Capitolo 1

Linguaggio e semantica

Un linguaggio di programmazione é un linguaggio formale, ovvero un insieme di simboli e regole che definiscono la struttura e il significato delle istruzioni che compongono un programma. In questo capitolo definisco il linguaggio che andremo ad utilizzare per descrivere i programmi e la semantica che gli daremo. Il linguaggio é stato pensato per essere comprensibile e allo stesso tempo espressivo, in modo da poter descrivere programmi complessi. La semantica é stata pensata per essere semplice da implementare e allo stesso tempo potente, in modo da poter descrivere programmi complessi.
Un linguaggio viene definito da una grammatica $G = (N, T, \rightarrow, S)$ dove:

- N é l'insieme dei **non terminali**
- T é l'insieme dei **terminali**
- $\rightarrow$ é l'insieme delle **produzioni**
- S é il **simbolo iniziale**

Mentre il linguaggio generato da una grammatica G é definito come:

$$\mathcal{L}(G) = \{\gamma | \gamma \in T^* \wedge \Rightarrow^+ w\} \tag{1.1}$$

e T^* é la chiusura di Kleene.
La grammatica (che descriveremo nel dettaglio nel capitolo successivo), é riconosciuta dal plugin ANTLR che ci permette di generare un parser per il linguaggio. Il parser generato da ANTLR é un parser LL(*), ovvero un parser che riconosce linguaggi non ambigui e che non richiede backtracking, utilizzando un numero arbitrario di lookahead symbols, questo ci permette di avere un parser efficiente ottimizzando il riconoscimento del linguaggio.

1.1 Grammatica

1.1.1 Grammatica del linguaggio

Di seguito espongo la grammatica del linguaggio **HLCostLan** che descrive il linguaggio che utilizzeró per descrivere i programmi.
Riporto di seguito le produzioni della grammatica che descrivono il linguaggio, mentre per visualizzare il file g4 nella sua totalitá si rimanda alla Repository Git del progetto.

```
grammar HLCostLan;
prg : complexType* serviceDecl* functionDecl* init;

init: '('formalParams? ')''=''>' '{' stm '}';

serviceDecl:
    'service'ID':''('(type(','type)*)?')''→''type';

functionDecl:
    'fn'ID'('formalParams?')''→'(type)'{'stm'}' ;

stm :
    | serviceCall
    |'if''('expOrCall')''{'stm'}''else''{'stm'}'
    |'for''('ID'in''('NUMBER','exp ')'')''{'stm'}'
    | letIn
    | functionCall
    | 'return' expPlus ;

serviceCall:'call'ID'('(exp(','exp)*)?')'(';'stm)?;

functionCall : ID'('( exp (','exp)* )? ')';

letIn: 'let' (ID '=' expPlus)+ 'in' stm;
```

Listing 1.1: Grammatica del linguaggio HLCostLan

Il non terminale **prg** è il terminale iniziale della grammatica, e descrive un programma. Un programma è composto da una sequenza di dichiarazioni di tipi complessi, dichiarazioni di servizi(che possono avere un overhead in termini di invocazioni che influiscono sul costo), dichiarazioni di funzioni, e infine l'init. L'init è la funzione che viene invocata all'avvio del programma.

Il non terminale **init** descrive la funzione init, che deve essere dichiarata una sola volta ed è composta da una sequenza di parametri formali, e da una istruzioni.
La **serviceDecl** descrive la dichiarazione di un servizio, che deve essere dichiarato una sola volta. Un servizio è composto da un nome, una sequenza di parametri formali, e un tipo di ritorno.
La **functionDecl** descrive la dichiarazione di una funzione, una funzione è composta da un nome(univoco), una sequenza di parametri formali, e un tipo di ritorno. Nel checking semantico controlliamo che non vengano dichiarate due volte funzioni o servizi con lo stesso nome.
Inoltre il Type checking controlla che i tipi di ritorno delle funzioni e dei servizi siano coerenti con i tipi di ritorno delle chiamate.

