Linguaggi e Traduttori: Analisi sintattica

Armando Tacchella

Sistemi e Tecnologie per il Ragionamento Automatico (STAR-Lab)
Dipartimento di Informatica Sistemistica e Telematica (DIST)
Università di Genova

A.A. 2006/2007 - Primo semestre

Outline

Introduzione e motivazioni

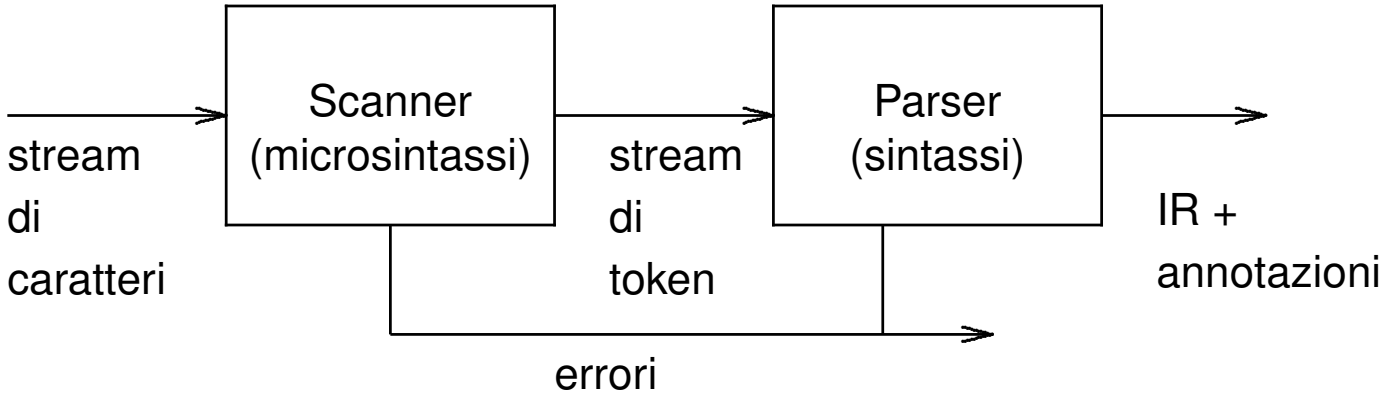
Strumenti formali

- Grammatiche context-free (CFG)
- Non determinismo e CFG

Generazione automatica di analizzatori sintattici

- Parser top-down
- Parser bottom-up

Il Front End



I compiti del parser sono:

- Validare il flusso di token e le relative parti del discorso relativamente alla loro correttezza grammaticale
- Controllare che il programma in ingresso sia ben formato
- Eseguire verifiche guidate dalla sintassi relativamente al contesto semantico
- Costruire una rappresentazione intermedia (IR) del codice sorgente

Analisi sintattica e compilatori

Siamo interessati a scoprire derivazioni per delle frasi

- Necessitiamo di un modello matematico della sintassi - una grammatica G,
- Necessitiamo di un modo per controllare che la frase data appartiene a $L(G)$ - il linguaggio descritto dalla grammatica G
- Abbiamo l'obiettivo di costruire analizzatori sintattici efficienti (possibilmente in maniera automatica)

Percorso

- Strumenti formali: grammatiche context-free (CFG) e derivazioni
- Analizzatori sintattici top-down
- Analizzatori sintattici bottom-up

Outline

Specifica della sintassi

- Gli elementi di sintassi liberi dal contesto semantico sono specificati con una CFG, ad es:

  1. miagolio ::= MIAO | MIAO miagolio

  descrive (in forma astratta) il miagolio di un gatto
- La CFG sopra è specificata con una variante della notazione BNF (Backus-Naur form)
- Formalmente, una grammatica $G = (S, N, T, P)$
  - S è il simbolo iniziale
  - N è l'insieme di simboli non terminali
  - T è l'insieme di simboli terminali (o parole)
  - P è un insieme di produzioni o regole di riscrittura
- Nell'esempio S= miagolio, N={miagolio}, T={MIAO} e P={1}

CFG e relativi vincoli

Una grammatiche G deve rispettare tre condizioni per essere context-free

- $N \cap T = \emptyset$, i simboli terminali sono distinti dai non terminali
- $S \in N$, il simbolo iniziale è un non terminale
- $P \subseteq \{(V, \alpha) | V \in N, \alpha \in (N \cup T)^*\}$, ossia:
  - il simbolo a sinistra di una produzione è un singolo non terminale, mentre
  - a destra della regola vi è una seguenza di simboli terminali e non terminali (e null'altro)

Esempio di CFG

Specifica

1	expr ::= expr op expr
2	| NUM
3	| ID
4	op ::= +
5	| -
6	| *
7	| /

Derivazione di $x - 2 \cdot y$

–	expr
1	expr op expr
3	<ID,x> op expr
5	<ID,x> - expr
1	<ID,x> - expr op expr
2	<ID,x> - <NUM,2> op expr
6	<ID,x> - <NUM,2> * expr
3	<ID,x> - <NUM,2> * <ID,y>

Lo scopo dell'analizzatore sintattico (parser) è quello di verificare la correttezza delle espressioni trovando una derivazione ammissibile

Outline

Derivazioni

L'analisi sintattica comporta

- ad ogni passo, la scelta di un simbolo non terminale da espandere;
- derivazioni diverse a seconda delle scelte effettuate.

