دانشگاه آزاد اسلامی واحد هشترود

کامپایلر ها
اصول، ابزار ها، روش ها

مولف: آلفردوی. آهو – راوی سدی - جفری دی. المن
مدرس: مهندس باقری نیاء

تهیه کننده: سعدی همتی

به نام خدا

جلسه اول

ترجمه:

عبارت است از تبدیل یک ساختار از یک زبان(زبان مبدا) به زبان دیگر(مقصد).

مترجم:

برنامه ای است که عمل ترجمه از زبان مبدا به زبان مقصد را انجام می دهد.

انواع مترجم:

☐ **کامپایلر:** اگر زبان مبدا یکی از زبان های سطح بالا(پاسکال،فرترن PL/1) و زبان مقصد زبان ماشین باشد به چنین مترجمی کامپایلر گویند.

☐ **مفسر:** مفسر نیز یک زبان سطح بالا را دستور به دستور ترجمه کرده و اجرا می کند و زبان ماشین تولید نمی کند.

نکته۱: فرق کامپایلر با مفسر در این است که کامپایلر کد قابل اجرا تولید می کند ولی برای اجرای یک برنامه توسط مفسر هر بار می بایست ترجمه و اجرا صورت گیرد..

نکته۲: حجم مفسر از حجم کامپایلر کمتر است ولی سرعت کامپایلر بیشتر است

نکته۳: از لحاظ مصرف حافظه بهتر است از مفسر استفاده شود، چون مفسر فضای کمتری اشغال می کند و از لحاظ امنیت کامپایلر بهتر است زیرا کل برنامه را یک جا ترجمه می کند..

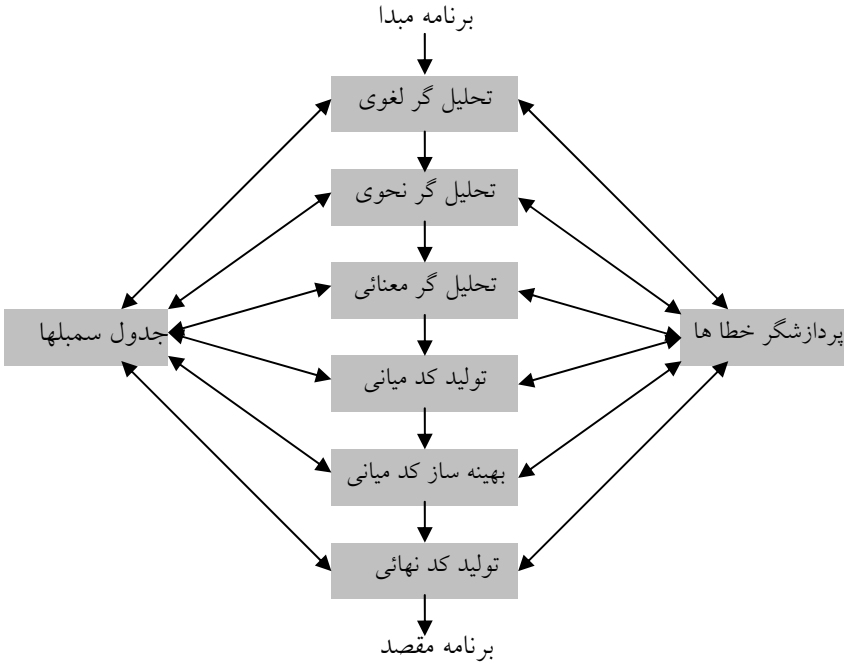
☐ **اسمبلر:** ورودی اسمبلر زبان اسمبلی(کد های یادمان) می باشد که آن را به زبان ماشین تبدیل می کند. اسمبلر های اولیه هر دستور زبان اسمبلی را تبدیل به یک دستور زبان ماشین می کردند که جهت سرعت بخشیدن به عمل ترجمه ، اسمبلر هائی ابداع گردید که قادرند یک دستور اسمبلی(دستور کلان) را تبدیل به چندین دستور زبان ماشین کنند.

☐ **پیش پردازنده:** یک زبان سطح بالا را تبدیل به یک زبان سطح بالای دیگری می کند که اغلب زبان مبدا شکل توسعه یافته زبان است مثل Turbu C، و زبان مقصد شکل استاندارد زبان است مثل ANSI C

مراحل کامپایل در شش مرحله صورت می گیرد.

۱- تحلیل لغوی(Lexical analysis)
۲- تحلیل نحوی(syntax analysis)
۳- تحلیل معنائی(semantic analysis)
۴- تولید کد میانی(intermediate code generation)
۵- بهینه سازی کد(code optimization)
۶- تولید کد نهائی(code generation)

این شش بخش به همراه بخش های "پردازش خطا" و "جدول سمبل ها" یک کامپایلر را تشکیل می دهند ارتباط بین این بخش ها در زیر نشان داده شده است.



تحلیلگر لغوی:

در یک کامپایلر به تحلیل خطی(Linear Analysis) ، تحلیل لغوی(lexical Analysis) ، تحلیل پویش(scaning Analysis) گفته می شود. کاراکتر ها را از چپ به راست خوانده و با رسیدن به یک جدا کننده آنها را در گروه های منطقا به هم مرتبط به نام Token قرار می دهد و Token ها که خروجی تحلیل گر لغوی می باشند در جدول سمبل ها با فرمت خاصی ذخیره می شوند و همچنین به عنوان ورودی تحلیل گر نحوی استفاده می شوند به عبارتی تحلیل گر لغوی واسط بین برنامه مبدا و تحلیلگر نحوی است.

انواع مختلف نشانه ها(Tokenها) عبارتند از

کلمات کلیدی(keywords)

عملگر ها(operators)

ثابت ها(literals)

شناسه ها(identifiers) که به اسامی متغیر ها، توابع، رویه ها و به طور کلی اسامی که کاربر انتخاب می کند گفته می شود.

جداکننده ها(delimiters) { *space*, *tab*, ↵, ;, : }

ساختار Token برای a=b+2

Token type	Token value
id	a

نکته:

تحلیلگر لغوی comment ها(توضیحات) را رد می کند و فاصله های خالی(white space) را که کاراکتر های شناسه ها را که از هم جدا می کند را حذف می کند و اگر خطائی رخ بدهد آن را گزارش می دهد. مثلا با دیدن عبارت " 6a " تحلیل گر لغوی خطائی را گزارش می کند.

تحلیلگر نحوی:

تحلیل سلسله مراتبی(Hierachical analysis) ، تجزیه(parsing analysis) یا نحوی(syntax analysis) نامیده می شود. تحلیلگر نحوی با داشتن گرامر مربوط به زبان و Token های دریافتی از تحلیلگر لغوی برای یک دستور درخت اشتقاق آن را ساخته و اگر دستور مربوط به زبان نباشد خطا گزارش می دهد وگرنه درخت اشتقاق را به عنوان خروجی به تحلیلگر معنائی می دهد.. مثلا در زبان C اگر دستوری به شکل $for(i=0;i++)$ داشته باشیم خطای نحوی رخ میدهد.

تحلیلگر معنائی:

مهمترین وظیفه تحلیلگر معنائی کنترل نوع یا(Type cheking) است. اگر تبدیل نوع مجاز باشد عمل تبدیل نوع را انجام می دهد وگرنه خطای تبدیل نوع را گزارش میدهد

تبدیل نوع $\leftarrow float+int$

خطا $\leftarrow float+char$

تولید کننده کد میانی:

در این بخش برنامه ورودی به یک زبان میانی تبدیل می شود. میتوان این زبان میانی را به برنامه ای برای یک ماشین انتزاعی تشبیه کرد. این زبان میانی حداقل بایستی خواص ذیل را دارا باشد.

سهولت تولید کد : زبان ماشین منطقی تا حد امکان باید ساده در نظر گرفته شود تا تولید کد برای آن آسان باشد.

سهولت ترجمه به زبان مقصد: زبان ماشین منطقی باید به گونه ای باشد که تبدیل آن به زبان ماشین آسان باشد.

