Lösningsförslag till Tentamen

Översättarteknik, 5p
22 December 1993

Uppgift 1 (12 p)

Ganska lätt va?

Uppgift 2 (12 p) (4 + 6 + 2)

Följande grammatik beskriver uttryck i ett funktionellt språk.

expr → atom
expr → expr atom
atom → **num**
atom → **id**
atom → (expr)

a) Skriv om grammatiken så att den blir LL(1). Förklara vilka omskrivningar du gör.

Eliminering av vänsterrekursion

	Produktion	**FIRST**	**FOLLOW**
(1)	expr → atom expr’	(, **id**, **num**	$,)
(2)	expr’ → atom expr’	(, **id**, **num**	$,)
(3)	expr’ → ε	ε	
(4)	atom → **num**	**num**	
(5)	atom → **id**	**id**	
(6)	atom → (expr)	(	

b) Konstruera en LL(1)-parser för den omskrivna grammatiken i a).

Tabell 1:

	id	**num**	**(**	**)**	**$**
expr	1	1	1		
expr’	2	2	2	3	3
atom	5	4	6		

c) Visa hur LL-parsern analyserar strängen `id (id num)`.

	Stack	Input	Produktion
1)	$E	**id (id num)$**	E → A E’
2)	$E’A	**id (id num)$**	A → **id**
3)	$E’**id**	**id (id num)$**	
4)	$E’	**(id num)$**	E’ → A E’
5)	$E’A	**(id num)$**	A → (E)
6)	$E’)E(	**(id num)$**	
7)	$E’)E	**id num)$**	E → A E’
8)	$E’)E’A	**id num)$**	A → **id**
9)	$E’)E’**id**	**id num)$**	
10)	$E’)E’	**num)$**	E’ → A E’
11)	$E’E’A	**num)$**	A → **num**
12)	$E’)E’**num**	**num)$**	
13)	$E’)E’	**)$**	E’ → ε
14)	$E’)	**)$**	
15)	$E’	**$**	E’ → ε
16)	$	**$**	**accept**

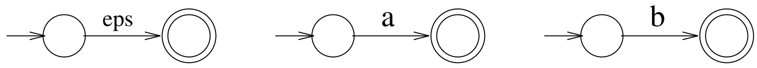
Uppgift 3 (12 p)

Utgående från följande reguljära uttryck:
((ε | a) b*)*

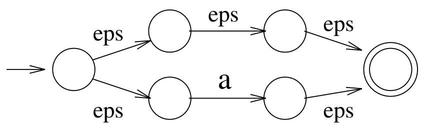
a) Konstruera en icke-deterministisk finit automat som accepterar det språk som beskrivs av uttrycket. Använd de metoder som beskrivs i boken med följande tillägg:
   1. Ingen övergång får finnas från ett accepterande tillstånd.
   2. Ingen övergång får finnas till starttillståndet.

Maskiner för:

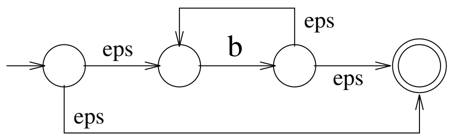
eps, a, b



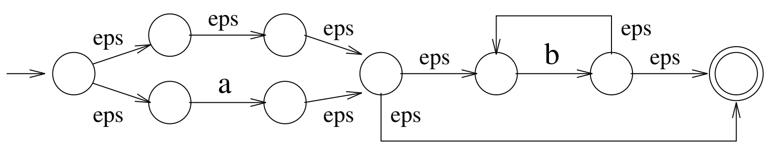
eps | a



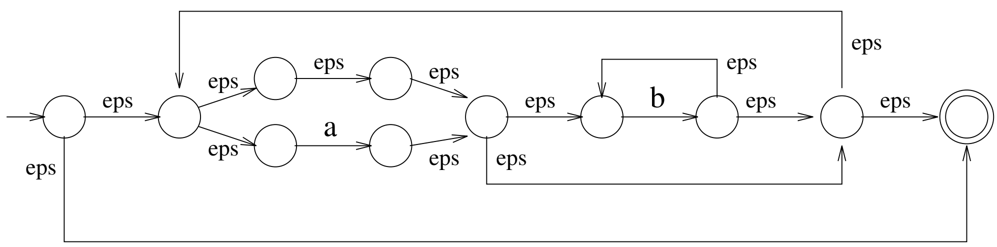
b*

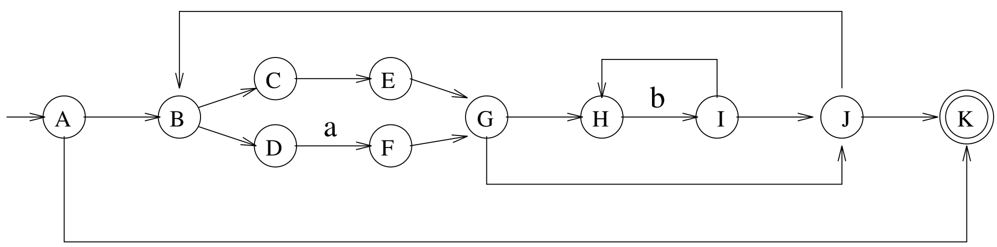


(eps | a)b*

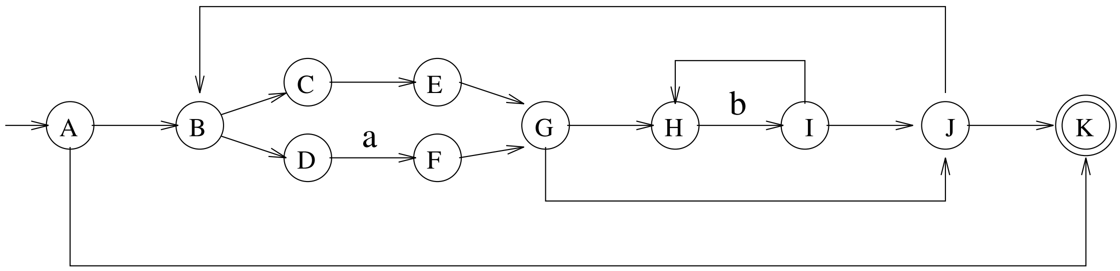


((eps | a)b*)*

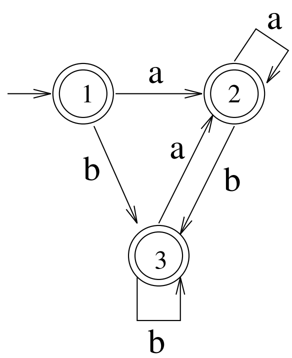




b)



Current State		Input a	b
1	[ABCDEGHJK]	[BCDEFGHJK]	[BCDEGHIJK]
2	[BCDEFGHJK]	[BCDEFGHJK]	[BCDEGHIJK]
3	[BCDEGHIJK]	[BCDEFGHJK]	[BCDEGHIJK]



Uppgift 4 (10 p) (6 + 4)

expr → λ **id .** expr
expr → **(** expr expr **)**
expr → **id**
expr → **(** expr **)**

a) Konstruera en SLR(1)-parser för grammatiken. Om du stöter på konflikter välj någon tolkning och kommentera denna. Använd _ istället för **.** i LR(0)-items för att undvika missförstånd.

	Produktion		FIRST	FOLLOW
(0)	S →	expr	λ, (, **id**	**$**
(1)	expr →	λ **id .** expr	λ	λ, (,), **id**, **$**
(2)	expr →	(expr expr)	(	
(3)	expr →	**id**	**id**	
(4)	expr →	(expr)	(	