Vi sono due modalità interessanti in ambito compilatori

- Derivazione a sinistra (leftmost derivation)
- Derivazione a destra (rightmost derivation)

Si tratta di due modalità sistematiche (altre modalità sono difficilmente automatizzabili)

Derivazioni a sinistra e a destra

Derivazione a destra per $x - 2 \cdot y$

–	expr
1	expr op expr
3	expr op <ID,y>
6	expr * <ID,y>
1	expr op expr * <ID,y>
2	expr op <NUM,2> * <ID,y>
5	expr - <NUM,2> * <ID,y>
3	<ID,x> - <NUM,2> * <ID,y>

Derivazione a sinistra per $x - 2 \cdot y$

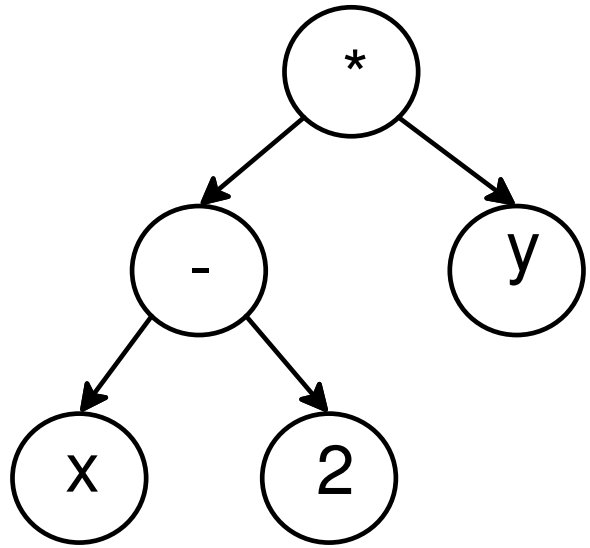
–	expr
1	expr op expr
3	<ID,x> op expr
5	<ID,x> - expr
1	<ID,x> - expr op expr
2	<ID,x> - <NUM,2> op expr
6	<ID,x> - <NUM,2> * expr
3	<ID,x> - <NUM,2> * <ID,y>

- Le due derivazioni producono diversi AST
- Questo implica un diverso ordine di valutazione

Derivazioni e AST (1/2)

Derivazione a dx $\rightarrow ((x - 2) \cdot y)$

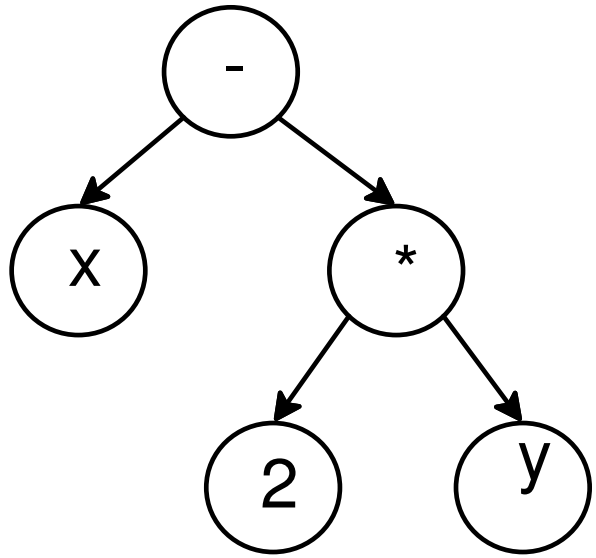
–	expr
1	expr op expr
3	expr op <ID,y>
6	expr * <ID,y>
1	expr op expr * <ID,y>
2	expr op <NUM,2> * <ID,y>
5	expr - <NUM,2> * <ID,y>
3	<ID,x> - <NUM,2> * <ID,y>



Derivazioni e AST (2/2)

Derivazione a sx $\rightarrow (x - (2 \cdot y))$

–	expr
1	expr op expr
3	<ID,x> op expr
5	<ID,x> - expr
1	<ID,x> - expr op expr
2	<ID,x> - <NUM,2> op expr
6	<ID,x> - <NUM,2> * expr
3	<ID,x> - <NUM,2> * <ID,y>



Ordine di valuatazione

- La diversità tra derivazione a sinistra e a destra è una forma di non determinismo
- La realizzazione di un analizzatore sintattico deve avvenire mediante un automa deterministico
- Possibili soluzioni:
  - Forzare l'utilizzo delle parentesi (`SimpleCompiler`)
  - Codificare l'ordine di valutazione
- Una nuova CFG per le espressioni:
  - Introdurre un nuovo simbolo non terminale per ogni livello di precedenza
  - Isolare le parti corrispondenti della grammatica
  - Forzare il parser a riconoscere le espressioni con precedenza maggiore per prime