بهینه سازی کد میانی:

در این بخش سعی می شود تا کد میانی به صورتی در آید که سریع تر اجرا شود مثلا در کد a=func(b) +func(b) به جای دو بار فراخوانی تابع func() این تابع با پارامتر b یک بار فراخوانی شده و مقدار برگشتی آن در یک متغیر موقت ریخته می شود و سپس از مقدار متغیر موقت استفاده می شود.

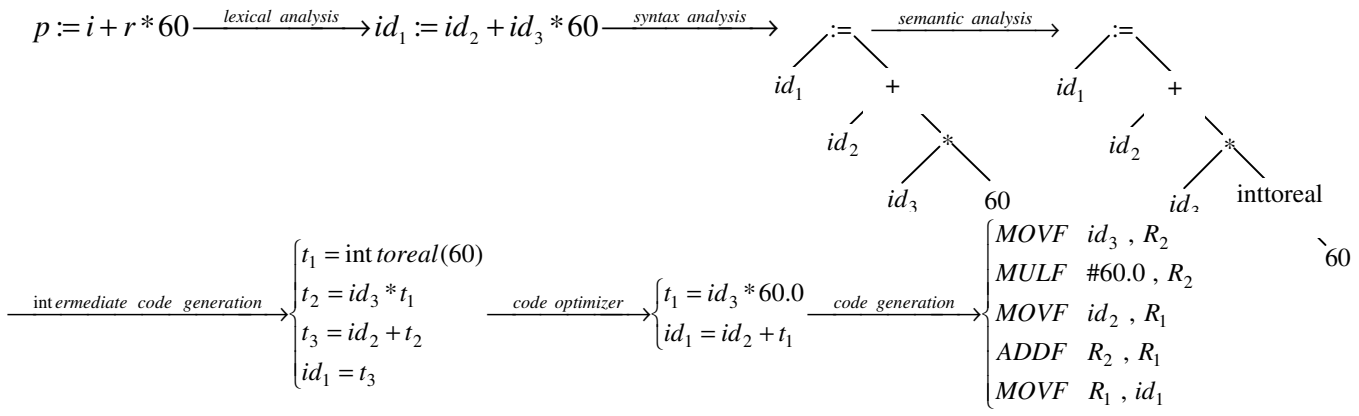
دلیل این که عمل بهینه سازی همراه با کد میانی انجام نمی شود چیست؟

جواب- عمل تولید کد میانی همراه با درخت انجام می شود و کار کردن با درخت مشکل است ولی بهینه ساز کد، یک فایل را به عنوان ورودی می گیرد.

تولید کننده کد نهائی:

در این مرحله کد اسمبلی قابل اجرا بر روی ماشین ایجاد می شود. در واقع کد تهیه شده میانی که در آن از ثبات استفاده نشده می بایست به کدی تبدیل شود که در آن از ثبات استفاده می شود.

به عنوان مثال شکل زیر مراحل کامپایلر p=i+r*60 و تاثیر هر مرحله از کامپایل بر روی این دستور را نشان می دهد(فرض شده است که i,p,r همگی از نوع real هستند)



نکات تکمیلی جلسه اول

- بخش front-end یک کامپایلر برای تمام ماشین ها یکسان است ولی back-end برای ماشین های مختلف متفاوت است
- اگر کاراکتر غیر مجاز در متن برنامه نوشته شود تحلیل لغوی(scanner) آن را تشخیص میدهد(علوم کامپیوتر-۷۹)
- در فرایند کامپایل یک برنامه توکن ها در مرحله آنالیز لغوی(Lexical) شناسائی میشوند.(مهندسی کامپیوتر آزاد-۷۱)
- در مرحله تحلیل لغوی کلیه شناسه های موجود در برنامه وارد جدول علائم(symbol Table)میشود.(مولف-راهیان ارشد)
- back-end بخشی از مرحله بهینه سازی و مرحله تولید کد نهائی که وابسته به ماشین است، می باشد.(مولف- راهیان ارشد)
- تفاوت بین compiler و preprocessor در این است که کامپایلر تحلیل لغوی و معنائی را روی برنامه انجام میدهد، در حالی که preprocessor این کار را انجام نمیدهد(مولف- راهیان ارشد)
- دلیل استفاده ازfront-end و back-end در کامپایلر ها، تدارک ترکیب چند front-end و back-end در یک خانوادهٔ کامپایلر است(مولف-راهیان ارشد)
- در صورتی که شناسه قبلا اعلان نشده باشد، یک خطای معنائی رخ میدهد.(نوپردازان-مولف)
- کشف خطا های مربوط به ساختار تک تک لغات وظیفه تحلیل گر لغوی است(کتاب-پیام نور)
- در دستور $\text{var } 7temp := integer;$ که به زبان پاسکال می باشد، تحلیل گر لغوی لغت $7temp$ را به عنوان خطا گزارش میدهد زیرا در پاسکال یک شناسه نباید با عدد شروع شود.
- در زبان پاسکال عبارت، $A \quad B :=$ به دلیل عدم رعایت ترتیب صحیح انتساب دارای خطای نحوی می باشد.
- در زبان پاسکال عبارت، $if\,(a = b \quad then$ به دلیل عدم توازن پرانتزها دارای خطای نحوی می باشد.
- تحلیل گر معنائی معنی دار بودن عباراتی که از نظر نحوی درست بوده اند را مورد بررسی قرار میدهد.

دو مزیت عمده کد میانی عبارت است از:

- کامپایلر را می توان مستقل از ماشین نوشت لذا با تغییر ماشین ها و تکنولوژی انها کافی است تنها بخش مولد کد تغییر کند و نه دیگر بخش های کامپایلر.
- یک بهینه ساز کد مستقل از ماشین را می توان برای سریع تر کردن کد میانی استفاده کرد، در نتیجه بدون توجه به نوع ماشین، کد تا حد ممکن بهینه خواهد بود.

به طور کلی کد های میانی شامل انواع زیر می باشد :

۱- کد شبه اسمبلی
۲- درخت خلاصه نحوی
۳- جملات سه آدرسه

☐ با یک پیمایش پسوندی به سادگی می توان از درخت خلاصه نحوی کد اسمبلی تولید کرد.

وظایف تحلیل گر نحوی:

۱- بررسی صحت و درستی ترتیب لغات برنامه(بررسی ساختار برنامه مبدا)

۲- بررسی جریان کنترل(Flow of control checks)،مثلا عبارتBreak در زبان c موجب خروج از نزدیکترین حلقه(switch، for، while) می گردد که در صورت نبودن چنین حلقه ای خطا رخ میدهد.

۳- کنترل منحصر به فردبودن، مثلا نام یک Lable می بایست دقیقا یک بار در برنامه ظاهر شود.

۴- چک کردن نام های مرتبط، مثلا در زبان Ada یک حلقه دارای یک نام در ابتدا و انتهای ساختار خود است که می بایست یکسان باشند.

۵- چک کردن ساختار های تودرتو

برخی از مواردی که توسط تحلیل گر معنائی انجام میشود عبارتند از:

بررسی هماهنگی پارامترها : فراخوانی یک روال باید با تعریف روال هماهنگی داشته باشد. مثلا در زبان C تعداد پارمتر های ارسالی، نوع آنها و ترتیب آنها باید در تعریف تابع هماهنگی داشته باشد.

بررسی و کنترل نوع : در این قسمت نوع عملوند های یک عملگر مورد بررسی قرار می گیرد. مثلا در $a[1.5]:=12$ خطای معنائی وجود دارد زیرا اندیس یک آرایه(a) نبایستی یک عد د اعشاری باشد.

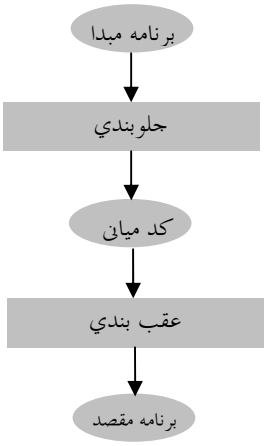
تبدیل نوع : عملیات جمع برای اعداد صحیح و اعشاری متفاوت است برای انجام عمل جمع در عبارت $c = a + b$ در صورتی که c و b از نوع float و a از نوع int باشد، تحلیلگر معنائی عمل ارتقاء نوع را انجام میدهد. بدین معنا که موقتا متغیر a به نوع float تبدیل میشود.

