Linguaggi Formali e Traduttori

Parsificazione: grammatiche LL(1), algoritmo iterativo e algoritmo discendente ricorsivo

Traduzione: definizioni guidate dalla sintassi, schemi di traduzione, traduzione deterministica top-down

Analisi sintattica

Analizzatori discendenti

Costruiscono un albero di parsificazione iniziando dalla radice e creando i nodi in preordine.
Equivalentemente la parsificazione top-down può essre vista come ricerca di una *derivazione leftmost* per la stringa in input.

Analizzatori ascendenti

Costruiscono l'albero dalle foglie alla radice.
L'analisi ascendente può essere vista come ricerca di una *derivazione rightmost* della stringa in ordine contrario.

La parsificazione nella forma più generale può richiedere il backtracking per trovare la produzione corretta da applicare durante l'esame di una stringa. In molte aree dell'informatica i linguaggi sono progettati in modo da essere analizzabili deterministicamente.

Le grammatiche che permettono analisi sintattica top-down deterministica o "*parsing predittivo discendente*" sono chiamate **LL(k)**, quelle che permettono analisi sintattica bottom-up deterministica o "*parsing predittivo ascendente*" sono chiamate **LR(k)**, dove k è il numero di simboli necessari per individuare la produzione senza ambiguità.

Analisi sintattica top-down deterministica

Automa con "*look ahead*" di lunghezza 1.

1. determinare la produzione da applicare quando un nonterminale si trova in cima alla pila.
2. se in cima alla pila si trova un terminale basta verificare la corrispondenza di questo col simbolo sul nastro di input e avanzare la testina di lettura.
3. in mancanza di questa corrispondenza la stringa in input deve essere rifiutata.

Quali sono le condizioni cui deve soddisfare la grammatica per costruire un *analizzatore deterministico discendente* usando 1 simbolo in input? Cioè quando una grammatica è LL(1)?

si associa ad ogni produzione l'«**insieme guida**».

Parsing look ahead 1: insiemi FIRST e FOLLOW

$$FIRST(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\} \cup \{\varepsilon \mid \text{se } \alpha \Rightarrow^* \varepsilon\}$$

F(α) soddisfa queste regole (ricorsive):

1. $F(\varepsilon) = \{\varepsilon\}$
2. $F(a\beta) = \{a\}$
3. $$F(A\beta) = \begin{cases} F(A) & \text{se } \varepsilon \notin F(A) \\ (F(A) - \{\varepsilon\}) \cup F(\beta) & \text{se } \varepsilon \in F(A) \end{cases}$$

dove $F(A) = \bigcup_{i=1}^{K} F(\gamma_i)$ Se $A \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$

$$FOLLOW(A) = \{a \mid S \Rightarrow^* \alpha A a \beta\} \cup \{\$ \mid \text{se } S \Rightarrow^* \alpha A\}$$

FW(A) soddisfa la seguente equazione:

$$FW(A) = \{F(\beta) - \{\varepsilon\} \mid B \rightarrow \alpha A \beta \in P\}$$

$$\{FW(B) \mid B \neq A \text{ e } B \rightarrow \alpha A \beta \in P \ \& \ \varepsilon \in F(\beta)\}$$

$$\{\$ \mid A \text{ è lo start symbol di } G\}$$

Insieme GUIDA e ricorsione sinistra

$$Gui(A \to \alpha) = \begin{cases} F(\alpha) & \text{se } \varepsilon \notin F(\alpha) \\ (F(\alpha) - \{\varepsilon\}) \cup FW(A) & \text{se } \varepsilon \in F(\alpha) \end{cases}$$

Una *grammatica è LL(1)* se per ogni non terminale A e per ogni coppia di produzioni $A \to \alpha$ e $A \to \beta$, gli insiemi guida sono disgiunti:

Eliminazione delle ricorsioni sinistre immediate

$A \to A\alpha_1 | A\alpha_2 | \ldots | A\alpha_k$ $\quad k \geq 1,\ \alpha_i \neq \varepsilon$

$A \to \beta_1 | \beta_2 | \ldots | \beta_h$ $\quad h \geq 1$

$A \to \beta_1 A' | \beta_2 A' | \ldots | \beta_h A'$

$A' \to \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_k A' | \varepsilon$

Analizzatore iterativo

Algoritmo che implementa l'automa a pila look ahead 1.

Input: 1) Stringa da parsificare, seguita da un simbolo di fine stringa: $

2) Tabella M che memorizza, per ogni coppia
< variabile A, simbolo in input *a* >
la produzione $A \to \alpha$ se $a \in Gui(A \to \alpha)$
' ' se $a \notin Gui(A \to \alpha)$.

Output: Stringa accettata o segnalazione di errore;
Elenco delle produzioni usate per generare la stringa.

Analizzatore iterativo

```
AUTOMA( )
        cc ← PROSS( )                    /*1^ simbolo della stringa in input */
        X ← top (STACK)
        while (not empty (STACK))
                if (X ∈ Σ and X = cc)
                        cc ← PROSS( )
                        pop (STACK)
                elseif (X ∈ V and M[X, cc] = X → α and α ≠ cc.β )
                        pop (STACK)
                        push (α, STACK)
                        stampa (X → α)
                elseif (X ∈ V and M[X, cc] = X → α and α = cc.β )
                        cc ← PROSS( )
                        pop (STACK)
                        push (β, STACK)
                        stampa (X → α)
                else ERRORE (…)
                X ← top (STACK)
        if (cc = '$') stampa ("stringa accettata")
        else ERRORE (…)
```

Analizzatore a discesa ricorsiva

Ad ogni variabile A con produzioni

$A \rightarrow \alpha_1$

$A \rightarrow \alpha_2$

...

$A \rightarrow \alpha_k$

si associa una funzione:

function A( )

if (cc $\in$ Gui($A \rightarrow \alpha_1$)) body (α_1)

elseif (cc $\in$ Gui($A \rightarrow \alpha_2$)) body (α_2)

....

elseif (cc $\in$ Gui($A \rightarrow \alpha_k$)) body (α_k)

else ERRORE (…)

Analizzatore a discesa ricorsiva

Se $\alpha = \varepsilon$, body(ε) = do nothing

Se $\alpha = X_1 \ldots X_m$, body($X_1 \ldots X_m$) è così definito:

body($X_1 \ldots X_m$) = act (X_1) act (X_2) … act (X_m)

$$act(X) = \begin{cases} X( ) & \text{se } X \in V \\ cc \leftarrow \text{PROSS}( ) & \text{se } X = cc \\ \text{ERRORE}(\ldots) & \text{altrimenti} \end{cases}$$

```
main discesa_ricorsiva( )
        cc ← PROSS( )
        S( )
        if (cc = '$')  stampa "stringa accettata"
        else ERRORE(…)
```

Definizioni guidate dalla sintassi: SDD

Tecniche di traduzione guidate dalle grammatiche libere.

- si associa informazione a un costrutto di un linguaggio associando "*attributi*" al simbolo della grammatica che rappresenta il costrutto.

- in una *definizione guidata dalla sintassi* (syntax-directed definition) i valori degli attributi sono calcolati da "*regole semantiche*" associate alle produzioni della grammatica.

La traduzione è il risultato della valutazione delle regole semantiche.

Le definizioni guidate dalla sintassi, quando i valori degli attributi sono definiti solo in funzione di costanti e dei valori di altri attributi, vengono anche chiamate *grammatiche ad attributi*.

Gli attributi per i simboli terminali hanno i valori lessicali forniti dall'analizzatore lessicale. Nelle SDD non vi sono regole semantiche per calcolare i valori degli attributi per i simboli terminali.

Definizioni guidate dalla sintassi: SDD

Due tipi di attributi:

- ***sintetizzati***: un attributo sintetizzato per una variabile A in un nodo *n* dell'albero di parsificazione è definito da una regola semantica associata alla produzione in *n* e il suo valore è calcolato solo in termini dei valori degli attributi nei nodi figli di *n* e in *n* stesso.
  (A è il simbolo a sinistra nella produzione, cioè la testa).
- ***ereditati***: un attributo ereditato per una variabile A in un nodo *n* dell'albero di parsificazione è definito da una regola semantica associata alla produzione nel nodo padre di *n* e il suo valore è calcolato solo in termini dei valori degli attributi del padre di *n*, di *n* stesso e dei suoi fratelli.
  (A è un simbolo nel corpo della produzione, cioè al membro destro).

- *albero annotato*
- attributi sintetizzati valutati in ordine *bottom-up*
- negli SDD che hanno sia attributi sintetizzati, sia ereditati, non vi è garanzia che vi sia almeno un ordine in cui valutare gli attributi
- *grafo delle dipendenze*: Se il grafo presenta un ciclo non è possibile valutare gli attributi. L'ordine di valutazione deve rispettare un ordinamento topologico dei vertici del grafo.

Schemi di traduzione: SDT

Gli **schemi di traduzione** (**SDT**) sono un'utile notazione per specificare la traduzione durante la parsificazione.

Uno schema di traduzione è una definizione guidata dalla sintassi in cui le azioni semantiche, racchiuse tra parentesi graffe, sono inserite nei membri destri delle produzioni, in posizione tale che il valore di un attributo sia disponibile quando un'azione fa ad esso riferimento.

Gli schemi di traduzione impongono un ordine di valutazione da sinistra a destra e permettono che nelle azioni semantiche siano contenuti frammenti di programma e in generale side-effect che non influiscano sulla valutazione degli attributi.

Definizioni S-attribuite e L-attribuite

Una SDD è **S-attribuita** se tutti gli attributi sono sintetizzati

Una SDD è **L-attribuita** se gli attributi sono:
- sintetizzati e/o
- ereditati, che soddisfano il seguente vincolo:

per ogni produzione $A \rightarrow X_1 X_2 \ldots X_n$, ogni attributo ereditato di X_j dipende solo da:

- attributi ereditati o sintetizzati dei simboli $X_1 X_2 \ldots X_{j-1}$ a sinistra di X_j nella produzione;
- attributi ereditati di A;
- altri attributi di X_j, purchè non vi siano cicli nel grafo delle dipendenze formati dagli attributi di questa occorrenza di X_j.

Trasformazione di SDD S-attribuite e L-attribuite in Schemi di Traduzione (con azioni specificate al momento giusto)

Tipicamente gli SDT sono implementati durante la parsificazione, in particolare in due casi:

- SDD L-attribuita e grammatica LL-parsificabile
- SDD S-attribuita e grammatica LR-parsificabile

Valutazione top-down di SDD L-attribuite

```
main traduzione_discesa_ricorsiva( )
  var risultato
        cc ← PROSS
         risultato ← S( )
         if (cc = '$') stampa ("stringa corretta,
                    la sua traduzione è:" risultato)
         else ERRORE(…)
```

```
function A(e1, . . . en)
     var s1,...,sm, X1_a1,...,X1_ak,...,Xh_a1,...,Xh_ar
           if (cc ∈ Gui(A → α1)) body'(α1)
           elseif (cc ∈ Gui(A → α2)) body'(α2)
           ….
           elseif (cc ∈ Gui(A → αk)) body'(αk)
           else ERRORE(…)
           return (<s1, . . . , sm>)
```

Codice per le parti destre delle produzioni (body'(α_i)):

- Per ogni terminale i valori degli attributi vengono assegnati alle corrispondenti variabili e l'esame passa al simbolo successivo.
- Per ogni non terminale B si genera un'assegnazione c ← B(b_1, . . . b_n), che è una chiamata alla funzione associata a B.
  Poiché la SDD è L-attribuita, gli argomenti (attributi) saranno già stati calcolati e memorizzati nelle variabili locali.
- Le azioni semantiche vengono ricopiate dopo aver sostituito i riferimenti agli attributi con le variabili corrispondenti.

Il linguaggio $\mathbb{P}$ da tradurre

Programma

P ::= SL EOF

Istruzioni

- S → **ID** := E
- S → `print` (E)
- S → `read` (ID)
- S → `while` (B) S
- S → `if` (B) S
- S → `if` (B) S `else` S
- S → {SL}
- SL → SL ; S
- SL → S

Espressioni aritmetiche

- E → T E'
- E' → + T E'
- E' → - T E'
- E' → ε

- T → F T'
- T' → * F T'
- T' → / F T'
- T' → ε

- F → (E)
- F → **NUM**
- F → **ID**

Espressioni booleane

- B → E RELOP E

Dal linguaggio P al bytecode: espressioni aritmetiche

Per le espressioni aritmetiche usiamo un attributo sintetizzato *.code* il cui valore è la traduzione dell'espressione nel bytecode.

$E \rightarrow T\ E'$ {E.*code* = T.*code* || E'.*code*}

$E' \rightarrow +\ T\ E'_1$ {E'.*code* = T.*code* || "iadd" || E'_1.*code*}

$E' \rightarrow -\ T\ E'_1$ {E'.*code* = T.*code* || "isub" || E'_1.*code*}

$E' \rightarrow \varepsilon$ {E'.*code* = " "}

$T \rightarrow F\ T'$ {T.*code* = F.*code* || T'.*code*}

$T' \rightarrow *\ F\ T'_1$ {T'.*code* = F.*code* || "imul" || T'_1.*code*}

$T' \rightarrow /\ F\ T'_1$ {T'.*code* = F.*code* || "idiv" || T'_1.*code*}

$T' \rightarrow \varepsilon$ {T'.*code* = " "}

$F \rightarrow (E)$ {F.*code* = E.*code*}

$F \rightarrow$ **NUM** {F.*code* = ldc(**NUM**.val)}

$F \rightarrow$ **ID** {F.*code* = iload(*addr*(**ID**.lessema)}

Dal linguaggio ℙ al bytecode: espressioni booleane

Per le espressioni booleane usiamo tre attributi:

a) .*code* sintetizzato per la traduzione in bytecode
b) .*true* e .*false* , attributi ereditati che servono per definire le label delle prime istruzioni che traducono gli statement da eseguire nei casi B vero e B falso rispettivamente.

$B \rightarrow E_1 == E_2$ {B.*code* = E_1.*code* || E_2.*code* ||
|| ‘if_icmpeq’ B.*true* || ‘goto’ B.*false*}

Dal linguaggio ℙ al bytecode: SDT istruzioni

P → {SL.*next* = *newlabel*( )} SL EOF {P.*code* = SL.*code* || *label*(SL.*next*) || 'stop'}

S → **ID** := E {S.*code* = E.*code* || istore(*addr*(**id**.lessema))}

S → `if` ({B.*true* = *newlabel*( ) ; B.*false* = S.*next*}
B) {S_1.*next* = S.*next*} S_1 {S.*code* = B.*code* || label(B.*true*) || S_1.*code*}

S → `if` ({B.*true* = *newlabel*( ) ; B.*false* = *newlabel*( )}
B) {S_1.*next* = S.*next*} S_1 `else`
{S_2.*next* = S.*next*} S_2 {S.*code* = B.*code* || *label*(B.*true*) || S_1.*code* ||
|| '*goto*' S_1.*next*) || *label*(B.*false*) || S_2.*code*}

S → `while(` {begin = *newlabel*( ) ; B.*true* = *newlabel*( ) ; B.*false* = S.*next*}
B) {S_1.*next* = begin}
S_1 {S.*code* = *label*(begin) || B.*code* || *label*(B.*true*) || S_1.*code* || 'goto S_1.*next*}

S → { {SL.*next* = S.*next*} SL } {S.*code* = SL.*code*}

SL → {SL_1.*next* = *newlabel*( )} SL_1 ; {S.*next* = SL.*next*}
S {SL.*code* = SL_1.*code* || *label*(SL_1.*next*) || S.*code*}

SL → {S.*next* = SL.*next*} S {SL.*code* = S.*code*}

Dal linguaggio ℙ al bytecode: espressioni aritmetiche

Sia per le espressioni aritmetiche, sia, e soprattutto, per gli statement, i valori degli attributi .*code* sono stringhe piuttosto lunghe.

Ad esempio, per lo statement: `while` (x < 100) y := x * 2

S.*code* = L2 iload(*addr*(x)) ldc 100 if_icmplt L3 goto L1 L3
iload(*addr*(x)) ldc 2 imul istore(*addr*(y)) goto L2

1. L'attributo principale è sintetizzato
2. Le regole semantiche sono tali che:
   a) L'attributo è ottenuto dal concatenamento di stringhe
   b) Gli attributi dei non terminali si presentano nella regola semantica nello stesso ordine in cui i non terminali si presentano nel corpo della produzione

Traduzione on-the-fly

Schemi di traduzione ‘on-the-fly’: espressioni

Espressioni aritmetiche

$E \rightarrow T\ E'$

$E' \rightarrow + T$ {*emit*(iadd)} E'_1

$E' \rightarrow - T$ {*emit*(isub)} E'_1

$E' \rightarrow \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow * F$ {*emit*(imul)} T'_1

$T' \rightarrow / T$ {*emit*(idiv)} T'_1

$T' \rightarrow \varepsilon$

$F \rightarrow (E)$

$F \rightarrow$ **NUM** {*emit*(ldc(**NUM**.val))}

$F \rightarrow$ **ID** {*emit*(iload(*addr*(**ID**.lessema))}

Espressioni booleane

$B \rightarrow E_1 == E_2$ {*emit*(‘if_icmpeq’ B.*true* ;
emit(‘goto’ B.*false*}

Schemi di traduzione ‘on-the-fly’: statement

P → {SL.*next* = *newlabel*( )} SL {*emitlabel*(SL.*next*)} EOF {*emit*(‘stop’)}

S → **ID** := E {*emit*(istore (*addr*(**ID.**lessema)))}

S → `if` ({B.*true* = *newlabel*( ) ; B.*false* = S.*next*}
B) {*emitlabel*(B.*true*) ; S_1.*next* = S.*next*} S_1

S → `if` ({B.*true* = *newlabel*( ) ; B.*false* = newlabel( )}
B) {*emitlabel*(B.*true*) ; S_1.*next* = S.*next* }
S_1 {*emit*(‘goto’ S_1.*next*))} `else` {*emitlabel*(B.*false*), S_2.*next* = S.*next* } S_2

S → `while(` {begin=*newlabel*(); *emitlabel*(begin); B.*true*=*newlabel*( ); B.*false*=S.*next*}
B) {*emitlabel*(B.*true*)); S_1.*next* = begin}
S_1 {*emit*(‘goto’ S_1.*next*)}

S → { {SL.*next* = S.*next*} SL }

SL → {SL_1.*next* = *newlabel*( )} SL_1 ; {*emitlabel*(SL_1.*next*) ; S.*next* = SL.*next*} S

SL → {S.*next* = SL.*next*} S

Schemi di traduzione ‘on-the-fly’: statement

P → {SL.*next* = *newlabel*( )} SL {*emitlabel*(SL.*next*)} EOF {*emit*(‘stop’)}

S → **ID** := E {*emit*(istore (*addr*(**ID.**lessema)))}

S → `if` ({B.*true* = *newlabel*( ) ; B.*false* = S.*next*}
B) {*emitlabel*(B.*true*) ; S_1.*next* = S.*next*} S_1

S → `if` ({B.*true* = *newlabel*( ) ; B.*false* = newlabel( )}
B) {*emitlabel*(B.*true*) ; S_1.*next* = S.*next* }
S_1 {*emit*(‘goto’ S_1.*next*))} `else` {*emitlabel*(B.*false*), S_2.*next* = S.*next* } S_2

S → `while(` {begin=*newlabel*(); *emitlabel*(begin); B.*true*=*newlabel*( ); B.*false*=S.*next*}
B) {*emitlabel*(B.*true*)); S_1.*next* = begin}
S_1 {*emit*(‘goto’ S_1.*next*)}

S → { {SL.*next* = S.*next*} SL }

SL → {SL_1.*next* = *newlabel*( )} SL_1 ; {*emitlabel*(SL_1.*next*) ; S.*next* = SL.*next*} S

SL → {S.*next* = SL.*next*} S