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:
Thermokípio

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

ANNO ACCADEMICO 2019 - 2020

Sommario

Descrizione del Team

Il Team Thermokípio 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 delle persone di circondarsi di dispositivi che si mettono a loro disposizione in modo non invadente, ha scelto di lavorare nell'ambito dell'Ambient Intelligence (AMI).

L'AMI, quindi, è lo scenario nel quale gli uomini vivono in un ambiente con tecnologie informatica e telematiche, dove sono presenti dispositivi con capacità computazionali e di connessione in rete, che operano per sgravare il lavoro degli uomini.

Per ottenere Ambient Intelligence occorrono diverse tecnologie chiave, sia hardware (miniaturizzazione dei componenti, sensori, dispositivi intelligenti) che software.

Alla base dell'idea dell'AMI ci sono due caratteristiche:

- I dispositivi inseriti non devono essere invadenti nei nostri ambienti
- L'utilizzo deve essere User-Friendliness.

Il nostro progetto si è concentrato sulla seconda caratteristica, ossia sullo sviluppo di un linguaggio per dispositivi AMI che ne renda semplice la programmazione; ed in particolare si è deciso di sviluppare, un linguaggio per l'automazione dei sistemi che posso gestire la cura di giardini, orti e serre, chiamato SERRA.

2. Stato dell'arte

3. Descrizione del progetto

3.1 Analisi dei requisiti

4. Caratteristiche del Linguaggio

Grammatica:

Per realizzare un analizzatore lessicale è stato utilizzato 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 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:

<A> -> α [α 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).

Analizzatore lessicale:

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

Nella prima due sezioni del lexer sono state individuate le seguenti definizioni regolari da usare con i corrispondenti token. I token individuati sono quindi seguenti:

```
/* declare tokens */
%token <d> NUMBER
%token <str> STRING
%token <s> NAME
%token <func> FUNC
%token <func> SYSTEM
%token <func> FUNCDEV
%token <func> INTERVAL
%token <func> INSERT
%token <func> CMP
%token <str> ARROW
%token EOL
%token <str> TERM
%token IF THEN ELSE WHILE DO CMD
```

Descrivendoli nel dettaglio:

Definizioni regolari:

- Il token **number** è individuato dalla seguente definizione regolare:
  number -> [0-9]+"."[0-9] | "."?[0-9]*
  - **Ex:** 1 , 1.1 , 1.12 , 1.25

Tale token consente di inserire nel nostro lessico tutti i numeri floating point e interi 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. 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-> \"([^\\\"]|\\.)*\"

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

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 i possibili nomi di variabili e funzioni (token nama);

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 pretendono 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 epressioni regolari "do", "while".

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

- Clear individuata dall'espressione "clear"
- ReadFile individuata dall'esprssione "readFile"

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 due parametri)

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

- Funzioni di sistema (clear e readFile)
- Funzioni di device (connect, reconnect, status, interval ecc)
- 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 verrano segnalati come caratteri sconosciuti e quindi come errore lessicale.

Nella seguente tabella riassumiamo quanto spiegato in precedenza:

Token	Espressioni Regolari	Parole chiavi
STRING	["][a-zA-Z][a-zA-Z0-9]*["]	
NUMBER	[0-9]+"."[0-9] | "."?[0-9]*	
NAME	[a-zA-Z][a-zA-Z0-9]*	
ARROW	"->"	
EOL	\n	
MATCH	"terminatore"	
IF		"if"
ELSE		"else"
THEN		"then"
WHILE		"while"
DO		"do"
CMP		> < >= <=
FUNCDEV		"connect"
		"status"
		"reconnect"
		"swtichOn"
		"switchOff"
		"archive"
		"interval"
		"diagnostic"
SYSTEM		"clear"
FUNC		"print"
		"readFile"

Analizzatore sintattico:

È stato realizzato utilizzando il SW 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
    - Espressioni
- Exp: È un simbolo non terminale che consente l'individuazione delle:
    - Variabili, funzioni di sistema/embedded e funzioni runtime.

