

# **Appunti di Compilatori**

**Riccardo Lo Iacono**

Dipartimento di Matematica & Informatica  
Università degli studi di Palermo  
Sicilia  
a.a. 2023-2024

# Indice.

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Struttura di un compilatore . . . . .	2
<b>2</b>	<b>Analisi lessicale</b>	<b>3</b>
2.1	Gestione degli input . . . . .	3
<b>3</b>	<b>Analisi sintattica</b>	<b>4</b>
3.1	Algoritmo di Earley . . . . .	4
3.2	Gestione degli errori in un parser . . . . .	5
3.3	Top-Down parser . . . . .	5
3.4	Parser LL(1) . . . . .	7
3.5	Parser Bottom-Up . . . . .	10

## – 1 – Introduzione.

La necessità di semplificare la scrittura di codice sorgente, porta alla nascita dei primi linguaggi di programmazione. Ciò preclude quindi un “meccanismo” che permetta di descrivere il come e il cosa, si possa fare con un dato linguaggio. E, segue banalmente, che tale meccanismo può essere definito solo con un linguaggio esistente. Per questa e altre ragioni, nascono i compilatori e gli interpreti.

**Nota.** sebbene non di interesse ai fini del corso, a seguito si fa una breve digressione sugli interpreti.

A partire dal codice sorgente, un interprete converte, istruzione per istruzione, il sorgente che è immediatamente eseguito. Linguaggi di questo tipo sono *python*, *perl*, ecc.

Parlando ora dei compilatori, questi convertono il source-code in un codice macchina *equivalente*. Ulteriore compito dei compilatori è quello di segnalare eventuali errori.

**Nota.** esistono linguaggi (eg. JAVA) che fanno uso di compilatori ibridi: ossia compilatori che implementano sia la compilazione, sia l’interpretazione del sorgente.

### – 1.1 – Struttura di un compilatore.

La struttura di un compilatore può essere suddivisa in due parti: il “front-end” composto dalle fasi in *Figure 1.1*; e il “back-end” composto dalle fasi di generazione del codice.

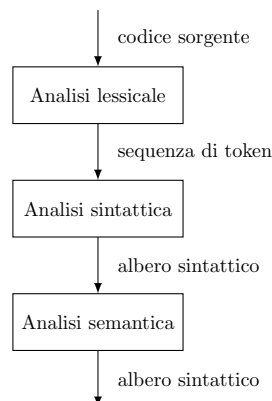


Figura 1.1: Struttura di un compilatore.

**Nota.** di interesse al corso risulta principalmente le fasi di front-end. Eventuali accenni alle altre fasi saranno discusse alla fine.

## – 2 – Analisi lessicale.

Come mostrato in *Figura 1.1*, l'analisi lessicale è la prima fase della compilazione. I suoi compiti sono sintetizzati a seguire.

1. Il sorgente è scansionato e da questi si compongono i *lessemi*: sequenze di caratteri con un determinato significato.

**Esempio:** un lessema per la gestione dei dati sarà del tipo: `t_dataType`.

2. Per ciascuno dei lessemi, un analizzatore sintattico genera dei token della forma `(token_name, address)`, successivamente gestiti dall'analisi sintattica.

Qui `token_name` identifica un lessema, mentre `address` è un puntatore alla cosiddetta *symbol table*. Quest'ultima, in breve, contiene le diverse proprietà di un'istanza di un lessema.

Per quel che riguarda lo scanner questi ha essenzialmente due compiti:

- costruire la *symbol table*;
- semplificare il sorgente.

Prima che ciò possa essere fatto però, è necessario, a meno che non sia stata eseguita una fase di precompilazione, che:

- vengano rimossi i commenti: come ovvio sono utili al solo programmatore, dunque, per alleggerir l'eseguibile, si procede alla loro rimozione;
- si effettui una case conversion: se il linguaggio non distingue tra maiuscole e minuscole, allora si converte il sorgente interamente in minuscolo;
- si rimuovano gli spazi: per motivi analoghi ai commenti, si elimina gli spazi superflui;
- si deve tenere traccia del numero di linea: ciò è utile per la segnalazione di eventuali errori.

**Nota.** sebbene in *Figura 1.1* sia mostrata come fase precedente l'analisi sintattica, più correttamente, l'analisi lessicale è da intendere come una sub-routine di quella sintattica.