1.2 Another Tool for Language Recognition

ANTLR (Another Tool for Language Recognition) é uno strumento potente e flessibile per l'analisi di linguaggi di programmazione, linguaggi di markup e dati strutturati. é ampiamente utilizzato per generare parser e lexer per vari linguaggi di programmazione e per costruire compilatori, interpreti, traduttori di linguaggi e altre applicazioni che necessitano di analisi di linguaggi. Ecco alcuni usi comuni di ANTLR:

1. Generazione di parser e lexer: ANTLR può generare parser e lexer per molti linguaggi di programmazione, rendendo più semplice l'analisi sintattica e lessicale.

2. Costruzione di compilatori e interpreti: ANTLR é spesso utilizzato nella costruzione di compilatori e interpreti, poiché fornisce gli strumenti per analizzare il codice sorgente e costruire l'albero di sintassi astratta.

3. Traduzione di linguaggi: ANTLR può essere utilizzato per tradurre codice da un linguaggio di programmazione a un altro. Questo é utile per la migrazione del codice, la refactoring e altre attivitá di manutenzione del software.

4. Analisi di dati strutturati: ANTLR può essere utilizzato per analizzare dati strutturati, come file XML o JSON, rendendo più semplice l'estrazione e la manipolazione dei dati.

ANTLR data una grammatica in input, con estensione *.g4*, genera una cartella *gen* contenente i file necessari per generare il parser.

1.2.1 EBNF: Simboli e notazioni in ANTLR

I simboli utilizzati da ANTLR, rispettano la notazione EBNF(Extended Backus-Naur Form), che é una notazione formale per descrivere la sintassi di un linguaggio, definita come standard internazionale da ISO-14977.

- () - Le parentesi vengono utizzate per raggruppare elementi diversi, e per definire l'ordine di valutazione.
- **?** - i Token seguiti da un punto interrogativo sono opzionali, e possono presentarsi 0 o 1 volta.
- * - Chiusura di Kleene, i token seguiti da un asterisco possono presentarsi 0 o piú volte.
- + - i token seguiti da un segno piú possono presentarsi 1 o piú volte.
- — - il simbolo pipe viene utilizzato per definire una scelta, ovvero un token o un'altro.
- **.** - Il punto definisce che un token puó apparire una singola volta.
- ∼ - Il simbolo not definisce che un simbolo non puó apparire.
- **..** - Il simbolo Range definisce un intervallo di token.

L'uso di lettere maiuscole e minuscole é importante, in quanto ANTLR distingue tra token maiuscoli e minuscoli.
[4]

1.2.2 Differenza tra Lexer e Parser in ANTLR

I linguaggi di programmazione sono costituiti da parole chiavi(keywords) e costrutti definiti in modo preciso. Un file sorgente viene inviato come flusso ad un lexer, carattere per carattere da una qualche interfaccia di input. Il lexer si occuppa di raggruppare i caratteri in token, ovvero sequenze di caratteri che rappresentano parole chiavi, identificatori, numeri, stringhe, e altri costrutti del linguaggio. Analogamente al lexer, il parser si occupa di riconoscere la grammatica del linguaggio, dandogli un qualche significato semantico. La cartella *gen* contiene il codice sorgente del parser e lexel, generati automaticamente; é presente anche un file *.tokens* che contiene i token riconosciuti dal lexer, e un file *.interp* che contiene la tabella di interpretazione del parser.