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

EXP:

E' un simbolo non terminale per individuare tutte le espressioni presenti nel lingauggio. Un exp/espressione può essere:

- Un numero (simbolo terminale): vuol dire che l'utente ha digitato un numero riconosciuto dal token NUMBER
  Exp -> NUMBER
    - 2 3 5
- Una stringa (simbolo terminale): vuol dire che l'utente ha digitato una stringa riconosciuto dal token STRING
  Exp->STRING
    - "ciao" "pippo"
- 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 riporate 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 ecc) verranno richiamate prendendo come parametri o i token STRING, NAME, NUMBER ecc, 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 (controllare):
  Exp->FUNCDEV expListStmt
    - Esempio in cui passo come parametro un simbolo terminale:
        - Connect "ciao"
        - Connect "pippo"
        - Connect "pluto"
        - SwitchOn "nome"
        - SwitchOff 1
        - Connect 2
    - Esempio in cui passo come paraemtro un simbolo non terminale:
        - Connect switchOff "pluto": passo come parametro una funzione embedded
        - Connect var : passo come parametro una variabile

  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 explistStmt

- Una variabile (simbolo terminale) seguito da un uguale (simbolo terminale) seguito da un'espressione (simbolo non terminale): una variabile (che è semplicmente una stringa) è

individuata infatti dal token descritto nella sezione dell'analisi lessicale. E' seguita dal simbolo terminale uguale seguita 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

- 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-> INSERT STRING
    - NewDevice "pippo"
  - La seconda consente non solo di creare un device ma anche di indicare che è collegato ad altri device:
    EXP-> INSERT STRING ARROW '[' argsListDevice ']'
    - NewDevic "pippo" -> ["ciao"]
- Interval: analogo alle funzioni embedded

EXEC (esecuzione del comando):

Una volta individuata l'espressione exp e l'istruzione stm 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'inivio (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 nota che però 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.

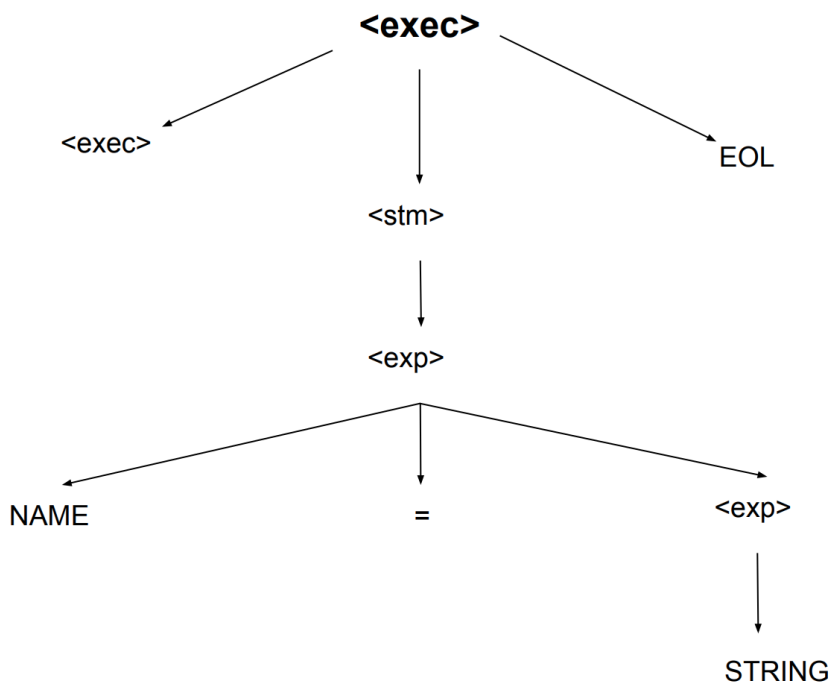
```
exec ->         exec stmt EOL ;

stmt ->         IF exp THEN listStmt                    |
                exp                                     ;

exp ->          NUMBER | STRING | NAME                  |
                INSERT STRING                           |
                INSERT STRING ARROW [ argsListDevice ]  |
                FUNC exp                                |
                FUNCDEV expliStmt                       |
                NAME = exp                              |
                INTERVAL explistmt - explistmt          ;

explistStmt ->  ex      p                               |
                exp , explistStm        t               ;

~
~
~
~
~
~
~
~
```

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

Esempi di applicazione e relativo albero:

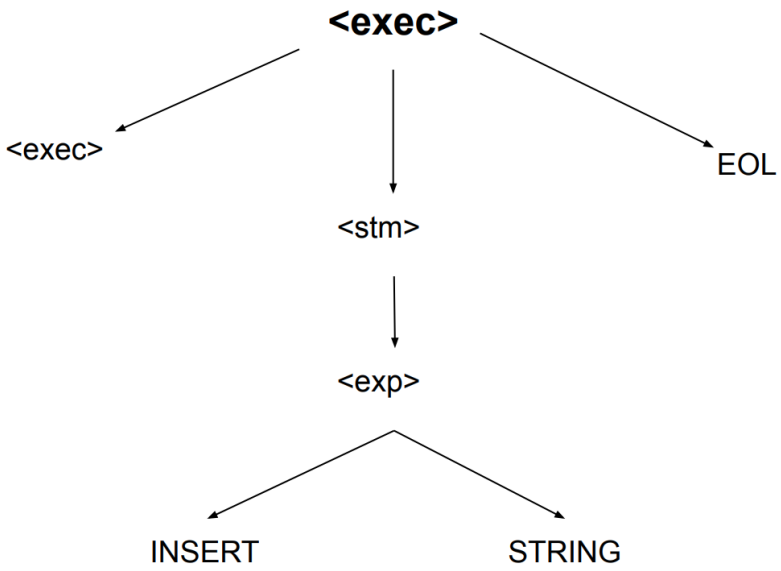
Per l'analisi dell'albero effettuare un'analisi discendente per semplificare l'analisi:

Creazione variabile:
Istruzione:
variabile= "pippo"



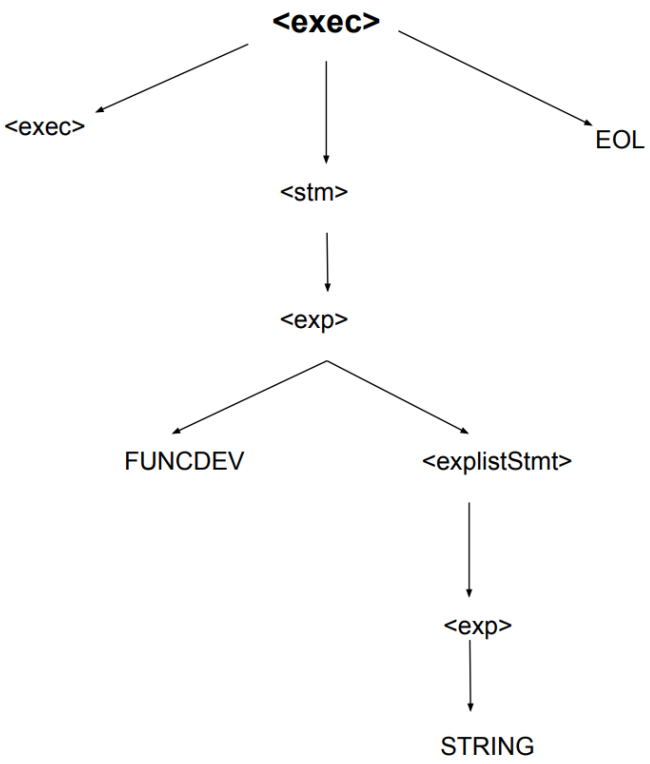
Analogamente non è detto che la seconda exp riportata al terzo livello dell'albero sia una STRING. Potrebbe essere un NUMBER o una qualsiasi altra delle espressioni che il programma ha definito.

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:

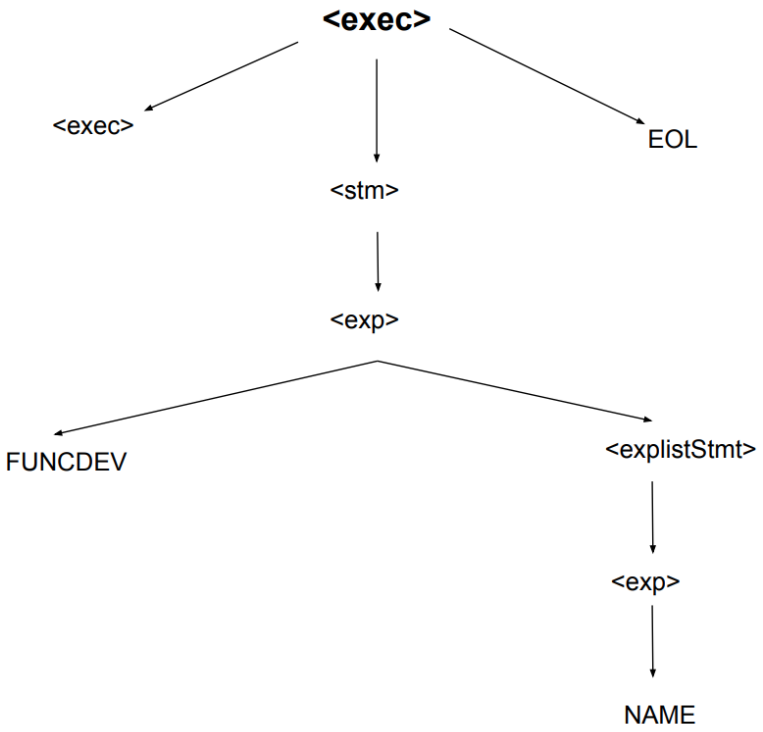
- switchOn “pippo”
- Connect “pippo”
- Reconnect “pippo”
- Status “pippo”
- Archive “pippo”



