UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

A SIMPLE

{C}

PARSER

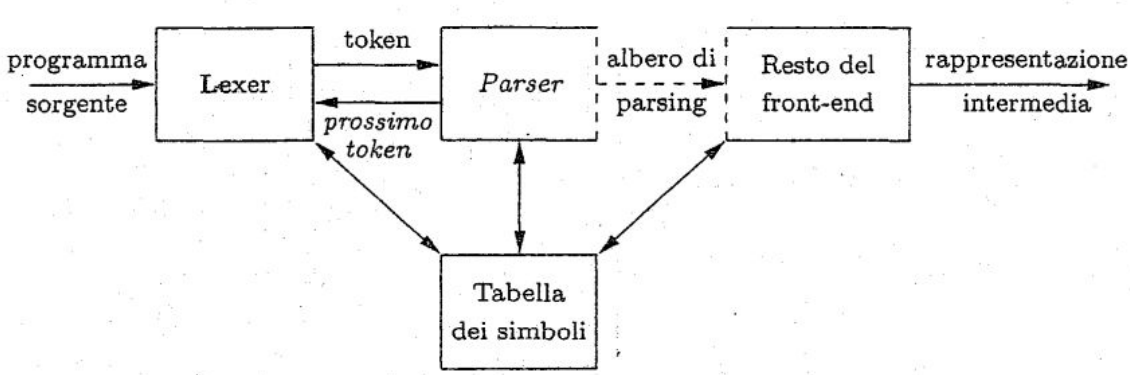
Corso di Compilatori
Anno accademico
2015/2016

Supervisor
Prof. Gennaro Costagliola

Studenti
Rosario Di Florio
Vincenzo Venosi

Obiettivi

L'obiettivo di questo progetto è stato quello di implementare un semplice parser, scrivendo una grammatica che riconoscesse un sotto insieme del linguaggio C standard ansi c-89, generando l'albero sintattico per un programma C, gestendo eventuali token ERROR dal lexer, usando la panic mode per gli errori sintattici, visualizzando graficamente l'albero attraverso una libreria grafica.
Il ruolo del parser all'interno di un compilatore è quello di ricevere una sequenza di token dall'analizzatore lessicale creato precedentemente tramite JFlex e verificare se tale sequenza di token può essere generata dalla grammatica del linguaggio sorgente, in questo caso la grammatica che rappresenta il linguaggio C.
Il parser inoltre è anche in grado di segnalare in forma chiara gli eventuali errori più comuni, per poi riprendere l'analisi della parte restante del programma.
Inoltre il parser costruisce un albero di parsing e lo passa alla parte restante del compilatore per una succesiva elaborazione.



Esistono 3 tipi di parser per le grammatiche: *universali, top-down* o *discendenti* e *bottom-up* o *discendenti.*
Il tipo di parser da noi generato tramite è un parser bottom-up, LALR generato con l'utilizzo di CUP, un generatore automatico di parser che illustreremo successivamente.

Progettazione e implementazione

La realizzazione del nostro progetto è passata attraverso tre step principali:

- Creazione del parser attraverso l'utilizzo di CUP , un generatore automatico di parser LALR.
- Costruzione dell'albero di parsing implementando le seguenti classi in Java illustrate in seguito:
    - Node

  - Tree
  - CTree
  - CNode

- Rappresentazione grafica dell'albero attraverso la libreria grafica **Jgraphx**.

CUP Parser generator

Constructor of Useful Parser (CUP abbreviato) è un generatore di parser LALR per JAVA. E' stato sviluppato da C. Scott Ananian, Frank Flannery, Dan Wang, Andrew W. Appel and Michael Petter.
CUP ha in pratica lo stesso ruolo del più famoso YACC ed offre molte delle features di YACC.
Questo generatore automatico implementa lo standard LALR(1), come autori del parser abbiamo specificato i simboli della nostra grammatica (terminal T1,T2; N1, N2;), le produzioni del tipo (LHS : == RHS1 | RHS2 ;) e le azioni da compiere all'interno di una produzione con l'action code ({: RESULT = myfunction(); :}), attraverso le azioni siamo stati in grado di assemblare un AST (Abstract Syntax Tree) che poi siamo andati a rappresentare graficamente.

Struttura di un file .cup

La struttura di una specifica CUP generale è descritta al seguente link:
https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html#spec

In generale all'inizio vi sono tutti i classici errori import necessari per il funzionamento del programma, subito dopo viene creata un classe interna che contiene tutte le azioni inserite dal programmatore tramite diverse dichiarazione.

