



UNIVERSITÀ DEGLI STUDI DI PALERMO

DIPARTIMENTO DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

# SERRA

Un linguaggio per l'Ambient Intelligence  
Tesina per “Linguaggi e Traduttori”

**Team:**

Thermokípío

**Docenti:**

Prof. Ing. Antonio Chella  
Ing. Francesco Lanza

## Sommario

Descrizione del Team.....	3
1. Introduzione .....	3
2. Stato dell'arte .....	4
3. Descrizione del progetto .....	6
3.1 Analisi dei requisiti .....	6
4. Caratteristiche del Linguaggio.....	8
4.1 Grammatica.....	8
4.2 Analizzatore lessicale .....	8
4.3 Descrizione del Parser .....	13
Analizzatore sintattico.....	13
Parse Tree.....	17
5.Casi d'uso: .....	26
5.1 Tipi:.....	26
5.1.1 Tipi primitivi: .....	26
5.1.2 Tipi compositi: .....	26
Variabili .....	26
Variabile di sistema: .....	27
Tipi compositi:.....	28
1.     Liste.....	28
2.     Array .....	29
Interprete .....	36
Indentazione e formattazione.....	37
Parole chiavi del linguaggio.....	37
Funzioni embedded:.....	37
Funzioni annidate .....	49
Operatori condizionali.....	50
5.2 Comandi/funzioni definite dall'utente: .....	51
5.2.1 Risultati ottenuti e punti di forza.....	79
6 Esempio di codice di nel linguaggio SERRA .....	83
6.1     InnaffiaElse .....	83
6.2 InnaffiaElseReconnectDelete .....	84
6.3     Irrigazione .....	84
6.4     IrrigazioneArray .....	85
6.5     IrrigazioneFunzione .....	86
7. Conclusioni .....	87
7.1 Sviluppi futuri .....	87
Bibliografia .....	88

## Descrizione del Team

Il Team Thermokípío è composto dagli Allievi Ingegneri:

- Gaetano La Bua
- Mario Caruso
- Vincenzo Guglielmo La Mantia

iscritti al Corso di Laurea Magistrale in Ingegneria Informatica UNIPA.

## 1. Introduzione

Il nostro Team, avendo valutato la crescente richiesta del mercato di dispositivi non invasivi in grado di affiancare i lavoratori durante attività più o meno complesse, ha deciso, come ambito di sviluppo per il progetto, di scegliere l'Ambient Intelligence (AMI).

Quando si parla di AMI ci si riferisce ad ambienti elettronici, con a bordo una sostanziale componente software, sensibili e reattivi alla presenza di persone, si ha quindi un ambiente, in cui attraverso una adeguata sensoristica e un opportuno sistema di gestione software, si interconnettono tra loro, dando così una “voce”, macchinari e strumenti di vario genere, permettendo di interagire con questi, direttamente e da remoto, riducendo il carico di lavoro e aumentando l'usabilità.

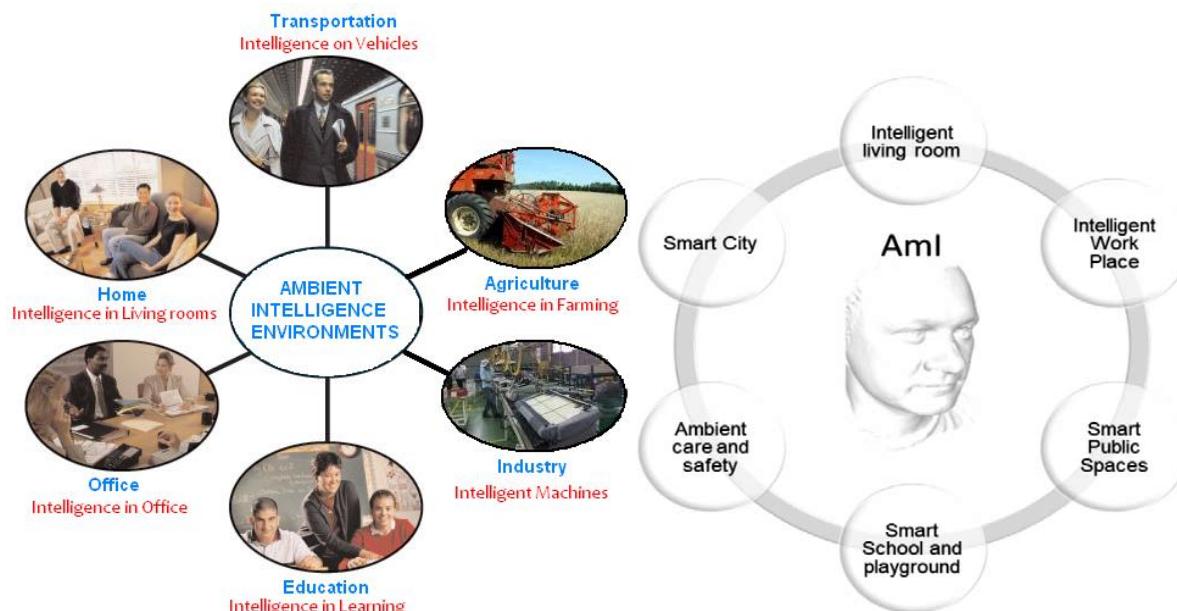


Figura 1 - <https://6ff66f56-a-62cb3a1a-sites.googlegroups.com>

Alla base dell'idea dell'AMI ci sono due concetti fondamentali:

- I dispositivi inseriti non devono essere invasivi
- L'utilizzo deve essere User-Friendly.

Il nostro progetto si è concentrato sul secondo aspetto, ossia sullo sviluppo di un linguaggio per lo sviluppo di un Ambient Intelligence, rendendo semplice la programmazione e facile l'utilizzo.

In particolare, si è deciso di sviluppare, un linguaggio per l'automazione di sistemi per la cura di giardini, orti e serre. A questo progetto abbiamo dato il nome “SERRA”.

## 2. Stato dell'arte

Nella seguente sezione ci occupiamo di analizzare altri linguaggi in ambito scientifico utilizzati per il sistema di irrigazione e di giardinaggio.

Il giardinaggio è la tecnica (manuale o digitale) che si occupa della coltivazione di piante, irrigazione di terreni. Rappresenta infatti quel complesso di attività che si occupa della manutenzione di un giardino pubblico o privato. Può avere uno scopo oltre che naturale anche ornamentale.

Una delle tecniche maggiormente utilizzate per l'irrigazione è l'irrigazione localizzata o a goccia dove, tramite apposito strumento, viene somministrata lentamente dell'acqua al terreno o alle piante. Gli strumenti che consentono tali meccanismi sono: valvole, pompe, condotte da cui viene fatta passare l'acqua e i gocciolatoi (da cui fuoriesce l'acqua). In generale tali strumenti possono essere controllati a distanza tramite appositi dispositivi. Ovviamente in base alle dimensioni del terreno e alle tipologie di piante che bisogna irrigare, i dispositivi e i sensori dovranno essere programmati in modo da causare uno "schizzo" più o meno forte dell'acqua che viene fatta passare dalle valvole e dai gocciolatoi.

Si è pensato che l'utilizzatore finale del linguaggio di programmazione da noi ideato, non sia un esperto di linguaggi di programmazione, ma sappia bene come e quanti dispositivi dovrà avere la sua serra. L'utilizzatore dovrà quindi sapere quanti dispositivi saranno necessari per il suo sistema di irrigazione. Per esempio, la dimensione di tubi, il tipo di gocciolatoi da usare, e altre cose che riguardano l'aspetto più legato a scelte di progettazione dell'impianto di annaffiamento della serra, e di disposizione dei vari dispositivi e sensori all'interno della stessa, non saranno, ovviamente, quindi gestibili dal programma, ma sarà compito dell'agricoltore predisporre il tutto. Poi, attraverso un programma scritto nel nostro linguaggio di programmazione, sarà possibile creare delle astrazioni degli oggetti "device" presenti nella realtà, permettendo così la creazione di programmi articolati in grado di connettere i dispositivi fisici della serra e di programmarne l'utilizzo. Quindi l'utente tramite il nostro linguaggio potrà:

- Creare un'istanza di un oggetto Device;
- Verificare se un device è attivo nel sistema di giardinaggio e in caso contrario richiederne l'attivazione;
- Rimuovere l'istanza del dispositivo;
- Cambiare lo stato del device per avviare l'operazione di innaffiamento;
- E altre operazioni per la gestione e la programmazione dei dispositivi della serra.

Nel progetto le operazioni di collegamento a un device o variazione di un device sono state esemplificate con stampe a video di messaggi, ma in generale sarà necessario creare un collegamento tra i device e il sistema in cui dovrà essere realizzato il programma.

Rispetto ad altri linguaggi come ML, un linguaggio usato per studiare altri linguaggi, non consente per esempio la definizione di nuovi tipi. Non può neanche essere considerato un linguaggio scientifico in quanto di fatto non consente alcuna operazione matematica (sono, infatti, state considerate superflue per l'ambito che stiamo trattando). Per tale motivo è molto distante da linguaggi come Cobol, nati a scopo finanziario. Rispetto ai comuni linguaggi C, fortran, etc consente di realizzare le basilari operazioni di:

- Assegnazioni variabili;
- Definizioni variabili;
- Realizzazione di array;
- Tipi stringhe e interi;
- Realizzazione di liste (di device);

- Definizione di nuovi comandi runtime.

Non esistono, neanche, i tipi puntatori per l'utente finale del linguaggio per cui l'operazione di dealloccamento della memoria dovrebbe essere prevista da un programma esterno (Garbage collector o comunque appena il programma terminerà verrà comunque liberata tutta l'area di memoria occupata dall'utente). Per cui è un linguaggio assimilabile all'ISP per il fatto che non consente operazioni molto complesse con i tipi primitivi o composti presenti nel linguaggio.

Il programma permette di essere usato da persone inesperte in ambito di programmazione per cui sono stati previsti molte funzioni embedded/comandi da utilizzare per effettuare le operazioni sopra elencate di connessione con device, attivazione del device, ecc. Per tale motivo può essere assimilabile a una sorta di File Batch

([https://it.wikipedia.org/wiki/File\\_batch](https://it.wikipedia.org/wiki/File_batch)) che contiene i comandi principali per tali operazioni. Con in più dei comandi per le operazioni di assegnamento e definizione di tipi e nuovi comandi. Il linguaggio prevede quindi un uso interattivo come una qualsiasi altro Bourne shell programming ([https://it.wikipedia.org/wiki/Bourne\\_shell](https://it.wikipedia.org/wiki/Bourne_shell)).

Nonostante abbia scopi differenti da quelli didattici può essere assimilabile a Cool ([https://it.wikipedia.org/wiki/Cool\\_\(linguaggio\)](https://it.wikipedia.org/wiki/Cool_(linguaggio))) presentando solo alcune caratteristiche basilari dei comuni linguaggi di programmazione, come C, Java ecc.

Sono presenti vari SW specifici che si occupano nello specifico di sistemi agrari. Infatti, l'azienda FarmBot negli ultimi anni ha avuto l'idea di:

“utilizzare robot autonomi nei campi è un'idea interessante che attira sempre più attenzione. La Sfida AgBot del 2016 è un esempio di un tentativo di sviluppare robot di semina senza equipaggio. Molti sponsor come Yamaha, AGCO e John Deere hanno un occhio sull'esito di sfide come questa. Considerando la capacità di allevamento di questi robot autonomi 24 ore su 24, 7 giorni su 7, possiamo ovviamente ottenere un notevole ritorno da queste macchine. Oltre alla semina autonoma, altre questioni agricole come la raccolta autonoma e la rimozione di erbe infestanti e parassiti sono anch'esse argomenti di interesse.”

( <https://it.electronics-council.com/farmbot-intends-revolutionize-home-gardening-31975> )

Per fare ciò stanno realizzato un SW di un sistema di giardinaggio chiamato FarmotBot Genesis. E' un programma basato sull'interfacciamento grafico (come quello che vedete sotto) in cui tramite le coordinate x,y,z si riesce a gestire un robot che si occupi della gestione del giardino.

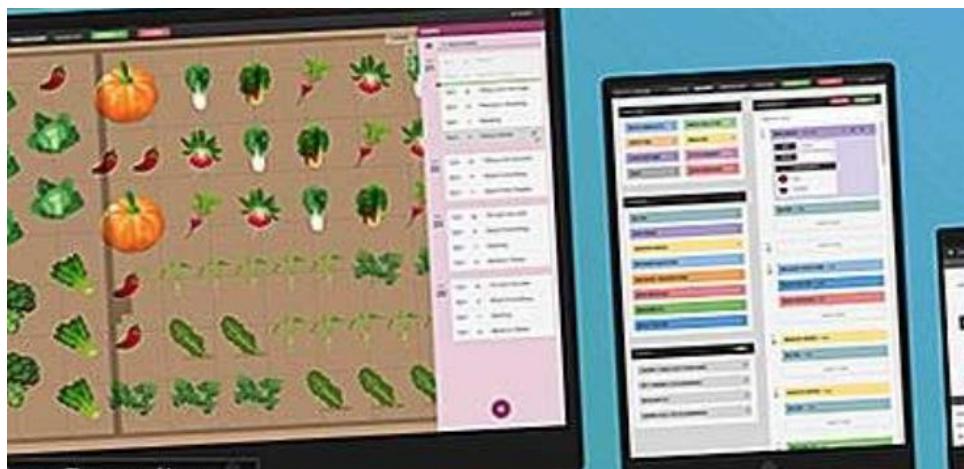


Figura 2 - <https://it.electronics-council.com/farmbot-intends-revolutionize-home-gardening-31975>

In particolare, nel nostro linguaggio, l'interfaccia è rappresentata dal terminale e con i vari device è possibile definire le varie zone che è necessario annaffiare e irrigare.

### 3. Descrizione del progetto

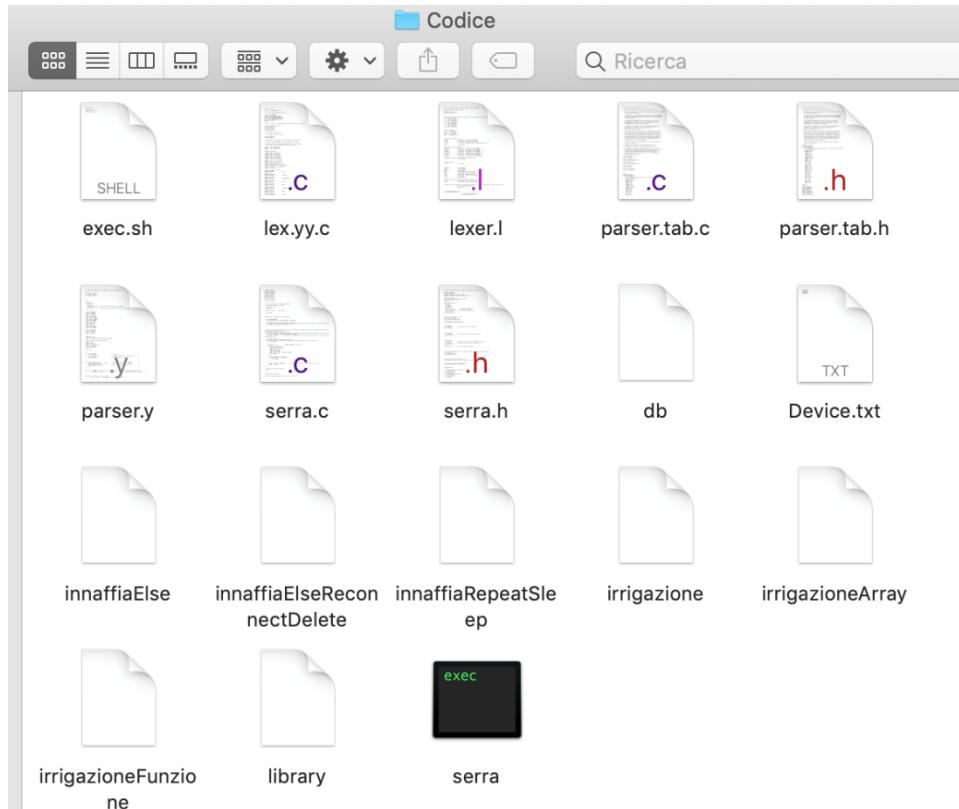
Il linguaggio “agrario” con lo scopo di rendere le operazioni manuali di gestione di un sistema di irrigazione il più rapide e semplici possibili. Esistono vari meccanismi di annaffiamento come accennato nelle sezioni precedenti. Quella che prendiamo di riferimento è l’irrigazione a gocce. Gestendo lo stato dei device che attivano la procedura di gocciolamento si può pensare di effettuare operazioni annaffio.

#### 3.1 Analisi dei requisiti

L’obiettivo nel nostro lavoro è quello di rendere il linguaggio il più semplice possibile all’utilizzatore finale; infatti, l’utente non si renderà conto della generazione di una struct device che contiene le varie informazioni del device ma semplicemente lui si occuperà di inserire stringhe ( o al più numeri ) tramite i quali potrà istanziare in memoria oggetti device. Il linguaggio, quindi, deve essere il più semplice possibile consentendo solo le più banali operazioni di un qualunque linguaggio. Allo scopo di rendere il programma semplice si dà la possibile all’utente di generare nuovi comandi runtime per semplificarsi possibili operazioni future. In particolare:

- A livello di giardinaggio le operazioni previste sono:
  - Creazione di una istanza oggetto device;
  - Variazione dello status del device e quindi accenderlo;
  - Variazione dello status del device per un intervallo di tempo;
  - Eliminazione di un dei device.
- A livello di linguaggio classico sono previsti:
  - Cicli iterativi;
  - Definizioni e assegnazioni variabili;
  - Comandi Runtime;
  - Istruzioni Condizionali;
  - Esecuzione di programma scritto su file;

Nella cartella inviata (cartella: codice) sono presenti i seguenti file:



- Lexer.l -> contiene il lexer
- Parser.y -> contiene il parser
- Lex.yy.c , parser.tab.h -> sono file generati eseguendo il lexer e bison
- Serra.c , Serra.h -> file C e header
- Tutti i file innaffia e irrigazione (saranno descritti in seguito) sono esempi di utilizzo del linguaggio
- Il file DB contiene l'elenco dei device (e sarà caricato all'avvio del linguaggio)
- Library->contiene funzioni di sistema di libreria (e sarà caricato all'avvio del linguaggio)
- Exec.sh -> file di comodo per eseguire tutti i file C, lexer e bison

Esecuzione del programma:

\$./exec.sh

\$./serra

```
(base) MBP-di-Vincenzo:Codice vincenzolabua$ ./exec.sh
serra.c:1106:24: warning: address of stack memory associated with local variable
  'chararray' returned [-Wreturn-stack-address]
      return chararray;
      ~~~~~~
```

1 warning generated.

(base) MBP-di-Vincenzo:Codice vincenzolabua\$ ./serra

```
+++++-----+ +---+ +----+
|W|e|1|c|o|m|e| |t|o| |t|h|e|
+++++-----+ +---+ +----+
ad88888ba 8888888888 888888888ba 888888888ba      db
d8'   '8b  88      '8b  88      '8b      d88b
Y8,     88      88      ,8P  88      ,8P      d8' `8b
`Y8aaaaa, 88aaaaa   88aaaaaa8P' 88aaaaaa8P'  d8'   `8b
`aaaayy8  88aaaaa   88''''88' 88''''88h    8Yaaaaaa8b
`8b  88      88      `8b  88      `8b      d8''''''8b
Y8a   a8P  88      88      `8b  88      `8b      d8'       `8b
YY88888PP 8888888888 88      `8b  88      `8b  d8'           `8b
+++++-----+ +---+ +----+
|p|r|o|g|r|a|m|m|i|n|g| |l|a|n|g|u|a|g|e|!|
+++++-----+ +---+ +----+
```

```
+++++-----+ +---+ +----+
|W|r|i|t|e| |h|e|l|p| |i|f| |y|o|u| |n|e|e|d| |i|t| |
+++++-----+ +---+ +----+
+++++-----+ +---+ +----+
```

## 4. Caratteristiche del Linguaggio

### 4.1 Grammatica

Per realizzare un analizzatore lessicale è stata utilizzata la libreria Flex. L'analizzatore lessicale si occupa di estrarre i token che definiranno la grammatica del nostro linguaggio. I token estratti verranno a loro volta presi in input da Bison che si occuperà di effettuare un'analisi sintattica del linguaggio occupandosi delle precedenze tra operatori/token.

La grammatica che è stata realizzata è, secondo la classificazione di Chomsky, di tipo 2 (libera dal contesto) dato che tutti i simboli non terminali (o produzioni) della nostra grammatica vengono tradotte in una sequenza di simboli/o terminali/e e/o non terminali/e:

$\langle A \rangle \rightarrow \alpha$  [ $\alpha$  non è vuota]

La caratteristica dei linguaggi di tipo 2 è che possono essere espressi sottoforma di un automa a stati finiti con uno stack di dimensione infinita (in seguito ne rappresenteremo una forma generale di tale diagramma).

### 4.2 Analizzatore lessicale

È stato realizzato utilizzando la libreria Flex. Tramite Flex vengono individuati i token e quindi la sequenza di caratteri digitati dall'utente in fase di esecuzione del programma.

Nella seconda sezione del lexer sono state individuate le seguenti definizioni regolari da usare con i corrispondenti token. I token individuati, che ritroviamo anche in bison, sono:

```

21  /* declare tokens */
22  %token <d> NUMBER
23  %token <str> STRING
24  %token <s> NAME
25  %token <func> FUNC
26  %token <func> SYSTEM
27  %token <func> FUNCDEV
28  %token <func> INTERVAL
29  %token <func> INSERT
30  %token <str> ARROW
31  %token <s> DATA
32  %token <func> ADD
33  %token <func> GET
34  %token <func> SET
35  %token <func> REMOVE
36  %token <func> SLEEP
37  |
38  %token <func> ARRAY
39  %token <d> INTEGER
40  %token <d> CHAR
41  %token <d> DEVICE
42
43  %token EOL
44  %token <str> TERM
45  %token IF THEN ELSE WHILE DO CMD HELP RET REPEAT

```

Figura 3 - token file bison

Descrivendoli nel dettaglio:

#### Definizioni regolari:

- Il token **NUMBER** è individuato dalla seguente definizione regolare:  
**NUMBER -> [0-9]+**
  - Ex:** 1 , 2, 3, 44

Tale token consente di inserire nel nostro lessico tutti i numeri interi positivi rappresentabili nel nostro sistema.

- Il token **EOL** è invece individuato dalla seguente espressione regolare:  
**EOL -> \n { return EOL; }**
  - Ex:** [invio]

Corrisponderà al terminatore di ogni istruzione. Un'istruzione nel programma termina nel momento in cui l'utente digita invio

- Il token **NAME** è usato invece per individuare un carattere seguito da una qualsiasi stringa di lettere e numeri (se presente). Consente di fatto di individuare le variabili memorizzate nel sistema. E' usato anche per individuare non solo i nomi di variabili

ma anche array. Verrà anche utilizzato nella definizione di funzioni vedremo.