CFG con precedenze

Specifica

1	goal	::= expr
2	expr	::= expr + term
3		| expr - term
4		| term
5	term	::= term * factor
6		| term / factor
7		| factor
8	factor	::= NUM
9		| ID
10		| (expr)

Questa CFG ha più regole:

- comporta un numero maggiore di riscritture per raggiungere alcuni simboli terminali
- codifica le regole di precedenza usuali
- produce lo stesso AST indipendentemente dal tipo di derivazione

CFG ambigue (1/3)

La CFG originale per descrivere le espressioni è ambigua

1	expr ::= expr op expr
2	| NUM
3	| ID
4	op ::= +
5	| -
6	| *
7	| /

–	expr
1	expr op expr
1	expr op expr op expr
3	<ID,x> op expr op expr
5	<ID,x> - expr op expr
2	<ID,x> - <NUM,2> op expr
6	<ID,x> - <NUM,2> * expr
3	<ID,x> - <NUM,2> * <ID,y>

- Questa CFG consente derivazioni multiple
- Si tratta di un'altra forma di non determinismo
- Difficile da automatizzare se vi sono molteplici scelte

CFG ambigue (2/3)

Diverse derivazioni a sinistra per $x - 2 \cdot y$

–	expr
1	expr op expr
3	<ID,x> op expr
5	<ID,x> - expr
1	<ID,x> - expr op expr
2	<ID,x> - <NUM,2> op expr
6	<ID,x> - <NUM,2> * expr
3	<ID,x> - <NUM,2> * <ID,y>

–	expr
1	expr op expr
1	expr op expr op expr
3	<ID,x> op expr op expr
5	<ID,x> - expr op expr
2	<ID,x> - <NUM,2> op expr
6	<ID,x> - <NUM,2> * expr
3	<ID,x> - <NUM,2> * <ID,y>

CFG ambigue (3/3)

- Definizioni di ambiguità
  - Se una grammatica ha più di una derivazione a sinistra per la stessa frase, allora è ambigua
  - Idem come sopra per le derivazioni a destra
- Nota bene
  - le derivazioni a destra e a sinistra di una frase possono differire anche in una grammatica non ambigua
  - comunque l'AST deve coincidere
- Esempio classico: If-then-else

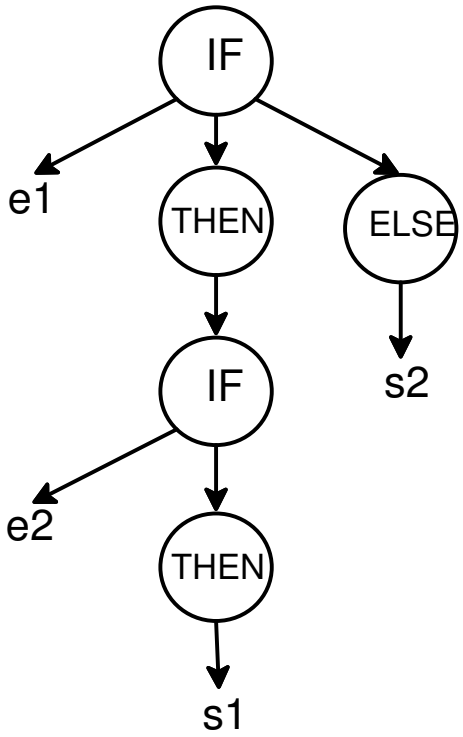
  statement ::= IF expr THEN statement
  | IF expr THEN statement ELSE statement
  | . . .

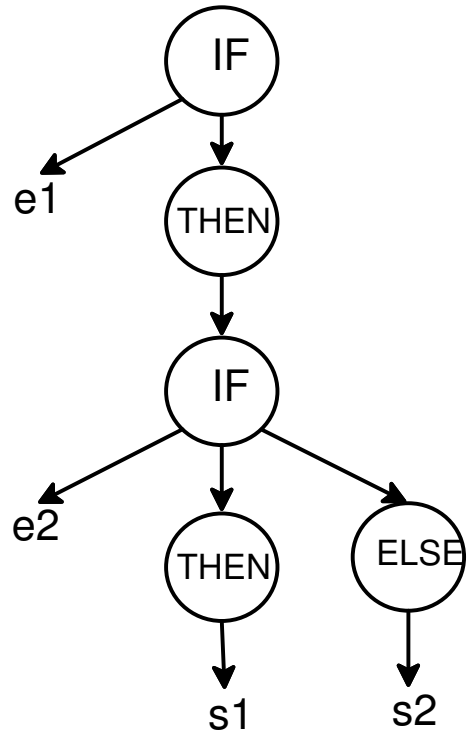
  Si tratta di un'ambiguità risolvibile a livello grammaticale