User Code component

`action code {: ... :};`
questa dichiarazione permette di inserire codice in questa classe interna. Le funzioni e le variabili usate da parte del codice all'interno delle azioni della grammatica sono inserite all'interno di questa sezione, ad esempio abbiamo inserito la funzione che tiene traccia del numero di riga del lessema arrivato dal lexer.

```
action code {:
    int curr_line() {
       return ((CLexer)parser.getScanner()).curr_line();
    }
:}
```

`parser code {: ... :};`
questa dichiarazione permette ai metodi e alle variabili di essere inseriti direttamente all'interno del parser generato, questo serve per poter personalizzare il parser.
In questo caso abbiamo inserito le funzioni:

- `public void syntax_error(Symbol)`
    - questa funzione stampa chiaramente il tipo di errore e la linea dove è stato trovato.
- `public static void main(String args[])`
    - qui viene istanziato un parser che genera l'albero e crea tramite la libreria grafica la rappresentazione dell'albero, inoltre permette anche una stampa da terminale per debug.

```
parser code {:
    public void syntax_error(Symbol cur_token){
        int lineno = action_obj.curr_line();
       System.err.println("Syntax error at "+ cur_token + " at line " + (lineno + 1));

    }

    public static void main(String args[]) {
       try {

            CLexer lexer = new CLexer(new FileReader(args[0]));
           // start parsing
           Parser p = new Parser(lexer);
           System.out.println("Parser runs: ");
            p.parse();
           System.out.println("Parsing finished!");

           //debug
            C_Tree.tree.printTree();

            //Uso della libreria Jgraphx per visualizzare l'albero
            GraphicTree frame = new GraphicTree();
            Node root = C_Tree.tree.getRoot();
            int width = 2000;
            frame.generateTree(root,null,0,0,width);
            frame.printGraphicTree();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(width, 720);
            frame.setVisible(true);

       } catch (Exception e) {
           e.printStackTrace();
       }
    }
:};
```

Symbol Lists

Dopo questa sezione vengono definiti i terminali e i non terminali usati per generare il file dei simboli utilizzato dal nostro lexer per dare in output i lessemi utilizzati dalla nostra grammatica.
I terminali inseriti all'interno della nostra grammatica sono:

```
DOT, EQ, AND, NULL_LITERAL, ERROR, ID, CHAR_CONSTANT, INT_CONSTANT, REAL_CONSTANT,
OCT_CONSTANT, HEX_CONSTANT, STRING, SIZEOF, PTR_OP, PLUSPLUS, MINUSMINUS, LSHIFT,
RSHIFT, LTEQ, GTEQ, EQEQ, NOTEQ, ANDAND, OROR, MULTEQ, DIVEQ, MODEQ, PLUSEQ, MINUSEQ,
LSHIFTEQ, RSHIFTEQ, ANDEQ, XOREQ, OREQ, TYPE_NAME, TYPEDEF, EXTERN, STATIC, AUTO,
REGISTER, CHAR_TYPE,     SHORT_TYPE, INT_TYPE, LONG_TYPE, SIGNED, UNSIGNED, FLOAT_TYPE,
DOUBLE_TYPE, CONST, VOLATILE, VOID, STRUCT,    ENUM, ELLIPSIS, CASE, DEFAULT, IF, ELSE,
SWITCH, WHILE, DO, FOR, GOTO, CONTINUE, BREAK, RETURN, SEMICOLON, LBRACE, RBRACE, COMMA,
COLON, LPAREN, RPAREN, LBRACK, RBRACK, POINT_TYPE, ADRESS, NOT, COMP, MINUS, PLUS, MULT,
DIV, MOD, LT, GT, XOR, OR, QUESTION
```

I non terminali invece sono:

```
program; var_declaration; function_declaration; type_var; type_specifier; type_sign;
type_var_sign_unsign;  param_list; statement; expression; operator; assignment_operator;
rpar_generic; lpar_generic; boolean_operator; loop_statement; assignment_expression;
cond_expression; statement_list; cond_statement; call_function; param_list_call;
list_number; array_statement; array_declaration; unary_operator; string_end;
constant_type; conjunction; identifier; pointer_id;
```

Precedence declarations

Questa sezione specifica le precedenze da dare ad alcuni terminali, questo è molto utile la grammatica utilizzata è ambigua (ad esempio le espressioni).
Le precedenze sono state date alle operazioni secondo la specifica GNU C presente all'interno della documentazione C.
https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#Operator-Precedence

```
precedence left OROR;
precedence left ANDAND;
precedence left OR;
precedence left XOR;
precedence left AND;
precedence left EQEQ ,NOTEQ,NOT;
precedence left LT ,GT, LTEQ ,GTEQ;
precedence left MINUS, PLUS;
precedence left MULT, DIV,MOD;
```

Le precedenze risolvono problemi di shif/reduce. Per esempio quando abbiamo in input al parser **3 + 4 * 8** il parser non sare se ridurre **3 + 4** o effettuare lo shit sul **'*'** all'interno dello stack. Viene risolto assegnando a **'*'** una precedenza maggiore di **'+'** e quindi viene risolto in favore dello shift e ridotto successivamente **4 * 8**.

