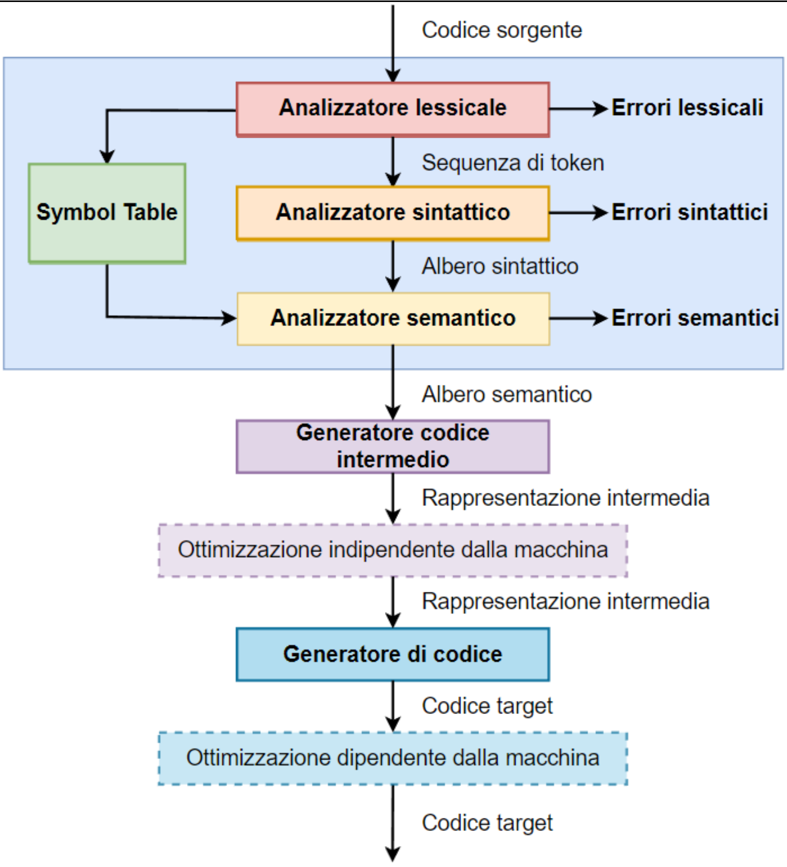
SOMMARIO COMPILATORI – PARSER

2022/2023



EARLEY ALGORITHM

L'idea è quella di costruire progressivamente tutte le possibili derivazioni leftmost (o rightmost) compatibili con la stringa in input.

Durante il procedimento si analizza la stringa in input da sinistra verso destra scartando via via le derivazioni in cui non vi sia corrispondenza tra i simboli derivati e quelli della stringa.

Se esistono due derivazioni leftmost (o rightmost) possibili, l'algoritmo restituisce i due alberi di derivazione.

Uno stato è costituito da un insieme di coppie, descritte da:

$$X = (Dotted\_rule,\ Pointer)$$

- $\boldsymbol{Dotted\_rule}$, è una produzione di G avente sul lato destro un punto che ne marca la posizione;
- $\boldsymbol{Pointer}$, è un intero che indica la posizione dell'input a partire dalla quale è iniziato l'esame della produzione.

Per la costruzione della tabella di Earley i passi sono:

- Partendo dall'assioma $S \to E\$$, si pone nella prima colonna/stato S_0 la coppia $(S \to .E\$, 0)$;
- Si scansiona l'insieme delle coppie dello stato con il simbolo della stringa di input a e si applica una delle seguenti azioni:
  - **Scanner**, se c'è una coppia X_i che nella $Dotted\_rule$ ha come successivo il <u>terminale</u> a, allora aggiungi al <u>prossimo</u> stato la coppia X_i con la $Dotted\_rule$ avanti di un posto e lo stesso $Pointer$. Altrimenti nulla;
  - **Predictor**, se c'è una coppia X_i che nella $Dotted\_rule$ ha come successivo un <u>NON terminale</u> E, allora aggiungi nello stato <u>corrente</u> tutte le produzioni di E come coppie X_k con $Dotted\_rule$ all'inizio e $Pointer$ pari allo stato corrente.
  - **Completer**, se c'è una coppia X_i che nella $Dotted\_rule$ ha come successivo il <u>simbolo spontaneo</u> ε, allora aggiungi nello stato <u>corrente</u> tutte le coppie precedenti, con la $Dotted\_rule$ avanti di un posto e il max $Pointer$, che hanno come successivo il simbolo che genera X_i.
- Si iterano queste azioni finché non si genera una tabella lunga $|input\_string| + 2$;

FIRST & FOLLOW

Sono due funzioni, utilizzate sia dai parser top-down che dai parser bottom-up, associate a una grammatica G.
In particolare nel parser top-down permettono di effettuare una scelta fra diverse produzioni in base al solo simbolo di input immediatamente successivo alla posizione della testina di lettura.

