo di errore generato insieme al rispettivo numero di riga.

In presenza di errori semantici, sintattici o lessicali, il file di output non verrà generato.

Controlli sulle inizializzazioni e assegnazioni

I controlli semantici sulle inizializzazioni vengono effettuati al fine di generare errori nel caso in cui sia inizializzata una variabile già presente all'interno dello scope.

Per quanto riguarda le assegnazioni, viene verificato che esse vengano effettuate tra membri aventi lo stesso tipo di dato.

Controlli sulle espressioni

Sulle espressioni, viene verificato che non vengano effettuate operazioni di somma, sottrazione, moltiplicazione e divisione tra membri aventi data type differente.

Viene inoltre verificato che non si possano effettuare divisioni per 0.

Al fine di poter gestire i numeri float o int negativi all'interno di una operazione, viene effettuato un controllo in cui anche se l'utente ha scritto l'espressione come "A -B" invece di "A - B", l'espressione risulta comunque corretta.

Controlli sulle funzioni

Per quanto riguarda la chiamata di funzioni, mediante il metodo check_function_call() vengono effettuati controlli sull'effettiva presenza all'interno della symbol table dell'identificativo della funzione, la quale dovrà quindi già essere stata definita. Inoltre, viene verificata la coerenza in numero e tipo dei parametri inseriti con quelli definiti nel rispettivo prototipo.

In merito alla definizione di funzione, viene effettuato un controllo in modo che il data type relativo alla funzione sia uguale al data type ritornato.

Controlli sugli if

Per quanto riguarda l'istruzione di diramazione if, viene effettuato un controllo sulla condizione che deve verificarsi in quanto essa dovrà essere di tipo booleano.

Controlli sul for loop

Infine, nell'istruzione del for loop viene verificato che il tipo di dato della variabile inizializzata per l'iterazione sia esclusivamente intero in quanto la funzione scelta per la traduzione del ciclo for in linguaggio Python (range()) non permette l'utilizzo di valori di tipo float.

Generazione del codice

La generazione del codice tradotto viene effettuata attraversando i nodi dell'Abstract Syntax Tree: tale struttura dati costituisce infatti la rappresentazione intermedia del programma di partenza.

La traduzione viene effettuata attraversando i nodi dell'AST seguendo la strategia pre-order, ovvero attraversando prima il sottoalbero sinistro e poi il sottoalbero destro.

Tutte le funzioni dedicate alla traduzione sono definite nel file "translation.h" e hanno il prefisso "translate_". Esse vengono poi richiamate dalla funzione "traverse()" dedicata all'attraversamento dei nodi nel file parser.y.

Classi ed ereditarietà singola

Una delle caratteristiche fondamentali del linguaggio C++ è l'ereditarietà singola. Tale meccanismo prevede che, data una classe padre, alcuni attributi o metodi appartenenti ad essa possano essere passati alle proprie classi figlie, dando quindi la possibilità di accedervi tramite la loro istanziazione. Nello specifico, si è scelto di far ereditare a classi figlie attributi e metodi appartenenti alla sezione public della classe padre.

Nodi dell'AST

Al fine di poter tradurre questa caratteristica dal linguaggio C++ al linguaggio Python, sono stati creati quattro nodi nell'AST:

AST_Node_Class

Ciascun nodo AST_Node_Class è stato dotato di quattro puntatori:

- ***class_name** (puntatore a char che indica il nome della classe);
- ***c_body** (puntatore alla struttura CBody che indica il corpo della classe);
- ***parent_class** (puntatore al nodo AST_Node_Class che indica la classe genitore del nodo classe di partenza);
- ***parent_class_public** (puntatore al nodo AST_Node_Parent_Public che indica la parte public del body del genitore).

AST_Node_CBody

Ciascun nodo AST_Node_CBody è una struttura creata per definire il body di una classe. Essa possiede due puntatori:

- ***pri_body** (puntatore al nodo AST_Node_Statements che indica la parte private del body di una classe);
- ***pub_body** (puntatore al nodo AST_Node_Statements che indica la parte public del body di una classe).

AST_Node_Parent_Public

Ciascun nodo AST_Node_Parent_Public è una struttura creata per gestire la parte public del nodo genitore di una classe che dovrà essere ereditata. Essa possiede due puntatori:

- ***parent_pub_body** (puntatore al nodo AST_Node_Statements che indica la parte public del body della classe genitore);
- ***next_parent_public** (puntatore al nodo AST_Node_Parent_Public che indica la parte public della prossima classe genitore, utile in caso di presenza di classi "nonno".