**NAME -> [a-zA-Z][a-zA-Z0-9]\***

- **Ex:** ciao , DEVICE1, device2 , d2

- Il token **STRING** viene utilizzato per individuare le stringhe nel nostro programma nel formato “ciao”:

**STRING-> "["[a-zA-Z][a-zA-Z0-9]\*"]**

- **Ex:** “ciao “, “DEVICE1”, “device2” , “d2”

- Il token **DATA** viene utilizzato per individuare le date nel nostro programma. Per semplicità stiamo considerando solo il secolo corrente nell’analisi delle date:

**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, 2021.11.01.9, 2021.01.31.9, 2020.01.01.9
- **Errori sintattici:** 2021.01.01.90, 2021.25.01.9, 3021.01.01.9,  
2021.13.01.9

Di fatto questi sono i token principali delle definizioni regolari necessari nel nostro linguaggio. Tramite i token identificati infatti la nostra grammatica consente di individuare finora con 3 token diversi:

- Tutti i possibili numeri (token number),
- Tutte le possibili stringhe tra virgolette (token string),
- Tutte le possibili date,
- Tutte i possibili nomi di variabili e funzioni (token name);

Notare che la differenza tra le stringhe e le variabili stanno che le prime iniziano con le “”, le seconde invece no. Vuol dire che:

- “Ciao” verrà riconosciuto col token STRING
- Ciao verrà riconosciuto col token NAME

Nel nostro sistema di giardinaggio si prevede infatti che tutte le funzioni embedded, tutte le funzioni Runtime e tutte le operazioni necessarie pretendano il riconoscimento solo di questi tre tipi di token.

I token successivi consentiranno di individuare **parole chiavi del linguaggio**, il quale significato è evidente:

- I token **IF, THEN, ELSE** sono individuati dalle espressioni regolari “if”, “then”, “else”. Ovviamente consentono di individuare i caratteri che consentiranno con l’analisi sintattica di effettuare le operazioni condizionali
- I token **DO, WHILE** sono individuati analogamente dalle espressioni regolari “do”, “while”.
- I token **REPEAT** individuano un ciclo iterativo da ripetere un certo numero di volte fissato come parametro

Altri token che individuano parole chiavi sono state inseriti a individuare funzioni di sistema che sono:

- **SYSTEM**
  - clear individuata dall’espressione “clear”;
  - help individuata dall’espressione “help”;
- **FUNC**
  - readFile individuata dall’espressione “readFile”.

Gli altri token **ADD**, **REMOVE**, **GET**, **SET** (che individuano rispettivamente le espressioni: add, get, set e remove) sono le operazioni base utilizzati per la gestione di liste e array come tipi compositi del linguaggio. In seguito, verrà fatta un'analisi approfondita di tali token.

- L'array avrà un tipo definito per cui sono presenti i token:
  - **CHAR** che identifica l'espressione "char"
  - **INTEGER** che identifica l'espressione "integer"
  - **DEVICE** che identifica l'espressione "device"
- Il token **ARRAY** che individua l'espressione "newArray" necessaria per la creazione dell'array
- Il token add verrà riconosciuto nel momento in cui l'utente digita add
- Il token remove analogamente verrà riconosciuto con l'espressione remove
- Set e get analogamente con le espressioni set e get

Gli ultimi token che descriviamo sono quelli che individuano parole chiavi necessari a definire funzioni embedded:

- **FUNCDEV** individua tutte le funzioni embedded necessarie a collegare i device per avviare le operazioni di giardinaggio. Tali funzioni sono individuate dalle espressioni: "connect", "reconnect", "switchOn", "switchOff", "status" e "archive"
- **INTERVAL** individua l'espressione "interval" e anche questa è una funzione di tipo **FUNCDEV** (è stato preferito però inserirla con un token differente perché al contrario delle altre funzioni, come vedremo nell'analisi sintattica prevede tre parametri).
- **FUNC** individua invece tutte le funzioni di sistema generali non strettamente legata ai device, per esempio la print. Il token verrà riconosciuto quando l'utente scriverà: print o clear.

Tutti i token finora descritti sono quelli utilizzati per effettuare l'analisi lessicale del nostro linguaggio. In funzione di questi token l'analizzatore sintattico si occuperà di valutare le sentence indicate dall'utente. Quindi coi nostri token in generale individuiamo:

- Funzioni di sistema (clear, help e readFile),
- Funzioni di device (connect, reconnect, status, interval ecc),
- Operazioni sui tipi compositi (add, get, remove e set),
- Costanti (stringhe e numeri),
- Variabili e funzioni runtime (individuate da una lettera seguite da una sequenza di lettere e numeri).

I restanti digit, ovviamente, non verranno riconosciuti dall'utente e verranno segnalati come caratteri sconosciuti e quindi come errore lessicale.

Nella seguente tabella riassumiamo quanto spiegato in precedenza:

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

### 4.3 Descrizione del Parser

#### Analizzatore sintattico

È stato realizzato utilizzando la libreria Bison.

Lo **scopo** nel nostro parser è individuato dalla produzione **exec**. Derivando più volte tale produzione riusciamo ad ottenere l'analisi sintattica del nostro linguaggio. Per effettuare tale analisi scegliamo di effettuare un'analisi discendente del nostro albero sintattico.

Le principali produzioni che definiscono la nostra grammatica sono le seguenti:

- Exec: rappresenta lo scopo. Ogni Statement verrà eseguita tramite tale produzione. Da essa infatti avviene lo scorrimento dell'albero sintattico.
- Statement: È un simbolo non terminale che consente l'individuazione della nostra istruzione. Tale produzione si occuperà infatti di riconoscere opportunamente il comando definito dall'utente. Riconosce quindi:
  - If, then, else;
  - Do, while;
  - Repeat;
  - Espressioni.
- Exp: È un simbolo non terminale che consente l'individuazione delle:
  - Variabili, funzioni di sistema/embedded e funzioni runtime.
- Type: In type sono stati inseriti tutti i tipi considerati primitivi nel linguaggio, o comunque derivanti da una combinazione di stringhe e numeri. Di conseguenza un type è:
  - Una stringa, un numero, una data, una sequenza particolare di parole chiavi per creare un device

Per comprendere meglio, analizziamo l'albero sintattico delle diverse espressioni regolari:

#### **EXP:**

È un simbolo non terminale per individuare tutte le espressioni presenti nel linguaggio. Un exp/espressione può essere:

- Un simbolo non terminale type. Type è un simbolo non terminale del linguaggio che identifica i simboli terminali rappresentati dai tipi primitivi e/o composti del linguaggio. Type può quindi essere:
  - Un numero (simbolo terminale): vuol dire che l'utente ha digitato un numero riconosciuto dal token NUMBER
 

**Exp -> Type; Type-> NUMBER**

    - 2 3 5
  - Una stringa (simbolo terminale): vuol dire che l'utente ha digitato una stringa riconosciuto dal token STRING
 

**Exp-> Type; Type-> STRING**

    - “ciao” “pippo”
  - Definizione di un tipo device: per la definizione di un device sono state usate due espressioni:
    - La prima prevede di accostare due simboli terminali rappresentati da INSERT (individuato dalla parola chiave “newDevice”) e STRING: tale istruzione sarà quindi del tipo:
 

**EXP-> Type; Type-> INSERT STRING**

      - NewDevice “pippo”

- La seconda consente non solo di creare un device ma anche di indicare che è collegato ad altri device:

**EXP-> Type; Type->INSERT STRING ARROW '[' argsListDevice ']'**

- NewDevice “pippo” -> [“ciao”]

Quindi un'espressione è un type che può essere o una stringa o un intero o un device, che sono i tipi base previsti nel linguaggio.

- Una variabile (simbolo terminale): vuol dire che l'utente ha digitato un nome di variabile riconosciuto dal token NAME

**Exp->NAME**

- Ciao pippo

- Una funzione embedded (simbolo terminale) e un ulteriore espressione (simbolo non terminale): la funzione embedded è individuata dal token FUNCDEV che è riconosciuto nel momento in cui l'utente avrà richiamato una delle funzioni di sistema riportate nella figura precedente (connect, reconnect, interval ecc). Tale funzione pretenderà dei parametri che potranno essere rappresentati da un ulteriore espressione. L'espressione exp (che stiamo descrivendo in tale sezione) può essere: una stringa, un numero, ecc. Per cui le funzioni embedded (connect, etc) verranno richiamate prendendo come parametri o i token STRING, NAME, NUMBER, etc, oppure magari contenendo altre funzioni come parametri al loro interno. In quest'ultimo caso verrà eseguita la funzione più interna per poi valutare la funzione più esterna. Sarà compito dell'albero controllare l'eventuale errore di conseguenza nel passaggio dei parametri. E' stata prevista in quest'ultimo caso la possibilità di annidare più funzioni embedded (vedremo in seguito):

**Exp->FUNCDEV exp**

- Esempio in cui passo come parametro un simbolo terminale:

- Connect “ciao”
- Connect “pippo”
- Connect “pluto”
- SwitchOn “nome”
- SwitchOff 1
- Connect 2
- SwitchOn connect “pippo”

- Esempio in cui passo come parametro un simbolo non terminale:

- Connect switchOff “pluto”: passo come parametro una funzione embedded.
- Connect var: passo come parametro una variabile.
- SwitchOn connect “pippo”: esempio di annidamento in cui passo come parametro di connect la stringa “pippo”, e come parametro di switchOn il valore restituito dall'espressione connect “pippo”.

Questo ultimo aspetto è possibile perché nella nostra grammatica abbiamo considerato come un'espressione anche i parametri da passare alla funzione, e non come un simbolo terminale.

- Una funzione di sistema (simbolo terminale) seguita da un'espressione: è analogo alla funzione embedded spiegata in precedenza. Ancora una volta, i parametri di questa funzione possono essere una qualsiasi espressione.

**Exp-> FUNC exp**

- Una variabile (simbolo terminale) seguito da un uguale (simbolo terminale) seguito da un'espressione (simbolo non terminale): una variabile (che è semplicemente una stringa) è individuata infatti dal token descritto nella sezione dell'analisi lessicale. E' seguita dal simbolo terminale uguale seguito da un'espressione. Ancora una volta,

(come nel caso delle funzioni embedded) tale espressione può rappresentare: un numero, una stringa, una chiamata a funzione embedded, ecc.

### Exp-> NAME '=' exp

- Esempio in cui passo come parametro un simbolo non terminale:
  - NomeVariabile = connect "pippo"
  - NomeVaribaile = switchOn "pluto"
  - NomeVariabile = NomeVariabile = 2
- Esempio in cui passo come parametro un simbolo terminale:
  - NomeVariabile = "pippo"
  - NomeVariabile = "pluto"
  - NomeVariabile = 1
- Interval: analogo alle funzioni embedded
- Una espressione può infine anche essere una lista. La lista è un ulteriore simbolo non terminale che consente di individuare tutte le operazioni possibili con un linguaggio. Una lista infatti può essere:
  - Creazione array: per creare l'array usa i token ARRAY NAME NUMBER e il simbolo non terminale nameType:
    - Il simbolo non terminale nameType identifica i simboli terminali per effettuare la tipizzazione dell'array. Infatti, può essere rappresentato dai token:
      - Char
      - Integer
      - Device
    - Esempio: newArray integer nomeArray (2);
      - NewArray integer sono identificati dai token ARRAY e INTEGER (descritti in precedenza). NomeArray è invece individuato dal token nomeArray
  - Add: Un NAME seguito da ARROW seguito da ADD seguito da = seguito dal simbolo non terminale type (descritto in precedenza, identifica: stringhe, interi e device):
    - Esempio: nomeArray -> add = 2
      - NomeArray è individuato dal token NAME, -> token arrow, add dal token ADD, invece 2 dal simbolo non terminale type (che abbiamo potere essere individuata dai token: string, number e newDevice)
  - Set:
    - Esempio: nomeArray -> set = 1,2
      - I token sono gli stessi di add solo che l'ultimo parametro (2) individua il simbolo terminale NUMBER
  - Remove: Analoga a add ma non pretende come parametro un'espressione type:
    - Esempio: nomeArray->remove
      - Individuata dai token NAME, REMOVE E ARROW

### EXEC (esecuzione del comando):

Una volta individuata l'espressione exp e l'istruzione stmt si potrà risalire quindi allo scopo exec da cui si inizierà a scorrere l'albero per poter eseguire l'istruzione individuata:

EXEC -> exec stmt EOL

L'exec (esecuzione del comando) è dunque un simbolo non terminale dato da una sequenza di un comando eseguito in precedenza (exec), da una nuova istruzione (stmt) e dall'inizio

(EOL). Di conseguenza il ritorno a capo cioè il carattere EOL individua la fine di ogni comando. In generale, quindi, un comando sarà seguito a partire da questa espressione. Si specifica, però, che l'istruzione da eseguire a partire dalla quale scorrere l'albero è contenuto nella statement.

Di seguito riportiamo una forma semplificata della grammatica utilizzata nel progetto per riassumere i concetti di parsing finora riportati.

```

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
;

argsListDevice: ->      STRING
                STRING ',' argsListDevice
;

```

Il concetto fondamentale è che nella nostra grammatica le espressioni possono annidarsi a vicenda; questo consente a ogni funzione embedded, assegnazione di variabile di prendere come parametro un qualsiasi oggetto (dalla stringa, all'intero, alla funzione richiamata).

### Parse Tree

L'albero sintattico è un albero che rappresenta la struttura sintattica in accordo con la grammatica del linguaggio.

Grazie a una feature di Bison, attraverso le istruzioni:

```

README.md innaffiaElse           irrigazione     lex.yy.c  output.png  parser.tab.h serra-Funzionante
db      innaffiaElseReconnectDelete irrigazioneArray lexer.l  parser.dot  parser.y    serra.c
exec.sh innaffiaRepeatSleep      irrigazioneFunzione library  parser.tab.c serra      serra.h
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

```

è possibile creare la rappresentazione dello stesso.

Nel grafo troviamo:

- Le regole di ogni stato sono raggruppate in nodi del grafico
- Le transazioni sono rappresentate da linee dirette tra lo stato corrente e quello target
- Gli **shift** sono rappresentati con le frecce solide etichettate con il lookahead token
- Le **riduzioni** sono mostrate come frecce piene, che portano a un nodo a forma di diamante verde che riporta il numero della regola di riduzione.
  - Se sono presenti le regole
    - 3 a: "0" . [";"]
    - 4 b: "0" . ["."]
  - Allora:
    - "." reduce using rule 4 (b)
    - \$default reduce using rule 3 (a)
- La riduzione corrispondente alla regola numero 0 è lo stato di accettazione. Viene visualizzato come un diamante blu, etichettato "Acc"
- Quando sono presenti conflitti irrisolti, poiché nell'analisi deve prendere una unica decisione, Bison sceglie arbitrariamente di disabilitare la riduzione (Shift / Reduce Conflicts) . Le azioni scartate sono contraddistinte da nodi rappresentati con rombi di colore rosso.



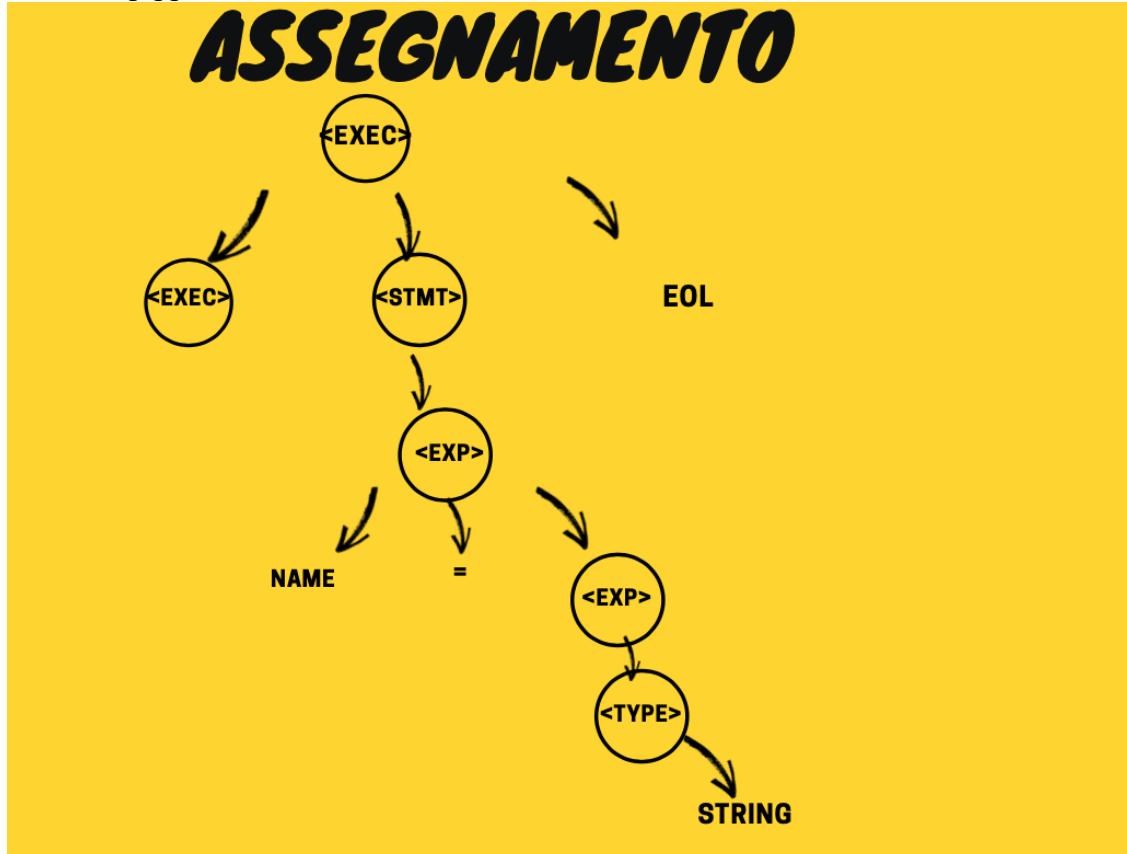
### Esempi di applicazione e relativo albero:

data la complessità della grammatica, l'albero sintattico risulta essere molto articolato e di conseguenza di non facile analisi. Si è, quindi, scelto di farne l'analisi effettuando un'analisi discendente dello stesso:

#### Creazione variabile:

Istruzione:

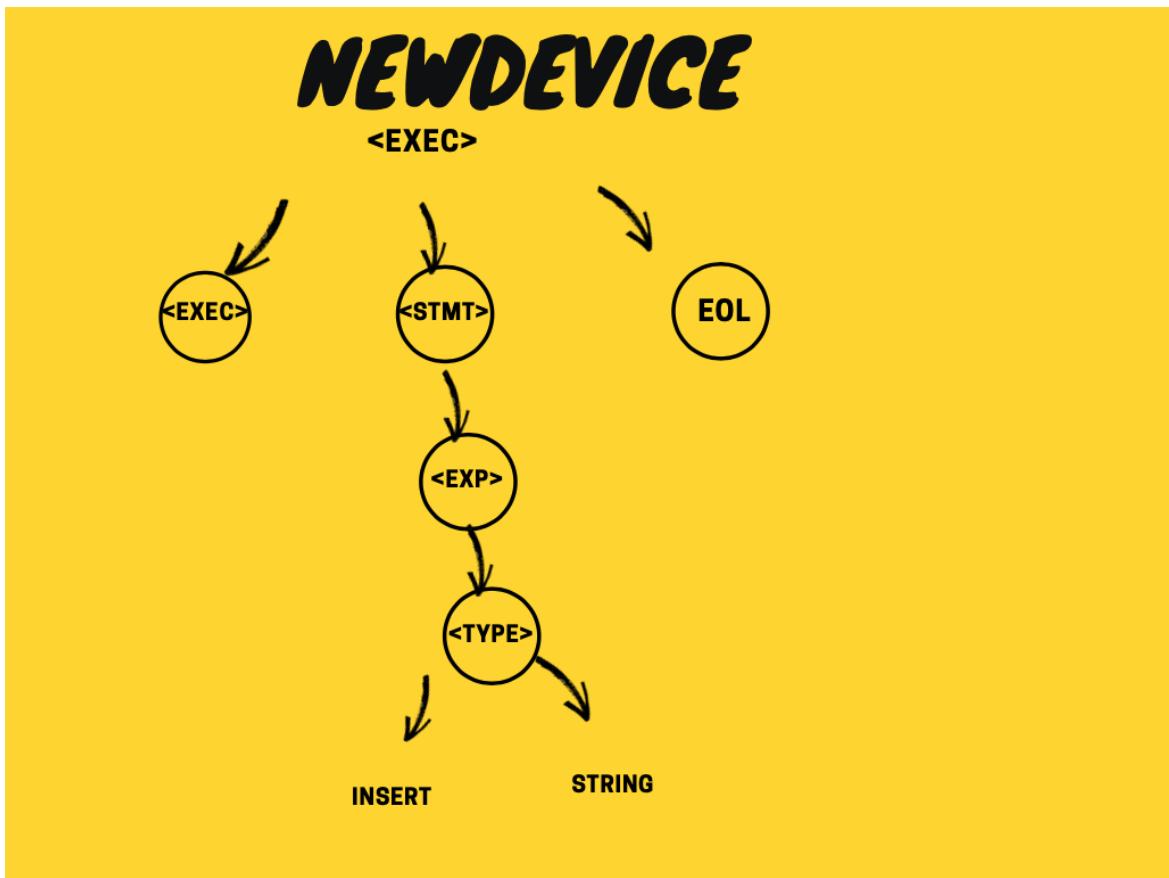
variabile= "pippo"



Analogamente, non è detto che la seconda exp riportata al terzo livello dell'albero sia una STRING. Potrebbe essere un NUMBER o una qualsiasi altra delle espressioni che il programma ha definito.

#### Creazione del device:

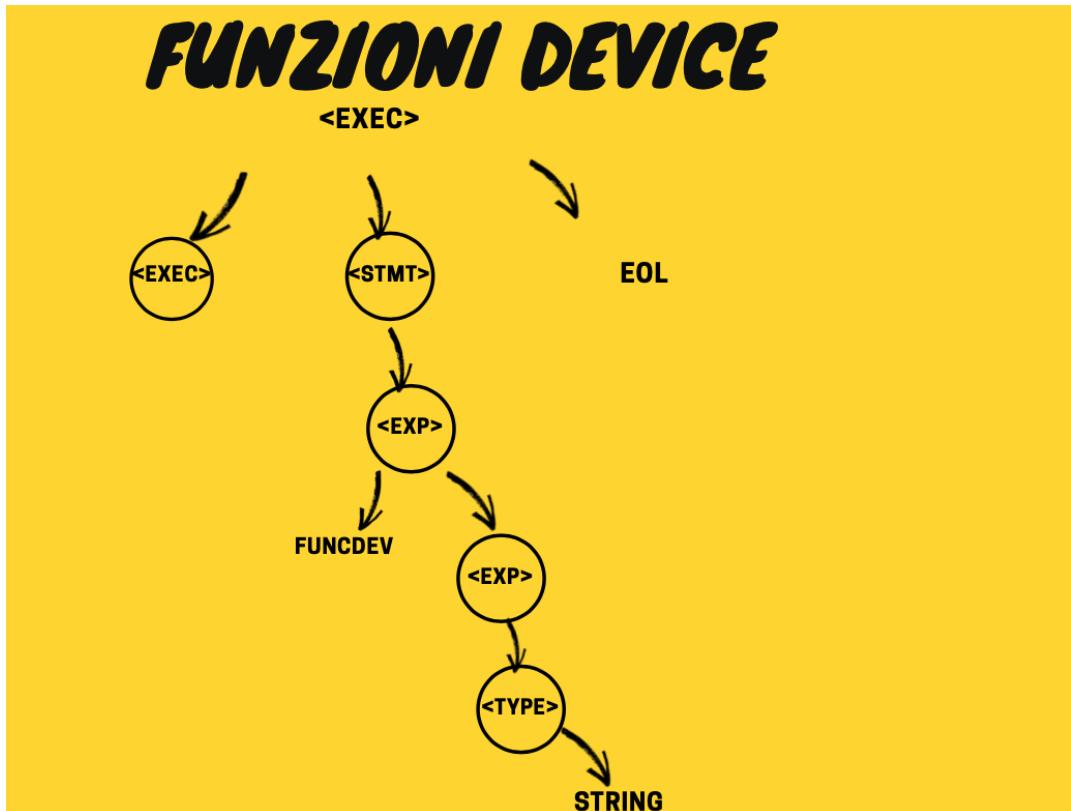
Istruzione: newDevice "pippo"



**FUNCDEV:** Tutte le funzioni di device (switchOn, switchOff, connect, reconnect, status e archive) seguono un albero di questo tipo.