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

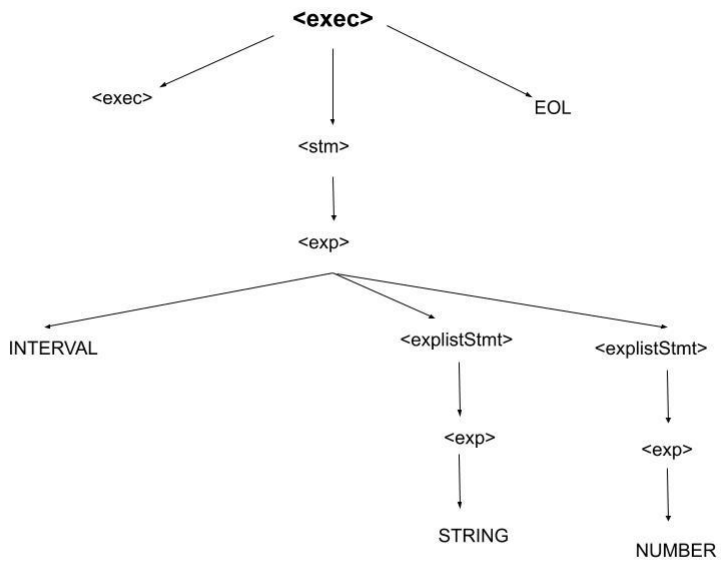
- Supponiamo di creare la variabile: dino="pippo"
- E successivamente di effettuare: switchOn dino

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 due parametri di input e quindi l'albero risulta con un nodo terminale in più nel livello più basso dell'albero:

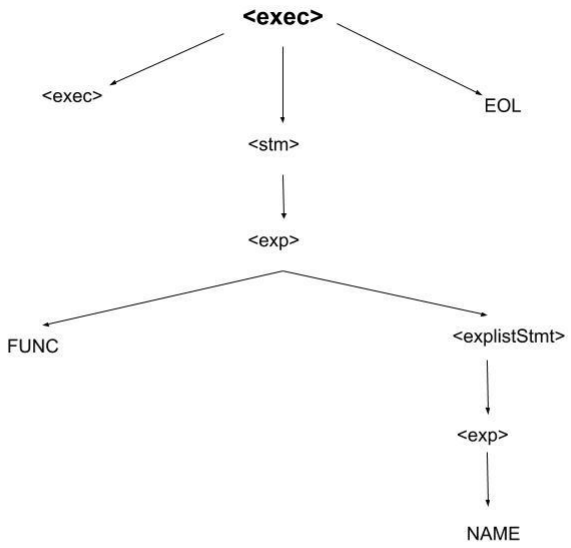
- Interval "pippo"-5



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.

Sintassi ad alto livello:

Tipi:

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"
- Puntatori: Non sarà un tipo dichiarabile dall'utente ma spesso le funzioni embedded potranno restituire un valore di ritorno.

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

Tipi compositi:

- Mapping: Da accordarci se dobbiamo inserirlo o no (Nelle specifiche ci sono)
- Liste (tipi ricorsivi): Da accordarci (nelle specifiche ci sono)
- 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.

        - ReferenceDevice {1, [DeviceA, DeviceB....DeviceN], [NomeDevice;] }

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.

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.

Definizione costanti e stringhe:

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". Nell'esempio è più chiaro ciò che si intende. L'utente è come se 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 compositi 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 decritte nelle sezioni seguenti). Una funzione embedded ovviamente potrà restituire una Stringa, un

intero o anche un riferimento a una zona di memoria. In tal caso all'interno della variabile verrà memorizzato il riferimento a tale zona di memoria. Per esempio, la funzione "INSERT DEVICE nomeDevice" restituisce il riferimento alla zona di memoria in cui è stato memorizzato l'oggetto device:

```
> var= newDevice "dev2"
Dispositivo inserito con successo con ID: dev2#9781

> print dev2
ERROR: Il parametro passato alla funzione printf non è nè un numero nè una stringa
> 
```

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

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 ovvero 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 (da verificare). 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. E' 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. E' inoltre un linguaggio case sensitive.