L'ultima sezione invece è dedicata alla grammatica utilizzata per il riconoscimento del linguaggio.

Grammatica Utilizzata

Nella nostra specifica CUP, abbiamo scritto una grammatica semplificata che racchiudesse un sotto-insieme abbastanza corposo del linguaggio C.

La grammatica utilizzata e scritta da noi è una grammatica ambigua, cioè produce più di un albero di parsing per un determinato input, S-attribuita perché ogni suo attributo e sintetizzato cioè il valore di ogni nodo è dato solo dal valore dei sui figli.
Per poterla usare all'interno di CUP che è un generatore di parser LALR abbiamo risolto le ambiguità con le precedenze, inoltre CUP implicitamente quando si trova di fronte a un conflitto shift/reduce sceglie di applicare lo shift sul prossimo carattere ad esempio il problema del dagling else viene risolto di default in CUP.
Di seguito viene mostrata la porzione di grammatica che descrive un semplice programma C

```
start with program;


program
                        ::=
                        function_declaration:f {: C_Tree.tree.getRoot().addChild(f); :}
program

                        |CONST var_declaration:d SEMICOLON {:
C_Tree.tree.getRoot().addChild(new Const(d,curr_line())); :}   program
                        |var_declaration:d SEMICOLON {: C_Tree.tree.getRoot().addChild(d);
:} program
                        |array_declaration:d string_end:e
                        {:
                            C_Tree.tree.getRoot().addChild(d);
                            if(e != null)
                            C_Tree.tree.getRoot().addChild(e);
                        :}
                        program
                        |ERROR:e  {: C_Tree.tree.getRoot().addChild(new Node("ERROR " + e));
:}  program
                        |
                        |error:e {: C_Tree.tree.getRoot().addChild(new Node("sintax error at
line: " + curr_line())); :} program
                        ;
```

Di seguito invece è mostrato un dump human readable della nostra grammatica usando il comando -dump_grammar.

```
[0] $START ::= program EOF
[1] NT$0 ::=
[2] program ::= function_declaration NT$0 program
[3] NT$1 ::=
[4] program ::= CONST var_declaration SEMICOLON NT$1 program
[5] NT$2 ::=
[6] program ::= VOLATILE var_declaration SEMICOLON NT$2 program
[7] NT$3 ::=
[8] program ::= var_declaration SEMICOLON NT$3 program
[9] NT$4 ::=
[10] program ::= array_declaration string_end NT$4 program
[11] NT$5 ::=
[12] program ::= ERROR NT$5 program
[13] program ::=
[14] NT$6 ::=
[15] program ::= error NT$6 program
```

```
[16] var_declaration ::= type_var identifier var_declaration
[17] var_declaration ::= COMMA identifier var_declaration
[18] var_declaration ::= COMMA expression var_declaration
[19] var_declaration ::= COMMA assignment_expression var_declaration
[20] var_declaration ::= COMMA identifier assignment_operator expression var_declaration
[21] var_declaration ::= type_var identifier assignment_operator expression
var_declaration
[22] var_declaration ::=
[23] list_number ::= constant_type COMMA list_number
[24] list_number ::= constant_type
[25] list_string ::= STRING COMMA list_string
[26] list_string ::= STRING
[27] array_statement ::= LBRACE RBRACE
[28] array_statement ::= LBRACE list_number RBRACE
[29] array_statement ::= LBRACE list_string RBRACE
[30] array_declaration ::= type_var identifier LBRACK constant_type RBRACK
[31] array_declaration ::= type_var identifier LBRACK constant_type RBRACK EQ
array_statement
[32] array_declaration ::= type_var identifier LBRACK RBRACK EQ array_statement
[33] array_declaration ::= type_var identifier LBRACK RBRACK EQ string_end
[34] array_declaration ::= type_var identifier LBRACK constant_type RBRACK EQ string_end
[35] string_end ::= STRING string_end
[36] string_end ::= ERROR string_end
[37] string_end ::= SEMICOLON
[38] string_end ::=
[39] type_var_sign_unsign ::= type_sign CHAR_TYPE
[40] type_var_sign_unsign ::= type_sign SHORT_TYPE
[41] type_var_sign_unsign ::= type_sign INT_TYPE
[42] type_var_sign_unsign ::= type_sign LONG_TYPE
[43] type_var ::= FLOAT_TYPE
[44] type_var ::= DOUBLE_TYPE
[45] type_var ::= type_var_sign_unsign
[46] type_sign ::= SIGNED
[47] type_sign ::= UNSIGNED
[48] type_sign ::=
[49] call_function ::= identifier LPAREN param_list_call RPAREN
[50] function_declaration ::= type_var identifier LPAREN param_list RPAREN
cond_statement
[51] function_declaration ::= type_specifier identifier LPAREN param_list RPAREN
cond_statement
[52] type_specifier ::= VOID
[53] param_list_call ::= STRING COMMA param_list_call
[54] param_list_call ::= STRING
[55] param_list_call ::= identifier COMMA param_list_call
[56] param_list_call ::= identifier
[57] param_list_call ::= call_function COMMA param_list_call
[58] param_list_call ::= call_function
[59] param_list_call ::=
[60] param_list_call ::= error
[61] param_list_call ::= error SEMICOLON
[62] param_list ::= type_var identifier COMMA param_list
[63] param_list ::= type_var identifier
[64] param_list ::=
```

