Appunti di Compilatori

Riccardo Lo Iacono

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

Indice.

– 1 – Introduzione: interpreti e compilatori.

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.

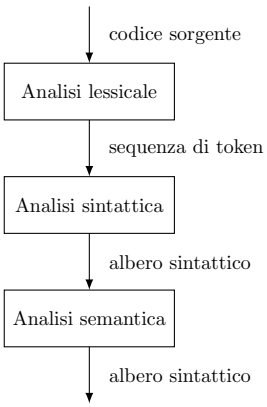

Figura 1.1: Struttura di un compilatore.

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 alleggeri 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 per tenere traccia della posizione della "sotto-stringa" esaminata. Con address si indica invece la posizione del punto.

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

L'algoritmo in se è il seguente.

```
input = x_1 ... x_n
x_{n+1} = $
for j = 0 to n do
    per ogni stato di S[j] scegli
        scansione, predizione, completamento.
if S[n + 1] == $
    accetta
rifiuta
```

– 3.1.1 – Scanner.

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

– 3.1.2 – Predictor.

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

– 3.1.3 – Completed.

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

– 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 $\Longrightarrow^*$, 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

   $$A \to \gamma\alpha_1 \mid \gamma\alpha_2 \mid \cdots \mid \gamma\alpha_n \mid \omega$$

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

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

   $$A \to \gamma B \mid \gamma B \mid \cdots \mid \gamma B \mid \omega$$
   $$B \to \gamma\alpha_1 \mid \cdots \mid \alpha_n$$

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

   $$A \to A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$$

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

   $$A \to \beta_1 B \mid \beta_2 B$$
   $$B \to \alpha_1 B \mid \alpha_2 B \mid \varepsilon$$

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).