Linguaggi e Traduttori: Analisi semantica

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 ad attributi
- Utilizzo delle grammatiche ad attributi

Gestione efficiente degli aspetti semantici
- Traduzione guidata dalla sintassi
- Formati intermedi (IR)

Oltre la sintassi... (1/2)

```
void piccola(
    int a, int b,
    int c, int d) {
  int a, b, c, d;
  ...
}
void grande( ) {
  int f[3],g[0];
  int h, i, j, k;
  char* p;
  piccola(h,i,"ab",j,k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>.\n",p,q);
  p = 10;
}
```

Oltre la sintassi... (1/2)

```
void piccola(
    int a, int b,
    int c, int d) {
  int a, b, c, d;
  ...
}
void grande( ) {
  int f[3],g[0];
  int h, i, j, k;
  char* p;
  piccola(h,i,"ab",j,k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>.\n",p,q);
  p = 10;
}
```

Il codice contiene errori semantici:

- numero di argomenti a `prova`
- dichiaro `g[0]` e uso `g[17]`
- "ab" non è un `int`
- utilizzo di `f` improprio
- variable `q` non dichiarata
- `10` non è una stringa

Oltre la sintassi... (1/2)

```
void piccola(
    int a, int b,
    int c, int d) {
  int a, b, c, d;
  ...
}
void grande( ) {
  int f[3],g[0];
  int h, i, j, k;
  char* p;
  piccola(h,i,"ab",j,k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>.\n",p,q);
  p = 10;
}
```

Il codice contiene errori semantici:

- numero di argomenti a `prova`
- dichiaro `g[0]` e uso `g[17]`
- "`ab`" non è un `int`
- utilizzo di `f` improprio
- variable `q` non dichiarata
- `10` non è una stringa

Compilare codice richiede comprensione del significato!

Oltre la sintassi... (2/2)

Per generare codice il compilatore deve rispondere a diverse domande

- `x` è uno scalare, un vettore, una funzione? È dichiarato?
- Ci sono nomi usati e non dichiarati? (o viceversa?)
- Quale dichiarazione di `x` è pertinente al suo utilizzo?
- L'espressione `x*y+z` ha significato?
- In `a[i,j,k]`, `a` ha tre dimensioni?
- Dove è possibile allocare `z`? (registro? stack? heap?)
- In `f=15` come rappresento `15`?
- Quanti argomenti accetta una funzione?
- `*p` si riferisce al risultato di una `malloc()`?

Analisi semantica

La sintassi non è sufficiente per valutare la correttezza quando

- sono richiesti i valori degli elementi sintattici
- le informazioni necessarie non sono locali
- sono necessarie computazioni aggiuntive

Come risolvere il problema?

- Utilizzando metodologie formali, analoghe alle RE e alle CFG che consentano la gestione della semantica
- Utilizzando tecniche specifiche, ossia algoritmi e strutture dati specializzati per i vari compiti

Outline

Introduzione e motivazioni

Strumenti formali
- Grammatiche ad attributi
- Utilizzo delle grammatiche ad attributi

Gestione efficiente degli aspetti semantici
- Traduzione guidata dalla sintassi
- Formati intermedi (IR)

Grammatiche ad attributi

Un'estensione delle CFG:

- Aumentate da un insieme di regole
- Ogni simbolo in una derivazione è associato con un insieme di attributi (valori associati al simbolo)
- Le regole specificano come calcolare gli attributi

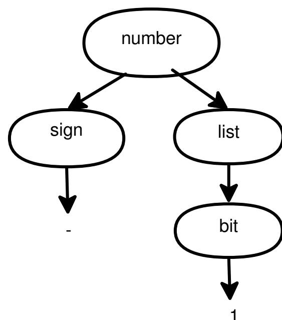
Esempio di utilizzo di grammatiche ad attributi:

```
number ::= sign list
sign   ::= + | -
list   ::= list bit | bit
bit    ::= 0 | 1
```

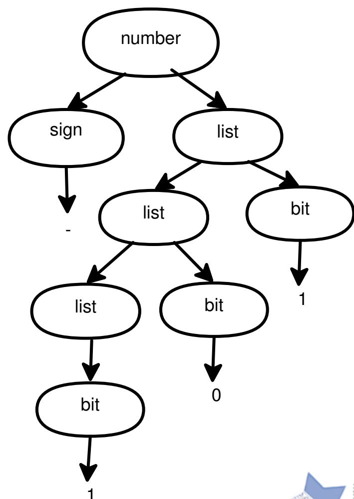
- Una grammatica che descrive numeri binari con segno
- Grammatica ad attributi per il calcolo del valore decimale

Derivazioni (solo sintassi)

Derivazione per "-1"



Derivazione per "-101"