Definizione tipo device: (struct device)-> Tramite la connect (?)

_____chiedere a Vincenzo_____

Parole chiavi del linguaggio:

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

Parole chiavi	
connect	F.Embedded
readFile	F.Embedded
insert	F.Embedded
device	Tipo
term	Terminatore file
interval	F.Embedded
switchOn	F.Embedded
switchOff	F.Embedded
archive	F.Embedded
print	F.Embedded
rconnect	F.Embedded
Arrow	Indicatore ->
status	F.Embedded
newDevice	F.Embedded
CMD	Identificatore
if	Op. condizionale
then	Op. condizionale
else	Op. condizionale
for	Op. condizionale
do	Op. condizionale
while	Op. condizionale
clear	F.Embedded

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 del programmatore. Nella seguente sezione sono descritte le principali funzioni di sistema previste:

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

È una funzione a cui passi come parametro un tipo primitivo stringa e restituisce il riferimento alla zona di memoria (nel caso alla tabella dei simboli) dove è stato memorizzato il device. Una stringa abbiamo spiegato nelle sezioni precedenti essere un tipo primitivo del linguaggio specificato 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 "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] ]

>
```

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 *connect (string):**

Anche questa rappresenta una funzione embedded con lo scopo di effettuare la connessione tra i vari device dell'albero. Una volta che è stato generato l'oggetto Device (tramite le funzioni embedded precedenti) è possibile verificare la raggiungibilità di un device. La funzione connect è quindi assimilabile a un ping il quale scopo è di verificare la raggiungibilità di un device. La funzione quindi preso in input il device come stringa verificherà la sua esistenza nella tabella dei simboli. In caso affermativo effettua una connessione HTTP 200 col Device (da noi non implementata). Il meccanismo con cui verrà effettuata quindi tale connessione non è stato dunque implementato e viene semplicemente ipotizzato che il device risponderà con un ping affermativo.

Viene riportato sotto un esempio. E' 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 "device_ovest"
Dispositivo inserito con successo con ID: device_ovest#6068
Operazione di inserimento dispositivo completata con successo

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

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

```
>newDevice "device_ovest"
Dispositivo inserito con successo con ID: device_ovest#6068
Operazione di inserimento dispositivo completata con successo

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

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

- **Nodo * switchOn (string):**

Una volta effettuata una connect (per verificare che il device è effettivamente raggiungibile) l'utente può pensare di considerare il dispostivo 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. Un device che quindi è stato in precedenza creato viene quindi 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 "device_sud"
Dispositivo inserito con successo con ID: device_sud#557
Operazione di inserimento dispositivo completata con successo

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

- **Nodo * switchOff (string):**

Analogo alla funzione switchOff ma considera il dispostivo come disattivo settando quindi lo status a 0:

- Status: off/0
- 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 "deviceDaAttivare"

```
> newDevice "device_sud"
Dispositivo inserito con successo con ID: device_sud#557
Operazione di inserimento dispositivo completata con successo

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


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

- **Void * interval(string, number):**

Si occupa di attivare il device per un certo intervallo di tempo. La funzione prende come parametro il device sottoforma di stringa (definito precedentemente con la funzione newDevice) e il numero di secondi in cui il device deve rimanere attivo. La funzione effettua quindi le seguenti operazioni:

a. SwitchOn: accende il device settandolo come attivo
b. Fa rimanere attivo il device per number secondi
c. 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 devic per un intervallo di tempo limitato necessario a effettuare una determinata operazione di giardinaggio.

Sintassi: interval "deviceDaAttivare"-numeroSecondi

In fasi:

a. Attivo il device
b. Fase di sleep

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

> interval "device_Ovest"-9
9numero:9, stringa:device_Ovest

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

c. Disattiva Device

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

> interval "device_Ovest"-9
9numero:9, stringa:device_Ovest

AVVIO OPERAZIONE:
                        Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo device_Ovest 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 device_Ovest in corso...
Dispositivo Esistente
Richiesta connessione...
                        Il dispositivo è spento
```