Altri esempi di CFG ambigue (1/2)

Derivazioni per la frase:

IF expr1 THEN IF expr2 THEN statement1 ELSE statement2





Altri esempi di CFG ambigue (2/2)

- Anche in questo caso, bisogna riscrivere la CFG
- Idea: ogni ELSE viene messo in corrispondenza dell'IF più interno privo di ELSE

```
1 | statement ::= IF expr THEN statement
2 |             | IF expr THEN withElse ELSE statement
3 |             | ...
4 | withElse  ::= IF expr THEN withElse ELSE withElse
5 |             | ...
```

- Questa grammatica ammette una sola derivazione a destra
- Una volta dentro ad un costrutto "withElse", non possiamo generare un ELSE che non sia in corrispondenza con un IF

Ambiguità data dal contesto

- Negli esempi visti in precedenza, l'ambiguità sorgeva dal fatto che le CFG sono un formalismo intrinsecamente non-deterministico
- Esistono ambiguità più profonde, risolvibili solo a livello semantico
- Esempio, istruzione `z = a + b` in C++:
  - Se a,b e z sono numeri, allora l'AST codifica la struttura dell'espressione e la relativa assegnazione
  - Se a,b e z sono oggetti, allora l'AST codifica due chiamate a funzione (`operator+` e il costruttore di copia)

Tecniche di parsing

- Parser top-down
  - Inizia costruendo la radice dell'AST e si muove verso le foglie
  - Sceglie una produzione e cerca di applicarla al programma
  - Errore nella scelta della produzione $\longrightarrow$ backtracking
  - Alcune grammatiche consento di evitare il backtracking grazie all'utilizzo di un parsing predittivo
- Parser bottom-up
  - Inizia costruendo le foglie dell'AST e si muove verso la radice
  - Legge il programma e codifica le alternative in un registro interno
  - Inizia in uno stato legale per i token che possono comparire all'inizio delle frasi
  - Gestiscono (in modo efficiente) un'ampia classe di linguaggi

Outline

Algoritmo per parsing top-down

Costruire il nodo radice dell'AST etichettandolo con il simbolo S
Ripetere i seguenti passi:

1. In un nodo etichettato con A, selezionare una produzione con A sul lato sinistro e costruire per ogni simbolo sul lato destro della produzione un figlio appropriato del nodo corrente
2. Un simbolo terminale che non corrisponde al token letto in ingresso, richiede di tornare al passo precedente
3. Espandere il nodo successivo (i nodi da espandere sono quelli per cui $A \in N$)

Chiaramente, la chiave per un efficace processo di parsing è nella scelta della produzione da effettuare al passo 1

Utilizzo di parser top-down

Specifica

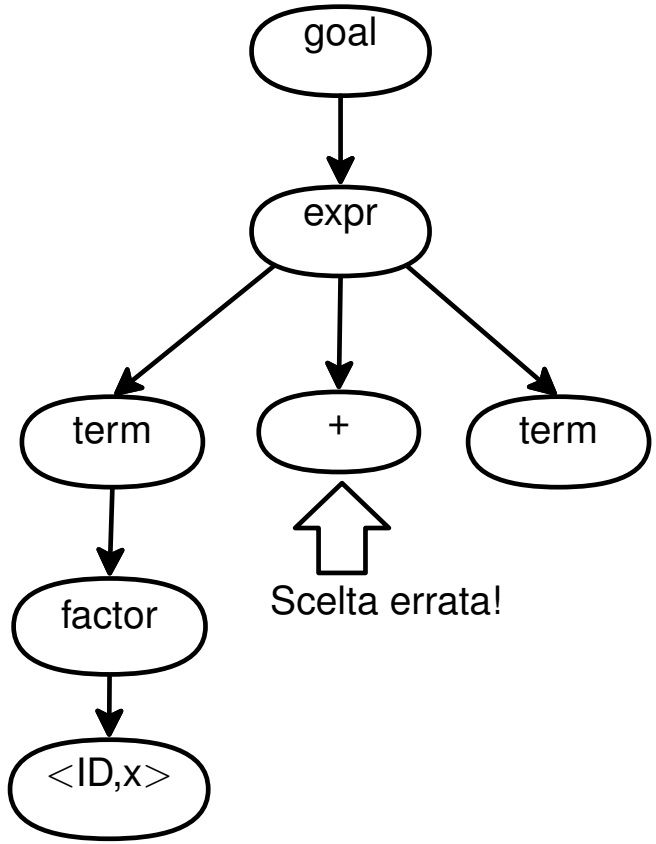
1	goal	::= expr
2	expr	::= expr + term
3		| expr - term
4		| term
5	term	::= term * factor
6		| term / factor
7		| factor
8	factor	::= NUM
9		| ID
10		| (expr)

Consideriamo $x - 2 \cdot y$:

- Come si comporta l'algoritmo visto in precedenza?
- Ottengo parser efficienti?
- Altri problemi?

