UNIVERSITÀ DEGLI STUDI DI PALERMO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Ingegneria

SERRA

Un linguaggio per l'Ambient Intelligence

Tesina per "Linguaggi e Traduttori"

TEAM:
Thermokípio

ANNO ACCADEMICO
2019 - 2020

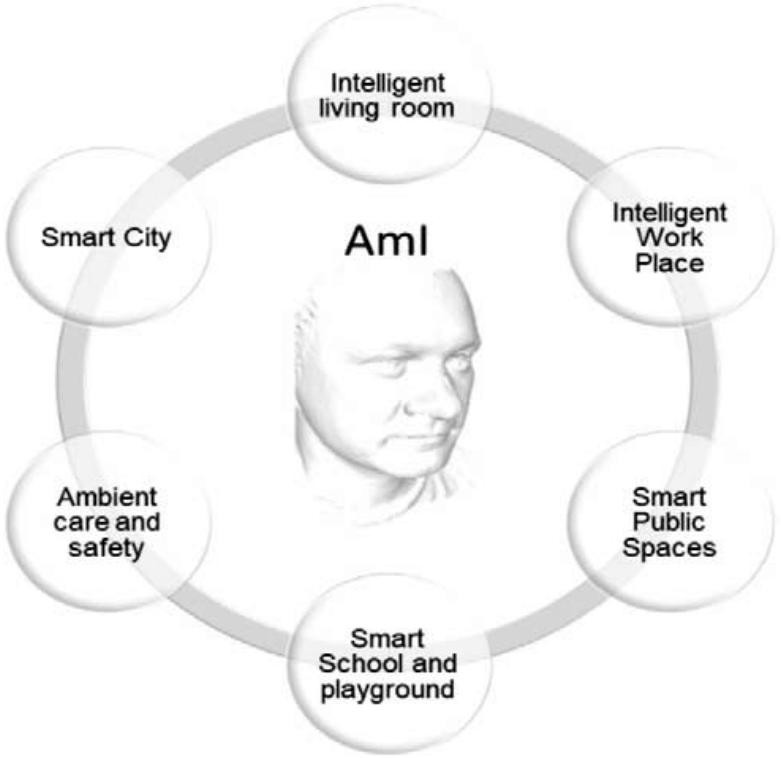
DOCENTI:
Prof. Ing. Antonio Chella
Ing. Francesco Lanza

Introduzione

- Ambient Intelligence (AMI)
  - Richiesta crescente di dispositivi che aiutano l'uomo nelle proprie attività lavorative
  - Caratteristiche principale:
    - User-Friendly

- SERRA
  - Linguaggio per dispositivi AMI
    - per la cura di giardini, serre e orti
  - User-Friendly

AMI Scenarios



Stato dell’arte

- Tecniche per la cura dei giardini
  - Irrigazione
  - controllo dell’umidità e della temperatura
  - ed ogni altro dispositivo controllabile.
- FarmaBot Genesis
  - SW per la gestione di sistemi agrari basato su interfaccia grafica che consente la gestione del giardino tramite coordinate x,y,z.

https://it.electronics-council.com/farmbot-intends-revolutionize-home-gardening-31975

- SERRA permetterà la gestione del giardino tramite un’interfaccia a linea di comando.

Stato dell'arte

- SERRA
  - L'utente obiettivo di SERRA
    - sapere come gestire il giardino
    - Non è richiesto sia esperto di linguaggi di programmazione
  - Il linguaggio è in grado di:
    - Creare/rimuovere istanze di oggetti Device
      - pompa, valvola, irrigatore, termometro, etc…
    - Verificare se un dispositivo è raggiungibile
    - Verificare/cambiare lo status del device (on/off)
      - Avviare/arrestare l'operazione di irrigazione
  - Comparazione con altri linguaggi di programmazione
    - È distante da
      - Meta Language, Ad esempio, non consente la definizione di nuovi tipi
      - È distante da COBOL, non consente operazioni matematiche.
    - SERRA
      - **Utilizza tipi primitivi/composti**
      - **Consente di:**
        - **Definire variabili**
        - **Realizzare Array**
        - **Realizzare liste di device**
        - **Definire funzioni runtime**
        - **Caricare programmi da file**

Fortran	*Pascal*	*ML*	*C*	*SERRA*
LOGICAL	boolean	bool		
INTEGER	integer	int	int	NUMBER
REAL*8	real			NUMBER
REAL*16			double	
CHARACTER	char		char	STRING

	C++	SERRA	Ada	FORTRAN
array	x	x	x	
struct	x			
union	x			
class	x			
device		x		
records			x	
liste di device		x		

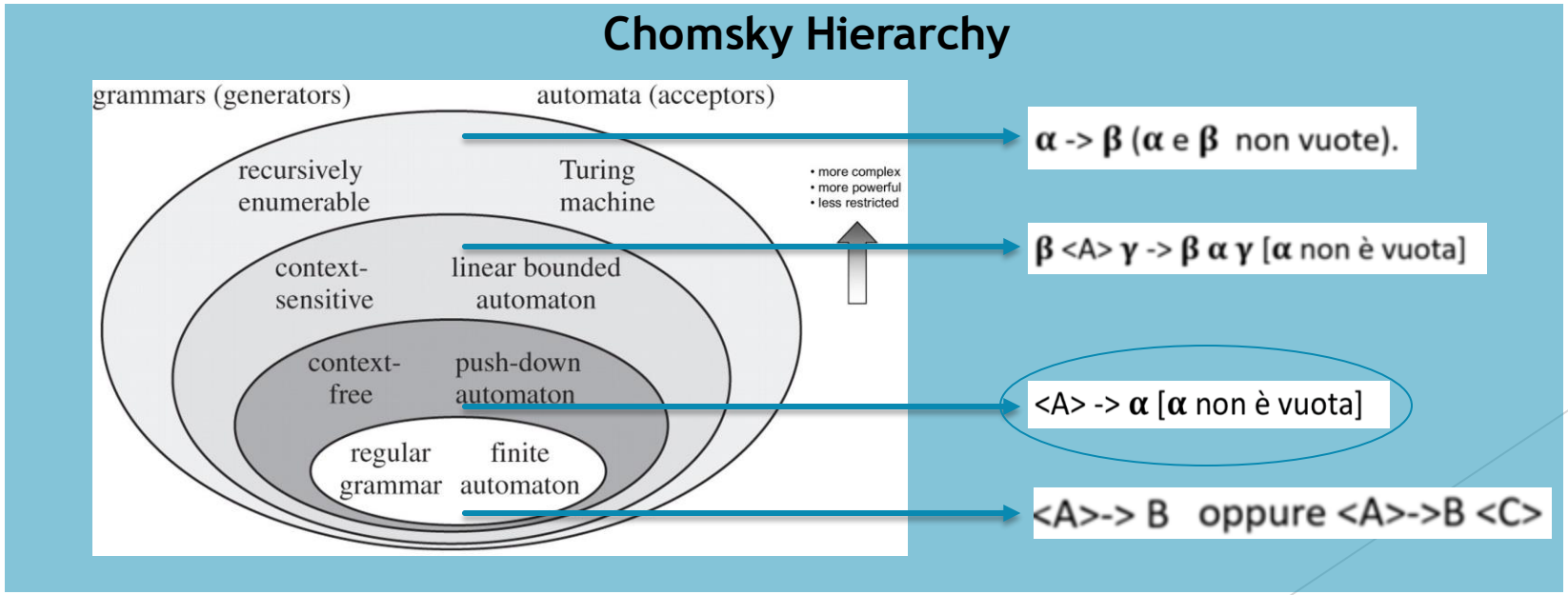
- È assimilabile a
  - ISPL perché non consente operazioni complesse su i tipi primitivi e composti, e perché con ISPL si possono simulare circuiti elettrici.
  - un File Batch perché, essendo pensato per utenti non esperti di programmazione, si sono inserite tante funzioni embedded (connessione, attivazione, etc..) che possono essere inserite in un file di testo.
  - È assimilabile, grazie al suo uso interattivo, alle shell programming; ossia le command-line interpreters
  - A Cool perché presenta caratteristiche base per renderne facile il suo utilizzo

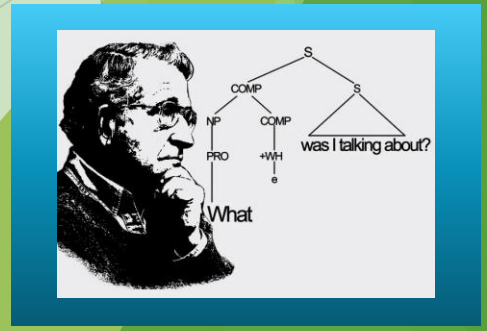
Descrizione del progetto

- SERRA: linguaggio per l'agricoltura
  - rende le operazioni di gestione dell'impianto
    - Rapide
    - Semplici
      - l'utente ignorerà la generazione di struct Device contenenti le informazioni degli stessi
        - dovrà unicamente di inserire una stringa per istanziare oggetti Device.
      - L'interprete carica in modo autonomo due file in fase di inizializzazione
        - database con i device
        - Libreria con funzioni ad alto livello
- Requisiti prefissati per SERRA:
  - A livello di gestione, le operazioni previste sono:
    - Creazione/Eliminazione istanza oggetto device
    - Verifica/Variazione dello status del device switchOn / switchOff
    - Variazione dello status del device per intervallo di tempo (thread)
  - A livello di linguaggio sono previsti:
    - Definizioni e assegnamento di variabili
    - Comandi Runtime
    - Istruzioni Condizionali
    - Cicli iterativi
    - Esecuzione di programmi da File

Caratteristiche del Linguaggio -> Flex & Bison

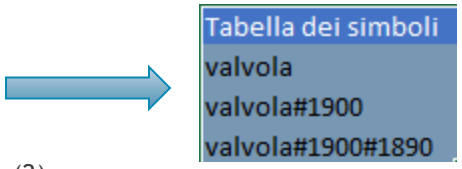
- Per la realizzazione del linguaggio ci si è avvalsi delle librerie:
  - Flex, che consente di realizzare un analizzatore lessicale
    - prende in input le sequenze utente
    - individua i token che definiscono la nostra grammatica
  - Bison, che consente di realizzare l'analizzatore sintattico
    - Cerca le regole che matchano con i token riconosciuti dal lexer
    - La grammatica è stata realizzata secondo Chomsky Type 2 (libera da contesto)
      - Tutti i simboli non terminali (produzioni) vengono tradotte in sequenze di simboli terminali/non terminali
      - È possibile esprimerlo sottoforma di automa dotato di una pila di dimensione infinita





Caratteristiche del Linguaggio -> Analizzatore lessicale

- Token principali sono
  - **NUMBER**
  - **STRING**, individua le stringhe
  - **NAME**, individua le variabili
    - Il linguaggio individua automaticamente cosa gli viene passato
      - data
      - String---------→ ''valvola''
      - Device --------→ newDevice ''valvola''
      - Array---------→ newArray char valvola (2)



NUMBER -> [0-9]+
- **Ex:** 1, 2, 3, 44

STRING-> ["][a-zA-Z][a-zA-Z0-9]*["]
- **Ex:** "ciao ", "DEVICE1", "device2", "d2"

NAME -> [a-zA-Z][a-zA-Z0-9]*
- **Ex:** ciao , DEVICE1, device2 , d2

DATE-> [2][0-9]{3}"."(0?[1-9]|1[012])"."(0?[1-9]|[12][0-9]|3[01])"."(0?[1-9]|1[0-9]|2[0-3])"."(0?[0-9]|[1-5][0-9])
- **Ex: 2021.01.01.9.10, 2021.11.01.9, 2021.01.31.09, 2020.01.01.09**

  - **DATA**, individuare le date

- Token che individuano le parole chiavi del sistema
  - **INSERT**, individua la parola chiave newDevice che permette la creazione di oggetti Device
  - **IF, THEN, ELSE e DO, WHILE** permettono, con l'analisi sintattica, di fare operazioni condizionali e post-condizionali
  - **SYSTEM** individuano le funzioni di sistema Help, clear
  - **FUNCDEV**, individua le funzioni embedded
    - connect, reconnect, switchOn, switchOff, status, delete, interval
  - **ADD, REMOVE, GET, SET**
    - Per le operazioni per la gestione degli array
  - I restanti digit saranno segnalati come sconosciuti e quindi come errore lessicale

Token	Espressioni Regolari	Parole chiavi
STRING	["][a-zA-Z][a-zA-Z0-9]*["]	
NUMBER	[0-9]+	
NAME	[a-zA-Z][a-zA-Z0-9]*	
ARROW	"->"	
EOL	\n	
<<EOF>>	EOF	
IF		"if"
ELSE		"else"
THEN		"then"
WHILE		"while"
DO		"do"
REPEAT		"repeat"
DATA	time	
CMD		"CMD"
ARRAY		"newArray"
INTEGER		"integer"
CHAR		"char"
DEVICE		"device"
ADD		"add"
GET		"get"
SET		"set"
REMOVE		"remove"
RET		"ret"
INSERT		"newDevice"
FUNCDEV		"connect"
		"status"
		"reconnect"
		"swtichOn"
		"switchOff"
		"delete"
		"interval"
FUNC		"print"
		"readFile"
SYSTEM		"clear"
		"help"