- $\boldsymbol{First(X)}$, è l'insieme dei simboli <u>terminali</u> che si trovano all'inizio delle stringhe derivabili da X.

$$X \to aA|bB|cC, \qquad First(X) = \{a, b, c\}$$

In particolare, sia $X \to x$:

1. Se x è un simbolo **terminale** a, allora $First(x) = \{a\}$;
2. Se x è un simbolo **NON terminale** A, allora $First(x) = First(A)$;
3. Se x è il **simbolo spontaneo** ε, allora $First(x) = \{\varepsilon\}$;
4. Se x è un insieme $First(x_1), \dots, First(x_k)$ e tutti questi contengono ε, allora anche $First(x)$ ha ε.

- $\boldsymbol{Follow(X)}$, è l'insieme dei simboli <u>terminali</u> che possono apparire immediatamente a destra di X.

$$S \to Xa, \qquad Follow(X) = \{a\}$$

In particolare, sia $X \to x$:

1. Se x è l'**assioma** X, allora $Follow(x) = \{\$\}$;
2. Se x è una produzione xa, con a **terminale**, allora $Follow(x) = First(a)$;
3. Se x è una produzione xA, con A **NON terminale**, allora $Follow(x) = First(X) \cup Follow(X)$
4. Se x è una produzione x (oppure xy, con y **simbolo spontaneo** ε), allora $Follow(x) = Follow(X)$;

PARSER $\boldsymbol{LL(1)}$

Una **grammatica** in cui ogni casella della tabella di parsing non contiene più di un elemento è detta $LL(1)$.

<u>Osservazione 1</u>: per costruzione una grammatica $LL(1)$ non è ambigua, né ricorsiva sinistra.
Quindi in questi casi la grammatica non può essere $LL(1)$.

<u>Osservazione 2</u>: Per stabilire se una grammatica è $LL(1)$ bisogna costruire la tabella e verificare che in ogni casella ci sia al più un elemento.

Una grammatica si dice $LL(1)$ se e solo se per ogni produzione del tipo $A \to \alpha \mid \beta$ si ha:

- α e β non derivano stringhe che cominciano con lo stesso simbolo x;
- Al più uno tra i due può derivare la stringa vuota;
- Se $\beta \to * \, \varepsilon$ allora α non deriva stringhe che cominciano con terminali che stanno in $Follow(A)$. Analogamente per α.

Equivalentemente affinché una grammatica sia $LL(1)$, per ogni coppia di produzioni $A \to \alpha \mid \beta$:

- $First(\alpha)$ e $First(\beta)$ devono essere disgiunti;
- Se ε è in $First(\beta)$ allora $First(\alpha)$ e $Follow(A)$ devono essere disgiunti.

Per la costruzione della tabella del parser $LL(1)$ i passi sono:

- Fattorizzazione a sinistra e disambiguazione della grammatica;
- Calcolo dei $First()$ e dei $Follow()$ dei simboli NON terminali;
- Creazione della tabella con righe i NON terminali e colonne i terminali.

SHIFT & REDUCE

Sono due operazioni principali utilizzati nei parser bottom-up. Data una grammatica non ambigua G si fa uso di una pila, che avrà inizialmente il simbolo di fondo pila \$ e, leggendo carattere per carattere l'input, si effettuano le operazioni:

- $\boldsymbol{Shift}()$, il successivo simbolo terminale della stringa di input viene spostato nella pila;
- $\boldsymbol{Reduce}()$, una stringa (o sottostringa) x in cima alla pila è stata ridotta al NON terminale X.

In supporto alle due operazioni, nel parser shift-reduce (**parser SR**), si hanno:

- $\boldsymbol{Error}$, errore di sintassi;
- $\boldsymbol{Accept}$, la pila ha in cima solo l'assioma S (esattamente $\$S$) e la lettura dell'input è finita.

PARSER $\boldsymbol{LR(0)}$

In genere nei parser SR si hanno i problemi di conflitti tra $Shift - Reduce$ e $Reduce - Reduce$.
Per risolvere questi due casi basta semplicemente guardare un simbolo avanti dell'input.

Esistono algoritmi che guardano avanti di k simboli, sono i cosiddetti $LR(k)$.
Le grammatiche ambigue non possono essere $LR(k)$ per nessun k.

