ProgettoLC parteA Alvise Bruniera Relazione

Esercizio 1

Assumo che validare i dati in input sia opzionale, comunque faccio in modo che i risultati intermedi siano in forma valida, e controllo che le dimensioni di matrici e vettori in input alla funzione f siano corretti, per evitare di dover controllare situazioni inconsistenti nelle funzioni ausiliarie. Fornisco comunque anche una versione della funzione che valida l'input, perché ho scritto la validazione come ripasso e per aiutarmi nei test.

Per rappresentare i vettori ho scelto di imitare la rappresentazione delle matrici ed usare un albero (in questo caso binario) ed un parametro per la profondità. In questo modo è più semplice definire funzioni ricorsive che lavorano sia su vettori che matrici.

Sfruttando la proprietà distributiva della moltiplicazione matriciale, insieme alle proprietà commutativa ed associativa della somma, ed alla proprietà delle addizioni di trasposte, posso riscrivere la formula iniziale come:

$$
\begin{aligned}
& v^T(A + B^T)v + v^T(A^T + B)v = \\
& v^T((A + B^T)v + (A^T + B)v) = && [\text{distributività}] \\
& v^T((A + B^T + A^T + B)v) = && [\text{distributività}] \\
& v^T((A + B + A^T + B^T)v) = && [\text{commutatività}] \\
& v^T(((A + B) + (A^T + B^T))v) = && [\text{associatività}] \\
& v^T(((A + B) + (A + B)^T)v) && [\text{addizione}]
\end{aligned}
$$

In questo modo possiamo calcolarla in modo più efficiente usando le funzioni per: la somma tra matrici $\lambda A\ .\ \lambda B\ .\ A + B$, la somma tra una matrice e la sua trasposta $\lambda A\ .\ A + A^T$, l'applicazione di una matrice ad un vettore $\lambda A\ .\ \lambda v\ .\ Av$, ed infine il prodotto scalare tra vettori $\lambda v\ .\ \lambda u\ .\ v^T u$. Mettendo insieme queste funzioni implemento la funzione.

Tutte le altre funzioni definite sono d'appoggio per implementare queste quattro.

Nel file "test.txt" definisco alcune variabili su cui eseguire le query di test, e le query di test stesse, insieme ai risultati da aspettarsi in forma di commento per poter copiare ed incollare tutto il file in GHCi. Usando il tool di code coverage fornito insieme a GHC ho verificato che i test coprissero tutti i branch delle funzioni.

Esercizio 2

Per risolvere l'esercizio 2 ho seguito questa traccia:

1. Definizione equivalente
2. Calcolo di FIRST e primo controllo che sia effettivamente LL(1)
3. Calcolo di FOLLOW e verificare che effettivamente è LL(1)
4. Costrizione tabella di parsing
5. Analisi degli errori
6. Parsing della stringa di esempio

Per definire una grammatica equivalente ho osservato che i non terminali erano organizzati secondo una gerarchia stretta (P>B>A>E), nessun non terminale produceva non terminali più "alti", e quindi non esistevano cicli indiretti (P->?->P, ma solo cicli diretti). Quindi potevo semplicemente prendere in esame un non terminale per volta, partendo da quello più in basso (inizio da E e finisco con P), capire che grammatica produce e cercare di renderlo LL(1) ad intuito. Così ho ottenuto la grammatica seguente:

$$
\begin{aligned}
A &\to \quad \text{pred } A' \\
A' &\to \quad EA'' \\
A'' &\to \quad ,EA'' \quad | \epsilon \\
B &\to \quad A \quad | EB' \\
B' &\to \quad = E \quad | > E \\
E &\to \quad \text{id}E' \quad | \text{num}E' \\
E' &\to \quad + E \quad | \epsilon \\
P &\to \quad (PP' \quad | \text{not } P \quad | B \\
P' &\to \quad \text{or } P) \quad | \text{and } P)
\end{aligned}
$$

Inizialmente avevo fatto un errore nella definizione di A' che ho notato costruendo la tabella e risolto aggiungendo A''. Si potrebbe scrivere una grammatica più semplice che non ha bisogno di A'', ma in seguito ha permesso di differenziare meglio alcuni errori, quindi ho deciso che era meglio così.

Successivamente ho calcolato i FIRST di ogni produzione come mostrato a lezione, e li ho organizzati in base al simbolo che li produce. Non riporto il FIRST dei non terminali stessi (necessario per calcolare il FOLLOW) perché è semplicemente l'unione dei FIRST delle sue produzioni.

