Applicazioni alla traduzione e compilazione

Corso di Fondamenti di Informatica - 1 modulo
Corso di Laurea in Informatica
Università di Roma ``Tor Vergata''

Giorgio Gambosi

a.a. 2018-2019

Fasi del processo di compilazione

1. Analisi lessicale. La sequenza di caratteri viene suddivisa in *token*, elementi significanti (come simboli terminali) nella definizione della sintassi del linguaggio. Costruzione della *tabella dei simboli*
2. Analisi sintattica. La sequenza di terminali viene esaminata per derivare la struttura sintattica del programma (albero sintattico, derivazione destra, derivazione sinistra)
3. Generazione del codice intermedio. A partire dalla struttura sintattica del programma e dalla tabella dei simboli, viene generata una prima versione tradotta del programma, in un linguaggio intermedio. Vengono effettuati una serie di controlli su aspetti non context-free del codice, come ad esempio il *type checking*.
4. Generazione codice finale e ottimizzazione.

Fasi del processo di compilazione

Le prime due fasi sono basate sulla teoria dei linguaggi e degli automi.

- Analisi lessicale. Basata sui linguaggi di tipo 3: la struttura dei vari tipi di token è definita mediante espressioni regolari. Idealmente, l'analisi è effettuata esaminando la sequenza di caratteri per mezzo di tanti ASFD quanti sono i tipi di token.
- Analisi sintattica. Basata sui linguaggi context free: la struttura del linguaggio è definita mediante una grammatica CF. L'analisi è effettuata applicando un opportuno automa a pila per decidere la correttezza del programma (derivabilità nella grammatica del linguaggio) e, in tal caso, il corrispondente albero sintattico (o derivazione destra, o derivazione sinistra).

Analisi lessicale

Basata sulla definizione di un insieme di *pattern*, mediante espressioni regolari. Ad esempio $[a-z, A-Z][a-z, A-Z, 0-9, .]^*$ può corrispondere alla struttura di un identificatore alfanumerico nel linguaggio, *for* alla parola chiave `for`

Un *lessema* è sequenza di caratteri che corrisponde a un pattern. Ad esempio 'a0_z' o 'my_counter' per il primo caso, o 'for' per il secondo

L'analizzatore lessicale associa un token (associato al pattern) ad ogni lessema: la sequenza di caratteri viene sostituita da una sequenza di token

Esempio

pattern	token
$[a-z, A-Z][a-z, A-Z, 0-9, \_, .]^*$	`id`
$0 + [1-9][0-9]^*$	`digit`
$==, <, <=, >, >=$	`rel_op`
$=$	`assign`
for	`for`
if	`if`
else	`else`

if a0 > my_counter i=3 else j=0

`if id rel_op id assign digit else id assign digit`

Tabella dei simboli

L'analizzatore lessicale costruisce una tabella dei simboli incontrati, che associa ad ogni token (almeno quelli per i quali è necessario) il corrispondente lessema, insieme eventualmente ad altre informazioni (ad es. il tipo)

L'analisi sintattica fa riferimento ai soli token, mentre la generazione del codice successiva utilizza valori derivati dai lessemi.

Analizzatore lessicale

- Simula un ASF per ogni espressione regolare definita (corrispondente a un pattern e un token)
- Quando un ASF accetta la stringa letta fino ad ora, restituisce il token corrispondente

Dato un insieme di espressioni regolari, è possibile derivare in modo automatico l'insieme di ASF che le accettano, e quindi l'analizzatore lessicale corrispondente: *Generatori di analizzatori lessicali*

Analisi sintattica

Analizzatore sintattico, o *parser*:

- è associato ad una grammatica context free non ambigua, che definisce la struttura sintattica del linguaggio utilizzato
- riceve in input una stringa di token (simboli terminali della grammatica)
- verifica se la stringa è derivabile nella grammatica
- in tal caso, restituisce l'unico albero sintattico associato alla stringa

Analisi sintattica

Un parser opera esaminando la stringa da sinistra a destra, in due possibili modi

- in funzione dei primi simboli letti, predice quale è la prossima produzione utilizzata in una derivazione sinistra della stringa, se esiste; l'albero sintattico viene prodotto dall'alto verso il basso (*parsing predittivo* o *top-down*)
- in funzione dei primi simboli letti, predice quale produzione è stata utilizzata per produrli, sostituendoli nella stringa con la parte sinistra della produzione: iterando questa operazione, si produce, se esiste, la produzione destra che genera la stringa, al contrario; l'albero sintattico viene prodotto dal basso verso l'alto (*bottom-up parsing*)

Analisi sintattica

Predizioni non guidate in nessun modo delle produzioni utilizzate ad ogni passo (e quindi delle derivazioni) equivalgono alla simulazione di un NPDA, e comportano tempi di parsing esponenziali nella lunghezza della stringa.