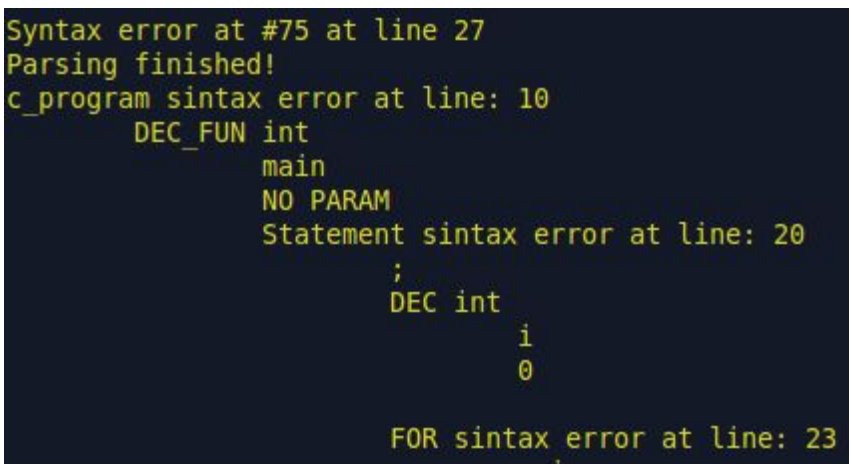
```
[65] param_list ::= error
[66] param_list ::= error cond_statement
[67] statement_list ::= statement statement_list
[68] statement_list ::= statement
[69] statement_list ::= LBRACE statement RBRACE
[70] statement ::= var_declaration SEMICOLON
[71] statement ::= expression SEMICOLON
[72] statement ::= assignment_expression SEMICOLON
[73] statement ::= cond_expression
[74] statement ::= call_function SEMICOLON
[75] statement ::= array_declaration SEMICOLON
[76] statement ::= SEMICOLON
[77] statement ::= error statement
[78] statement ::= error cond_statement
[79] statement ::= error RBRACE
[80] statement ::= error RPAREN
[81] statement ::= RETURN return_statement
[82] loop_statement ::= expression
[83] loop_statement ::= assignment_expression
[84] loop_statement ::= cond_expression
[85] loop_statement ::= error loop_statement
[86] loop_statement ::= error
[87] assignment_expression ::= identifier assignment_operator expression
[88] assignment_expression ::= identifier LBRACK constant_type RBRACK
assignment_operator expression
[89] cond_expression ::= IF LPAREN loop_statement RPAREN cond_statement blockElseIf
[90] cond_expression ::= IF LPAREN loop_statement RPAREN cond_statement ELSE
cond_statement
[91] cond_expression ::= WHILE LPAREN loop_statement RPAREN cond_statement_loop
[92] cond_expression ::= FOR LPAREN loop_statement_for loop_statement_for
loop_statement_for_last RPAREN cond_statement_loop
[93] loop_statement_for ::= expression SEMICOLON
[94] loop_statement_for ::= assignment_expression SEMICOLON
[95] loop_statement_for ::= cond_expression SEMICOLON
[96] loop_statement_for ::= error
[97] loop_statement_for ::= SEMICOLON
[98] loop_statement_for_last ::= loop_statement
[99] loop_statement_for_last ::=
[100] cond_statement ::= LBRACE statement_list RBRACE
[101] cond_statement ::= LBRACE statement_list ERROR
[102] cond_statement ::= LBRACE RBRACE
[103] cond_statement ::= statement
[104] cond_statement_loop ::= LBRACE statement_list_loop RBRACE
[105] cond_statement_loop ::= LBRACE statement_list_loop ERROR
[106] cond_statement_loop ::= LBRACE RBRACE
[107] cond_statement_loop ::= statement
[108] statement_list_loop ::= statement statement_list_loop
[109] statement_list_loop ::= statement
[110] statement_list_loop ::= BREAK SEMICOLON
[111] statement_list_loop ::= BREAK SEMICOLON
[112] statement_list_loop ::= CONTINUE SEMICOLON
[113] statement_list_loop ::= error
[114] return_statement ::= expression SEMICOLON
```

```
[115] return_statement ::= assignment_expression SEMICOLON
[116] return_statement ::= cond_expression
[117] return_statement ::= call_function SEMICOLON
[118] return_statement ::= error return_statement
[119] return_statement ::= error RBRACE
[120] return_statement ::=
[121] expression ::= expression OROR expression
[122] expression ::= expression EQEQ expression
