PARSER LL(1)

Parser LL(1)

Il termine LL(1) ha il seguente significato:

- 1. la prima L, significa che l'input è analizzato da sinistra verso destra
- 2. la seconda L, significa il parser costruisce una derivazione leftmost per la stringa di input
- 3. il numero 1, significa che l'algoritmo utilizza soltanto un solo simbolo dell'input per risolvere le scelte del parser (ci sono varianti con k simboli)

ESEMPIO

Il linguaggio delle parentesi bilanciate

$S \rightarrow (S) S$

$S \rightarrow \varepsilon$

e vediamo come opera il parser **LL(1)**

per riconoscere la stringa "( )"

Il parser consiste di una ***pila***, che contiene inizialmente il simbolo "$" (fondo della pila), ed un ***input***, la cui fine è marcata dal simbolo "$"

(EOF generato dallo scanner)

pila	**input**	**azione**
$	**( ) $**	

-il parsing inizia inserendo il simbolo iniziale in testa alla pila

pila	input	azione
$ S	( ) $	

- il parser **accetta** una stringa di input se, dopo una sequenza di azioni, la pila contiene "$" e la stringa di input è "$"

pila	input	azione
…	… …	
$	$	accept

- ogni volta che in testa alla pila c'è un simbolo non terminale X, lo si espande secondo una produzione X → γ, che viene scelta a seconda del simbolo in testa all'input e ai valori di una tabella (la parte destra della produzione viene invertita sulla pila)

pila	input	azione
$ S	( ) $	S → (S) S

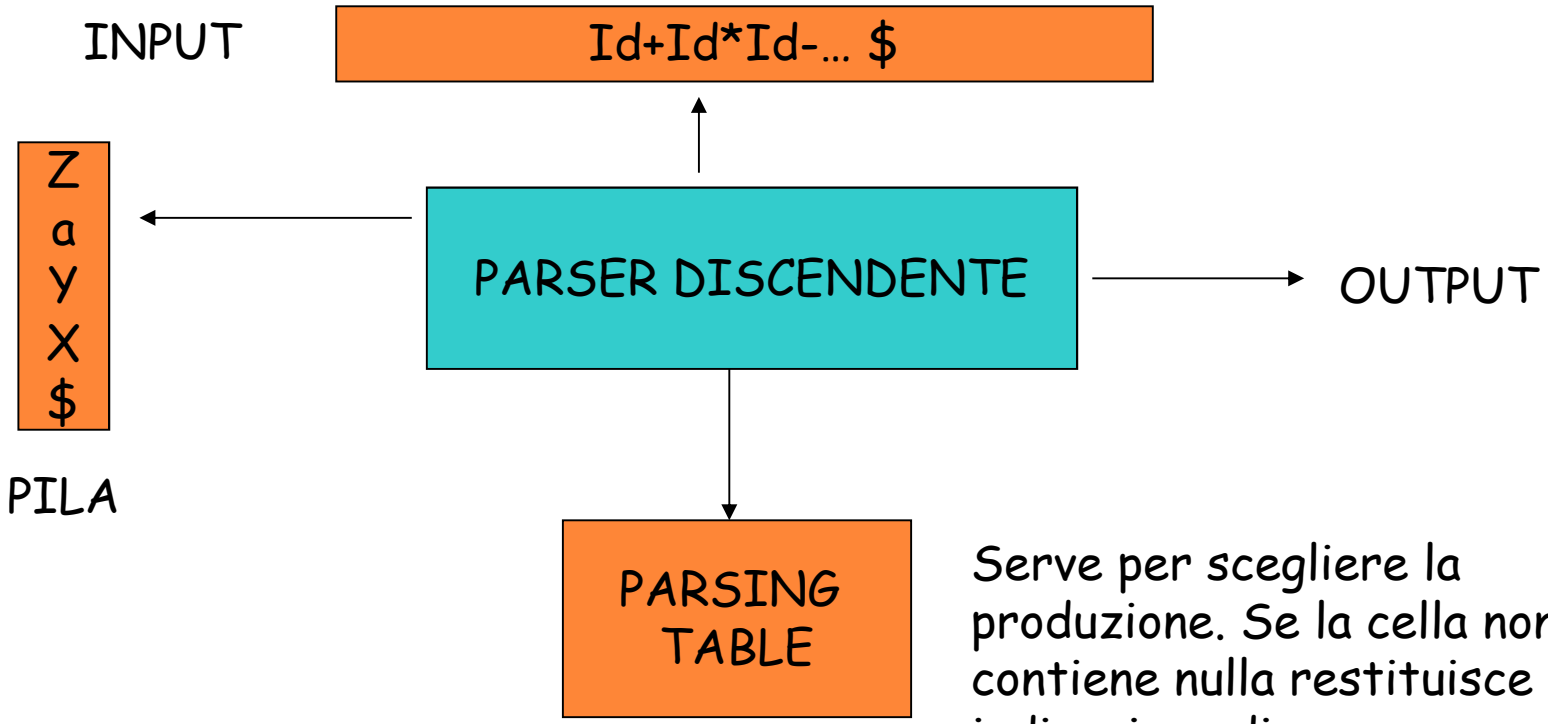
-ogni volta che sulla pila c'è un simbolo terminale **t**, si controlla che in testa all'input ci sia anche lo stesso simbolo, nel qual caso lo si elimina sia dalla pila che dall'input; altrimenti è **errore**