I parser LR sono più convenienti per: riconoscere virtualmente tutti i costrutti di linguaggi di programmazione definiti da grammatiche CF, è il più generale parsing SR che NON richiede backtraking, contiene propriamente le grammatiche LL; infatti, un parser $LR(k)$ deve riconoscere l'occorrenza di una parte destra di una produzione in una forma sentenziale destra con k simboli di prospezione, un parser $LL(k)$ deve scegliere una produzione in base ai primi k simboli della stringa da derivare.

Una grammatica è LR quando un parser $Shift - Reduce$ è in grado di riconoscere le Handle quando appaiono in cima allo stack (l'informazione è contenuta nella tabella). Un parser LR non deve guardare tutto lo stack: lo stato che in ogni momento si trova in testa ad esso contiene tutta l'informazione di cui il parser ha bisogno.

Se è possibile riconoscere una Handle guardando solo i simboli della grammatica che sono sullo stack allora esiste un automa finito che, leggendo i simboli dello stack, determina se una certa handle è presente in testa.

Lo stato in testa allo stack è esattamente lo stato in cui l'automa si troverebbe se lo si facesse partire dallo stato iniziale dopo aver letto la stringa composta dai simboli dello stack che si trovano dal fondo alla testa.

Per la costruzione dell'automa del parser $LR(0)$ i passi sono:

- Estensione della grammatica G a G' con la regola $S' \to S$;
- Chiusura dello stato S';
- Per ciascun simbolo dopo il puntatore in una chiusura, si crea una nuova chiusura in cui se il nuovo simbolo:
  - È ε, non fare nulla;
  - È un terminale, non fare nulla;
  - È un NON terminale, aggiungi i suoi rispettivi stati con il puntatore all'inizio;
- Le una chiusura crea un'altra chiusura uguale ad una esistente, si usa quest'ultima;
- Le chiusure che non sono del tipo $X \to x.$ si possono "dimenticare".

Per la costruzione della tabella del parser $LR(0)$ i passi sono:

- Per ogni arco (i, j) dell'automa etichettato con un simbolo terminale a, si inserisce $Shift(j)$ in $M[i, a]$;
- Per ogni arco (i, j) dell'automa etichettato con un NON terminale X, si inserisce $GoTo(j)$ in $M[i, X]$;
- Per ogni stato i contenente $X \to x.$ (tale che $X \to x$ sia la produzione k-esima della grammatica), si inserisce in tutta la riga i l'azione $Reduce(k)$;
- Per ogni stato i contenente $S' \to S.$ si inserisce $Accept$ in $M[i, \$]$

PARSER $SLR(1)$

In quanto i parser $LR(0)$ hanno uso solo per scopo didattico, e la sua versione $LR(1)$ per quanto potente è poco controllabile nell'implementazione, si passa ai parser $SLR(1)$ (Simple LR) che considerano esclusivamente un simbolo di Lookahed e gli elementi di $Follow()$.

Una grammatica G si dice SLR se ogni sua casella contiene al più una sola regola.

Tutte le grammatiche SLR non sono ambigue; tuttavia, esistono grammatiche non ambigue che non sono SLR.

Per la costruzione dell'automa del parser $SLR(1)$ i passi sono:

- Estensione della grammatica G a G' con la regola $S' \to S$;
- Costruire le chiusure come nel parsing $LR(0)$ e considerare i sottoinsiemi di tutti le chiusure;
- Calcolare il $Follow()$ di tutti i simboli <u>NON terminali</u>;

Per la costruzione della tabella del parser $SLR(1)$ i passi sono:

- Per ogni stato $X \to x.a$ appartenente alla chiusura I_i, si inserisce $Shift(j)$ in $M[i,a]$;
- Per ogni stato $X \to x.$ appartenente alla chiusura I_i, si inserisce $Reduce(X \to x.)$ a ogni terminale in $Follow(X)$;
- Per ogni arco (i,j) dell'automa etichettato con un NON terminale X, si inserisce $GoTo(j)$ in $M[i,X]$;
- Per lo stato $S' \to S$ appartenente alla chiusura I_i, si inserisce $Accept$ in $M[i,\$]$.

PARSER $LR(1)$

Consente di ovviare a molte ambiguità dei parser $LR(0)$ al prezzo di una crescita sostanziale della complessità dell'algoritmo. Il metodo $LR(1)$ è poco usato in pratica proprio perché poco efficiente:

- Si preferisce il più semplice $LALR(1)$;
- L'automa riconoscitore per i parser $LR(1)$ è simile agli automi $LR(0)$: cambiano gli item e le operazioni $Closure()$, $GoTo()$ e $Reduce()$.