La funzione nel caso in cui il secondo parametro non viene passato o è errato ipotizza di doverlo accendere per 0 secondi:

```
> interval "pippo"-"aa"
numero:0, stringa:pippo

AVVIO OPERAZIONE:
                        Verifica in corso della connessione del dispositivo....
Ricerca del dispositivo pippo 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 pippo in corso...
Dispositivo Esistente
Richiesta connessione...
                        Il dispositivo è spento
```

- **Void * archive (string):**

La funzione di archiviazione quando applicata si occupa di eliminare i device presenti nel sistema (cancellandoli di conseguenza dalla tabella dei simboli) e salvarli in un file. La necessità di salvataggio di un file è stata ipotizzata esserci perché se l'utente in un secondo momento avrà la necessità di recuperare tali device gli basterà per reinserli nella rete di device del sistema di giardinaggio gli basterà effettuare un'operazione di lettura da questo file per poterli eventualmente ricaricare nel sistema. Il file è stato ipotizzato essere di sistema "device.txt". Inizialmente questo file risulterà vuoto, poi in base alle operazioni effettuate dall'utente andrà riempendosi di nomi di device non più considerati (almeno per il momento) utili nel sistema di giardinaggio. La funzione pertato 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: archive "nomeDevice"

Esempio d'uso:

Inizialmente il file è vuoto. Una volta caricati questi 4 device con la funzione newDevice. Vengono archiviati nel file Device.txt. Di seguito si trova a regime il comportamento della funzione archive:

```
(base) MBP-di-Vincenzo:alberi vincenzolabua$ cat device.txt
device_Ovest#8100
device_Nord#5304
device_Sud#8477
device_Ovest#8100
(base) MBP-di-Vincenzo:alberi vincenzolabua$
```

```
>newDevice "device_Ovest"
Dispositivo inserito con successo con ID: device_Ovest#8100
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

> archive "device_Nord"

> archive "device_Sud"

> archive "device_Ovest"

>
```

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

È stata prevista la possibilità di consentire all'utente di eseguire i programmi non su linea di comando 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"

Le fasi sono quindi le seguenti:

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

file

```
newDevice "d1"
connect "d1"
interval "d2"-1
newDevice "d3"
var=3
print var
```

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. Se andiamo a leggere il file (dopo averlo eseguito) si nota:

Il programma per riconoscere la terminazione del file ha scritto “terminatore” come ultima riga del file. In tal caso sarà prevista la cancellazione di questa riga per evitare di far notare all’utente dettaglio di sistema.

*: da aggiungere

- **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 cerca di stampare informazioni non referenti a stringhe e numeri il programma ritornerà un errore. Tale scelta è stata fatta perché i tipi previsti nel linguaggio (Number, string, Device) le uniche informazioni utili da stampare a video possono essre stringhe e numeri. Non avrebbe infatti senso in un linguaggio come il nostro (che deve essere utilizzato anche da gente inesperta al linguaggio di programmazione) stampare indirizzi di memoria o altri dettagli di basso livello. È una funzione di comodo usata dall’utente per funzioni di debug ecc

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 nè un numero nè una stringa
```

Descrizione del linguaggio:

Analisi del codice a basso livello:

INSERT_DEVICE=10:

GESTIONE DEVICE:

La prima built function che ci occupiamo di analizzare è la:

ALTO LIVELLO:

Per la gestione dei Device 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 del device su cui bisogna effettuare la operazione andando a immagazinare tale stringa in una struct symbol (si noti per il momento ad alto livello che la funzione search si occupa semplicemente ritorni un simbolo):

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

{

CREA UN SIMBOLO NELLA TABELLA DEI SIMBOLI SE NON ESISTE, ALTRIMENTI TI RITORNA IL PUTNATORE AL SIMBOLO ESISTENTE NELLA TABELLA DEI SIBMBOLI

}

3. Una volta definito il nome del device su cui si vuole effettuare una certa operazione l'utente dovrà inserire la lista dei device associati al device inserito. Lo specificherà tra parentesi tonde. Il motivo della specifica dei device possono essere vari: effettuare in fase successiva una connessione, effettuare una disconnessione ecc.

'(' argsList ')'

   Per il momento di argList basti sapere che è un ulteriore produzione che ritorna la lista che va a creare una lista concatenata dei device già esistenti con quello che si sta inserendo. Semplicemente la produzione ritornerà una lista dei device da collegare.

4. L'utente potrà eventualmente decidere una volta specificato il device e la lista del device il tipo di operazioni che vorrà eseguire. Dopo che l'utente inserisce l'uguale puo 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" { yylval.func = B_insertDevice; return INSERT; }
```