[123] expression ::= expression NOTEQ expression
[124] expression ::= NOT expression
[125] expression ::= expression NOT expression
[126] expression ::= expression LTEQ expression
[127] expression ::= expression GTEQ expression
[128] expression ::= expression LT expression
[129] expression ::= expression GT expression
[130] expression ::= expression ANDAND expression
[131] expression ::= expression MOD expression
[132] expression ::= expression OR expression
[133] expression ::= expression AND expression
[134] expression ::= expression XOR expression
[135] expression ::= expression MULT expression
[136] expression ::= expression PLUS expression
[137] expression ::= expression DIV expression
[138] expression ::= expression MINUS expression
[139] expression ::= expression MINUS expression
[140] expression ::= expression boolean_operator expression
[141] expression ::= expression conjunction expression
[142] expression ::= lpar_generic expression rpar_generic
[143] expression ::= identifier
[144] expression ::= identifier unary_operator
[145] expression ::= unary_operator identifier
[146] expression ::= constant_type
[147] expression ::= MINUS identifier
[148] expression ::= MINUS constant_type
[149] expression ::= call_function
[150] unary_operator ::= PLUSPLUS
[151] unary_operator ::= MINUSMINUS
[152] lpar_generic ::= LPAREN
[153] lpar_generic ::= LBRACK
[154] rpar_generic ::= RPAREN
[155] rpar_generic ::= RBRACK
[156] assignment_operator ::= EQ
[157] assignment_operator ::= MULTEQ
[158] assignment_operator ::= DIVEQ
[159] assignment_operator ::= MODEQ
[160] assignment_operator ::= PLUSEQ
[161] assignment_operator ::= MINUSEQ
[162] assignment_operator ::= LSHIFTEQ
[163] assignment_operator ::= RSHIFTEQ
[164] assignment_operator ::= ANDEQ
[165] assignment_operator ::= XOREQ
[166] assignment_operator ::= OREQ
[167] boolean_operator ::= EQEQ
```

```
[168] boolean_operator ::= LT
[169] boolean_operator ::= GT
[170] boolean_operator ::= LTEQ
[171] boolean_operator ::= GTEQ
[172] boolean_operator ::= NOT
[173] boolean_operator ::= NOTEQ
[174] constant_type ::= INT_CONSTANT
[175] constant_type ::= HEX_CONSTANT
[176] constant_type ::= OCT_CONSTANT
[177] constant_type ::= REAL_CONSTANT
[178] constant_type ::= CHAR_CONSTANT
[179] conjunction ::= ANDAND
[180] conjunction ::= OROR
[181] identifier ::= ID
[182] identifier ::= pointer_id
[183] identifier ::= ID LBRACK ID RBRACK
[184] identifier ::= ID LBRACK INT_CONSTANT RBRACK
[185] pointer_id ::= MULT ID
[186] pointer_id ::= MULT MULT ID
[187] pointer_id ::= AND ID
[188] blockElseIf ::= ELSE IF LPAREN loop_statement RPAREN cond_statement blockElseIf
[189] blockElseIf ::= ELSE IF LPAREN loop_statement RPAREN cond_statement
[190] blockElseIf ::= ELSE IF LPAREN loop_statement RPAREN cond_statement ELSE
cond_statement
[191] blockElseIf ::=
```