**Nota.** per quel che riguarda l'implementazione della *symbol table*, ciò è generalmente realizzato con una *hash table*.

**Nota.** per rappresentare la struttura dei token, si fa uso delle espressioni regolari. Poiché si suppongono conoscenze pregresse, non saranno trattate.

### – 2.1 – Gestione degli input.

Lo scanner analizza il sorgente carattere per carattere, ma poiché un token potrebbe essere composto da più caratteri, è necessario un metodo di “backtracking”: cioè un modo per poter tenere traccia di dove un token inizi. Ciò è generalmente realizzato con un doppio buffering. Con l'ausilio di due puntatori, `forward` e `lexem_begin`, si procede ad identificare i token. Nello specifico inizialmente i due puntatori coincidono, successivamente si fa avanzare `forward` fintantoché si riscontra un lessema. Fatto ciò si aggiorna la posizione di `lexem_begin`.

## – 3 – Analisi sintattica.

Secondo step della compilazione è l'analisi sintattica, con la quale si verifica se il programma rispetta le regole sintattiche del linguaggio. Tali regole sono definite attraverso grammatiche context-free.

**Nota.** come per le espressioni regolari, si presuppongono conoscenze assodate sulle CFG. Per tale ragione non saranno trattate.

Definita la sintassi del linguaggio, è compito del parser verificare che ciascuno dei token generati dal lexer, possa effettivamente essere generato.

Parlando dei parser, se ne distinguono tre classi: top-down, i bottom-up e gli universali. A quest'ultima categoria appartengono gli algoritmi di *Cocke-Younger-Kasami* e quello di *Earley* a seguito descritto. Entrambi gli algoritmi appena citati sono, come tutti i parser universali, capaci di riconoscere qualsiasi CFG, ma per tale ragione risultano troppo inefficienti per scopi pratici.

### – 3.1 – Algoritmo di Earley.

Come detto Earley accetta qualsiasi CFG, più nello specifico: data  $x_1 \dots x_n$  una stringa, scandendo la stessa da sinistra a destra, per ogni  $x_i$  si costruiscono stati  $S_j$ , stati del tipo (dotted\_rule, address). Qui dotted\_rule sta ad indicare una produzione della grammatica alla cui destra è posto un punto, per tenere traccia della posizione della "sotto-stringa" esaminata. Con address si indica invece la posizione del punto.

**Esempio:** si supponga uno stato  $(A \rightarrow \alpha.\beta, i)$ . Ciò sta ad indicare che si è esaminata la sola sotto-stringa  $\alpha$ .

**Algoritmo di Earley.** Si riporta di seguito lo pseudo codice per implementare Earley.

```

input =  $x_1 \dots x_n$ 
 $x_{n+1} = \$$ 
for j = 0 to n do
  foreach state in  $S[j]$  choose between: scanner,
    predictor, completed
if  $S[n+1] == \$$ 
  accept()
refuse()

```

#### – 3.1.1 – Scanner.

Come detto la scansione (o Scanner) è una delle tre possibili operazioni di Earley. Circa l'operazione in se: sia  $(A \rightarrow \alpha.\beta, i) \in S_j$ . Se  $\beta = a\beta'$ , con  $a$  carattere terminale, quel che si fa è aggiungere a  $S_{j+1}$  lo stato  $(A \rightarrow \alpha.\beta, i)$ .

#### – 3.1.2 – Predictor.

Altra possibile operazione di Earley, è utilizzata nel caso in cui la sotto-stringa  $\beta$  inizi con un carattere non terminale. Cioè, supposto  $(A \rightarrow \alpha.\beta, i) \in S_j \wedge \beta = B\beta'$ , allora  $\forall B \rightarrow \gamma$  si aggiunge ad  $S_j$  uno stato  $(B \rightarrow \gamma, j)$ .

#### – 3.1.3 – Completed.

Ultima delle operazioni possibili, è utilizzata nel caso  $\beta = \varepsilon$ . Nello specifico, supposto  $(A \rightarrow \alpha.\beta, i) \in S_j \wedge \beta = \varepsilon$ , per ogni  $(C \rightarrow \eta A.\delta, h) \in S_i$  si aggiunge ad  $S_j$  uno stato  $(C \rightarrow \eta A.\delta, h)$ .