Caratteristiche del Linguaggio -> Analizzatore Sintattico

- La grammatica è stata realizzata in modo che le espressioni possano annidarsi a vicenda
  - funzione embedded possono prendere come parametro attuale qualsiasi oggetto
    - Stringa
    - intero
    - funzione richiamata
- Principali produzioni
  - **exec** (esecuzione del comando)
    - è lo scopo del nostro parser
    - da questa produzione avviene lo scorrimento dell'albero sintattico
    - Derivando più volte tale produzione riusciamo ad effettuare l'analisi sintattica del linguaggio
  - **statement**, è il simbolo non terminale che consente di individuare le istruzioni:
    - If, then, else
    - Do, while
    - Espressioni
  - **exp**, è il simbolo non terminale che consente l'individuazione di:
    - Variabili
    - Tutte e espressioni
      - Funzioni inerenti al device
      - Funzioni inerenti agli array
      - Funzioni di sistema/embedded
      - tipizzazione

```
exec:       ->
                exec stmt EOL
                exec function EOL
                exec TERM EOL
                exec error EOL
;

;
stmt:       ->  IF exp THEN listStmt
                IF exp THEN listStmt ELSE listStmt
                WHILE exp DO listStmt
                exp
;
function:   ->  CMD NAME '(' argsList ')' '=' listStmt RET exp ';'
                CMD NAME '(' argsList ')' '=' listStmt

;
listStmt:   ->
                stmt ';' listStmt
;
exp:        ->
                '(' exp ')'
                type
                FUNC exp
                FUNCDEV exp
                SYSTEM
                NAME
                NAME '=' exp
                INTERVAL exp '-' exp '-' exp
                NAME '(' explistStmt ')'
                liste
;

liste:      ->  ARRAY nameType NAME '(' NUMBER ')'
                NAME ARROW  ADD '=' type
                NAME ARROW GET
                NAME ARROW  GET '=' NUMBER
                NAME ARROW  SET '=' type ',' NUMBER
                NAME ARROW REMOVE
;
nameType:   ->  CHAR
                INTEGER
                DEVICE
;
nameType:       ->      CHAR
                        INTEGER
                        DEVICE
;
type:                   INSERT STRING
                        NUMBER
                        STRING
                        INSERT STRING ARROW '[' argsListDevice ']'
                        DATA
;


explistStmt:    ->      exp
                        exp ',' explistStmt
;

argsList:       ->      NAME
                        NAME ',' argsList
;

argsListDevic:  ->      STRING
                        STRING ',' argsListDevice
```

Forma semplificata della grammatica utilizzata

Caratteristiche del Linguaggio -> Analizzatore Sintattico

- **exp**, inoltre, può essere definito come
  - **type**, simbolo non terminale definito come
    - **NUMBER**, simbolo terminale per i numeri (es. 2)
    - **STRING**, simbolo terminale per le stringhe (es. ‘’ciao”)
    - due espressioni per definire il tipo device
      - **INSERT STRING**, due simboli terminali, il primo individuato dalla parola chiave newDevice, (es. newDevice ‘’pompa”)
      - **INSERT STRING ARROW ‘[‘ argsListDevice’]’** , crea il dispositivo e indica con quali altri è collegato. (es. newDevice ‘’pompa” -> [‘’serbatoio”])
  - **FUNCDEV exp**, simbolo terminale per le funzioni embedded e simbolo non terminale per un’altra espressione. es.:
    - connect ‘’pompa”
    - switchON ‘’pompa”
    - switchOFF ‘’pompa”
    - switchON connect ‘’pompa” , è un esempio di annidamento in cui si passa una stringa come paramentro alla connect e il parametro restituito viene passato come parametro all’espressione switchOn

```
exec:       ->
                exec stmt EOL
                exec function EOL
                exec TERM EOL
                exec error EOL
;

;
stmt:       ->  IF exp THEN listStmt
                IF exp THEN listStmt ELSE listStmt
                WHILE exp DO listStmt
                exp
;
function:   ->  CMD NAME '(' argsList ')' '=' listStmt RET exp ';'
                CMD NAME '(' argsList ')' '=' listStmt

;

listStmt:   ->
                stmt ';' listStmt
;
exp:        ->
                '(' exp ')'
                type
                FUNC exp
                FUNCDEV exp
                SYSTEM
                NAME
                NAME '=' exp
                INTERVAL exp '-' exp '-' exp
                NAME '(' explistStmt ')'
                liste
;

liste:      ->  ARRAY nameType NAME '(' NUMBER ')'
                NAME ARROW  ADD '=' type
                NAME ARROW GET
                NAME ARROW  GET '=' NUMBER
                NAME ARROW  SET '=' type ',' NUMBER
                NAME ARROW REMOVE
;
nameType:   ->  CHAR
                INTEGER
                DEVICE
;
nameType:       ->      CHAR
                        INTEGER
                        DEVICE
;
type:                   INSERT STRING
                        NUMBER
                        STRING
                        INSERT STRING ARROW '[' argsListDevice ']'
                        DATA
;


explistStmt:    ->      exp
                        exp ',' explistStmt
;

argsList:       ->      NAME
                        NAME ',' argsList
;

argsListDevic:  ->      STRING
                        STRING ',' argsListDevice
```

Forma semplificata della grammatica utilizzata

Caratteristiche del Linguaggio -> Analizzatore Sintattico

- **exp**, inoltre, può essere definito come
  - **FUNC exp**, simbolo terminale seguito da un'espressione, per le funzioni di sistema (print, readFile)
  - **NAME '=' exp**
    - Con passaggio di un simbolo terminale/non terminale come parametro
      - (es. nomeVariabile = ''pompa'')
      - (es. nomeVariabile = connect ''pompa'')
  - **INTERVAL exp '-' exp '-' exp**
  - **lista**, è un simbolo non terminale; è definita
    - **ARRAY** nameType **NAME** (NUMBER), dove il simbolo non terminale andrà a identificare la tipizzazione dell'array con i simboli terminali INTEGER, CHAR, DEVICE. (es. newArray integer nomeArray (2))
    - **NAME ARROW ADD = type**, dove il simbolo non terminale effettua la tipizzazione. (es. nomeArray -> add = 2)
    - **NAME ARROW SET = type, NUMBER** , come il precedente ma col simbolo terminale NUMBER che identifica la posizione nell'array (es. nomeArray -> set = 2,0)
    - **NAME ARROW DELETE = type** , come per ADD ma rimuove l'ultimo elemento inserito

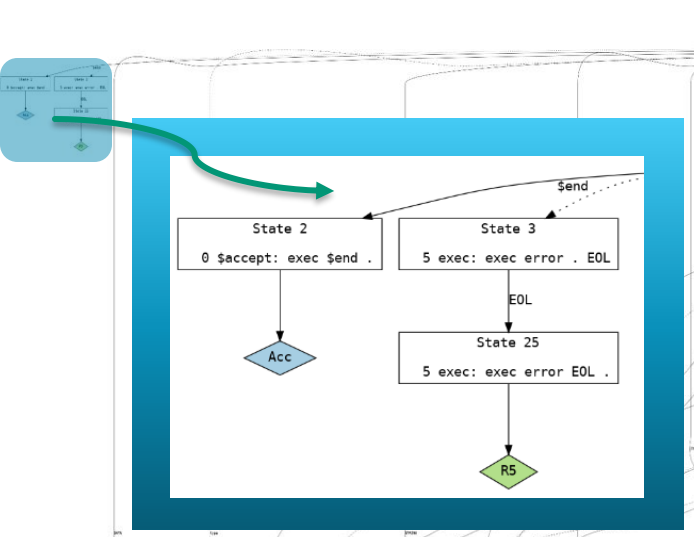
```
exec:       ->
                exec stmt EOL
                exec function EOL
                exec TERM EOL
                exec error EOL
;

;
stmt:       ->  IF exp THEN listStmt
                IF exp THEN listStmt ELSE listStmt
                WHILE exp DO listStmt
                exp
;
function:   ->  CMD NAME '(' argsList ')' '=' listStmt RET exp ';'
                CMD NAME '(' argsList ')' '=' listStmt
;
listStmt:   ->
                stmt ';' listStmt
;
exp:        ->
                '(' exp ')'
                type
                FUNC exp
                FUNCDEV exp
                SYSTEM
                NAME
                NAME '=' exp
                INTERVAL exp '-' exp '-' exp
                NAME '(' explistStmt ')'
                liste
;
liste:      ->  ARRAY nameType NAME '(' NUMBER ')'
                NAME ARROW  ADD '=' type
                NAME ARROW GET
                NAME ARROW  GET '=' NUMBER
                NAME ARROW  SET '=' type ',' NUMBER
                NAME ARROW REMOVE
;
nameType:   ->  CHAR
                INTEGER
                DEVICE
;
nameType:       ->      CHAR
                        INTEGER
                        DEVICE
;
type:                   INSERT STRING
                        NUMBER
                        STRING
                        INSERT STRING ARROW '[' argsListDevice ']'
                        DATA
;


explistStmt:    ->      exp
                        exp ',' explistStmt
;
argsList:       ->      NAME
                        NAME ',' argsList
;
argsListDevic:  ->      STRING
                        STRING ',' argsListDevice
```

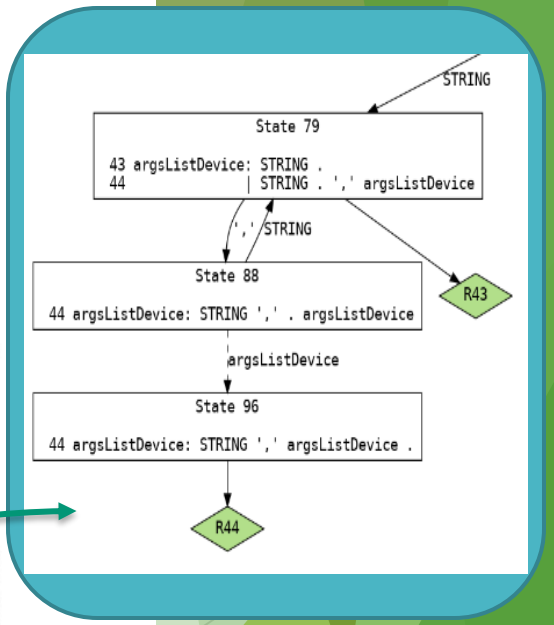
Forma semplificata della grammatica utilizzata

Caratteristiche del Linguaggio -> Parse Tree

```
mario@DESKTOP-0U6L0FJ:/mnt/c/4_Linux/funzionante$ bison -g parser.y
mario@DESKTOP-0U6L0FJ:/mnt/c/4_Linux/funzionante$ dot -Tpng parser.dot > output.png
```