AST_Node_Object

Ciascun nodo AST_Node_Object è una struttura creata per gestire gli oggetti, ovvero le istanziazioni di classi precedentemente definite. Essa possiede i seguenti puntatori:

- ***obj_name** (puntatore a char che indica il nome dell'oggetto);
- **access_value** (puntatore a Value_sym che serve per fornire all'oggetto il valore dell'attributo o del metodo a cui l'oggetto deve accedere);
- ***obj_class** (puntatore al nodo AST_Node_Class che indica la classe di cui l'oggetto rappresenta l'istanza);
- **access_type** (puntatore ad ACCESS_TYPE che indica il tipo di contenuto della classe cui l'oggetto deve accedere).

Tali nodi vengono allocati all'interno delle regole dei non-terminal dedicati alla gestione delle classi. Essi sono:

- **create_object**: esso è dedicato all'istruzione in C++ per l'istanziazione di un nuovo oggetto;
- **access_class**: esso è dedicato alle istruzioni per l'accesso da parte di oggetti ad attributi o metodi di una determinata classe;
- **create_class**: esso è dedicato alle istruzioni per la definizione di una nuova classe;
- **create_child_class**: esso è dedicato alle istruzioni per la definizione di una classe derivata.

Infine, sempre all'interno del file parser.y, è stata definita la funzione search_class_body() che serve ad attraversare gli statements all'interno del body di un classe. Essa riceve in input il nome della variabile da cercare e il puntatore relativo al body della classe e cicla soltanto tra le initialization, assegnazioni e le definizioni di funzioni. Successivamente, la funzione restituisce un valore booleano pari a true se l'attributo o il metodo richiesto è ritrovato all'interno del body della classe e viene utilizzata per effettuare i controlli semantici sull'effettiva esistenza della classe che si vuole istanziare o del relativo attributo/metodo.

Controlli semantici sulle classi

Per quanto riguarda classi e oggetti, i controlli semantici che vengono effettuati riguardano la verifica dell'effettiva esistenza e istanziazione di una classe cui si vuole accedere e la verifica che l'accesso a metodi e attributi di una classe sia possibile esclusivamente tramite oggetti che istanziano la classe stessa o classi figlie: in caso contrario verrà restituito un messaggio di errore che segnalerà l'impossibilità di accedervi. Viene inoltre verificato che una classe non sia istanziata più volte.

Generazione del codice per le classi

La definizione delle funzioni di traduzione del codice a partire dai nodi dell'AST avviene, come per tutte le altre tipologie di istruzioni e strutture, all'interno del file "translation.h".

La traduzione di classi e oggetti avviene, sempre sul file di output "python_transpiled.py", tramite l'utilizzo delle seguenti funzioni:

translate_class(): funzione per la traduzione della definizione di una classe;

translate_class_child(): funzione per la traduzione della definizione di una classe figlia;

translate_object(): funzione per la traduzione dell'istanziazione di una classe;

translate_access_class(): funzione per la traduzione delle istruzioni per l'accesso ad attributi e metodi di una classe;

translate_public_class_func_def(): funzione per la traduzione della definizione di metodi all'interno della parte public del body di una classe;

translate_private_class_func_def(): funzione per la traduzione della definizione di metodi all'interno della parte private del body di una classe. Tale funzione è stata creata in maniera distinta dalla precedente per agevolare la traduzione delle definizioni di funzione nella parte private di una classe poiché vi sono delle differenze di sintassi in traduzione.

translate_object_assign(): funzione per la traduzione dell'istruzione per l'assegnazione di un nuovo valore a un attributo di una classe.

Le funzioni di traverse specifiche per la traduzione delle classi (traverse_class_init(), traverse_class_private_init(), traverse_class_public_init(), traverse_class_public_func(), traverse_class_private_func()) servono per capire quale tipo di nodo tradurre e fanno da supporto alle funzioni di traduzione vere e proprie.

Fase di test

Al fine di testare il funzionamento del transpiler, sono stati organizzati alcuni file di test organizzati in sottocartelle: ciascuna cartella identifica il tipo di istruzione o struttura da testare.

Per ogni test sono stati previsti due file di codice sorgente in C++: il primo scritto in maniera corretta per verificare che la traduzione venga effettuata in maniera efficace, il secondo invece sarà volutamente errato al fine di testare l'efficacia dei controlli lessicali, sintattici e semantici.