2. L'utente quindi definirà il nome del device che vuole craeare. 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 crea semplicemente un nuovo simbolo della struct symbol

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

    b. 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;  //LO PONGO CON STATO SPENTO DI DEFAULT

d->s= sym;

d->l = l;
```

E' 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.

Ipotizzado che l'utente quindi ha 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 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'alabero:

BASSO LIVELLO

Fatta una panoramica delle funzioni ad alto livello possiamo preoccuparci di definire le funzioni piu 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.

Name	Value	Function	Device	List

SymRef:

```
struct symref {

  int nodetype;        /* tipo nodo N -> riferimento ad un simbolo*/

  struct symbol *s;

};
```

E' 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;

};
```

E' 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;

};
```

E' 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 strct ast *l identifica i parametri nodo che sono stati passati alla funzione **(chiedi conferma)** . Il terzo campo della struttura identifica la funzione che verrà eseguita tramite un tipo enumerativo. Le funzioni che sono state pensate (e defninte nel tipo enumerativo builFunc) sono le seguenti:

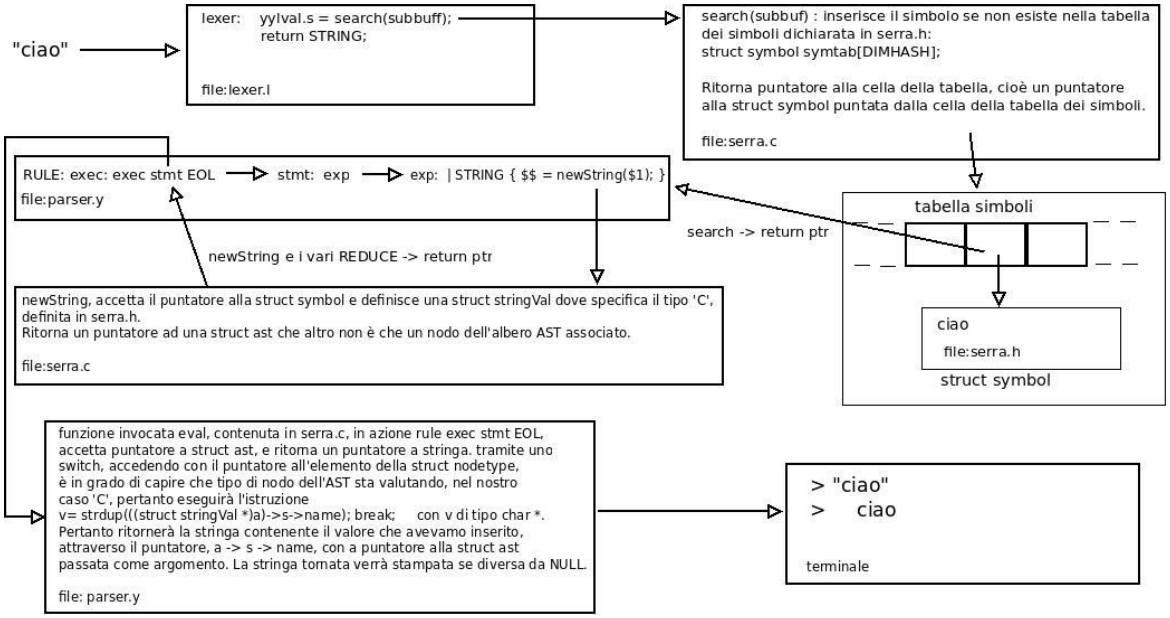
- B_print:
- B_connect: si occupa di effettuare il ping tra due dispositivi. A partire da un dispositivo poter visualizzare se l'altro è raggiungibile.
- B_reconnect = 3,
- B_status = 4,
- B_switchOn: setta lo stato del dispositivo a ON dopo aver controllato che sia effettivamente raggiungibile (con un ping di connect)
- B_switchOff = 6: setta lo stato del dispositivo a OFF
- B_diagnostic = 7
- B_archive = 8,
- B_interval = 9,
- B_insertDevice = 10

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 fuznioni. 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. Con tale funzione viene creato quindi un oggetto stringVal associandogli come parametri il simbolo creato e 'c'



>”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

	Tabella Simboli				
Indice (Hash+Name)	**Name**	**Value**	**Function**	**Device**	**List**
ciao20	ciao	0	NULL	NULL	NULL
123a1	123	0	NULL	NULL	NULL
	Nodo AST				
StringVal	C	ciao20			
NumVal	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:

- Clear: Si occupa---- (da vedere)