Istruzione:

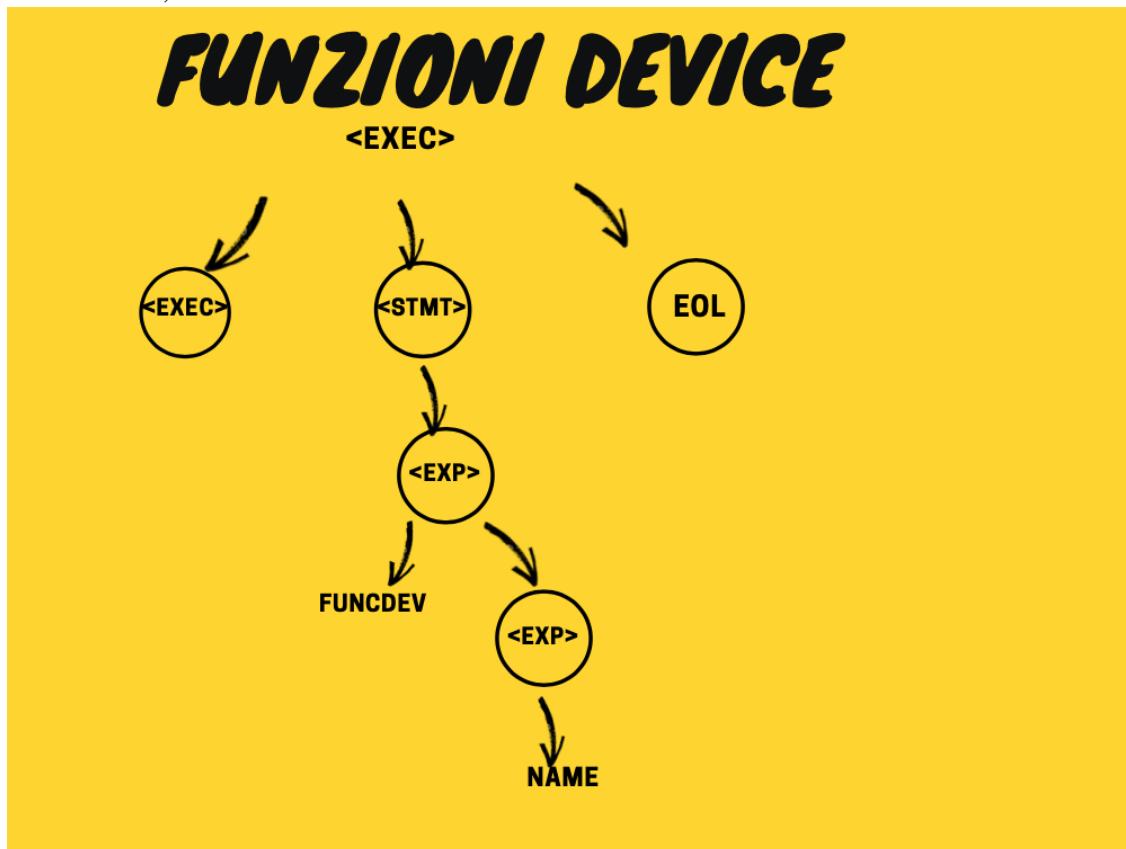
- switchOn “pippo”
- Connect “pippo”
- Reconnect “pippo”
- Status “pippo”
- Delete “pippo”



Ovviamente l'ultimo nodo terminale STRING è una generica espressione pertanto può essere anche un'espressione più sofisticata. Per esempio:

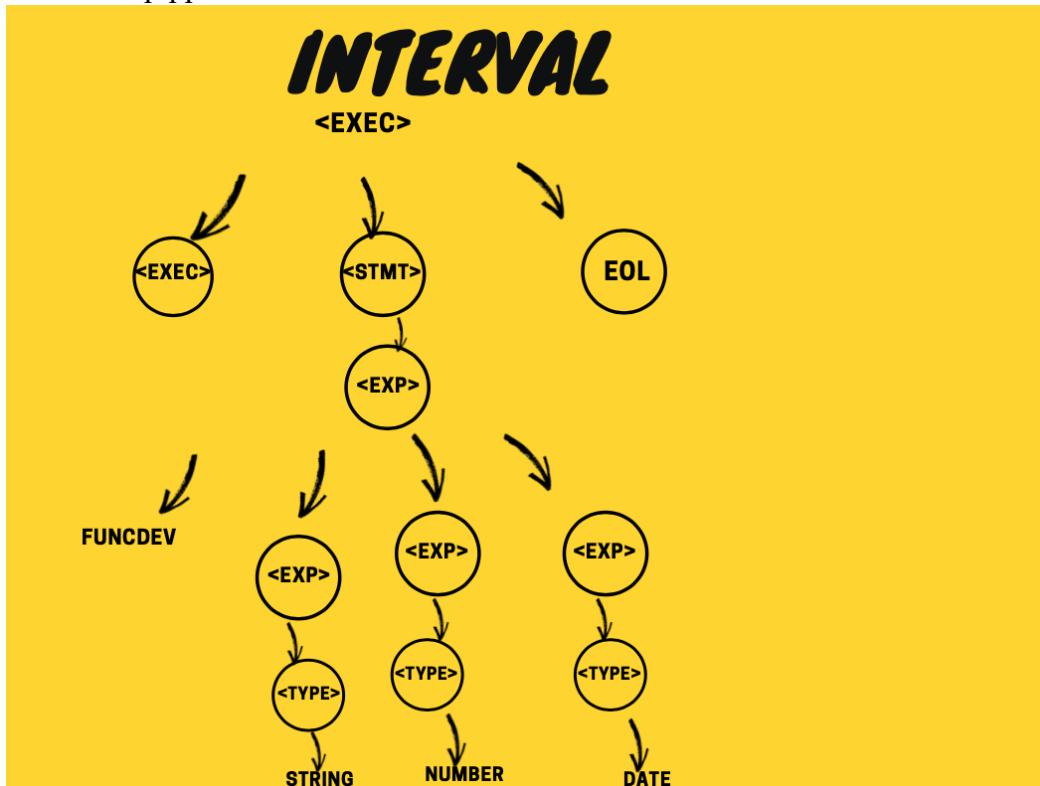
- Supponiamo di creare la variabile: dino="pippo"
- E successivamente di effettuare: switchOn dino

L'operazione risulterà analoga a quella effettuata nel passo precedente con switchOn "pippo". Nonostante ciò, l'albero risulterà notevolmente differente:



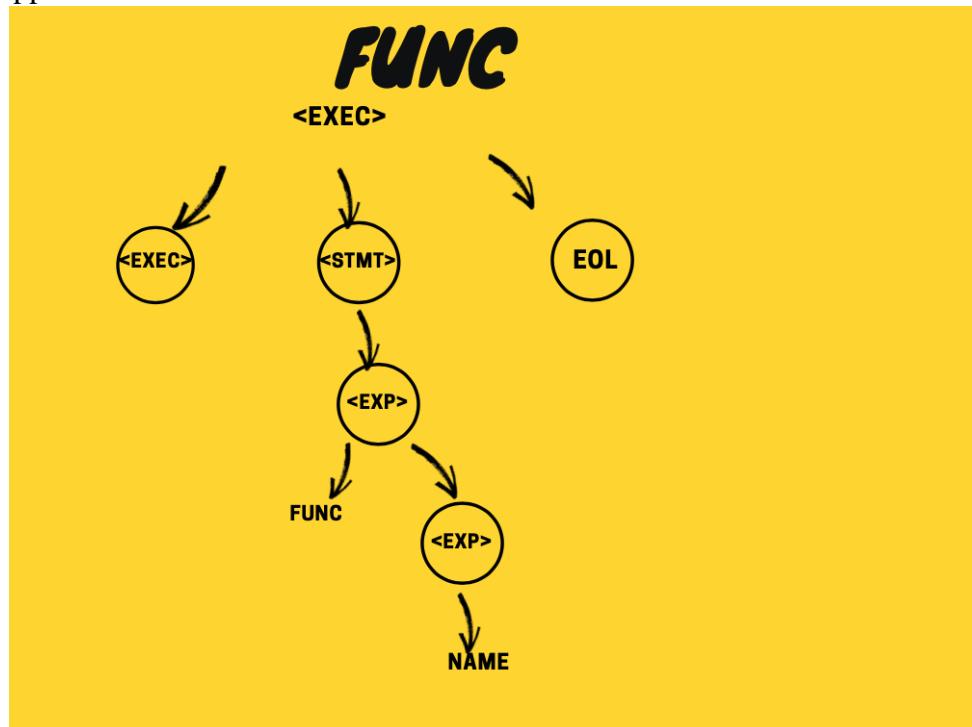
**INTERVAL:** Ha la novità di prevedere tre parametri di input e quindi l'albero risulta con due nodi terminali in più nel livello più basso dell'albero:

- Interval “pippo”-5-2021.9.9.9.5



**FUNC:** Tutte le funzioni di sistema create (print e readFile) seguono un albero del tipo:

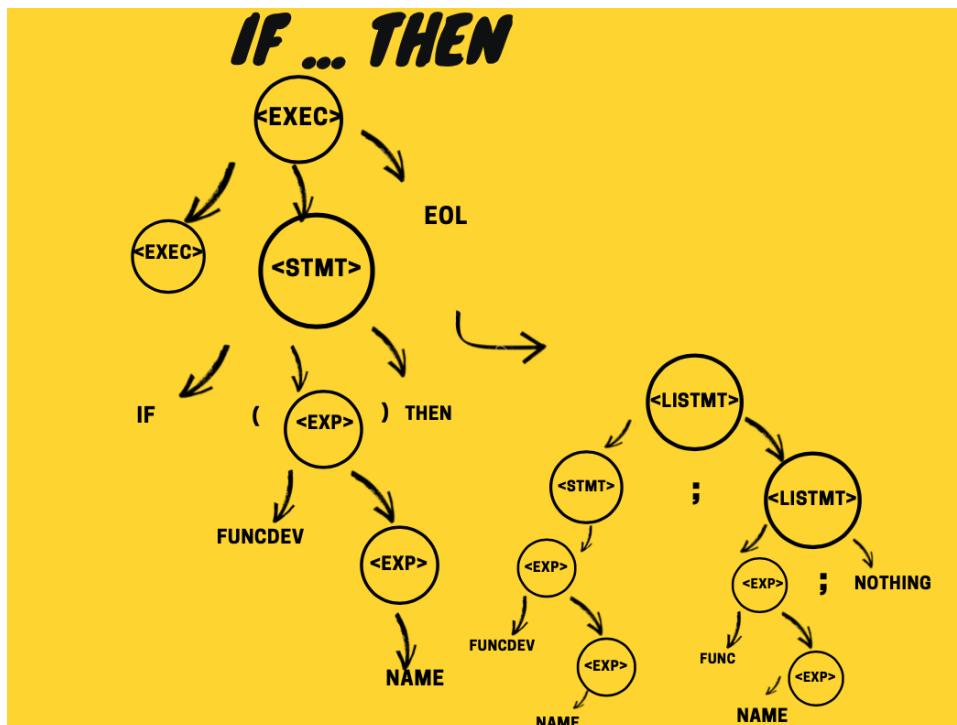
- Supponiamo di creare la variabile: dino=”pippo”
- E successivamente di effettuare: print dino
- Oppure readFile dino



Gli alberi riportati sono alcuni casi che si possono presentare e consentono di analizzare le funzionalità di base del linguaggio.

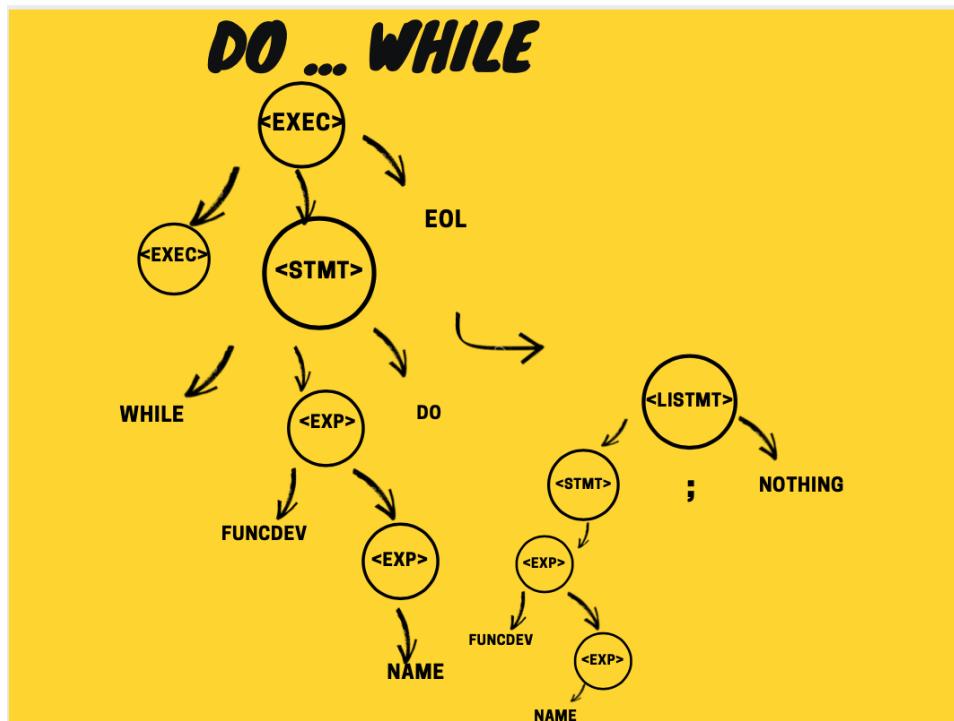
**IF .... THEN:** Tutte le operazioni condizionali (if then) seguono un albero del tipo:

- Supponendo di avere:
  - V="pippo", H="FINISH"
  - If (connect v) then switchOn v; print "FINISH";
    - significato: Se il device pippo è raggiungibile allora accendilo e stampa FINISH
  - Si noti che i nodi terminali dell'albero sono:
    - IF
    - THEN
    - NAME (identifica le variabili v,h)
    - FUNCDEV (rappresenta connect e switchOn)
    - FUNC (rappresenta print)



**DO.... WHILE:** Tutte le operazioni iterative (do while) seguono un albero del tipo:

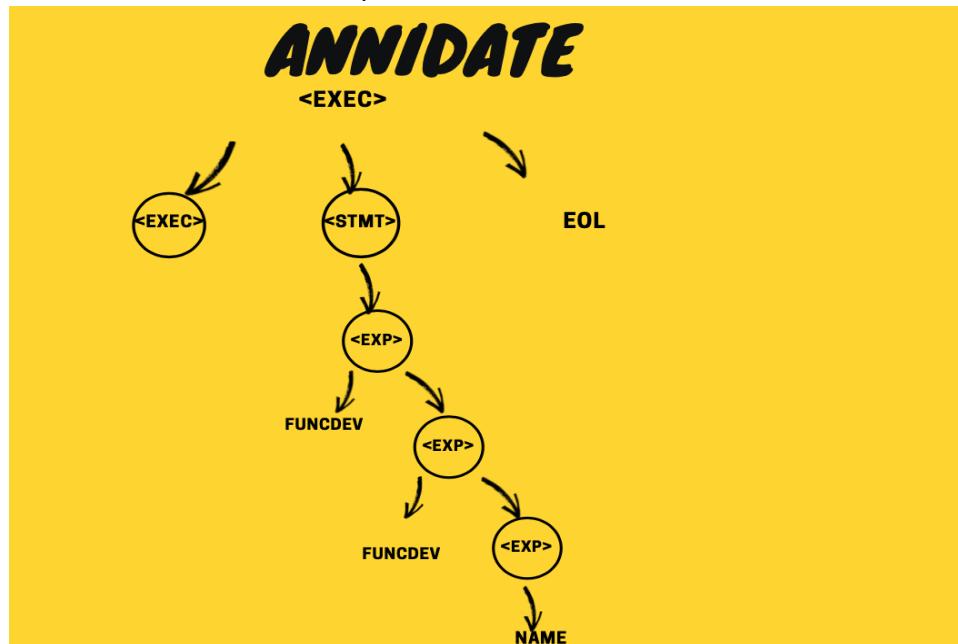
- Esempio: v="ciccio"
  - while (connect v) do reconnect v;



**Funzioni embedded annidate:** Per rendere il linguaggio più semplice è stata prevista la possibilità di richiamare funzioni annidate. All’utente viene quindi data la possibilità di annidare le diverse operazioni embedded che vuole effettuare.

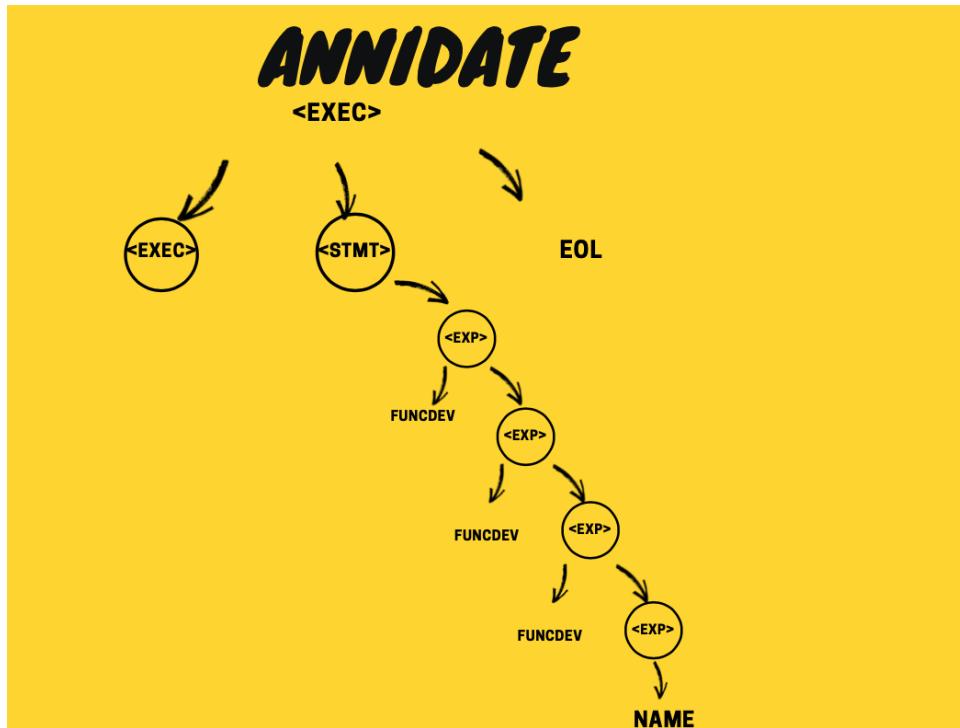
- Esempio:
  - 1. Se l’utente volesse prima connettersi al device è poi attivarne lo status per avviare l’operazione di innaffiatura potrebbe effettuare:
    - SwitchOn connect “ciccio”

Prima il device verrà connesso e solo dopo acceso. È un modo di rendere il linguaggio più semplice a chi lo usa e vanificando possibili errori in fase di esecuzione.



Di fatto non si è reso che l’albero più profondo. Tale profondità può aumentare sempre di più in base al livello di annidamento delle funzioni.

- 2. L'utente può anche pensare di prima verificare se il device è raggiungibile, e se effettivamente lo è prima ne disattiva lo status e poi lo cancella:  
delete switchOff connect "ciccio"

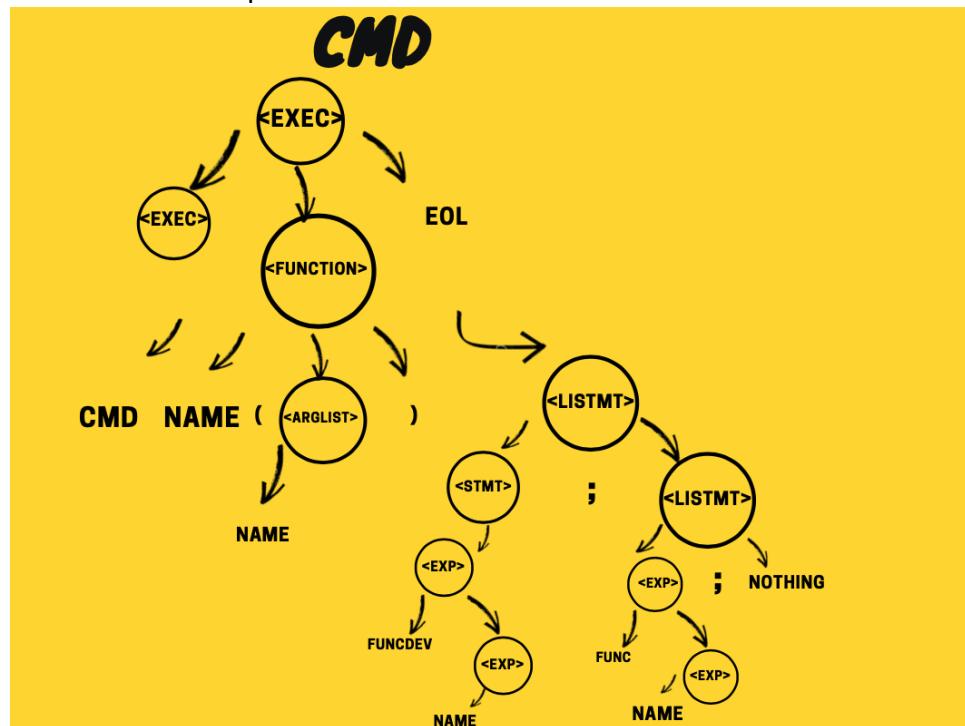


## **Funzioni definite di sistema:**

E' stata anche data la possibilità all'utente di creare le sue funzioni di comodo che possono risultare più o meno complesso. Ipotizziamo una funzione del tipo:

- o CMD nomeFunzione (v) = If (connect v) then switchOn v; print "FINISH";

L'albero risulta essere del tipo:



## 5.Casi d'uso:

### Sintassi ad alto livello:

#### 5.1 Tipi:

##### 5.1.1 Tipi primitivi:

I tipi previsti nel linguaggio di programmazione sono:

- Int: numeri interi con segno
- Float: numeri con la virgola dove l'operatore punto separa i numeri decimali da quelli non. La precisione è fino alla seconda cifra decimale.
  - 11.12
- Char: Sono delle stringhe che contengono un solo carattere. Indicate tra virgolette:
  - "c"
- Stringhe: Sono delle stringhe che contengono un insieme di caratteri. Si indicano tra virgolette.
  - "NomeStringa"

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

Figura 4 - confronto dei Tipi tra linguaggi di programmazione

##### 5.1.2 Tipi composti:

- Mapping: Array
- Liste (tipi ricorsivi): Liste che contengono l'elenco dei device.
- Device: Sarà presente una funzione embedded (newDevice “nomeDevice”) che si occupa di generare un oggetto Device. Tipo struttura composto dai seguenti elementi:
  - Status: intero (0 o 1) che indica se il dispositivo è acceso o spento,
  - L: indica la lista dei device a esso collegati,
  - Symbol: nome e caratteristiche del device considerato.
  - ReferenceListDevice {1, [DeviceA, DeviceB....DeviceN], [NomeDevice;]}

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

#### Variabili:

Una variabile è un oggetto che può essere esaminato e aggiornato. Ogni variabile nel programma sarà caratterizzata da una definizione (in definisci il tipo e il valore) e da un ciclo di vita. Nel nostro caso tale cassetto potrà contenere o tipi interi o tipi stringa o tipi NULL. Le

date per semplicità all'interno del programma sono state trattate come stringhe per, poi eventualmente essere convertite in type.