Nelle immagini seguenti si riporta un esempio di test effettuato:

```
KEYWORD 'INT' found at line 6.
ID 'Funzione1' found at line 6.
KEYWORD 'INT' found at line 6.
ID 'x' found at line 6.
KEYWORD 'COUT' found at line 7.
OPERATOR '<<' found at line 7.
STRING '"Funzione 1 corretta"' found at line 7.
OPERATOR '<<' found at line 7.
ID 'x' found at line 7.
KEYWORD 'RETURN' found at line 8.
ID 'x' found at line 8.
KEYWORD 'INT' found at line 11.
ID 'Funzione2' found at line 11.
KEYWORD 'COUT' found at line 12.
OPERATOR '<<' found at line 12.
STRING '"Funzione 2 corretta"' found at line 12.
KEYWORD 'RETURN' found at line 13.
INT NUMBER '0' found at line 13.
KEYWORD 'INT' found at line 16.
ID 'Funzione3' found at line 16.
KEYWORD 'INT' found at line 16.
ID 'x' found at line 16.
KEYWORD 'STRING' found at line 16.
ID 'y' found at line 16.
KEYWORD 'BOOL' found at line 16.
ID 'z' found at line 16.
KEYWORD 'COUT' found at line 17.
OPERATOR '<<' found at line 17.
STRING '"Funzione 1 corretta"' found at line 17.
OPERATOR '<<' found at line 17.
ID 'x' found at line 17.
OPERATOR '<<' found at line 17.
ID 'y' found at line 17.
OPERATOR '<<' found at line 17.
ID 'z' found at line 17.
KEYWORD 'RETURN' found at line 18.
INT NUMBER '0' found at line 18.
KEYWORD 'INT' found at line 22.
KEYWORD 'MAIN' found at line 22.
ID 'Funzione1' found at line 24.
INT NUMBER '1' found at line 24.
ID 'Funzione2' found at line 25.
ID 'Funzione3' found at line 26.
INT NUMBER '1' found at line 26.
STRING '"2"' found at line 26.
BOOL VALUE 'true' found at line 26.
KEYWORD 'RETURN' found at line 28.
INT NUMBER '0' found at line 28.

Parsing completed
```

```
        ***Error: Funzione4 already declared***
        ***Line: 29***


KEYWORD 'INT' found at line 32.
KEYWORD 'MAIN' found at line 32.
ID 'Funzione1' found at line 34.
FLOAT NUMBER '1.1' found at line 34.
ID 'Funzione2' found at line 35.
STRING '"sbagliato"' found at line 35.


        ***Error: Function Funzione2 has no parameters ***
        ***Line: 35***


ID 'Funzione3' found at line 36.
INT NUMBER '1' found at line 36.
STRING '"2"' found at line 36.
BOOL VALUE 'true' found at line 36.
STRING '"4" parametro in più"' found at line 36.


        ***Error: Number of defined function parameter and called function differs ***


ID 'Funzione4' found at line 37.
FLOAT NUMBER '99.999' found at line 37.


        ***Error: Parameter 99.999 is type float, while declared function needs type int ***


ID 'Funzione5' found at line 38.
STRING '"non dichiarata"' found at line 38.


        ***Error: Funzione5 is not declared***
        ***Line: 38***


KEYWORD 'RETURN' found at line 39.
INT NUMBER '0' found at line 39.

Parsing failed. errors found: 6
```

Esempio di test sulla definizione e chiamata di funzione

Riferimenti

1. **Floriano Scioscia. Formal Languages and Compilers** - slide del corso, 2020/2021. http://sisinflab.poliba.it/scioscia
2. **A.H. Aho , M.S. Lam, R. Sethi , J.D. Ullman, "Compilers: principles, techniques & tools"**, 2nd Edition, Pearson/Addison Wesley, 2007
3. **Troy D. Hanson, Arthur O'Dwyer, "uthash User Guide"**, version 2.3.0, February 2021, http://troydhanson.github.io/uthash/userguide.html
4. **Lexical Analysis with Flex**, Edition 2.6.0, 10 November 2015, https://epaperpress.com/lexandyacc/download/flex.pdf
5. **GNU Bison - The Yacc-compatible Parser Generator**, Free Software Foundation, https://www.gnu.org/software/bison/manual/
6. **C++ Programming Language**, Last Updated : 29 Aug, 2022, https://www.geeksforgeeks.org/c-plus-plus/
7. **Python Programming Language**, Last Updated : 16 Jun, 2022, https://www.geeksforgeeks.org/python-programming-language/