E' necessario prevedere strutture particolari delle grammatiche CF utilizzate, che consentano di utilizzare parser che operano in modo più efficiente, guidati al meglio dai simboli letti

Parsing predittivo

In una derivazione sinistra di una stringa, una forma di frase è necessariamente del tipo $V_T^+(V_T \cup V_N)^*$.

Esempio: la grammatica

$$\begin{aligned} T &\longrightarrow R \mid aTc \\ R &\longrightarrow RbR \mid \varepsilon \end{aligned}$$

e la produzione sinistra

$$T \Longrightarrow \underline{a}Tc \Longrightarrow \underline{aa}Tcc \Longrightarrow \underline{aa}Rcc \Longrightarrow \underline{aa}RbRcc \Longrightarrow \underline{aa}RbRbRcc \Longrightarrow$$
$$\underline{aab}RbRcc \Longrightarrow \underline{aab}RbRbRcc \Longrightarrow \underline{aabb}RbRcc \Longrightarrow \underline{aabbb}Rcc \Longrightarrow \underline{aabbbcc}$$

Parsing predittivo

Nel corso di un parsing predittivo, alla forma di frase xAw, con $x \in V_T^*$, $A \in V_N$, $w \in (V_T \cup V_N)^+$, corrisponde una situazione in cui il parser ha letto il prefisso x della stringa e deve determinare, sulla base di esso, quale delle produzioni aventi A a sinistra applicare.

Se la produzione selezionata è $A \longrightarrow yBu$, con $y \in V_T^*$, $B \in V_N$, $u \in (V_T \cup V_N)^+$, la nuova forma di frase è $xyBuw$ e il parser, letta y, deve decidere quale produzione applicare per riscrivere B.

Il parser inizia da $\varepsilon S \sigma$, se σ è la stringa in input e S l'assioma.

Parsing predittivo

letta	forma di frase	stringa	operazione
ε	T	aabbbcc	$T \longrightarrow aTc$
a	aTc	abbbcc	$T \longrightarrow aTc$
aa	aaTcc	bbbcc	$T \longrightarrow R$
aa	aaRcc	bbbcc	$R \longrightarrow RbR$
aa	aaRbRcc	bbbcc	$R \longrightarrow RbR$
aa	aaRbRbRcc	bbbcc	$R \longrightarrow \varepsilon$
aab	aabRbRcc	bbcc	-
aab	aabRbRcc	bbcc	$R \longrightarrow \varepsilon$
aabb	aabbRcc	bcc	-
aabb	aabbcc	cc	$R \longrightarrow \varepsilon$
aabbc	aabbcc	c	-
aabbcc	aabbcc	ε	-

Parsing predittivo

Grammatica

$$
\begin{aligned}
E &\longrightarrow TE' \\
E' &\longrightarrow +TE' \mid \varepsilon \\
T &\longrightarrow FT' \\
T' &\longrightarrow *FT' \mid \varepsilon \\
F &\longrightarrow (E) \mid \texttt{id}
\end{aligned}
$$

Stringa

`id+id*id`

Parsing predittivo

ε	E	id+id*id	$E \longrightarrow TE'$
ε	TE'	id+id*id	$T \longrightarrow FT'$
ε	FT'E'	id+id*id	$F \longrightarrow \texttt{id}$
id	idT'E'	+id*id	$T' \longrightarrow \varepsilon$
id	idE'	+id*id	$E' \longrightarrow +TE'$
id+	id+TE'	id*id	$T \longrightarrow FT'$
id+	id+FT'E'	id*id	$F \longrightarrow \texttt{id}$
id+id	id+idT'E'	*id	$T' \longrightarrow *FT'$
id+id*	id+id*FT'E'	id	$F \longrightarrow \texttt{id}$
id+id*id	id+id*idT'E'	ε	$T' \longrightarrow \varepsilon$
id+id*id	id+id*idE'	ε	$E' \longrightarrow \varepsilon$
id+id*id	id+id*id	ε	-

Ad ogni passo è possibile selezionare una sola produzione, guardando un solo terminale (token): *LL(1)*

Parsing a discesa ricorsiva

Implementazione di parser top down: una funzione **A( )** per ogni $A \in V_N$, con la struttura seguente. Il programma inizia da S()

A():
    for each $A \longrightarrow X_1 X_2 \cdots X_k \in P$:
        for i in range$(1, k+1)$:
            if $X_i \in V_N$:
                if $X_i()$:
                    return 1
            elif X_i uguale al prossimo simbolo a della stringa:
                avanza al simbolo successivo
            else :
                break
        return 0

Parsing a discesa ricorsiva

Utilizzo del *backtracking*: può essere molto inefficiente

Backtracking: esplorazione ricorsiva di tutte le possibilità.