### Variabile di sistema:

Nel programma è stata definita una variabile di sistema “ans”. Tale variabile contiene l'ultima variabile literal definita all'interno del programma. La variabile “ans” funge, quindi, da contenitore dell'ultima costante inserita dall'utente ma non assegnata ad alcuna variabile.

### Tipi primitivi: (Definizione costanti e stringhe, assegnazione)

È, dunque, prevista una variabile di sistema “ans” in cui vengono memorizzate le variabili literal (costanti) passate nel programma. Se l'utente scrive sulla command line valori costanti come 1 o “ciao” questa viene memorizzato nella variabile “ans”.

Riportiamo un esempio esplicativo, è come se l'utente avesse scritto:

Prima -> Ans=1

Dopo -> Ans=”ciao”

```
> 1
> print ans
 1
> "ciao"
> print ans
ciao
```

L'utente può assegnare alle variabili costanti o identificativi di tipi primitivi e composti definiti nella sezione dedicata ai tipi. La definizione delle variabili è del tipo:

- NomeVariabile= ValueType

Ogni variabile ha un ciclo di vita per tutta la durata del programma. La variabile può essere aggiornata associandogli un nuovo valore. La variabile ricorderà solo l'ultimo valore associato alla stessa.

```
> nomeVariabile=1
> print nomeVariabile
 1
> nomeVariabile="ciao"
> print nomeVariabile
ciao
```

- NomeVariabile= FunctionEmbedded

È possibile dunque anche assegnare a una variabile il valor di ritorno di una funzione embedded (sono descritte nelle sezioni seguenti). Una funzione embedded, ovviamente, potrà restituire una String o un intero. Tutte le funzioni embedded sono state costruite per restituire il nome del device (quindi una stringa) se l'operazione è andata a buon fine, altrimenti falso.

Se facciamo riferimento alle funzioni embedded (da utilizzare sull'istanza del device precedentemente creata) restituiscono il nome del device.

Per esempio, la funzione “FUNCDEV nomeDevice” restituisce il nome del device se la FUNCDEV è andata a buon fine, altrimenti NULL:

> v= connect “ciao”

> print v

ERROR:.....

```
> newDevice "ciao"
> v= connect "ciao"
> print v
```

Ciao

>

Verrà eseguita la funzione embedded newDevice che si occupa di creare un oggetto device (dev2) e memorizzarlo nella variabile v. Si nota nel caso che la funzione di sistema print è stata progettata per stampare solo variabili che contengono stringhe, numeri e date. Nel caso la variabile var non contiene né numeri né stringhe per cui viene lanciata un'eccezione.

Da notare che, per rendere molto semplice l'uso del linguaggio, l'utente non si rende conto della creazione di un oggetto device; infatti, lo stesso potrà sempre riferirsi ai device tramite le stringhe. Di conseguenza se esiste un device "ciao" nel sistema, l'utente non dovrà fare altro che scrivere ogni qualvolta la stringa "ciao" per riferirsi al device.

```
> var = newDevice "ciao"
```

Dispositivo inserito con successo e con ID: dev#3924

```
> print var
```

Ciao#3924

Le funzioni embedded inerenti al device pretendono infatti tutte come parametro il nome del device e nient'altro. Per cui l'utente inserendo il nome del device "ciao" potrà, passandolo alle varie funzioni embedded, riferirsi all'oggetto device facendo le varie operazioni necessarie. Di fatto, l'utente non agirà mai direttamente sulla struttura device del programma, con operazioni di assegnamento. L'unico modo per accedere al device è quello di passare una stringa o variabile o array alle funzioni embedded.

Infatti, se l'utente digita:

```
>ciao="10"
```

```
>print ciao
```

10

si sta riferendo a un'altra variabile ciao. Di conseguenza se richiama la funzione embedded connect "ciao", ritroverà l'oggetto device riferente a ciao. L'oggetto device a cui fa riferimento la stringa ciao, non viene quindi modificato dalla variabile ciao creata. Quindi tutte le variabili che noi generiamo (anche se hanno lo stesso nome della stringa del device) non modificano nella lookup l'oggetto device

L'idea di base è che tutte le funzioni ritornano un intero o stringa che può essere memorizzato tramite l'assegnazione a una variabile.

### Tipi composti:

I tipi composti che sono stati realizzati sono due: array e liste.

#### 1. Liste:

Le liste sono state realizzate per il tipo composito device. All'utente viene concesso di realizzare solo liste di device. Al contrario vedremo che gli array consentono di realizzare strutture composite anche per i tipi primitivi interi e stringhe. La scelta di realizzare liste solo di device è per caratterizzare meglio la struttura principale del linguaggio. Inoltre per ipotizzare device collegati il modo più semplice di memorizzarli in memoria è tramite liste. Una lista di device è specificata dall'espressione:

---

```
argsListDevice: STRING           { $$ = newargsList((struct symbol *)$1, NULL); }
| STRING ',' argsListDevice { $$ = newargsList((struct symbol *)$1, $3); }
;
```

dove newargsList è

```
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;
}
```

E l'utente potrà specificarla tramite il seguente comando:

> newDevice “a” -> [“b”, “c”]

La lista è rappresentata dalla parte dell'espressione: [“b”, “c”]. In seguito, verrà descritta nel dettaglio, nella sezione dedicata al NewDevice.

## 2. Array:

Array	
<b>newArray</b>	F.Array
<b>integer</b>	Identificatore
<b>char</b>	Identificatore
<b>integer</b>	Identificatore
<b>device</b>	Identificatore
<b>set</b>	F.Embedded
<b>get</b>	F.Embedded
<b>add</b>	F.Embedded
<b>remove</b>	F.Embedded

È stato costruito un array flessibile dato che la sua dimensione di memoria non è fissa ma può essere modificata durante l'utilizzo dell'array stesso. Per cui in fase di inizializzazione l'utente fissa una dimensione iniziale dell'array, ma questa può essere modificata dinamicamente. Gli array possono essere creati rispetto ai 3 tipi primitivi sopra elencati:

- Stringa
- Device

- Interi

Per semplificare la vita al programmatore sono state progettate 3 funzioni di sistema che consentono di effettuare le operazioni fondamentali di un array:

- modificare i valori dell'array,
- creare l'array,
- aggiungere elementi nella posizione successiva dell'array,
- rimuovere l'ultimo elemento,
- stampare l'array,
- restituire l'elemento nella posizione i-esima dell'array.

Il motivo per cui si è preferito non dare un controllo diretto dell'array all'utente è sempre quello di rendere il programma più semplice possibile. Se non avessimo implementato le funzioni embedded (che di seguito descriviamo) l'utente avrebbe dovuto avere un accesso diretto agli array usando strutture come puntatori. L'idea è che quindi l'utilizzatore finale userà tali funzioni embedded per la gestione base di un array.

Tutte le funzioni embedded sono di questo tipo:

- NomeArray operazione = parametro1, parametro2

Operazione sarà: add, set, get, remove

Parametri: interi, stringhe o device

NomeArray: sarà una stringa

Gli array possono essere generati basandosi sui 3 tipi primitivi sopra descritti:

### Array di interi:

- Creazione di un array:

Per creare l'array è stata prevista la seguente produzione:

Array: ARRAY nameType NAME '(' NUMBER ')' ';'

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

I token della produzione sono:

- ARRAY -> token avviato quando l'utente digita “newArray”
- NAME -> token che identifica una qualsiasi sequenza di caratteri, serve di fatto a identificare una stringa
- NUMBER -> token NUMBER identifica

NameType invece è un ulteriore produzione che serve a identificare il tipo dell'array, se intero, stringa o device.

Per cui l'array è creato quando l'utente digita da terminale:

---

```
> newArray integer pippo (2);
ARRAY CREATO CORRETTAMENTE
Operazione di inserimento dispositivo completata con successo
```

Con tale operazione, a basso livello, viene aggiornata la tabella memorizzando l'area di memoria in cui è stato creato l'array.

- Aggiunta di elementi in un array:

Per aggiungere elementi in un array di interi precedentemente creato, è sufficiente che l'utente utilizzi la funzione add nel seguente modo:

- NomeArray-> add = valore da inserire nell'array

Come parametro per cui prende l'intero da inserire.

```
> newArray integer pippo (2);
ARRAY CREATO CORRETTAMENTE
Operazione di inserimento dispositivo completata con successo
```

```
> pippo->add = 2
> pippo-> add = 3
> pippo-> add = 9
> pippo-> add = "ciao"
> pippo->add = newDevice "pippo"
Dispositivo già Esistente con ID: pippo#5694
```

Nel caso che nell'array pippo siano stati aggiunti i 3 interi: 2,3,9. Provando ad aggiungere un valore primitivo non corrispondente al tipo di creazione dell'array l'elemento non verrà eseguito. Nel caso all'array pippo si è cercato di associare una stringa e un tipo device ma non venendo memorizzato in memoria (in seguito vedremo con la funzione get che l'array infatti conterrà 2,3,9; ma non i valori non corrispondenti al tipo stringa e device)

- Stampa array:

Analogamente per la stampa di un array è stata prevista la funzione get, che stampa l'intero contenuto dell'array. Nell'esempio si noti come l'array contenga i tipi interi che gli sono stati assegnati ma non i tipi stringa e device.

L'elemento che viene aggiunto verrà inserito sempre in coda alla lista dell'array. Per inserire infatti elementi in una posizione specifica vedremo in seguito la funzione set.

```

> pippo->add = 2
> pippo-> add = 3
> pippo-> add = 9
> pippo-> add = "ciao"
> pippo->add = newDevice "pippo"
Dispositivo già Esistente con ID: pippo#5694
> pippo-> get
2
3
9

```

- Rimozione elemento dell'array:

Per la rimozione di un elemento dell'array è sufficiente usare la funzione remove, per consentire la rimozione dell'ultimo elemento dell'array. Si noti che le funzioni remove e add accoppiate consentono di realizzare una sorta di coda aggiungendo e rimuovendo elementi in testa e in coda.

```

> pippo-> get
2
3
9

> pippo-> remove

> pippo-> get
2
3

> pippo-> get
2
3

> pippo-> remove

> pippo-> get
2

> pippo->remove

```

- Aggiornamento di un elemento dell'array in una posizione specifica:

All'utente viene anche data la possibilità di settare un elemento di un array in una specifica posizione. La funzione set prende due parametri che sono rispettivamente: il nuovo valore da associare all'array e la posizione in cui inserire quel valore. Entrambi i valori sono specificati a destra dell'uguale. Nel caso in cui la posizione specificata dall'utente è maggiore della dimensione massima dell'array quell'elemento non verrà settato perché non esistente.

Con questo primo esempio viene settato l'elemento in posizione 0:

```

> pippo-> get
5
6

> pippo-> set = 0,0

> pippo->get
0
6

```

In quest'altro l'elemento in posizione 1:

```
> pippo-> set = 1,0
> pippo->set = 4,1
> pippo-> set = 9,9
> pippo->get
1
4
```

Si noti che in quest'ultimo esempio si è provato a settare anche l'elemento in posizione 9 dell'array col valore 9; ma come si nota con la get gli elementi dell'array rimangono due.

- Ritorno dell'elemento in posizione i-esima dell'array:

Finora le funzioni embedded non restituivano alcun valore memorizzabile in una qualche variabile all'utente ma servivano semplicemente o a stampare i valori dell'array o a modificare la struttura interna dell'array. Quest'ultima funzione consente di ritornare l'elemento in una posizione specifica dell'array. È la funzione get a cui passi a destra dell'uguale l'indice dell'elemento dell'array che ti interessa. L'utilità di questa funzione è che consente di usare gli array (scorrendo opportunamente l'albero) insieme alle varie funzioni embedded del device che saranno spiegate in maniera approfondita nelle sezioni seguenti.

Nell'esempio che segue vengono ritornati nell'array prima l'elemento in posizione 1 e poi l'elemento in posizione 0.

```
> pippo->get
1
4

> pippo->get = 1
4
>
> pippo->get=0
1
>
> pippo->get
1
4
```

### **Array di stringhe:**

Analogo uso degli array può essere fatto per creare un array di stringhe. La caratteristica fondamentale degli array è quindi la tipizzazione. Appena si definisce l'array ne sarà definito il tipo; e in base a quello farò le operazioni classiche per un array

- Creazione di un array:

La creazione dell'array di stringhe è lo stesso per gli array di interi solo che stavolta verrà specificato che il tipo di riferimento degli array è il tipo stringa tramite il token CHAR. Di seguito, si riporta un esempio in cui viene creato l'array di stringhe:

```
> newArray char angelo (2);
ARRAY CREATO CORRETTAMENTE
Operazione di inserimento dispositivo completata con successo
```

Le operazioni di aggiunta, rimozione, etc sono analoghe a quelle spiegate per gli array di interi. Le funzioni spiegate sono esattamente le stesse solo che prendono parametri diversi. Per cui si riporta solo un esempio basilare per ognuna

- Aggiunta di elementi in un array:

Prende come parametro una stringa e lo aggiunge nell'array. La funzione sarà sempre add:

```
> angelo->add = "minotauro"
> angelo->add=1
> angelo->newDevice "ciao"
35: Errore: syntax error

> angelo->add = newDevice "ciao"
Dispositivo già Esistente con ID: ciao#3924
```

```
> angelo->add = "artica"
> angelo->add = "cane"
```

- Stampa array:

```
> angelo->get
minotauro
artica
cane
```

- Rimozione elemento dell'array:

```
> angelo->remove
> angelo->remove
> angelo->get
minotauro
> angelo->remove
> angelo->get
> angelo->add ="nonna"
> angelo->remove
> angelo->get
```

- Aggiornamento di un elemento dell'array in una posizione specifica:

```

> angelo->get

> angelo->add = "ciao"

> angelo->add = "dino"

> angelo-> get
ciao
dino

> angelo -> set = "peppino",1

> angelo->set= 9,9

> angelo->get
ciao
peppino

```

- Ritorno dell'elemento in posizione i-esima dell'array:

```

> angelo -> get
ciao
peppino

> angelo -> get = 1
peppino
>
> angelo -> get = 0
ciao

```

### **Array di Device:**

Analogo discorso può essere fatto per i device, con l'unica differenza che memorizzano il nome dell'istanza del device, questo permette di interfacciarsici. Ovviamente, le operazioni che consentono di generare un device saranno spiegate in seguito, per cui si consiglia di leggere prima la funzione newDevice, e solo in seguito visualizzare tale sezione. Le operazioni sono comunque le stesse degli array di stringhe e interi:

- Creazione di un array:

```

> newArray device dev (2);
ARRAY CREATO CORRETTAMENTE
Operazione di inserimento dispositivo completata con successo

```

- Aggiunta di elementi in un array:

```

> dev->add = newDevice "dev1"
Dispositivo inserito con successo con ID: dev1#9782

> dev->add = newDevice "dev2"
Dispositivo inserito con successo con ID: dev2#9781

```

- Stampa array:

```

> dev->get
dev1#9782
dev2#9781

```

- Rimozione elemento dell'array:

```
> dev->get
dev1#9782
dev2#9781

> dev-> remove

> dev->get
dev1#9782

> dev-> remove

> dev-> remove

> dev->get
```

- Ritorno dell'elemento in posizione i-esima dell'array:

```
> newDevice "pippo"
Dispositivo inserito con successo con ID: pippo#5694
Operazione di inserimento dispositivo completata con successo

> newArray char v (2);
ARRAY CREATO CORRETTAMENTE
Operazione di inserimento dispositivo completata con successo

> v->add = "pippo"

> v->get
pippo

> print v->get=0
a:pippo

> connect v-> get
pippo
Ricerca del dispositivo 0 in corso...
Dispositivo 0#48: Non Esistente

> connect v->get=0
Ricerca del dispositivo pippo in corso...
Dispositivo Esistente
Richiesta connessione...
pippo#5694
```

### Ciclo di vita delle variabili (non Runtime):

Le variabili una volta definite sono state memorizzate in memoria per cui hanno un ciclo di vita equivalente all'esecuzione dell'interprete. Una volta che una variabile è stata infatti creata viene memorizzata in memoria e terminerà solo quando l'interprete verrà disattivato.

## Interprete

Il linguaggio che è stato sviluppato è interpretato, ossia ogni volta che viene eseguita un'istruzione questa verrà immediatamente analizzata. Non sarà quindi presente una fase di compilazione che si occupa di effettuare l'analisi sintattica del programma. Nonostante ciò, abbiamo previsto la possibilità di eseguire il programma da un file esterno in cui però la funzione embedded corrispondente (readFile, che sarà descritta nelle sezioni seguenti) si occuperà semplicemente di leggere il file e caricarlo per farlo eseguire immediatamente all'interprete, senza una previa fase di compilazione. In seguito, si trova un esempio di tale comando (readFile).

### Indentazione e formattazione

Tutti gli spazi inseriti dal programmatore verranno trascurati. L'utente può quindi indentare il codice in base alle sue preferenze, sempre considerando il fatto che a ogni digit di INVIO il comando verrà eseguito. È possibile scrivere più comandi sulla stessa linea (a discapito di una buona programmazione). L'ideale è quindi scrivere prima il programma su un file indentato e poi caricarlo tramite la funzione embedded readFile. È inoltre un linguaggio case sensitive.

### Parole chiavi del linguaggio

Nel linguaggio sono presenti alcune parole chiavi non utilizzabili dall'utente. Le parole chiavi previste sono le seguenti:

Parole chiavi	
<b>connect</b>	F.Embedded
<b>readFile</b>	F.Embedded
<b>insert</b>	F.Embedded
<b>device</b>	Identificatore
<b>bye</b>	abort
<b>interval</b>	F.Embedded
<b>switchOn</b>	F.Embedded
<b>switchOff</b>	F.Embedded
<b>delete</b>	F.Embedded
<b>print</b>	F.Embedded
<b>reconnect</b>	F.Embedded
<b>arrow</b>	Indicatore ->
<b>status</b>	F.Embedded
<b>newDevice</b>	F.Embedded
<b>CMD</b>	Identificatore
<b>if</b>	Op. condizionale
<b>then</b>	Op. condizionale
<b>else</b>	Op. condizionale
<b>repeat</b>	Op. condizionale
<b>do</b>	Op. condizionale
<b>while</b>	Op. condizionale
<b>clear</b>	F.Embedded
<b>add</b>	Indicatore ->
<b>remove</b>	F.Embedded
<b>get</b>	F.Embedded
<b>integer</b>	Identificatore
<b>char</b>	Identificatore

### Funzioni embedded:

Potendo effettuare una prima descrizione del linguaggio di programmazione si nota la necessità di avere delle funzioni di sistema per attivare e collegare i device necessari per avviare il sistema di annaffiamento e di giardinaggio. Tali funzioni hanno lo scopo di

facilitare la vita dell'utente finale. Nella seguente sezione sono descritte le principali funzioni di sistema previste:

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

- **Device \*NEWDEVICE (string):**

È una funzione a cui si passa come parametro un tipo primitivo stringa e restituisce il riferimento all' area di memoria (in questo caso alla tabella dei simboli) dove è stato memorizzato il device. Una stringa, come spiegato nelle sezioni precedenti, è un tipo primitivo del linguaggio ed è definito tra virgolette.

Nella figura sotto troviamo nell'esempio la definizione di 4 device nella posizione Nord, Sud, Est e Ovest del sistema di giardinaggio. La funzione, si nota, stampa a video le informazioni del device creato a cui è stato associato un id rappresentato dalla coppia:

**Sintassi:** newDevice “nomeDeviceString”

```

> newDevice "device_Ovest"
Dispositivo inserito con successo con ID: device_Ovest#8100
Operazione di inserimento dispositivo completata con successo

> newDevice "device_Est"
Dispositivo inserito con successo con ID: device_Est#7565
Operazione di inserimento dispositivo completata con successo

> newDevice "device_Nord"
Dispositivo inserito con successo con ID: device_Nord#5304
Operazione di inserimento dispositivo completata con successo

> newDevice "device_Sud"
Dispositivo inserito con successo con ID: device_Sud#8477
Operazione di inserimento dispositivo completata con successo

```

> █ +

L'effetto della funzione è quello di generare e ritornare un oggetto device con:

- Status: off/0 (il device inizialmente sarà disattivato/spento e quindi non raggiungibile dall'utente per effettuare operazioni di giardinaggio)
- L: NULL (lista dei device a cui è collegato. Nel nostro caso sarà null dato che tale device non coopera con altri dispositivi)

#### • **Device \*NEWDEVICE (string, listaDevice):**

Questa funzione ha le stesse caratteristiche della funzione precedente solo che oltre a creare l'oggetto device si preoccupa di definire la lista dei device con cui coopererà per effettuare tale operazione. Quindi l'operazione consiste nella creazione di un nuovo oggetto device (rappresentato dal primo parametro string). Questo nuovo device creato sarà collegato a una lista di device definito come secondo parametro. I device passati come secondo parametro dovranno essere esistenti, in caso contrario verrà segnalata all'utente la necessità di creare quei device prima di effettuare una successiva operazione.