تعریف دوباره متغیر: تحلیلگر معنائی بررسی می کند تا هیچ متغیری دو بار تعریف نشده باشد

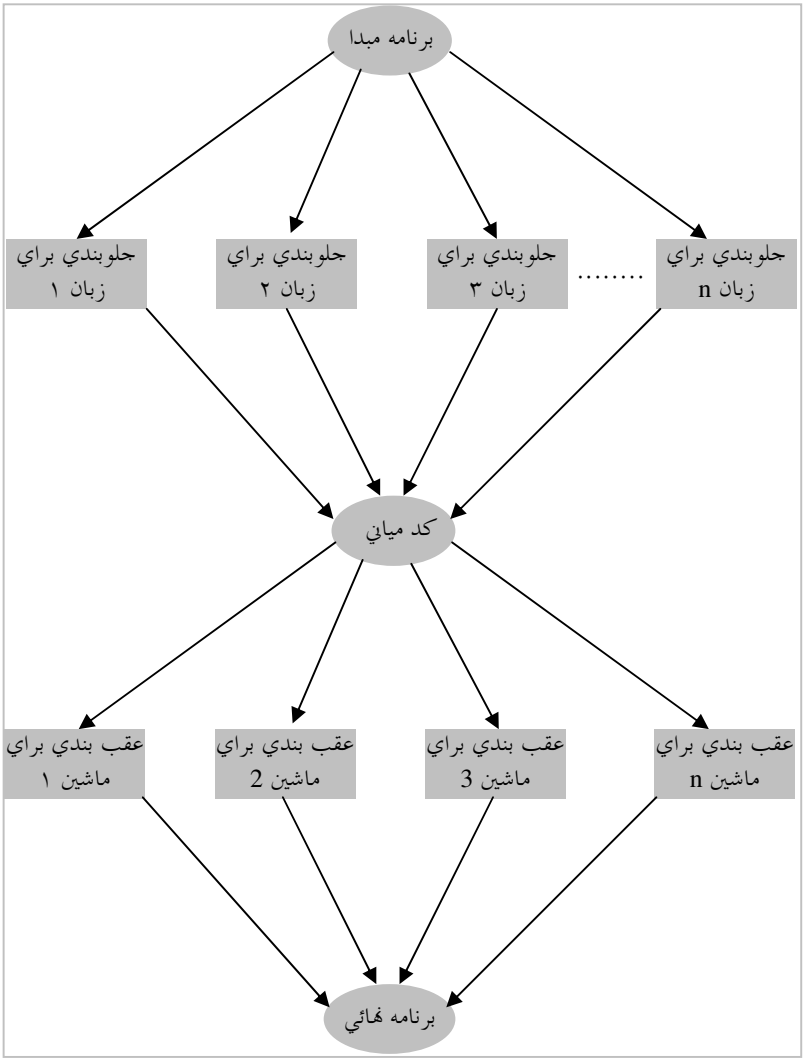
جلسه دوم

جلوبندی (Front End) و عقب بندی(Back End) کامپایلر

می خواهیم برنامه ای را به زبان C روی انواع مختلف کامپیوتر(مانند Vax، Mainfram، IBM) اجرا کنیم در این صورت برای هر نوع کامپیوتر، باید یک کامپایلر جداگانه بسازیم. اگر n تعداد زبان های برنامه سازی(مانند C و پاسکال و...) و K تعداد انواع مختلف کامپیوتر ها باشد در این صورت بهnk کامپایلر نیاز است. ایجاد این تعداد کامپایلر بسیار زمانبر و پر هزینه است. برای حل این مشکل از تقسیم کامپایلر به جلوبندی و عقب بندی استفاده می کنیم. به این شکل که یک زبان میانی در نظر می گیریم در ابتدا برنامه مبدا را به این زبان میانی ترجمه کرده سپس از زبان میانی به زبان مقصد ترجمه می کنیم. **بخشی از کامپایلر که وظیفه ترجمه برنامه مبدا به برنامه زبان میانی را بر عهده دارد و وابسته به زبان ماشین نیست را جلوبندی و بخشی از کامپایلر که وظیفه ترجمه برنامه از زبان میانی را به زبان مقصد را بر عهده دارد و وابسته به زبان ماشین است را عقب بندی کامپایلر می گوئیم(شکل زیر).**



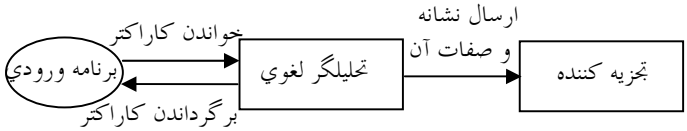
با استفاده از این روش برای n زبان مبدا و k کامپیوتر مختلف به n جلوبندی و k عقب بندی نیاز است که در مجموع $n+k$ برنامه(کامپایلر) احتیاج است(شکل زیر این موضوع را نشان می دهد).



تقسیم بندی کامپایلر به عقب بندی و جلو بندی چه مزایائی دارد؟

- سادگی طراحی
- استقلال جلوبندی از زبان مقصد
- استقلال عقب بندگی از زبان مبدا
- کاهش پیچیدگی
- افزایش قابلیت استفاده مجدد
- افزایش سرعت تولید کامپایلر برای سخت افزار جدید و زبان های جدید.

تحلیگر لغوی(scanner) :واسط بین کامپایلر و برنامه مبدا می باشد و مهمترین وظیفه آن خواندن برنامه ورودی به صورت کـاراکتر بـه کاراکتر و تشخیص نشانه ها یا **token** ها میباشد(شکل زیر نحوه قرار گرفتن تحلیلگر لغوی بین ورودی و تجزیه کننده را نشان می دهد).



Scanner میتو اند به صورت هم روال یا زیرروال با **parser** پیاده سازی شود، **Scanner** به محض درخواست **token** بعدی از **parser** ،توکن بعدی را به پارسر می دهد.

وظایف دیگر scanner

- حذف فضا های خالی و توضیحات(comment)
- گزارش خطا ها

از آنجا که **scaner** برنامه را به صورت کاراکتر به کاراکتر می خواند، می تواند تعداد کاراکتر های هر خط و به تبع آن شماره خطوطی را که توکن ها در آن قرار دارد را نیز مشخص کند. بنابر این هنگام اعلام خطا شماره خطی را که خطا رخ داده است را نیز گزارش می کند.

به چه دلایلی بهتر است که Scanner **و**parser **به صورت مجزا پیاه سازی شوند؟.**

- سادگی
- کارائی بالاتر
- قابلیت حمل

سادگی:

پارسری که دربردارنده قواعد مربوط به حذف فضا های خالی و توضیحات می باشد پیچیده تر از پارسری است که فرض می کند این اعمال را **scanner** انجام می دهد.

کارائی بالاتر:

خواندن کاراکتر به کاراکتر یک عمل ورودی است قرار دادن این اعمال در**Scanner** و در نظر گرفتن تمهیداتی برای بالا بردن سرعت در **scanner** سبب افزایش کارائی می شود.

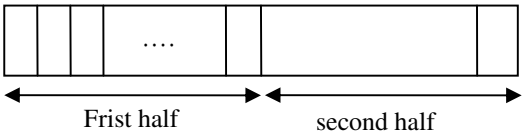
قابلیت حمل:

اعمال قرارداد های مربوط به کاراکتر های ویژه(مثلا علامت ^ در پاسکال) در **scanner** و همین طور اعمال مدیریت ورودی وابسته به هر زبان در **scanner** قابلیت حمل تحلیلگر را بالا می برد.

□ وقت گیرترین کار کامپایلر همان کاری است که **scanner** انجام می دهد پس باید سرعت را بالا ببریم، باید از بافر استفاده شود چون بافر باعث بالا رفتن سرعت می شود.

افزایش سرعت:

تحلیلگر لغوی یا **scanner** تنها فازی از کامپایلر است که عمل ورودی را انجام می دهد بنابراین به دلیل این که به ازای هر کاراکتر می بایست یک عمل ورودی انجام داد استفاده از روشی جهت افزایش سرعت مفید می باشد این روش استفاده از بافر می باشد.



هر یک ار انواع بافر ها دارای دو نوع اشاره گر مکی باشند

Forward : با خواندن هر کاراکتر یک واحد جلو می رود .

Lexeme_beginning : ابتدای توکنی را که می خواهیم مشخص کنیم نشان می دهد

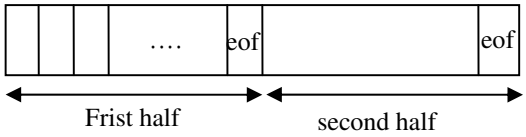
کد مربوط به پیشروی اشاره گر پیش رو(forward) به شکل زیر خواهد بود.

```
if  forward at the end of  frist  half  then  begine
reload  second  half;
forward = forward +1
end
else  if  forward at the end of  second  half  then  begine
reload  frist  half;
move  forward  to  beginning  of  frist  half
end
else  forward = forward +1
```

به ازای خواندن هر کاراکتر به دو تا مقایسه نیاز داریم می خواهیم تعداد مقایسه ها را کمتر کنیم(یک مقایسه) راه حل آن است که از نگهبان استفاده کنیم

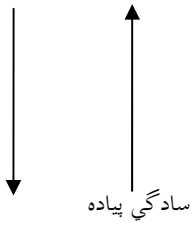
کاراکتر نگهبان:eof

کاراکتر نگهبان انتهای هر نیمه بافر قرار می گیرد و کاراکتر ویژه ای است که جزء متن نیست.



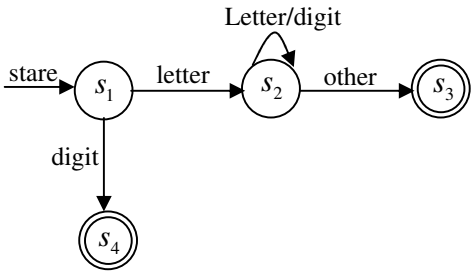
کد مربوط به پیشروی اشاره گر پیش رو(forward) همراه با کاراکتر نگهبان به شکل زیر خواهد بود.

```
forward = forward +1;
if  forward ↑= eof  then  begine
if  forward at the end of  frist  half  then  begine
reload  second  half;
forward = forward +1
end
else  if  forward at the end of  second  half  then  begine
reload  frist  half;
move  forward  to  beginning  of  frist  half
end
else/*eof  within  a  buffer  signifiying  end  of  input8/
end
```



۲- پیاده سازی به کمک زبان برنامه سازی سیستم و استفاده از امکانات مدیریت ورودی در آن زبان
۳- پیاده سازی به کمک زبان اسمبلی و مدیریت مستقیم ورودی

☐به ابزار هائی که خودشان کامپایلر تولید می کنند کامپایلر کامپایلرها می گویند مثلlex در سیستم عامل unix
☐ الگو های توکن ها را به وسیله عبارت منظم نمایش می دهند و هر عبارت منظمی قابل تبدیل به dfa است.
شکل زیر نمودار تغییر حالت برای شناسه ها را نشان می دهد، قاعده تشخیص شناسه عبارت است از دنباله ای از حروف و ارقام که اولین نماد آن حرف باشد .



حال table ها را مشخص می کنیم.
جدول char-class

class	letter	letter	digit	other
char	a-z	A-Z	0-9	other

جدول state

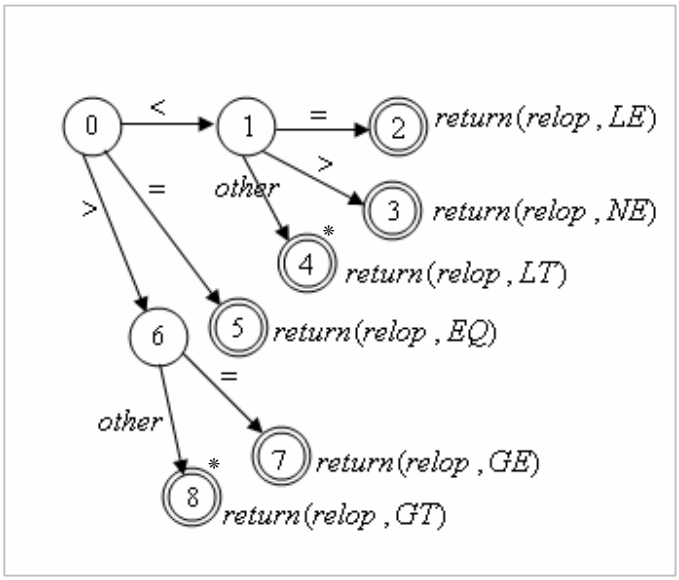
state	letter	digit	other
s_1	s_2	s_4	s_4
s_2	s_2	s_2	s_3
s_3	–	–	–
s_4	–	–	–

هر token از دو مولفه تشکیل می شود یکی نوع token و دیگری مقدار token
بعد از یافتن هر توکن می بایست توکن در جدول نماد ها ذخیره شود.
در جدول نماد ها نوع و نام و شماره اولین سطر ذخیره می شود.

< *token type* , *token value* >

جلسه سوم

دیاگرام تشخیص عملگر های رابطه ای



عملگر های رابطه ای =>, > , <> , < , <= , = , >

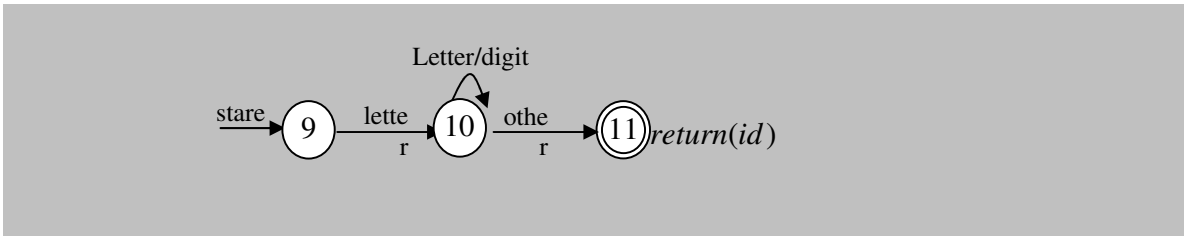
L : $lese$

E : $Equal$

G : $grather$

کاراکتر خوانده شده بایستی به بافر برگردانده شود.

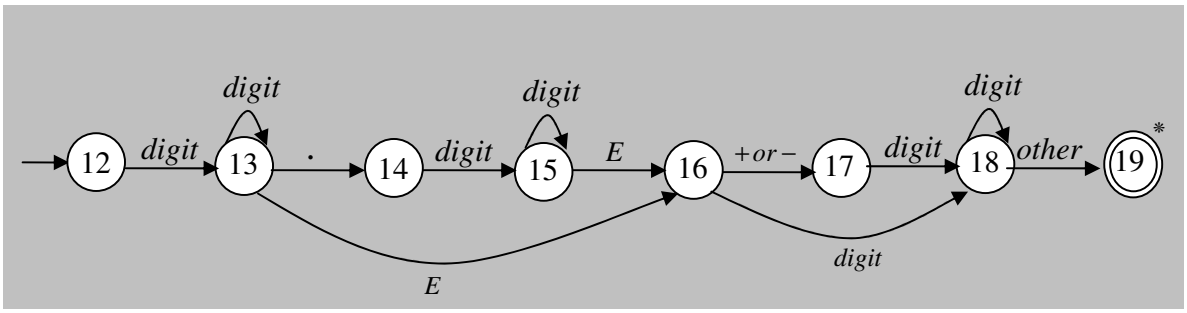
دیاگرام تشخیص شناسه(identifire).



دیاگرام تشخیص اعداد

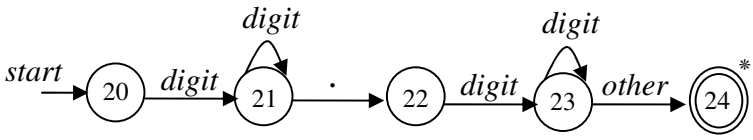
تشخیص اعداد به صورت نماد علمی

$digit\,.\,digit\ \ E\ (+/-)\ \ digit$	$12.34E+1$
$digit\,.\,digit\ \ E\ \ digit$	$12.34\,E1$
$digit\ \ E\ (+/-)\ \ digit$	$12E+1$
$digit\ \ E\ \ digit$	$12E1$

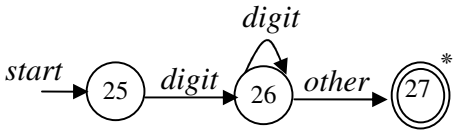


تشخیص اعداد به صورت اعشاری

digit.digi



تشخیص اعداد به صورت صحیح



برنامه به زبان c برای پیدانمودن حالت شروع بعدی

```
int state=0 , start=0;
int  fail()
{
switch (state) {
case 0 :      start = 9:break;
case 9 :      start =12:break;
case 12 :      start = 20:break;
case 20 :      start = 25:break;
case 25 :      recover():break;
default :      /* compiler  error /*
}
return  start;
}
```

Default

رشته خوانده شده با الگوی هیچ کدام از توکن ها مطابقت نمی کند در این صورت خطا گزارش می شود ولی کامپایلر نباید متوقف شود بلکه اسکن بقیه رشته ها را ادامه می دهد.

برنامه به زبان c برای تحلیگر لغوی در صفحه بعد

توضیحات: تابع**install()** به جدول سمبل ها نـگاه می کند اگر **Token** خوانده شده کلمه کلیدی باشد صفر را بر می گرداند اگر کلمه کلیدی نباشد و شناسه باشد دو حالت داریم اگر این شناسه در جدول سمبلها موجود باشد اشاره گر به مدخلی که ایـن توکن در آن قرار دارد برگردانده میشود و اگر در جدول توکن ها موجود نباشد در جدول سمبل ها قرار داده میشود و اشاره گر به مدخلی که قرار داده میشود بر گردانده می شود(شکل زیر جدول سمبل).

جدول سمبل ها

If	k
wa	id
a2	id

☐ اگر برای تشخیص کلمات کلیدی از دیاگـرام انتقال حالت استفاده شود، تعداد حالات زیاد شده و از این رو **state** گردانی مشکل خواهد بود و سرعت پایین میاید ، پس بهتر است کلمات کلیدی در آغاز کار در جدول سمبلها یا جدول نشانه ها قرار داده شوند

☐ نام تابع شناسه است

☐ تابع **gettoken()** اگر توکن خوانده شده کلمه کلیدی باشد نشانه متناظر آن برگردانده میشود و اگر شناسه باشد **id** را برمیگرداند.

☐ 9 , 10 , 11 برای تشخیص کلمات کلیدی است.

☐ بهتر است که توکن هائی که بیشتر تکرار می شوند، در **state** های آغازین قرار گیرند، یعنی در**case** های اول که باعث می شود سرعت بالا رود.

```
while(1){
  switch(state){
case 0: c = nextchar(); if(isspace(c)){lexeme_beginnin++, state = 0;}
      else if(c =='<')   state = 1;
      else if(c =='=')   state = 5;
      else if(c =='>')   state = 6;
      else state = fail();
      break;
case 1  c = nextchar();
      if(c =='=')   state = 2;
      else if(c =='>')   state = 3;
      else state = 4;
      break;
case 2 :  tokentype = relop; tokenvalue = LE; break;
case 3 :  tokentype = relop; tokenvalue = NE; break;
case 4 :  ungetch(c); tokentype = relop; tokenvalue = L; break;
⋮
case 9 : c = nextchar();
      if(isletter(c))    state = 10;
      else   state = fail();
      break;
case 10: c = nextchar();
      if(isletter(c) || isdigit(c))   state = 10;
      else   state = 11;   break;
case 11: ungetch(c); install(token value);
      return(gettoken(token value));
      break;
case 12: c = nextchar();
      if(isdigit(c))   state = 13;
      else   state = fail();
      break;
case 13: c = nextchar();
      if(isdigit(c))   state = 13;
      else if(c ='.')   state = 14;
      else if(c ='E')   state = 16;
      else   state = fail();
      break;
case 14: c = nextchar();
      if(isdigit(c))   state = 15
      else   state = fail();
      break;
case 15: c = nextchar();
      if(isdigit(c))   state = 15
      else if(c ='E')   state = 16;
      else   state = fail();
      break;
```

ادامه برنامه قبلی

```
cace 16: c = nextchar();
         if (isdigit(c))    state = 18;
          else if (c ='+'|| c ='–')   state = 17;
         else   state = fail();
        break;
cace 17: c = nextchar();
         if (isdigit(c))    state = 18;
         else   state = fail();
        break;
cace 18: c = nextchar();
         if (isdigit(c))    state = 18;
        state = 19
        break;
cace 19: ungetch(c); tokentype ='num'; tokenvalue =     ;
         if (isdigit(c))    state = 18;
        break;
cace 20: c = nextchar();
         if (isdigit(c))    state = 21;
         else   state = fail();
        break;
cace 21: c = nextchar();
         if (isdigit(c))    state = 21;
         else if (c ='.')   state = 22;
         else   state = fail();
        break;
cace 22: c = nextchar();
         if (isdigit(c))    state = 23;
         else   state = fail();
        break;
cace 23: c = nextchar();
         if (isdigit(c))    state = 23;
         else   state = 24;
        break;
cace 24: c = ungetch(c); tokentype ='num'; tokenvalue =     ;
cace 25: c = nextchar();
         if (isdigit(c))    state = 26;
         else   state = fail();
        break;
cace 26: c = nextchar();
         if (isdigit(c))    state = 26;
         else   state = 27;
        break;
cace 27: ungetch(c); tokentype ='num'; tokenvalue =     ;
}
}
```

جلسه چهارم

تشخیص خطا

Scanner خطا های زیادی را نمی تواند تشخیص دهد، زیرا دیدگاه محلی نسبت به برنامه دارد به عبارتی برنامه مبدا را به صورت کاراکتر به کاراکتر می خواند به عنوان مثال در عبارت زیری **Scanner** نمی تواند تشخیص دهد که **if** به صورت غلط نوشته شده است(**fi**) و آن را به عنوان یک شناسه معتبر به **parser** تحویل می دهد و کنترل این قبیل خطا ها را به **parser** و مولفه های بعد آن محول می سازد. $fi(x == f(x))$

به عنوان شناسه معتبر مي شناسد

☐ اگر رشته ای از کاراکتر ها با الگوی هیچ کدام از توکن ها مطابقت نداشته باشد در این صورت خطای لغوی رخ می دهد و**Scanner** می بایست این قابلیت را داشته باشد که از این خطا و خطا های دیگر گذر کرده و عمل **Scan** (تحلیل لغوی) را تا پایان فایل ادامه دهد.

☐ در روش **panic mode** اسکنر از رشته ورودی حذف می کند تا جائی که ادامه کاراکتر ها با الگوی یکی از **token** ها تطابقت داشته باشد.

مکانیزم های گذر از خطا

- **panic mode**
- اضافه کردن یک کاراکتر جدید مثل =:) که دو نقطه را اضافه کردیم
- تعویض یا جایگزینی دو کاراکتر مجاور مثل >< → <>
- تغییر یک کاراکتر

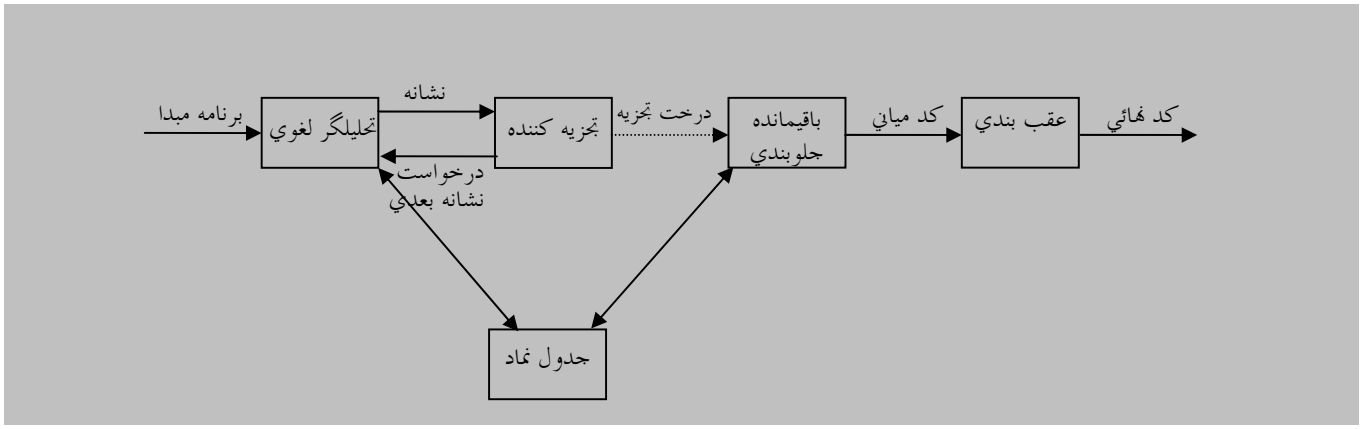
تحلیل نحوی

ساختار هر زبان با قواعد آن مشخص می شوند ساختار زبان های برنامه سازی با گرامر های مستقل از متن پیاده سازی میشود.

$G(S\ ,\ NT\ ,\ T\ ,\ P)$

$S : start\ symbol$
$NT : None\ Terminal\ Set$
$T : Terminal\ Set$
$P : Production\ Rule$

شکل زیر موقعیت تجزیه کننده در مدل کامپایلر را نشان می دهد.



- وظیفه اسکنر تشخیص توکن ها می باشد.
- وظیفه اصلی **parser** تشخیص این که رشته توکن های تولید شده توسط **scanner** ، آیا توسط قواعد زبان قابل تولید هست یا نه

انواع پارسر ها

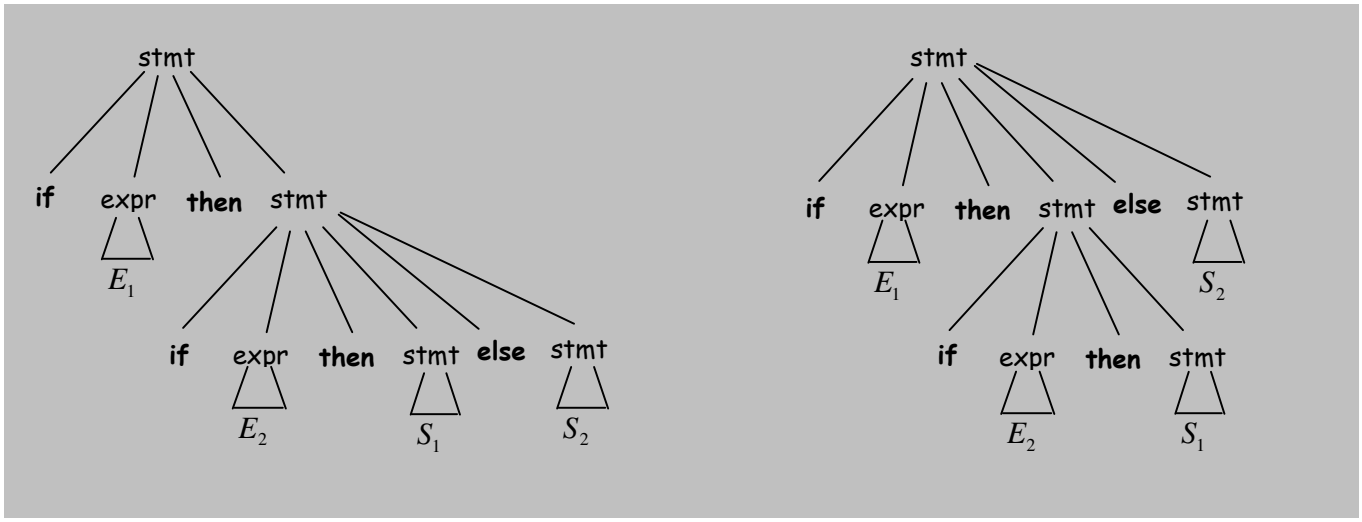
- پارسرهای غیر پیشگو کننده، باید قابلیت **back tracking** داشته باشد.
- پارسرهای پیشگو کننده ۱- پارسر های بالا به پایین ۲- پارسر های پایین به بالا

- در غیر پیشگو کننده با گرفتن رشته ای از توکن ها با سعی و خطا سعی می کنند که تشخیص دهند آیا این رشته قابل تولید توسط گرامر زبان هست یا نه که این پارسر ها باید خاصیت **back tracking** داشته باشند
- در پارسرهای بالا به پایین از جمله **start symbol** شروع کرده و با اشتقاق سعی در تولید جمله مورد نظر را دارند به عبارتی جمله مورد نظر برگ های درخت اشتقاق می باشد.
- در پارسر های پایین به بالا از جمله ورودی(رشته توکن ها) شروع کرده و سعی در رسیدن به **start symbol** را دارند به عبارتی به جای عبارت سمت چپ، سمت راست را می گذارند.
- در پیشگو کننده گرامر ها بایستی فاقد ابهام باشند.

ابهام:

گرامری دارای ابهام است اگر بتوان برای حداقل یک جمله از زبان تولید شده توسط آن گرامر دو درخت اشتقاق چپ یا دو درخت اشتقاق راست مختلف پیدا کرد به عنوان مثال. گرامر زیر را در نظر بگیرید ملاحظه می شود که برای تولید جمله **If** E_1 **then if** E_2 **then** S_1 **else** S_2 دو درخت اشتقاق چپ وجود دارد و گرامر مبهم است. و می بایست رفع ابهام شود.