L'idea è che nei parser $LR(1)$ il lookahead è utilizzato durante la costruzione dell'automa (quindi si prendono in considerazione i simboli che veramente possano seguire un certo handle).

Per la costruzione dell'automa del parser $LR(1)$ i passi sono simili a quello del parser $LR(0)$, con la differenza che:

- Gli stati sono descritti nella forma:

$$(Dotted\_rule,\ Token)$$

  Dove $Token$ è un simbolo <u>terminale</u> x oppure il <u>simbolo di fine</u> \$.

- I $Token$ degli stati della prima chiusura sono pari al $First()$ dei simboli <u>NON terminali</u> che generano le $Dotted\_rule$;
- I $Token$ degli stati delle altre chiusure sono ereditati in base al simbolo di transazione:
  - Se il simbolo di transazione è un <u>NON terminale</u>, allora i $Token$ sono pari a \$;
  - Se il simbolo di transazione è un <u>terminale</u>, allora i $Token$ sono pari al $Token$ dello stato che lo genera.

Per la costruzione della tabella del parser $LR(1)$ i passi sono simili a quello del parser $LR(0)$, con la differenza che:

- Solo per ogni stato $(X \to x., t)$ appartenente alla chiusura I_i, si inserisce $Reduce(X \to x.)$ in $M[i,t]$.

PARSER $LALR(1)$

Si osserva che ignorando i Lookahead alcune coppie di stati nelle grammatiche $LR(1)$ sono simili, pertanto si fa uso dei parser $LALR(1)$ per combinare questi Lookahead simili (detti **core**) e generare DFA $LR(1)$ simili ai DFA $LR(0)$. Infatti:

- Le prime componenti degli item $LR(1)$ sono item $LR(0)$;
- Se due stati p e q $LR(1)$ hanno lo stesso core e se da p esce una transizione con X verso lo stato p', allora anche da q uscirà una transizione con X verso q' e p' e q' avranno lo stesso core.

Per la costruzione dell'automa del parser $SLR(1)$ i passi sono simili a quello del parser $LR(1)$, con la differenza che:

- Si raggruppano gli stati con i core <u>in comune</u>.

Per la costruzione della tabella del parser $SLR(1)$ i passi sono simili a quello del parser $LR(1)$, con la differenza che:

- Le chiusure con i core in comune si comprimono in un'unica riga della tabella.

PROPRIETA DEI LINGUAGGI E DELLE GRAMMATICHE $\boldsymbol{LR(K)}$

La famiglia dei linguaggi verificabili da parser deterministici coincide con quella dei linguaggi generati dalle grammatiche $LR(1)$. Ciò non significa che ogni grammatica il cui linguaggio è deterministico, sia necessariamente $LR(1)$:
potrebbe richiedere una prospezione di lunghezza $K > 1$; tuttavia esisterà una grammatica equivalente $LR(1)$.

La famiglia dei linguaggi generati dalle grammatiche $LR(K)$ coincide con quella dei linguaggi generati da $LR(1)$.
Quindi un linguaggio CF ma non-deterministico non può avere una grammatica $LR(K)$ per nessun K.

Per ogni $K \geq 1$, esistono grammatiche che sono $LR(K)$ ma non $LR(K-1)$.

Data una grammatica, è indecidibile se esista un $K > 0$ per cui tale grammatica risulti $LR(K)$; di conseguenza non è decidibile se il linguaggio generato da una grammatica CF è deterministico. È decidibile soltanto se K è fissato
(si applica la costruzione del parser).

CONSIDERAZIONI FINALI SU LINGUAGGI E GRAMMATICHE $\boldsymbol{LL(K)}$ E $\boldsymbol{LR(K)}$

Ogni linguaggio regolare è $LL(1)$.

Ogni linguaggio $LL(K)$ è deterministico, ma ci sono linguaggi deterministici per cui non esiste alcuna grammatica $LL(k)$;

Per ogni $K \geq 0$, una grammatica $LL(K)$ è anche $LR(K)$.

Le grammatiche $LL(1)$ e $LR(0)$ non sono incluse una nell'altra.

Quasi tutte le grammatiche $LL(1)$ sono $LALR(1)$.

DOMANDE POTENZIALI PER L'ORALE

- Cosa si intende per front-end e back-end di un compilatore?
- Qual è il ruolo dell'analizzatore lessicale in un compilatore?