Sintassi: newDevice “deviceDaCreare” -> [“device1”, “device2”, ..... , “deviceN”]

Nell'esempio riportato viene generato il device d3 che sarà collegato ai device d1 e al device d2 (già esistenti perché creati in precedenza)

```

> newDevice "deviceOvest"
Dispositivo inserito con successo con ID: deviceOvest#451
Operazione di inserimento dispositivo completata con successo

> newDevice "deviceEst"
Dispositivo inserito con successo con ID: deviceEst#762
Operazione di inserimento dispositivo completata con successo

> newDevice "deviceNord"
Dispositivo inserito con successo con ID: deviceNord#295
Operazione di inserimento dispositivo completata con successo

> newDevice "deviceSud"
Dispositivo inserito con successo con ID: deviceSud#106
Operazione di inserimento dispositivo completata con successo

```

Nell'esempio seguente invece viene creato il device d4. Tale device sarà collegato ai device d1,d2,d3,d5. Il device d5 non è però esistente, per tale motivo il sistema segnala la necessità di attivare nel sistema di giardinaggio il dispositivo. L'utente nella fase successiva si occupa infatti di generare il device D5.

```
>newDevice "d1"
Dispositivo inserito con successo con ID: d1#949
Operazione di inserimento dispositivo completata con successo

> newDevice "d2"
Dispositivo inserito con successo con ID: d2#950
Operazione di inserimento dispositivo completata con successo

> newDevice "d3" -> ["d1", "d2"]
Dispositivo inserito con successo con ID: d3#951
d3#951 connesso con -> [ [d1#949] - [d2#950] ]

> newDevice "d4" -> ["d1", "d2", "d3", "d5"]
Dispositivo inserito con successo con ID: d4#944
d4#944 connesso con -> [ [d1#949] - [d2#950] - [d3#951] - [d5#945] * ]
Devices con (*) sconosciuti, inserire devices

> newDevice "d5"
Dispositivo inserito con successo con ID: d5#945
Operazione di inserimento dispositivo completata con successo

>
```

La funzione applicata restituisce il riferimento alla zona di memoria in cui viene memorizzato il device creato. A differenza della funzione newDevice della sezione precedente però il device generato avrà le seguenti caratteristiche:

- Status: off/0 (il device inizialmente sarà disattivato/spento e quindi non raggiungibile dall'utente per effettuare operazioni di giardinaggio)
- L: lista di device a cui è collegato. (nel caso di D4 saranno d1, d2, d3, d5)

#### • **Nodo/char \*connect (string):**

Anche questa rappresenta una funzione embedded con lo scopo di effettuare richiedere la connessione ad un device precedentemente creato. Una volta che è stato istanziato l'oggetto Device (tramite le funzioni embedded precedenti) è possibile verificare la raggiungibilità di un device. La funzione connect è, quindi, assimilabile all' handshake; il quale scopo è di stabilire una connessione con il device. La funzione, quindi, preso in input il device come stringa verificherà la sua esistenza nella tabella dei simboli. In caso affermativo riceverà una response da parte del Device, da noi non implementata, contenente il messaggio di risposta HTTP 200 (risposta standard per le risposte HTTP andate a buon fine). Il meccanismo con cui verrà effettuata, quindi, tale connessione non è stato dunque implementato e viene soltanto simulato tramite una printf che stampa un messaggio di avvenuta connessione in linea di comando.

Viene riportato sotto un esempio.

È, ovviamente, necessario memorizzare il device nello schema di device presenti nel sistema. Non è stato implementato un meccanismo di risposta di alcun tipo, ma concettualmente il device al successivo messaggio di richiesta connessione avrebbe dovuto verificare l'avvenuta raggiungibilità del dispositivo.

```
> newDevice "deviceOvest"
Dispositivo inserito con successo con ID: deviceOvest#451
Operazione di inserimento dispositivo completata con successo

> connect "deviceOvest"
Ricerca del dispositivo deviceOvest in corso...
Dispositivo Esistente
Richiesta connessione...
```

Nel caso in cui il device non esista, verrà segnalato l'errore:

```
> newDevice "deviceOvest"
Dispositivo inserito con successo con ID: deviceOvest#451
Operazione di inserimento dispositivo completata con successo

> connect "deviceOvest"
Ricerca del dispositivo deviceOvest in corso...
Dispositivo Esistente
Richiesta connessione...

> connect "device"
Ricerca del dispositivo device in corso...
Dispositivo device#8496: Non Esistente
```

- **Nodo/char \* reconnect (string):**

Analogia alla funzione connect ma effettua la riconnessione del dispositivo. Può essere paragonata al recupero di una connessione dopo un handshake fallito:

```
> newDevice "dev0"
Dispositivo inserito con successo con ID: dev0#9783
Operazione di inserimento dispositivo completata con successo

> reconnect "dev0"
Ricerca del dispositivo dev0 in corso...
Dispositivo Esistente
Richiesta Riconnessione...
```

- **Nodo/char \* switchOn (string):**

Una volta effettuata una connect (per verificare che il device sia effettivamente raggiungibile) l'utente può pensare di considerare il dispositivo come attivo, e quindi pronto per effettuare l'operazione di giardinaggio (per esempio di annaffiamento). L'accensione del dispositivo è stata intesa come la variazione dello status dell'oggetto del device. Una volta che il device è stato memorizzato nella tabella dei simboli è infatti sufficiente variare lo status del dispositivo da on a off. Nel caso in cui l'istanza del device sia già stata creata, ritornerà il nome del device, altrimenti ritornerà NULL.

Un device che, quindi, è stato in precedenza creato viene aggiornato con le seguenti caratteristiche:

- Status: on/1 (il device inizialmente era disattivato/spento e quindi non raggiungibile dall'utente per effettuare operazioni di giardinaggio, ora si è verificato essere attivo e pronto per l'utilizzo)

- L: lista di device a cui è collegato: invariato

La funzione prende come parametro una stringa che identifica l'oggetto Device in precedenza memorizzato nel sistema.

Sintassi: switchOn “deviceDaAttivare”

```
> newDevice "deviceSud"
Dispositivo inserito con successo con ID: deviceSud#106
Operazione di inserimento dispositivo completata con successo

> switchOn "deviceSud"
inizia switchOn
                                Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo deviceSud in corso...
Dispositivo Esistente
Richiesta connessione...
                                La connect è andata a buon fine e il dispositivo è stato acceso
```

- **Nodo/char\* switchOff (string):**

Analogo alla funzione switchOn ma commuta il dispositivo nello stato disattivo, ossia settando lo status a 0:

- Status: off/0 (il device si pone nello stato disattivato/spento e, quindi, non raggiungibile dall'utente per effettuare operazioni di giardinaggio)
- L: lista di device a cui è collegato: invariato

La funzione prende come parametro una stringa che identifica l'oggetto Device in precedenza memorizzato nel sistema.

Sintassi: switchOff “deviceDaDisattivare”

```
> newDevice "deviceSud"
Dispositivo inserito con successo con ID: deviceSud#106
Operazione di inserimento dispositivo completata con successo

> switchOn "deviceSud"
inizia switchOn
                                Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo deviceSud in corso...
Dispositivo Esistente
Richiesta connessione...
                                La connect è andata a buon fine e il dispositivo è stato acceso

> switchOff "deviceSud"
                                Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo deviceSud in corso...
Dispositivo Esistente
Richiesta connessione...
                                Il dispositivo è spento
```

- **char \* interval(string, number,date):**

Si occupa di attivare il device per un certo intervallo di tempo. Di fatto effettua l'annaffiamento. La caratteristica di tale funzione è di attivare il device (per avviare l'operazione di gocciolamento) per un certo intervallo di tempo (definito dall'utente) a partire da una certa data (anche essa specificata dall'utente). La funzione prende come parametro:

- il device sottoforma di stringa (definito precedentemente con la funzione newDevice),
- il numero di secondi in cui il device deve rimanere attivo.
- E la data a partire dal quale il device deve essere attivato (la data è stata di fatto considerata come un tipo string).
- La funzione effettua quindi le seguenti operazioni:
  - a. Avvia un thread/processo che resterà in attesa fino alla data passata come parametro
  - b. Arrivati alla data passata come parametro il device riceve un segnale che ne fa variare lo status.
  - c. SwitchOn: accende il device settandolo come attivo
  - d. Fa rimanere attivo il device per number secondi
  - e. SwitchOff: passati number secondi disattiva il device

Si nota che quindi la funzione embedded interval è di fatto una ridefinizione delle due funzioni embedded analizzate nelle due sezioni precedenti. L'obiettivo è quindi attivare lo status del device per un intervallo di tempo limitato necessario a effettuare una determinata operazione di giardinaggio. Come per la funzione switchOn, nel caso in cui l'istanza del device è già stata creata, ritorna il nome del device, altrimenti ritorna null.

Sintassi: interval "deviceDaAttivare" -numeroSecondi-dataPotatura

Passi:

- a. Attesa del device fino alla data passata come parametro (simulato per semplicità con uno sleep):

```
> newDevice "ciccio"
Dispositivo inserito con successo con ID: ciccio#6132
Operazione di inserimento dispositivo completata con successo

> interval "ciccio"--9-2020.9.10.16.43

IL DEVICE ciccio SI ATTIVERÀ PER 9 SECONDI, ALLE 35.000000
```

- b. Attivazione del device per l'operazione di annaffiatura:

```
>
AVVIO OPERAZIONE:
                    Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo ciccio in corso...
Dispositivo Esistente
Richiesta connessione...
                    La connect è andata a buon fine e il dispositivo è stato acceso
```

- c. Il device annaffia per 9 secondi nel caso
- d. Conclusi i 9 secondi il device viene spento:

Verifica in corso della connessione del dispositivo....  
 Ricerca del dispositivo ciccio in corso...  
 Dispositivo Esistente  
 Richiesta connessione...  
 Il dispositivo è spento

L'OPERAZIONE DI INTERVAL è CONCLUSA, IL THREAD è TERMINATO

Esempio completo:

```
> interval "ciccio"-9-2020.9.10.16.43
IL DEVICE ciccio SI ATTIVERÀ PER 9 SECONDI, ALLE 35.000000

> connect "ciccio"
Ricerca del dispositivo ciccio in corso...
Dispositivo Esistente
Richiesta connessione...

> newDevice "arcobaleno"
Dispositivo inserito con successo con ID: arcobaleno#6708
Operazione di inserimento dispositivo completata con successo

>
AVVIO OPERAZIONE:
Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo ciccio in corso...
Dispositivo Esistente
Richiesta connessione...
La connect è andata a buon fine e il dispositivo è stato acceso

Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo ciccio in corso...
Dispositivo Esistente
Richiesta connessione...
Il dispositivo è spento

L'OPERAZIONE DI INTERVAL è CONCLUSA, IL THREAD È TERMINATO
```

Nel caso in cui i parametri passati non siano coerenti non è previsto l'avvio di alcun thread per attivare il device:

```
> interval "ciccio"-9-2020.9.9.17.54
IL DEVICE ciccio SI ATTIVERÀ PER 9 SECONDI, ALLE -1.000000
I dati passati in input non necessitano dell'attivazione del device

> interval "conno"--"dino"-2020.9.10.17.55
IL DEVICE conno SI ATTIVERÀ PER 0 SECONDI, ALLE 1156.000000
I dati passati in input non necessitano dell'attivazione del device

> interval 1-9-2020.9.9.18.00
IL DEVICE 1 SI ATTIVERÀ PER 9 SECONDI, ALLE -1.000000
I dati passati in input non necessitano dell'attivazione del device

> interval 1-9-2020.9.10.18.0
IL DEVICE 1 SI ATTIVERÀ PER 9 SECONDI, ALLE 1399.000000
I dati passati in input non necessitano dell'attivazione del device
```

- **Void \*delete(string):**

La funzione di cancellazione quando applicata si occupa di eliminare i device presenti nel sistema (cancellandoli di conseguenza dalla tabella dei simboli). La funzione pertanto prende come parametro una stringa individuata dal nome del device da cancellare nel sistema di giardinaggio attualmente considerato nel sistema. Le operazioni pertanto sono:

- a. Verificare l'esistenza del device nel sistema
- b. Cancellazione del device utilizzabili (cancellazione nella tabella dei simboli)
- c. Salvare il nome del device nel file “device.txt”

Sintassi: delete “nomeDevice”

Esempio d'uso:

```
> newDevice "deviceOvest"
Dispositivo inserito con successo con ID: deviceOvest#451
Operazione di inserimento dispositivo completata con successo

> newDevice "deviceNord"
Dispositivo inserito con successo con ID: deviceNord#295
Operazione di inserimento dispositivo completata con successo

> newDevice "deviceSud"
Dispositivo inserito con successo con ID: deviceSud#106
Operazione di inserimento dispositivo completata con successo

> delete "deviceOvest"

> delete "deviceNord"

> delete "deviceSud"

> connect "deviceSud"
Ricerca del dispositivo deviceSud in corso...
Dispositivo deviceSud#106: Non Esistente
```

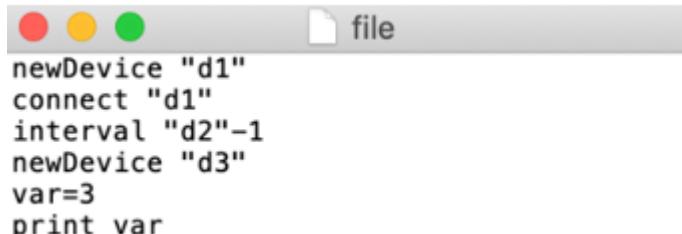
- **Void \* readFile (String):**

È stata prevista la possibilità di consentire all'utente di eseguire i programmi sia da linea di comando che da file. Per far ciò è stata prevista una funzione di sistema readFile che, passato come parametro una stringa che rappresenta il nome del file, si occupa di effettuare la lettura del file andando a eseguire sulla command line i comandi inseriti dall'utente. Il file deve trovarsi nella directory in cui sono presenti gli eseguibili del linguaggio di programmazione; dunque per nostra semplicità è sufficiente passare il nome del file (che si deve trovare dove si trovano gli eseguibili Bison).

Sintassi: readFile “nomeFile”

I passi sono:

- a. Creazione di un file in cui andare a scrivere il nostro programma:



- b. Scrivere su linea di comando: readFile “nomeFile”

```
> readFile "file"
Sto leggendo il file

> Dispositivo inserito con successo con ID: d1#949
Operazione di inserimento dispositivo completata con successo

> Ricerca del dispositivo d1 in corso...
Dispositivo Esistente
Richiesta connessione...

> numero:1, stringa:d2

AVVIO OPERAZIONE:
Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo d2 in corso...
Dispositivo d2#950: Non Esistente
[                               Dispositivo inesistente e quindi non acceso

> Dispositivo inserito con successo con ID: d3#951
Operazione di inserimento dispositivo completata con successo

>
>   3

> █
```

Come si nota sono stati eseguiti i comandi riportati all'interno del file come se l'utente li avesse scritti su linea di comando. Terminata la lettura tutti i dati saranno memorizzati nella workstation attuale del programma e quindi potranno essere utilizzati

- c. La terminazione del file è riconosciuta dall'EOF.

Finora sono state analizzate tutte le funzionalità che fanno riferimento a FUNCDEV. Le funzionalità che si riferiscono a FUN sono:

- **Void print(literal/Variabile):**

Nel programma è stata prevista una funzione analoga alla printf con lo scopo di stampare: variabili, costanti literal (come stringhe e numeri) e valori di ritorno delle funzioni. Nel caso in cui si cerchi di stampare informazioni non relative a stringhe e numeri, il programma ritornerà un errore. Tale scelta è stata fatta perché i tipi previsti nel linguaggio (Number, String) le uniche informazioni utili da stampare a video possono essere stringhe e numeri. Infatti, in un linguaggio di alto livello come il nostro, non ha senso (dato che deve essere utilizzato, anche, da chi non sa programmare) stampare indirizzi di memoria o altri dettagli di basso livello. È una funzione di comodo usata dall'utente per funzioni di debug, etc.

Di seguito riportiamo esempi di utilizzo:

```
> pluto = 7
> paperino = "disney"
> nocerina = connect "pippo"
Ricerca del dispositivo pippo in corso...
Dispositivo pippo#5694: Non Esistente
> print "pluto"
pluto
> print pluto
7
> print 7
7
> print paperino
disney
> print nocerina
ERROR: Il parametro passato alla funzione printf non è né un numero né una stringa
```

- **Void help()**

Anche questa è una funzione di comodo utile per descrivere le funzionalità del linguaggio:

```
> help
```

```
+-----+  
|  
|      ** Manuale di istruzione **  
|  
+-----+
```

---

Elenco comandi base con esempi :

---

| Inserimento Stringa : |

```
-> "Testo Stringa" [INVIO]
```

- newString return struct ast \* pointerSimbolo

- Esempio: 'Sono una Stringa'

| Stampa Stringa : |

```
-> sintassi: print "Stringa" [ INVIO ]
```

- callbuiltin return char \* pointerSymbol

- Note: Anche l'inserimento dell'istruzione sopra produce la stampa.

- Esempio: print "Ciao sono una Stringa" [ INVIO ]

| Inserimento Semplice Nuovo Dispositivo : |

```
-> sintassi: newDev "Nome Device" [INVIO]
```

- newDev return struct ast \* pointerDevice

- Esempio: newDevice "dev1" [ INVIO ]

| Inserimento Nuovo Dispositivo con Collegamenti ad altri Dispositivi : |

```
-> sintassi: newDev "Nome Device" → [ "Nome Device 1", "Nome Device 2", ..] [ INVIO ]
```

- newDev return struct ast \* pointerDevice

- **Void \* clear():**

È una funzione di comodo che effettua il clear dell'interprete avviato.



- Le funzionalità degli array, che sono state precedentemente descritte e quindi si evita di riportare: add, remove, get e set
- Bye:

```
> 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

> Display: EndDBLoading

> Display: LoadingLibrary

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

> print "ciao"
Display: ciao

> print ciao
ERROR: Il parametro passato alla funzione printf non è né un numero né una stringa

> bye
input in flex scanner failed
mario@DESKTOP-0U6L0FJ:/mnt/c/4_Linux/funzionante$
```

```
> connect "deviceOvest"
Ricerca del dispositivo deviceOvest in corso...
Dispositivo deviceOvest#451: Non Esistente

> bye
input in flex scanner failed
(base) MBP-di-Vincenzo:dino copia vincenzolabua$
```

Comando di comodo per consentire all'utente di uscire dall'interprete.

### Funzioni annidate

Come spiegato in precedenza una delle caratteristiche principali del nostro parser è quella di consentire di effettuare delle callback. Cioè dare la possibilità all'utente di eseguire diverse funzioni annidate. L'utilità di tale operazione è rendere il linguaggio più semplice e immediato. La sintassi in generale è:

Funzione1 Funzione2 ..... FunzioneN parametroDaPassareAllaFunzioneN

Le funzioni verranno eseguite dalla funzione N fino alla prima funzione. Si mostrano degli esempi per rendere il tutto più chiaro:

```
> archive switchOff connect "ciccio"
Ricerca del dispositivo ciccio in corso...
Dispositivo Esistente
Richiesta connessione...
                                Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo ciccio in corso...
Dispositivo Esistente
Richiesta connessione...
                                Il dispositivo è spento
```

```
> connect "ciccio"
Ricerca del dispositivo ciccio in corso...
Dispositivo ciccio#6132: Non Esistente
```

La funzione delete switchOff connect “ciccio” è di fatto analoga ad eseguire 3 istruzioni diverse del tipo:

```
> connect "ciccio"
Ricerca del dispositivo ciccio in corso...
Dispositivo Esistente
Richiesta connessione...

> switchOff "ciccio"
                                Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo ciccio in corso...
Dispositivo Esistente
Richiesta connessione...
                                Il dispositivo è spento

> archive "ciccio"
```

## Operatori condizionali

Concluse le funzioni di sistema, analizziamo gli operatori condizionali previsti nel nostro linguaggio. Come accennato in precedenza tutte le funzioni embedded definite nel nostro linguaggio ritornano un NULL o il nome del device che è stato connesso/acceso/ecc. Nel caso in cui la funzioni embedded ritorni NULL, allora non verrà eseguito il corpo dell’operatore condizionale, altrimenti si.

Riportiamo degli esempi:

1. **If** espressione1;espressione2;.....;espressioneN **then**  
espressione1;espressione2;.....;espressioneN;
  - a. L'espressione può essere per esempio una funzione embedded.
    - i. Se il device non esiste allora la connessione non andrà a buon fine, e quindi l'operazione di cancellazione non verrà effettuata.

```

device_Ovest#8100
device_Nord#5304
device_Sud#8477
device_Ovest#8100


.....


pippo#5694
pluto#3842
ciao#3924
ciao#3924
dino#1276
ciccio#6132

```

- ii. Se invece il device esiste la connessione andrà a buon fine e verrà cancellato il device

```

device_Ovest#8100
device_Nord#5304
device_Sud#8477
device_Ovest#8100


.....


pippo#5694
pluto#3842
ciao#3924
ciao#3924
dino#1276
ciccio#6132
calimero#4578

```

Dispositivo inserito con successo con ID: calimero#4578  
 Operazione di inserimento dispositivo completata con successo

```

> if connect "calimero" then archive "calimero" ;
Ricerca del dispositivo calimero in corso...
Dispositivo Esistente
Richiesta connessione...

>

```

2. **If** espressione1;espressione2;.....;espressioneN; **then**  
 espressione1;espressione2;.....;espressioneN ; **else**  
 espressione1;espressione2;.....;espressioneN ;

- a. Ad esempio, se la connessione col device è andata a buon fine (e quindi si è stabilito un collegamento con il dispositivo) allora verrà attivato per un certo intervallo di tempo. Altrimenti se non è stato raggiunto verrà creata un'istanza di quell' oggetto:

```

> if connect "gatto" then interval "gatto"-2 ; else newDevice "gatto" ;
Dispositivo inserito con successo con ID: gatto#7633
Ricerca del dispositivo gatto#7633 in corso...
Dispositivo gatto#7633#9243: Non Esistente
Operazione di inserimento dispositivo completata con successo

> if connect "gatto" then interval "gatto"-2 ; else newDevice "gatto" ;
Dispositivo già Esistente con ID: gatto#7633
Ricerca del dispositivo gatto#7633 in corso...
Dispositivo gatto#7633#9243: Non Esistente
Operazione di inserimento dispositivo completata con successo

```

3. Non si riportano casi d'uso specifici ma il do-while funzionano analogamente. Stando ovviamente attenti al fatto che non avendo implementato per il linguaggio operatori di comparazione e matematici (perché considerati superflui per l'ambito) che la condizione nel corpo del while prima o poi dovrà essere diversa da NULL.

## 5.2 Comandi/funzioni definite dall'utente:

Può essere utile realizzare comandi utili a semplificare la vita dell'utente. Finora nel parser e lexer non sono state descritte nella produzione exec il caso in cui ciò che si esegua sia una funzione.

Caratteristiche funzioni:

- Le variabili definite all'interno di una funzione sono globali;
- Le variabili definite all'esterno della funzione esistono;
- Le funzioni tramite il token ret definiscono il valore di ritorno;
- Se non viene specificato il token ret la funzione non restituisce nulla;
- La comunicazione tra la funzione chiamante e la funzione chiamata avviene tramite il passaggio dei parametri e i valori di ritorno.