Aggiunta delle regole di attribuzione

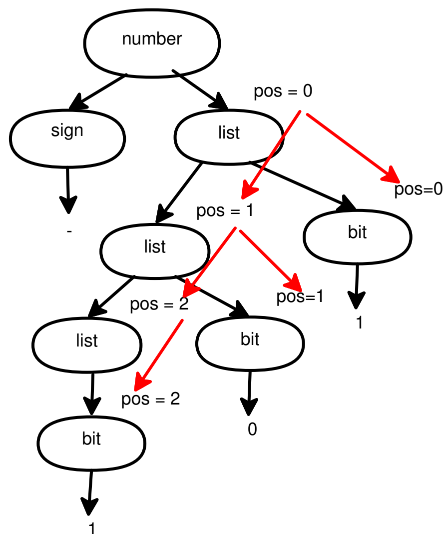
Produzioni	Regole di attribuzione

`number ::= sign list`	list.pos $\leftarrow$ 0
	if sign.neg **then**
	number.val $\leftarrow$ -list.val
	else
	number.val $\leftarrow$ list.val

`sign ::= +`	sign.neg $\leftarrow$ false
`sign ::= -`	sign.neg $\leftarrow$ true
$\texttt{list}_0 \leftarrow \texttt{list}_1$ `bit`	list_1.pos $\leftarrow$ list_0.pos + 1 bit.pos $\leftarrow$ list_0.pos list_0.val $\leftarrow$ list_1.val + bit.val
`bit ::= 0`	bit.val $\leftarrow$ 0
`bit ::= 1`	bit.val $\leftarrow 2^{\text{bit.pos}}$

Derivazioni (attributi ereditati)

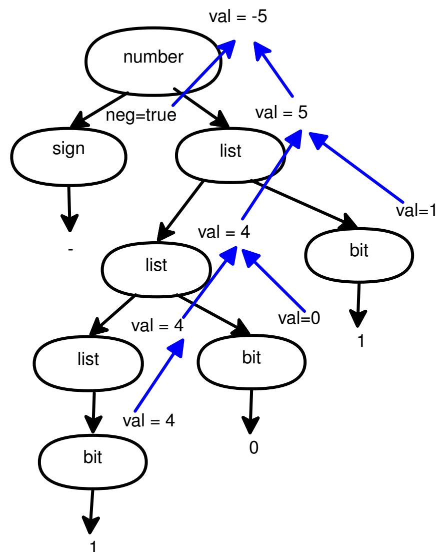
Derivazione per “-101”



- Nell’esempio: bit.pos e list.pos
- Esprimono direttamente il contesto
- Sono calcolati naturalmente dall’alto verso il basso
- Utilizzano valori assunti da “fratelli”, “genitori” e costanti

Derivazioni (attributi sintetizzati)

Derivazione per “-101”



- Nell’esempio: bit.val, list.val e sign.neg
- Sono calcolati naturalmente dal basso verso l’alto
- Utilizzano valori assunti da “figli” e costanti
- Si adattano bene a tecniche di parsing LR

Valutazione degli attributi

Metodi dinamici

- Costruzione dell'albero sintattico
- Costruzione del grafo delle dipendenze
- Ordinamento topologico del grafo delle dipendenze
- Calcolo degli attributi in base all'ordine

Metodi basati su regole

- Analisi delle regole in fase di generazione del compilatore
- Determinazione di un ordine statico
- Calcolo degli attributi in base all'ordine

Metodi specializzati

- Indipendenti dalle regole e dal parse treee
- Scelta di un ordine opportuno (in fase di progetto)

Outline

Introduzione e motivazioni

Strumenti formali
- Grammatiche ad attributi
- Utilizzo delle grammatiche ad attributi

Gestione efficiente degli aspetti semantici
- Traduzione guidata dalla sintassi
- Formati intermedi (IR)

Un esempio più complesso (1/2)

```
block0 ::= block1 assign
  | assign
assign ::= ID = expr
expr0 ::= expr1 + term
  | expr1 - term
  | term
term0 ::= term1 * factor
  | term1 : factor
  | factor
factor ::= ( expr )
  | NUM
  | ID
```

Stima del costo delle operazioni

- Ogni operazione ha un costo associato
- La somma dei costi è il costo del programma
- Assunzioni
  - ogni valore deve essere caricato dalla memoria
  - i valori non vengono riutilizzati

Un esempio più complesso (2/2)

- Un solo attributo sintetizzato (cost) per ogni simbolo
- I costi delle operazioni elementari sono costanti