```
enum builtFuncSystem {
        B_clear = 1
};
```

- 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

E' 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: (Nel programma ancora non funziona, va in segmentation)*

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

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

>>Pippo10

	Tabella Simboli				
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
	Nodo AST				
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 allroa 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** 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 funzion **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';

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 puo 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 ogetto 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
	Ogetto 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
	Ogetto Device				
	NodeType	**Status**	**S (Symbol)**	**I (argList)**	
	D	0	Devicenuovo_c1	Null	
	D	0	Devicenuovo_d1	Null	

Una volta creati degli ast/ogetti 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 colleagto a ulteriori possibili device ("Device_nuovo_C") creati in precedenza. Cioè il campo 'I' dell'oggeto 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:

a. 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:

	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	NULL

b. ARROW (**newDevice Device_A ->**)è un token che specifica la ->
c. 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 che ritorna semplicemente una struttura argsList. 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): la funzione prende come parametri come parametri nel caso 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:

Ogetto 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}
	Ogetto Device				
	NodeType	**Status**	**S (Symbol)**	**l (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            { $$ = newContent('I', $2, $4, NULL); }
   | IF exp THEN listStmt ELSE listStmt  { $$ = newContent('I', $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**" { yylval.func = B_connect; return FUNCDEV; }
- "**reconnect**" { yylval.func = B_reconnect; return FUNCDEV; }
- "**status**" { yylval.func = B_status; return FUNCDEV; }
- "**switchOn**" { yylval.func = B_switchOn; return FUNCDEV; }
- "**switchOff**" { yylval.func = B_switchOff; return FUNCDEV; }
- "**diagnostic**" { yylval.func = B_diagnostic; return FUNCDEV; }
- "**archive**" { yylval.func = B_archive; return FUNCDEV; }
- "**interval**" { yylval.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 (da controllare se sarà cosi).

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 stampare 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 speiegate:

>CMD Device_H20(Device_diottrico) = newDevice DeviceB

>CMD Device_H20(Device_diottrico) = 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:

	Tabella Simboli				
Indice (Hash+Name)	**Name**	**Value**	**Function**	**Device**	**List**
DeviceB100	DeviceB	0	NULL	NULL	NULL
DeviceA100	DeviceA	0	NULL	NULL	{DeviceB100}
	Nodo AST				
Device	D	0	DeviceB100	Null	
Device	D	0	DeviceA100	{DeviceB100}	

L'utente adesso vuole effettivamente connettere i due device. Ipotizziamo quindi il seguente comando:

>CONNECT Device_A

Si avvierà quindi la seguente produzione:

exec stmt EOL , in cui:

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

```
FUNCDEV explistStmt {

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

}
```

   a. 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; }
   b. 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 sybol della

tabella dei simboli che fa riferimento al Device_A. Riconosciuto il token string verrà eseguita la funzione:
STRING { $$ = newString($1); }

La funzione newString creerà un nodo dell'albero di tipo StringVal 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:

	Tabella Simboli				
Indice (Hash+Name)	**Name**	**Value**	**Function**	**Device**	**List**
DeviceB100	DeviceB	0	NULL	NULL	NULL
DeviceA100	DeviceA	0	NULL	NULL	{DeviceB100}
	Nodo AST				Indirizzo Memoria
Device	D	0	DeviceB100	Null	
Device	D	0	DeviceA100	{DeviceB100}	
StringVal	C	DeviceA100			3FFFFFFF

Rivalutando la produzione di riferimento: $$ = newfunc($1, $2). Si ha che $1contiene 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** come nodo AST che conterrà i campi specificati di seguito:

	Nodo AST				Indirizzo Memoria
Device	D	0	DeviceB100	Null	
Device	D	0	DeviceA100	{DeviceB100}	
StringVal	C	DeviceA100			3FFFFFFF
funcBuiltIn	F	3FFFFFFF	"Connect"		

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

____disegno___

c. Ritornando alla produzione princiaple: 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 3FFFFF 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.

Errore da correggere sia in switchOn che connect:

- Connect be
- SwitchOn be ->segmentation