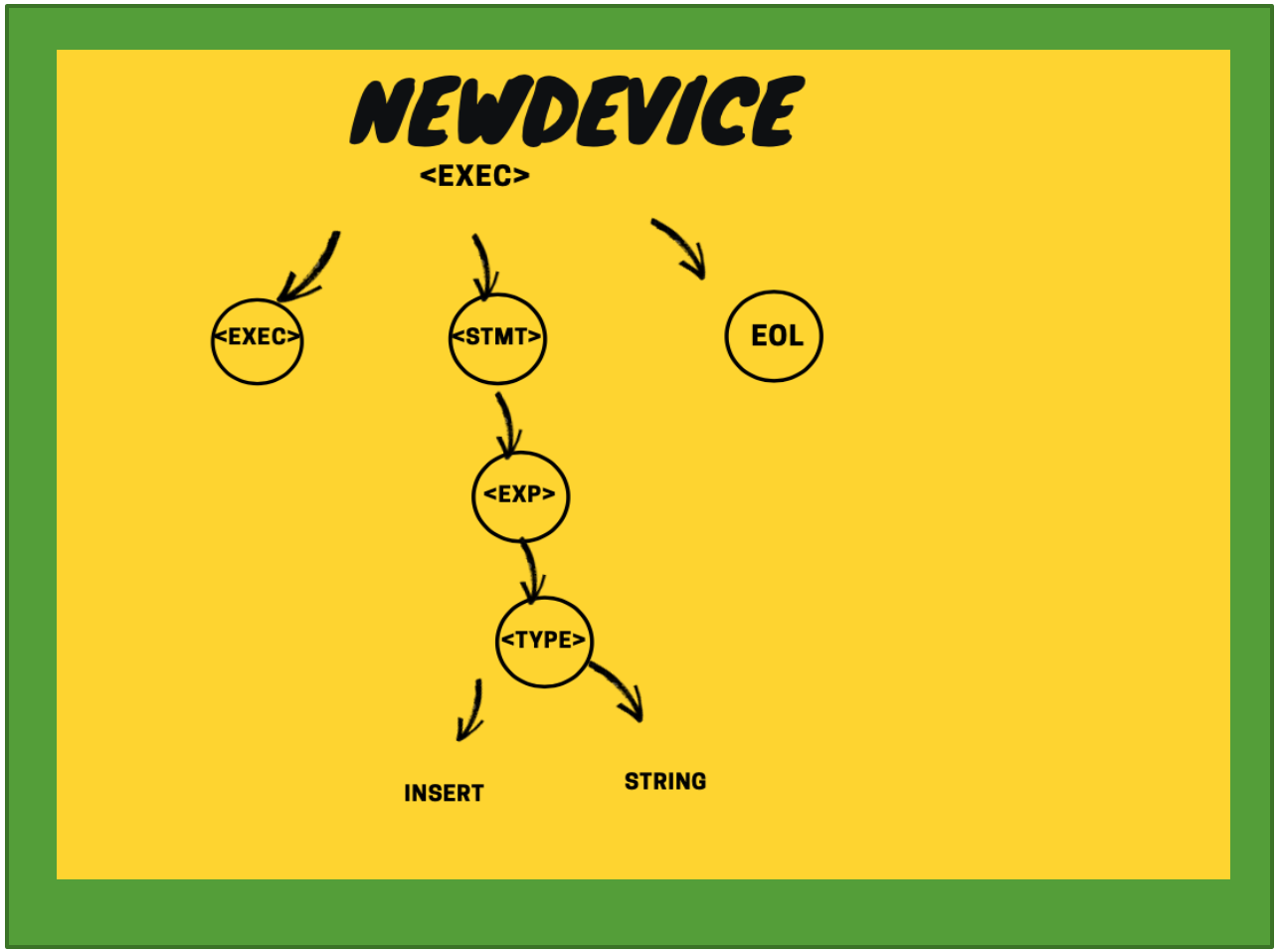
Albero Sintattico





Caratteristiche del Linguaggio -> Analizzatore Sintattico

- **Esempi di applicazioni e relativo albero**
  - Creazione del device
    - newDevice ''valvola''



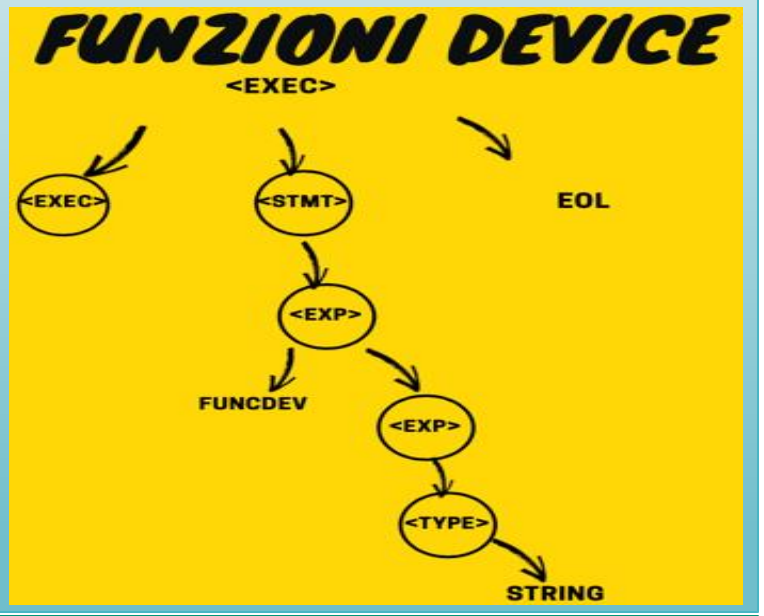

```
exec:       ->
                exec stmt EOL
                exec function EOL
                exec TERM EOL
                exec error EOL

;

;
stmt:       ->  IF exp THEN listStmt
                IF exp THEN listStmt ELSE listStmt
                WHILE exp DO listStmt
                exp
;
function:   ->  CMD NAME '(' argsList ')' '=' listStmt RET exp ';'
                CMD NAME '(' argsList ')' '=' listStmt

;

listStmt:   ->
                stmt ';' listStmt
;
exp:        ->
                '(' exp ')'
                type
                FUNC exp
                FUNCDEV exp
                SYSTEM
                NAME
                NAME '=' exp
                INTERVAL exp '-' exp '-' exp
                NAME '(' explistStmt ')'
                liste
;

liste:      ->  ARRAY nameType NAME '(' NUMBER ')'
                NAME ARROW  ADD '=' type
                NAME ARROW GET
                NAME ARROW  GET '=' NUMBER
                NAME ARROW  SET '=' type ',' NUMBER
                NAME ARROW REMOVE
;
nameType:   ->  CHAR
                INTEGER
                DEVICE
;
nameType:       ->      CHAR
                        INTEGER
                        DEVICE
;
type:                   INSERT STRING
                        NUMBER
                        STRING
                        INSERT STRING ARROW '[' argsListDevice ']'
                        DATA
;


explistStmt:    ->      exp
                        exp ',' explistStmt
;
argsList:       ->      NAME
                        NAME ',' argsList
;

argsListDevic:  ->      STRING
                        STRING ',' argsListDevice
```

Forma semplificata della grammatica utilizzata

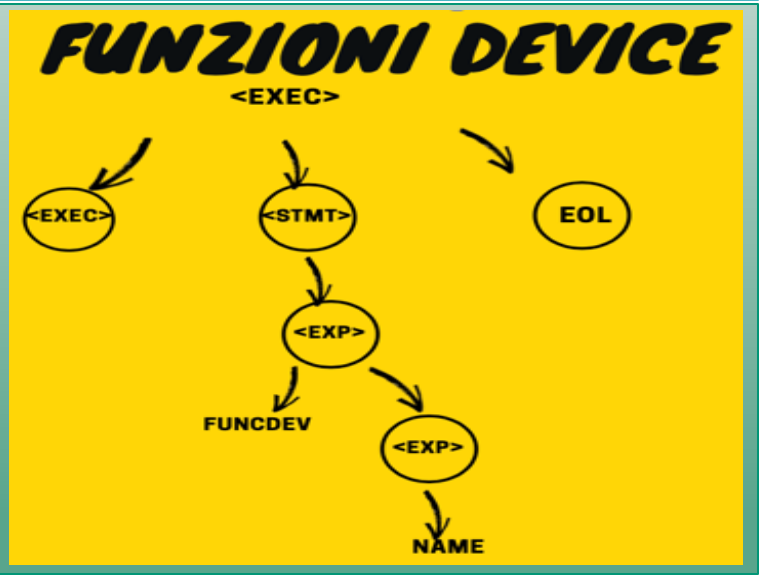
Caratteristiche del Linguaggio -> Analizzatore Sintattico

- **Esempi di applicazioni e relativo albero**
  - **FUNCDEV**, tutte le funzioni per i device seguono un albero di questo tipo

- switchOn ‘’valvola”
- switchOff ‘’valvola”
- connect ‘’valvola”
- reconnect ‘’valvola”
- status ‘’valvola”



- L’ultimo nodo, il simbolo terminale STRING, può essere sostituito da qualcosa di più complesso e di conseguenza di avrà un albero diverso; ad esempio, dopo aver creato la variabile valvola = ‘’bosch”
  - switchOn valvola

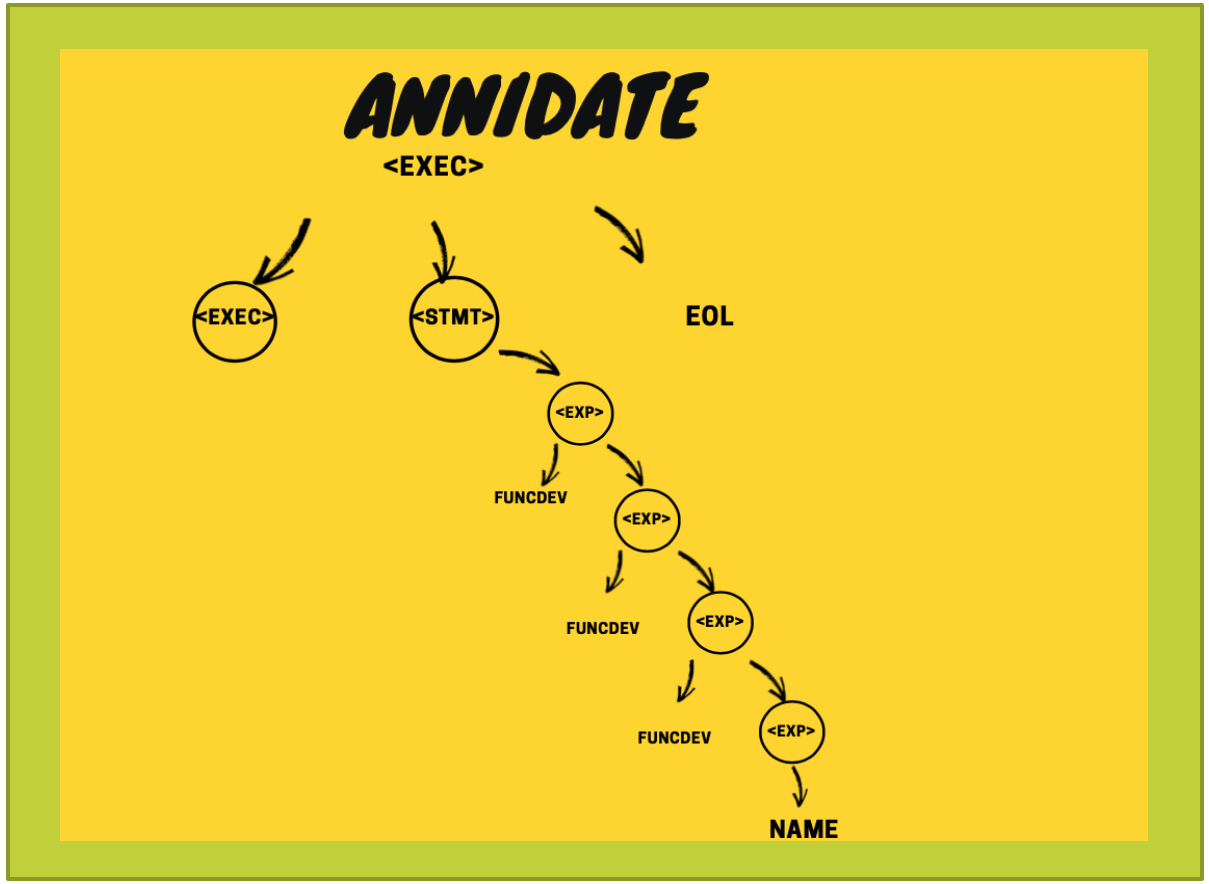



```
exec:       ->
                exec stmt EOL
                exec function EOL
                exec TERM EOL
                exec error EOL
;

;
stmt:       ->  IF exp THEN listStmt
                IF exp THEN listStmt ELSE listStmt
                WHILE exp DO listStmt
                exp
;
function:   ->  CMD NAME '(' argsList ')' '=' listStmt RET exp ';'
                CMD NAME '(' argsList ')' '=' listStmt

;
listStmt:   ->
                stmt ';' listStmt
;
exp:        ->
                '(' exp ')'
                type
                FUNC exp
                FUNCDEV exp
                SYSTEM
                NAME
                NAME '=' exp
                INTERVAL exp '-' exp '-' exp
                NAME '(' explistStmt ')'
                liste
;

liste:      ->  ARRAY nameType NAME '(' NUMBER ')'
                NAME ARROW  ADD '=' type
                NAME ARROW GET
                NAME ARROW  GET '=' NUMBER
                NAME ARROW  SET '=' type ',' NUMBER
                NAME ARROW REMOVE
;
nameType:   ->  CHAR
                INTEGER
                DEVICE
;
nameType:       ->      CHAR
                        INTEGER
                        DEVICE
;
type:                   INSERT STRING
                        NUMBER
                        STRING
                        INSERT STRING ARROW '[' argsListDevice ']'
                        DATA
;


explistStmt:    ->      exp
                        exp ',' explistStmt
;
argsList:       ->      NAME
                        NAME ',' argsList
;

argsListDevic:  ->      STRING
                        STRING ',' argsListDevice
```