La scelta della produzione da considerare può essere guidata dall'esame dei caratteri successivi.

La grammatica non può essere *ricorsiva sinistra*

$$A \longrightarrow Aw$$

Parsing predittivo efficiente

Se per ogni simbolo non terminale da espandere i prossimi k caratteri della stringa consente di individuare la produzione da applicare, il parser è $LL(k)$

- Left-to-right: la derivazione è calcolata da sinistra a destra (dalla prima produzione applicata all'ultima)
- Leftmost derivation: la derivazione calcolata è sinistra
- k simboli (di *look-ahead*) da considerare

Un linguaggio CF è $LL(k)$ se esiste un parser $LL(k)$ che può effettuarne l'analisi sintattica

La funzione FIRST

Consideriamo il caso $LL(1)$, per semplicità.

Per ogni sequenza $\alpha \in (V_T \cup V_N)^+$, $c \in V_T$, diciamo che $c \in$ FIRST(α) se e solo se $\alpha \stackrel{*}{\Longrightarrow} c\beta$ per $\beta \in (V_T \cup V_N)^*$

Quindi FIRST(α) è l'insieme dei terminali che possono comparire in prima posizione in forme di frase derivate a partire da α

Siamo in particolare interessati a $FIRST(\alpha)$ se α è la parte destra di una produzione $A \longrightarrow \alpha$: perché?

Perché se $c \in$ FIRST(α) e $A \longrightarrow \alpha$, una stringa $w = cy$ che inizia per c potrebbe essere derivata a partire da A, in quanto $A \Longrightarrow \alpha \stackrel{*}{\Longrightarrow} c\beta$

La funzione FIRST

Date le A-produzioni in P

$$A \quad \longrightarrow \quad \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$$

se ogni terminale appartiene a non più di un insieme FIRST(α_i), allora possiamo sempre individuare quale produzione applicare per riscrivere A, esaminando il solo prossimo carattere c

Infatti, va applicata $A \longrightarrow \alpha_i$ se e solo se $c \in$ FIRST(α_i)

Se non esiste α_i tale che $c \in$ FIRST(α_i), c'è un errore e la parte di stringa da leggere non è derivabile a partire da A

Costruzione della funzione FIRST

Per la costruzione di FIRST(α) va utilizzato il predicato Nullable(β) definito come Nullable(β) = TRUE se e solo se β è *annullabile*, cioè se e solo se esiste una derivazione $\beta \stackrel{*}{\Longrightarrow} \varepsilon$. Una produzione $B \longrightarrow \beta$ è *annullabile* se e solo se β è annullabile.

La costruzione di Nullable(β) è basata sulle seguenti proprietà

$$\begin{aligned}
\text{Nullable}(\varepsilon) &= \texttt{TRUE} \\
\text{Nullable}(a) &= \texttt{False} \qquad \forall a \in V_T \\
\text{Nullable}(\alpha\beta) &= \text{Nullable}(\alpha) \wedge \text{Nullable}(\beta) \\
\text{Nullable}(A) &= \bigvee_i \text{Nullable}(\alpha_i) \qquad \forall A \in V_N, \forall A \longrightarrow \alpha_i \in P
\end{aligned}$$

Costruzione della funzione FIRST

La costruzione di FIRST(α) avviene in modo simile.

Consideriamo in primo luogo la costruzione di FIRST(X), dove X è un simbolo della grammatica, $X \in V_T \cup V_N$

1. Se $X \in V_T$, allora FIRST(X) $= \{X\}$
2. Se $X \in V_N$, per ogni $X \longrightarrow Y_1 Y_2 \cdots Y_k \in P$ $(k \geq 1)$:
   - 2.1 FIRST(Y_1) $\subseteq$ FIRST(X)
   - 2.2 Per $i = 2, \ldots k$, se Nullable($Y_1 \cdots Y_{i-1}$) allora FIRST(Y_i) $\subseteq$ FIRST(X)

Costruzione della funzione FIRST

Costruzione di FIRST($X_1 \cdots X_n$) da FIRST(X) per ogni X:

- FIRST(X_1) $\subseteq$ FIRST($X_1 \cdots X_n$)
- Per $i = 2, \ldots n$, se Nullable($X_1 \cdots X_{i-1}$) allora FIRST(X_i) $\subseteq$ FIRST($X_1 \cdots X_n$)

La funzione FIRST

Da quanto detto, se $x = cy$ è la stringa da leggere e A il terminale da riscrivere, le possibili produzioni da applicare sono tutte le $A \longrightarrow \alpha_i$ tali che $c \in$ FIRST(α_i). Se in tutti i casi possibili c'è al più una di tali produzioni, abbiamo un parser $LL(1)$.