Definizione funzione:

### **Senza valore di ritorno:**

Function: exec CMD NAME '(' argsList ')' '=' listStmt EOL

- CMD è un token che identifica l'espressione CMD.
- NAME è invece il token per identificare una qualsiasi variabile.
- ArgList identifica la lista di parametri da passare all'utente ed è una ulteriore espressione del tipo:
  - argsList: NAME | NAME ',' argsList
    - NAME identifica una qualsiasi variabile del programma. Quindi prenderà come parametri una lista di stringhe. NAME infatti è il token che identifica una qualsiasi stringa.
    - Esempio : var1, var2, var3
- ListStmt infine è una ulteriore espressione che identifica di fatto la lista di espressioni o funzioni embedded da eseguire all'interno della funzione. Potranno essere connect, print, newDevice.
- listStmt: /\* nothing \*/ | stmt ';' listStmt
  - Es: if connect var1 then delete var1 ; newDevice var2; pippo = “ciao” ;

Mettendo insieme le tre espressioni la definizione di una funzione può essere del tipo:

> CMD avviaDevice (var1, var2, var3) = if connect var1 then delete var1 ; newDevice var2; pippo = “ciao” ;

Esecuzione della funzione:

Per l'esecuzione della funzione verrà valutata la seguente produzione:

Exp: NAME '(' explistStmt ')'

- NAME è un token per identificare una qualsiasi variabile. Nel caso specifico consente di individuare il nome della funzione che stiamo richiamando
  - Esempio: avviaDevice
- Dopo tra parantesi viene individuata la produzione:
  - explistStmt: exp | exp ',' explistStmt
    - Di fatto identifica i parametri della funzione che potranno essere una o più espressioni. Per esempio, variabili, ma anche funzioni embedded.
    - Esempi:
      - (ciccio, pluto, paperino)
      - (“ciao”, 1, 2)
      - (dino, 1, 2)

- (connect “ciao”, var = 2, newDevice “ciccio”)

Il richiamo di funzione sarà dunque del tipo:

```
> avviaDevice (“ciao”, “come”, “va”);
```

### Esempio pratico:

#### 1. Passaggio parametri e variabili globali:

È stata definita la funzione “nomeFunzione” che prende come parametro v1 e lo stampa a video. La funzione quindi verrà eseguita e stamperà a video il parametro passato, che nel caso dell’esempio sarà 9:

```
> CMD nomeFunzione (v1) = print v1;
User Mode:      funzione definita
> nomeFunzione (9)
WIN:-1
Display:      9
WIN:-1
```

Se si provasse a stampare fuori la funzione la variabile v1 (variabile interna alla funzione nomeFunzione) il valore restituito sarebbe inesistente.

```
> CMD nomeFunzione (v1) = print v1;
User Mode:      funzione definita
> nomeFunzione (9)
WIN:-1
Display:      9
WIN:-1

> print v1
ERROR: Il parametro passato alla funzione printf non è né un numero né una stringa
```

La funzione può ovviamente essere richiamata passando dei paramenti diversi (8):

```
> nomeFunzione (8)
WIN:-1
Display:      8
WIN:-1

> print v1
ERROR: Il parametro passato alla funzione printf non è né un numero né una stringa
```

Da notare che, nel momento in cui vengono create col token CMD, le funzioni prendono come parametro un argList,

```
//Struttura momentanea usata:
// nelle funzioni per contenere la lista dei nomi delle variabili nel momento in cui la funzione viene definita
// nella creazione di un oggetto device per 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;
}
```

```
/* Struttura di collegamento tra un simbolo della tabella e una lista di simboli */
struct argsList {
    struct symbol *sym;
    struct argsList *next;
};
```

ossia una lista di NAME (cioè una lista di variabili). Nel momento, invece in cui, la funzione viene chiamata come parametri (da passare) può prendere una qualsiasi espressione (che non sia quindi per forza la singola variabile). Il risultato di tale espressione viene memorizzata nella variabile definita come parametro formale:

```
>CMD nomeFunzione (var1) ....  
> nomeFunzione (connect "ciao")
```

Il valore di connect ciao viene memorizzato nella variabile var1 nel momento in cui viene richiamata. CMD pretende come parametri delle funzioni variabili per forza (var1). Nel momento in cui la richiamo posso anche passare espressioni.

Una cosa fondamentale è che, per rendere il linguaggio il più semplice possibile, l'utente si interesserà solamente a tipi stringa e numero. Ciò vuol dire che tutti i parametri passati alle funzioni potranno essere tipi stringa e numero. Per cui non è possibile passare interi array, ma solo singoli valori dell'array. La scelta è stata improntata a rendere il linguaggio il più semplice possibile facendo restituire di fatto a tutte le funzioni embedded e di sistema una stringa

### Con valore di ritorno:

Function: CMD NAME '(' argsList ')' '=' listStmt RET exp ';'

Analoga alla produzione delle funzioni senza parametri, con la differenza che è presente il token RET che rappresenta l'espressione "ret". Una volta infatti specificati

- il nome della funzione (NAME): nomeFunzione
- la lista degli argomenti (argList): v1
- la lista delle istruzioni da eseguire (listStmt): print v1; v3=8;
- Una volta specificata la lista delle espressioni (corpo dell'espressione) viene specificato il token RET: ret
- Affiancata a ret vi è il valore di ritorno della funzione che può essere una qualsiasi espressione:
  - Una funzione embedded
  - Una variabile
  - Una funzione definita dall'utente prima definita ecc.

La regola può quindi essere soddisfatta da un'espressione del tipo:

CMD nomeFunzione (v1) = print v1; v3=8; ret v3;

### Esempio pratico:

Nel caso viene creata una funzione nomeFunzione che stampa a video il parametro preso come parametro e definisce una variabile locale (v3) che ritorna alla funzione chiamata. Nel caso viene:

- Definita la funzione: CMD nomeFunzione (v1) = print v1; v3=8; ret v3;
- Richiamata: nomeFunzione (1)
  - Una volta richiamata ritorna il seguente valore:
    - Display: 1 -> cioè stampa a video il valore passato come parametro, che è 1

- Una volta eseguita la funzione la variabile ritornata è v3. V3 sarà quindi esistente anche nella funzione che subisce la chiamata. Se infatti provi a stampare il valore di v3 ritornerà 8 (cioè il valore ritornato dalla funzione):
  - Print v3
    - > display: 8

```
> CMD nomeFunzione (v1) = print v1; v3=8; ret v3
User Mode:      funzione definita
> nomeFunzione (1)
WIN:-1
Display:      1
WIN:-1

> print v3
Display:      8

> nomeFunzione (2)
WIN:-1
Display:      2
WIN:-1

> print v1
Display:      5

> clear
```

### Uso degli array nelle funzioni:

Nella sezione inerente al passaggio dei parametri è stato sottolineato come i parametri passati possano essere solo stringhe e numeri. Tra le funzioni degli array che abbiamo visto nella sezione apposita avevamo detto che tutte le funzioni servissero a modificare internamente l'array. Per esempio, la funzione add non restituiva nulla ma semplicemente modificava l'area di memoria in cui è allocato l'array. Analogamente le funzioni remove e set modificano internamente quell'area di memoria. L'unica funzione che ci consente di accedere all'array è get. La funzione get può ricevere un parametro, rappresentato dall'indice della posizione dell'array che si vuole restituito. Se si effettua:

```
> newArray integer ciao (2);
> ciao->add = 2
> ciao->get = 0
```

La funzione restituisce l'elemento in posizione 0, cioè due. Questo intero può essere quindi passato alla funzione. Per cui non è stato previsto il passaggio dell'intero array alla funzione ma dei singoli parametri. Eventualmente l'utente può passare tanti parametri quanto è la dimensione dell'array, ognuno dei quali contenente il valore i-esimo dell'array:

Intanto nell'esempio sotto viene mostrato un caso in cui all'interno di una funzione può essere creato un array:

```

> CMD defArray (v1) = print v1; newArray char maria (2); maria->add = "virgoal" ; maria->add = "gattino"; maria->get;
User Mode:      funzione definita
> defArray ("ciccio")
WIN:-1
Display: ciccio
ARRAY CREATO CORRETTAMENTE:maria#7190#6970
virgoal
gattino
WIN:-1

> newArray integer maria (2);
138: Errore: syntax error

> newArray integer maria (2)
ARRAY CREATO CORRETTAMENTE:maria#7190#6970

> maria -> get

> maria-> add = "dsandsa"

> maria->get

> maria->add = 2

> maria-> get
2

```

#### Uso delle funzioni embedded nelle funzioni definite dall'utente:

Si ha la possibilità di richiamare all'interno delle funzioni dell'utente tutte le funzioni embedded che sono state spiegate nei paragrafi precedenti: connect, reconnect, switchOn, print, ecc. Si ha anche la possibilità di creare all'interno della funzione nuove istanze device che esisteranno all'interno della funzione. Un esempio può quindi essere il seguente:

- Viene di fatto creata una prima funzione giardino (che non ha un significato specifico ma serve a far capire come richiamare le funzioni embedded all'interno di una funzione):
  - Crea una funzione giardino che prende due parametri di input.
  - Questa funzione non fa altro che:
    - Verificare se il dispositivo “ciccio” è raggiungibile-> non esiste un'istanza dell'oggetto device che per cui viene considerato non raggiungibile
    - Il dispositivo non esiste e quindi viene creato (newDevice) una nuova istanza di tale oggetto
    - E verificare se l'oggetto raggiungibile questa volta con una nuova connect

I device istanziati all'interno della funzione non esisteranno all'esterno della funzione.

```
> CMD giardino (v1, v2) = connect "ciccio"; newDevice "ciccio"; connect "ciccio";
User Mode:      Dispositivo già Esistente con ID: ciccio#6132
funzione definita
> newDevice "ciccio"
Dispositivo già Esistente con ID: ciccio#6132
Operazione di inserimento dispositivo completata con successo

> giardino (1,2)
WIN:-1
Ricerca del dispositivo ciccio in corso...
Dispositivo ciccio#6132: Non Esistente
Dispositivo inserito con successo con ID: ciccio#6132
Ricerca del dispositivo ciccio in corso...
Dispositivo Esistente
Richiesta connessione...
WTN--1
```

- La funzione Giardino con un po' più senso può essere la seguente:
  - Verificare la connessione del dispositivo con una connect
  - Creare un'istanza dell'oggetto device e vedere se è raggiungibile (newDevice e connect)
  - Una volta connesso eliminiamo/archiviamo l'oggetto device creato (archive)
  - E infine verificare che il dispositivo non sia più raggiungibile

```
> CMD giardino (v1) = connect v1; newDevice "ciccio"; connect "ciccio"; archive "ciccio"; connect "ciccio";
User Mode:      Dispositivo già Esistente con ID: ciccio#6132
funzione definita
> giardino ("ciccio")
WIN:-1
Ricerca del dispositivo ciccio in corso...
Dispositivo ciccio#6132: Non Esistente
Dispositivo inserito con successo con ID: ciccio#6132
Ricerca del dispositivo ciccio in corso...
Dispositivo Esistente
Richiesta connessione...
Ricerca del dispositivo ciccio in corso...
Dispositivo ciccio#6132: Non Esistente
```

#### Funzioni annidate:

Viene anche data la possibilità all'utente di ridefinire funzioni all'interno di altre. Le funzioni devono essere tutte definite all'interno dell'interprete principale, per poi eventualmente essere chiamate. Nel momento in cui una funzione viene richiamata viene eseguita l'ultima definizione che ne è stata data. Se quindi la funzione pippo è definita più volte quando verrà richiamata eseguirà l'ultima CMD che è stata definita.

Nel caso seguente all'interno della funzione f viene richiamata la funzione h.

```

> CMD f (v1) = print v1; h(9); print v1;
User Mode:      funzione definita
> CMD h (v1) = print v1;
User Mode:      funzione definita
> h (9)
WIN:-1
Display:    9
WIN:-1

> f (1)
WIN:-1
Display:    1
WIN:0
Display:    9
WIN:0
Display:    1
WIN:-1

> print v1
ERROR: Il parametro passato alla funzione printf non è né un numero né una stringa
Un analogo esempio al precedente ma con la differenza che alla funzione h viene passata
non una costante (9, nel caso sopra) ma una variabile locale alla funzione f. È ovvio che le
variabili locali di F non saranno definite anche in h, e viceversa.

> CMD f(v1) = print v1; h(v1); print v1;
User Mode:      funzione definita
> CMD h(v1)= print v1;
User Mode:      funzione definita
> f(5)
WIN:-1
Display:    5
WIN:0
Display:    5
WIN:0
Display:    5
WIN:-1

```

#### Esempio di utilizzo delle funzioni di sistema:

L'utente può definire funzioni di comodo per effettuare le operazioni di giardinaggio. Nel caso seguente si propone una funzione annaffia che:

- Genera il device dev0 come istanza
- Verifica se il ping va a buon fine
  - Se va a buon fine all'ora attiva il device per un certo intervallo di tempo (4 secondi nel caso)
  - Altrimenti prova a riconnettersi al device e ad attivarlo per avviarlo per un certo intervallo di tempo
- Una volta conclusa l'operazione di irrigazione archivia/elimina il device (delete/archive v1)
- Verifica che non esiste più un'istanza tramite la connect
- E stampa finish

```
> CMD annaffia (v1) = newDevice "dev0"; if connect v1 then interval v1-4; else reconnect v1; interval v1-4;; archive v1; connect v1 ; print "FINISH";
User Mode:      Dispositivo già Esistente con ID: dev0#9783
funzione definita
> annaffia ("dev0")
WIN:-1
Dispositivo inserito con successo con ID: dev0#9783
Ricerca del dispositivo dev0 in corso...
Dispositivo Esistente
Richiesta connessione...
numero:4, stringa:dev0

AVVIO OPERAZIONE:
                    Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo dev0 in corso...
Dispositivo Esistente
Richiesta connessione...
ino
                    La connect è andata a buon fine e il dispositivo è stato acceso

                    Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo dev0 in corso...
Dispositivo Esistente
Richiesta connessione...
ino
                    Il dispositivo è spento

Ricerca del dispositivo dev0 in corso...
Dispositivo dev0#9783: Non Esistente
Display: FINISH
WIN:-1
```

Un altro modo di vedere la funzione di annaffia è analizzarla non con un singolo device ma come tanti device che è necessario attivare:

```

> CMD irriga (v1,v2) = newDevice "dev0"; switchOff connect v1; switchOn connect v1; newDevice "dev1"->["dev0"]; interval "dev1"-5; interval "dev2"-5; archive "dev1"; archive "dev0";
User Mode: Dispositivo già Esistente con ID: dev0#9783
Dispositivo già Esistente con ID: dev1#9782
funzione definita
> irriga (1,2)
WIN:-1
Dispositivo inserito con successo con ID: dev0#9783
Ricerca del dispositivo 1 in corso...
Dispositivo 1#1009: Non Esistente
Verifica in corso della connessione del dispositivo...
Ricerca del dispositivo 0 in corso...
Dispositivo 0#48: Non Esistente
Dispositivo inesistente

Ricerca del dispositivo 1 in corso...
Dispositivo 1#1009: Non Esistente
iniziasi switchOn
Verifica in corso della connessione del dispositivo...
Ricerca del dispositivo 0 in corso...
Dispositivo 0#48: Non Esistente
Dispositivo inesistente e quindi non acceso

Dispositivo inserito con successo con ID: dev1#9782
numero:5, stringa:dev1

AVVIO OPERAZIONE:
Verifica in corso della connessione del dispositivo...
Ricerca del dispositivo dev1 in corso...
Dispositivo Esistente
Richiesta connessione...
ino
La connect è andata a buon fine e il dispositivo è stato acceso

Verifica in corso della connessione del dispositivo...
Ricerca del dispositivo dev1 in corso...
Dispositivo Esistente
Richiesta connessione...
ino
Il dispositivo è spento

numero:5, stringa:dev2

AVVIO OPERAZIONE:
Verifica in corso della connessione del dispositivo...
Ricerca del dispositivo dev2 in corso...
Dispositivo dev2#9781: Non Esistente
Dispositivo inesistente e quindi non acceso

WIN:-1

```

### NewDevice e funzioni:

Quando all'interno di una funzione viene richiamata la funzione di creazione di un'istanza di device questa viene anche creata nella funzione principale rappresentata dell'interprete.

Esempio:

```
> CMD f (v) = newDevice "ciccio";
> connect "ciccio"
```

Connect andata a buon fine

Il device esiste nel momento in cui creo la funzione anche per la funzione principale. La funzione newDevice non prende come parametro un'espressione, ma una stringa, per cui non è possibile passargli parametri della funzione.

### Riepilogo funzioni:

- Prendono come parametro una qualsiasi espressione: tale espressione è rappresentata da una stringa o un numero. Quindi le funzioni prenderanno come

- parametri solo i tipi primitivi stringa e intero. Scelta fatta per consentire l'annidamento delle funzioni e per semplificare la vita dell'utente evitando possibile passaggio di riferimenti con array;
- Restituiscono o NULL o una stringa o un numero (usando il token RET);
  - Se la funzione è ridefinita al primo richiamo verrà eseguita l'ultima definizione;
  - Degli array puoi passare singoli elementi dell'array, non l'intero riferimento.

## **Descrizione del linguaggio:**

### **Analisi del codice**

#### **GESTIONE DEVICE:**

Iniziamo ad analizzare le built-in function.

### **ALTO LIVELLO:**

Per la definizione di funzioni da parte dell'utente, la produzione utilizzata nel parser è:

- exec CMD NAME '(' argsList ')' '=' listStmt EOL

#### **Analisi delle produzioni ad alto livello:**

In fase di avvio quindi per creare un device l'utente dovrà digitare la seguente stringa di caratteri:

1. Il simbolo CMD nel lexer identifica semplicemente una stampa a video per indicare la modalità con cui sta gestendo le operazioni l'utente:

"CMD" { printf("User Mode:\t"); }

2. L'utente una volta specificato il CMD d'uso inserirà il nome della funzione su cui bisogna effettuare la operazione andando a immagazzinare tale NAME in una struct symbol (si noti per il momento ad alto livello che la funzione search si occupa semplicemente di ritornare un simbolo):

[a-zA-Z][a-zA-Z0-9]\*

{

l'azione associata cre un simbolo nella tabella dei simboli se non esiste, altrimenti ne ritorna il puntatore al simbolo presente nella stessa

}

3. Una volta definito il nome della funzione su cui si vuole effettuare una certa operazione l'utente dovrà inserire la lista di parametri associati alla funzione. Lo specificherà tra parentesi tonde.

'(' argsList ')'

Per il momento di argList basti sapere che è un ulteriore produzione che va a creare una lista concatenata dei parametri della funzione.

4. L'utente potrà eventualmente decidere una volta specificato il nome della funzione e la lista dei parametri, il tipo di operazioni che vorrà eseguire. Dopo che l'utente

inserisce l'uguale può definire, quindi, il tipo di operazione da effettuare col device.

Questa operazione potrà riguardare:

- a. Degli stati condizionali (if, else, ecc)
  - b. Funzioni di sistema
  - c. Funzioni runtime
  - d. Inserimento di device
  - e. Collegamento di Device
5. Una volta scelta il tipo di operazione questa verrà effettuata e quindi conclusa.
  6. Il comando terminerà con l'invio a capo.

## **FUNZIONI DA ESEGUIRE:**

Una volta che l'utente ha inserito già eseguito i primi 3 passi della produzione precedente, ipotizzando che siano i seguenti:

**CMD Device\_H20(Device\_diottrico) =**

Da questo passo in poi siamo al punto 4. della precedente descrizione e l'utente potrà scegliere di eseguire una delle seguenti funzioni:

## **CREAZIONE DEL PRIMO DEVICE:**

Per l'inserimento del device la produzione da eseguire è la seguente:

**INSERT STRING {**

```
defSymRef($2, NULL, NULL);
$$ = newDev($2,NULL);
}
```

Una volta che l'utente dall'1 al 3 della produzione principale nella fase 4 sceglierà di eseguire la produzione identificata dai token **INSERT STRING**:

1. L'utente ha semplicemente inserito sul prompt la word “newDevice” per identificare che tale funzione ha l'obiettivo di creare un device.

**"newDevice" { yyval.func = B\_insertDevice; return INSERT; }**

2. L'utente quindi definirà il nome del device che vuole creare. Tale stringa viene identificata nel flexer con il seguente token:

**["] [a-zA-Z][a-zA-Z0-9]\*["]**

3. La situazione è quindi la seguente:

**CMD Device\_H20(Device\_diottrico) = newDevice Device\_nuovo\_cata**

4. Quello definito è quindi un primo comando completo eseguito dal nostro programma che di fatto esegue la seguente operazione:

a. Tramite la funzione defSymRef

```
//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); }
    //if(name->func) treefree(name->func);

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

b. crea un nuovo simbolo della struct symbol.

```
void defSymRef(struct symbol *name, struct argsList *syms, struct ast *func)
```

c. Viene quindi chiamata la funzione newDev che si occupa effettivamente della creazione del nuovo Device. Viene quindi creando un oggetto device con le seguenti caratteristiche.

```
d->nodetype = 'D';
```

```
d->status = 0; //VIENE POSTO NELLO STATO SPENTO DI DEFAULT
```

```
d->s= sym;
```

```
d->l = l;
```

È stato quindi formato un nodo del nostro albero AST su cui poi effettueremo delle operazioni.

Analoghi comandi possono essere usati per creare device indipendenti. Ipotizziamo invece di creare un secondo device che concettualmente dovrà essere collegato a questo

### **CREAZIONE DEL SECONDO DEVICE:**

La produzione che verrà utilizzata è la seguente:

```
INSERT STRING ARROW '[' argsListDevice ']' {
```

```
    defSymRef($2, $5, NULL);
```

```
    $$ = newDev($2,$5);
```

```
}
```

COSTRUISCE LA LISTA DI PUNTATORI AI SIMBOLI CIOÈ AI DEVICE COLLEGATI AL DEVICE CHE SI STA INSERENDO.

Il device creato non viene quindi al momento collegato con gli altri ma semplicemente memorizziamo in memoria tale informazione in una lista.

Partendo sempre dalla produzione di base **CMD Device\_H20(Device\_diottrico) =**

L'utente digiterà:

1. Il nuovo device da creare esattamente come nella fase di creazione del primo device:  
**CMD Device\_H20(Device\_diottrico)= newDevice device\_Sud**
2. ARROW identifica il seguente token: "->" { return ARROW;} che identifica quindi:  
**CMD Device\_H20(Device\_diottrico)= newDevice device\_Sud->**
3. L'utente in questa operazione potrà inoltre definire la lista dei device a cui ha intenzione di collegarsi tramite la produzione argListDevice:

```
argsListDevice: STRING { $$ = newargsList($1, NULL); }
```

---

```
| STRING '' argsListDevice { $$ = newargsList($1, $3); }
```

```
;
```

Tale produzione non fa altro che ritornare una struttura in cui inserisce una lista il device\_nuovo\_cata creato nel passo precedente.

Ipotizzato che l'utente quindi abbia digitato il comando:

**CMD Device\_H20(Device\_diottrico) = newDevice Device\_sud -> [Device\_nuovo\_cata]**

L'utente, quindi, con la produzione sopra scritta avvierà le funzioni necessarie per creare il device Device\_sud:

- Con defSymRef

```
//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)
{
    if(name->syms){ argsListfree(name->syms); }
    if(name->func) treefree(name->func);

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

va a memorizzare in memoria la lista dei device a cui si collegherà

- Con newDev va effettivamente a creare il nuovo device

Alla fine delle due operazioni sopra citate saranno stati creati due nodi al momento indipendenti dell'albero:

## BASSO LIVELLO

Fatta una panoramica delle funzioni ad alto livello possiamo preoccuparci di definire le funzioni più a basso livello.