Forma semplificata della grammatica utilizzata

Caratteristiche del Linguaggio -> Analizzatore Sintattico

- **Esempi di applicazioni e relativo albero**
  - **Funzioni Embedded Annidate**
    - h="Pompa"
    - delete switchOff connect h
      - verificato se il device è raggiungibile, lo spegne e viene eliminato




```
exec:       ->
                exec stmt EOL
                exec function EOL
                exec TERM EOL
                exec error EOL
;

;
stmt:       ->  IF exp THEN listStmt
                IF exp THEN listStmt ELSE listStmt
                WHILE exp DO listStmt
                exp
;
function:   ->  CMD NAME '(' argsList ')' '=' listStmt RET exp ';'
                CMD NAME '(' argsList ')' '=' listStmt

;
listStmt:   ->
                stmt ';' listStmt
;
exp:        ->
                '(' exp ')'
                type
                FUNC exp
                FUNCDEV exp
                SYSTEM
                NAME
                NAME '=' exp
                INTERVAL exp '-' exp '-' exp
                NAME '(' explistStmt ')'
                liste
;

liste:      ->  ARRAY nameType NAME '(' NUMBER ')'
                NAME ARROW  ADD '=' type
                NAME ARROW GET
                NAME ARROW  GET '=' NUMBER
                NAME ARROW  SET '=' type ',' NUMBER
                NAME ARROW REMOVE
;
nameType:   ->  CHAR
                INTEGER
                DEVICE
;
nameType:       ->      CHAR
                        INTEGER
                        DEVICE
;
type:                   INSERT STRING
                        NUMBER
                        STRING
                        INSERT STRING ARROW '[' argsListDevice ']'
                        DATA
;


explistStmt:    ->      exp
                        exp ',' explistStmt
;

argsList:       ->      NAME
                        NAME ',' argsList
;

argsListDevic:  ->      STRING
                        STRING ',' argsListDevice
```

Forma semplificata della grammatica utilizzata

Basso Livello

Eval

CallbuiltIn (F)

Funzioni Embedded	
connect	F.Embedded
readFile	F.Embedded
insert	F.Embedded
device	Tipo
bye	abort
interval	F.Embedded
switchOn	F.Embedded
switchOff	F.Embedded
delete	F.Embedded
reconnect	F.Embedded
arrow	Indicatore ->
status	F.Embedded
newDevice	F.Embedded
print	F.Embedded

CallbuiltArray (M)

Array	
newArray	F.Array
integer	Identificatore
char	Identificatore
integer	Identificatore
device	Identificatore
set	F.Embedded
get	F.Embedded
add	F.Embedded
remove	F.Embedded

CallbuiltInSystem(O)

System	
help	F.System
clear	F.System
sleep	F.Embedded

Basso Livello -> Tabella dei Simboli

Una **tabella dei simboli** è una struttura dati, in cui ogni simbolo è associato con le informazioni relative alla sua dichiarazioni.

La tabella dei simboli memorizza quindi le informazioni relative sul simbolo nella tabella, nell'apposita cella di memoria, ognuna delle quali contiene un puntatore al nome e un elenco di riferimenti ad altre struttue o valori.

Implementazione :

1- Struttura symbol, definisce la singola cella della tabella.

2- Definizione di una tabella nominata symtab (array di tipo symbol) di dimensione DIMHASH

Utilizzo fatto:

- Principalmente per tenere traccia dei nomi utilizzati in input e definire riferimenti ad altre strutture, raggiungibili attraverso il simbolo inserito nella tabella.
- Ricerca di un simbolo, con la quale viene identificato un Device ad esempio, attraverso la tecnica di ricerca, nota come **hashing** con probe lineare.

1)

```
/* Tabella dei Simboli */
struct symbol {
  char *name;
  char *value;
  double dim;
  struct ast *func;          /* stmt per le funzioni */
  struct ast *dev;          /* stmt per le funzioni */
  struct argsList *syms;     /* Lista dei simboli */
  struct ast *arr;
  struct ast *ret;
};
```

2)

```
/* Tabella di dimensione fissa */
#define DIMHASH 10000
struct symbol symtab[DIMHASH];
```

Tipi Compositi -> Device

- Fasi della creazione, semplice, di un tipo composito Device attraverso un esempio
  - Analisi lessicale e tabella dei simboli

Sentence: newDevice “deviceNord”

Lexer

```
"newDevice" {
     yylval.func = B_insertDevice;
     return INSERT;
}
```

```
/* Stringhe alfanumeriche.*/
["][a-zA-Z][a-zA-Z0-9]*["] {

     int dimString= strlen(yytext)+1;
     char subbuff[dimString-2];
     memmove( subbuff, &yytext[1], dimString-3 );
     subbuff[dimString-3] = '\0';
     yylval.s = search(subbuff, NULL);
     return STRING;
}
```

Serra.c

```
struct symbol *search(char* sym, char* type){
  struct symbol *symptr = &symtab[ symhash(sym) % DIMHASH ];
  int symcount = DIMHASH;

  while(--symcount >= 0) {
    if(symptr->name && !strcmp(symptr->name, sym)) {
        return symptr;
    }

    if(!symptr->name) { /* NUOVO SIMBOLO */
      if(!type){
         symptr->name =type strdup(sym);
         symptr->value = 0;
         symptr->dev = NULL;
         symptr->func = NULL;
         symptr->syms = NULL;  //LISTA DI SIMBOLI
          return symptr;
      }
      else{
    ..............................

    if(++symptr >= symtab + DIMHASH) {
        symptr = symtab;
    }
```

Utilizza una funzione hash, **symhash**, per trasformare la stringa in una entry number nella tabella, quindi controlla la entry e, se è già presa da un simbolo diverso, esegue la scansione lineare fino a trovare una voce libera.
Controlla prima il tipo di ricerca, in base al secondo parametro.

Tipi Compositi -> Device

Sentence: newDevice “deviceNord”

Lexer

```
"newDevice" {
     yylval.func = B_insertDevice;
     return INSERT;
}
```

```
/* Stringhe alfanumeriche.*/
["][a-zA-Z][a-zA-Z0-9]*["] {

     int dimString= strlen(yytext)+1;
     char subbuff[dimString-2];
     memmove( subbuff, &yytext[1], dimString-3 );
     subbuff[dimString-3] = '\0';
     yylval.s = search(subbuff, NULL);
     return STRING;
}
```

Parser

```
exec:  /* nothing */
    | exec stmt EOL { run ($2); }
    .............

stmt: ........                    exp:
    | exp                            | type
                                     .............

type:
    INSERT STRING {
        defSymRef((struct symbol *)$2, NULL, NULL, NULL);
        $$ = newDev((struct symbol *)$2,NULL);
    }
```

Serra.c

```
struct ast * newDev(struct symbol *ps, struct argsList *l)
{
  struct device *d = malloc(sizeof(struct device));
  struct argsList *lpt;

  if(!d) {
    yyerror("Spazio di memoria insufficiente\n");
    exit(0);
  }

  char *nameSymbol;
  char *devNameList;
  nameSymbol= ps->name;
  nameSymbol= symhashDev(nameSymbol);

  struct symbol *symbolDev= NULL;
  symbolDev = search(nameSymbol, "searchSym");

  if(symbolDev==NULL)    //SE IL DISPOSITIVO NON ESISTE
  {
       struct symbol *sym= search(nameSymbol, NULL);
       d->nodetype = 'D';
       d->status = 0;  //LO PONGO CON STATO SPENTO DI DEFAULT
       d->s= sym;
       d->l = (struct ast *)l;
       d->c=0;
       printf("Dispositivo inserito con successo con ID: %s \n", nameSymbol);

      struct ast *dino= (struct ast *)d;
      (sym->dev)=dino;
      ..............................

  }else{
    printf("Dispositivo già Esistente con ID: %s\n", nameSymbol);
      d->s=symbolDev;
      d->c=1;
  }
    d->nodetype = 'D';
  return (struct ast *)d;
}
```

	Tabella Simboli				
Indice	**Name**	**Value**	**Function**	**Device**	**List**
Hash+Name	DeviceNord	0	NULL	xxx	NULL
	Ogetto Device				
	NodeType	**Status**		**S (Symbol)**	
xxx	D	0	Hash+Name	NULL	

Tipi Compositi -> Device

- Fasi della creazione con lista di Device di un tipo composito Device attraverso un esempio
  - Analisi lessicale e tabella dei simboli
  - Obiettivo: Usando una lista di Device si definiscono i collegamenti tra Device appartenente alla stessa rete, potendo così definire diverse topologie.

Sentence: *newDevice “deviceNord” -> [“device1”, “device2”, ... , “deviceN”]*

Lexer

```
"newDevice" {
      yylval.func = B_insertDevice;
      return INSERT;
}
```

```
/* Stringhe alfanumeriche.*/
["][a-zA-Z][a-zA-Z0-9]*["] {

      int dimString= strlen(yytext)+1;
      char subbuff[dimString-2];
      memmove( subbuff, &yytext[1], dimString-3 );
      subbuff[dimString-3] = '\0';
      yylval.s = search(subbuff, NULL);
      return STRING;
}
```

Serra.c

```
struct symbol *search(char* sym, char* type){
  struct symbol *symptr = &symtab[ symhash(sym) % DIMHASH ];
  int symcount = DIMHASH;

  while(--symcount >= 0) {
    if(symptr->name && !strcmp(symptr->name, sym)) {
        return symptr;
    }

    if(!symptr->name) { /* NUOVO SIMBOLO */
      if(!type){
        symptr->name =type strdup(sym);
        symptr->value = 0;
        symptr->dev = NULL;
        symptr->func = NULL;
        symptr->syms = NULL;  //LISTA DI SIMBOLI
         return symptr;
      }
      else{
    ..............................

    if(++symptr >= symtab + DIMHASH) {
        symptr = symtab;
    }
```

La lista dei simboli dei Device collegati verrà inserita successivamente e il puntatore a questa verrà memorizzato in syms nella struttura symbol del Device che si sta creando, quindi oltre a puntare alla struttura del nuovo Device, punterà alla lista dei Device.

Tipi Compositi -> Device

Sentence: *newDevice “deviceNord” -> [“device1”, “device2”, ... , “deviceN”]*

Parser

```
exec:  /* nothing */
    | exec stmt EOL { run ($2); }
    .............

stmt: ........                    exp:
    | exp                            | type
                                     .............

type:  ......
    | INSERT STRING ARROW '[' argsListDevice ']'{
        defSymRef((struct symbol *)$2, $5, NULL, NULL);
        $$ = newDev((struct symbol *)$2,$5);
    }


argsListDevice: STRING {
                    $$ = newargsList((struct symbol *)$1, NULL);
                }

              | STRING ',' argsListDevice {

                    $$ = newargsList((struct symbol *)$1, $3);
                }
;
```

Casi: 1- Lista di un solo Device
2- Lista di n Device

Nel primo caso, la regola termina è viene richiamata la funzione newargsList

newargsList viene usata in due casi:
- nelle funzioni per contenere la lista dei nomi delle variabili nel momento in cui la funzione viene definita.
- nella creazione di un oggetto device serve per memorizzare la lista dei device a cui è collegato

```
struct argsList *newargsList(struct symbol *sym,
                             struct argsList *next
                            ){
  struct argsList *sl = malloc(sizeof(struct argsList));

  if(!sl) {
    yyerror("Spazio di memoria insufficiente\n");
    exit(0);
  }

  sl->sym = sym;
  (sl->sym)->name=sym->name;
  sl->next = next;
  return sl;
}
```

Tipi Compositi -> Device

Sentence: *newDevice “deviceNord” -> [“device1”, “device2”, ... , “deviceN”]*

Parser

```
exec:  /* nothing */
    | exec stmt EOL { run ($2); }
    .............

stmt: ........                    exp:
    | exp                            | type
                                     .............

type:  ......
    | INSERT STRING ARROW '[' argsListDevice ']'{
        defSymRef((struct symbol *)$2, $5, NULL, NULL);
        $$ = newDev((struct symbol *)$2,$5);
    }


argsListDevice: STRING {
                    $$ = newargsList((struct symbol *)$1, NULL);
                }

              | STRING ',' argsListDevice {

                    $$ = newargsList((struct symbol *)$1, $3);
                }
;
```

Nel secondo caso, la regola non termina subito, deriverà ricorsivamente se stessa finché non termina, a quel punto risalirà l'albero delle ricorsioni richiamando ad ogni ricorsione la funzione newargsList.

Esempio con due dispositivi collegati :

1° ricorsione: -> [dev1, dev2]

2° ricorsione: -> [dev2, NULL]

Risale l'albero, ed esegue :

1- newargsList ("dev2", NULL);

Torna un puntatore ad argsList

*ptr ->

2- newargsList ("dev1", "dev2");

Torna un puntatore ad argsList

*ptr ->

Tipi Compositi -> Device

Sentence: newDevice “deviceNord” -> [“device1”, “device2”, ... , “deviceN”]

```
"newDevice" {
     yylval.func = B_insertDevice;
     return INSERT;
}
```

```
/* Stringhe alfanumeriche.*/
["][a-zA-Z][a-zA-Z0-9]*["] {

     int dimString= strlen(yytext)+1;
     char subbuff[dimString-2];
     memmove( subbuff, &yytext[1], dimString-3 );
     subbuff[dimString-3] = '\0';
     yylval.s = search(subbuff, NULL);
     return STRING;
}
```

Parser

```
exec:  /* nothing */
     | exec stmt EOL { run ($2); }
     .............

stmt: ........                    exp:
     | exp                           | type
                                     .............


type:  ......
     | INSERT STRING ARROW '[' argsListDevice ']'{
          defSymRef((struct symbol *)$2, $5, NULL, NULL);
          $$ = newDev((struct symbol *)$2,$5);
     }
```

La funzione defSymRef collega nella struct symbol passata come parametro i riferimenti alla argsList e all'AST che definisce.

```
//Collega nella struct symbol passata
//come parametro i riferimenti alla argsList
//e all'AST che definisce
void defSymRef(struct symbol *name,
               struct argsList *syms,
               struct ast *func,
               struct ast*ret)
{
  if(name->syms){ argsListfree(name->syms); }

  name->syms = syms;
  name->ret=ret;
}
```

Prende tre parametri: L'entry symbol che si è appena creata e una lista di device a cui dovranno collegarsi.