Error Handling

La gestione degli errori è stata fatta attraverso la panic mode di CUP.
CUP usa lo stesso meccanismo di error recovery di YACC. In particolare, supporta un simbolo speciale di errore (denotato semplicemente come error).
Questo simbolo gioca il ruolo di un non terminale speciale il quale, invece di essere definito da produzioni, rileva un errata sequenza di input.
Il simbolo di errore può essere generato solo quando un errore sintattico è rilevato.
Se è presente un errore il parser prova a rimpiazzare una porzione dello stream di token in input con error e continua ad analizzare appena rileva una sequenza di input corretta.



Abstract Syntax Tree

Si tratta di un albero in cui ogni nodo interno rappresenta un operatore e i figli di tale nodo rappresentano gli operandi.
Gli alberi sintattici sono simili agli alberi di derivazione, salvo che nei primi i nodi interni rappresentano gli operatori, mentre nei secondi rappresentano i non terminali.
Però molti dei non terminali sono simboli di “supporto” di varia natura, in un albero sintattico questi elementi non sono necessari e vengono omessi portando solo l’informazione sintetica ed astratta.
Questo serve spesso come rappresentazione intermedia di un programma attraverso diversi stage che effettua il compilatore e ha un forte impatto sull’input finale del compilatore.

Classi

L’albero è stato implementato in Java attraverso la creazione di quattro classi:

- Node
- Tree
- C_Tree
- CNode

La classe Node è la classe che rappresenta un nodo dell’albero. Un’ istanza di Node porta con se le la lista dei nodi figli, l’etichetta del nodo, ed eventualmente il riferimento al nodo padre.

```
public class Node<T> {

         private T data;
        private Node<T> parent;
        private List<Node<T>> children;

        public Node(T data){
               this.data = data;
               this.children = new ArrayList<Node<T>>();
       }

          public Node(T data,Node<T> parent){
                this.parent = parent;
               this.data = data;
               this.children = new ArrayList<Node<T>>();
       }

        public Node<T> getParent(){
            return this.parent;
        }

        public void setParent(Node<T> parent){
```

```
            this.parent = parent;
        }

        public List<Node<T>> getChilds(){
               return this.children;
        }

        public void addChild(Node<T> child){
               this.children.add(child);
        }
        public void addFirstChild(Node<T> child){
               this.children.add(0,child);
        }

        public boolean hasChilds(){
               return !this.children.isEmpty();
        }

        public T getData(){
               return this.data;
        }

        public void setData(T data){
            this.data = data;
        }

        public void dump(){
               System.out.print(this.data + " ");
        }


    }
```

La classe Tree rappresenta un albero nel quale le informazioni contenute sono i riferimenti alla radice dell' albero ed eventuali metodi per accedervi. Abbiamo anche implementanto un metodo per effettuare un dump dell' albero a video che ci ha aiutato nella fase di debug.

```
public class Tree<T> {
    private Node<T> root;

    public Tree(T rootData) {
        root = new Node<T>(rootData);

    }

    public Node<T> getRoot(){
       return this.root;
    }

    public List<Node<T>> getRootChilds(){
       return this.root.getChilds();
    }

    public void printTree(){

       printRecursive(this.root,0);
    }

    private void printRecursive(Node<T> node, int level){
```

```
                //System.out.print("["+ node.getData() + "]");
                //System.out.print("level: " + level + " ");
                node.dump();
                for(int i = 0;i < node.getChilds().size();i++){
                      printRecursive(node.getChilds().get(i),level+1);
                }
                System.out.println();
                printTab(level);
    }

    private void printTab(int n){

                for(int i = 0; i<n;i++)
                        System.out.print("\t");
    }

}
```

Le classi C_Tree e CNode rappresentano l' AST. La classe CNode è una sottoclasse di Node. La classe CNode è stata creata in quanto vi era bisogno di poter mantenere l'informazione riguardante la linea di codice corrente.

```
public class C_Tree  {

        public static Tree<String> tree = new Tree<String>("c_program");

}
```

```
class CNode<String> extends Node{
    int line;
    public CNode(String data,int line){
        super(data);
        this.line = line;
    }

    public void setCurrentLine(int line){
        this.line = line;
    }

}
```

Sono state successivamente create delle sottoclassi di CNode per andare a specializzare i nodi dell'albero.
Es:
Di seguito è riportata la classe Plus che è la sottoclasse di CNode e rappresenta un' operazione di addizione.

```
class Plus<String> extends CNode{

        public Plus(String rootData,  Node<String> a, Node<String> b,int line) {
                super(rootData,line);

                this.addChild(a);
```

```
                this.addChild(b);
        }

}
```

Come si può vedere, la classe prende come parametri nel costruttore due nodi che saranno i nodi contenenti i valori sui quali si dovrà effettuare l'operazione di addizione. I nodi presi in input come parametri potranno a loro volta essere ricorsivamente un istanza di Plus o di altre operazioni. Questo per codificare espressioni con più di due operatori, come ad esempio: 2 + 3 + 4;

Tabelle

Per mantenere i valori o i nomi degli identificatori trovati dal Lexer sono state utilizzate tre tabelle. Queste tabelle sono state inserite con il solo scopo di mostrare il funzionamento di queste ultime nella fase di analisi lessicale. Nelle fasi successive, come ad esempio nell' analisi semantica, queste strutture si rivelano necessarie, però la loro complessità dovrebbe essere molto più elevata di quelle inserite da noi:

- symTable: la tabella in cui vengono messi i nomi degli identificatori
- stringTable: la tabella in cui viene messo il contenuto di una string costant
- numTable: la tabella in cui viene messo il valore di un qualsiasi numero trovato, sia esso esadecimale, reale, intero, etc.

Nel file ci sono tre contatori, uno per ogni tabella. Quando viene trovato un lessema da inserire nella tabella viene utilizzato il contatore come indirizzo della tabella. Ogni tabella è una Hashtable.