Produzioni	Regole di attribuzione
$block_0$ ::= $block_1$ assign	$block_0$.cost ← $block_1$.cost + assign.cost
$block_0$ ::= assign	$block_0$.cost ← assign.cost
assign ::= ID = expr	assign.cost ← c(STORE) + expr.cost
$expr_0$ ::= $expr_1$ +/- term	$expr_0$.cost ← $expr_1$.cost + c(ADD/SUB) + term.cost
$expr_0$ ::= term	$expr_0$.cost ← term.cost
$term_0$ ::= $term_1$ */: factor	$term_0$.cost ← $term_1$.cost + c(MUL/DIV) + factor.cost
$term_0$ ::= factor	$term_0$.cost ← factor.cost
factor ::= (expr)	factor.cost ← expr.cost
factor ::= NUM	factor.cost ← c(REG)
factor ::= ID	factor.cost ← c(LOAD)

Discussione

Nell'esempio considerato

- Tutti gli attributi sono sintetizzati
- Le regole possono essere valutate bottom-up in un singola passata
- È sufficiente "aumentare" un parser LR
- La soluzione è semplice ed efficiente

Un'esempio più realistico?

- L'istruzione di LOAD avviene solo una volta in ogni blocco
- Dobbiamo tener traccia dei valori già caricati

Tracciamento dell'istruzione LOAD (1/2)

- Sono necessari insiemi che tengano traccia degli identificatori caricati prima e dopo l'esecuzione di un LOAD
- Tali insiemi devono essere gestiti come attributi

`factor ::= expr`	factor.cost $\leftarrow$ expr.cost expr.before $\leftarrow$ factor.before factor.after $\leftarrow$ expr.after
`factor ::= NUM`	factor.cost $\leftarrow$ c(REG) factor.after $\leftarrow$ factor.before

`factor ::= ID`	**if** ID.name $\notin$ factor.before **then**
	factor.cost $\leftarrow$ c(LOAD)
	factor.after $\leftarrow$ factor.before $\cup$ { ID.name }
	else
	factor.cost $\leftarrow$ 0
	factor.after $\leftarrow$ factor.before

Tracciamento dell'istruzione LOAD (2/2)

- Tracciare l'istruzione LOAD aumenta la complessità
- Il maggior onere si riscontra nelle regole che devono gestire il contesto

Un esempio di regola che deve gestire il contesto:

`expr`$_0$ `::= expr`$_1$ `+/- term`	expr_0.cost ← expr_1.cost + c(ADD/SUB) + term.cost expr_1.before ← expr_0.before term.before ← expr_1.after expr_0.after ← term.after

Discussione

- La gestione di computazione non locale aumenta considerevolmente la complessità delle grammatiche ad attributi
- La gestione della computazione locale (anche complessa) è relativamente semplice

Altri problemi

- Il passaggio del contesto aggiunge overhead computazionale
- Le regole necessarie alla gestione del contesto richiedono spazio e operazioni di copia
- Il risultato del parse è un albero con attributi che non sempre può essere lasciato implicito
- Le computazioni possono essere effettuate sull'albero finale, ma possono richiedere un'ulteriore passata

Outline

Introduzione e motivazioni

Strumenti formali
- Grammatiche ad attributi
- Utilizzo delle grammatiche ad attributi

Gestione efficiente degli aspetti semantici
- Traduzione guidata dalla sintassi
- Formati intermedi (IR)

Traduzione efficiente

Idea:

- Associare una porzione di codice ad ogni regola di produzione
- La porzione di codice viene eseguita
  - ad ogni riduzione (parser LR)
  - ad ogni discesa ricorsiva (parser LL)
- La gestione di codice arbitrario consente una maggiore flessibilità

In pratica sono necessari:

- Un meccanismo per attribuire nomi ai simboli sui lati sinistro e destro della regola di produzione
- Uno schema di valutazione (top-down o bottom-up)

... [TO DO] Aggiungere lucidi su formalismi ad-hoc ...

Outline

Introduction e motivazioni

Strumenti formali
- Grammatiche ad attributi
- Utilizzo delle grammatiche ad attributi

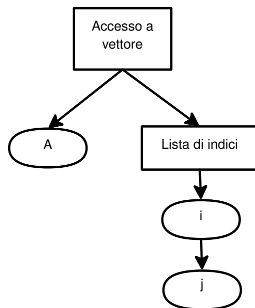
Gestione efficiente degli aspetti semantici
- Traduzione guidata dalla sintassi
- Formati intermedi (IR)

Tipologie di rappresentazione intermedia (IR)

- Strutturali
  - Rappresentazione del codice mediante grafi
  - Usate massicciamente nei traduttori
  - Intuitive, ma poco compatte
- Lineari
  - ISA astratta (ad es., 3 address, Stack machine)
  - Livello di astrazione variabile
  - Semplici e compatte
- Ibride
  - Combinazione di grafi e codice
  - Ad esempio: control flow

Livello di astrazione (1/2)