Quindi nel caso l’operazione risulta superflua dato che il device non deve al momento essere collegato ad altri device.

Tipi Compositi -> Device

Sentence: newDevice “deviceNord” -> [“device1”, “device2”, ... , “deviceN”]

	Tabella Simboli				
Indice (Hash+Name)	**Name**	**Value**	**Function**	**Device**	**List**
Device_c#....	DeviceNord	0	NULL	xxx	NULL
Device_1#....	Device1	0	NULL	YYY	NULL
Device_2#....	Device2	0	NULL	ZZZ	NULL
	Ogetto Device				
	NodeType	**Status**	**S (Symbol)**	**I (argList)**	
xxx	D	0	Devicenuovo_c1	{Device1, ..}	
yyy	D	0	Devicenuovo_d1	NULL	
zzz	D	0	Device_A1	NULL	

}

Parser

```
exec:  /* nothing */
    | exec stmt EOL { run ($2); }
    ............

stmt: ........                    exp:
    | exp                            | type
                                     ............

type:  ......
    | INSERT STRING ARROW '[' argsListDevice ']'{
        defSymRef((struct symbol *)$2, $5, NULL, NULL);
        $$ = newDev((struct symbol *)$2,$5);
    }
```

Serra.c

```
.....................

if(l!= NULL) {
   struct symbol *ptrSymDevices= search(nameSymbol, "searchSym");
   //ptrSymDevices se diverso da null punta al simbolo.
   //con il searchSym si sta soltanto effettuando la ricerca,
   //non sta effettuando la creazione du un nuovo simbolo nella
   //tabella dei simboli.
   printf("%s connesso con ->    ", nameSymbol);
   int countDeviceUnknown = 0;
   printf("[");

   for( lpt=l; lpt; lpt = lpt->next){
          devNameList = lpt->sym->name;
          const char strchSearchChar= '#';
          if( !strchr(devNameList,strchSearchChar))
                 devNameList = symhashDev(devNameList);

          if(strcmp(devNameList, nameSymbol)){  //SOLO SE DIVERSI
              printf(" [%s] ", devNameList);

              if( !search(devNameList, "searchSym")){
               /* Se ptrSymDevices->dev non è settato, cioè se dev,
                * puntatore ad un nodo struttura device è NULL,
                * allora bisogna creare ancora il device, anche
                * se il simbolo esiste.*/
                   printf("* ", ptrSymDevices->dev);
                    countDeviceUnknown++;
              }
          }else{
              printf(" itself ");
          }

          if(lpt->next != NULL)
                   printf("-");
   }
   printf("]\n");

   if(countDeviceUnknown > 0)
       printf("Devices con (*) sconosciuti, inserire devices\n");
}
```

Basso Livello -> QUI PARLARE DI EVAL

Eval

CallbuiltIn (F)

Funzioni Embedded	
connect	F.Embedded
readFile	F.Embedded
insert	F.Embedded
device	Tipo
bye	abort
interval	F.Embedded
switchOn	F.Embedded
switchOff	F.Embedded
delete	F.Embedded
reconnect	F.Embedded
arrow	Indicatore ->
status	F.Embedded
newDevice	F.Embedded
print	F.Embedded

CallbuiltArray (M)

Array	
newArray	F.Array
integer	Identificatore
char	Identificatore
integer	Identificatore
device	Identificatore
set	F.Embedded
get	F.Embedded
add	F.Embedded
remove	F.Embedded

CallbuiltInSystem(O)

System	
help	F.System
clear	F.System
sleep	F.Embedded

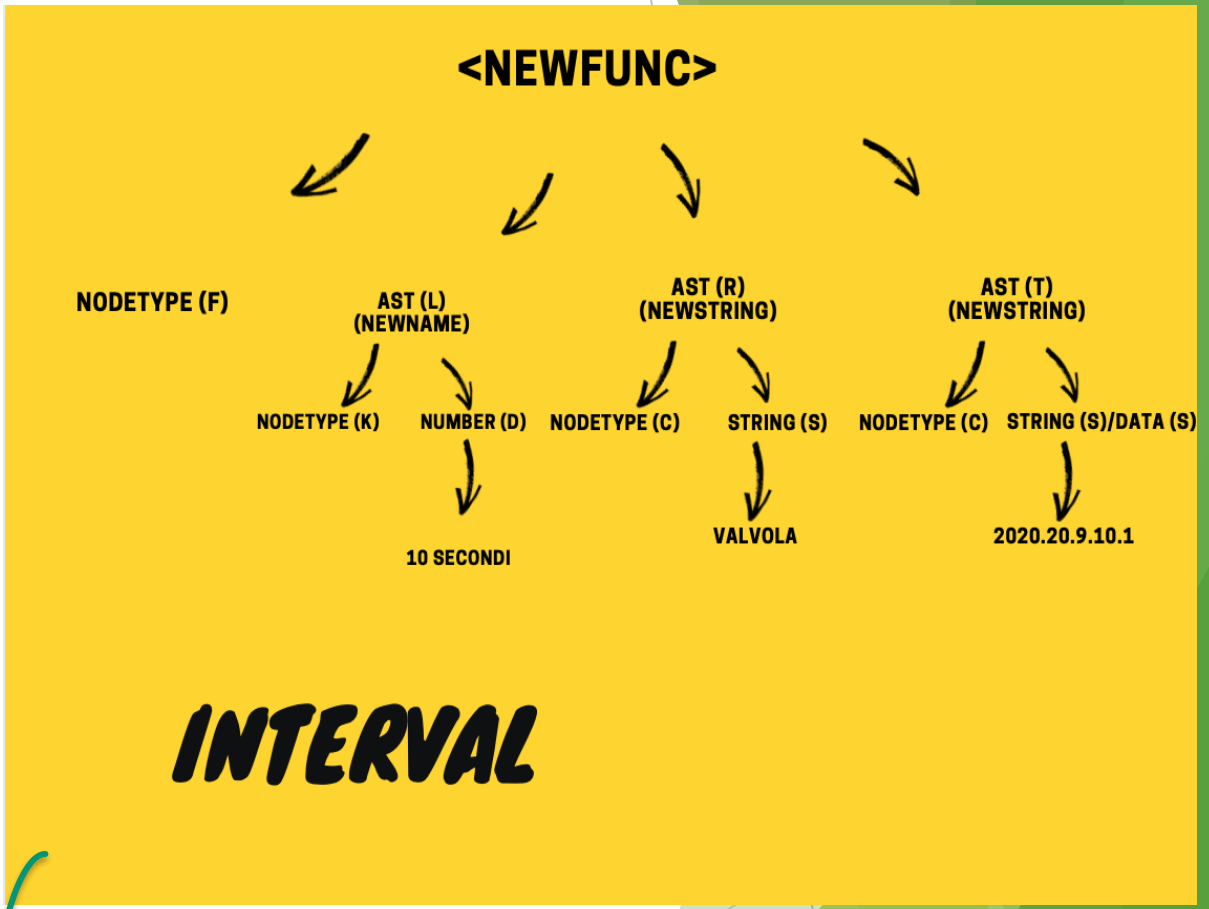
Funzioni device->Interval:

Sentence: *interval "valvola"-10-2020.12.9.1*

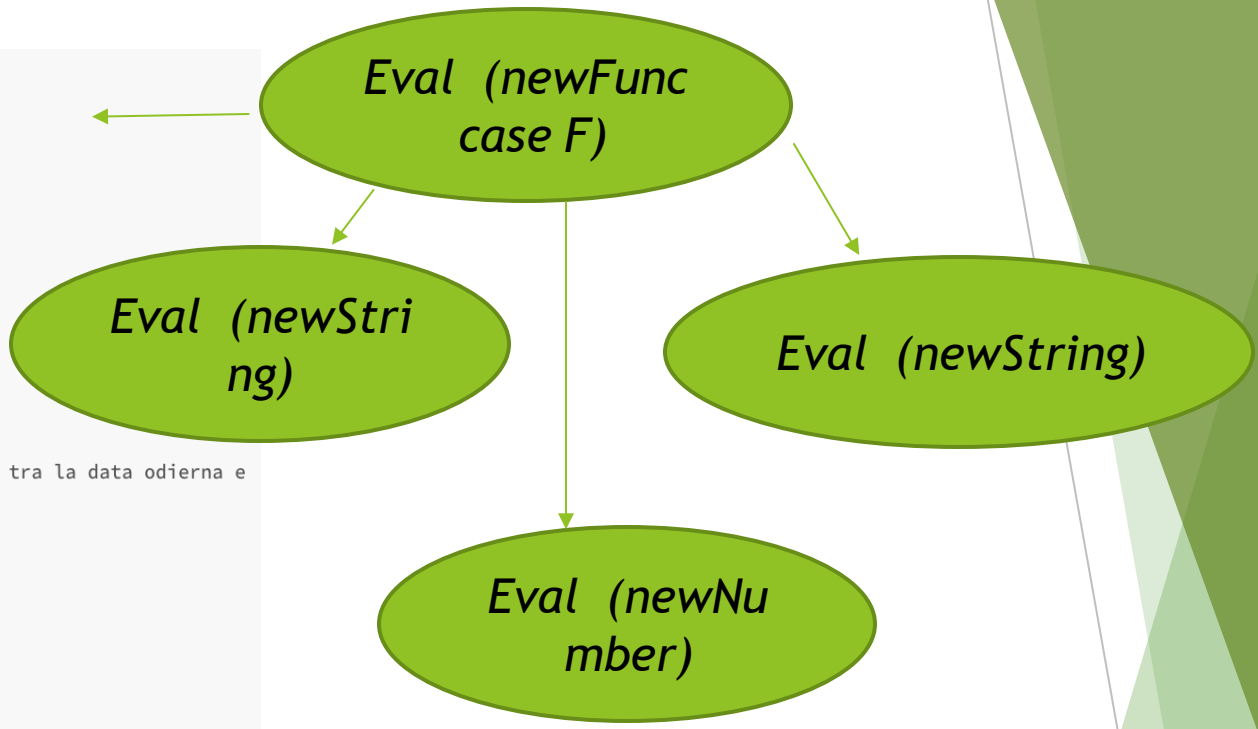
```
"interval"     { yylval.func = B_interval; return INTERVAL;   }

["][a-zA-Z][a-zA-Z0-9]*["] {
                              //riconosce Stringhe, già spiegato prima
                            }

[2][0-9]{3}"."(0?[1-9]|1[012])"."(0?[1-9]|[12][0-9]|3[01])"."(0?[1-9]|1[0-9]|2[0-3])"."(0?[0-9]|[1-5][0-9]) {
                        yylval.s = search(yytext, NULL);
                        return DATA;
}

[0-9]+               { yylval.d = atof(yytext); return NUMBER;     }
```

Parser

```
exec:  /* nothing */
     | exec stmt EOL { run ($2); }
     .............

stmt: ........          exp:
     | exp                 | type
                           .............

type:
      I NUMBER     $$ = newnum($1);
      I STRING     $$ = newString((struct symbol *)$1);
      I DATA       $$ = newString((struct symbol *)$1);
;


exp:....
   INTERVAL exp '-' exp '-' exp     $$ = newfunc($1, $2, $4, (struct ast *)$6);
```