#### – 3.1.4 – Complessità di Earley.

La complessità dell'algoritmo è strettamente legata alla grammatica. Si ha infatti che se la grammatica identifica linguaggi REG, Earley impiega  $\mathcal{O}(n)$ ; se la grammatica è non ambigua  $\mathcal{O}(n^2)$ , mentre per qualsiasi altra grammatica  $\mathcal{O}(n^3)$ .

### – 3.2 – Gestione degli errori in un parser.

Affinché la compilazione sia corretta, è necessario che un parser sia in grado di scoprire, diagnosticare e correggere efficientemente gli errori, così da riprendere l'analisi quanto prima.

Si consideri ora il caso in cui un parser abbia rilevato un errore, resta il problema di come procedere per risolverlo. In generale si utilizza una delle seguenti tecniche.

- **Panic mode:** l'idea è quella di saltare simboli fintantoché non si legge un token di sincronizzazione (eg. begin-end). L'efficacia dipende fortemente dalla scelta dei token, scelta che può essere effettuata euristicamente.
- **Phrase level:** si fa in modo che il parser proceda a correzioni locali. Per far ciò, inevitabilmente si procederà ad alterare lo stack.

### – 3.3 – Top-Down parser.

Come suggerito dal nome, si tratta di parser che verificano la generabilità dell'input a partire dall'assioma. Si osservi però che tali parser soffrono di un problema: non sono deterministici. Banalmente, motivo di ciò è dato dal fatto che un non terminale possa produrre delle derivazioni, nel seguito indicate come  $\Rightarrow^*$ , che iniziano con uno stesso carattere.

Prima di descrivere il principale dei parser top-down, si descrivono due tecniche in genere utilizzate per eliminare il non determinismo.

1. Si supponga il caso di un non terminale che ha un prefisso comune a diversi terminali. Cioè si ha qualcosa del tipo

$$A \rightarrow \gamma\alpha_1 : \gamma\alpha_2 : \dots : \gamma\alpha_n : \omega$$

con  $\gamma \in T, \alpha_i \in N \cup T, \forall i = 0, \dots, n$ .  $T$  insieme di terminali,  $N$  dei non terminali.

Per risolvere il problema si introduce un nuovo non terminale, così da posticipare la scelta. Cioè, la grammatica di cui sopra diventa

$$\begin{aligned} A &\rightarrow \gamma B : \gamma B : \dots : \gamma B : \omega \\ B &\rightarrow \gamma\alpha_1 : \dots : \alpha_n \end{aligned}$$

2. Si supponga ora che un non terminale presenti una ricorsione sinistra. Si ha cioè qualcosa del tipo

$$A \rightarrow A\alpha_1 : A\alpha_2 : \beta_1 : \beta_2$$

considerando unicamente il caso in cui la ricorsione sia immediatamente a sinistra, come prima si introduce un nuovo non terminale così da ritardare la scelta. Dalla grammatica di sopra si ottiene dunque qualcosa del tipo

$$\begin{aligned} A &\rightarrow \beta_1 B : \beta_2 B \\ B &\rightarrow \alpha_1 B : \alpha_2 B : \varepsilon \end{aligned}$$

Circa i parser top-down in se, si distinguono

- i parser *a discesa ricorsiva*: di poco interesse al corso;
- i parser  $LL(k)$ : sono parser che analizzano l'input da sinistra a destra, costruendo una derivazione sinistra sulla base dei  $k$  simboli successivi.

**Nota.** saranno considerati parser LL(1).

### – 3.4 – Parser LL(1).

Come anticipato, i parser LL(k) sono parser che analizzano l'input da sinistra verso destra, effettuando derivazioni sinistre sulla base dei k simboli successivi.

Considerando il parser in se, questi deve la propria efficacia alla costruzione di una matrice/tabella, strutturata come segue:

- ogni carattere non-terminale rappresenta una riga della tabella, i terminali le colonne;
- a partire dall'assioma, per ogni regola del tipo  $X \rightarrow \gamma$  tale per cui  $\gamma \Rightarrow^* tB$ , è inserita in posizione  $(X, t)$ ;
- ogni regola del tipo  $X \rightarrow \gamma$  tale che  $\gamma \Rightarrow^* \varepsilon$ , è inserita in  $(X, t), \forall t : S \Rightarrow^* \beta X t a$ .