- La granularità di una IR determina la possibilità di eseguire certe ottimizzazioni e ne influenza il costo
- Esempio: accesso all'elemento di una matrice `A[i,j]`

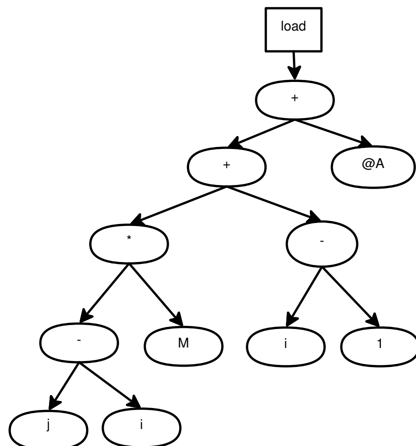


loadI	1	$\Rightarrow$	r_1
sub	r_j, r_1	$\Rightarrow$	r_2
loadI	M	$\Rightarrow$	r_3
mult	r_2, r_3	$\Rightarrow$	r_4
sub	r_i, r_1	$\Rightarrow$	r_5
add	r_4, r_5	$\Rightarrow$	r_6
loadI	@A	$\Rightarrow$	r_7
add	r_7, r_6	$\Rightarrow$	r_8
load	r_8	$\Rightarrow$	r_{Aij}

Ipotesi: matrice $M \times N$ in "column major order", base 1

Livello di astrazione (2/2)

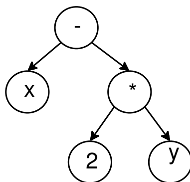
- Le IR strutturali sono solitamente considerate di “alto livello”, mentre le IR lineari di “basso livello”
- Nulla impedisce di gestire un diverso livello di granularità con entrambe i formalismi




```
loadArray A,i,j
```

Albero Sintattico Astratto (AST)

- L'albero sintattico privato dei nodi corrispondenti ai simboli non terminali



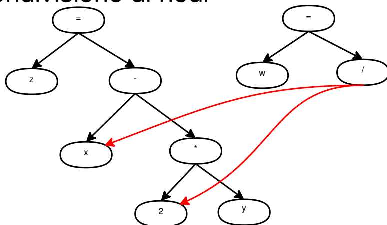
- Equivalente agli IR lineari che corrispondono alla visita dell'albero in qualche ordine

  `(x (2 y MUL) SUB)` forma postfissa

  `(SUB (MUL 2 y) x)` forma prefissa

Grafi diretti aciclici (DAG)

- I DAG sono una versione compatta degli AST in cui si attua una condivisione di nodi




```
z = x - 2 * y
w = x / 2
```

- Più compatti degli AST equivalenti
- Consentono al compilatore di ottimizzare il numero di valutazioni
- Richiedono l'utilizzo di una hash-table per la costruzione efficiente

Codice per macchina a stack

- Codice assembly nativo per CPU a stack (anni ’70)
- Rappresentazione intermedia per linguaggi di alto livello (UCSD-Pascal, ISA della JVM, CIL di .NET)
- Traduzione di (x - 2 * y)

```
push x
push 2
push y
multiply
subtract
```

- Compatto, ma al tempo stesso leggibile
- L’introduzione di nomi (registri) avviene implicitamente
- Semplice da generare e simulare

Three Address Code (3AC)

- Esistono diverse IR che usano questa filosofia
- Le istruzioni hanno una forma del tipo

  $x \leftarrow y$ `OP` z

  con un solo operatore (`OP`) e, al più, 3 nomi (`x`, `y` e `z`)
- Esempio per x - 2 * y

  $t \leftarrow 2 * y$

  $z \leftarrow x - t$
- Vantaggi
  - Somiglianza con le ISA attualmente più diffuse
  - Introduce nuovi nomi per la valutazione di espressioni
  - Compatto

Rappresentazioni per 3AC: quadruple

È la forma più semplice per 3AC:

- Tabella con 4 colonne
- Semplice struttura dei record
- Facile da riordinare
- Nomi espliciti

```
load    r1, y
loadI   r2, 2
mult    r3, r2, r1
load    r4,x
sub     r5, r4, r3
```

load	1	y	
loadi	2	2	
mult	3	2	1
load	4	x	
sub	5	4	3

Rappresentazioni per 3AC: triple

- Utilizzo dell'indice nella tabella come nome implicito
- 25% più compatto delle quadruple
- Molto più difficile da riordinare

(1)	load	1	y
(2)	loadi	2	
(3)	mult	(2)	(1)
(4)	load	x	
(5)	sub	(4)	(3)

Grafo del flusso di controllo (CG)

- I nodi del grafo sono sequenze di istruzioni (rappresentate in 3AC, o altri IR lineari)
- Gli archi rappresentano il flusso di controllo