Errore! In realtà cy potrebbe essere prodotta in modo diverso:

Supponiamo che la forma di frase attuale sia ABw, con $w \in (V_T \cup V_N)^*$ e supponiamo che A sia annullabile, e quindi che esista una derivazione $A \stackrel{*}{\Longrightarrow} \varepsilon$: allora, x potrebbe essere ancora derivabile se $c \in$ FIRST(B), e quindi $B \stackrel{*}{\Longrightarrow} c\beta$, in quanto

$$ABw \stackrel{*}{\Longrightarrow} Bw \stackrel{*}{\Longrightarrow} c\beta w$$

La funzione FOLLOW

Dati $A \in V_N$ e $a \in V_T$, $a \in$ FOLLOW(A) se e solo se esiste una forma di frase derivata dall'assioma in cui a segue immediatamente A.

Quindi $a \in$ FOLLOW(A) se e solo se esistono $\alpha, \beta \in (V_T \cup V_N)^*$ tali che $S \stackrel{*}{\Longrightarrow} \alpha A a \beta$

Durante il parsing, se A è il non terminale da riscrivere e c il simbolo attualmente letto, la produzione $A \longrightarrow \varepsilon$ (se esiste) può essere applicata nel caso in cui $c \in$ FOLLOW(A)

Costruzione della funzione FOLLOW

Per tener conto del caso in cui A potrebbe essere l'ultimo simbolo di una forma di frase, cioè in cui $S \stackrel{*}{\Longrightarrow} \alpha A$, estendiamo la grammatica con:

- un non terminale \$ di fine stringa
- un nuovo assioma S'
- una produzione $S' \longrightarrow S\$$

Evidentemente, A può comparire a fine stringa nella prima grammatica se e solo se $\$ \in \text{FOLLOW}(A)$ nella nuova grammatica.

Costruzione della funzione FOLLOW

FOLLOW viene costruita a partire da un insieme di vincoli derivati dalle produzioni.

- $\$ \in \text{FOLLOW}(S)$
- Se $A \longrightarrow \alpha B \beta \in P$, allora $\text{FIRST}(\beta) \subseteq \text{FOLLOW}(B)$
- se $A \longrightarrow \alpha B \beta \in P$ e $\text{Nullable}(\beta)$, allora $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$
- se $A \longrightarrow \alpha B \in P$ allora $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$

Esempio

Grammatica

$$
\begin{aligned}
E &\longrightarrow TE' \\
E' &\longrightarrow +TE' \mid \varepsilon \\
T &\longrightarrow FT' \\
T' &\longrightarrow *FT' \mid \varepsilon \\
F &\longrightarrow (E) \mid \texttt{id}
\end{aligned}
$$

Esempio

$\text{Nullable}(E') = \text{Nullable}(T') = \texttt{TRUE}$

- $\text{FIRST}(F) = \{(, \texttt{id}\}$
- $\text{FIRST}(T') = \{*\}$
- $\text{FIRST}(E') = \{+\}$
- $\text{FIRST}(T) = \text{FIRST}(F) = \{(, \texttt{id}\}$
- $\text{FIRST}(E) = \text{FIRST}(T) = \{(, \texttt{id}\}$

Esempio

- $\$ \in \text{FOLLOW}(E)$
- $\text{FIRST}(E') = \{+\} \subseteq \text{FOLLOW}(T)$
- $\text{FIRST}(T') = \{*\} \subseteq \text{FOLLOW}(F)$
- $\text{FIRST}(')') = \{)\} \subseteq \text{FOLLOW}(E)$
- $\text{FOLLOW}(E) \subseteq \text{FOLLOW}(E')$
- $\text{FOLLOW}(E) \subseteq \text{FOLLOW}(T)$
- $\text{FOLLOW}(T) \subseteq \text{FOLLOW}(T')$
- $\text{FOLLOW}(E') \subseteq \text{FOLLOW}(T)$
- $\text{FOLLOW}(T') \subseteq \text{FOLLOW}(F)$

Esempio

Da cui deriva

- $\text{FOLLOW}(E) = \{\$,)\}$
- $\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{\$,)\}$
- $\text{FOLLOW}(T) = \text{FOLLOW}(E) \cup \{+\} = \{\$,), +\}$
- $\text{FOLLOW}(T') = \text{FOLLOW}(T) = \{\$,), +\}$
- $\text{FOLLOW}(F) = \text{FOLLOW}(T') \cup \{*\} = \{\$,), +, *\}$

Tabella di parsing predittivo

Associa ad ogni coppia (a, X), $a \in V_T$, $X \in V_N$, un insieme di produzioni (1 se $LL(1)$) da applicare nel caso in cui X sia il non terminale da riscrivere e a sia il simbolo letto in input.