**Nota.** le ultime due “regole” sono implementate tramite le funzioni FIRST e FOLLOW, descritte nel seguito.

#### – 3.4.1 – FIRST.

Come detto si tratta di una delle funzioni utilizzate per la costruzione della tabella. Nello specifico, assunto una stringa  $\alpha \in N \times T^1$ , si ha che

$$FIRST(\alpha) = \{x \in T : \alpha \Rightarrow^* x\beta\}$$

In altri termini,  $FIRST(\alpha)$  rappresenta l'insieme dei terminali tali che gli stessi compaiano come primo termine di una qualche derivazione di  $\alpha$ . Inoltre se  $\alpha \Rightarrow^* \varepsilon$  allora  $\varepsilon \in FIRST(\alpha)$ .

**Algoritmo.** Per quel che concerne l'algoritmo per l'implementazione di FIRST, questi è il seguente.

```

for all terminal  $x$  and  $\varepsilon$  do
  FIRST(x) = {x};
for all non-terminal  $X$  do
  FIRST(X) =  $\emptyset$ ;
while there are changes in any FIRST do
  for each  $X \rightarrow Y_1, \dots, Y_k$  do {
    i = 1;
    continue = true;
    while continue == true AND i <= k do {
      add  $FIRST(Y_i) \setminus \{\varepsilon\}$  to FIRST(X);
      if  $\varepsilon \notin FIRST(Y_i)$ 
        continue = false;
      i += 1;
    }
    if continue == true
      add  $\varepsilon$  to FIRST(X);
  }
}

```

#### – 3.4.2 – FOLLOW.

Per varie ragioni, FIRST non è sufficiente alla realizzazione di un parser deterministico quale sono gli LL(1), per tale ragione è necessaria l'implementazione di FOLLOW.

**Algoritmo.** Per quel che concerne l'implementazione di FOLLOW, questa è riportata a seguito.

```

FOLLOW(AXIOM) = {$};
for each non-terminal  $X$  AND  $X \neq AXIOM$  do
  FOLLOW(X) =  $\emptyset$ ;
  while there are changes to any FOLLOW do
    for each  $X \rightarrow Y_1, \dots, Y_k$  do

```

---

<sup>1</sup>Rispettivamente, insieme dei non-terminali e dei terminali.



```
for each  $Y_i$  do {  
  add  $\text{FIRST}(Y_{i+1}, \dots, Y_k) \setminus \{\varepsilon\}$  to  $\text{FOLLOW}(Y_i)$ ;  
  if  $\varepsilon \in \text{FIRST}(Y_{i+1}, \dots, Y_k)$  then  
    add  $\text{FOLLOW}(X)$  to  $\text{FOLLOW}(Y_i)$   
}
```

### – 3.4.3 – Costruzione della tabella.

Si considera come costruire la tabella a supporto di LL(1), sfruttando le due operazioni precedentemente descritte. Sia in tale contesto  $M$  la matrice da costruire, allora questa sarà realizzata come segue:

- per ogni regola  $X \rightarrow \alpha$ , e per ogni  $a \in FIRST(\alpha)$ , si aggiunge  $X \rightarrow \alpha$  in  $M[X, a]$ ;
- per ogni regola  $X \rightarrow \alpha$ , se  $\varepsilon \in FIRST(\alpha)$ ,  $\forall b \in FOLLOW(X)$ , si aggiunge  $X \rightarrow \alpha$  in  $M[X, b]$ . Se  $\varepsilon \in FIRST(\alpha)$  e  $\$ \in FOLLOW(X)$ , si inserisce in  $M[X, \$]$  la regola.

### – 3.4.4 – Grammatiche LL(1) e loro parsing.

**Definizione:** sia  $G$  una grammatica, questa è detta essere LL(1) se e solo se  $\forall A \rightarrow \alpha : \beta$  si ha che:

- $\alpha\varepsilon\beta$  non derivano stringhe che iniziano con uno stesso terminale  $a$ . Cioè  $FIRST(\alpha) \neq FIRST(\beta)$ .
- al più uno dei due deriva  $\varepsilon$ .
- se  $\beta \Rightarrow^* \varepsilon$ , allora  $\alpha$  non deriva stringhe che iniziano con terminali in  $FOLLOW(A)$ .  
O viceversa.

