COMPILATORI

APPUNTI A CURA DI: RICCARDO LO IACONO

Università degli studi di Palermo
a.a. 2023-2024

Indice.

– 1 – Introduzione.

Con lo svilupparsi dei linguaggi di programmazione, si sono sviluppati parallelamente gli *interpreti* e i *compilatori*. Questi ultimi, la cui struttura principale è mostrata in *Figura 1*, permettono di descrivere il come e il cosa si possa fare con il linguaggio

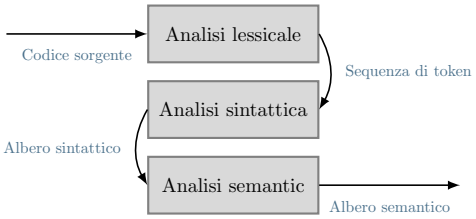

Figura 1: Struttura di un compilatore.

che essi definiscono. Nello specifico, un compilatore converte il codice sorgente in un codice macchina equivalente, in aggiunta al segnalare eventuali errori. Per quel che riguarda gli interpreti, questi convertono istruzione per istruzione il sorgente e lo eseguono immediatamente. Tra i linguaggi di questo tipo: *python, perl, ecc.*
Osservazione. La struttura di *Figura 1*, è limitata alle fasi di interesse del corso; a seguire la fase di analisi sintattica, vi sono delle ulteriori fasi che non risultano di interesse.

– 1.1 – Richiami alle RegEx e alle grammatiche.

Poiché i concetti di Regex e grammatiche CF sono alla base della definizione di un compilatore, si riprende a seguito la definizione delle stesse.

Definizione: Si definisce *espressione regolare*, (o RegEx), la descrizione algebrica delle stringhe di un dato linguaggio.

In particolare, la costruzione di una RegEx e è di tipo ricorsivo. Si ha infatti che

- ε e $\varnothing$ sono espressioni regolari, ove $L(\varepsilon) = \{\varepsilon\}, L(\varnothing) = \{\}$;
- se α è un simbolo, allora questi è una RegEx, ove $L(\alpha) = \{\alpha\}$;

da queste

- se e ed f sono due RegEx. Allora $e + f$ è un'espressione regolare;
- se e ed f due RegEx. Allora ef è un'espressione regolare;
- se e RegEx. Allora e^* è un'espressione regolare;
- se e RegEx. Allora (e) è un'espressione regolare.

Definizione: Dato T un certo alfabeto, si definisce grammatica G la seguente quadrupla:

$$G = (T, N, S, P).$$

Ove

- T è l'alfabeto di simboli terminali;
- N è l'alfabeto dei simbolo non terminali;
- S è un simbolo non terminale detto assioma;
- P è l'insieme delle regole di produzione.

– 2 – Analisi lessicale.

Come mostrato in *Figura 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.

Sebbene in Figura 1 *sia mostrata come fase precedente l'analisi sintattica, più correttamente, l'analisi lessicale è da intendere come una sua sub-routine.*

L'implementazione della symbol table, sebbene realizzabile diversamente, in generale è realizzata tramite hash-table.

– 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.

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

Si riporta di seguito lo pseudo codice per implementare Earley.

```
input = x_1 ... 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()
```

Figura 2: Pseudo codice Earley.

– 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. Sia supposto un parser che 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

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

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. Nello specifico saranno trattati i parse 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 $X \to \gamma$ tale per cui $\gamma \Rightarrow^* tB$, è inserita in posizione (X, t);
- ogni regola del tipo $X \to \gamma$ tale che $\gamma \Rightarrow^* \varepsilon$, è inserita in $(X, t), \forall t : S \Rightarrow^* \beta X t a$.

– 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 α. Inoltre se $\alpha \Rightarrow^* \varepsilon$ allora $\varepsilon \in FIRST(\alpha)$. Per quel che concerne l'algoritmo per l'implementazione di FIRST, questi è il seguente.

```
for all terminal x and ε do
    FIRST(x) = {x};
for all non-terminal X do
    FIRST(X) = ∅;
while there are changes in any FIRST do
    for each X → Y1,...,Yk do {
        i = 1;
        continue = true;
        while continue == true AND i <= k
            do {
            add FIRST(Yi) \ {ε} to FIRST(X);
            if ε ∉ FIRST(Yi)
                continue = false;
            i += 1;
        }
        if continue == true
            add ε to FIRST(X);
     }
```

Figura 3: Pseudo codice procedura FIRST.

[1] Rispettivamente, insieme dei non-terminali e dei terminali.

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

```
FOLLOW(AXIOM) = { $ };
for each non-terminal X AND X ≠ AXIOM do
    FOLLOW(X) = ∅;
    while there are changes to any FOLLOW do
        for each X → Y_1,...,Y_k do
            for each Y_i do {
                add FIRST(Y_{i+1},...,Y_k)
                    \{ε} to FOLLOW(Y_i);
                if ε ∈ FIRST(Y_{i+1},...,Y_k) then
                    add FOLLOW(X) to FOLLOW(
                        Y_i)
            }
```

Figura 4: Pseudo codice procedura FOLLOW.

– 3.4.3 – Grmmatiche LL(1) e loro parsing.

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

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

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

```
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();
```

Figura 5: Pseudo codice parse LL(1).

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.

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