$$
\begin{aligned}
A &: (\text{pred } A') \mapsto \{\text{pred}\} \\
A' &: (EA'') \mapsto \{\text{id}, \text{num}\} \\
A'' &: (, EA'') \mapsto \{,\} \mid (\epsilon) \mapsto \{\epsilon\} \\
B &: (A) \mapsto \{\text{pred}\} \mid (EB') \mapsto \{\text{id}, \text{num}\} \\
B' &: (= E) \mapsto \{=\} \mid (> E) \mapsto \{>\} \\
E &: (\text{id}E') \mapsto \{\text{id}\} \mid (\text{num}E') \mapsto \{\text{num}\} \\
E' &: (+E) \mapsto \{+\} \mid (\epsilon) \mapsto \{\epsilon\} \\
P &: ((PP') \mapsto \{(\} \mid (\text{not}P) \mapsto \{\text{not}\} \mid (B) \mapsto \{\text{id}, \text{num}, \text{pred}\} \\
P' &: (\text{or } P)) \mapsto \{\text{or}\} \mid (\text{and } P)) \mapsto \{\text{and}\}
\end{aligned}
$$

Osserviamo che per ogni coppia d produzioni dello stesso non terminale, i due FIRST non hanno intersezioni. Potrebbe essere una grammatica LL(1), ma bisogna ancora controllare le due ε-produzioni. Quini calcolo il FOLLOW come mostrato a lezione:

$$
\begin{aligned}
FOLLOW(A) &= \{\$,), \text{or}, \text{and}\} \\
FOLLOW(A') &= \{\$,), \text{or}, \text{and}\} \\
FOLLOW(A'') &= \{\$,), \text{or}, \text{and}\} \\
FOLLOW(B) &= \{\$,), \text{or}, \text{and}\} \\
FOLLOW(B') &= \{\$,), \text{or}, \text{and}\} \\
FOLLOW(E) &= \{,\ \ , =, >, \$,), \text{or}, \text{and}\} \\
FOLLOW(E') &= \{,\ \ , =, >, \$,), \text{or}, \text{and}\} \\
FOLLOW(P) &= \{\$,), \text{or}, \text{and}\} \\
FOLLOW(P') &= \{\$,), \text{or}, \text{and}\}
\end{aligned}
$$

Dobbiamo verificare che i FOLLOW di A'' ed E' non abbiano intersezioni con i loro FIRST, ed è così. Quindi è una grammatica LL(1) come mostrato a lezione.

Costruisco la tabella di parsing come mostrato a lezione ed inserisco la gestione degli errori nelle caselle vuote. Cerco di analizzare in che situazione potrei trovarmi se sono arrivato a quello stato, per capire cosa potrebbe significare il simbolo incontrato. Ho anche guardato come si comporterebbero in stringhe simili a quella fornita come esempio, per ottenere un buon risultato almeno in quella.

Quasi tutti i terminali (tranne la parentesi chiusa) vengono inseriti solo in cima allo stack e solo quando vengono letti nel lookahead. Quindi non necessitano di gestire un mismatch tra stack e lookahead. La partentesi chiusa invece si, in quel caso stampa un semplice messaggio.

Infine ho svolto l'esecuzione del parser per la stringa d'esempio, indicando ogni azione.

Tabella di parsing

NT\la	(	)	+	,	=	>	and	id	not	num	or	pred	$
A	2	2	2	2	2	2	2	2	2	2	2	A-> pred A'	1
A'	3	10	4	5	12	12	10	A'-> EA''	12	A'-> EA''	10	3	1
A''	3	A''-> ε	4	A''-> ,EA''	6	6	A''-> ε	11	6	11	A''-> ε	6	A''-> ε
B	2	2	2	2	2	2	2	B->EB'	2	B->EB'	2	B->A	1
B'	3	3	13	3	B'-> =E	B'-> >E	8	14	3	14	8	3	1
E	3	9	4	3	3	3	16	E-> id E'	3	E-> num E'	16	3	1
E'	3	E'-> ε	E'-> +E	E'-> ε	E'-> ε	E'-> ε	E'-> ε	15	3	15	E'-> ε	3	E'-> ε
P	P-> (PP'	3	17	3	18	18	7	P-> B	P-> not P	P-> B	7	P-> B	1
P'	19	3	19	3	19	19	P'-> and P)	19	19	19	P'-> or P)	19	1

1	print "Incomplete program"; exit
2	print "Internal error"; exit
3	print "Unexpected {lookahead} "; skip lookahead
4	print "Missing operand "; pop; push E'A''
5	print "Missing expression "; pop; push A''
6	print "Cannot use {lookahead} in predicate"; remove until ",) and or $" excluded
7	print "Missing operand "; pop; push P';
8	print "Incomplete compare "; pop
9	print "Incomplete expression "; pop
10	print "Empty predicate "; pop
11	print "Missing comma "; push E
12	pop; push A''
13	print "Missing second operand "; skip lookahead
14	print "Missing operator"; pop; push E
15	print "Missing operand"; pop; push E
16	print "Missing expression"; pop
17	print "Missing pred"; pop; push E'A''
18	print "Missing operand"; pop; push B'
19	print "Missing operator"; pop; push P