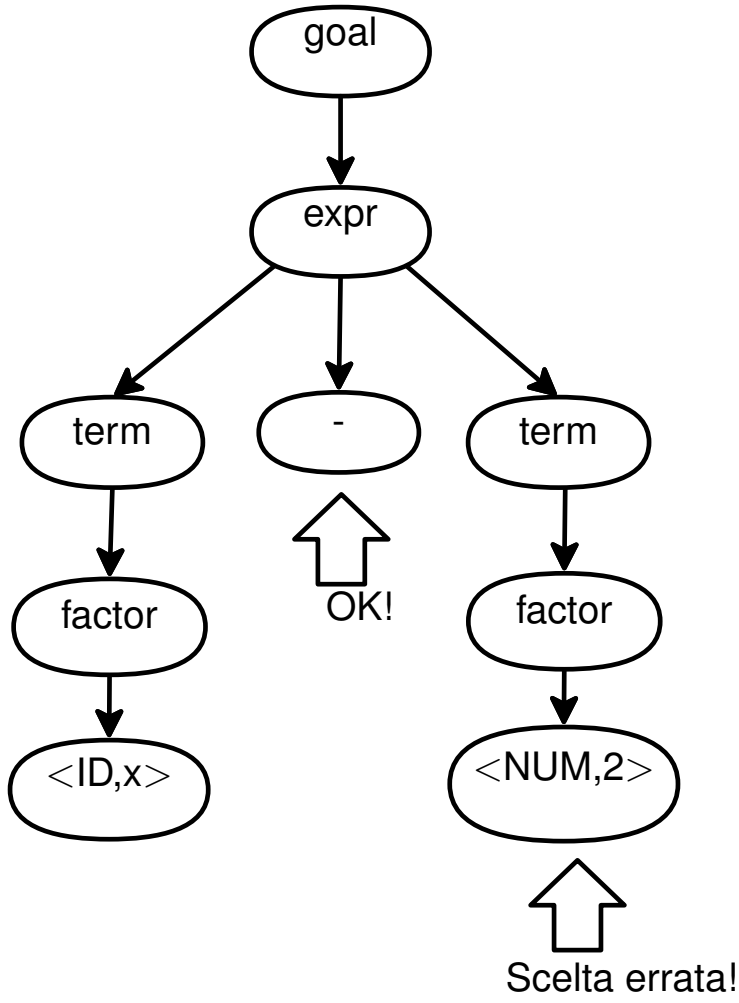
Parsing top-down di $x - 2 \cdot y$ (1/3)

Regola	Derivazione	Input
–	goal	$\underline{x} - 2 \cdot y$
1	expr	$\underline{x} - 2 \cdot y$
2	expr + term	$\underline{x} - 2 \cdot y$
4	term + term	$\underline{x} - 2 \cdot y$
7	factor + term	$\underline{x} - 2 \cdot y$
9	<ID,x> + term	$x\underline{-}2 \cdot y$



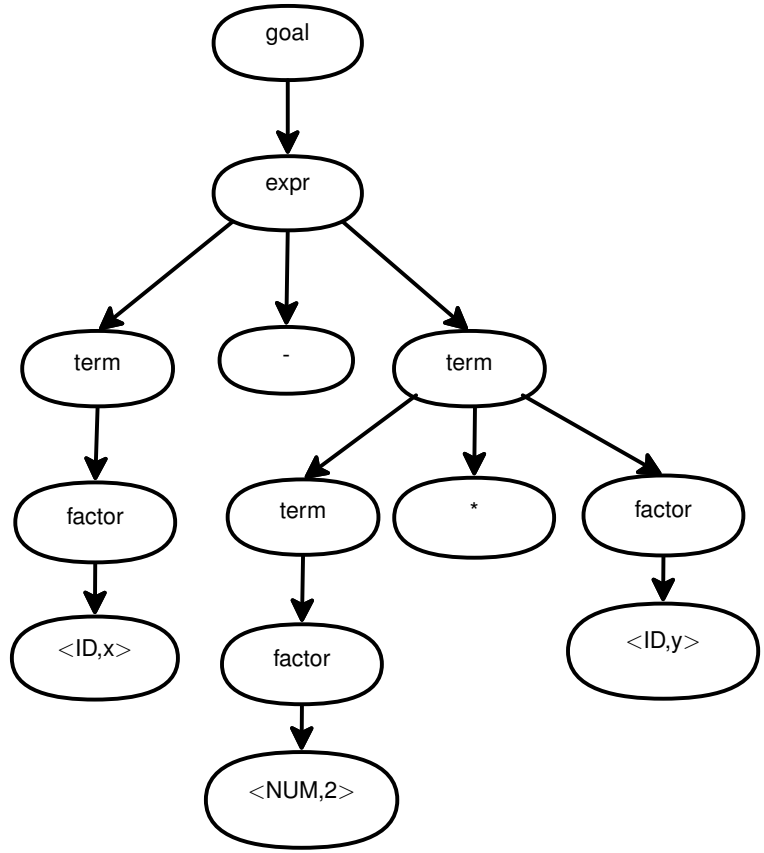
Parsing top-down di $x - 2 \cdot y$ (2/3)