Es:

"Ciao Mondo!"

diventa

<STRING,1>

dove "1" nella stringTable è la chiave per avere la stringa "Ciao Mondo!".

AGGIUNGERE UN ESEMPIO DI COME SI ACCEDE ALLA TABELLA NEL CUP

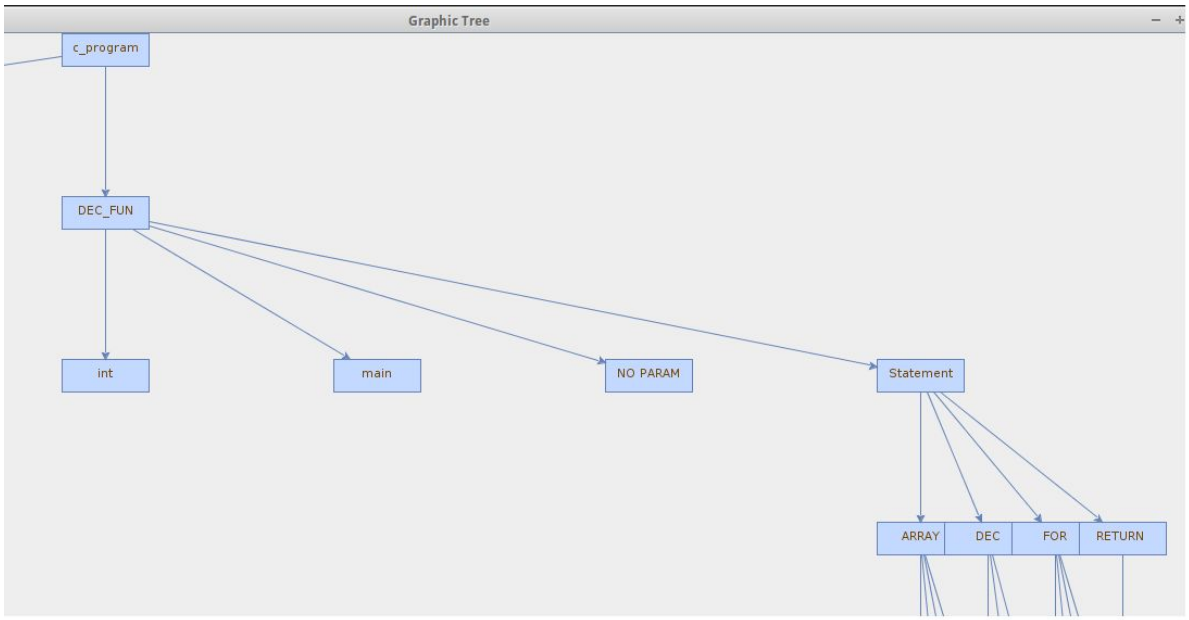
JGraphX

Per poter visualizzare il nostro Abstract Syntax Tree abbiamo utilizzato JGraphX.

JGraphX è una versione di mxGraph che è una famiglia di librerie grafiche, è chiamata così poiché l'autore prima di mxGraph ha avuto una lunga produzione chiamata JGraph.
Con JGraphX è possibile creare numerose applicazioni di visualizzazione di grafi come ad esempio diagrammi di processo, workflow, mapping applications etc…

Il codice di questa libreria è disponibile al seguente link:
https://github.com/jgraph/jgraphx

Di seguito è mostrato una porzione dell'albero generato attraverso l'uso di questa libreria.



Principali difficoltà incontrate

In generale abbiamo affrontato problemi derivati dalla creazione di una grammatica che esprimesse un corposo sottoinsieme del linguaggio C.
Tra i problemi più significati incontrati durante lo sviluppo del parser vi è stato il problema che, a differenza di altri linguaggi, il manuale ufficiale di riferimento C non conteneva nessuna grammatica che descrivesse i costrutti del linguaggio che erano descritti informalmente in tutti i possibili casi in cui si potessero incontrare e quindi da un discorso informale abbiamo dovuto ricavare una grammatica formale e testarne la bontà.
Intere grammatiche per il nostro linguaggio esistono già implementate completamente, però considerata l'elevata complessità di queste ultime e poiché lo scopo del progetto è quello di generare un parser, copiare una grammatica complessa e non ci avrebbe aiutato ai fini dell'apprendimento risultando a volte un arma a doppio taglio poiché avremmo dovuto in ogni caso capire l'intera grammatica per poter costruire l'Abstract Syntax Tree.
Un ulteriore problema è stato quello affrontato nella gestione dell' errore nel costrutto *for* in quanto c'era il bisogno di catturare l'errore in più posizioni. Un costrutto *for* può avere errori nel corpo o racchiuso tra le parentesi tonde.
Di seguito riportiamo alcuni esempi dei possibili errori nel costrutto *for*:

mancato carattere ";" nella dichiarazione del costrutto *for*

```
    for ( i = 0 i < MAX ; i++)
    {
       printf("Value of names[%d] = %s\n", i, names[i] );
    }
```

error nel corpo del costrutto *for*:
in questo caso nella funzione printf(...) non è stato inserito il carattere “virgolette” di chiusura stringa.

```
    for ( i = 0; i < MAX ; i++)
    {
       printf("Value of names[%d] = %s\n , i, names[i] );
    }
```

inizializzazione della variabile “i” non scritta correttamente nella dichiarazione del costrutto *for*

```
    for ( i = ; i < MAX ; i++)
    {
       printf("Value of names[%d] = %s\n", i, names[i] );
    }
```