0: V[ε] = { S → _expr
expr → _ λ **id .** expr
expr → _ (expr expr)
expr → _ **id**
expr → _ (expr) }
goto(0, expr) =
1: V[expr] = { S → expr _ }
goto(0, λ) =
2: V[λ] = { expr → λ _ **id .** expr }
goto(0, () =
3: V[(] = { expr → (_ expr expr)
expr → (_ expr)
expr → _ λ **id .** expr
expr → _ (expr expr)
expr → _ **id**
expr → _ (expr) }
goto(0, **id**) =
4: V[**id**] = { expr → **id** _ }
goto(2, **id**) =
5: V[λ **id**] = { expr →λ **id** _ **.** expr }
goto(3, expr) =
6:V[(expr] = { expr → (expr_ expr)
expr → (expr _)
expr → _ λ **id .** expr
expr → _ (expr expr)
expr → _ **id**
expr → _ (expr) }
goto(3, λ) = 2: V[(λ]
goto(3, () = 3: V[((]
goto(3, **id**) = 4: V[(**id**]
goto(5, **.**) =
7:V[λ **id .**] = { expr → _λ **id .** _expr
expr → _ λ **id .** expr
expr → _ (expr expr)
expr → _ **id**
expr → _ (expr) }
goto(6, expr) =
8:V[(expr expr] ={ expr → (expr expr _) }
goto(6,)) =
9:V[(expr)] = { expr → (expr) _ }
goto(6, λ) = 2: V[(expr λ]
goto(6, () = 3: V[(expr (]
goto(6, **id**) = 4: V[(expr **id**]
goto(7, expr) =
10:V[λ **id .** expr] ={ expr → λ **id .** expr _ }
goto(7, λ) = 2: V[λ **id .** λ]
goto(7, () = 3: V[λ **id .** (]
goto(7, **id**) = 4: V[λ **id . id**]
goto(8,)) =
11:V[(expr expr)] ={ expr → (expr expr) _ }

Tabell 2:

	λ	**id**	(	)	**.**	**$**	expr
0	s2	s4	s3				1
1						acc	
2		s5					
3	s2	s4	s3				6
4	r3	r3	r3	r3		r3	
5					s7		
6	s2	s4	s3	s9			8
7	s2	s4	s3				10
8				s11			
9	r4	r4	r4	r4		r4	
10	r1	r1	r1	r1		r1	
11	r2	r2	r2	r2		r2	

b) Visa hur SLR-parsern analyserar strängen λx**.**λy**.**(xy).

	Stack	Input	Produktion
1)	$0	**λid. λid.(id id)$**	
2)	$0λ2	**id. λid.(id id)$**	
3)	$0λ2**id** 5	**.λid.(id id)$**	
4)	$0λ2**id** 5.7	**λid.(id id)$**	
5)	$0λ2**id** 5.7λ2	**id.(id id)$**	
6)	$0λ2**id** 5.7λ2**id** 5	**.(id id)$**	
7)	$0λ2**id** 5.7λ2**id** 5.7	**(id id)$**	
8)	$0λ2**id** 5.7λ2**id** 5.7(3	**id id)$**	
9)	$0λ2**id** 5.7λ2**id** 5.7(3**id** 4	**id)$**	expr → **id**
10)	$0λ2**id** 5.7λ2**id** 5.7(3E6	**id)$**	
11)	$0λ2**id** 5.7λ2**id** 5.7(3E6**id** 4	**)$**	expr → **id**
12)	$0λ2**id** 5.7λ2**id** 5.7(3E6E8	**)$**	
13)	$0λ2**id** 5.7λ2**id** 5.7(3E6E8)11	**$**	expr → (expr expr)
14)	$0λ2**id** 5.7λ2**id** 5.7E10	**$**	expr → λ **id .** expr
15)	$0λ2**id** 5.7E10	**$**	expr → λ **id .** expr
16)	$0E1	**$**	
17)	**accept**		

Uppgift 5 (8 p) (2 + 2 + 2 + 2)

Förklara kortfattat följande begrepp

a) En **tvetydig** grammatik är en grammatik som kan parsas på mer än ett sätt. (Mer än ett parseträd.).

b) En **vänsterrekursiv** grammatik har en icketerminal A sådan att ett högerled som börjar på samma icketerminal A, kan härledas från A i ett eller flera steg.

$$A \Rightarrow \ldots \Rightarrow A\alpha$$

c) En **recursive descent** parser. En boolsk funktion för varje icke-terminal, som försöker matcha icke-terminalens alla högerled, i tur och ordning tills en matchning hittas. Returnerar TRUE om matchning lyckas, annars returneras FALSE. Högerleden “imiteras” genom att terminalsymboler matchas och för icke-terminaler så anropas motsvarande funktion.

d) En **prediktiv parser** kan genom att titta på nästa symbol på input bestämma vilken produktion som ska användas. (Annars som ovan fast den behöver bara prova ett högerled.) Skulle det finnas två möjliga högerled kan ej denna typ av parser användas.