Regola	Derivazione	Input
–	goal	$\underline{x} - 2 \cdot y$
1	expr	$\underline{x} - 2 \cdot y$
3	expr - term	$\underline{x} - 2 \cdot y$
4	term - term	$\underline{x} - 2 \cdot y$
7	factor - term	$\underline{x} - 2 \cdot y$
9	<ID,x> - term	$x \underline{-} 2 \cdot y$
–	<ID,x> - term	$x - \underline{2} \cdot y$
7	<ID,x> - factor	$x - \underline{2} \cdot y$
9	<ID,x> - <num,2>	$x - \underline{2} \cdot y$
–	<ID,x> - <num,2>	$x - 2 \underline{\cdot} y$



Parsing top-down di $x - 2 \cdot y$ (3/3)

Regola	Derivazione	Input
–	goal	$\underline{x} - 2 \cdot y$
1	expr	$\underline{x} - 2 \cdot y$
3	expr - term	$\underline{x} - 2 \cdot y$
4	term - term	$\underline{x} - 2 \cdot y$
7	factor - term	$\underline{x} - 2 \cdot y$
9	<ID,x> - term	$x \underline{-} 2 \cdot y$
–	<ID,x> - term	$x - \underline{2} \cdot y$
5	<ID,x> - term * factor	$x - \underline{2} \cdot y$
7	<ID,x> - factor * factor	$x - \underline{2} \cdot y$
8	<ID,x> - <NUM,2> * factor	$x - \underline{2} \cdot y$
–	<ID,x> - <NUM,2> * factor	$x - 2 \underline{\cdot} y$
–	<ID,x> - <NUM,2> * factor	$x - 2 \cdot \underline{y}$
9	<ID,x> - <NUM,2> * <ID,y>	$x - 2 \cdot \underline{y}$
–	<ID,x> - <NUM,2> * <ID,y>	$x - 2 \cdot \underline{y}$



Parsing top-down e terminazione

È possibile effettuare un'altro tipo di espansione sistematica:

Regola	Derivazione	Input
–	goal	$\underline{x} - 2 \cdot y$
1	expr	$\underline{x} - 2 \cdot y$
2	expr + term	$\underline{x} - 2 \cdot y$
2	expr + term + term	$\underline{x} - 2 \cdot y$
2	expr + term + term + term	$\underline{x} - 2 \cdot y$
2	expr + term + term + term + . . .	$\underline{x} - 2 \cdot y$

- Una scelta sbagliata dell'espansione può comportare ricorsione infinita
- È possibile forzare il parser a fare la "scelta giusta"?

Grammatiche ricorsive a sinistra

- Una grammatica è ricorsiva a sinistra se esiste almeno un simbolo non terminale $a \in N$, tale che esiste una derivazione da a ad $a\alpha$ per qualche stringa $\alpha \in (N \cup T)^+$
- La grammatica che descrive le espressioni è ricorsiva a sinistra
- Per poterla analizzare con un parser top-down è necessario trasformarla in modo che le ricorsioni siano tutte a destra
- Non esiste altro modo di gestire il problema!

Eliminazione della ricorsione a sinistra

- Consideriamo un frammento della grammatica nella forma
  n ::= n α | β
  in cui α e β non cominciano per “n”
- Possiamo riscrivere la regola in modo equivalente come
  n ::= β m
  m ::= α m | ϵ
  in cui “m” è un nuovo simbolo non terminale
- Nota: bisogna tener conto anche delle ricorsioni indirette!

CFG per espressioni aritmetiche

Specifica

1	goal	::= expr
2	expr	::= term expr’
3	expr’	::= + term expr’
4		| - term expr’
5		| ϵ
6	term	::= factor term’
7	term’	::= * factor term’
8		| / factor term’
9		| ϵ
10	factor	::= NUM
11		| ID
12		| (expr)

- La grammatica è corretta, anche se non intuitiva
- Preserva le caratteristiche di quella originale (associatività e precedenze)
- Un parser top-down non ha problemi di terminazione
- Rimane ancora il problema dell’efficienza...

Scelta delle produzioni, non determinismo, efficienza

La scelta sbagliata di una produzione implica che un parser debba riconsiderare scelte fatte in precedenza (backtracking)

- È possibile utilizzare il contesto per migliorare l'efficienza?
- Quanto contesto è necessario acquisire (lookahead)?