Tabella dei mismatch sui non terminali

st\la	(	)	+	,	=	>	and	id	not	num	or	pred	$
(	acc	2	2	2	2	2	2	2	2	2	2	2	2
)	4	acc	4	4	4	4	4	4	4	4	4	4	1
+	2	2	acc	2	2	2	2	2	2	2	2	2	2
,	2	2		acc	2	2	2	2	2	2	2	2	2
=	2	2	2	2	acc	2	2	2	2	2	2	2	2
>	2	2	2	2	2	acc	2	2	2	2	2	2	2
and	2	2	2	2	2	2	acc	2	2	2	2	2	2
id	2	2	2	2	2	2	2	acc	2	2	2	2	2
not	2	2	2	2	2	2	2	2	acc	2	2	2	2
num	2	2	2	2	2	2	2	2	2	acc	2	2	2
or	2	2	2	2	2	2	2	2	2	2	acc	2	2
pred	2	2	2	2	2	2	2	2	2	2	2	acc	2
$	3	3	3	3	3	3	3	3	3	3	3	3	halt

1	print "Incomplete program"; exit
2	print "Internal error"; exit
3	print "Unexpected {lookahead} "; skip lookahead
4	print "Missing)"; pop

Esecuzione su una stringa d’esempio

Stack	Input	Azione
$P	((pred and id + num > num)) or (id not = id and pred num ,))$	P->(PP'
$P'P(	((pred and id + num > num)) or (id not = id and pred num ,))$	acc
$P'P	(pred and id + num > num)) or (id not = id and pred num ,))$	P->(PP'
$P'P'P(	(pred and id + num > num)) or (id not = id and pred num ,))$	acc
$P'P'P	pred and id + num > num)) or (id not = id and pred num ,))$	P->B
$P'P'B	pred and id + num > num)) or (id not = id and pred num ,))$	B->A
$P'P'A	pred and id + num > num)) or (id not = id and pred num ,))$	A->pred A'
$P'P'A' pred	pred and id + num > num)) or (id not = id and pred num ,))$	acc
$P'P'A'	and id + num > num)) or (id not = id and pred num ,))$	Empty predicate; pop
$P'P'	and id + num > num)) or (id not = id and pred num ,))$	P'-> and P)
$P')P and	and id + num > num)) or (id not = id and pred num ,))$	acc
$P')P	id + num > num)) or (id not = id and pred num ,))$	P->B
$P')B	id + num > num)) or (id not = id and pred num ,))$	B->EB'
$P')B'E	id + num > num)) or (id not = id and pred num ,))$	E->id E'
$P')B'E' id	id + num > num)) or (id not = id and pred num ,))$	acc
$P')B'E'	+ num > num)) or (id not = id and pred num ,))$	E'->+E
$P')B'E+	+ num > num)) or (id not = id and pred num ,))$	acc
$P')B'E	num > num)) or (id not = id and pred num ,))$	E->num E'
$P')B'E' num	num > num)) or (id not = id and pred num ,))$	acc
$P')B'E'	> num)) or (id not = id and pred num ,))$	E->ε
$P')B'	> num)) or (id not = id and pred num ,))$	B'-> >E
$P')E>	> num)) or (id not = id and pred num ,))$	acc
$P')E	num)) or (id not = id and pred num ,))$	E-> num E'
$P')E' num	num)) or (id not = id and pred num ,))$	acc
$P')E'	)) or (id not = id and pred num ,))$	E->ε
$P')	)) or (id not = id and pred num ,))$	acc
$P'	) or (id not = id and pred num ,))$	Unexpected); skip)
$P'	or (id not = id and pred num ,))$	P'-> or P)
$)P or	or (id not = id and pred num ,))$	acc
$)P	(id not = id and pred num ,))$	P->(PP'
$)P'P(	(id not = id and pred num ,))$	acc
$)P'P	id not = id and pred num ,))$	P->B
$)P'P	id not = id and pred num ,))$	B->EB'
$)P'B'E	id not = id and pred num ,))$	E->id E'
$)P'B'E' id	id not = id and pred num ,))$	acc
$)P'B'E'	not = id and pred num ,))$	Unexpected not; skip not
$)P'B'E'	= id and pred num ,))$	E'->ε

$)P'B'	= id and pred num ,))$	B'-> =E
$)P'E=	= id and pred num ,))$	acc
$)P'E	id and pred num ,))$	E-> id E'
$)P'E' id	id and pred num ,))$	acc
$)P'E'	and pred num ,))$	E'->ε
$)P'	and pred num ,))$	P'-> and P)
$))P and	and pred num ,))$	acc
$))P	pred num ,))$	P->B
$))B	pred num ,))$	B->A
$))A	pred num ,))$	A-> pred A'
$))A' pred	pred num ,))$	acc
$))A'	num ,))$	A'->EA''
$))A''E	num ,))$	E-> num E'
$))A''E' num	num ,))$	acc
$))A''E'	,))$	E'->ε
$))A''	,))$	A''->,EA''
$))A''E,	,))$	acc
$))A''E	))$	Incomplete expression; pop
$))A''	))$	A''->ε
$))	))$	acc
$)	)$	acc
$	$	halt