Costruzione della tabella M:

Per ogni produzione $A \longrightarrow \alpha \in P$:

- se $\alpha \neq \varepsilon$, per ogni $a \in$ FIRST(A) aggiungi $A \longrightarrow \alpha$ a $M[A, a]$
- se Nullable(α), per ogni $b \in$ FOLLOW(A) aggiungi $A \longrightarrow \alpha$ a $M[A, b]$

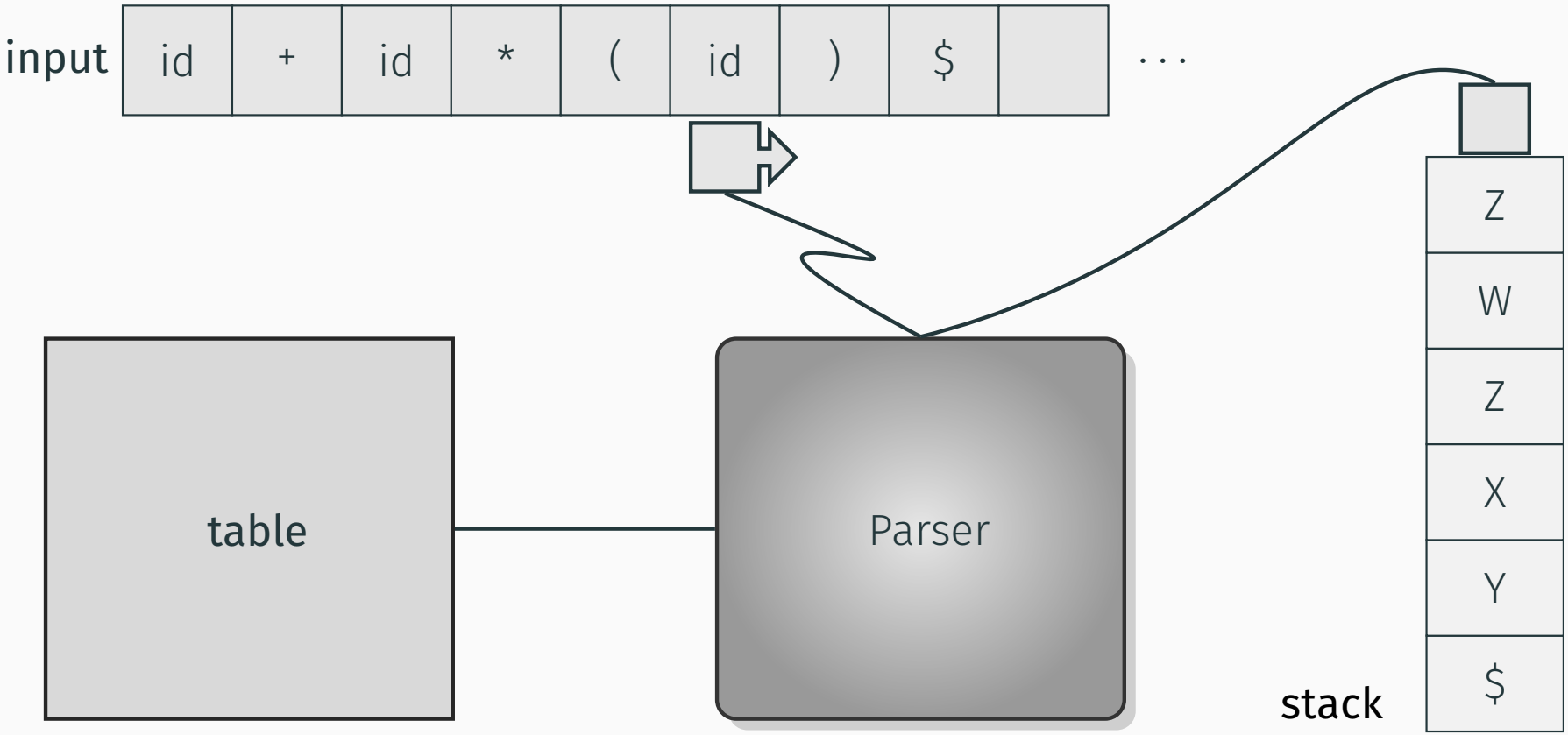
Esempio

Per la grammatica precedente

	`id`	`+`	`*`	`(`	`)`	`$`
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \texttt{id}$			$F \to (E)$		

Parsing predittivo non ricorsivo

Utilizza uno stack (pila) in modo esplicito, invece che implicitamente , simulando una derivazione sinistra della stringa.