È possibile evitare il backtracking

- Utilizzando il lookahead per decidere quale regola applicare
- In generale, può essere necessario un lookahead arbitrariamente grande
- Importanti sottoclassi di CFG richiedono solo un lookahead limitato
- La maggior parte dei costrutti dei linguaggi di programmazione ricade in queste sottoclassi

Parsing predittivo

- Idea: data una produzione del tipo a ::= α | β il parser deve essere in grado di scegliere tra α e β
- Definiamo FIRST(α) come l'insieme di token che appaiono come primo simbolo in qualche stringa derivata da α
- Se la regole a ::= α | β appare nella grammatica e si verifica che

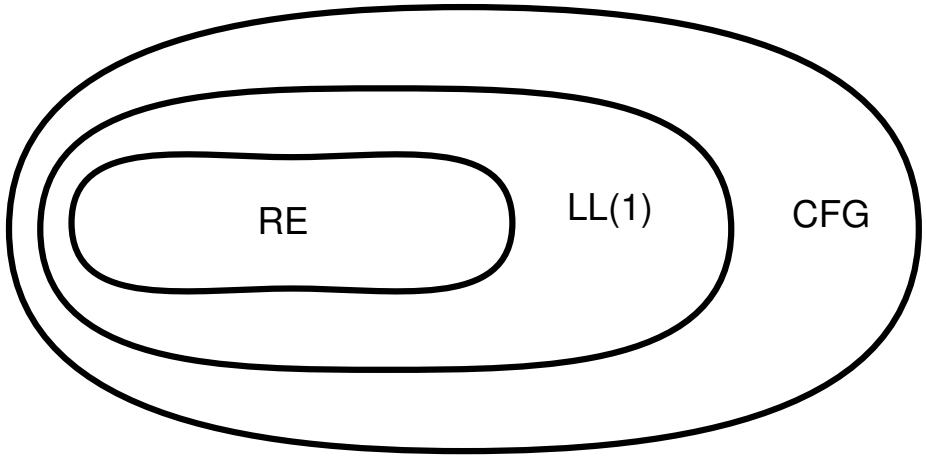
$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

allora al parser è sufficiente il lookahead di un simbolo per espandere in modo corretto la produzione

Parsing predittivo e grammatiche LL(1)

- Problema: le produzioni con ϵ sul lato destro non sono gestite correttamente da FIRST(α) $\cap$ FIRST(β) = $\emptyset$
- Se a ::= α | β e $\epsilon \in$ FIRST(α) allora ci dobbiamo assicurare che FIRST(β) sia disgiunto anche dai simboli terminali che possono immediatamente seguire "a"
- Definiamo FOLLOW(*a*) come l'insieme dei simboli terminali che possono seguire "*a*" in qualche stringa derivata da "*a*"
- Definiamo FIRST$^+$(*a* ::= α) come
  - FIRST(α) $\cup$ FOLLOW(*a*), se $\epsilon \in$ FIRST(α)
  - FIRST(α), altrimenti
- Una grammatica è LL(1) se e solo se *a* ::= α | β implica FIRST$^+$(*a* ::= α) $\cap$ FIRST$^+$(*a* ::= β) = $\emptyset$

Relazione tra CFG, RE e grammatiche LL(1)



- Le CFG (e le grammatiche LL(1)) consentono la specifica di tutti i linguaggi regolari
- Esistono linguaggi LL(1) che non sono esprimibili con una RE (ad esempio, la grammatica base di `SimpleCompiler`)
- Esistono CFG che descrivono linguaggi non LL(1)

Generazione di parser LL(1) (1/2)

È possibile costruire in modo automatico un parser LL(1)?

Partendo dalle definizioni di FIRST e FOLLOW:

FIRST(α) L'insieme dei token che compaiono come primo simbolo in qualche stringa derivata da $\alpha \in (T \cup NT)^+$

FOLLOW(a) l'insieme dei simboli che possono comparire immediatamente dopo "a"

è possibile sintetizzare il codice che corrisponde all'automa push-down deterministico (DPDA) riconoscitore

Generazione di parser LL(1) (2/2)

Data una grammatica LL(1) e i relativi insiemi FIRST e FOLLOW è possibile

- Costruire una tabella che contiene la funzione di transizione del DPDA
- Utilizzare un simulatore di DPDA che legge la funzione di transizione in forma tabellare

Esempio (grammatica per le espressioni)

- il simbolo non terminale “factor” ha tre espansioni
  factor ::= (expr) | ID | NUM
- lo schema per questa regola potrebbe essere

	+	-	*	/	ID	NUM	(	)	EOF
factor	–	–	–	–	R_x	R_y	R_z	–	–

Simulatore di DPDA

$t \leftarrow$ NEXTTOKEN()
PUSH(EOF)
PUSH(S) {$S \rightarrow$ *simbolo iniziale*}
$s \leftarrow$ TOP()
while (TRUE)
  if s = EOF **and** t = EOF **then**
    return TRUE
  else if $s \in T$ **then**
    if $s = t$ **then**
      POP()
      $t \leftarrow$ NEXTTOKEN()
    else return FALSE
  else
    if $\delta(s, t) = (a ::= b_1 \ldots b_k)$ **then**
      POP()
      PUSH(b_1) ... PUSH(b_k)
    else return FALSE
  $s \leftarrow$ TOP()