Il lexer si occupa di riconoscere i token, in ANTLR viene generato un lexer DFA, ovvero un lexer che riconosce linguaggi non ambigui e che non richiede backtracking. Questo ci permette di avere un lexer efficiente ottimizzando il riconoscimento del linguaggio. Il parser, invece, data una sequenza di token, riconosce la grammatica, e genera un albero di parsing(noto anche come albero di derivazione), che rappresenta la derivazione secondo le regole della grammatica libera da contesto. Nell'albero di parsing ogni nodo interno corrisponde a una regola della grammatica e ogni foglia corrisponde a un token del linguaggio.[2]

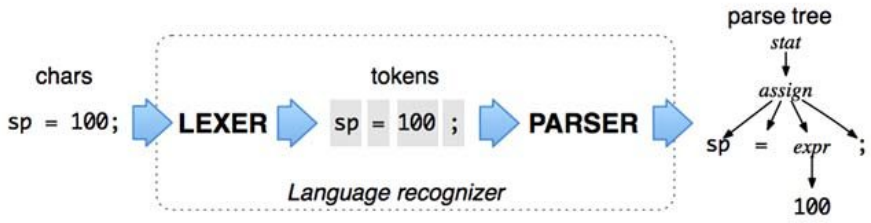

Figura 1.1: Funzione di parsing e lexing

Una volta ottenuto l'albero di parsing possiamo andare a visitarlo, e generare un albero di sintassi astratta, che rappresenta il significato del programma. L'albero di sintassi astratta(AST) é un albero che rappresenta il significato del programma, e viene utilizzato per eseguire il programma. A questo punto si estende l'interfaccia *BaseListener* per costruire un albero di sintassi astratta per implementare i metodi necessari per visitare l'albero di parsing e generare l'albero di sintassi astratta.[3]

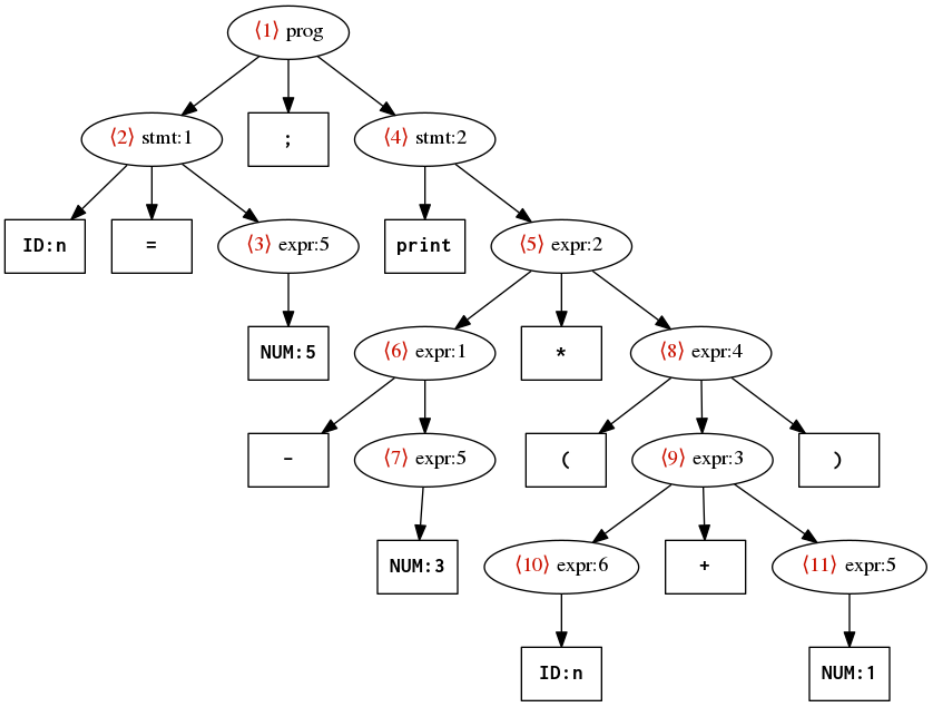

Figura 1.2: Esempio di albero di AST

Facciamo un esempio di un frammento di codice realmente generato da ANTLR, per la grammatica *HLCostLan*:

```
service BasicService: (int) -> void;
fn svc(i: int) -> void{
    for(m in (0,10)){
        call BasicService(i)
    }

}
(len : int) => {
  svc(len)
}
```

Listing 1.2: Esempio di codice HLCostLan: example/Listing6

Come abbiamo visto nella grammatica 1.1.1, il non terminale **prg** e' il terminale iniziale della grammatica. In questo caso prg, prende un parametro len di tipo intero, ed avrá un CallNode ad una funzione *svc* che prende un parametro di tipo intero, e ritorna void.
La funzione *svc* invece, contiene un for, che itera da 0 a 10, e ad ogni itera-

zione effettuata una chiamata al servizio *BasicService* con parametro *i*.