per costruire un parser **LL(1)**, bisogna costruire una tabella – la ***tabella LL(1)*** – che determina la regola da usare per l'espansione, dati il simbolo non-terminale e il carattere in input

pila	input	azione
$ S	( ) $	S → (S) S
$ S) S (	( ) $	match
$ S) S	) $	S → ε
$ S)	) $	match
$ S	$	S → ε
$	$	accept

Se l'input è generato dalla grammatica questo parsing fornisce una derivazione leftmost, altrimenti produce un'indicazione d'errore.

I Parser LL(1) sono parser discendenti non ricorsivi



Serve per scegliere la produzione. Se la cella non contiene nulla restituisce una indicazione di errore.

Costruzione della parsing table di un LL(1)

1. la tabella ha simboli non-terminali come righe, e simboli terminali come colonne
2. per ogni regola $X \rightarrow \gamma$ di G, si inserisce tale regola nella casella (X, **t**), per ogni **t** tale che $\gamma \Rightarrow^* \mathbf{t}\ \beta$
3. per ogni regola $X \rightarrow \gamma$ di G, per cui $\gamma \Rightarrow^* \varepsilon$, si inserisce nella casella (X, **t**) tale regola, per ogni **t** tale che $S \Rightarrow^* \beta\ X\ \mathbf{t}\ \alpha$

(Bisogna conoscere questi simboli **t** in grado di far effettuare la scelta della produzione
le regole 2 e 3 sono difficili da implementare: gli algoritmi per risolverle sono discussi in seguito vedi **FIRST** e **FOLLOW**)

per esempio: la tabella per la grammatica

S → (S) S , S → ε

	(	)	$
S	**S →(S) S**	**S → ε**	**S → ε**

First

- E' una funzione che consente di costruire le entries della tabella, quando possibile.

FIRST(γ): insieme dei simboli **terminali** che si trovano all'inizio delle stringhe derivate da γ (γ è una stringa di simboli terminali e non)

FIRST

1. se X è un terminale, allora FIRST(X) = { X },
2. se X-> ε appartiene alla grammatica, allora aggiungi ε a FIRST(X),
3. se X -> $Y_1Y_2...Y_k$ appartiene alla grammatica, allora:
   - se a∈FIRST(Y_i) per qualche i ed ε sta in FIRST(Y_1), FIRST(Y_2),..,FIRST(Y_{i-1}), aggiungi a in FIRST(X) ;
   - se tutti gli insiemi FIRST(Y_1), FIRST(Y_2),..,FIRST(Y_k), contengono ε, aggiungi ε a FIRST(X);
4. per definire l'insieme **FIRST**(γ), dove γ = $X_1X_2...X_k$ (una stringa di terminali e non), si procede reiterando le regole seguenti:
   - aggiungi **FIRST**(X_1) \ { ε } a **FIRST**(γ)
   - se per qualche **i<k** , tutti gli insiemi **FIRST**(X_1), . . . , **FIRST**(X_i) contengono ε, allora aggiungi **FIRST**(X_{i+1}) \ { ε } a **FIRST**(γ)
   - se tutti gli insiemi **FIRST**(X_1), . . . , **FIRST**(X_k) contengono ε, allora aggiungi ε a **FIRST**(γ)

Se X->*ε allora ε ∈FIRST (X) (X è detto annullabile)

Esempio di calcolo di First

Nel caso della grammatica delle espressioni aritmetiche senza ricorsioni sinistre:

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid \mathbf{id}$

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid \mathbf{id}$

FIRST(**id**)={**id**} FIRST(()={(} FIRST(+)={+} FIRST(*)={*}
FIRST())={)} FIRST(F)={**id**,(}
FIRST(T)=FIRST(E)=FIRST(F)
FIRST(E')={+, ε} FIRST(T')={*, ε}

ALGORITMO IN PSEUDO-C PER CALCOLARE FIRST

```
for all terminals X and ε do FIRST(X) = {X} ;
for all non-terminals X do FIRST(X) = { } ;
while (there are changes to any FIRST(X)) do
    for each production X → Y1 Y2... Yk do
            {
            i := 1 ; continue := true ;
            while (continue == true && i<=k) do
                        {
                        add FIRST(Yi)\{ε } to FIRST(X) ;
                        if (ε is not in FIRST(Yi)) continue := false ;
                        i := i+1 ;
                        }
            if (continue == true) add ε to FIRST(X) ;
            }
```

FIRST PUÒ NON BASTARE

Osservazione: se la grammatica contiene due produzioni
X -> γ_1
X -> γ_2
(*stesso simbolo non-terminale a sinistra, due sequenze differenti a destra*)
e **FIRST**(γ_1)∩**FIRST**(γ_2) è non vuota.

Allora la grammatica non può essere analizzata col parsing top-down: se **t** ∈ **FIRST**(γ_1)∩**FIRST**(γ_2) allora, il parsing discendente non saprà cosa fare quando il primo simbolo dell'input è **t**

FOLLOW

Dato un **non terminale** X, l'insieme **FOLLOW**(X) è l'insieme dei simboli terminali, eventualmente $, che appaiono alla destra di X in qualche forma sentenziale ed è definito come segue:

1. se X è l'assioma, allora aggiungi $ a **FOLLOW**(X)
2. se c'è una produzione A -> α X γ , allora aggiungi **FIRST**(γ)**{ ε}** a **FOLLOW**(X)
3. se c'è una produzione A -> α X γ per cui **ε** $\in$ **FIRST**(γ), allora aggiungi **FOLLOW**(A) a **FOLLOW**(X)

Osservazioni:

1. quando l'assioma non compare a destra delle produzioni, il "$" è l'unico simbolo nel suo insieme **FOLLOW**

2. l'insieme **FOLLOW** non contiene mai "**ε**"

3. FOLLOW è definito soltanto per non-terminali: potremmo generalizzare la definizione ma ciò è inutile per la tabella **LL(1)**

4. la definizione di **FOLLOW** lavora "alla destra" di una produzione, mentre quella di **FIRST** lavora "alla sinistra": una produzione X ->α non ha alcuna informazione su **FOLLOW**(X), se X non è presente in α

ALGORITMO IN PSEUDO-C PER CALCOLARE FOLLOW

```
FOLLOW(Axiom) = { $ } ;
for all (nonterminal(X) && X != Axiom) do FOLLOW(X)= {};
while (there are changes to any FOLLOW set) do
        for each production X ->Y 1 Y 2... Y k do
          for each nonterminal(Y i) do {
                add FIRST(Yi+1... Y k )\{ ε } to FOLLOW(Y i) ;
/* Note: if i=k then Yi+1... Y k = ε */
                if ε is in FIRST(Yi+1... Y k)
                  add FOLLOW(X) to FOLLOW(Yi);
                                        }
```

Esempio di calcolo di Follow

Nel caso della grammatica delle espressioni aritmetiche senza ricorsioni sinistre:

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid \mathbf{id}$

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid \mathbf{id}$

FIRST(**id**)={**id**} FIRST('('){'('} FIRST('+'){+} FIRST(*){*}
FIRST(')'){')'} FIRST(F)={**id**,'('}
FIRST(T)=FIRST(E)=FIRST(F)
FIRST(E')={+, ε} FIRST(T')={*, ε}

FOLLOW(E)={$,)} FOLLOW(E')={$,)}
FOLLOW(T)={$,),+} FOLLOW(T')={$,),+}
FOLLOW(F)={$,),+,*}

Altra definizione di Grammatica LL(1)

*Una grammatica la cui tabella **LL(1)** non contiene più di un elemento nelle caselle è detta **LL(1)***

Osservazione: per costruzione una grammatica **LL(1)** *non è ambigua, né ricorsiva sinistra*

Grammatica LL(1)

Una grammatica è LL(1) se e solo se per ogni produzione del tipo A->α|β si ha:

- α e β non derivano stringhe che cominciano con lo stesso simbolo a.
- Al più uno tra i due può derivare la stringa vuota.
- Se β->*ε allora α non deriva stringhe che cominciano con terminali che stanno in FOLLOW(A). Analogamente per α

Equivalentemente affinchè una grammatica sia LL(1) deve avvenire che per ogni coppia di produzioni A->α|β

1. FIRST(α) e FIRST(β) devono essere disgiunti
2. Se ε è in FIRST(β) allora FIRST(α) e FOLLOW (A) devono essere disgiunti.