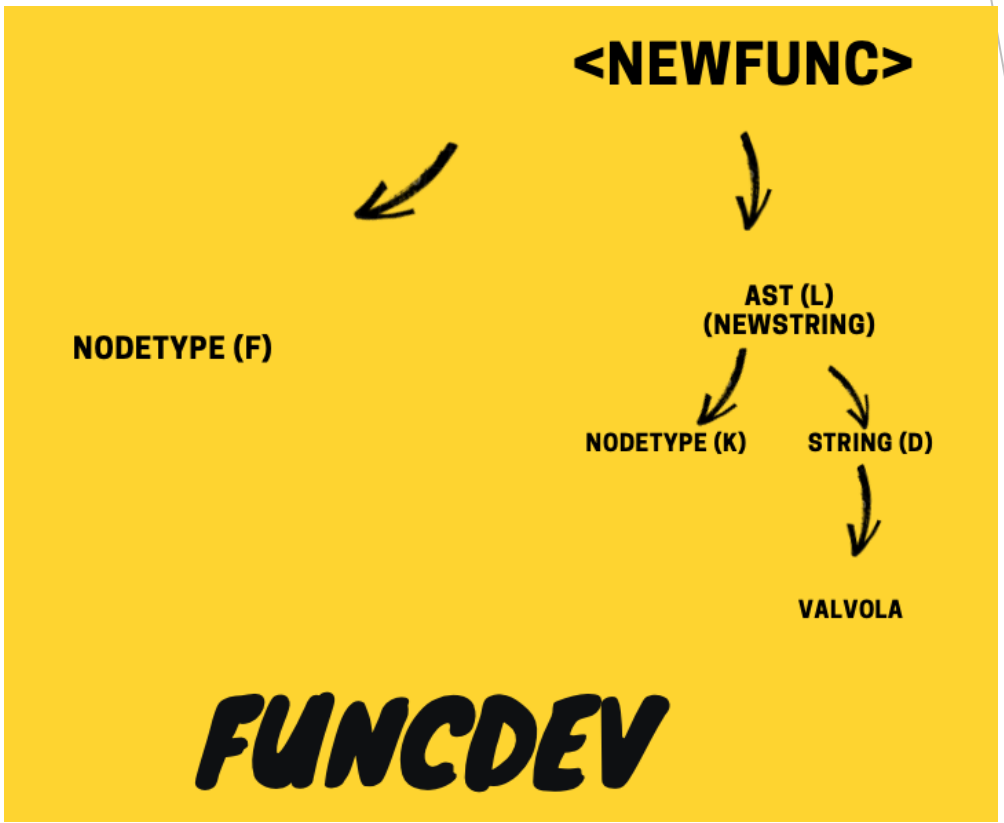
```
eval (){
  .....
  case 'F':
         callbuiltin((struct funcBuiltIn *)a);
  .....
}

callbuiltin (f){
......
    case B_interval:
            //nome device:
         1. Scorrimento albero:
            x=eval(f->l);           //device
            r= atoi (eval(f->r));   //numero secondi
            m=eval(f->t);           //data

         2. Calcolo di quanti secondi calcolo in secondi della differenza tra la data odierna e
            la data come parametro:
            float seconds=data_op (m);

         3. interval(r, z, seconds);
    ;
.....
}


void interval (int r, char *v, float seconds){
    pthread_t threads;
    n->r=r;
    n->v=v;
    n->seconds=seconds;

    pthread_create (&threads, NULL, annaffia, (void *)n);
}

void *annaffia(void *x) {
    sleep (x->seconds);

        //Operazioni:
    struct symbol *h= switchOn(y->v);
    if(h){
        sleep(x->r); //time.h, 1000 è 1 secondo, aspettiamo min secondi
        switchOff(temp);
    }

}
```



Sentence: *interval "valvola"-10-2020.12.9.1*

Funzioni device->SwitchOn, SwitchOff, Status:

```
"connect"               { yylval.func = B_connect; return FUNCDEV;      }
"reconnect"             { yylval.func = B_reconnect; return FUNCDEV;    }
"status"                { yylval.func = B_status; return FUNCDEV;       }
"switchOn"              { yylval.func = B_switchOn; return FUNCDEV;     }
"switchOff"             { yylval.func = B_switchOff; return FUNCDEV;    }
"delete"                { yylval.func = B_delete; return FUNCDEV;       }

["][a-zA-Z][a-zA-Z0-9]*["] {
                            int dimString= strlen(yytext)+1;
                            char subbuff[dimString-2];
                            memmove( subbuff, &yytext[1], dimString-3 );
                            subbuff[dimString-3] = '\0';
                            yylval.s = search(subbuff, NULL);
                            return STRING;
                          }
```



Parser

```
exec:  /* nothing */
     | exec stmt EOL { run ($2); }
     .............
stmt: ........
     | exp
```

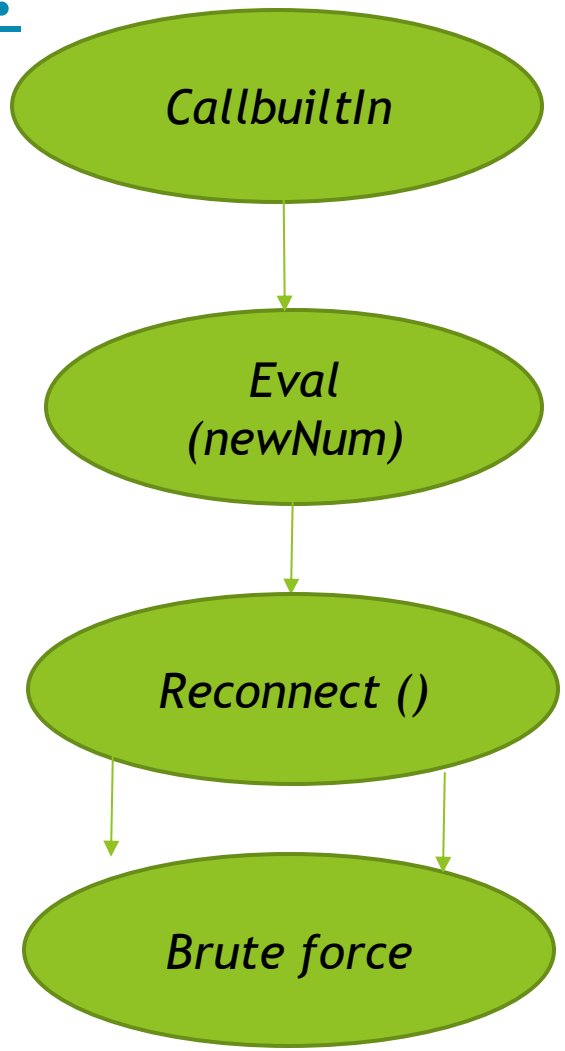
```
exp:....
   type
   | FUNCDEV exp     $$ = newfunc($1, $2, NULL, NULL);


type:
      STRING      $$ = newString((struct symbol *)$1);
;
```

Sentence:
- *SwitchOn "valvola"*
- *SwitchOff "valvola"*
- *Connect "valvola"*
- *Status "valvola"*

Funzioni device->reconnect:

Sentence: reconnect "valvola"

```
callbuiltin (f){
    ......
   case B_reconnect:
        //v=strdup(eval(f->l));
        x=eval(f->l);

        if(symDev){
           printf("Dispositivo Esistente\nRichiesta Riconnessione...\n");
        }else{
           reconnect (x);

        }
    ........
}

reconnect (char *nome)
    for (int i=0; i<2; i++){
        pthread_create (&(threads[i]), NULL, brute_force, (void *)nome);


void *brute_force (void *x)
    sleep (1+rand()%30);
    connect(v);
```



Basso Livello -> QUI PARLARE DI EVAL

Eval

CallbuiltIn (F)

Funzioni Embedded	
connect	F.Embedded
readFile	F.Embedded
insert	F.Embedded
device	Tipo
bye	abort
interval	F.Embedded
switchOn	F.Embedded
switchOff	F.Embedded
delete	F.Embedded
reconnect	F.Embedded
arrow	Indicatore ->
status	F.Embedded
newDevice	F.Embedded
print	F.Embedded

CallbuiltArray (M)

Array	
newArray	F.Array
integer	Identificatore
char	Identificatore
integer	Identificatore
device	Identificatore
set	F.Embedded
get	F.Embedded
add	F.Embedded
remove	F.Embedded

CallbuiltInSystem(O)

System	
help	F.System
clear	F.System
sleep	F.Embedded

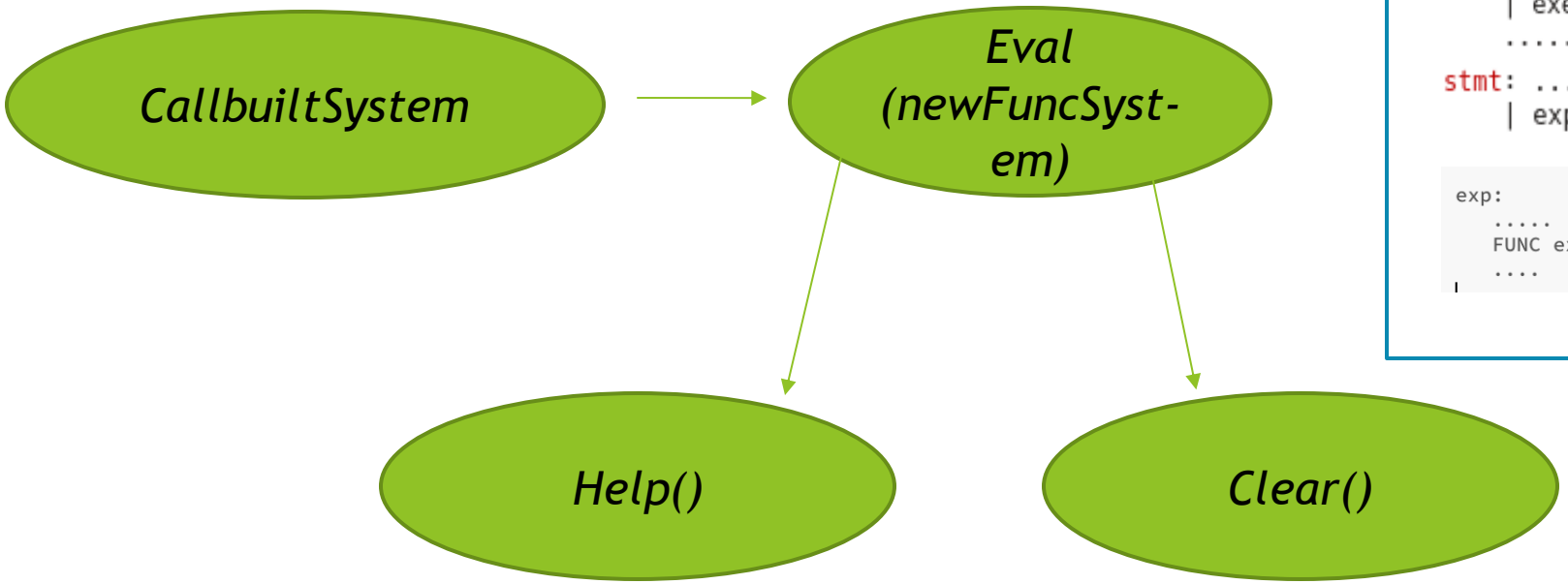
Funzioni di sistema:

I token per definire le funzioni di "sistema" sono:

- FUNC individua invece tutte le funzioni generali non strettamente legate ai device.

  Il token verrà riconosciuto quando l'utente scriverà: "print", "sleep" o "readFile".

- SYSTEM individua invece tutte le funzioni che invocano funzioni del sistema su cui viene eseguito il programma, e che non sono strettamente legate ai device. Le espressioni sono: "clear" o "help":
  - Sono funzioni che non prendono paramtri e di fatto non effettuano scorrimento di nodi ast



Parser

```
exec:  /* nothing */
     | exec stmt EOL { run ($2); }
     .............
stmt: ........
     | exp
```

```
exp:
   .....
   FUNC exp   { $$ = newfunc($1, $2, NULL, NULL);    }
   ....
```

ReadFile:

Sentence:
- ./serra
- readFile "nomeFile"

```
"readFile"     { yylval.func = B_readFile; return FUNC;              }
<<EOF>>        {  if(flag==0) yyrestart (stdin); else yyterminate ();  }
["][a-zA-Z][a-zA-Z0-9]*["] {
                            int dimString= strlen(yytext)+1;
                            char subbuff[dimString-2];
                            memmove( subbuff, &yytext[1], dimString-3 );
                            subbuff[dimString-3] = '\0';
                            yylval.s = search(subbuff, NULL);
                            return STRING;
                        }
```

array

```
newArray integer pippo(2)

pippo->add = 2
pippo->add = 3
pippo->get


pippo->remove
pippo->get

pippo->set=1,0
pippo->get

print pippo->get=0

newArray char pluto(2)

pluto->add="ciao"
connect pluto->get = 0
```

Parser

```
exec:  /* nothing */
    | exec stmt EOL { run ($2); }
    .............
stmt: ........
    | exp
```

```
exp:
   .....
   FUNC exp   { $$ = newfunc($1, $2, NULL, NULL);    }
   ....
```

```
callBuiltIn (f){

......
case  B_readFile:
        x=eval(f->l);
        v= x ? strdup(x):strdup("0");
           //v=strdup(eval(f->l));

        let=fopen(v, "r"); //esistenza file

        if (let!=NULL){
              yyin=let;
        }else{
              printf("file inesistente\n");
        }

.....

}
```

Basso Livello ->

Eval

CallbuiltIn (F)

Funzioni Embedded	
connect	F.Embedded
readFile	F.Embedded
insert	F.Embedded
device	Tipo
bye	abort
interval	F.Embedded
switchOn	F.Embedded
switchOff	F.Embedded
delete	F.Embedded
reconnect	F.Embedded
arrow	Indicatore ->
status	F.Embedded
newDevice	F.Embedded
print	F.Embedded

CallbuiltArray (M)

Array	
newArray	F.Array
integer	Identificatore
char	Identificatore
integer	Identificatore
device	Identificatore
set	F.Embedded
get	F.Embedded
add	F.Embedded
remove	F.Embedded

CallbuiltInSystem(O)