**Algoritmo di parsing per LL(1).** Dalla definizione segue la possibilità di realizzare un parser predittivo per le grammatiche LL(1), il cui pseudo-codice è di seguito riportato.

```
LL_1_parser(stack p, input i, LL_1_table M, axiom S):
    error = false;
    p.push(\textdollar);
    p.push(S);
    while ( () p.top() != $ AND i.top() != $ AND !error ) do {
        if isTerminal(p.top()) {
            if p.top() == i.top() {
                p.pop();
                i.next();
            }
            error = true;
        }
        if isEmpty(M[p.top(), i.top()])
            error = true;
        else {
            p.pop();
            for (j = n; j > 0; j--)
                p.push( X_j )
        }
    }
    if (!error)
        accept();
    raise_error();
```

Per struttura delle grammatiche LL(1) e da un'analisi dell'algoritmo, si osserva che il parser LL(1) ha complessità  $\mathcal{O}(n)$ , con  $n$  lunghezza dell'input.

### – 3.5 – Parser Bottom-Up.

Un parser ascendente (o bottom-up), è un parser che a partire dalla stringa di input, tenta di risalire all'assioma. La definizione di “bottom-up”, deriva dal fatto che la costruzione dell'albero sintattico procede dalle foglie verso la radice.

**Nota.** Tutti i parser bottom-up procedono cercando derivazioni right-most della grammatica.

**Esempio:** sia supposta la seguente grammatica

$$E \rightarrow E : E + T$$

$$T \rightarrow \text{int} : (E)$$

e si supponga di dover analizzare  $\text{int} + (\text{int} + \text{int} + \text{int})$ . Di interesse risulta verificare che tale stringa sia generabile con la grammatica di cui sopra. Osservando la stringa in input si nota che

$$\begin{aligned} & \text{int} + (\text{int} + \text{int} + \text{int}) \\ &= T + (\text{int} + \text{int} + \text{int}) \\ &= E + (\text{int} + \text{int} + \text{int}) \\ &= E + (E + \text{int} + \text{int}) \\ &= E + (E + T + \text{int}) \\ &= E + (E + \text{int}) \\ &= E + (E + T) \\ &= E + (E) \\ &= E + T \\ &= E \end{aligned}$$

per cui si conclude, essendo risaliti all'assioma, che la stringa è derivabile.

#### – 3.5.1 – Parser shift-reduce.

Si tratta di un parser che fa uso di una pila di supporto, pila che inizialmente contiene unicamente il simbolo \$. ad indicare appunto la pila vuota. Ricevuto un input, anch'esso contenente \$ ad indicarne la fine, sulla base dello stesso effettua quattro operazioni.

- **Shift:** un terminale della stringa in input, è spostato nella pila.
- **Reduce:** una stringa  $\alpha$  in cima alla pila è sostituita con un  $A \in N$ , secondo la regola  $A \rightarrow \alpha$ .
- **Error:** comunica la presenza di un errore sintattico.
- **Accept:** la pila è vuota (si ha solo il simbolo \$) e la lettura è terminata, procede ad accettare la stringa.

#### – 3.5.2 – Grammatiche LR e parsing LR(k).

Il parser shift-reduce descritto precedentemente, potrebbe non essere sufficiente a riconoscere alcune grammatiche. Per tale ragione, in generale, si preferisce utilizzare i parser LR(k), con “k” numero di caratteri successivi di cui tenere traccia ad ogni passo.

**Nota.** Saranno trattati nel dettaglio i parser LR(0). Altri parser LR (eg: SLR e/o LALR(1)) saranno solamente accennati.

**Osservazione.** Se una grammatica è ambigua, non è LR(k) per nessun k.

– 3.5.2.1 – Costruzione della tabella LR(0).

Similarmente per il parser LL(1), per indicare a che punto una produzione è stata analizzata, si aggiunge in quella posizione un punto.

**Esempio:** si supponga la seguente produzione:  $A \rightarrow XYZ$ ; in questo caso le posizioni distinte in cui è possibile trovarsi sono

$$A \rightarrow .XYZ \quad A \rightarrow X.YZ \quad A \rightarrow XY.Z \quad A \rightarrow XYZ.$$

ognuna delle “produzioni” di cui sopra è detta *item*.