Uppgift 6 (4 p) (2 + 2)

Använd tekniken med listor av kvadrupelnummer och *backpatching* för att skriva ett översättningsschema för följande produktioner:

a)

$S \rightarrow$ **do** M_1 S_1 **while** $M_2(E)$ {
 backpatch(S_1.nextlist, M_2.quad)
 backpatch(E.truelist, M_1.quad)
 S.nextlist := E.falselist
}

b)

$E \rightarrow$ E_1 **nand** M E_2 {
 backpatch(E_1.truelist, M.quad)
 E_1.truelist := merge(E_1.falselist, E_2.falselist)
 E.falselist := E_2.truelist
}

E_1 **nand** E_2 är falskt om både E_1 och E_2 är sanna, annars sant.

Uppgift 7 (2 p)

För varje prioritetsnivå skapar vi en ny icke-terminal. Vi ändrar på omskrivningsreglerna för icke-terminalerna på den närmast lägre prioritetsnivån från att vara av typen A → A **op** A till att innehålla den nya icke-terminalen, A → A **op** Q om operatorn är vänsterassociativ och till A → Q **op** A om operatorn är högerassociativ. Det vill säga om en operator är högerassociativ ska den skrivas med en högerrekursiv regel (vänsterassociativ med vänsterrekursiv). Dessutom måste regeln A → Q läggas till eftersom vi måste kunna "hoppa över" produktioner med den lägre prioriterade operatorn. Q ska sedan skirivas om till alla regler som finns för nästa prioritetsnivå. Exempel:

$$A \rightarrow A\ \textbf{oplåg}_1\ A \quad | \quad \ldots \quad | A\ \textbf{oplåg}_n\ A$$
$$A \rightarrow A\ \textbf{ophög}_1\ A \quad | \quad \ldots \quad | A\ \textbf{ophög}_n\ A$$
$$A \rightarrow \alpha$$

skrivs till följande, om alla operatorer är vänsterassociativa:

$$A \rightarrow A\ \textbf{oplåg}_1\ B \quad | \quad \ldots \quad | A\ \textbf{oplåg}_n\ B$$
$$A \rightarrow B$$
$$B \rightarrow B\ \textbf{ophög}_1\ C \quad | \quad \ldots \quad | B\ \textbf{ophög}_n\ C$$
$$B \rightarrow C$$
$$C \rightarrow \alpha$$

Uppgift 8 (2 p)

Specificera ett översättningsschema där man kontrollerar typen på uttryck bestående av identifierare och funktionsapplikation

$E \rightarrow$ **id** { E.types := lookup(**id** .entry) }

$E \rightarrow E_1\ (\ E_2\ )$ { E.types := { **t** | det existerar ett **s** i E_2.types så att **s** → **t** finns i E_1.types } }

Uppgift 9 (2 p)

a) Konflikter uppkommer när parsern inte vet om den ska shifta eller reducera trots att den vet vad som finns på stacken och vad nästa token är . För reduce/reduce gäller samma sak fast här har den flera olika produktioner att välja mellan. På stacken finns i båda fallen något som kan reduceras, ett handtag. I reduce/reduce fallet har man alltså flera högerled som motsvarar stackinnehållet och dessutom så finns inputsymbolen med i FOLLOW -mängerna till de aktuella vänstersidorna. I shift/reduce fallet så kan inputsymbolen följa efter det som kan reduceras från stacken, dvs vi ska reducera, men dessutom så kan man genom att shifta inputsymbolen få någonting annat på stacken som senare kan reduceras.

b) Om det är möjligt kan man använda en kraftfullare LR-parser . T ex genom att övergå från SLR till LALR eliminerar man alla shift/reduce konflikter (givet en deterministisk kontextfri grammatik).
I båda fallen måste man nog titta efter i sina "items" för att se efter hur man ska lösa dessa problem.
shift/reduce : Om det som ligger på stacken har högre prioritet än det på input eller om operatorn är vänsterassociativ så ska man välja att reducera, annars shifta.
reduce/reduce : Svårare att ge någon generell regel, men genom att titta på det item som ger upphov till konflikten kan man kanske genom att studera FOLLOW -mängden komma fram till vilken reduktion som ska utföras.