System	
help	F.System
clear	F.System
sleep	F.Embedded

ReadFile array

```
> readFile "array"

> ARRAY CREATO CORRETTAMENTE:pippo#5694#403

>
>
> 2
3

>
> 2

>
> 1

> Display: 1

> ARRAY CREATO CORRETTAMENTE:pluto#3842#2132

>
> Ricerca del dispositivo ciao in corso...
Dispositivo ciao#3924: Non Esistente
```

dino copia — vi array — 108×32

```
newArray integer pippo(2)

pippo->add = 2
pippo->add = 3
pippo->get


pippo->remove
pippo->get

pippo->set=1,0
pippo->get

print pippo->get=0

newArray char pluto(2)

pluto->add="ciao"
connect pluto->get = 0
~
~
```

Operazioni ARRAY

Creazione:

```
> a->add = "ciao"
> a->add = "arco"
```

Grammatica:

```
case B_add:
     add_array(f);
     return NULL;
     break;

case B_get:
     if ((f->l)==NULL){
        get_array (f);
        return NULL;
     }else{
        return (struct ast *) ( get_index (f) ) ;
     }
     break;

case B_set:
      set_array (f);
      return NULL;
      break;

case B_remove:
      remove_array (f);
      return NULL;
      break;
```

```
exec:  /* nothing */
    | exec stmt EOL          -> scorro: radice: eval($2)
;

stmt: exp ...
;

exp: liste ...
;

 liste: | NAME ARROW  ADD '=' type                  ......
        | NAME ARROW  GET                           ......
        | NAME ARROW  GET '=' NUMBER                ......
        | NAME ARROW  SET '=' type ',' NUMBER  -> $$=newfunc($3, (struct ast *)$5, (struct ast *)$1, (struct ast (newnum($7)))
        | NAME ARROW  REMOVE                        ......

type:
          INSERT STRING                                  .......
        | NUMBER                                         { $$ = newnum($1);                              }
        | STRING                                         { $$ = newString((struct symbol *)$1);          }
        | INSERT STRING ARROW '[' argsListDevice ']'     ......
        | DATA                                           { $$ = newString((struct symbol *)$1);}
;
```

Eval (newFuncArr case F)

```
void add_array(struct funcBuiltIn *f){
    1. ricerca nella tabella dei simboli
    2. eval (newNum, newString, newDevice)
    3. In base al tipo alloca una nuova area di memoria per inserire il nuovo elemeneto dell'array
}
```

Eval cond (newString case C)

Ritorna la stringa inserita dall'utente

Se chiami get_index ritorna il valore

TIPI COMPOSITI ARRAY

Creazione:

```
> newArray char a(2)
ARRAY CREATO CORRETTAMENTE
```

Token:

```
"newArray"              { return ARRAY;                                  }
"integer"               { return INTEGER;                                }
"char"                  { return CHAR;                                   }
"device"                { return DEVICE;                                 }

[a-zA-Z][a-zA-Z0-9]*    { yylval.s = search(yytext, NULL); return NAME;  }
[0-9]+                  { yylval.d = atof(yytext); return NUMBER;        }
```

Grammatica:

```
exec:  /* nothing */
    | exec stmt EOL
;

stmt: exp ...
;

exp: liste ...
;

liste: ARRAY nameType NAME '(' NUMBER ')' {$$=newAstArray($3, $5, $2);}

nameType: CHAR {$$=2;} | INTEGER {$$=1;} | DEVICE {$$=3;}
;
```

```
void newArray (struct symbol *nome, double dimensione, double tipo){
    char *v=nome->name ;
    v=symhashDev(name_dev(v));
    v=symhashDev(v);

    if (tipo==1){
        int *d=malloc(sizeof(int));
    }
    else if (tipo==2){
        struct symbol *d=malloc(sizeof(struct symbol));
    }else{
        struct device *d=malloc(sizeof(struct device));
    }
    update_lookup(v, (struct ast  *)d, tipo, dimensione);
}
```

1- Informazioni che andrai a memorizzare nella lookup

Conclusioni e Punti di forza:

- Semplicità: DB e Library
  - Avvio del terminale: testing
- Funzioni legate al contesto
- Riuso di funzioni

Di seguito vengono riportate le funzioni predefinite nella libreria caricate in fase di avvio:

```
1   print "LoadingLibrary"
2   CMD routine (v) = print "routineAvviata"; repeat 2 do print "stepAvviato";
3   innaffia(v); sleep 30; print "stepConcluso";; print "routineConclusa";
4   CMD innaffia (v)= if connect v then switchOn v; sleep 5; switchOff v;;
5   CMD attivaDevice(dev,data,temp) = if connect dev then interval dev-temp-data;;
6   print "EndLibraryLoading"
```

In fase di avvio viene anche caricato il DataBase (file locale) definito e modificabile dall'utente.

```
print "LoadingDB"
newDevice "valvolaSud"
newDevice "valvolaEst"
newDevice "valvolaOvest"
newDevice "pompa"
print "EndDBLoading"
```

Istruzione di prova:

```
>Display: LoadingDB

> Dispositivo inserito con successo con ID: valvolaSud#7349
Operazione di inserimento dispositivo completata con successo

> Dispositivo inserito con successo con ID: valvolaEst#8517
Operazione di inserimento dispositivo completata con successo

> Dispositivo inserito con successo con ID: valvolaOvest#7276
Operazione di inserimento dispositivo completata con successo

> Dispositivo inserito con successo con ID: pompa#3755
Operazione di inserimento dispositivo completata con successo

6: Errore: syntax error
> Display: LoadingLibrary

> funzione definita
> funzione definita
> funzione definita
> Display: EndLibraryLoading

> innaffia ("valvolaSud")
Ricerca del dispositivo valvolaSud in corso..
Dispositivo Esistente
```

db

```
print "LoadingDB"
newDevice "valvolaSud"
newDevice "valvolaEst"
newDevice "valvolaOvest"
newDevice "pompa"
print "EndDBLoading"
```

library

```
print "LoadingLibrary"
CMD routine (v) = print "routineAvviata"; repeat 2 do print "stepAvviato"; innaffia(v);
sleep 30; print "stepConcluso";; print "routineConclusa";
CMD innaffia (v)= if connect v then switchOn v; sleep 5; switchOff v;;
print "EndLibraryLoading"
```

Stmt: istruzioni

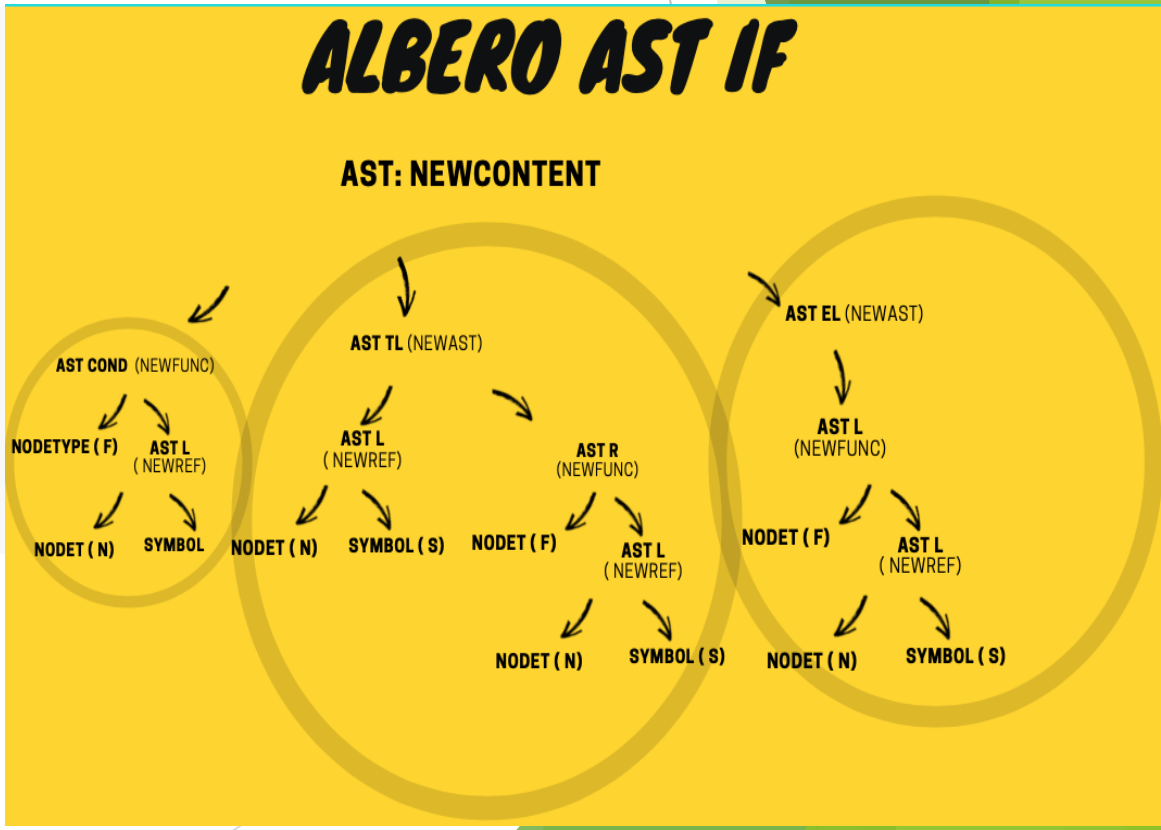
If exp then listStmt

While exp do listStmt

Repeat NUM do listStmt

Function

```
stmt: IF exp THEN listStmt                -> nodeIF    =  newContent('I', $2, $4, NULL);
    | IF exp THEN listStmt ELSE listStmt  -> nodeElse = newContent('I', $2, $4, $6);
    | WHILE exp DO listStmt               -> nodeCorpo =  newContent('W', $4, $2, NULL);
    | REPEAT NUMBER DO listStmt           -> nodeCorpo =  newRepeat('R', $2, $4);
    | exp
;

listStmt:  /* nothing */
        | stmt ';' listStmt     -> nodeCorpo ($3 == NULL ? $$=$1 : newast('L', $1, $3));
;

exp:.....
   | FUNCDEV exp                        ->  nodeCorpo = newfunc($1, $2, NULL, NULL);
   | NAME                               ->  nodeCorpo = newref($1);
   | NAME '=' exp                       ->  nodeCorpo = newasgn($1, $3);
   .....
;
```



Definizione di funzione

Creazione:

```
> CMD nomeFunzione (v1,v2) = h=9; print h; print v1; print v2;
funzione definita
```

Token:

```
- "CMD" return CMD;
- nome Funzione: [a-zA-Z][a-zA-Z0-9]*    { yylval.s = search(yytext, NULL); return NAME;   }
- Parametri: Una qualsiasi espressione (connect, name, intero, stringa e data)
- Corpo: Un qualisasi listStm, lista di espressioni
```

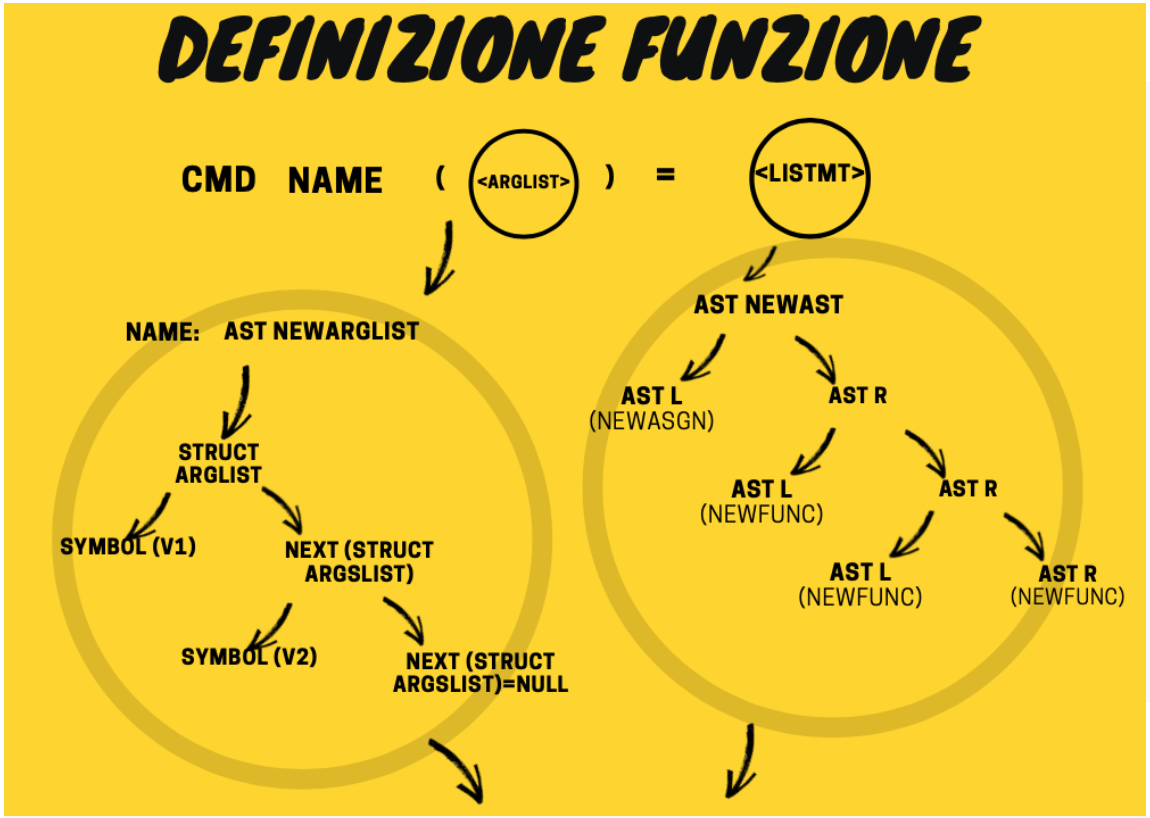
Grammatica:

```
exec: exec function EOL  ->  eval($2);  treefree($2);  printf("\n> ");

;

function:   CMD NAME '(' argsList ')' '=' listStmt RET exp ';'   -> $$=defFunction ($2, $4, $7, $9);
          | CMD NAME '(' argsList ')' '=' listStmt               -> $$=defFunction ($2, $4, $7, NULL)
;

listStmt:  /* nothing */
        | stmt ';' listStmt     -> nodeCorpo ($3 == NULL ? $$=$1 : newast('L', $1, $3));
;

argsList: NAME                  -> nodeParams: $$ = newargsList($1, NULL);
        | NAME ',' argsList     -> nodeParams: $$ = newargsList($1, $3);
;
```

```
struct argsList *newargsList(struct symbol *sym, struct argsList *next)
{
  struct argsList *sl = malloc(sizeof(struct argsList));

  if(!sl) {
    yyerror("Spazio di memoria insufficiente\n");
    exit(0);
  }

  sl->sym = sym;
  (sl->sym)->name=sym->name;
  sl->next = next;
  return sl;
}
```