### STRUTTURE:

#### Tabella dei simboli:

```
#define DIMHASH 10000
struct symbol symtab[DIMHASH];
```

Nel programma un ruolo fondamentale lo gioca la tabella dei simboli. La tabella dei simboli è una struttura dati (struct symbol) in cui ogni entry è un identificativo a cui vengono associate informazioni legate alla sua dichiarazione. La tabella dei simboli avrà come indice un codice HASH concatenato col nome del simbolo che farà da identificativo/indice per il simbolo a cui si vorrà accedere.

Le strutture principali usate sono definite nel file serra.h e sono le seguenti:

```
struct symbol {
    char *name;
    char *value;
    struct ast *func; /* stmt per le funzioni */
```

```

    struct ast *dev;      /* stmt per le funzioni */
    struct argsList *syms; /* Lista dei simboli */

};

```

Ogni simbolo conterrà 4 campi: nome, valore, funzione da applicare, device a cui collegarsi e lista dei simboli associati. Inizialmente la tabella sarà ovviamente vuota.

<b>Tabella Simboli</b>							
Name	Value	Function	Device	List	Dimensone	Syms	Valore di ritorno

### SymRef:

```

struct symref {
    int nodetype;      /* tipo nodo N -> riferimento ad un simbolo*/
    struct symbol *s;
};


```

È una struttura che identifica un riferimento a un simbolo della tabella tramite il secondo campo ‘s’. Il primo campo identifica il tipo di nodo dell’albero che lo identifica (nel caso N)

### Device:

Ogni volta che verrà creato un device verrà memorizzata in memoria e nella tabella dei simboli un tipo struct device:

```

struct device {
    int nodetype; /* tipo nodo D -> dispositivo inserito nella rete */
    int status;    /* definisce lo stato, acceso 1, spento 0 */
    struct symbol *s;
    struct ast *l; /* Lista dei device collegati */
};


```

Tipicamente verrà definito da 3 parametri:

- NodeType: identifica il tipo di Nodo e nel nostro programma verrà quasi sempre settato come un tipo ‘D’ (device)
- Lo status è un intero che identifica se il device è in stato acceso (1) oppure spento (0)
- Symbol: è il riferimento in memoria di questo device alla struttura symbol che sarà stata creata in precedenza.
- L’ultimo parametro è fondamentale e identifica i nodi dell’albero a cui questo device è collegato. Nel caso sono rappresentati da ulteriori device.

### Stringhe:

```

struct stringVal {
    int nodetype; /* tipo nodo C -> valore stringa costante*/
    struct symbol *s;
};


```

---

```
};
```

È una struttura che viene richiamata nel momento in cui l'utente scrive una stringa su linea di comando. La stringa verrà inserita nella tabella dei simboli e gli verrà etichettato nella variabile nodetype il valore C nell'albero.

### Numeri:

```
struct numval {
    int nodetype;          /* tipo nodo K -> valore costante*/
    double number;
};
```

È una struttura analoga alla precedente solo che definisce i numeri e il nodo dell'albero verrà etichettato con K per indicare che è una costante.

### Collegamento tra simboli della tabella dei simboli:

```
/* Struttura di collegamento tra un simbolo della tabella e una lista di simboli */
struct argsList {
    struct symbol *sym;
    struct argsList *next;
};
```

### Struttura funzioni embedded:

```
/* Struttura Nodo per l'AST, per le funzioni predefinite */
struct funcBuiltIn {
    int nodetype;          /* tipo nodo F -> Funzioni Built-In*/
    struct ast *l;
    enum builtFunc functype;
};
```

Per la gestione delle funzioni embedded è stata creata una struttura specifica il cui tipo del nodo dell'albero sarà identificato da 'F'.

Inoltre, struct ast \*l identifica i parametri nodo che sono stati passati alla funzione. Il terzo campo della struttura identifica la funzione che verrà eseguita tramite un tipo enumerativo. Le funzioni che sono state pensate (e definite nel tipo enumerativo builFunc) sono le seguenti:

- B\_print = 1,
- B\_connect = 2,
- B\_reconnect = 3,
- B\_status = 4,
- B\_switchOn = 5,
- B\_switchOff = 6,
- B\_delete = 7,

- B\_interval = 8,
- B\_insertDevice = 9,
- B\_readFile = 10,
- B\_sleep=11,

### ***OPERAZIONI DI BASE, CENNI:***

Prima di descrivere le principali funzionalità si nota che all'interno di exp ci sono alcune funzionalità di base definite da alcuni token:

- *STRING:*

String è un token che sarà descritto più nel dettaglio in seguito con la descrizione delle funzioni. In generale è un token per identificare una qualsiasi stringa scritta su linea di comando dall'utente a seguito di una specifica operazione e/o funzione embedded. La sua caratteristica è quella di inserire (tramite la funzione search, spiegata in seguito) un symbol nella tabella dei simboli. La funzione che viene chiamata una volta riconosciuta una stringa dal parser è newString.

```
struct ast *newString(struct symbol *s)
{
    struct stringVal *a = malloc(sizeof(struct stringVal));

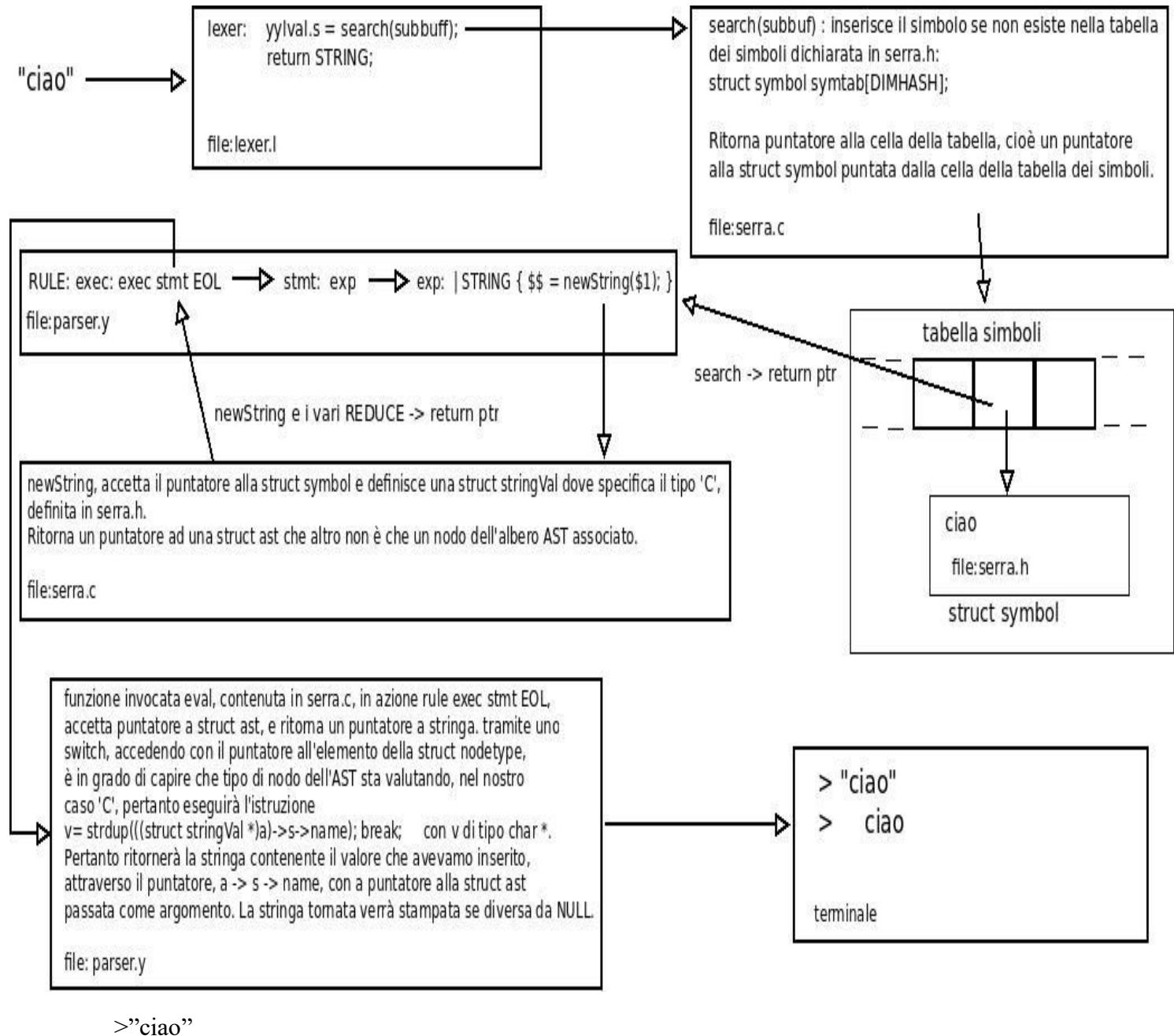
    if(!a) {
        yyerror("Spazio di memoria insufficiente\n");
        exit(0);
    }
    a->nodetype = 'C';
    a->s = s;

    return (struct ast *)a;
}
```

Con tale funzione viene creato quindi un oggetto stringVal

```
struct stringVal {
    int nodetype;          /* tipo nodo C -> valore stringa costante*/
    struct symbol *s;
};
```

associandogli come parametri il simbolo creato e ‘c’



&gt;"ciao"

Scrivendo tale comando sul prompt le operazioni causeranno il seguente risultato in memoria:

Tabella Simboli					
Indice (Hash+Name)	Name	Value	Function	Device	List
ciao20	ciao		0 NULL	NULL	NULL
Nodo AST					
StringVal	C	ciao20			

Le produzioni richiamate nel caso sono:

- A. Exec: exec stm EOL
- B. Stmt: exp
- C. Exp: STRING

- NUMBER:

Analoga alla struttura String solo con etichette nei nodi dell'albero diverse. Il risultato sarà del tipo:

> 123

<b>Tabella Simboli</b>					
Indice (Hash+Name)	Name	Value	Function	Device	List
ciao20	ciao		0 NULL	NULL	NULL
123a1		123	0 NULL	NULL	NULL
<b>Nodo AST</b>					
<b>StringVal</b>	C	ciao20			
<b>NumVal</b>	K	123a1			

- **SYSTEM**

System è un token ritornato per tutte le funzioni di sistema/embedded definite dall'utente. Un esempio di funzione embedded realizzata è rappresentato dalla funzione clear.

```
struct funcBuiltInSystem {
    int nodetype; /* tipo nodo O -> Funzioni Built-In System o senza argomenti */
    enum builtFuncSystem functype;
};
```

Le funzioni di sistema previste sono:

- - ```
enum builtFuncSystem {
    B_clear = 1,
    B_help = 2,
};
```
  - ```
case B_clear:
system("clear");
return NULL;
break;
```
  - ```
case B_help:
helpMessage();
return NULL;
break;
```

- **FUNC:**

FUNC è analogamente un token al quale sono associate le funzioni incorporate dal nostro sistema. Un esempio di funzione è la print. La funzione ovviamente pretenderà dei parametri definiti tramite la produzione explistStmt. Sarà spiegata nel dettaglio più avanti quanto parleremo in un paragrafo dedicato delle funzioni corporate.

- **FUNCDEV**

È un token che verrà richiamato tutte le volte in cui l'utente vorrà richiamare funzioni embedded legate ai device inseriti nel sistema di giardinaggio. Analogamente come le

funzioni incorporate può pretendere dei parametri che vengono passati tramite la produzione explistStmt (che si ricollega a exp potendo quindi avere stringhe, simboli ecc). La definizione di queste funzioni avviene nel file serra.c all'interno della funzione newfunc. La funzione newFunc ritorna un nodo dell'albero ast identificato dal nodetype 'F'

- NAME:

Identifica di fatto una lettera iniziale seguita da una stringa (sia lettere che numeri). Viene anche in questo caso (come con STRING) creato un symbol per la tabella dei simboli. E inoltre, invoca la funzione newRef

```
struct ast *newref(struct symbol *s)
{
    struct symref *a = malloc(sizeof(struct symref));

    if(!a) {
        yyerror("Spazio di memoria insufficiente\n");
        exit(0);
    }
    a->nodetype = 'N';
    a->s = s;
    return (struct ast *)a;
}
```

che dato il simbolo appena creato si occupa di creare un nodo dell'albero. Tale nodo sarà definito utilizzando la struttura funcBuiltIn:

```
/* Struttura Nodo per l'AST, per le funzioni predefinite */
struct funcBuiltIn {
    int nodetype;          /* tipo nodo F -> Funzioni Built-In*/
    struct ast *l;
    struct ast *r;
    struct ast *t;
    enum builtFunc functype;
};
```

- NodeType F: per indicare che si tratta di una funzione

- 

>>Pippo10

| <b>Tabella Simboli</b> |        |       |          |        |      |
|------------------------|--------|-------|----------|--------|------|
| Indice (Hash+Name)     | Name   | Value | Function | Device | List |
| ciao20                 | ciao   |       | 0 NULL   | NULL   | NULL |
| 123a1                  |        | 123   | 0 NULL   | NULL   | NULL |
| pippo1a2               | pippo1 |       | 0 NULL   | NULL   | NULL |

| <b>Nodo AST</b> |   |          |
|-----------------|---|----------|
| StringVal       | C | ciao20   |
| NumVal          | K | 123a1    |
| SymRef          | N | pippo1a2 |

## FUNZIONI NEL DETTAGLIO:

### COME LAVORARE COI DEVICE:

```
1. INSERT STRING {  
    defSymRef($2, NULL, NULL);  
  
    $$ = newDev($2,NULL);  
}
```

Tramite il token **INSERT** specificheremo che vorremo avviare la funzione `insertDevice` corrispondente alla decima funzione di sistema.

Tramite il token **STRING** l'utente definirà il nome del device da inserire. Nel token varrà definito un tipo symbol ritornato dalla funzione `search`.

La funzione **search** non fa altro che controllare se il simbolo passato da **STRING** dall'utente esiste già. Se il simbolo già esiste allora ti ritorna il puntatore alla entry della tabella dei simboli che contiene quel simbolo. Altrimenti viene creato con i seguenti parametri nella tabella dei simboli:

| Name              | Value | Function | Device | List |
|-------------------|-------|----------|--------|------|
| Device_nuovo_cata | 0     | NULL     | NULL   | NULL |
|                   |       |          |        |      |
|                   |       |          |        |      |
|                   |       |          |        |      |

Una volta che i token sono stati riconosciuti si avvierà il contenuto della produzione che prevede due funzioni: `defSymRef` e `newDev`.

#### La funzione `defSymRef`

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

prende tre parametri: la entry symbol che si è appena creata e una lista di device a cui dovranno collegarsi. Al momento ipotizziamo che i due ulteriori parametri siano `NULL`. Per cui la funzione nel caso lascia inalterata la entry di `Device_nuovo_cata` della tabella dei simboli. Quindi nel caso l'operazione risulta superflua dato che il device non deve al momento essere collegato ad altri device.

Infine, viene creato il nuovo device tramite la funzione `newDev`.

```
struct ast * newDev(struct symbol *ps, struct argsList *l)
```

Tale funzione prende 3 parametri in input rappresentati dal simbolo della tabella dei simboli da inserire e la lista dei device di riferimento a cui bisogna collegarsi. Non fa altro che creare una struct device con i seguenti parametri:

```
d->nodetype = 'D';      /*Tipo di nodo D, ossia Dispositivo*/  
d->status = 0;        //LO PONGO CON STATO SPENTO DI DEFAULT
```

d->s= sym;

d->l = l;

Il device sarà quindi collegato a un symbol sym della tabella dei simboli.

Lo schema può essere riassunto come segue:

Dunque, l'operazione di inserimento consiste nell'inserire nella tabella dei simboli un symbol che contiene nel campo Name il nome del device creato. Inoltre, viene creato un primo nodo dell'albero AST di tipo Device.

### Risultati:

- Creazione di un oggetto symbol nella tabella dei simboli
- Creazione di un oggetto Device che rappresenta un nodo possibile del nostro albero

| Tabella Simboli    |                |                  |             |        |      |
|--------------------|----------------|------------------|-------------|--------|------|
| Indice (Hash+Name) | Name           | Value            | Function    | Device | List |
| Device_nuovo_c1    | Device_nuovo_c |                  | 0 NULL      | NULL   | NULL |
|                    |                |                  |             |        |      |
|                    |                |                  |             |        |      |
| Oggetto Device     |                |                  |             |        |      |
| NodeType           | Status         | S (Symbol)       | I (argList) |        |      |
| D                  |                | 0 Devicenuovo_c1 | Null        |        |      |
|                    |                |                  |             |        |      |
|                    |                |                  |             |        |      |

Ipotizziamo quindi di ripetere più volte l'operazione e quindi la situazione in memoria è la seguente:

| Tabella Simboli    |                |                  |             |        |      |
|--------------------|----------------|------------------|-------------|--------|------|
| Indice (Hash+Name) | Name           | Value            | Function    | Device | List |
| Device_nuovo_c1    | Device_nuovo_c |                  | 0 NULL      | NULL   | NULL |
| Device_nuovo_d1    | Device_nuovo_d |                  | 0 NULL      | NULL   | NULL |
|                    |                |                  |             |        |      |
| Oggetto Device     |                |                  |             |        |      |
| NodeType           | Status         | S (Symbol)       | I (argList) |        |      |
| D                  |                | 0 Devicenuovo_c1 | Null        |        |      |
| D                  |                | 0 Devicenuovo_d1 | Null        |        |      |
|                    |                |                  |             |        |      |
|                    |                |                  |             |        |      |

Una volta creati degli ast/oggetti device eseguiamo la seconda operazione di collegamento tra i nodi. Ovviamente puoi specificare anche device già esistenti e collegarli ad altri. Ovviamente è necessaria la loro esistenza.

2. | INSERT STRING ARROW '[' argsListDevice ']'  
{  
    defSymRef(\$2, \$5, NULL);  
    \$\$ = newDev(\$2,\$5);  
}

Ipotizzando che il comando digitato dall'utente sia il seguente:

## CMD Device\_H20(Device\_diottrico) = newDevice Device\_A -> [Device\_nuovo\_C]

Con questa operazione non viene solo creato il device “Device\_A” ma anche collegato a ulteriori possibili device (“Device\_nuovo\_C”) creati in precedenza. Cioè il campo ‘I’ dell’oggetto Device riportato nella figura sopra verrà riempito con una lista di device a cui l’oggetto vorrà collegarsi. Le operazioni che vengono eseguite sono analoghe a prima:

- INSERT STRING (**newDevice Device\_A**): tramite queste due token verrà richiamata la funzione search il quale compito come detto in precedenza è quello di creare aggiungere un elemento nella tabella dei simboli che quindi diventa di questo tipo:

| <b>Tabella Simboli</b> |                |       |          |        |      |
|------------------------|----------------|-------|----------|--------|------|
| Indice (Hash+Name)     | Name           | Value | Function | Device | List |
| Device_nuovo_c1        | Device_nuovo_c |       | 0 NULL   | NULL   | NULL |
| Device_nuovo_d1        | Device_nuovo_d |       | 0 NULL   | NULL   | NULL |
| Device_A1              | Device_A       |       | 0 NULL   | NULL   | NULL |

- ARROW (**newDevice Device\_A ->**) è un token che specifica la ->

- La produzione argsListDevice è la seguente

argsListDevice:

```
STRING { $$ = newargsList($1, NULL); }
| STRING '' argsListDevice { $$ = newargsList($1, $3); }
;
```

Nel momento in cui l’utente inserisce la stringa **Device\_nuovo\_C** verrà riconosciuto dal solito token STRING, il quale andrà a ritornare la entry della tabella dei simboli che rappresenta il Device\_nuovo\_c (è rappresentato dall’indice Device\_nuovo\_c1). Verrà quindi chiamata la funzione newargsList

```
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;
    printf("ARGSLIST:%s\n", sym->name);
    sl->next = next;
    return sl;
}
```

che ritorna semplicemente una struttura argsList.