Parsing predittivo non ricorsivo

input.first()
stack.push(S$)
while stack.top()! $= \$$:
  if stack.top() == input.current():
    stack.pop()
    input.next()
  elif table[stack.top(), input.current()]! = Null:
    Let table[stack.top(),input.current()] be $X \to Y_1 \cdots Y_k$
    output stack.top() $\longrightarrow Y_1 \cdots Y_k$
    stack.pop()
    stack.push($Y_1 \cdots Y_k$)
  else:
    error

Parsing predittivo non ricorsivo

Esempio di parsing di id $+$ id $*$ id

Matched	Stack	Input	Action
	E$	id+id*id$	
	TE'$	id+id*id$	output $E \to TE'$
	FT'E'$	id+id*id$	output $T \to FT'$
	idT'E'$	id+id*id$	output $F \to$ id
id	T'E'$	+id*id$	match id
id	E'$	+id*id$	output $T' \to \varepsilon$
id	+TE'$	+id*id$	output $E' \to +TE'$
id+	TE'$	id*id$	match $+$
id+	FT'E'$	id*id$	output $T \to FT'$
id+	idT'E'$	id*id$	output $F \to$ id
id+id	T'E'$	*id$	match id

Parsing predittivo non ricorsivo

Matched	Stack	Input	Action
id+id	*FT'E'$	*id$	output $T' \to *FT'$
id+id*	FT'E'$	id$	match $*$
id+id*	idT'E'$	id$	output $F \to$ id
id+id*id	T'E'$	$	match id
id+id*id	E'$	$	output $T' \to \varepsilon$
id+id*id	$	$	output $E' \to \varepsilon$