Di fatto non viene generato alcun nodo ast (è una foglia), ma viene solo aggiornata la lookup table

Memorizzazione parametri e corpo nella lookup




```
struct ast *newast(int nodetype, struct ast *l, struct ast *r)
{
  struct ast *a = malloc(sizeof(struct ast));
  a->nodetype = nodetype;
  a->l = l;
  a->r = r;
}

struct argsList *newargsList(struct symbol *sym, struct argsList *next)
{
  struct argsList *sl = malloc(sizeof(struct argsList));
  sl->sym = sym;
  (sl->sym)->name=sym->name;
  sl->next = next;

}
```

```
struct ast* defFunction (struct symbol *name, struct argsList *syms, struct ast *func, struct ast*ret)
{
  if(name->syms){ argsListfree(name->syms); }
  if(name->func) treefree(name->func);
  name->syms = syms;
  name->func = func;
  name->ret=ret;
}
```

Tabella Simboli

Name	Value	Function	Device	List	Dimensone	Syms	Valore di ritorno
NomeFunzione		Ast (newAst)				Ast (newArgsList)	newExp

Esecuzione funzione

Esecuzione:

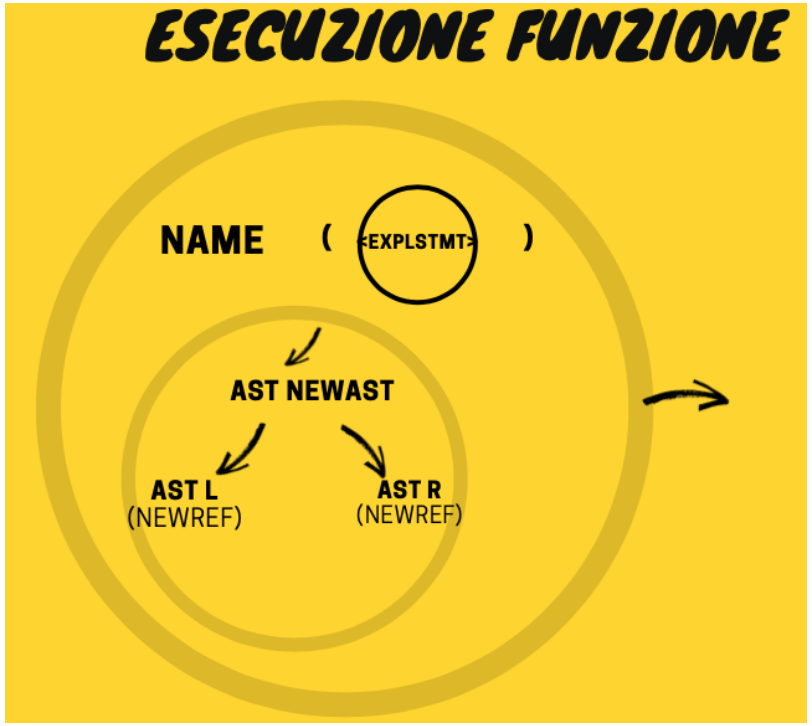
```
> nomeFunzione (9,10)
```

Token:

Grammatica:

```
exec: exec stmt EOL
stmt: exp
exp:  NAME '(' explistStmt ')'
explistStmt: exp
           | exp ',' explistStmt
;
```

```
struct ast *newcall(struct symbol *s, struct ast *l)
{
  struct userfunc *a = malloc(sizeof(struct userfunc));
  a->l = l; //argomento funzione
  a->s = s; //nome funzione, finisce nella tabella dei simboli
}
```




```
static char * calluser(struct userfunc *f)

  1. Accesso nella tabella dei simboli al nome della funzione: fn

  2. Verifica dal campo fun dell'esistenza della funzione
  if(!fn->func) {
    yyerror("Chiamata ad una funzione insesistente", fn->name);
    return 0;
  }

  3. args: memorizzi la lista dei nodi ast passati alla funzione e scorri
  l'albero
  for(int j = 0; j< nargs; j++) {

    if(args->nodetype == 'L') {        /* se è una lista di nodi */
      PARAMTRI= eval(args->l);
      args = args->r;
    } else {
      PARAMETRI= eval(args);
      args = NULL;
    }

  }

  4. SALVARE IN UNA VARIABILE TEMPORANEA I VECCHI VALORI DEI PARAMETRI
  for(i = 0; i < nargs; i++) {
     struct symbol *s = sl->sym;

     oldval[i]=(char *)malloc(strlen(n) * sizeof(char *));
     sprintf(PARAMETRI_OLD, "%s", n);
     sprintf( (s->value), "%s", PARAMETRI);

     sl = sl->next;
  }

 5. Scorri i nodi dell'albero memorizzati in func nella tabella dei simboli
 eval(fn->func);

 6. Rimemorizzi i vecchi valori dei parametri
 sl = fn->syms;
 for(i = 0; i < nargs; i++) {
    struct symbol *s = sl->sym;
    sl = sl->next;
    sprintf(s->value, "%s", PARAMETRI_OLD);
 }
```

```
>  readFile "if"

>
14: Errore: syntax error
>
> Ricerca del dispositivo dev in corso...
Dispositivo dev#8863: Non Esistente
Ricerca del dispositivo dev in corso...
Dispositivo dev#8863: Non Esistente
Ricerca del dispositivo dev in corso...
Dispositivo dev#8863: Non Esistente
Ricerca del dispositivo dev in corso...
Dispositivo dev#8863: Non Esistente
Ricerca del dispositivo dev in corso...
Dispositivo dev#8863: Non Esistente

> Dispositivo inserito con successo con ID: dev#8863
Operazione di inserimento dispositivo completata con successo

> Ricerca del dispositivo dev in corso...
Dispositivo Esistente
Richiesta connessione...
Display: CONNESSO

>
> Ricerca del dispositivo dev in corso...
Dispositivo dev#8863: Non Esistente
Display: PROBLEMI

> Dispositivo inserito con successo con ID: dev#8863
19: Errore: syntax error

> Ricerca del dispositivo dev in corso...
Dispositivo Esistente
Richiesta connessione...
Display: CONNESSO
inizia switchOn
                        Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo dev in corso...
Dispositivo Esistente
Richiesta connessione...

                        La connect è andata a buon fine e il dispositivo è stato acceso

Ricerca del dispositivo dev in corso...
Dispositivo dev#8863: Non Esistente
```

dino copia — vi if — 108×32

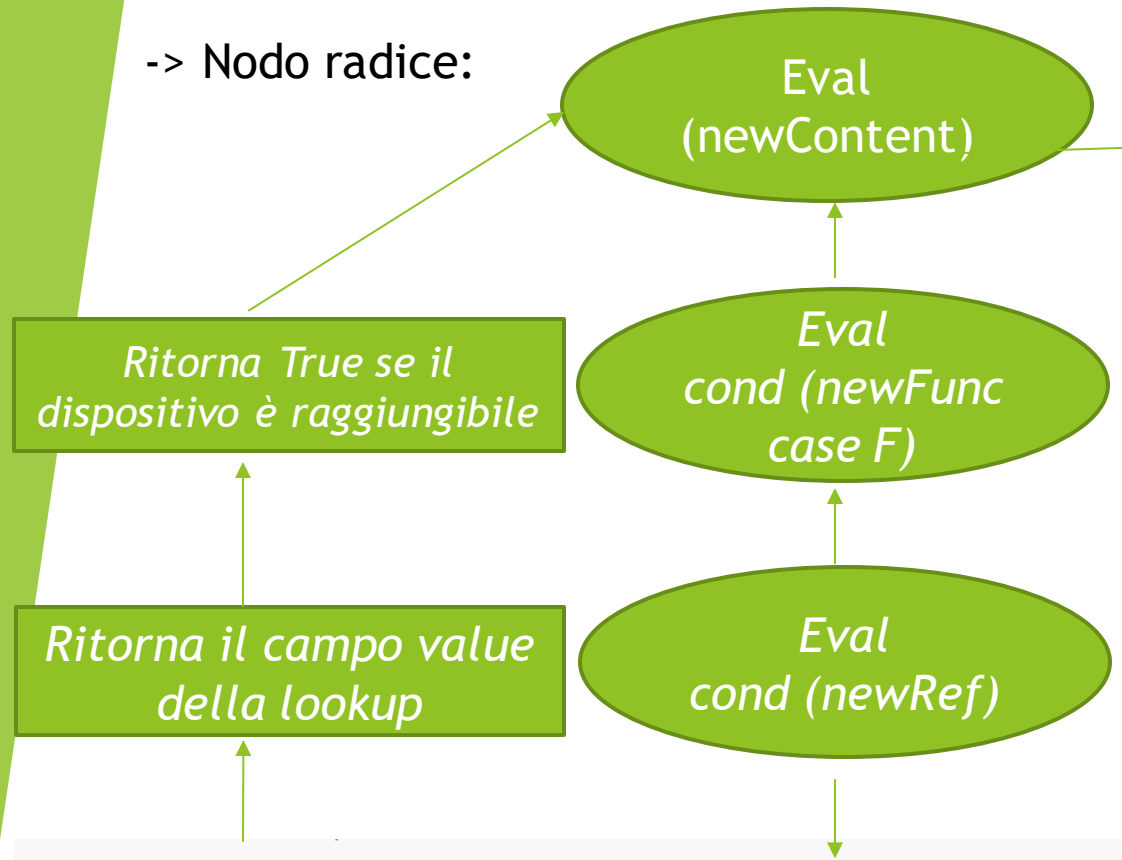
```
var="dev"

repeat 5 do connect var;

newDevice "dev"
if connect var then print "CONNESSO";

delete var
if connect var then print "CONNESSO"; else print "PROBLEMI";

newDevice "dev";
repeat 2 do if connect var then print "CONNESSO"; switchOn var; delete var;;

~
~
~
~
```

CASE I,W: ITERAZIONE

Scorrimento dalla foglia

- Una volta riconosciuta la produzione IF verrà generato un nodo AST newContent e partirà lo scorrimento dell'albero a partire dalla produzione dello scopo exec stmt EOL:

-> Nodo radice:




```
case 'I':
      v=malloc(sizeof(char));
    if( eval( ((struct content *)a)->cond) !=NULL){
       // printf ("a->condASAS\n");
        if( ((struct content *)a)->tl) {
             v = eval( ((struct content *)a)->tl);
       } else{
            v[0] = '0';         /* a default value */
        }
    } else {
        if( ((struct content *)a)->el) {
            v = eval(((struct content *)a)->el);
        } else{
            v[0] = '0';         /* a default value */
        }
     }
    break;
```

```
    /* Symref, token NAMe, ritorna il valore contenuto nel symbol name creato*/
case 'N':
        if((((struct symref *)a)->s->value))
            v=strdup((((struct symref *)a)->s->value));
```

```
case 'F':  ..callbuiltIn.......
             x=eval(f->l);
           ...functionConnect..
```

Scorrimento dalla foglia

- Una volta riconosciuta la produzione IF verrà generato un nodo AST newContent e partirà lo scorrimento dell'albero a partire dalla produzione dello scopo exec stmt EOL:

-> Nodo radice: