Università di Roma "La Sapienza"
corso di laurea in Ingegneria Informatica

Lezioni del corso di

Progetto di Linguaggi e Traduttori

a.a. 2006/2007

Riccardo Rosati
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma

Obiettivi del corso

Il corso approfondisce le problematiche relative ai processi di traduzione ed interpretazione di linguaggi formali, con l'obiettivo di fornire le tecniche, le metodologie e gli strumenti per la realizzazione di traduttori guidati dalla sintassi, sulla base della decomposizione di un traduttore nei moduli di analisi lessicale, analisi sintattica, traduzione guidata dalla sintassi.

Inoltre il corso introduce i linguaggi per il web, in particolare XML, e li esamina dal punto di vista delle problematiche dell'analisi sintattica.

Infine, i metodi e le tecniche acquisiti vengono concretamente applicati nella progettazione e realizzazione di un traduttore guidato dalla sintassi.

Programma del corso

1. progetto di un traduttore (14 ore)
   (a) linguaggi formali e traduttori (2 ore)
   (b) analisi lessicale (4 ore)
   (c) analisi sintattica (4 ore)
   (d) traduzione guidata dalla sintassi (4 ore)
2. il linguaggio XML (10 ore)
3. laboratorio (26 ore):
   (a) presentazione JavaCC (2 ore)
   (b) presentazione strumenti per XML (2 ore)
   (c) presentazione proposte di progetto (2 ore)
   (d) realizzazione di un traduttore (20 ore)

Informazioni sul corso

Materiale didattico:

- dispense e lucidi distribuiti dal docente
- esercitazioni svolte

Sito web del corso:

`http://www.dis.uniroma1.it/~rosati/lingtrad`

Lezione 1

Progetto di un traduttore

Traduttori

concetto di traduttore in informatica:

- un traduttore è un programma che effettua la traduzione automatica da un linguaggio ad un altro
- i linguaggi utilizzati nelle applicazioni informatiche sono *linguaggi formali*
- data una frase f_1 nel linguaggio formale L_1 (linguaggio sorgente), il traduttore costruisce una frase f_2 del linguaggio formale L_2 (linguaggio destinazione)
- la frase f_2 deve "corrispondere" a f_1

Linguaggi formali

definizione matematica di linguaggio:

- alfabeto $\mathcal{A}$ = insieme di elementi detti simboli o *caratteri*
- stringa su $\mathcal{A}$ = sequenza finita di caratteri di $\mathcal{A}$
- linguaggio su $\mathcal{A}$ = insieme (finito o infinito) di stringhe su $\mathcal{A}$

Pertanto:

definizione di un linguaggio = definizione di un insieme di stringhe (stabilisce quali stringhe appartengono al linguaggio e quali stringhe non appartengono)

$\Rightarrow$ la teoria dei linguaggi formali si occupa di definire la *sintassi* di un linguaggio, ma non la sua *semantica*

Sintassi e semantica di un linguaggio

sintassi = aspetti formali del linguaggio (struttura delle frasi)

semantica = "significato" associato alle frasi del linguaggio

metodi di definizione della sintassi:

- uniforme: vengono usati gli stessi metodi di definizione per tutti i linguaggi
- in particolare, vengono usati formalismi per la specifica di linguaggi formali

metodi di definizione della semantica:

- non uniforme: la modalità di definizione varia da linguaggio a linguaggio
- ciò è dovuto alla maggiore difficoltà di esprimere gli aspetti semantici

la traduzione dipende sia dalla sintassi che dalla semantica dei linguaggi sorgente e destinazione

Semantica

definire la semantica di un linguaggio L significa definire una funzione f che associa ogni frase (stringa) di L ad elementi di un dominio di interpretazione I

$$f : L \rightarrow I$$

problemi:

- i domini di interpretazione I utilizzati variano enormemente da linguaggio a linguaggio
- i domini di interpretazione possono essere molto complessi

Semantica

esempi (in ordine crescente di complessità):

1. logica proposizionale: si associano le stringhe del linguaggio (formule della logica proposizionale) a valori di verità (vero, falso)
2. linguaggio HTML: si associa ogni stringa del linguaggio ad un "ipertesto"
3. linguaggi di programmazione imperativi: si associano stringhe del linguaggio (programmi) ad evoluzioni di un modello di calcolo (macchina di von Neumann)

conseguenza: non è possibile trattare in modo uniforme la definizione della semantica di un linguaggio

Formalismi per la specifica della sintassi

vengono utilizzati i formalismi per la specifica di linguaggi formali:

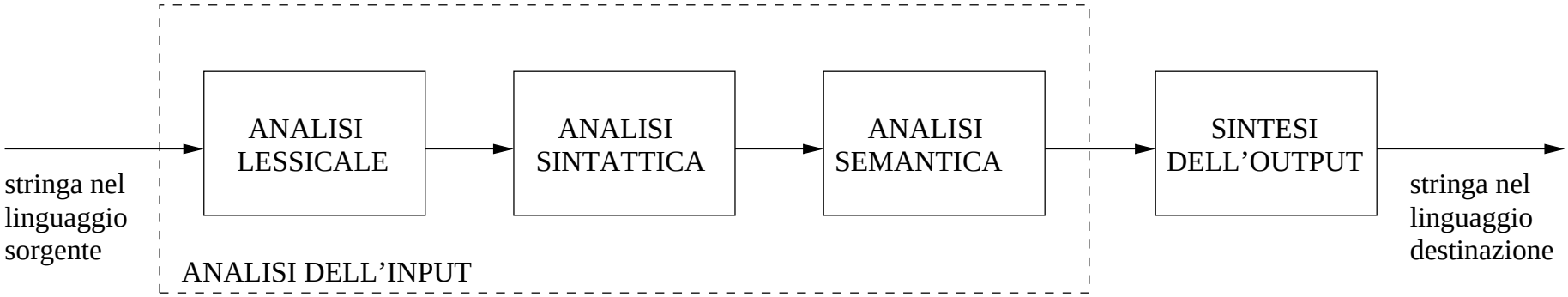
- espressioni regolari
- automi a stati finiti
- grammatiche non contestuali (BNF, EBNF)
- altri (DTD, XML schema,...)

Lessico di un linguaggio

In alcuni linguaggi è opportuno distinguere il *lessico* dalla sintassi complessiva del linguaggio

- elementi lessicali di un linguaggio = le parole ammesse dal linguaggio
- esempio: lessico del linguaggio Java = tutti gli identificatori, le costanti intere, le costanti reali, le parole chiave, stringhe tra doppi apici, ecc.
- le parole ammesse sono chiamate *token*
- con questa distinzione, il lessico stabilisce le regole di costruzione dei token (ognuno dei quali è una sequenza di caratteri), mentre la restante parte della sintassi stabilisce le regole di costruzione delle frasi del linguaggio (ogni frase è una sequenza di token)

Struttura di un traduttore



due fasi:

1. fase di *analisi dell'input*, che si distingue ulteriormente in:
   (a) analisi lessicale
   (b) analisi sintattica
   (c) analisi semantica
2. fase di *sintesi dell'output*

Analisi dell'input e sintesi dell'output

Rispetto alle caratteristiche del linguaggio sorgente e del linguaggio destinazione:

- le fasi di analisi lessicale e analisi sintattica dipendono *soltanto* dalla sintassi del linguaggio sorgente
- la fase di analisi semantica dipende sia dalla sintassi che dalla semantica del linguaggio sorgente
- la fase di sintesi dell'output dipende sia da sintassi e semantica del linguaggio sorgente che da sintassi e semantica del linguaggio destinazione

Analizzatore lessicale

modulo di analisi lessicale = analizzatore lessicale

- il lessico è in genere specificato in termini di un linguaggio regolare
- pertanto, per la specifica del lessico vengono utilizzati formalismi per la specifica di linguaggi regolari (espressioni regolari, automi a stati finiti)
- il modulo di analisi lessicale (analizzatore lessicale) viene realizzato sulla base di tale specifica
- in genere si utilizzano dei programmi chiamati *generatori di analizzatori lessicali* che, a partire dalla specifica del lessico, generano automaticamente l'analizzatore lessicale corrispondente

Analizzatore sintattico

modulo di analisi sintattica = analizzatore sintattico

- la sintassi è in genere specificata in termini di un linguaggio non contestuale
- pertanto, per la specifica della sintassi vengono utilizzati formalismi per la specifica di linguaggi non contestuali (grammatiche, EBNF, DTD)
- il modulo di analisi sintattica (analizzatore sintattico) viene realizzato sulla base di tale specifica
- in genere si utilizzano dei programmi chiamati *generatori di analizzatori sintattici* che, a partire dalla specifica della sintassi, generano automaticamente l'analizzatore sintattico corrispondente

Analisi semantica e sintesi dell'output

- la fase di analisi semantica dell'input dipende dalla semantica del linguaggio sorgente
- la fase di sintesi dell'output dipende sia dalla semantica del linguaggio sorgente che dalla semantica del linguaggio destinazione
- data la complessità della definizione della semantica di un linguaggio, le fasi di analisi semantica dell'input e di sintesi dell'output *non* possono essere automatizzate, come invece avviene per le fasi di analisi lessicale e analisi sintattica
- pertanto, i moduli che realizzano le fasi di analisi semantica e sintesi dell'output devono essere realizzati manualmente (non esistono strumenti di generazione automatica)

Traduzione guidata dalla sintassi

- la fase di traduzione (sintesi dell'output) può essere *guidata dalla sintassi*, cioè si può decomporre il processo di traduzione delle frasi del linguaggio sorgente sulla base della struttura sintattica di tali frasi
- a tal fine, è possibile estendere i formalismi di specifica della sintassi al fine di catturare alcuni aspetti "semantici"
- in particolare, vengono utilizzate le *grammatiche ad attributi*, che estendono le grammatiche non contestuali mediante *azioni semantiche*, collegate alle regole di produzione della grammatica
- tramite le azioni semantiche, è possibile specificare la fase di traduzione in parallelo alla specifica della sintassi del linguaggio sorgente

Lezione 2

Analisi lessicale

Richiami sui linguaggi formali

operazione sulle stringhe: concatenazione

se $s_1 = abc$ e $s_2 = def$,
la concatenazione di s_1 e s_2 è denotata da $s_1 s_2$, e $s_1 s_2 = abcdef$

costante: stringa vuota, denotata da ϵ

Richiami sui linguaggi formali

operazioni sui linguaggi:

1. operazioni insiemistiche, unione, intersezione, ...
2. concatenazione di linguaggi: $L_1 L_2 = \{st \mid s \in L_1 \text{ e } t \in L_2\}$
3. potenza di un linguaggio: L^n = concatenazione di L con se stesso n volte ($L^0 = \{\epsilon\}$)
4. chiusura di un linguaggio: $L^* = L^0 \cup L^1 \cup \ldots \cup L^n \cup \ldots$ (unione di tutte le potenze di L)
5. chiusura positiva di un linguaggio: $L^+ = L^1 \cup \ldots \cup L^n \cup \ldots$ (unione di tutte le potenze di L tranne L^0)
6. linguaggio universale su un alfabeto $\mathcal{A}$ (denotato da $\mathcal{A}^*$) $= (L_A)^*$, dove L_A è il linguaggio $\{a \mid a \in \mathcal{A}\}$

Richiami sui linguaggi formali

Gerarchia di Chomsky: 4 classi di linguaggi

1. linguaggi di tipo 0 o ricorsivamente enumerabili
2. linguaggi di tipo 1 o contestuali
3. linguaggi di tipo 2 o non contestuali
4. linguaggi di tipo 3 o regolari

in questo corso trattiamo solo linguaggi di tipo 2 e di tipo 3

- linguaggi di tipo 2 = i linguaggi generati dalle grammatiche non contestuali
- linguaggi di tipo 3 = i linguaggi esprimibili tramite espressioni regolari

la classe dei linguaggi di tipo 3 è contenuta nella classe dei linguaggi di tipo 2, ma non viceversa

Analisi lessicale

- **problema di base** dell'analisi lessicale: data una stringa (sequenza di caratteri) e una specifica del lessico, stabilire se la stringa è una parola del lessico

- il lessico è in genere specificato in termini di un linguaggio regolare

- pertanto, il problema equivale al problema del *riconoscimento* nei linguaggi regolari: data una stringa e un linguaggio regolare, stabilire se la stringa appartiene al linguaggio

Formalismi per la specifica del lessico

Vengono utilizzati formalismi per la specifica di linguaggi regolari (tipo 3).
In particolare, sono usati:

1. *espressioni regolari*:
   - espressioni che usano i caratteri di un alfabeto iniziale A, il simbolo speciale ϵ ed alcuni operatori ($*$, $|$, $+$, ?)
   - ad ogni espressione regolare e è associato un linguaggio $\mathcal{L}(e)$ sull'alfabeto A ($\mathcal{L}(e)$ = linguaggio denotato da e)
2. *automi a stati finiti*:
   - modello di calcolo costituito da un grafo orientato etichettato sugli archi
   - ad ogni automa a stati finiti F è associato un linguaggio $\mathcal{L}(F)$ sull'alfabeto delle etichette degli archi ($\mathcal{L}(F)$ = linguaggio riconosciuto da F)

Espressioni regolari: richiami

Dato un alfabeto $\mathcal{A}$, l'insieme delle espressioni regolari su $\mathcal{A}$ è l'insieme E definito induttivamente come segue:

1. $\epsilon \in E$ e $\mathcal{L}(\epsilon) = \{\epsilon\}$;
2. se $a \in \mathcal{A}$, allora $a \in E$ e $\mathcal{L}(a) = \{a\}$;
3. se $e_1 \in E$ e $e_2 \in E$, allora $e_1 e_2 \in E$ e $\mathcal{L}(e_1 e_2) = \mathcal{L}(e_1)\mathcal{L}(e_2)$;
4. se $e_1 \in E$ e $e_2 \in E$, allora $e_1 \mid e_2 \in E$ e $\mathcal{L}(e_1 \mid e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$;
5. se $e \in E$, allora $e^* \in E$ e $\mathcal{L}(e^*) = \mathcal{L}(e)^*$;
6. se $e \in E$, allora $e^+ \in E$ e $\mathcal{L}(e^+) = \mathcal{L}(e)^+$;
7. se $e \in E$, allora $e? \in E$ e $\mathcal{L}(e?) = \mathcal{L}(e) \cup \{\epsilon\}$;
8. se $e \in E$, allora $(e) \in E$ e $\mathcal{L}((e)) = \mathcal{L}(e)$.

Espressioni regolari: esempi

$$e_0 = (a|b)^*$$
$$\mathcal{L}(e_0) = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \ldots\}$$

$$e_1 = (a|b)c^*$$
$$\mathcal{L}(e_1) = \{a, b, ac, bc, acc, bcc, accc, bccc, acccc, bcccc, \ldots\}$$

$$e_2 = (a|b)^*c$$
$$\mathcal{L}(e_2) = \{c, ac, bc, aac, abc, bac, bbc, aaac, aabc, abac, \ldots\}$$

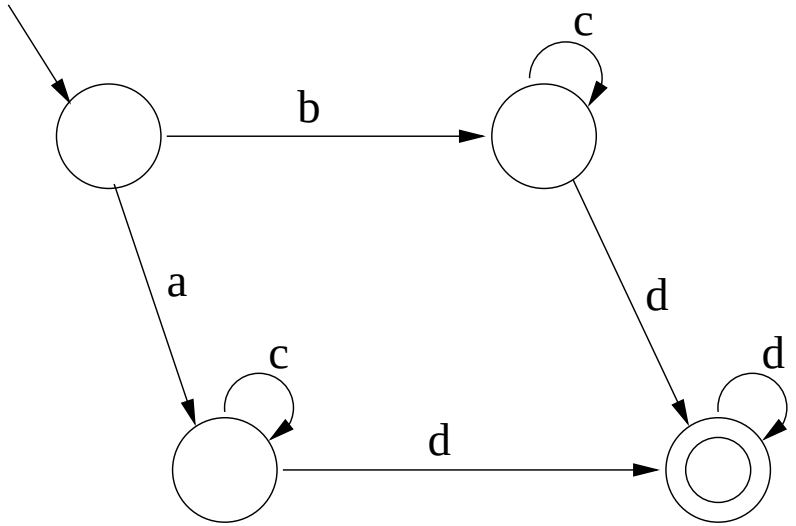
$$e_3 = (A|B|\ldots|Z|a|b|\ldots|z)(A|B|\ldots|Z|a|b|\ldots|z|0|1|\ldots|9)^*$$
$\mathcal{L}(e_3)$ è l'insieme degli identificatori ammessi in Java

$$e_4 = (+|-)?(0|1|\ldots|9)^*.(0|1|\ldots|9)^+((\mathrm{E}|\mathrm{e})(+|-)?(0|1|\ldots|9)^+)?$$
$\mathcal{L}(e_4)$ è l'insieme delle costanti reali ammesse in Java

Automi a stati finiti: richiami

- un automa a stati finiti è costituito da un insieme di stati (nodi) e un insieme di transizioni di stato (archi orientati tra coppie di nodi)
- un nodo speciale è detto stato iniziale (denotato da un arco senza nodo di partenza)
- uno o più nodi speciali sono detti stati finali (ogni stato finale è denotato da una doppia cerchiatura)
- gli archi sono etichettati
- un automa a stati finiti F riconosce un linguaggio $\mathcal{L}(F)$ sull'alfabeto delle etichette degli archi

Automi a stati finiti: richiami



- una stringa s appartiene al linguaggio $\mathcal{L}(F)$ se e solo se la sequenza di caratteri che costituisce s corrisponde alla sequenza delle etichette di un cammino dallo stato iniziale ad uno stato finale nel grafo F
- ad esempio, dato l'automa F in figura, la stringa acd appartiene a $\mathcal{L}(F)$, mentre la stringa abd non appartiene a $\mathcal{L}(F)$

Scopi dell'analisi lessicale

Compito di un analizzatore lessicale:

data una sequenza di caratteri (stringa) di un alfabeto, verificare se la sequenza può essere decomposta in una sequenza di token (parole), tale che ogni token appartiene al lessico:

- in caso positivo, restituire in uscita la sequenza di token
- in caso negativo, restituire un errore lessicale

Scopi dell'analisi lessicale

Esempio (Java):

stringa di input:

```
"class prova {
  // classe di prova
  static int numeri []=new int[20];
  static int count=0;
  public static void inizializza () { ..."
```

sequenza di token di output:

```
"class", "prova", "{", "static", "int", "numeri", "[", "]", "=", "new",
"int", "[", "20", "]", ";", ...
```

Caratteristiche di un analizzatore lessicale

- specifica del lessico = insieme di espressioni regolari
- il lessico è l'unione dei linguaggi denotati dalle espressioni regolari
- l'analizzatore lessicale riceve in input una sequenza di caratteri, e legge un carattere per volta (da sinistra verso destra)
- l'analizzatore opera in modo iterativo, e ad ogni iterazione cerca di generare una sottosequenza (token) che costituisce una stringa che appartiene al linguaggio denotato da una delle espressioni regolari del lessico
- ad ogni iterazione cerca di generare il token *di lunghezza massima* tra quelli ammissibili
- se invece non riesce a generare un token ammesso dal lessico, si ferma restituendo un errore (detto errore lessicale)

Esempio

- e_1: (a|b|...|z) (a|b|...|z|0|1|...|9)*
- e_2: if
- e_3: else
- e_4: {
- e_5: }
- e_6: ␣ (spazio bianco)

stringa in input: if␣pippo␣{pluto}␣else␣paperino

l'analizzatore restituisce la sequenza di token:

if, ␣, pippo, ␣, {, pluto, }, ␣, else, ␣, paperino

Token non restituiti

- spesso, alcuni di token non hanno importanza per la successiva fase di analisi sintattica e possono essere quindi non restituiti dall'analizzatore lessicale
- esempi:
  - i simboli di spaziatura (blank, tab, a capo), che in molti linguaggi hanno l'unico scopo di separare tra loro i token
  - i commenti nei linguaggi di programmazione
- nell'esempio precedente, è possibile che i simboli di spaziatura corrispondenti all'espressione e_6 possano non essere restituiti (perché non significativi per la successiva fase di analisi sintattica)
- in tal caso, la sequenza di token restituita dall'analizzatore lessicale è: `if`, `pippo`, {, `pluto`, }, `else`, `paperino`

Classificazione dei token

- in generale, nell'analisi lessicale vengono distinti diversi tipi (o classi) di elementi lessicali
- ad esempio, i token non restituiti visti in precedenza costituiscono una particolare classe di token
- la classificazione viene ottenuta associando una classe (o tipo) diversa ad ogni espressione regolare (è questo il motivo per cui nella definizione del lessico si usano più espressioni regolari invece di costruire un'unica espressione regolare)
- nel lessico dell'esempio precedente, ci sono 6 tipi di token diversi, corrispondenti alle 6 espressioni regolari $e_1, \ldots, e_6$
- ad ogni token viene associato il tipo corrispondente all'espressione regolare che riconosce il token
- l'attribuzione dei tipi ai token facilita la successiva analisi sintattica

Lezione 3

Analisi sintattica

Analisi sintattica

- **problema di base** dell'analisi sintattica: data una sequenza di token e una specifica della sintassi, stabilire se la sequenza è una frase ammessa dalla sintassi

- la sintassi è in genere specificata in termini di un linguaggio non contestuale (tipo 2), in cui ogni token costituisce un simbolo atomico del linguaggio

- pertanto, il problema equivale al problema del *riconoscimento* nei linguaggi non contestuali: data una stringa (sequenza di simboli) e un linguaggio non contestuale, stabilire se la stringa appartiene al linguaggio

Formalismi per la specifica della sintassi

vengono utilizzati i formalismi per la specifica di linguaggi non contestuali (tipo 2):

- grammatiche non contestuali (BNF, EBNF)
- altri (DTD, XML schema,...)

nel seguito vengono presentate le grammatiche non contestuali (DTD e XML schema vengono presentati nella seconda parte del corso)

Grammatiche non contestuali

- formalismo per la specifica di linguaggi formali
- più precisamente, una grammatica è un meccanismo *generativo* di un linguaggio tramite la nozione di *derivazione*

grammatica = quadrupla $\langle V_T, V_N, S, P \rangle$, dove:

- V_T è un insieme di simboli detto *alfabeto dei simboli terminali*
- V_N è un insieme di simboli detto *alfabeto dei simboli non terminali* (con $V_T \cap V_N = \emptyset$)
- S è un simbolo di V_N detto *assioma* (o simbolo iniziale) della grammatica
- P è un insieme di *regole di produzione*
- ogni regola di produzione è un'espressione del tipo $A \rightarrow \alpha$, dove $A \in V_N$ e $\alpha \in (V_T \cup V_N)^*$

Notazione EBNF

Arricchisce il formalismo delle grammatiche con gli operatori delle espressioni regolari

1) $A \rightarrow \alpha_1 \mid \alpha_2$ corrisponde alle regole $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$

2) $A \rightarrow (\beta)?$ corrisponde alle regole $A \rightarrow \epsilon$, $A \rightarrow \beta$

3) $A \rightarrow (\beta)^*$ corrisponde alle regole $A \rightarrow \epsilon$, $A \rightarrow \beta A$

4) $A \rightarrow (\beta)^+$ corrisponde alle regole $A \rightarrow \beta$, $A \rightarrow \beta A$

Derivazione

una grammatica è associata ad un linguaggio attraverso il concetto di derivazione di una stringa

data una grammatica $G = \langle V_T, V_N, S, P \rangle$, diciamo che:

- $\beta\alpha\gamma$ *deriva direttamente* da $\beta A\gamma$ in G (denotato da $\beta A\gamma \Rightarrow \beta\alpha\gamma$) se esiste una regola $A \rightarrow \alpha$ in P
- β *deriva* da α in G (denotato da $\alpha \Rightarrow^* \beta$) se esistono $\gamma_1 \in (V_T \cup V_N)^* \ldots, \gamma_k \in (V_T \cup V_N)^*$ con $k \geq 0$ tali che

$$\alpha \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_k \Rightarrow \beta$$

la derivazione usa le regole di produzione della grammatica come regole di riscrittura

Derivazione: esempi

Data la grammatica G contenente le regole di produzione

$$S \rightarrow aSb \mid ab \mid c$$

si ha ad esempio:

$S \Rightarrow aSb \Rightarrow acb$, pertanto $S \Rightarrow^* acb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aacbb$, pertanto $S \Rightarrow^* aacbb$

Data la grammatica G_2 contenente le regole di produzione

$$\begin{aligned} S &\rightarrow cSc \mid A \mid \epsilon \\ A &\rightarrow aA \mid a \end{aligned}$$

si ha $S \Rightarrow cSc \Rightarrow ccScc \Rightarrow ccAcc \Rightarrow ccaAcc \Rightarrow ccaaAcc \Rightarrow ccaaacc$,
pertanto $S \Rightarrow^* ccaaacc$

Linguaggio generato da una grammatica

- *forma di frase* di G = ogni stringa sull'alfabeto $V_T \cup V_N$ derivata dall'assioma S

- *frase* di G = ogni stringa sull'alfabeto V_T derivata dall'assioma S

- l'insieme delle frasi di G è detto *linguaggio generato dalla grammatica* G e si scrive $\mathcal{L}(G)$

Linguaggio generato da una grammatica: esempi

Data la grammatica G_1 contenente le regole di produzione

$$S \rightarrow aSb \mid ab \mid c$$

si ha $\mathcal{L}(G_1) = \{c, ab, acb, aabb, aacbb, aaabbb, aaacbbb, aaaabbbb, \ldots\}$

Data la grammatica G_2 contenente le regole di produzione

$$\begin{aligned} S &\rightarrow cSc \mid A \mid \epsilon \\ A &\rightarrow aA \mid a \end{aligned}$$

si ha $\mathcal{L}(G_2) = \{\ \epsilon, a, aa, aaa, aaaa, aaaaa, \ldots$
$cc, cac, caac, caaac, caaaac, \ldots$
$cccc, ccacc, ccaacc, ccaaacc, ccaaaacc, \ldots,$
$cccccc, cccaccc, cccaaccc, cccaaaccc, cccaaaaccc, \ldots$
$\ldots\ \}$

Albero sintattico

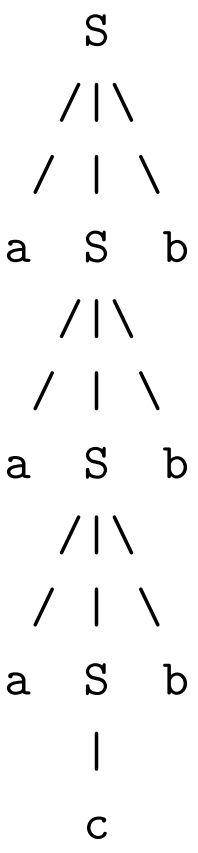
Albero sintattico (o albero di derivazione) di una frase s di G = albero tale che:

- etichetta della radice = assioma di G
- ogni nodo non foglia è etichettato con un simbolo di V_N
- ogni nodo foglia è etichettato con un simbolo di V_T
- le etichette delle foglie, lette da sinistra verso destra, corrispondono alla stringa s
- per ogni nodo non foglia, se A è la sua etichetta, allora esiste una regola di produzione $A \rightarrow \alpha$ in G tale che la sequenza delle etichette dei figli del nodo corrisponde ad α

una stringa s appartiene a $\mathcal{L}(G)$ se e solo se esiste un albero sintattico per s in G

Albero sintattico: esempio

Data la grammatica G contenente le regole di produzione $S \rightarrow aSb \mid ab \mid c$, un albero sintattico per la stringa $aaacbbb$ è:



Scopi dell'analisi sintattica

data una sequenza s di token, l'analizzatore sintattico verifica se la sequenza appartiene al linguaggio generato dalla grammatica G

a questo scopo, l'analizzatore sintattico cerca di costruire l'albero sintattico di s

- in caso positivo, restituisce in uscita l'albero sintattico per la sequenza di input nella grammatica G
- in caso negativo, restituisce un errore (errore sintattico)

Tipi di analisi sintattica

Due tipi di analisi sintattica:

- **analisi top-down**: l'albero sintattico della stringa di input viene generato a partire dalla radice (assioma), scendendo fino alle foglie (simboli terminali della stringa di input)
- **analisi bottom-up**: l'albero sintattico della stringa di input viene generato a partire dalle foglie (simboli terminali della stringa di input), risalendo fino alla radice (assioma)

Caratteristiche di un analizzatore sintattico

- effettuare il riconoscimento di una stringa rispetto ad una grammatica è un'operazione intrinsecamente *nondeterministica*
- questo comporta in generale tempi esponenziali per risolvere il problema del riconoscimento, e quindi dell'analisi sintattica
- tuttavia, l'analizzatore sintattico deve essere molto *efficiente*. In particolare:
  1. tipicamente si richiede all'analizzatore sintattico di girare in tempi lineari rispetto alle dimensioni della stringa di input
  2. inoltre, si richiede all'analizzatore di costruire l'albero sintattico leggendo pochi simboli della stringa di input per volta (tipicamente un solo simbolo della stringa per volta, detto simbolo di *lookahead*)
- a questo scopo, è necessario che la grammatica (in particolare, le regole di produzione) abbia delle caratteristiche che la rendono adatta all'analisi sintattica

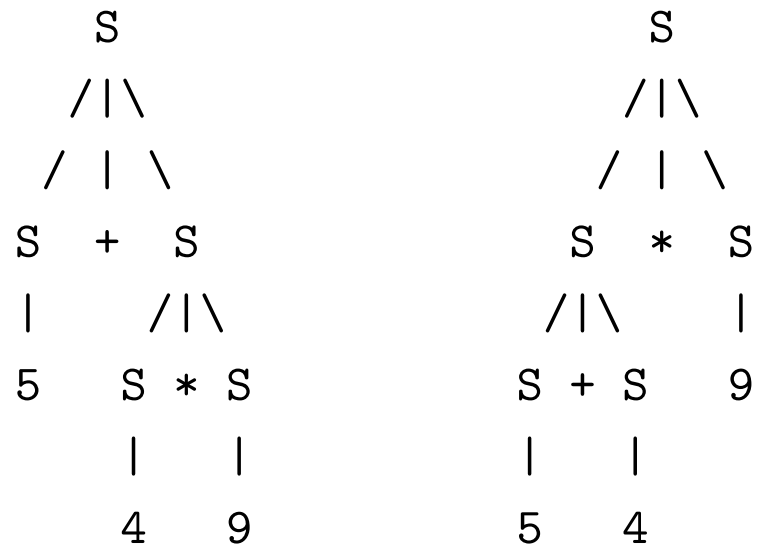
Grammatiche ambigue

Una grammatica G si dice *ambigua* se esiste una stringa che ammette due o più alberi sintattici distinti

Esempio: grammatica G per espressioni aritmetiche:

$$S \rightarrow S + S \mid S - S \mid S * S \mid S \,/\, S \mid (\, 0 \mid 1 \mid \ldots \mid 9 \,)^{+}$$

la stringa $s = 5 + 4 * 9$ ammette due alberi sintattici distinti:



Grammatiche ambigue e analisi sintattica

Una grammatica ambigua NON è adatta all'analisi sintattica (e alla traduzione automatica):

- la struttura sintattica di una frase non è unica
- l'analizzatore sintattico dovrebbe restituire tutti gli alberi sintattici corrispondenti alla stringa (oppure uno a caso)
- ambiguità sintattica $\Rightarrow$ ambiguità semantica
- vedi esempio precedente (espressioni aritmetiche)
- con la precedenza tra operatori indotta dagli alberi sintattici, segue che il significato (valore) della stringa in input è 41 se si considera il primo albero sintattico e 81 se si considera il secondo albero sintattico
- pertanto, in presenza di grammatiche ambigue non è possibile eseguire l'analisi sintattica (e la traduzione) in modo deterministico: di conseguenza, non vengono usate nelle applicazioni informatiche

Non determinismo

Esempio di non determinismo nell'analisi top-down: grammatica G con regole di produzione $\{S \rightarrow aCb \mid aaadbbb, \quad C \rightarrow cC \mid c\}$

- stringa $aaacbbb$
- 1 simbolo di lookahead (simbolo di lookahead iniziale = a)
- l'analizzatore non è in grado di decidere come espandere la radice dell'albero sintattico: quale regola utilizzare tra $S \rightarrow aaadbbb$ (regola R1) e $S \rightarrow aCb$ (regola R2)?
- dovrebbe procedere scegliendo una delle due possibilità, ad esempio la regola R1: se proseguendo nella costruzione però non riuscisse a generare l'albero sintattico, dovrebbe tornare indietro al punto iniziale e tentare di generare l'albero sintattico utilizzando la regola R2
- questo corrisponde ad un algoritmo di backtracking, che è inerentemente non efficiente (tempi di esecuzione esponenziali)

Non determinismo

- il nondeterminismo dell'esempio precedente potrebbe essere risolto da un lookahead di lunghezza 4 (dopo aver letto il quarto carattere della stringa di input, so qual è la regola di produzione da applicare)
- tuttavia, allungare il lookahead in generale non risolve il problema del nondeterminismo
- Esempio: data la seguente G:

$$S \rightarrow B \mid C$$
$$B \rightarrow aB \mid b$$
$$C \rightarrow aC \mid c$$

per qualunque lunghezza del lookahead, esiste una stringa appartenente a $\mathcal{L}(G)$ tale che l'analisi top-down è nondeterministica

(dato un qualunque lookahead n, per la stringa $(a)^{n+1}b$ non posso decidere qual è la regola da applicare per espandere l'assioma)

Grammatiche adatte all'analisi sintattica

Una grammatica è adatta all'analisi sintattica se per tale grammatica è possibile costruire un algoritmo (analizzatore sintattico) deterministico

Pertanto:

1. G non può essere ambigua, altrimenti il problema è inerentemente non deterministico: esistono più alberi sintattici (quindi più soluzioni) per la stessa sequenza di input
2. la forma delle regole di G deve essere tale da evitare il non determinismo nella derivazione (analisi top-down) o nella riduzione (analisi bottom-up)

Lezione 4

Analisi sintattica top-down (e cenni su analisi bottom-up)

Analisi top-down

- Generazione dell'albero sintattico dall'alto verso il basso (dalla radice alle foglie)
- ordine di espansione dei nodi da sinistra verso destra, corrispondente alla *derivazione canonica sinistra* della stringa di input
- derivazione canonica sinistra = derivazione in cui ad ogni passo si espande il simbolo non terminale più a sinistra della forma di frase corrente
- esempio: grammatica G con regole di produzione
  $\{\ S \to AB \mid cSc, \quad A \to a, \quad B \to bB \mid b\ \}$
- derivazione canonica sinistra della stringa *cabbc*:
  $S \Rightarrow cSc \Rightarrow cABc \Rightarrow caBc \Rightarrow cabBc \Rightarrow cabbc$
  (al terzo passo di derivazione si espande il simbolo non terminale A invece del non terminale B)

Metodi per l'analisi top-down

Non determinismo (esempio precedente):

- grammatica G con regole di produzione
  $\{S \rightarrow aCb \mid aaadbbb, \quad C \rightarrow cC \mid c\}$
- stringa $aaacbbb$
- 1 simbolo di lookahead (simbolo di lookahead iniziale = a)

Problema: come scegliere la regola da applicare per espandere un nodo non terminale?

Algoritmo naive: provare tutte le scelte (backtracking)

- in generale non termina
- anche se termina, ha tempi esponenziali

$\Rightarrow$ occorre trovare algoritmi migliori

Metodi per l'analisi top-down

Soluzione: si impongono proprietà alla grammatica al fine di permettere la costruzione di analizzatori sintattici "efficienti"

Per "efficiente" si intende:

1. tempo di esecuzione lineare nelle dimensioni dell'input $\Rightarrow$ analizzatore *deterministico*, che cioè è in grado di decidere in ogni momento quale è la regola da applicare per espandere un simbolo non terminale
2. utilizzo di un solo simbolo di *lookahead* = in ogni momento l'analizzatore conosce solo il prossimo simbolo della stringa di input, pertanto la stringa di input viene letta un simbolo alla volta (da sinistra verso destra)

Grammatiche LL(1)

- grammatica LL(k) = grammatica che ammette un riconoscitore deterministico che è in grado di costruire la derivazione canonica sinistra della stringa di input usando k simboli di lookahead e leggendo la stringa da sinistra verso destra

- grammatica LL(1) = grammatica che ammette un riconoscitore deterministico che è in grado di costruire la derivazione canonica sinistra della stringa di input usando 1 simbolo di lookahead e leggendo la stringa da sinistra verso destra

- siamo interessati a grammatiche LL(1)

- problema: come capire se una data grammatica e' LL(1)?

Trasformazioni delle grammatiche per l'analisi top-down

condizioni *sufficienti* perché una grammatica non sia LL(1):

- presenza di ricorsione sinistra
- presenza di prefissi comuni

ricorsione sinistra = esiste un $X \in V_N$ tale che esiste una derivazione $X \Rightarrow^* X\alpha$ (con $\alpha \in V^*$)

presenza di prefissi comuni = esistono due regole distinte $A \to a\alpha$, $A \to a\beta$ (con $a \in V_T$, $\alpha, \beta \in V^*$)

entrambe queste caratteristiche possono essere eliminate, attraverso *trasformazioni equivalenti* di G, cioè trasformazioni delle regole di G che lasciano invariato il linguaggio L_G

NOTA BENE: in generale queste trasformazioni NON sono sufficienti a rendere la grammatica LL(1)

Eliminazione della ricorsione sinistra diretta

Ricorsione sinistra *diretta* = esiste una regola che ha la forma $X \to X\alpha$

Trasformazione:

le seguenti regole che presentano il simbolo non terminale A a sinistra:

$$\begin{aligned} A &\to A\alpha_1 | \ldots | A\alpha_n \\ A &\to \beta_1 | \ldots | \beta_m \end{aligned}$$

vengono sostituite dalle seguenti regole:

$$\begin{aligned} A &\to \beta_1 A' | \ldots | \beta_m A' \\ A' &\to \epsilon | \alpha_1 A' | \ldots | \alpha_n A' \end{aligned}$$

(dove A' è un nuovo simbolo non terminale)

È inoltre possibile eliminare (con tecniche che estendono quella appena vista) la ricorsione sinistra indiretta

Esempio

Data la grammatica G con le regole di produzione:

$\{\ S \to AB,\ \ A \to Aa \mid a \mid BS,\ \ B \to Bb \mid\ \ Bc \mid \epsilon\ \}$

la trasformazione della ricorsione sinistra diretta produce le seguenti regole di produzione:

$\{\ S \to AB,\ \ A \to aA' \mid BSA',\ \ A' \to \epsilon \mid aA',\ \ B \to B',\ \ B' \to \epsilon \mid bB' \mid cB'\ \}$

Eliminazione dei prefissi comuni

Due regole che presentano prefisso comune:

$$A \to \alpha\beta_1$$

$$A \to \alpha\beta_2$$

vengono sostituite dalle seguenti regole:

$$A \to \alpha A'$$

$$A' \to \beta_1$$

$$A' \to \beta_2$$

(dove A' è un nuovo simbolo non terminale)

Applicando iterativamente tale trasformazione, è possibile eliminare tutti i prefissi comuni da G

Esempio

Data la grammatica G con le regole di produzione:

$S \to ABA \mid ABB,$
$A \to abA \mid aC \mid BS,$
$B \to bB \mid bccS \mid \epsilon$

l'eliminazione dei prefissi comuni produce le seguenti regole di produzione:

$S \to ABS', \quad S' \to A \mid B,$
$A \to aA' \mid BS, \quad A' \to bA \mid C,$
$B \to bB' \mid \epsilon, \quad B' \to B \mid ccS$

Esempio: grammatica per espressioni aritmetiche

grammatica iniziale:

$E \rightarrow E + E \mid E * E \mid (\ E\ ) \mid n$

1) trasformazione per eliminare l'ambiguità (si assegna precedenza all'operatore prodotto rispetto alla somma):

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (\ E\ ) \mid n$

2) trasformazione per eliminare la ricorsione sinistra diretta:

$E \rightarrow TE', \quad E' \rightarrow +TE' \mid \epsilon,$
$T \rightarrow FT', \quad T' \rightarrow *FT' \mid \epsilon,$
$F \rightarrow (\ E\ ) \mid n$

Tabella di parsing LL(1)

L'analisi top-down per grammatiche LL(1) è basata sulla costruzione di una tabella, detta *tabella di parsing LL(1)*

la tabella di parsing LL(1) è un array bidimensionale:

- una riga per ogni simbolo non terminale
- una colonna per ogni simbolo terminale
- ogni cella della tabella (elemento dell'array) contiene una o più regole di produzione della grammatica
- la presenza della produzione $A \rightarrow \alpha$ nella cella in posizione (A, a) significa che la regola $A \rightarrow \alpha$ può essere usata per espandere il simbolo non terminale A quando il simbolo di lookahead è a

Costruzione della tabella LL(1)

per poter costruire la tabella LL(1) di una grammatica G, si utilizzano gli insiemi FIRST e gli insiemi FOLLOW della grammatica G:

- data una forma di frase α, si indica con $FIRST(\alpha)$ l'insieme dei simboli terminali che compaiono come primo carattere di una frase derivata da α
- dato un simbolo non terminale A, si indica con $FOLLOW(A)$ l'insieme dei simboli terminali a tali che a segue immediatamente A in una forma di frase della grammatica, cioè l'insieme dei simboli terminali a tali che esiste una derivazione $S \Rightarrow^* \alpha A a \beta$

Costruzione della tabella LL(1)

per calcolare la tabella LL(1) di G:

1. si calcolano dapprima gli insiemi $FIRST(\alpha)$ per tutti i lati destri α delle regole di produzione di G
2. poi si calcolano gli insiemi $FOLLOW(A)$ per tutti i simboli non terminali A di G
3. poi si costruisce la tabella nel seguente modo: per ogni simbolo non terminale A e per ogni simbolo terminale a, si aggiunge la regola di produzione $A \rightarrow \alpha$ alla cella in posizione (A, a) se $a \in FIRST(\alpha)$ oppure se $\epsilon \in FIRST(\alpha)$ e $a \in FOLLOW(A)$
   ($\epsilon \in FIRST(\alpha)$ se $\alpha \Rightarrow^* \epsilon$)

Tabella LL(1): esempio

Data la seguente grammatica G per espressioni aritmetiche
$\{\ E \to TE',\ \ E' \to +TE' \mid \epsilon,\ \ T \to FT'\ \ T' \to *FT' \mid \epsilon\ \ F \to (\ E\ ) \mid n\ \}$

la tabella LL(1) per G è la seguente:

	n	$+$	$*$	$($	$)$	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to n$			$F \to (\ E\ )$		

Caratterizzazione delle grammatiche LL(1)

- *conflitto* in una tabella LL(1) = esistenza di una cella della tabella che contiene due o più regole di produzione della grammatica
- PROPRIETÀ: una grammatica G è LL(1) se e solo se la tabella LL(1) per G non presenta conflitti
- nell'esempio precedente, la tabella non presenta conflitti
- pertanto, la grammatica per espressioni aritmetiche dell'esempio precedente è una grammatica LL(1)

Algoritmo per l'analisi LL(1)

è possibile definire un unico algoritmo, detto *analizzatore LL(1)*, che è in grado di eseguire l'analisi sintattica per *tutte* le grammatiche LL(1)

l'analizzatore LL(1) riceve in input:

1. la grammatica G sotto forma di tabella LL(1) di G
2. la stringa w

NOTA BENE: la tabella LL(1) di G non deve presentare conflitti (cioè G deve essere LL(1))

Analizzatore LL(1)

```
INPUT: stringa w (terminata dal simbolo \$), tabella M (tabella LL(1) di G)
OUTPUT: se w appartiene a L(G) stampa la sequenza di regole che genera
        la derivazione canonica sinistra di w, altrimenti restituisce ERRORE
begin
  lookahead = primo simbolo di w; initialize(stack); push(stack,'\$'); push(stack,S);
  repeat
    x = top(stack); lookahead = prossimo simbolo di w;
    if (x e' un terminale) OR (x == '\$')
    then if (x == lookahead)
         then begin pop(stack,x); lookahead = prossimo simbolo di w end
         else ERROR
    else if (M[x,lookahead] == X -> Y1 Y2 ... Yk)
         then begin pop(stack,x); push(stack,Yk); ...push(stack,Y2); push(stack,Y1);
                    print('X -> Y1 Y2 ... Yk')
              end
         else ERROR
  until (x == '\$')
end
```

Analisi bottom-up

- analisi sintattica bottom-up = l'albero sintattico della stringa di input viene generato a partire dalle foglie (simboli terminali della stringa di input), risalendo fino alla radice (assioma)

- in particolare, l'albero sintattico viene generato effettuando la *riduzione canonica destra* della stringa di input

Riduzione

Riduzione = inverso della derivazione

data una grammatica $G = \langle V_T, V_N, S, P \rangle$, diciamo che:

- $\beta\alpha\gamma$ *si riduce direttamente* a $\beta A\gamma$ in G (denotato da $\beta\alpha\gamma \Rightarrow_r \beta A\gamma$) se esiste una regola $A \rightarrow \alpha$ in P
- β *si riduce* a α in G (denotato da $\beta \Rightarrow_r^* \alpha$) se esistono $\gamma_1 \in (V_T \cup V_N)^* \ldots, \gamma_k \in (V_T \cup V_N)^*$ con $k \geq 0$ tali che

$$\beta \Rightarrow_r \gamma_1 \Rightarrow_r \ldots \Rightarrow_r \gamma_k \Rightarrow_r \alpha$$

- riduzione canonica destra = inverso della derivazione canonica destra

la derivazione usa le regole di produzione della grammatica come regole di riscrittura, ma da destra verso sinistra (la derivazione le usa da sinistra verso destra)

Riduzione canonica destra: esempio

grammatica G con regole di produzione
$\{\ S \to AB \mid cSc, \quad A \to a, \quad B \to bB \mid b\ \}$

derivazione canonica destra della stringa *cabbc*:
$S \Rightarrow cSc \Rightarrow cABc \Rightarrow cAbBc \Rightarrow cAbbc \Rightarrow cabbc$
(al terzo passo di derivazione si espande il simbolo non terminale B invece del non terminale A)

riduzione canonica destra della stringa *cabbc*:
$cabbc \Rightarrow_r cAbbc \Rightarrow_r cAbBc \Rightarrow_r cABc \Rightarrow_r cSc \Rightarrow_r S$

Analisi LR(1)

- grammatica LR(1) = grammatica che ammette un riconoscitore deterministico che è in grado di costruire la derivazione canonica destra della stringa di input usando 1 simbolo di lookahead e leggendo la stringa da sinistra verso destra
- tabella LR(1) di una grammatica G (analoga alla tabella LR(1))
- PROPRIETÀ: una grammatica G è LR(1) se e solo se la tabella LR(1) di G non presenta conflitti
- analizzatore LR(1) = algoritmo che riceve in input la tabella LR(1) di una grammatica G e una stringa s di input, ed è in grado di decidere se $s \in \mathcal{L}(G)$

Analisi top-down e analisi bottom-up

confronto tra analisi LR(1) e analisi LL(1):

- l'analisi bottom-up è in generale più potente dell'analisi top-down, nel senso che la classe delle grammatiche LR(1) è più ampia della classe delle grammatiche LL(1)
- questo è dovuto al fatto che la strategia bottom-up sfrutta in modo maggiore l'informazione costituita dalla stringa di input (simboli di lookahead), perché la riduzione parte dai simboli terminali (foglie dell'albero sintattico) invece che dall'assioma della grammatica

Lezione 5

Traduzione guidata dalla sintassi

Sintesi dell'output

approccio classico (due passate):

1. si sintetizza l'albero sintattico (struttura dati) nella fase di analisi sintattica
2. si utilizza l'albero sintattico per la sintesi dell'output

approccio semplificato (una passata):

1. si sintetizza l'output direttamente durante l'analisi sintattica, senza costruire l'albero sintattico

l'approccio semplificato ad una passata funziona solo nei casi "semplici" di traduzione e, più in generale, di elaborazione dell'input

invece, costruendo l'albero sintattico e visitandolo successivamente si può implementare un insieme molto più grande di funzioni di traduzione e di elaborazione

Traduzione senza albero sintattico

- traduzione senza albero sintattico = analisi sintattica dell'input e sintesi dell'output in una sola passata
- pertanto, la traduzione deve avvenire *durante* l'analisi sintattica
- occorre modificare l'analizzatore sintattico, aggiungendo istruzioni che permettano la sintesi dell'output in parallelo all'analisi dell'input
- NOTA BENE: l'analizzatore sintattico LL(1) visto in precedenza NON genera l'albero sintattico

Traduzione guidata dalla sintassi

- si parla di *traduzione guidata dalla sintassi* nei casi in cui è possibile definire il processo di sintesi dell'output (traduzione) sulla base della struttura sintattica dell'input
- in altre parole, in questi casi l'operazione di traduzione è fortemente legata alla sintassi dell'input, pertanto la traduzione può essere definita in modo "parallelo" alla definizione della sintassi dell'input
- in moltissime applicazioni informatiche la funzione di traduzione (o di elaborazione) è guidata dalla sintassi
- nel seguito trattiamo solo funzioni di traduzione guidate dalla sintassi
- anche per traduzioni guidate dalla sintassi, si può avere la necessità di costruire l'albero sintattico (due passate distinte per analisi sintattica e sintesi dell'output) oppure no (una sola passata per analisi sintattica e sintesi dell'output)

Grammatiche ad attributi

per la traduzione guidata dalla sintassi viene definita una estensione del formalismo delle grammatiche non contestuali: le *grammatiche ad attributi*

le grammatiche ad attributi estendono le grammatiche non contestuali introducendo le nozioni di:

1. *attributi* associati ai simboli (terminali e non terminali) della grammatica
2. *azioni semantiche* e *regole semantiche*, che affiancano le regole sintattiche della grammatica

Attributi, regole semantiche, azioni semantiche

nelle grammatiche ad attributi:

- ogni simbolo terminale o non terminale può avere uno o più attributi
- *attributo* = proprietà associata al simbolo, che può essere letta o assegnata dalle azioni semantiche (= variabile di un linguaggio di programmazione)
- ad ogni regola di produzione (regola sintattica) può essere associata una regola semantica
- *regola semantica* = sequenza di azioni semantiche
- *azione semantica* = azione in grado di assegnare valori agli attributi e di avere altri effetti (= istruzione di un linguaggio di programmazione)

Attributi e traduzione

- gli attributi sono usati per rappresentare un "significato" che viene associato ai simboli sintattici
- significato = valore (numero, stringa, struttura dati, ecc.) che assume l'attributo
- le azioni semantiche sono il meccanismo che effettua il calcolo di tali valori, e che quindi calcola tale "significato"
- possiamo effettuare la traduzione utilizzando attributi e azioni semantiche, se assegnamo come "significato" di un simbolo la traduzione della parte di stringa di input che tale simbolo rappresenta nell'albero sintattico

Attributi e traduzione

esempio:

- in una grammatica per espressioni aritmetiche sui numeri interi, possiamo associare un attributo di tipo intero ad ogni simbolo non terminale della grammatica
- possiamo poi definire delle azioni semantiche in modo tale che il valore di ogni attributo corrisponda al valore della sottoespressione associata al corrispondente simbolo non terminale
- pertanto: valore dell'attributo associato alla radice dell'albero sintattico = valore dell'intera espressione aritmetica

Attributi sintetizzati e attributi ereditati

gli attributi si distinguono in:

- *attributi sintetizzati*: sono gli attributi il cui valore dipende solo dai valori degli attributi presenti nel sottoalbero del nodo dell'albero sintattico a cui sono associati
- *attributi ereditati*: sono gli attributi il cui valore dipende solo dai valori degli attributi presenti nei nodi predecessori e nei nodi fratelli del nodo dell'albero sintattico a cui sono associati

gli attributi sintetizzati realizzano un flusso informativo *ascendente* nell'albero sintattico (dalle foglie verso la radice)

gli attributi ereditati realizzano un flusso informativo *discendente* (dalla radice verso le foglie) e *laterale* (da sinistra verso destra e viceversa) nell'albero sintattico

Valutazione degli attributi

- l'esecuzione di una regola semantica avviene in parallelo all'applicazione della corrispondente regola sintattica nell'analisi sintattica
- pertanto, l'ordine di applicazione delle regole semantiche è conseguenza dell'ordine di applicazione delle regole di produzione nell'analisi sintattica
- in particolare, dato un certo ordine di applicazione, le istruzioni che nelle regole semantiche effettuano la valutazione degli attributi potrebbero non essere eseguibili in modo corretto
- ad esempio, può accadere che in una azione semantica si tenti di leggere il valore di un attributo non ancora assegnato
- questo problema, noto come problema della valutazione degli attributi, è un problema fondamentale per il trattamento automatico delle grammatiche ad attributi

S-attributi e L-attributi

- per risolvere il problema della valutazione degli attributi, occorre restringere i tipi di attributi che possono essere definiti

- in particolare, si definiscono due classi di attributi che garantiscono la corretta valutazione degli attributi:

  1. *S-attributi*: coincide con la classe degli attributi sintetizzati

  2. *L-attributi*: è una sottoclasse degli attributi ereditati, in cui si permette ad ogni attributo A di dipendere solo dal nodo padre o dai nodi fratelli *a sinistra* del nodo dell'albero sintattico a cui l'attributo A è associato

Analisi top-down con grammatiche ad attributi

S-attributi e L-attributi sono adatti all'analisi top-down:

- sia G' una grammatica ad attributi ottenuta aggiungendo ad una grammatica LL(1) G attributi ad azioni semantiche tali che ogni attributo di G' è o un S-attributo o un L-attributo
- allora, è possibile definire un analizzatore top-down LL(1) per G' che è in grado di effettuare la valutazione degli attributi in parallelo all'analisi sintattica

(una proprietà analoga vale per l'analisi bottom-up)

Lezione 6

Strumenti per la generazione automatica di analizzatori lessicali e sintattici

Strumenti per la generazione automatica di parser

partendo da una specifica del linguaggio formale, generano il riconoscitore per quel linguaggio

- generatori di analizzatori lessicali (linguaggi regolari): specifica in termini di espressioni regolari
- generatori di analizzatori sintattici (linguaggi non contestuali): specifica in termini di grammatiche (trattano anche grammatiche ad attributi)

diversi tool per diversi linguaggi di programmazione

Strumenti per la generazione automatica di parser

- ambiente Java:
  - JavaCC (generatore di parser top-down)
  - JLex,JFlex (generatori di analizzatori lessicali)
  - CUP, ANTLR (generatori di parser bottom-up)
- ambiente C/C++:
  - LEX, FLEX (generatori di analizzatori lessicali)
  - YACC, BISON, ANTLR (generatore di parser bottom-up)

JavaCC

- cosa fa: partendo da una specifica di una grammatica (con azioni semantiche), genera le classi Java che realizzano un analizzatore sintattico top-down per tale grammatica

- una specifica JavaCC (file con estensione .jj) è essenzialmente una grammatica EBNF

- si possono aggiungere a tale grammatica azioni semantiche (= istruzioni in Java)

Struttura di una specifica JavaCC

specifica JavaCC = file testuale (estensione .jj)

è composta da 3 sezioni:

1. lista di opzioni per JavaCC
2. unità di compilazione Java che inizia con `PARSER_BEGIN(nome_parser)` e termina con `PARSER_END(nome_parser)`
3. lista di regole di produzione

Specifica JavaCC: esempio

esempio: specifica JavaCC che implementa un *riconoscitore* per la seguente grammatica per espressioni aritmetiche:

$$
\begin{aligned}
&S \rightarrow E\ \mathit{eof}\\
&E \rightarrow T\ E_1\\
&E_1 \rightarrow +\ T\ E_1 \mid \epsilon\\
&T \rightarrow F\ T_1\\
&T_1 \rightarrow *\ F\ T_1 \mid \epsilon\\
&F \rightarrow (\ E\ ) \mid \mathit{num}
\end{aligned}
$$

(dove *num* denota le costanti intere senza segno e *eof* denota il simbolo di end-of-file)

Specifica JavaCC 1: dichiarazione classe parser

```
PARSER_BEGIN(expr1)

public class expr1 {

  public static void main(String args[]) throws ParseException {
    expr1 parser = new expr1(System.in);
    parser.start();
  }
}

PARSER_END(expr1)
```

Specifica JavaCC 2: regole lessicali

```
SKIP :
{
  " " | "\t" | "\n" | "\r"
}

TOKEN :
{
   <NUM: (["0"-"9"])+>
 | <PIU: "+"> | <MENO: "-">
 | <PER: "*"> | <DIV: "/">
 | <PARAP: "("> | <PARCH: ")">
}
```

(N.B.: `<EOF>` è un token predefinito in JavaCC)

Specifica JavaCC 3: regole sintattiche

```
void start() : {}
{ espr() <EOF> }

void espr() : {}
{ term() espr1() }

void espr1() : {}
{  <PIU> term() espr1() | <MENO> term() espr1() | {} }

void term() : {}
{ factor() term1() }

void term1() : {}
{  <PER> factor() term1() | <DIV> factor() term1() | {} }

void factor() : {}
{ <PARAP> espr() <PARCH> | <NUM> }
```

Simboli teminali e non terminali in JavaCC

- l'assioma della grammatica è dichiarato nella dichiarazione di classe del parser (nell'esempio precedente, `parser.start()` dichiara che il simbolo `start` è l'assioma della grammatica)
- viene definita una regola di produzione per ogni simbolo non terminale
- simboli non terminali = metodi (funzioni)
- simboli terminali = dichiarati nelle regole lessicali (nell'esempio precedente, le regole precedute da `SKIP:` e `TOKEN:`)
- per ogni token riconosciuto, viene creato un oggetto della classe `Token` predefinita

Simboli non terminali e analisi ricorsiva in JavaCC

- simboli non terminali = metodi (funzioni), che in generale possono avere argomenti e restituire valori
- la regola di produzione per un simbolo non terminale è codificata in termini di un metodo che a sua volta invoca i metodi corrispondenti alla parte destra della regola
- esempio:
  `void espr() :  {} { term() espr1() }`
  - l'espansione del simbolo `espr` nell'analisi top-down corrisponde alla chiamata del metodo `espr()` corrispondente
  - a sua volta, tale metodo invoca nell'ordine i metodi `term()` e `espr1()`
- $\Rightarrow$ l'analizzatore top-down implementato da JavaCC è un programma ricorsivo

Specifica JavaCC 4: regole sintattiche in forma compatta

```
void start() : {}
{ espr() <EOF> }

void espr() : {}
{ term() espr1() }

void espr1() : {}
{ [ (<PIU>|<MENO>) term() espr1() ] }

void term() : {}
{ factor() term1() }

void term1() : {}
{ [ (<PER>|<DIV>) factor() term1() ] }

void factor() : {}
{ <PARAP> espr() <PARCH> | <NUM> }
```

Uso delle azioni semantiche

- vogliamo estendere il riconoscitore generato dalla precedente specifica
- lo scopo è quello di stampare in uscita la sequenza di regole di produzione che genera la derivazione canonica sinistra della stringa di input
- aggiungiamo pertanto alla precedente specifica delle semplici azioni semantiche
- in particolare, *all'inizio* di ogni regola di produzione aggiungiamo una istruzione di stampa

Specifica JavaCC 5: uso delle azioni semantiche

```
void start() : {}
{ {System.out.println("S -> E eof");} term() espr1() }

void espr() : {}
{ {System.out.println("E -> T E1");} term() espr1() }

void espr1() : {}
{  {System.out.println("E1 -> '+' T E1");} <PIU> term() espr1()
 | {System.out.println("E1 -> '-' T E1");} <MENO> term() espr1()
 | {System.out.println("E1 -> epsilon");} }

void term() : {}
{ { System.out.println("T -> F T1"); } factor() term1() }

void term1() : {}
{  {System.out.println("T1 -> '*' F T1");} <PER> factor() term1()
 | {System.out.println("T1 -> '/' F T1");} <DIV> factor() term1()
 | {System.out.println("T1 -> epsilon");} }

void factor() : {}
{  {System.out.println("F -> '(' E ')'");} <PARAP> espr() <PARCH>
 | {System.out.println("F -> num");} <NUM> }
```

Uso degli attributi

- vogliamo ora realizzare con JavaCC un programma che effettua il calcolo del valore dell'espressione aritmetica
- a tale scopo, utilizziamo attributi che associamo ai simboli non terminali e terminali della grammatica
- estensione della grammatica per espressioni aritmetiche con attributi e azioni semantiche per il calcolo del valore dell'espressione
- gli attributi sono realizzati:
  1. definendo *argomenti* nelle funzioni che implementano i simboli non terminali;
  2. facendo *restituire un valore* a tali funzioni

Specifica JavaCC 6: uso degli attributi

```
void start() : {int ris;}
{ ris=espr() <EOF> {System.out.println(ris);} }

int espr() : {int t1,ris;}
{ t1=term() ris=espr1(t1) {return ris;} }

int espr1(int t1) : {int t2,t3,ris;}
{  <PIU> t2=term() {t3=t1+t2;} ris=espr1(t3) {return ris;}
 | <MENO> t2=term() {t3=t1-t2;} ris=espr1(t3) {return ris;}
 | {return t1;}
}
```

Specifica JavaCC 6: uso degli attributi (segue)

```
int term() : {int t1,ris;}
{ t1=factor() ris=term1(t1) {return ris;} }

int term1(int t1) : {int t2,t3,ris;}
{  <PER> t2=factor() {t3=t1*t2;} ris=term1(t3) {return ris;}
 | <DIV> t2=factor() {t3=t1/t2;} ris=term1(t3) {return ris;}
 | {return t1;}
}

int factor() : {int ris; Token n;}
{  <PARAP> ris=espr() <PARCH> {return ris;}
 | n=<NUM> {ris=Integer.parseInt(n.image); return ris;}
}
```

Attributi e simboli terminali

nell'esempio precedente, viene associato un attributo di tipo intero al tipo di token `<NUM>`

```
int factor() : {int ris; Token n;}
{  <PARAP> ris=espr() <PARCH> {return ris;}
 | n=<NUM> {ris=Integer.parseInt(n.image); return ris;}
}
```

- `n.image` contiene il valore (stringa) del token di tipo `<NUM>`
- `Integer.parseInt` converte la stringa in intero
- nella variabile `ris` viene memorizzato il valore intero corrispondente al valore del token `<NUM>`
- tale valore viene restituito in uscita dal metodo `factor`

Link utili su JavaCC

– JavaCC home page: `https://javacc.dev.java.net/`
(per scaricare JavaCC e la documentazione)

– tutorial su JavaCC (versione preliminare, in inglese):
`http://www.engr.mun.ca/~theo/JavaCC-Tutorial`