طراحی کامپایلرها

نیم‌سال دوم ۹۹ـ۹۸

مدرس: محمدرضا بهرامی

دانشکده‌ی مهندسی کامپیوتر

پاسخ تمرین سری دوم | پارسرها | موعد تحویل: ۹۹/۹/۵

مسئله‌ی ۱. گرامر مبهم

پاسخ.

الف) گرامر اشاره شده مبهم است! چرا که لااقل دو تا leftmost derivation برای کلمه زیر دارد.

$$00111 : S \to AS \to 0A1S \to 0A11S \to 00111S \to 00111$$

$$00111 : S \to AS \to A1S \to 0A11S \to 00111S \to 00111$$

برای رفع ابهام با روش لیرینگ یک لایه اضافه می‌کنیم تا هررشته بصورت یکتا پارس شود.

$$S \to AS|\epsilon$$

$$A \to 0A1|B$$

$$B \to B1|01$$

ب) ابهام در گرامرها می‌تواند دو مشکل بوجود بیاورد.

اولین مشکل این است که پارسرهای $predictive$ که مبتنی بر حدس قاعده‌های تولید هستند توانایی پارس این گرامرهارا ندارند چون حداقل دو درخت پارس موجود است.

دومین مشکل ناشی از تحلیل معنایی این گرامرها است. بطور گرامر مبهم زیر را در نظر بگیرید.

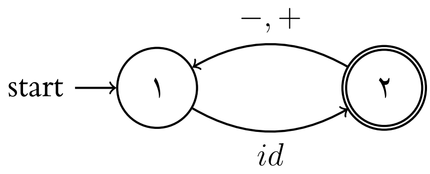
$E \to E + E$
$E \to E * E$
$E \to digit$

در تحلیل معنایی رشته $۳ + ۲ * ۲$ با این گرامر مشکل داریم زیرا نمی‌توانیم تشخیص دهیم که عملیات ضرب در ابتدا انجام می‌شود یا عملیات جمع انجام می‌شود.

مسئله‌ی ۲. عبارت آرمانی

پاسخ.

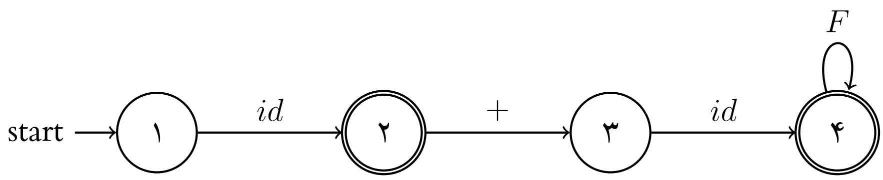
۱. گراف نحوی زیر را در نظر بگیرید:



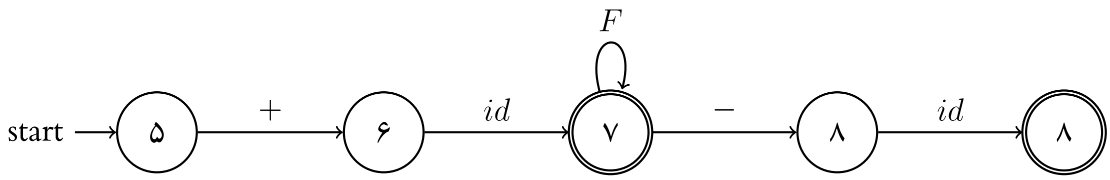
فرض کنید از استک اضافی ای که داریم به این صورت استفاده کنیم که هر باری که + دیدیم، یک + در استک‌مان پوش کنیم. سپس هر برای که ــ دیدیم، یک + را از استک پاپ کنیم. در واقع داریم هر ــ را با یکی از + های قبلی‌ش متناظر می‌کنیم. اگر در زمان پیمایش رشته، جایی ــ دیدیم و استک خالی بود یا که پس از پاپ کردن +، استک خالی شد، یعنی تا آنجا تعداد ــ ها بیشتر یا مساوی تعداد +ها بوده است و رشته نباید اکسپت شود. پس در این صورت ما هم الگوریتم پیمایش رشته را تمام می‌کنیم. اگر در هنگام پیمودن رشته هیچ موقع به این مشکل نخوریم و در نهایت در استیت اکسپت باشیم، رشته‌مان حتما آرمانی بوده است و ما نیز به درستی رشته را می‌پذیریم.

۲. گراف نحوی زیر را در نظر بگیرید:

Graph E :



Graph F :



ادعا می‌کنیم این گراف تمامی رشته‌های آرمانی را می‌پذیرد. زیرا حتما اولین عملگر بین identifier ها + است. اگر عملگر دیگری نداشته باشیم، کارمان در همان گراف E تمام می‌شود. در غیر این صورت پس از دیدن دومین identifier به گراف F می‌رویم. چون در عبارت آرمانی تعداد +ها بیشتر از تعداد ــ ها است، گراف F به این صورت طراحی شده است:

پس از دیدن اولین عملگر در رشته، هر بار که در جایگاه عملگرها، عملگر + را ببینیم، یک بار F را در استک parse پوش می‌کنیم و به ابتدای گراف F می‌رویم. اگر در جایگاه عملگر، عملگر ــ را دیدیم، کارمان در گراف F تمام می‌شود و یک F را از استک پاک می‌کنیم. هم‌چنین واضح است که برای valid بودن کل عبارت، بلافاصله پس از هر عملگر باید یک identifier هم ببینیم.

پس در واقع زمانی که ــ می‌بینیم، یک جمع متناظر قبلش دیده‌ایم و هیچ موقع تعداد ــ ها بیشتر از تعداد +ها نمی‌شود. ضمنا چون فرض کرده‌ایم اولین عملگرمان + است و در گراف E آن را دیده‌ایم، تعداد +ها همیشه از تعداد ــ ها بیشتر اکید است. پس نشان دادیم هر رشته‌ی اکسپت شده، آرمانی است و همه‌ی رشته‌های آرمانی نیز اکسپت می‌شوند.

مسئله‌ی ۳. Recursive Descent

پاسخ.

این گرامر دارای مشکل چپ گردی و در ناپایانه T است و به همین علت تابع پارس این ناپایانه در یک لوپ نامتناهی می‌افتد. به همین منظور این گرامر را بصورت زیر تغییر می‌دهیم.

$$E \to T + E|T - E'|T$$

$$E' \to T - E'|T$$

$$T \to \epsilon|۱۰T|۱۱T$$

pseudo-code

```
parseE()
        parseT()
        token = nextToken()
        if token = '+':
                parseE()
        else if token == '-':
                parseEp()
        else
                success!


parseEp()
        parseT()
        token = nextToken()
        if token = '-':
                parseEp()
        else if token = 'null'
                return
        else
                error!

parseT()
        token = nextToken()
        if token != '1'
                error!
        token = nextToken()
        if token not in {'0', '1'}
                error!
        token = nextToken()
        if token = 'null'
                return
        else
                parseT()
```

مسئله‌ی ۴. پرانتزگزاری معتبر

پاسخ.

۱. ابتدا مجموعه‌های first و follow را برای متغیرهای گرامر محاسبه می‌کنیم:

$$first(s) = \{\epsilon, (\}$$
$$first(p) = \{(\}$$
$$follow(s) = \{\$,)\}$$
$$follow(p) = \{(, \$\}$$

این گرامر یک گرامر LL(۱) است و دو مجموعه‌ی follow(s) و first(p) با هم اشتراک ندارند و بنابراین در استفاده از قاعده‌ی ۱ و ۲ به مشکل برنمی‌خوریم. برای متغیر P هم که تنها یک قاعده داریم.

	(	)	$
S	۱	۲	۲
P	۳	Err	Err

جدول ۱: $Parse\ Table$

۲. اولین نوع خطا هنگام پارس، مربوط به زمانی است که به استیتی برسیم که خانه‌ی مربوط به آن استیت و آن lookahead در جدول پارس تعریف نشده باشد و برای آن حالت قاعده‌ای برای استفاده وجود نداشته باشد (خانه‌های error در جدول پارس بالا). برای مثال در این گرامر اگر سر استک برابر P باشد و lookahead ما "$" یا ")" باشد هیچ کدام از قواعد قابل استفاده نیستند و خطا رخ می‌دهد.
البته در گرامر ما چنین حالتی رخ نخواهد داد. فرض کنید یکی از این حالات رخ دهد. در این صورت سر استک ما P خواهد بود و lookahead باید یا")" باشد یا "$". این در حالی است که تنها زمانی P به استک اضافه می‌شود که از قاعده‌ی ۱ استفاده کنیم. تنها زمانی از قاعده‌ی ۱ استفاده می‌کنیم که سر استک S بوده و lookahead برابر "(" باشد و وقتی از این قاعده استفاده می‌کنیم متغیر P در سر استک قرار خواهد گرفت و lookahead همچنان "("خواهد بود در نتیجه هرگاه در مراحل پارس سر استک ما P باشد حتما lookahead برابر "("خواهد بود و بنابراین چنین خطایی در مراحل پارس این گرامر رخ نمی‌دهد.

دومین حالت خطا برای زمانی است که پارس استک ما به انتها برسد اما همچنان رشته‌ی ورودی به انتها نرسیده باشد. یعنی درواقع به توکن "$" در پارس استک برسیم اما در رشته‌ی ورودی هنوز به "$" نرسیده باشیم و lookahead توکن دیگری باشد. بنابراین رشته‌ی ورودی به صورت کامل پارس نشده است و خطا رخ می‌دهد. برای مثال رشته‌ی ورودی ")" را درنظر بگیرید. مراحل پارس این رشته به این شکل است:

$$stack = [S, \$], lookahead = ")" \rightarrow stack = [\$], lookahead = ")" \rightarrow Error$$

در ابتدا طبق جدول پارس از قاعده‌ی شماره ۲ استفاده می‌کنیم و S را به ϵ تبدیل می‌کنیم. سپس سر استک توکن "$" خواهد بود درحالیکه lookahead برابر ")" است و خطا رخ می‌دهد.

مسئله‌ی ۵. با طعم اپسیلون

پاسخ.

۱. این گرامر دارای ابهام است. برای مثال رشته‌ی abd را درنظر بگیرید. به دو روش می‌توان به آن رسید:

$A \rightarrow abCd \rightarrow abd$
$A \rightarrow aBC \rightarrow abDC \rightarrow abdC \rightarrow abd$

بنابراین باید آن را تغییر بدهیم تا ابهام نداشته باشد. این تغییرات را در گرامر ایجاد می‌کنیم:
به جای قاعده‌ی $A \rightarrow abCd$ قاعده‌ی $A \rightarrow abcd$ را قرار می‌دهیم. با این تغییر تنها رشته‌ای که دیگر با استفاده از این قاعده نمی‌توان ساخت abd است چرا که یا C به c می‌رود و یا اپسیلون و حال جای C ما c را قرار دادیم پس تنها تفاوت زمانی است که $C \rightarrow \epsilon$ که در این حالت تنها رشته‌ی abd تولید می‌شود. اما دیدیم بدون استفاده از این قاعده نیز می‌توان رشته‌ی abd را تولید کرد پس با این تغییر مشکلی ایجاد نمی‌شود.

$A \rightarrow aBC|abcd$
$B \rightarrow bC|bD$
$C \rightarrow c|\epsilon$
$D \rightarrow d|DC$

تغییر بعدی این است که حرف b را از ابتدای دو قاعده‌ی $B \rightarrow bC$ و $B \rightarrow bD$ حذف کرده و به کنار A اضافه کنیم. یعنی قاعده‌ی $B \rightarrow bC$ را به $B \rightarrow C$ تبدیل کنیم و $B \rightarrow bD$ را به $B \rightarrow D$ تبدیل کنیم و همچنین $A \rightarrow aBC$ را به $A \rightarrow abBC$ تبدیل می‌کنیم. با این کار تفاوتی در زبان گرامر ایجاد نمی‌شود چون B در هر دو حالت در ابتدایش b قرار می‌گیرد پس می‌توان b را از B حذف کرد و در سمت راست قواعد به جای B قرار دهیم .bB

$A \rightarrow abBC|abcd$
$B \rightarrow C|D$
$C \rightarrow c|\epsilon$
$D \rightarrow d|DC$

همچنین C را از قاعده‌ی $A \rightarrow abBC$ حذف کرده و به جای آن $B \rightarrow C|D$ را تبدیل می‌کنیم به $B \rightarrow CC|DC$ باز هم تغییری در زبان ایجاد نمی‌شود چون در هر صورت از هر قاعده‌ی B که استفاده کنیم حتما در انتهای آن C آمده و زبانی که می‌سازد تفاوتی با گرامر قبلی ندارد.

$A \rightarrow abB|abcd$
$B \rightarrow CC|DC$
$C \rightarrow c|\epsilon$
$D \rightarrow d|DC$

حال می‌توان قاعده‌ی $B \rightarrow DC$ را به $B \rightarrow D$ تغییر دهیم چرا که در خود production های متغیر D قاعده‌ی $D \rightarrow DC$ را نیز داریم و در نتیجه با D خالی هم می‌توان DC را تولید کرد درنتیجه با این تغییر زبان ثابت می‌ماند.

$A \rightarrow abB|abcd$
$B \rightarrow CC|D$
$C \rightarrow c|\epsilon$
$D \rightarrow d|DC$

حال به جای قاعده‌ی $B \to CC$ این دو قاعده را جایگزین می‌کنیم: $B \to cC|\epsilon$ با این تغییر نیز زبان ثابت می‌ماند چراکه خود CC می‌توانست در این گرامر یا به cc یا c و یا اپسیلون تبدیل شود پس B می‌توانست در این حالت به یکی از این سه حالت برود. حال cC نیز می‌تواند به cc و c تبدیل شود و برای حالتی که CC به اپسیلون تبدیل شود نیز قاعده‌ی $B \to \epsilon$ را داریم پس باز هم زبان همان است و تغییری نمی‌کند.

$A \to abB|abcd$
$B \to cC|\epsilon|D$
$C \to c|\epsilon$
$D \to d|DC$

درنهایت هم قاعده‌ی $D \to DC$ را تبدیل می‌کنیم به $D \to Dc$ کلماتی که با استفاده از این قاعده‌ی $D \to DC$ قابل تولید هستند عبارتند از $dc*$ چرا که با استفاده از این قاعده تعدادی C سمت راست D می‌آید و درنهایت هر C یا به c می‌رود یا اپسیلون و D هم در نهایت به d می‌رود تا دیگر D نداشته باشیم.
اما با تغییری که دادیم هم همچنان می‌توان $dc*$ را تولید کرد چرا که اگر تعداد c ها برابر صفر بود یکبار از قاعده‌ی $D \to d$ استفاده می‌کنیم و در غیر این صورت به تعداد c های سمت راست از قاعده‌ی $D \to Dc$ استفاده کرده و درنهایت D را به d تبدیل می‌کنیم. گرامر نهایی عبارت است از:

$A \to abB|abcd$
$B \to cC|\epsilon|D$
$C \to c|\epsilon$
$D \to d|Dc$

این گرامر مبهم نیست. چراکه اگر رشته‌ی ما abcd بود باید از قاعده‌ی $A \to abcd$ استفاده کرد و درغیراینصورت از $A \to abB$ چرا که با استفاده از $A \to abB$ نمی‌توان این رشته را تولید کرد و آن هم به این دلیل است که با B نمی‌توان cd را تولید کرد و برای تولید c باید از $B \to cC$ استفاده کرد که درنتیجه حرف آخر نمی‌تواند d باشد.
حال اگر از قاعده‌ی اول استفاده کردیم، اگر حرف سوم c باشد باید حتما از $B \to cC$ استفاده کرد و اگر d باشد باید از $B \to D$ استفاده کرد و اگر هم دوحرفی باشد باید از $B \to \epsilon$ استفاده کرد. اگر از $B \to cC$ استفاده کرده بودیم که وقتی به C می‌رسیم اگر حرف بعدی c بود از $C \to c$ باید استفاده کنیم وگرنه از $C \to \epsilon$ استفاده می‌کنیم. اگر هم از $B \to D$ استفاده کردیم که وقتی به D رسیدیم به تعداد c هایی که در سمت راست d آمده است باید از $D \to Dc$ استفاده می‌کنیم و در نهایت هم جای D با استفاده از $D \to d$ حرف d را قرار می‌دهیم. پس گرامر ما مبهم نیست.
حال گرامر بدست آمده LL(۱) نیست. چراکه برای مثال اگر در استیت A باشیم و حرف بعدی a باشد نمی‌دانیم از قاعده‌ی اول استفاده کنیم یا قاعده‌ی دوم. حال از ab فاکتور می‌گیریم. داریم:

$A \to abB$
$B \to cC|cd|\epsilon|D$
$C \to c|\epsilon$
$D \to d|Dc$

حال از c در قواعد B فاکتور می‌گیریم:

$A \to abB$
$B \to cB'|\epsilon|D$
$B' \to d|C$
$C \to c|\epsilon$
$D \to d|Dc$

حال به جای $D \rightarrow d|Dc$ قرار می‌دهیم. گرامر نهایی برابر خواهد بود با :

$A \rightarrow abB$
$B \rightarrow cB'|\epsilon|D$
$B' \rightarrow d|C$
$C \rightarrow c|\epsilon$
$D \rightarrow dD'$
$D' \rightarrow cD'|\epsilon$

باز هم زبان تغییر نمی‌کند چرا که ابتدا d را با $D \rightarrow dD'$ تولید کرده و سپس به تعداد c های سمت راست d از قاعده‌ی $D' \rightarrow cD'$ استفاده کرده و در نهایت D' را به اپسیلون می‌بریم. گرامر حاصل LL(۱) است.

۲. مجموعه‌های first و follow عبارتند از:

$first(A) = \{a\}$
$first(B) = \{d, c, \epsilon\}$
$first(B') = \{d, c, \epsilon\}$
$first(C) = \{c, \epsilon\}$
$first(D) = \{d\}$
$first(D') = \{c, \epsilon\}$
$follow(A) = follow(B) = follow(B') = follow(C) = follow(D) = follow(D') = \{\$\}$

۳. جدول پارس LL(۱) به این شکل است:

	a	b	c	d	$
A	abB	Err	Err	Err	Err
B	Err	Err	cB’	D	ϵ
B’	Err	Err	C	d	C
C	Err	Err	c	Err	ϵ
D	Err	Err	Err	dD’	Err
D’	Err	Err	cD’	Err	ϵ

جدول ۲: $Parse\ Table$

۴. مراحل پارس رشته به این شکل است: این رشته در زبان است و ACCEPT می‌شود.

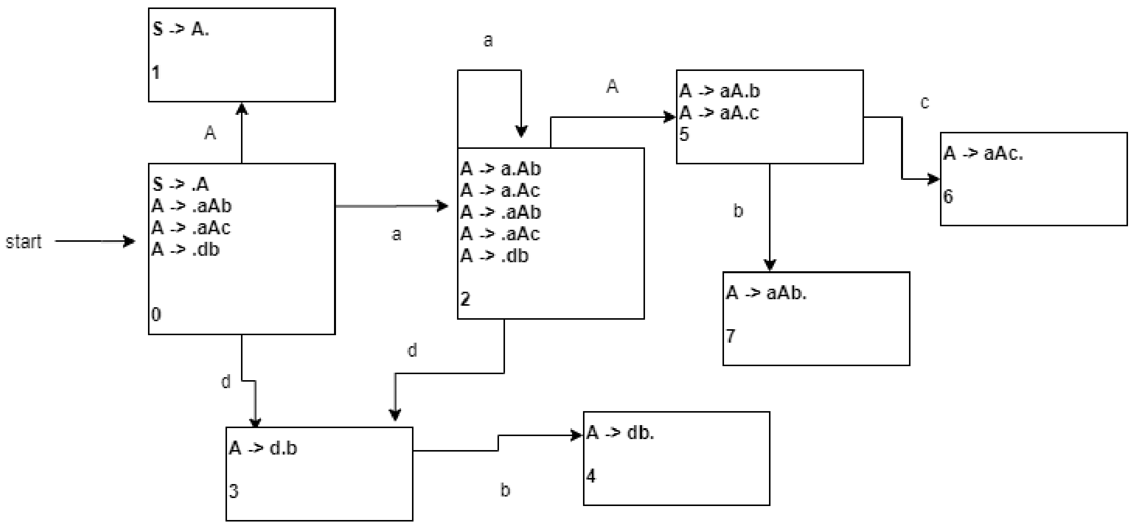
مسئله‌ی ۶. پارسر LR(۰)

۱. دیاگرام:

جدول پارس :

A$	abdcc$
abB$	abdcc$
bB$	bdcc$
B$	dcc$
D$	dcc$
dD'$	dcc$
D'$	cc$
cD'$	cc$
D'$	c$
cD'$	c$
D'$	$
$	$
	ACCEPT

جدول ۳: $Parse\ Table$



۲. به ازای ورودی $aadbc\$$ عملیات parse کردن به شکل زیر خواهد بود :
از علامت | برای جدا کردن قسمت work area استفاده شده است و نقش پوینتر دارد.

$$[stack], string : [۰], |aadbc\$ \rightarrow [۰, ۲], a|adbc\$ \rightarrow [۰, ۲, ۲], aa|dbc\$$$

$$\rightarrow [۰, ۲, ۲, ۳], aad|bc\$ \rightarrow [۰, ۲, ۲, ۳, ۴], aadb|c\$$$

$$[۰, ۲, ۲], aa|Ac\$ \rightarrow [۰, ۲, ۲, ۵], aaA|c\$ \rightarrow [۰, ۲, ۲, ۵, ۶], aaAc|\$ \rightarrow [۰, ۲], a|A\$ \rightarrow [۰, ۲, ۵], aA|\$$$

عملیات parse در استیتی متوقف شد که accept نمی باشد. پس این رشته توسط گرامر ما ساخته نمی شود.

مسئله‌ی ۷. LALR یا !SLR

پاسخ.

var	a	d	b	c	A
۰	S۲	S۳			G۱
۱	AC	AC	AC	AC	AC
۲	S۲	S۳			G۵
۳			S۴		
۴	R۴	R۴	R۴	R۴	R۴
۵			S۷	S۶	
۶	R۳	R۳	R۳	R۳	R۳
۷	R۲	R۲	R۲	R۲	R۲

جدول ۴: Table Parse

اگر در گراف LALR مقادیر Lookahead را نادیده بگیریم، گراف‌های LR(۰) ، SLR(۱) و LALR(۱) یکی خواهند بود.

حال برای گرامر زبان، گراف LALR(۱) را در نظر بگیرید. فرض کنید در یک استیت مانند q ، یک قاعده‌ی تولید مثل $A \to \gamma[lookaheads]$ وجود داشته باشد. می‌دانیم :

$lookaheads = \{t|S \Rightarrow^* \alpha A t \omega\}$ به شرطی که اگر از استیت استارت شروع کنیم، پس از خواندن عبارت $\alpha\gamma$ به استیت q برسیم. پس تمامی مقادیر درون lookaheads می‌توانند پس از تعدادی اشتقاق، پس از A ظاهر شوند. پس

$$lookaheads \subseteq follow(A)$$

حال ثابت می‌کنیم اگر یک استرینگ مانند S در n مرحله توسط SLR(۱) پذیرفته شود، دقیقا توسط n مرحله توسط LALR(۱) پذیرفته می‌شود و اکشن انجام شده در مرحله‌ی i ام توسط SLR(۱) دقیقا همان اکشنی است که در مرحله‌ی i ام توسط LALR(۱) انجام می‌شود. این را به استقرای ریاضی روی i ثابت می‌کنیم.

پایه: در ابتدای کار هر دو در یک استیت هستند.

فرض : تا مرحله‌ی i ام هر دو اکشن های یکسانی انجام داده اند. پس استرینگ‌های حاصل از اکشن‌های هر دو الگوریتم یکی است و در استیت‌های یکسانی نیز قرار داریم.

حکم: فرض در مرحله‌ی $i + ۱$ ام نیز موارد بالا اتفاق می‌افتد.

فرض کنید که تا مرحله‌ی i ام، رشته به شکل $\alpha t \beta$ در آمده است که α همان work area است.

- اگر در مرحله‌ی i ام، الگوریتم SLR(۱) ، اکشن shift یا goto را انجام دهد، الگوریتم LALR(۱) نیز به ترتیب shift یا goto را انجام می‌دهد. زیرا در رشته‌ی $\alpha t \beta$ هیچ handle مناسبی مانند ω یافت نشده است که $t \in follow(A)$ و ضمنا قاعده‌ی $A \to \omega.$ در استیت کنونی وجود داشته باشد. هم‌چنین $\alpha t \beta = \alpha' \omega t \beta$ نیز باید برقرار باشد.

ادعا می‌کنیم در الگوریتم LALR(۱) نیز نمی‌توانیم handle را پیدا کنیم. زیرا برای پیدا کردن handle ،تمامی شرط‌های بالا را نیز داریم به جز شرط $t \in follow(A)$ که به جای آن، شرط

$$t \in lookaheads \subseteq follow(A)$$

را داریم. پس شرط‌های ما برای یافتن handle سخت تر شده است و نمی‌توانیم Reduction انجام دهیم. بدیهتا به دلیل یکسان بودن گراف و استرینگ تا مرحله‌ی کنونی، هر دو الگوریتم یا shift می‌دهند و یا goto را انجام می‌دهند. پس از اکشن یکسان هم هر دو الگوریتم به استیت‌های یکسانی خواهند رفت و استرینگ‌های حاصل از اکشن‌ها نیز یکی خواهد بود. پس حکم در این حالت ثابت شد.

- اگر الگوریتم SLR(۱) عملیات Reduction را انجام دهد، توانسته است که handle مناسبی مانند ω بیابد که $\alpha t\beta = \alpha'\omega t\beta$ و ضمنا قاعده‌ی $A \to \omega.$ در استیت کنونی وجود داشته باشد. هم‌چنین باید $t \in follow(A)$ هم برقرار باشد.

حال اگر ثابت کنیم در LALR(۱) ، t در lookaeads قاعده‌ی متناظر وجود داشته باشد، می‌فهمیم که الگوریتم LALR(۱) هم در این مرحله عملیات Redcution را انجام می‌دهد.

می‌دانیم که رشته‌ی ورودی توسط گرامر پذیرفته می‌شود. پس تعدادی اشتقاق وجود داشته است که:

$$s \Rightarrow^* \alpha'\omega t\beta$$

و ضمنا با طی کردن $\alpha'\omega$ در گراف SLR(۱) که همان گراف LALR(۱) است، به استیت q برسیم. پس می‌فهمیم که $t \in lookaheads$. پس در شروط عملیات Reduction در الگوریتم LALR(۱) نیز وجود دارد. مرحله‌ی $i + ۱$ ام الگوریتم LALR(۱) نیز Reduction است. پس حکم در این حالت هم ثابت شد.

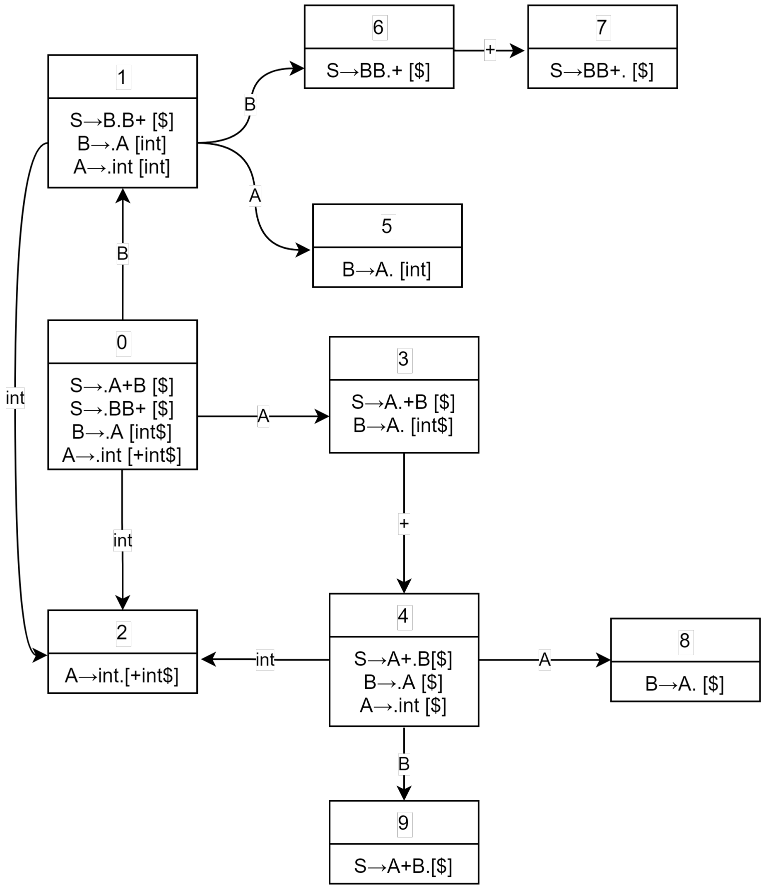
پس اگر الگوریتم SLR(۱) یک رشته را بدون برخورد بپذیرد، الگوریتم LALR(۱) نیز همین‌گونه عمل می‌کند.

حال یک گرامر و یک رشته ارائه می‌دهیم که الگوریتم LALR(۱) رشته‌ی مورد نظر را بدون برخورد parse می‌کند و الگوریتم SLR(۱) به برخورد می‌خورد. گرامر زیر را در نظر بگیرید :

$S \to A + A$
$S \to BB+$
$B \to A$
$A \to int$

و رشته‌ی ورودی نیز برابر است با : $int + int\$$

گراف SLR(۱) به شکل زیر است :



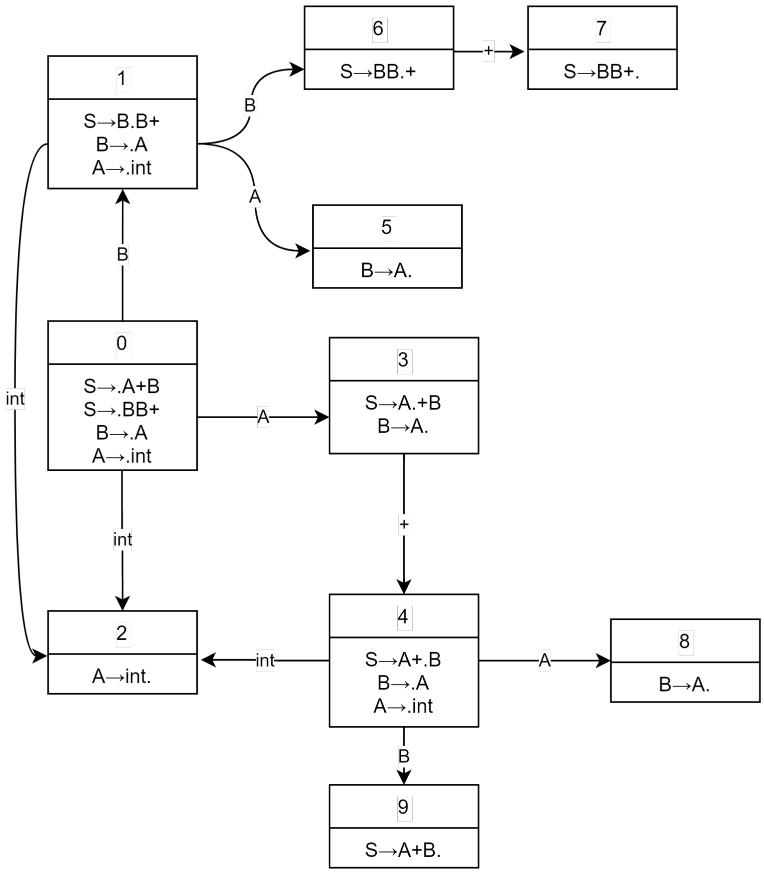
پارسر SLR(۱) به شکل زیر عمل می‌کند(از علامت | برای جدا کردن قسمت work area استفاده شده است و نقش پوینتر دارد):

$$stack, string : [۰], |int{+}int\$ \rightarrow [۰,۲], int|{+}int\$ \rightarrow [۰], |A{+}int\$ \rightarrow [۰,۳], A|{+}int\$ \rightarrow [۰], |B{+}int\$$$

$$\rightarrow [۰,۱], B| + int\$ \Rightarrow Error$$

و علت ارور این است که به هنگامی که به استیت شماره ۱ می‌رسیم، خروجی + نداریم.

اما گراف LALR(۱) به شکل زیر است:



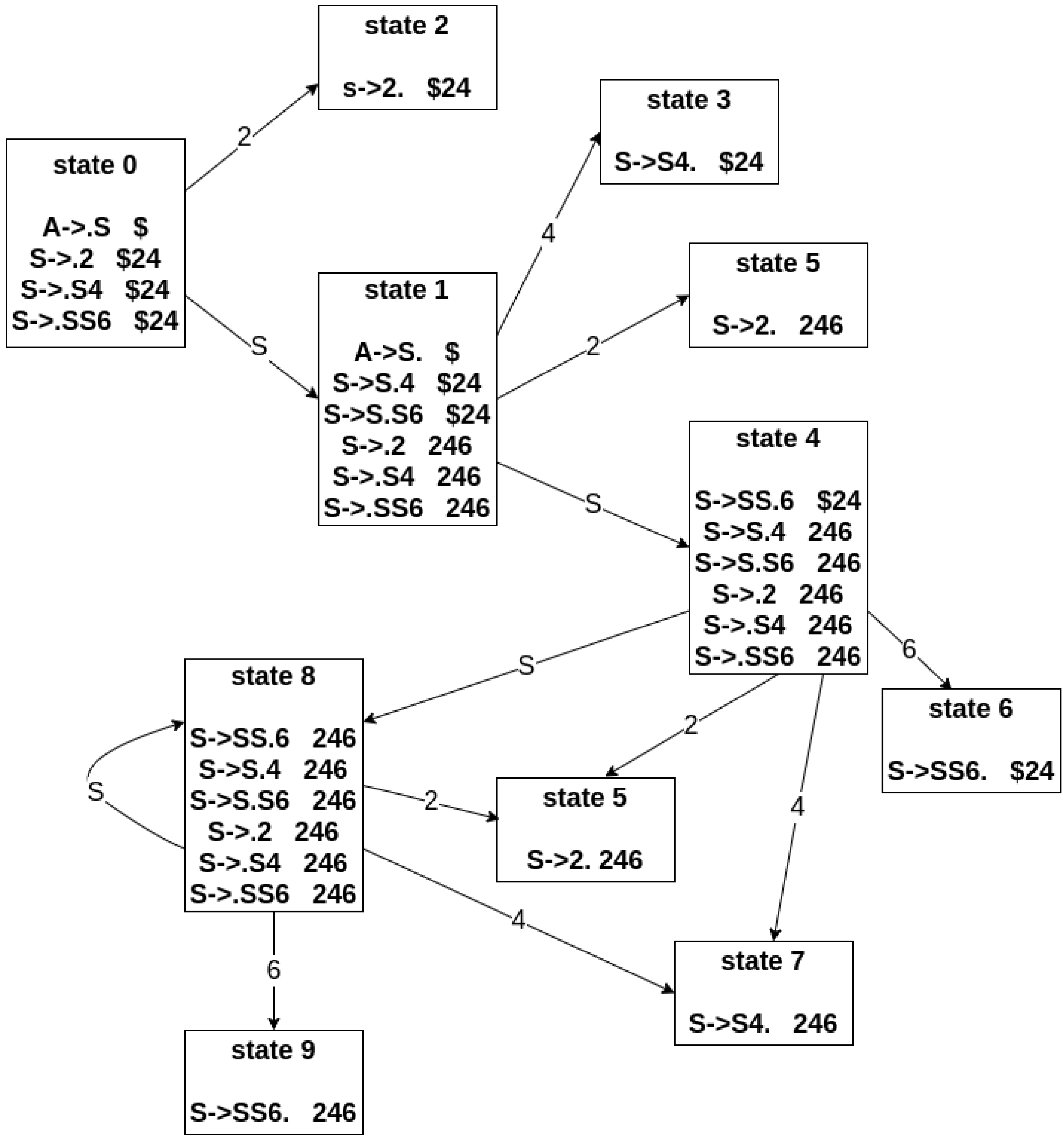
پارسر LALR(۱) به شکل زیر عمل می‌کند:

$stack, string : [۰], |int{+}int\$ \rightarrow [۰,۲], int|{+}int\$ \rightarrow [۰], |A{+}int\$ \rightarrow [۰,۳], A|{+}in\$ \rightarrow [۰,۳,۴], A{+}|int\$$

$\rightarrow [۰,۳,۴,۲], A{+}int|\$ \rightarrow [۰,۳,۴], A{+}|A\$ \Rightarrow [۰,۳,۴,۸], A{+}A|\$ \rightarrow [۰,۳,۴], A{+}|B\$, |S\$ \Rightarrow Accept$

مسئله‌ی ۸. پارسر **LR(1)**

پاسخ.
الف) عکس دیاگرام در زیر قرار داده شده است:



ب) عکس جدول در زیر قرار داده شده است:

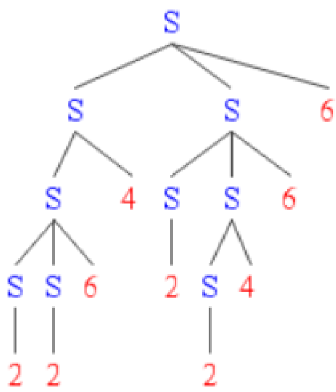
State	Action				Goto	
	2	4	6	$	A	S
0	s2					1
1	s5	s3		accept		4
2	r1	r1		r1		
3	r2	r2		r2		
4	s5	s7	s6			8
5	r1	r1	r1			
6	r3	r3		r3		
7	r2	r2	r2			
8	s5	s7	s9			8
9	r3	r3	r3			

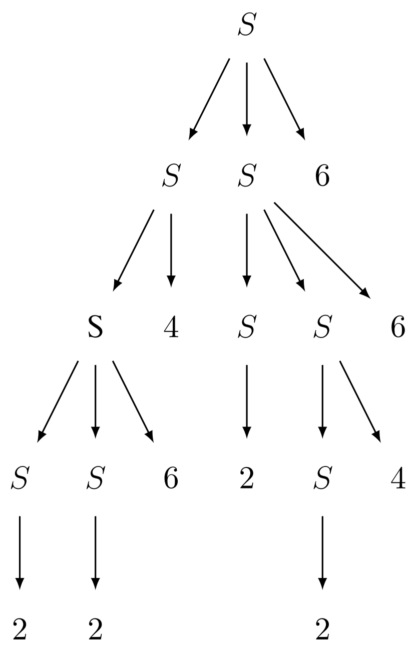
ج) مراحل ایجاد درخت پارس ۲۲۶۴۲۲۴۶۶ را در پایین نوشته‌ایم: (دقت شود که حروفی که بین دو | می‌آیند در مرحله بعدی قرار است با هم، به کمک قاعده تولید reduce شوند)

2
|2|
S
S 2
S |2|
S S
S S 6
|S S 6|
S
S 4
|S 4|
S
S 2
S |2|
S S
S S 2
S S |2|
S S S
S S S 4
S S |S 4|
S S S
S S S 6
S |S S 6|

S S
S S 6
|S S 6|
S

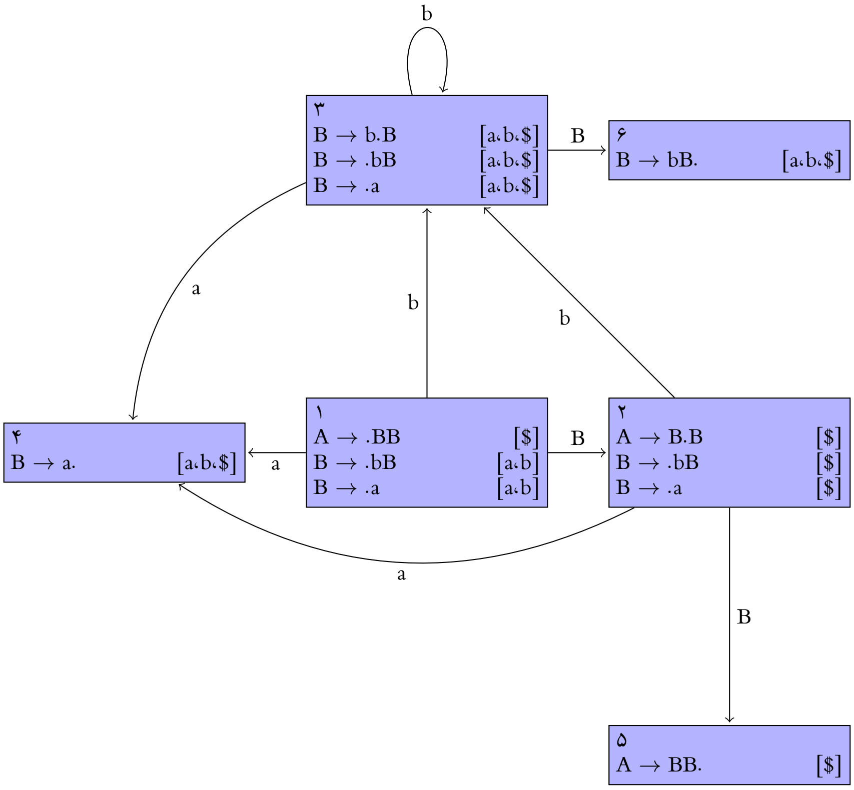
در نهایت درخت پارس رشته به صورت زیر می‌شود: (به دو صورت عکس و کتابخانه خود latex نمایش داده شده است)





مسئله‌ی ۹. پارسر LALR(۱)

پاسخ. دیاگرام پارس LALR گرامر را در زیر مشاهده می‌کنید:



۲. جدول پارس نیز به این صورت است.

$state$	a	b	$\$$	B
۱	۴	۳		۲
۲		۳		۵
۳	۴			۶
۴	R۳	R۳	R۳	
۵			R۱	
۶	R۲	R۲	R۲	

با پیمایش دیاگرام یا جدول پارس رشته بصورت زیر پارس می‌شود.

$|babba \Rightarrow b|abba \Rightarrow ba|bba \Rightarrow bB|bba \Rightarrow B|bba \Rightarrow Bb|ba \Rightarrow Bbb|a \Rightarrow Bbba| \Rightarrow BbbB| \Rightarrow BbB| \Rightarrow BB| \Rightarrow A| \Rightarrow accept$

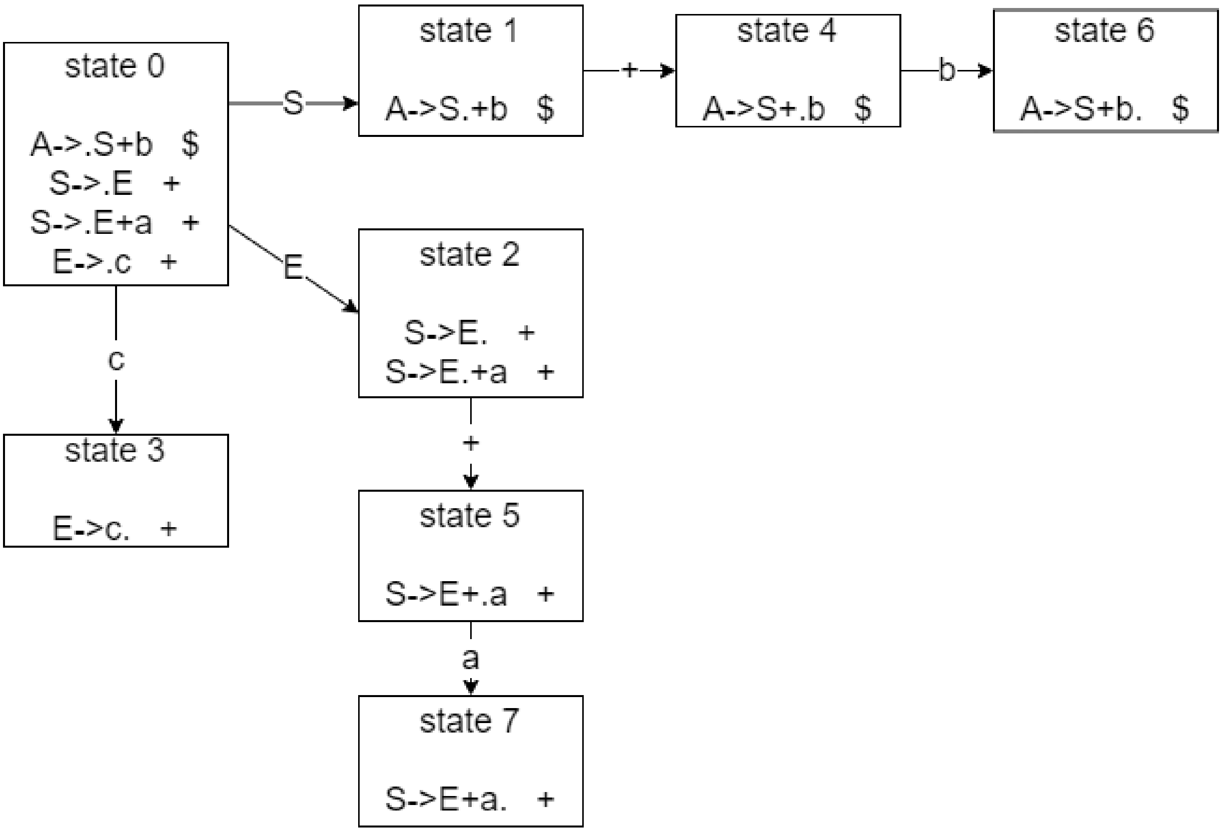
مسئله‌ی ۱۰. مقایسه‌ی **LR(1)** و **SLR(1)**

پاسخ.

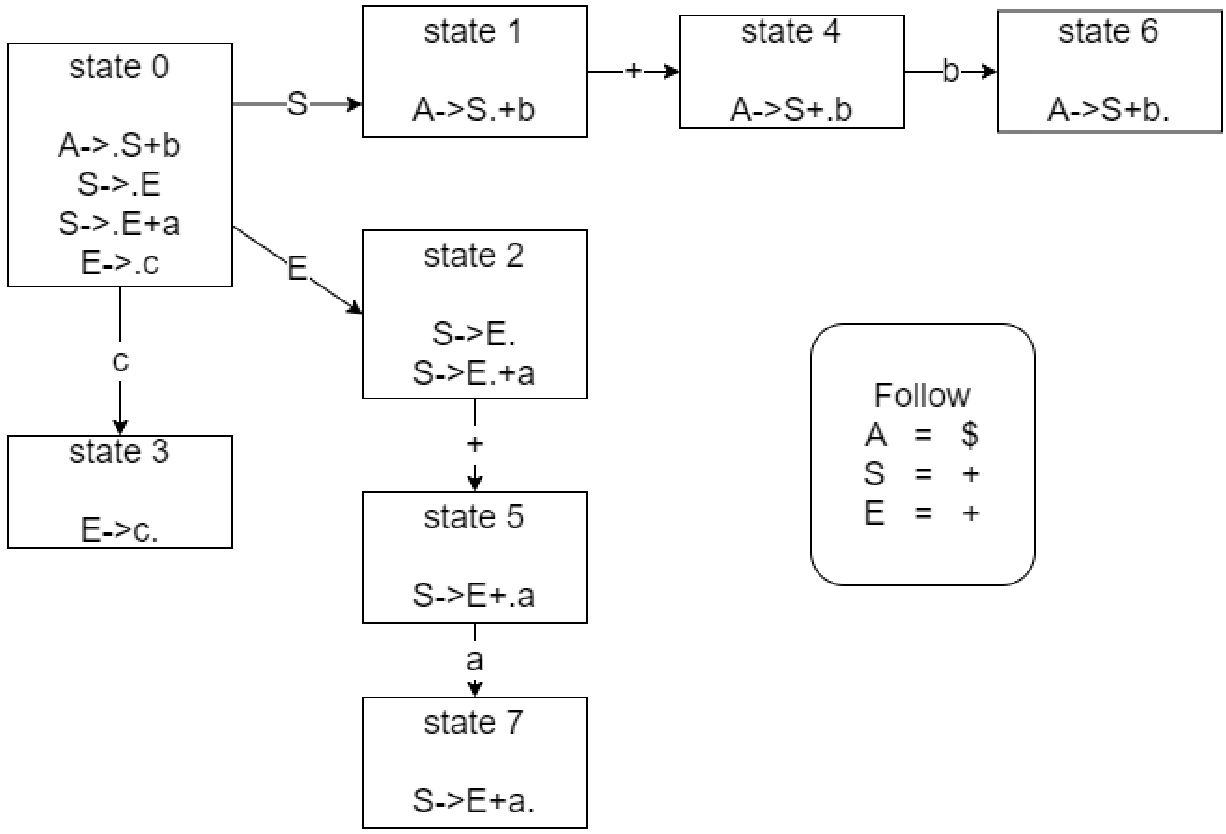
الف) قسمتی از متن غلط است. در SLR(۱) از Follow غیرپایانه ها استفاده میشود و در LR(۱) از LookA-head اما دقت SLR کمتر است، زیرا در SLR فقط به Follow غیرپایانه ها توجه داریم ولی در LR(۱) به فراتر از آن توجه می‌کنیم یعنی LA (Lookahead) ها. LA ها ممکن است در دو جای مختلف برای دو قاعده

تولید یکسان، متفاوت باشند. برعکس Follow ها که برای دو قاعده یکسان، لزوما یکسان اند. بنابراین LR(۱) دقت بیشتری در تمییز دادن حالت های مختلف، نسبت به SLR(۱) دارد. همچنین LR(۱) فضای بیشتری اشغال میکند، چون تعداد زیادی state داریم که یک قاعده تولید یکسان ممکن است در هرکدام با LA های مختلف ظاهر شود. اما در SLR به اندازه LR(۰) فضا اشغال میشود و یک قاعده تولید نمی تواند به اشکال مختلف در state های مختلف ظاهر شود، زیرا Follow غیرپایانه سمت چپ آن به هر حال ثابت است و در نتیجه می‌توان صرفا یک جدول داشت که Follow هر non-terminal در آن ذخیره شده است و با رجوع به آن Follow مورد نظر را بفهمیم.

ب) عکس دیاگرام در SLR(۱) زیر نمایش داده شده است:



عکس دیاگرام LR(۱) در زیر نمایش داده شده است:



گرامر نه SLR(۱) هست و نه LR(۱) .
SLR(۱) به این دلیل نیست که در استیت شماره ۲، با ترمینال + هم می‌توان به استیت شماره ۵ shift کرد و هم می‌توان با قاعده تولید شماره ۲ reduce کرد چون + در follow غیرپایانه S وجود دارد.
LR(۱) هم به دلیل مشابه نیست. در استیت شماره ۲، با ترمینال + هم می‌توان به استیت شماره ۵ shift کرد و هم می‌توان با قاعده تولید شماره ۲ reduce کرد و + در lookahead های این قاعده وجود دارد.