Funzione di transizione del DPDA

La costruzione della tabella δ richiede

- una riga per ogni simbolo in N e una colonna per ogni simbolo in T (dimensioni $O(|N||T|)$)
- un algoritmo per l'assegnazione dei valori alle celle

Il valore di $\delta(s, t)$, con $s \in N$ e $t \in T$, è

- la regola $s ::= \beta$, se $t \in \text{FIRST}^+(s ::= \beta)$, opppure
- errore, se quanto sopra non è definito

Se per una cella si danno più valori, allora la grammatica di partenza non è LL(1)

Esempio di parser LL(1) (1/2)

Specifica

1	goal	::= expr
2	expr	::= term expr’
3	expr’	::= + term expr’
4		| - term expr’
5		| ϵ
6	term	::= factor term’
7	term’	::= * factor term’
8		| / factor term’
9		| ϵ
10	factor	::= NUM
11		| ID
12		| (expr)

FIRST(goal) = FIRST(expr) =
FIRST(term) = FIRST(factor) =
{ID, NUM, (}
FIRST(expr’) = { +, -, ϵ}
FIRST(term’) = { *, /, ϵ}

FOLLOW(goal) = { EOF }
FOLLOW(expr) = {), EOF }
FOLLOW(expr’) = {), EOF }
FOLLOW(term) = { +, -,), EOF }
FOLLOW(term’) = { +, -,), EOF }
FOLLOW(factor) = { +, -, *, /,), EOF }

Esempio di parser LL(1) (2/2)

	+	-	*	/	ID	NUM	(	)	EOF
goal	–	–	–	–	1	1	1	–	–
expr	–	–	–	–	2	2	2	–	–
expr’	3	4	–	–	–	–	–	5	5
term	–	–	–	–	6	6	6	–	–
term’	9	9	7	8	–	–	–	9	9
factor	–	–	–	–	10	11	12	–	–

Outline

Derivazioni e parsing bottom-up (1/2)

Obiettivo del parsing è quello di costruire una derivazione, ossia una sequenza di passi:

$$S \Rightarrow \gamma_0 \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \text{frase}$$

- Ogni γ_i è una frase intermedia, a meno che non sia costituita da soli simboli terminali
- Per ottenere γ_i da γ_{i-1} è necessario espandere qualche non terminale "a" utilizzando la regola $a ::= \beta$
  - rimpiazzando l'occorrenza di a in γ_{i-1} con β per ottenere γ_i
  - utilizzando una strategia prefissata per la scelta del non terminale da espandere (derivazioni sinistre e destre)

Derivazioni e parsing bottom-up (2/2)

Un parser bottom-up costruisce la derivazione "all'indietro":

$$S \Leftarrow \gamma_0 \Leftarrow \ldots \Leftarrow \gamma_n \Leftarrow \text{frase}$$

Per ridurre γ_i a γ_{i-1} è necessario

- far coincidere il lato destro β di qualche produzione $a ::= \beta$ con una parte di γ_i, e
- rimpiazzare β con il lato sinistro "a" corrispondente

In termini di albero sintattico, significa procedere bottom-up

- i nodi senza genitori formano la frontiera superiore
- ogni sostituzione di β con "a" riduce la frontiera

Riduzioni e candidate

Ad ogni passo di riduzione il parser deve

- esaminare la frontiera superiore φ
- cercare una sottostringa β di φ, tale che
- la produzione $a ::= \beta$ è un passo della derivazione destra

Formalmente

- Una candidata in una frase intermedia γ è una coppia $< a ::= \beta, k >$ dove k è la posizione in γ del simbolo più a destra di β
- Data $< a ::= \beta, k >$, rimpiazzando β con "a" alla posizione k in γ_i si ottiene γ_{i-1}
- Se la grammatica G non è ambigua, allora ogni forma intermedia ammette una ed una sola candidata

Shift-reduce parser

Un parser per riduzione e scorrimento (shift-reduce) è un DPDA con quattro azioni

Shift il token successivo è spostato sullo stack

Reduce il suffisso destro della candidata $< a ::= \beta, k >$ è in cima allo stack

- ricerca di β nello stack
- pop degli elementi di β dallo stack e push del non terminale "a"

Accept termine del parsing con successo

Error termine del parsing con errore

Punto cruciale: quando eseguire l'azione di Shift piuttosto che l'azione di Reduce?

Parser LR(1)

Informalmente, una grammatica è LR(1) se, data una derivazione destra

$$S \Rightarrow \gamma_0 \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \text{frase}$$

è possibile

- isolare la candidata in ogni frase intermedia γ_i, e
- determinare la produzione medinte la quale ridurre

mediante una scansione di γ_i da sinistra a destra, leggendo al più un simbolo oltre la fine della candidata di γ_i

CFG, RE, linguaggi LL(1) e LR(1)