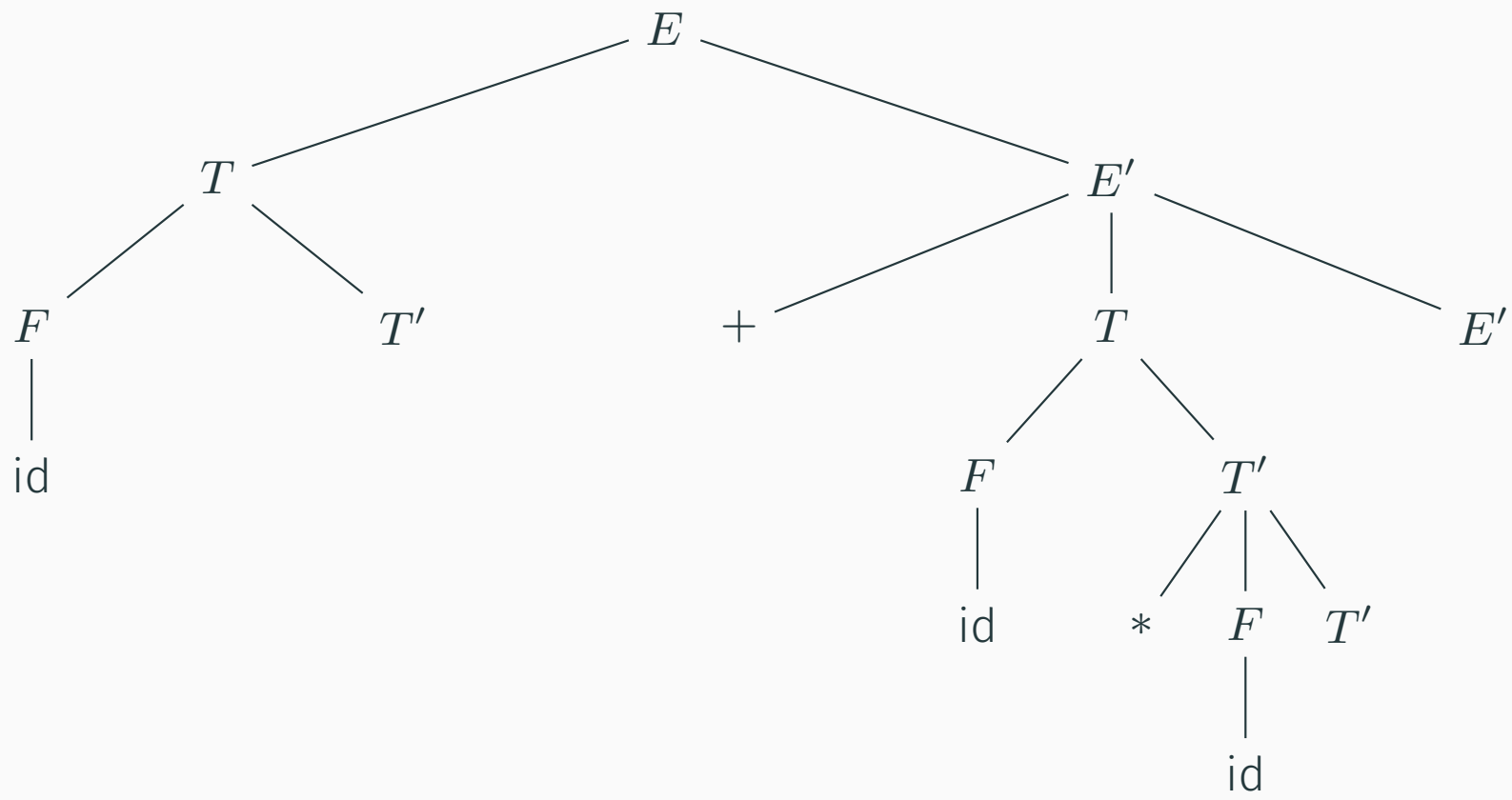
Parsing predittivo non ricorsivo

Ne risulta la derivazione sinistra

$$
\begin{aligned}
E &\Longrightarrow TE' \\
&\Longrightarrow \mathrm{id}T'E' \\
&\Longrightarrow \mathrm{id} + TE' \\
&\Longrightarrow \mathrm{id} + \mathrm{id}T'E' \\
&\Longrightarrow \mathrm{id} + \mathrm{id} * \mathrm{id}T'E' \\
&\Longrightarrow \mathrm{id} + \mathrm{id} * \mathrm{id}
\end{aligned}
\qquad
\begin{aligned}
&\Longrightarrow FT'E' \\
&\Longrightarrow \mathrm{id}E' \\
&\Longrightarrow \mathrm{id} + FT'E' \\
&\Longrightarrow \mathrm{id} + \mathrm{id} * FT'E' \\
&\Longrightarrow \mathrm{id} + \mathrm{id} * \mathrm{id}E'
\end{aligned}
$$

Parsing predittivo non ricorsivo

E l'albero sintattico



Parsing bottom up

Costruzione dell'albero sintattico dal basso verso l'alto.

Equivalentemente, costruzione della una derivazione destra (in ordine inverso rispetto alla derivazione stessa).

Parsing LR(k):

- Left-to-right: la derivazione è calcolata da sinistra a destra (dalla prima produzione applicata all'ultima)
- Rightmost derivation: la derivazione calcolata è destra
- k simboli (di *look-ahead*) da considerare

Riduzioni

Operazioni base del parsing LR:

Una sottostringa (handle) della forma di frase attuale α, corrispondente alla parte destra di una produzione, viene sostituita dalla parte sinistra.

- Forma di frase attuale $\alpha = \delta\eta\zeta$, con $\delta, \zeta \in (V_T \cup V_N)^*$, e $\eta \in (V_T \cup V_N)^* V_N (V_T \cup V_N)^*$
- Produzione $A \longrightarrow \eta \in P$

Riduzione: la nuova forma di frase è $\delta A \zeta$

Esempio di parsing bottom up

Consideriamo ancora la grammatica

$$
\begin{array}{rcl}
E & \longrightarrow & TE' \\
E' & \longrightarrow & +TE' \mid \varepsilon \\
T & \longrightarrow & FT' \\
T' & \longrightarrow & *FT' \mid \varepsilon \\
F & \longrightarrow & (E) \mid \texttt{id}
\end{array}
$$

e la stringa id $+$ id $*$ id.

Esempio di parsing bottom up

Assumiamo di poter individuare sempre la prima handle da sinistra nella forma di frase attuale.

Handle	Produzione	Result
id+id*id$	$F \to \texttt{id}$	F+id*id
F+id*id$	$T' \to \varepsilon$	FT'+id*id
FT'+id*id$	$T \to FT'$	T+id*id
T+id*id$	$F \to \texttt{id}$	T+F*id
T+F*id$	$F \to \texttt{id}$	T+F*F
T+F*F$	$T' \to \varepsilon$	T+F*FT'
T+F*FT'$	$T' \to *FT'$	T+FT'
T+FT'$	$T \to FT'$	T+T
T+T$	$E' \to \varepsilon$	T+TE'
T+TE'$	$E' \to +TE'$	TE'
TE'$	$E \to TE'$	E

Esempio di parsing bottom up

La sequenza di produzioni individuate, lette al contrario (dal basso in alto), forniscono la derivazione destra della stringa

$$E \Longrightarrow TE' \Longrightarrow T + TE' \Longrightarrow T + T \Longrightarrow T + FT' \Longrightarrow T + F * FT' \Longrightarrow$$
$$T + F * F \Longrightarrow T + F * \texttt{id} \Longrightarrow T + \texttt{id} * \texttt{id} \Longrightarrow FT' + \texttt{id} * \texttt{id} \Longrightarrow$$
$$F + \texttt{id} * \texttt{id} \Longrightarrow \texttt{id} + \texttt{id} * \texttt{id}$$

Problema: come individuare handle e riduzione?