```
/* Struttura di collegamento tra un simbolo della tabella e una lista di simboli */
struct argsList {
    struct symbol *sym;
    struct argsList *next;
};
```

Tale struttura sarà una lista di tutti i nodi/Device inseriti tra parentesi quadre dall’utente.

- d. L'utente ha quindi avviato il comando: INSERT STRING ARROW '[' argsListDevice ]'
- STRING contiene il symbol nella tabella dei simboli del nuovo device inserito.
  - ArgListDevice la lista dei device da collegare.

Noto ciò, vengono avviate le funzioni

- e. defSymRef(\$2, \$5, NULL):

```
//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)
{
    if(name->syms) argsListfree(name->syms);
    if(name->func) treefree(name->func);
    name->syms = syms;
    name->func = func;
}
```

la funzione prende, in questo caso, come parametri il simbolo da inserire e la lista dei device a cui è collegato. In tal caso rispetto all'esempio del semplice INSERT STRING la funzione ha un'utilità più profonda. Il suo obiettivo è infatti quello di aggiornare nella tabella dei simboli inserendo nel campo 'list' la lista dei device a cui andrebbe collegato

| Tabella Simboli    |                |       |          |        |                |
|--------------------|----------------|-------|----------|--------|----------------|
| Indice (Hash+Name) | Name           | Value | Function | Device | List           |
| Device_nuovo_c1    | Device_nuovo_c |       | 0 NULL   | NULL   | NULL           |
| Device_nuovo_d1    | Device_nuovo_d |       | 0 NULL   | NULL   | NULL           |
| Device_A1          | Device_A       |       | 0 NULL   | NULL   | Device_nuovo_c |

- f. La funzione \$\$ = newDev(\$2,\$5); esattamente come prima crea un nuovo nodo dell'albero. Cioè viene creato un nuovo Device aggiornando la corrispondente tabella:

| Oggetto Device |        |                  |                  |
|----------------|--------|------------------|------------------|
| NodeType       | Status | S (Symbol)       | I (argList)      |
| D              |        | 0 Devicenuovo_c1 | Null             |
| D              |        | 0 Devicenuovo_d1 | Null             |
| D              |        | 0 Device_A1      | {Device_nuovo_c} |

Sono stati esempi generici, ma è chiaro che un altro tipico comando potrebbe essere:

**CMD Device\_H20(Device\_diottrico) = newDevice Device\_D -> [Device\_nuovo\_C]**

Il risultato finale che si otterebbe con analoghi ragionamenti è il seguente

| Tabella Simboli    |                |       |          |        |                  |
|--------------------|----------------|-------|----------|--------|------------------|
| Indice (Hash+Name) | Name           | Value | Function | Device | List             |
| Device_nuovo_c1    | Device_nuovo_c |       | 0 NULL   | NULL   | NULL             |
| Device_nuovo_d1    | Device_nuovo_d |       | 0 NULL   | NULL   | {Device_nuovo_c} |
| Device_A1          | Device_A       |       | 0 NULL   | NULL   | {Device_nuovo_c} |

| Oggetto Device |        |                  |                  |
|----------------|--------|------------------|------------------|
| NodeType       | Status | S (Symbol)       | I (argList)      |
| D              |        | 0 Devicenuovo_c1 | Null             |
| D              |        | 0 Devicenuovo_d1 | {Device_nuovo_c} |
| D              |        | 0 Device_A1      | {Device_nuovo_c} |

Una volta effettuare queste operazioni principali l'utente potrà effettivamente iniziare a lavorare coi device. Per fare ciò sono state impostate diverse funzionalità di base che analizziamo:

## **FUNZIONI EMBEDDED/SISTEMA:**

La produzione principale è la seguente:

```
exec stmt EOL { ..... }
```

Una volta che l'utente ha inserito i device può avviare diversi statement su di essi. La produzione stmt definisce di fatto le varie operazioni che l'utente può effettuare. Tali operazioni riguardano if, else, for e funzioni run-time e embedded:

```
stmt: IF exp THEN listStmt      { $$ = newContent('T', $2, $4, NULL); }
      | IF exp THEN listStmt ELSE listStmt { $$ = newContent('T', $2, $4, $6); }
      | WHILE exp DO listStmt     { $$ = newContent('W', $2, $4, NULL); }
      | exp
      ;
```

Preoccupandoci dell'espressioni embedded richiamiamo ancora una volta la produzione exp che conterrà la produzione:

- FUNCDEV explistStmt

## **FUNCDEV:**

Le FUNCDEV sono state accennate in precedenza e le funzioni previste sono:

- "connect" { yyval.func = B\_connect; return FUNCDEV; }
- "reconnect" { yyval.func = B\_reconnect; return FUNCDEV; }
- "status" { yyval.func = B\_status; return FUNCDEV; }
- "switchOn" { yyval.func = B\_switchOn; return FUNCDEV; }
- "switchOff" { yyval.func = B\_switchOff; return FUNCDEV; }
- "delete" { yyval.func = B\_archive; return FUNCDEV; }
- "interval" { yyval.func = B\_interval; return FUNCDEV; }

## **EXPLISTSTMT:**

explistStmt: exp

```
| exp ',' explistStmt { $$ = newast('L', $1, $3); }
;
```

Ancora una volta un ruolo importante è rappresentato dalla produzione exp. Come spiegato nei CENNI, le più banali exp sono: STRINGHE, NAME. Possono eventualmente essere passate strutture condizionali, ecc...

## **CONNECT:**

**Scopo:** La connect di fatto funziona come ping è a come scopo primario quello di verificare se un dispositivo è connesso. Il come questa verifica avvenga, abbiamo preferito non analizzarlo nel dettaglio ma semplicemente stampa a video un messaggio di errore nel caso in cui la connessione non sia riuscita correttamente o, se viceversa, è riuscita correttamente.

Per la connessione dei device effettiva la funzione che bisogna usare è connect che nel builFunc è indicata come B\_connect = 2.

Nel caso banale che in precedenza siano state eseguite le funzioni precedentemente spiegate:

>CMD Device\_H20(Device) = newDevice DeviceB

>CMD Device\_H20(Device) = newDevice Device\_A -> [DeviceB]

Con le operazioni sopra effettuate, la tabella dei simboli e i nodi dell'albero (al momento non sono stati valutati e quindi collegati, ma restano indipendenti) è la seguente:

| <b>Tabella Simboli</b> |         |       |            |              |              |
|------------------------|---------|-------|------------|--------------|--------------|
| Indice (Hash+Name)     | Name    | Value | Function   | Device       | List         |
| DeviceB100             | DeviceB | 0     | NULL       | NULL         | NULL         |
| DeviceA100             | DeviceA | 0     | NULL       | NULL         | {DeviceB100} |
| <b>Nodo AST</b>        |         |       |            |              |              |
| Device                 | D       | 0     | DeviceB100 | Null         |              |
| Device                 | D       | 0     | DeviceA100 | {DeviceB100} |              |

Se l'utente volesse connettere i due device. Ipotizziamo, quindi, il seguente comando:

>connect Device\_A

Si avvierà quindi la seguente produzione:

**exec stmt EOL**, in cui:

- L'exec è la produzione precedente che ha portato l'inserimento del deviceA100
- La produzione stmt prevede al suo interno la produzione exp. Al suo interno è, quindi, prevista la seguente produzione:

```
FUNCDEV explistStmt {
```

```
 $$ = newfunc($1, $2);
```

```
}
```

- Verrà quindi rilevato la word CONNECT che come già detto identifica una funzione embedded specificata dal token FUNCDEV.

```
"connect" { yylval.func = B_connect; return FUNCDEV; }
```

- La produzione explistStmt contiene al suo interno la produzione exp che gestisce le singole stringhe tramite il token String. Nel caso ritornerà il puntatore al symbol della tabella dei simboli che fa riferimento al Device\_A. Riconosciuto il token string verrà eseguita la funzione:

```
STRING { $$ = newString($1); }
```

La funzione newString

```

struct ast *newString(struct symbol *s)
{
    struct stringVal *a = malloc(sizeof(struct stringVal));

    if(!a) {
        yyerror("Spazio di memoria insufficiente\n");
        exit(0);
    }
    a->nodetype = 'C';
    a->s = s;

    return (struct ast *)a;
}

```

creerà un nodo dell'albero di tipo StringVal

```

struct stringVal {
    int nodetype;           /* tipo nodo C -> valore stringa costante*/
    struct symbol *s;
}

```

composto da due campi:

- Il tipo di operazione: S
- Il symbol di memoria a cui fa riferimento (nel caso DeviceA100)

In memoria saranno quindi contenute le seguenti informazioni:

| <b>Tabella Simboli</b> |         |       |          |        |              |
|------------------------|---------|-------|----------|--------|--------------|
| Indice (Hash+Name)     | Name    | Value | Function | Device | List         |
| DeviceB100             | DeviceB |       | 0 NULL   | NULL   | NULL         |
| DeviceA100             | DeviceA |       | 0 NULL   | NULL   | {DeviceB100} |

| <b>Nodo AST</b> |   |              |              | Indirizzo Memoria |
|-----------------|---|--------------|--------------|-------------------|
| Device          | D | 0 DeviceB100 | Null         |                   |
| Device          | D | 0 DeviceA100 | {DeviceB100} |                   |
| StringVal       | C | DeviceA100   |              | 3FFFFFFF          |

Rivalutando la produzione di riferimento: \$\$ = newfunc(\$1, \$2).

```

struct ast *newfunc(int functype, struct ast *l, struct ast *r)
{
    struct funcBuiltIn *a = malloc(sizeof(struct funcBuiltIn));

    if(!a) {
        yyerror("Spazio di memoria insufficiente\n");
        exit(0);
    }
    a->nodetype = 'F';
    a->l = l;
    a->r = r;
    a->functype = functype;
    //printf("%d", functype);           //DEBUG

    return (struct ast *)a;
}

```

Si ha che \$1 contiene il tipo di funzione eseguita (connect), \$2 il valore StringVal del nodoAst

- c. Valutando il corpo si ha quindi l'esecuzione della funzione newfunc. Viene quindi creato un oggetto **funcBuiltIn**

```

/* Struttura Nodo per l'AST, per le funzioni predefinite */
struct funcBuiltIn {
    int nodetype;          /* tipo nodo F -> Funzioni Built-In*/
    struct ast *l;
    struct ast *r;
    enum builtFunc functype;
};

```

come nodo AST che conterrà i campi specificati di seguito:

|                    | <b>Nodo AST</b> |            |            | <b>Indirizzo Memoria</b> |
|--------------------|-----------------|------------|------------|--------------------------|
| <b>Device</b>      | D               | 0          | DeviceB100 | Null                     |
| <b>Device</b>      | D               | 0          | DeviceA100 | {DeviceB100}             |
| <b>StringVal</b>   | C               | DeviceA100 |            | 3FFFFFFF                 |
| <b>funcBuiltIn</b> | F               | 3FFFFFFF   | "Connect"  |                          |

Si è quindi collegato il nodo dell'albero connect a quello all'indirizzo di memoria 3FFFFFFF.

- c. Ritornando alla produzione principale: exec stmt EOL. Può essere eseguito il suo corpo il quale richiama la funzione eval. La funzione eval quando prende come ast d'ingresso un tipo F richiamerà callbuiltin. Callbuiltin è la funzione che si occuperà nel nostro programma della gestione di tutte le funzioni embedded. Tra queste è ovviamente presente la funzione B\_connect:

- a. Tramite il secondo campo 3FFFFFFF di funcBuiltIn (guarda tab sopra) si può recuperare il campo name del symbol "DeviceA100". Il campo name è ovviamente DeviceA
- b. Recuperato il dispositivo il dispositivo verrà connesso tramite un'operazione di ping. Ovviamente sarà necessario un sistema HTTP per fare ciò.

### 5.2.1 Risultati ottenuti e punti di forza

Una volta descritte le funzioni embedded che consentono di realizzare la gestione della serra, sono state realizzate funzioni di alto livello che consentono di realizzare operazioni complesse.

Al fine di facilitare l'utilizzo del linguaggio da parte dell'utente, sono stati realizzati due file che saranno caricati in fase di avvio. Tali file costituiscono:

- Una libreria dove vengono inserite funzioni di alto livello
- Un database dove l'utente può inserire i propri device al fine di ritrovarli già definiti ad ogni riavvio.

In particolare:

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"

```

```

+-----+-----+-----+
|W|e|l|c|o|m|e| |t|o| |t|h|e|
+-----+-----+-----+
ad88888ba 88888888888 888888888ba 888888888ba      db
d8'   '8b 88      88      '8b 88      '8b      d88b
Y8,      88      88      ,8P 88      ,8P      d8' '8b
`Y8aaaa,  88aaaaa 88aaaaaa8P' 88aaaaaa8P'      d8' '8b
`aaaayy8  88aaaaa 88''''88' 88''''88h      8Yaaaaaa8b
`8b 88      88      '8b 88      '8b      d8''''''8b
Y8a     88P 88      88      '8b 88      '8b  d8'      '8b
YY88888PP 88888888888 88      '8b 88      '8b  d8'      '8b
+-----+-----+-----+-----+-----+-----+-----+
|p|r|o|g|r|a|m|m|i|n|g| |l|a|n|g|u|a|g|e|||
+-----+-----+-----+-----+-----+-----+-----+

```

```

+-----+-----+-----+-----+-----+-----+-----+
| |W|r|i|t|e| |h|e|l|p| |i|f| |y|o|u| |n|e|e|d| |i|t| |
+-----+-----+-----+-----+-----+-----+-----+

```

>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

> Display: EndDBLoading

> Display: LoadingLibrary

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

### **Punto di forza:**

Possiamo quindi classificare 3 punti di forza nel nostro linguaggio:

#### **1. SEMPLICITA':**

Dato, quindi, il target rappresentato da gente, spesso, inesperta in ambito di linguaggi di programmazione, l'idea di base è stata quella di definire un file DB e un file library:

- Nel file DB l'utente andrà a caricare i device che si interfacciano alla serra. Di conseguenza il giardiniere non dovrà far altro che aprire tale file e sostituire i nomi dei device (nell'esempio sopra li abbiamo chiamati valvolaSud, est ecc)
- Nel file library sono caricate le funzioni di sistema che dovrà usare. Tali funzioni sono state definite da chi ha creato il linguaggio. E l'utilizzatore non dovrà modificare questo file ma solo utilizzarne le funzioni definite all'interno.

Il principale punto di forza del linguaggio è dunque la semplicità, dato che all'utente basta aggiornare il file DB con i propri device e successivamente avviare l'interprete richiamando le funzioni di libreria. Di seguito ne riportiamo un esempio:

- L'utente scrive nel file DB i device che utilizzerà nella Serra:

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

- L'utente avvia il linguaggio scrivendo:

```
>/exec.sh
>/serra
```

- Una volta avviato l'interprete gli basterà richiamare una funzione per la sua operazione. Nel caso richiamiamo la funzione routine:

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

> Display: LoadingLibrary

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

> routine ("valvolaSud")
Display: routineAvviata
Display: stepAvviato
Ricerca del dispositivo valvolaSud in corso...
Dispositivo Esistente
Richiesta connessione...
inizia switchOn
          Verifica in corso della connessione del dispositivo...
Ricerca del dispositivo valvolaSud in corso...
Dispositivo Esistente
Richiesta connessione...

          La connect è andata a buon fine e il dispositivo è stato acceso
          Verifica in corso della connessione del dispositivo...
Ricerca del dispositivo valvolaSud in corso...
Dispositivo Esistente
Richiesta connessione...

          Il dispositivo è spento

Display: stepConcluso
Display: stepAvviato
Ricerca del dispositivo valvolaSud in corso...
Dispositivo Esistente
Richiesta connessione...
inizia switchOn
          Verifica in corso della connessione del dispositivo...
Ricerca del dispositivo valvolaSud in corso...
Dispositivo Esistente
Richiesta connessione...

          La connect è andata a buon fine e il dispositivo è stato acceso
          Verifica in corso della connessione del dispositivo...
Ricerca del dispositivo valvolaSud in corso...
Dispositivo Esistente
Richiesta connessione...

          Il dispositivo è spento

Display: stepConcluso
Display: routineConclusa
> ■
```

Note: il file ./exec.sh esegue i comandi per generare l'eseguibile serra.

## 2. AGGIORNAMENTO e RIUSO:

Nel progetto sono state previste anche funzioni più complesse, dando, per esempio, la possibilità all'utente di definirsi lui le proprie funzioni di sistema. Tale aspetto è stato comunque previsto (nonostante l'idea è sempre quella di rendere il programma il più semplice possibile) per consentire a eventuali giardinieri di definire le proprie funzioni di comodo. Può essere, comunque, considerato un punto di forza perché consente all'utente di definire funzioni di comodo per fare operazioni magari che in futuro riutilizzerà spesso.

## 3. FUNZIONI LEGATE AL CONTESTO

Un punto di forza sta anche nel fatto che nel linguaggio sono state già definite delle funzioni embedded come parole chiavi (connect, reconnect, newDevice, switchOn, ecc). Di conseguenza l'utente dovrà usare un linguaggio che sia il più semplice possibile e comunque in ambito di linguaggi di programmazione; infatti, è la natura stessa del linguaggio che guida l'utente ad identificare il risultato che otterrà dall'esecuzione delle funzioni, come ad esempio ciò che si ottiene da connect "dev".

Di seguito riportiamo esempi di funzioni che un programmatore un po' più esperto può definire.

## 6 Esempio di codice di nel linguaggio SERRA

### 6.1 InnaffiaElse

```

1 print "InnaffiaElseStart"
2
3 newDevice "valvolaSud"
4 newDevice "valvolaEst"
5 newDevice "valvolaOvest"
6 newDevice "pompa"
7
8 status "pompa"
9
10 newArray char periferiche(4)
11 periferiche->add = "valvolaSud"
12 periferiche->add = "valvolaEst"
13 periferiche->add = "valvolaOvest"
14 periferiche->add = "pompa"
15
16 switchOn "pompa"
17 status "pompa"
18
19 CMD irriga (data1, data2, data3) =
20 if status periferiche->get=3 then interval periferiche->get=2 - 90 - data1;
21 interval periferiche->get=1-60-data2;
22 interval periferiche->get=0 - 30 - data3; switchOff "pompa"; status "pompa";;
23
24 if connect periferiche->get=3 then irriga(2020.9.11.14.52, 2020.9.11.14.51, 2020.9.11.14.51);
25 else reconnect periferiche->get=3;
26
27 delete "pompa"
28
29 if connect periferiche->get=3 then irriga(2020.9.11.14.52, 2020.9.11.14.51, 2020.9.11.14.51);
30 else reconnect periferiche->get=3; newDevice "pompa";
31
32 print "innaffiaElseFinish"

```

## 6.2 InnaffiaElseReconnectDelete

```
C: > 4_Linux > funzionante > innaffiaElseReconnectDelete
1 print "InnaffiaElseStart"
2
3 newDevice "valvolaSud"
4 newDevice "valvolaEst"
5 newDevice "valvolaOvest"
6 newDevice "pompa"
7
8 status "pompa"
9
10 newArray char periferiche(4)
11 periferiche->add = "valvolaSud"
12 periferiche->add = "valvolaEst"
13 periferiche->add = "valvolaOvest"
14 periferiche->add = "pompa"
15
16 switchOn "pompa"
17 status "pompa"
18
19 CMD irriga (data1, data2, data3) = if status periferiche->get=3 then interval periferiche->get=2 - 90 - data1;
20 interval periferiche->get=1-60-data2; interval periferiche->get=0 - 30 - data3; switchOff "pompa"; status "pompa";;
21
22 if connect periferiche->get=3 then irriga(2020.9.11.14.52, 2020.9.11.14.51, 2020.9.11.14.51);
23 else reconnect periferiche->get=3;
24
25 delete "pompa"
26
27 if connect periferiche->get=3 then irriga(2020.9.11.14.52, 2020.9.11.14.51, 2020.9.11.14.51);
28 else reconnect periferiche->get=3;
29
30 newDevice "pompa"
31
32 print "innaffiaElseFinish"
```

## 6.3 Irrigazione

```
C: > 4_Linux > funzionante > irrigazione
1 newDevice "valvolaSud"
2 newDevice "valvolaEst"
3 newDevice "valvolaOvest"
4 newDevice "pompa"
5
6 switchOn "pompa"
7
8 if connect "pompa" then if status "pompa" then interval "valvolaSud" - 90 - 2020.9.11.14.4;
9 interval "valvolaEst"-60-2020.9.11.14.4;
10 interval "valvolaOvest"-30-2020.9.11.14.3;
11 switchOff "pompa";
12 status "pompa";
13 ;
```

## 6.4 IrrigazioneArray

```
C: > 4_Linux > funzionante >  irrigazioneArray
 1 newDevice "valvolaSud"
 2 newDevice "valvolaEst"
 3 newDevice "valvolaOvest"
 4 newDevice "pompa"
 5
 6 switchOn "pompa"
 7
 8 newArray char periferiche(4)
 9 periferiche->add = "valvolaSud"
10 periferiche->add = "valvolaEst"
11 periferiche->add = "valvolaOvest"
12 periferiche->add = "pompa"
13
14 if connect periferiche->get=3 then
15   if status periferiche->get=3
16     then interval periferiche->get=2 - 90 - 2020.9.11.14.26;
17       interval periferiche->get=1-60-2020.9.11.14.26;
18       interval periferiche->get=0 - 30 - 2020.9.11.14.25;
19       switchOff "pompa";
20       status "pompa";
21 ;
```

## 6.5 IrrigazioneFunzione

```
C: > 4_Linux > funzionante >  irrigazioneFunzione
 1 newDevice "valvolaSud"
 2 newDevice "valvolaEst"
 3 newDevice "valvolaOvest"
 4 newDevice "pompa"
 5
 6 status "pompa"
 7
 8 newArray char periferiche(4)
 9 periferiche->add = "valvolaSud"
10 periferiche->add = "valvolaEst"
11 periferiche->add = "valvolaOvest"
12 periferiche->add = "pompa"
13
14 switchOn "pompa"
15
16 status "pompa"
17
18 CMD irriga (data1, data2, data3) =
19 if status periferiche->get=3 then
20 interval periferiche->get=2 - 90 - data1;
21 interval periferiche->get=1-60-data2;
22 interval periferiche->get=0 - 30 - data3;
23 switchOff "pompa";
24 status "pompa";
25 ;
26
27 if connect periferiche->get=3 then
28 irriga(2020.9.11.14.52, 2020.9.11.14.51, 2020.9.11.14.51);
```

## 7. Conclusioni

### 7.1 Sviluppi futuri

Scaletta:

- Stampa con interfaccia grafica che rappresenti i dispositivi e il loro stato
- Funzioni definite dall'utente:
  - Definire le variabili locali all'interno di una funzione (esempio sotto)
- Funzioni embedded:
  - Misurare temperature e umidità

#### Variabili locali:

Analoghi ragionamenti ed esempi possono essere fatti nel caso in cui passiamo la funzione più parametri. Nel caso vengono passati due parametri e ne viene definito uno interno alla funzione (v3). Si nota come le variabili v1, v2 della funzione chiamante non subiscano modifiche nel momento in cui viene chiamata “nomeFunzione”. La variabile v3 continuerà a non esistere all'esterno della funzione nomeFunzione.

Per implementare tale meccanismo l'idea è di allocare le variabili della funzione all'interna di una nuova tabella dei simboli creata per ogni singola funzione definita dall'utente. In basso si riporta un esempio:

```
> CMD nomeFunzione (v1,v2) = print v1; v3=7; print v3; print v2;
User Mode:      funzione definita
> nomeFunzione (1,2)
WIN:-1
Display:      1
Display:      7
Display:      2
WIN:-1

> print v1
Display:      5

> print v2
ERROR: Il parametro passato alla funzione printf non è né un numero né una stringa

> print v3
ERROR: Il parametro passato alla funzione printf non è né un numero né una stringa
```

Una versione del nostro linguaggio con le variabili locali è stata realizzata. Tuttavia, su macchine che girano su piattaforma diversa da MacOS si sono riscontrati dei problemi di portabilità. Si è, quindi, deciso di abbandonare tale versione del progetto in fase di sviluppo. Si è, quindi, deciso di realizzare la versione con le variabili globali e con un'unica tabella dei simboli.

#### Nota:

Per completezza si allega la versione definitiva multiplattafoma con le variabili globali e la versione abbandonata con le variabili locali per MacOS:

- <https://github.com/googlielmo93/serra.git> -> progetto completo
- <https://github.com/googlielmo93/serra-versione-macOS.git> -> versione abbandonata

## Bibliografia

- [Figura 5 - https://it.electronics-council.com/farmbot-intends-revolutionize-home-gardening-31975](https://it.electronics-council.com/farmbot-intends-revolutionize-home-gardening-31975)
- [Figura 1 - https://6ff66f56-a-62cb3a1a-s-sites.googlegroups.com](https://6ff66f56-a-62cb3a1a-s-sites.googlegroups.com)
- [flex & bison: Text Processing Tools, John Levine \(Autore\)](#)
- [Materiale Didattico, Ing. Prof. Chella](#)
- [Materiale Didattico, Ing. Lanza](#)
- <http://www.pluto.it/sites/default/files/journal/pj0810/bison.html>
- [https://aquamentus.com/flex\\_bison.html](https://aquamentus.com/flex_bison.html)
- <https://www.gnu.org/software/bison/>
- <http://gensoft.pasteur.fr/docs/flex/2.6.1/>
- <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html>
- [https://it.wikipedia.org/wiki/File\\_batch](https://it.wikipedia.org/wiki/File_batch)
- [https://it.wikipedia.org/wiki/Bourne\\_shell](https://it.wikipedia.org/wiki/Bourne_shell)
- [https://it.wikipedia.org/wiki/Cool\\_\(linguaggio\)](https://it.wikipedia.org/wiki/Cool_(linguaggio))
- <https://stackoverflow.com/questions/tagged/bison>