COME COSTRUIRE LA TABELLA?

1. Per ogni regola X -> α di *G*, si inserisce nella casella (X, **t**) la regola X -> α, per ogni **t** tale che **t** $\in$ **FIRST**(α)
2. Per ogni regola X -> α di *G*, per cui $\alpha \Rightarrow^*$ ε ($\varepsilon \in$ **FIRST**(α)), si inserisce nella casella (X, **t**) la regola X -> α , per ogni **t** tale che **t** $\in$ **FOLLOW**(X).
   Se $\varepsilon \in$ **FIRST**(α) and \$$\in$ **FOLLOW**(X), si inserisce la regola X -> α in (X,\$).
3. Le caselle non definite definiscono un errore.

NOTA: Se *G* è ricorsiva sinistra o ambigua, la tabella avrà caselle con valori multipli.

ALGORITMO DI PARSING LL(1)

```
LL1_parser( stack p, input i, LL1_table M, initial S ) {
 /* p, i sono pile, S è l'assioma*/
int error = 0 ; p = push(p, $) ;
p = push(p, S) ;
while ( top(p) ≠ $ && top(i) ≠ $ && !error){
        if isterminal(top(p)) {
                if top(p) == top(i) { p = pop(p) ; i = avanza(i) ; }
                else error = 1 ;}
        else {/* top(p) è un non-terminale, bisogna espandere */
                if isempty(M[top(p),top(i)]) error = 1;
                else { /* M[top(p),top(i)] == X -> X1 ... Xn; */
                        p = pop(p) ;
                        for (j = n ; j > 0 ; j = j-1)
                        p = push(p, Xj) ;}}
        }
    if (!error) accept() ;
    else raise_error() ;
}
```

Complessità di calcolo del parser LL(1)

E’ lineare nella lunghezza n della stringa sorgente, poiché viene consumato un carattere per volta.

ESEMPIO: costruire il parser **LL(1)** per la grammatica

S → (S) S → [S] S → { S }

S → ε

insiemi **FIRST**, **FOLLOW**:

	FIRST	**FOLLOW**
S	(,[,{,ε	),],},$

la tabella **LL(1)**:

	(	[	{	)	]	}	$
S	S →(S)	S →[S]	S →{S}	S →ε	S →ε	S→ε	S →ε

Esercizio con Flex

- Scrivere in Flex un tokenizzatore per il linguaggio C che riconosca i seguenti lessemi:

main
int
void
return
{
}
(
)
,
=
+
*

identificatori (che cominciano con lettera), con token ID.

Costanti intere con token INTCONST

Esercizio con Flex

- Il programma deve restituire l'elenco dei token, i relativi attributi (ovvero nel caso degli identificatori, il nome dell'identificatore, nel caso delle costanti, il numero) e il numero di riga in cui il token compare.
- I commenti devono essere riconosciuti e devono essere contate le righe che contengono commenti , sia quelli racchiusi tra /* e */ che il commento preceduto da //.

Esercizio con Flex

- Per esempio, se l'input è

```
int main(void)
        { int x;
        x = 19;
        x = x * x;
        /* commento
        su due righe*/
        return 0; }
```

- Verrà prodotto il seguente output

TOKEN	LESSEMA	LINENO
tok_int		1
tok_main		1
tok_lparen		1
tok_void		1
tok_rparen		1
tok_lbrace		2
tok_int		2
ID	x	2

Esercizio con Flex

tok_semicolon		2
ID	x	3
tok_equal		3
INTCONST	19	3
tok_semicolon		3
ID	x	4
tok_equal		4
ID	x	4
tok_mult		4
ID	x	4
tok_semicolon		4
tok_rbrace		4
tok_return		7
INTCONST	0	7
tok_semicolon		7
tok_rbrace		7

2 righe contengono commenti.