Parsing shift-reduce

Un parser shift-reduce effettua parsing bottom up utilizzando:

- una pila di simboli della grammatica
- un buffer di input, in cui è contenuta la parte dell'input ancora da leggere

Il carattere $\$ \notin V_T$ è utilizzato per marcare il fondo della pila e la fine della stringa di input, per cui inizialmente, se w è la stringa:

- la pila contiene $\$$
- il buffer contiene $w\$$

Pila	Input
$	w$

Parsing shift-reduce: operazioni

Reduce: i simboli in cima alla pila corrispondono ad una handle per una produzione $A \longrightarrow \alpha$. Viene effettuata una reduce, passando da

Pila	Input	a	Pila	Input
$\$\beta\alpha$	$w\$$		$\$\beta A$	$w\$$

Shift: i simboli in cima alla pila non corrispondono ad una handle. Il prossimo simbolo in input viene posto sulla pila, passando da

Pila	Input	a	Pila	Input
$\$\alpha$	$aw\$$		$\$\alpha a$	$w\$$

Parsing shift-reduce: operazioni

Accept: la pila contiene $\$S$, dove S è l'assioma della grammatica, il buffer di input contiene $\$$ (la stringa è terminata). La stringa è accettata

Error: non ci sono altre azioni eseguibili nella configurazione attuale di pila e buffer. La tringa è rifiutata

Parsing shift-reduce

Pila	Input	Azione
\$	`id`+`id`*id\$	shift
\$id	+`id`*id\$	reduce $F \longrightarrow$ `id`
\$$F$	+`id`*id\$	reduce $T' \longrightarrow \varepsilon$
\$$FT'$	+`id`*id\$	reduce $T \longrightarrow FT'$
\$$T$	+`id`*id\$	shift
\$$T+$	`id`*id\$	shift
\$$T+$id	*id\$	reduce $F \longrightarrow$ `id`
\$$T+F$	*id\$	shift
\$$T+F*$	id\$	shift
\$$T+F*$id	\$	reduce $F \longrightarrow$ `id`
\$$T+F*F$	\$	reduce $T' \longrightarrow \varepsilon$
\$$T+F*FT'$	\$	reduce $T' \longrightarrow *FT'$

Parsing shift-reduce

Pila	Input	Azione
$\$T + FT'$	\$	reduce $T \longrightarrow FT'$
$\$T + T$	\$	reduce $E' \longrightarrow \varepsilon$
$\$T+TE'$	\$	reduce $E' \longrightarrow +TE'$
$\$TE'$	\$	reduce $E \longrightarrow TE'$
$\$E$	\$	accept

Parsing shift-reduce

Pila	Input	Azione
$\$T + FT'$	\$	reduce $T \longrightarrow FT'$
$\$T + T$	\$	reduce $E' \longrightarrow \varepsilon$
$\$T + TE'$	\$	reduce $E' \longrightarrow +TE'$
$\$TE'$	\$	reduce $E \longrightarrow TE'$
$\$E$	\$	accept

Parsing shift-reduce

Si può mostrare che l'handle, in un parsing bottom up, apparirà sempre in cima alla pila. Due casi possibili.

Derivazione destra:

$$S \stackrel{*}{\Longrightarrow} \alpha A z \Longrightarrow \alpha\beta B y z \Longrightarrow \alpha\beta\gamma y z$$

dove si sono applicate le produzioni $A \longrightarrow \beta B y$ e $B \longrightarrow \gamma y$, con $\alpha, \beta, \gamma \in (V_T \cup V_N)^*$ e $y, z \in V_T^*$

Nel parsing, si ha la corrispondente sequenza di azioni

Pila	Input	Azione
$\$\alpha\beta\gamma$	yz$	reduce $B \longrightarrow \gamma$
$\$\alpha\beta B$	z$	shift y
$\$\alpha\beta B y$	z$	reduce $A \longrightarrow \beta y$
$\$\alpha A$	z$	

Parsing shift-reduce

Derivazione destra:

$$S \overset{*}{\Longrightarrow} \alpha B x A z \Longrightarrow \alpha B x y z \Longrightarrow \alpha \gamma x y z$$

dove si sono applicate le produzioni $A \longrightarrow y$ e $B \longrightarrow \gamma y$, con $\alpha, \gamma \in (V_T \cup V_N)^*$ e $x, y, z \in V_T^*$

Nel parsing, si ha la corrispondente sequenza di azioni

Pila	Input	Azione
$\$\alpha\gamma$	xyz$	reduce $B \longrightarrow \gamma$
$\$\alpha B$	xyz$	shift xy
$\$\alpha B x y$	z$	reduce $A \longrightarrow y$
$\$\alpha B x A$	z$	