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | other
```



- اگر زبانی داشته باشیم که نتوانیم گرامر مبهمی برا ی آن پیدا کنیم زبان ذاتا غیر مبهم است.

رفع ابهام گرامر صفحه قبل به شکل زیر خواهد بود

$$stmt \rightarrow m\_stmt \mid unm\_stmt$$
$$m\_stmt \rightarrow if \quad expr \quad then \quad m\_stmt \mid other$$
$$unm\_stmt \rightarrow if \quad expr \quad then \quad m\_stmt \quad else \quad unm\_stmt$$

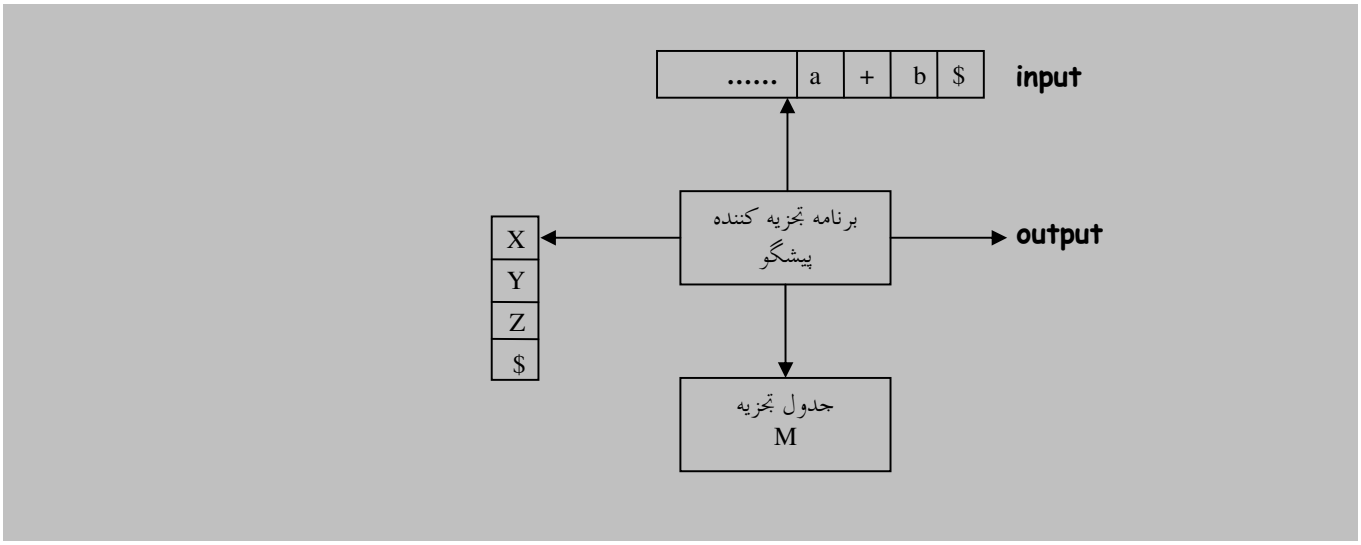
- در اسکنر ورودی کاراکتر است در پارسر ورودی یک توکن است

تجزیه گر پیشگوئی پذیر غیر بازگشتی(بالا به پایین)

$LL(1)$ در این روش "L" اول به این معنی است که رشته ورودی از سمت چپ خوانده میشود و "L" دوم به این معنی است که پارسر از سمت چپ ترین اشتقاقات استفاده می کند.

ورودی رشته ای از توکن هاست که پایان آن را با $ نمایش می دهیم

دارای یک پشته است که در ابتدای پشته $ و **startsymbol** قرار می گیرد، همچنین دارای یک جدول تجزیه می باشد، ساختار کلی این نوع پارسر به فرم زیر است.



مجموعه آغازین $first(\alpha)$

برای سمبل α مجموعه آغازین یا $first$ آن مجموعه ترمینال هائی است که شکل های جمله ای مشتق شده از α با آن شروع می شوند

قواعد ساخت مجموعه آغازین x

- اگر α پایانه باشد $first(\alpha)=\{\alpha\}$
- اگر α بتواند ε را تولید کند آنگاه ε را به مجموعه $first(x)$ اضافه کنید.
- اگر X یک غیر پایانه و $X \rightarrow Y_1Y_2\dots Y_k$ یک مولد باشد. و مجموعه های $first(Y_1), first(Y_2), first(Y_3) \dots first(Y_n)$ شامل ε باشند یعنی همه Y_i ها بتوانند تهی را تولید کنند در نتیجه X نیز می تواند ε را تولید کند که در این صورت ε به مجموعه $first(X)$ اضافه می شود.
- اگر X یک غیر پایانه و $X \rightarrow Y_1Y_2\dots Y_k$ یک مولد باشد. مجموعه $first(Y_1)$ (به جز ε) به مجموعه $first(X)$ اضافه میشود.
- اگر X یک غیر پایانه و $X \rightarrow Y_1Y_2\dots Y_k$ یک مولد باشد. و ε در مجموعه $first(Y_1)$ باشد(Y_1 میتواند ε را تولید کند) در این صورت علاوه بر $first(Y_1)$ (به جز ε) مجموعه $first(Y_2)$ (به جز ε)ا نیز به $first(X)$ اضافه میشود.
- با توجه به قاعده تولید $X \rightarrow Y_1Y_2\dots Y_i\dots Y_k$ اگر ε در $first(Y_1), first(Y_2), \dots first(Y_{i-1})$ باشد مجموعه $first(Y_i)$ (به جز ε) به مجموعه $first(X)$ اضافه میشود.

مثال: گرامر زیر را در نظر بگیرید.

1) $E \to TE'$
2) $E' \to +TE' \mid \varepsilon$
3) $T \to FT'$
4) $T' \to *FT' \mid \varepsilon$
5) $F \to (E) \mid id$

پاسخ

$first(T') = \{ * , \varepsilon \}$
$first(F) = \{ (, id \}$
$first(E') = \{ + , \varepsilon \}$
$first(T) = \{ (, id \}$
$first(E) = \{ (, id \}$

مثال. رشته a را توسط گرامر زیر تولید کنید، آیا a عضو این زبان هست یا نه

$S \to aB \mid b$
$B \to S \mid \varepsilon$

پاسخ. $S \to aB \to a$ و $first(B) = \{\varepsilon, a, b\}$, $first(s) = \{b, a\}$

مثال.در گرامر زیر ()first ها را مشخص کنید.

$S \to BAa \mid b$
$B \to aF \mid \varepsilon$
$A \to a \mid b \mid aF$
$F \to a \mid c$

پاسخ

$first(B) = \{a, \varepsilon\}$
$first(A) = \{a, b\}$
$first(f) = \{a, c\}$
$first(s) = \{a, b\}$

مثال. در گرامر زیر ()first ها را مشخص کنید.

$S \to BAa \mid b \mid Ba$
$B \to aF \mid \varepsilon$
$A \to a \mid b \mid aF \mid \varepsilon$
$F \to a \mid c$

پاسخ

$first(B) = \{a, \varepsilon\}$
$first(A) = \{a, b, \varepsilon\}$
$first(f) = \{a, c\}$
$first(s) = \{a, b\}$

جلسه پنجم و ششم

مجموعه Follow(x) (مجموعه دنباله(x))

Follow یک None Terminal (X) :

مجموعه ترمینال هائی است که در شکل های جمله ای بلافاصله سمت راست **x** می آیند(اولین سمت راست آن)

$$\alpha \to BXa\gamma \to follow(X) = \{a\}$$

قواعد بدست آوردن مجموعه $follow(\alpha)$

- اگر α، $start\ \ symbol$ باشد آنگاه $ عضوی از $follow(\alpha)$ می باشد.
- اگر مولدی به صورت $B \to A\alpha$ داشته باشیم،، آنگاه هر چیزی در $first(\alpha)$ به جز ε به مجموعه $follow(A)$ اضافه می شود.
- اگر مولدی به صورت $B \to \alpha A$ داشته باشیم،، آنگاه هر چیزی در $follow(B)$ به مجموعه $follow(A)$ اضافه می شود.
- اگر مولدی به صورت $A \to \alpha B\beta$ وجود داشته باشد، آنگاه هر چیزی در $first(\beta)$ به جز ε به مجموعه $follow(B)$ اضافه می شود.
- مولدی به صورت $A \to \alpha B\beta$ که $first(\beta)$ حاوی ε باشد آنگاه هر چیزی در مجموعه $follow(A)$ به $follow(B)$ اضافه می شود.

مثال.FIRST() و Follow() هر کدام را حساب کنید.

1) $E \to TE'$
2) $E' \to +TE' \mid \varepsilon$
3) $T \to FT'$
4) $T' \to *FT' \mid \varepsilon$
5) $F \to (E) \mid id$

پاسخ

$first(T') = \{ *, \varepsilon \}$
$first(F) = \{ (, id \}$
$first(E') = \{ +, \varepsilon \}$
$first(T) = \{ (, id \}$
$first(E) = \{ (, id \}$

$follow(E) = \{ \$,) \}$
$follow(E') = \{ \$,) \}$
$follow(T) = \{ +, ,\$,) \}$
$follow(T') = \{ +, ,\$,) \}$
$follow(F) = \{ *, +, \$,) \}$

مثال.FIRST() و Follow() هر کدام را حساب کنید.

1) $S \to ABD$
2) $A \to a \mid BSB$
3) $B \to b \mid D$
4) $D \to d \mid \varepsilon$

پاسخ

$first(S) = \{ a, b, d \}$
$first(A) = \{ a, b, d \}$
$first(B) = \{ b, d\ \varepsilon \}$
$first(D) = \{ d, \varepsilon \}$

$follow\ (S) = \{ \$, b, d \}$
$follow\ (A) = \{ \$, b, d \}$
$follow\ (B) = \{ a, b, d, \$ \}$
$follow\ (D) = \{ a, b, d, \$ \}$

مثال.FIRST() و Follow() هر کدام را حساب کنید.

1) $S \to BCd \mid \varepsilon$
2) $A \to AaSb \mid SbC \mid \varepsilon$
3) $B \to b \mid \varepsilon$
4) $C \to c \mid B$

پاسخ

first(S) = { ε , b,c,d }
first(A) = { ε , b , c,d }
first(B) = { b , ε}
first(C) = { c,b,ε}

follow (S) = { b ,$}
follow (A) = { a }
follow (B) = { a , b , c , d }
follow (C) = { a , d }

گرامر زیر که در آن S , E و F سمبل های غیر ترمینال و $+$, $-$,$*$ سمبل های ترمینال هستند را در نظر بگیرید مجموعه $First(E)$ و $Follow(E)$ را بیابید.

$$S \to +E-$$
$$E \to F \mid *$$
$$F \to +E* \mid \lambda$$

پاسخ $first(E)=\{+,*\}$ $follow(E)=\{-,*\}$

☐ ترمینال ها **folow** ندارند.

گرامر بالا به پایین پیشگوکننده:

☐ مبهم نباشد

☐ بازگشتی چپ نداشته باشد.

☐ فاکتورگیری از چپ بر روی آن اعمال شده باشد.

بازگشتی چپ(left Rotation)

۱- بازگشتی چپ آشکار(بدیهی) $A \to A\alpha$

۲- بازگشتی چپ ضمنی $A \to B\alpha \mid c$, $B \to Ad \mid Bc \mid d$ این بازگشتی چپ برای **A** ضمنی است $A \to B\alpha \to Ad\alpha$

حذف بازگشتی چپ بدیهی:

چنانکه قانونی به شکل $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid Aa_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ داشته باشیم به طوریکه β_i ها با A شروع نشوند و هیچکدام از α_i ها نبایستی ε باشند می توان مشکل بازگشتی چپ را با جایگزینی دو قانون زیر حل کرد

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \varepsilon$$

مثال. بازگشتی گرامر زیر را حذف کنید.

$$E \to EAc \mid Ea \mid d \mid Bc \mid Ed$$

پاسخ
$$E \to dE' \mid BcE'$$
$$E' \to AcE' \mid aE' \mid dE' \mid \varepsilon$$

حذف بازگشتی چپ ضمنی:

☐ گرامر**G** قاعده اپسیلون نداشته باشد.

☐ دورهای $A \xrightarrow{+} A$ در گرامر**G** نباشند

الگوریتم