Test

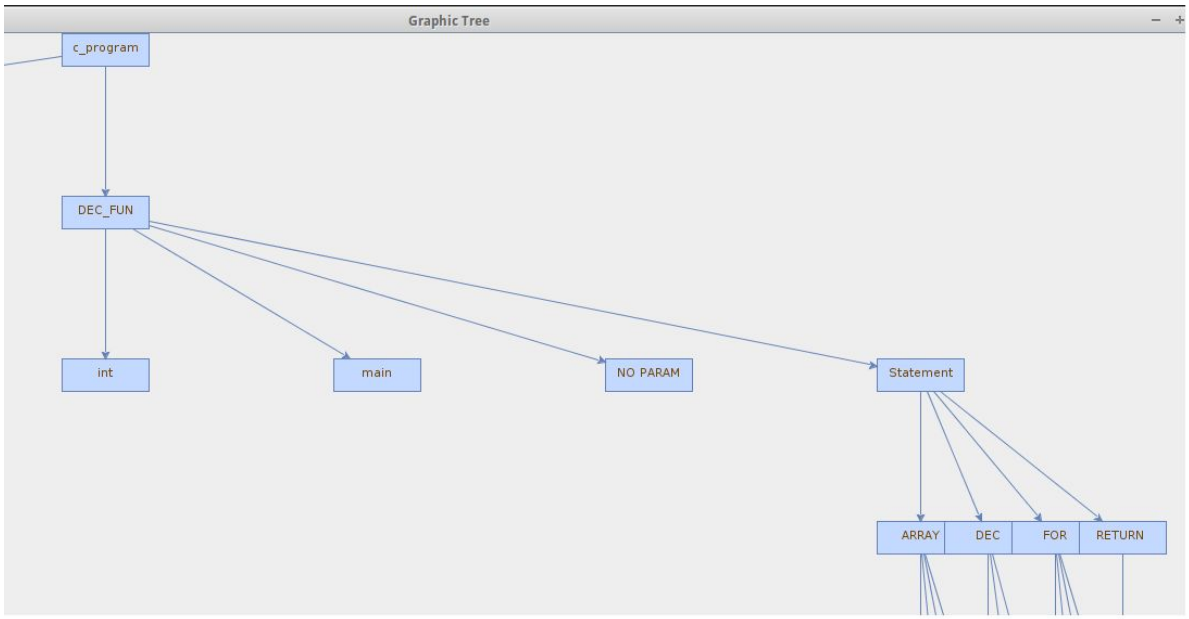
Sono di seguito riportati due dei vari test riguardanti il funzionamento del parser e la gestione dell’errore.

Test good
Il seguente codice è stato usato come test per mostrare il comportamento del parser su un semplice programma C.

```
const int MAX = 4;

int main ()
{
    char *names[] = {
                    "Zara Ali",
                    "Hina Ali",
                    "Nuha Ali",
```

```
                    "Sara Ali"
    };

    int i = 0;

    for ( i = 0; i < MAX; i++)
    {
       printf("Value of names[%d] = %s\n", i, names[i] );
    }

 return 0;

}
```

Il parser generato dalla nostra specifica CUP ha correttamente prodotto l' albero di parsing dal codice sopra riportato.
Di seguito riportiamo una parte dell' albero che viene mostrato a video.



Test bad
Il seguente codice è stato usato come test per mostrare la gestione dell'errore del parser generato dalla nostra specifica CUP. Il codice seguente è il codice visto nel test precedente, ma con degli errori sintattici.

```
//mancata chiusura di un istruzione con carattere ";"
const int MAX = 4

int main ()
{
    //mancato carattere "]"
```

```
    char *names[ = {
                    "Zara Ali",
                    "Hina Ali",
                    "Nuha Ali",
                    "Sara Ali"
    };

    int i = 0;

    //mancato carattere ";"  nella dichiarazione del costrutto for
    for ( i = 0 i < MAX ; i++)
    {
       printf("Value of names[%d] = %s\n", i, names[i] );
    }

 return 0;

}
```

Il parser ha generato l' albero sintattico con i nodi conteneti i rispettivi errori sintattici trovati nel codice.
Di seguito riportiamo una porzione di albero mostrato a video.