1.3 Semantica del Linguaggio

La semantica di un linguaggio di programmazione é l'insieme delle regole che definiscono il significato delle istruzioni, delle espressioni e delle strutture di controllo del flusso nel linguaggio. In altre parole, la semantica definisce "cosa fa" un programma scritto in quel linguaggio.

Ad esempio, se definiamo un linguaggio che permette l'assegnamento(non é il nostro caso perché la grammatica HLCostLan non lo permette) dovremmo andare a controllare che la variabile assegnata sia stata dichiarata altrimenti dovrá generare un errore.
I controlli semantici del compilatore in oggetto sono di altra natura, quali:

- Controllare che non vengano dichiarate due volte funzioni o servizi con lo stesso nome.
- Controllare che i parametri utilizzati nelle chiamate di funzioni o servizi siano corretti e quindi giá stati precedentemente dichiarati
- Controllare che i tipi di ritorno delle funzioni e dei servizi siano corretti.

Inoltre effettuiamo il type checking, che é un sottoinsieme del controllo semantico, che controlla che i tipi delle espressioni siano corretti.

- I tipi delle chiamate devono essere corretti, ovvero i parametri attuali devono essere dello stesso tipo dei parametri formali.
- In un costrutto *let* le espressioni devono essere dello stesso tipo della variabile a cui vengono assegnate.
- In un costrutto *if* l'espressione deve essere di tipo booleano, mentre i valori di ritorno degli statemente *then* e *else* devono essere dello stesso tipo.
- La funzione deve tornare il tipo dichiarato.

1.4 Compilatore o Interprete?

Nel processo di traduzione del codice di un determinato linguaggio in istruzioni eseguibili, emergono due approcci predominanti: il compilatore e l'in-

terprete. Mentre entrambi condividono l'obiettivo di rendere il codice sorgente eseguibile, le loro metodologie differiscono significativamente, portando a implicazioni distinte per le prestazioni, la portabilità e lo sviluppo delle applicazioni software.
Il compilatore adotta un approccio "one-shot", traducendo l'intero codice sorgente in un'unica fase e producendo un file eseguibile indipendente dal codice originale. Questo file può essere eseguito più volte senza la necessità di ulteriori traduzioni, garantendo una maggiore efficienza nell'esecuzione del programma. Tuttavia, il processo di compilazione può richiedere più tempo e risorse iniziali rispetto all'approccio interpretato, poiché l'intero codice deve essere tradotto prima dell'esecuzione.
D'altra parte, l'interprete adotta un approccio "line-by-line", traducendo e eseguendo istruzioni del codice sorgente una alla volta. Questo significa che il codice viene tradotto e eseguito in tempo reale durante l'esecuzione del programma. Sebbene questo approccio possa ridurre il tempo di avvio e la memoria richiesta per l'esecuzione, può comportare una minore efficienza durante l'esecuzione effettiva, poiché le istruzioni devono essere interpretate ad ogni esecuzione.
Nel nostro progetto, grazie anche al lexer e il parser di Antlr, abbiamo scelto di adottare un approccio ibrido, che combina le caratteristiche di entrambi i metodi. In particolare, utilizziamo un approccio "one-shot" per tradurre il codice per le cost equation, ma generiamo anche un codice intermedio come WAT, come vedremo nel paragrafo **??**. Questo approccio ci consente di ottenere i vantaggi di entrambi i metodi, garantendo al contempo un'efficienza e una flessibilità ottimali durante il processo di traduzione e esecuzione del codice, considerando anche i due file di output a cui si deve far fronte. [1]