```
for i = 1 to n do
    for j = 1 to (i-1) do
```

- به جاي هر قانون به شكل كلي $A_i \to A_j\beta$ به طوريكه $A_j \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots \mid \alpha_k$ قرار بده
$A_i \to \alpha_1\beta \mid \alpha_2\beta \mid \ldots \mid \alpha_k\beta$
- حال اگر A_i داراي بازگشتي چپ اشكار است، آن را حذف كن.

مثال. بازگشتی چپ را از گرامر زیر حذف کنید.

$$S \to Aa \mid b$$
$$A \to Ac \mid Sd$$

برای حذف بازگشتی ضمنی چپ داریم

$$A \to Ac \mid Aad \mid bd$$
$$S \to Aa \mid b$$

حال با حذف بازگشتی چپ آشکار داریم

$$S \to Aa \mid b$$
$$A \to bdA'$$
$$A' \to cA' \mid adA' \mid \varepsilon$$

فاکتور چپ.

یک گرامر دارای فاکتور چپ است اگر در سمت چپ حداقل دو تا از تولیدات آن یک عنصر مشترک باشد. مثلا $A \to a\beta_1 \mid a\beta_2$ دارای فاکتور چپ α می باشد به روشنی می توان دید که از این نوع گرامر ها نمی توان در پارسر های بالا به پایین استفاده کرد چون در هنگام پارس کردن نمی توان تعیین کرد که کدام یک از این قوانین تولید ما را به جواب می رساند. برای رفع مشکل بالا می توان از α فاکتور گرفت.

□ در حالت کلی اگر داشته باشیم $A \to a\beta_1 \mid a\beta_2 \mid \ldots \mid a\beta_m \mid \delta_1 \mid \delta_n \mid \ldots \mid \delta_n$ آنگاه با فاکتور گیری از α می توان فاکتور چپ را به صورت زیر حذف نمود

$$A \to \alpha A' \mid \delta_1 \mid \delta_2 \mid \ldots \mid \delta_n$$
$$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

مثال گرامر زیر را در نظر بگیرید

$$S \to iEtS$$
$$S \to iEtSeS \mid a$$
$$E \to b$$

پس از فاکتور گیری داریم

$$S \to iEtSS' \mid a$$
$$S' \to eS \mid \varepsilon$$
$$E \to b$$

مثال گرامر زیر را در نظر بگیرید

$$A \to abcE \mid abcdf \mid abcdT \mid abcdmT'$$

فاکتور گیری مرحله اول

$$A \to abcB'$$
$$B' \to E \mid df \mid dT \mid dmT'$$

فاکتور گیری مرحله دوم

$$A \to abcB'$$
$$B' \to E \mid dZ'$$
$$Z' \to f \mid T \mid mT'$$

ساخت جدول تجزیه(parse Table)

گرامر ورودی جدول تجزیه بایستی خواص زیر را داشته باشد.

- گرامر مبهم نباشد
- بازگشتی چپ نداشته باشد
- فاکتور گیری از چپ روی آن اعمال شده باشد.

جدول تجزیه:

- ماتریس دو بعدی است
- سطر ها غیر پایانه ها را نشان می دهد
- ستون ها پایانه ها را نشان می دهد.

برای تشکیل جدول پارسینگ به صورت زیر عمل می کنیم

۱- برای هر قانون به شکل $A \to \alpha$ مراحل زیر را تکرار کنید

a) به ازای $a \in first(\alpha)$ شماره قاعده $A \to \alpha$ را در محل $M[A,a]$ قرار می دهیم.

b) اگر $\varepsilon \in first(\alpha)$ آنگاه به ازای هر $B \in follow(A)$ شماره قاعده $A \to \alpha$ را در محل $M[A,b]$ قرار می دهیم.

c) اگر $\varepsilon \in first(\alpha)$ و $\$ \in follow(A)$ آنگاه در محل $M[A,\$]$ شماره قاعده $A \to \varepsilon$ را قرار می دهیم.

مثال.

$follow\ (E) = \{\ \$\ ,\ )\ \}$ $first\ (T') = \{\ *\ ,\ \varepsilon\ \}$ $1)\ E \to TE'$

$follow\ (E') = \{\ \$\ ,\ )\ \}$ $first\ (F) = \{\ (\ ,\ id\ \}$ $2-3)\ E' \to +TE' \mid \varepsilon$

$follow\ (T) = \{\ +\ ,\ ,\$,\ )\ \}$ $first\ (E') = \{\ +\ ,\ \varepsilon\ \}$ $4)\ T \to FT'$

$follow\ (T') = \{\ +\ ,\ ,\$,\ )\ \}$ $first\ (T) = \{\ (\ ,\ id\ \}$ $5-6)\ T' \to *FT' \mid \varepsilon$

$follow\ (F) = \{\ *\ ,\ +\ ,\ \$\ ,\ )\ \}$ $first\ (E) = \{\ (\ ,\ id\ \}$ $7-8) F \to (E) \mid id$

	+	*	(	)	id	$
E	E	E	1	E	1	E
E'	2	E	E	3	E	3
T	E	E	4	E	4	E
T'	6	5	E	6	E	6
F	E	E	7	E	8	E

مثال. برای گرامر زیر جدول پارسینگ تشکیل دهید.

(1,2,3) $S \to A;\mid for(A;C;A)S \mid B$

(4,5) $A \to V = E \mid \varepsilon$

(6,7) $C \to E \mid \varepsilon$

(8) $E \to V$

(9) $V \to idX$

(10,11) $X \to 0V \mid \varepsilon$

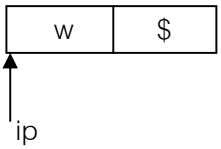
(12) $B \to \{L\}$

(13,14} $L \to SL\varepsilon \mid \varepsilon$

	;	for	(	)	=	id	0	{	}	$
S	1	2	E	E	E	1	E	3	E	E
A	5	E	E	5	E	4	E	E	E	E
C	7	E	E	E	E	6	E	E	E	E
E		E	E	E	E	8	E	E	E	E
V		E	E	E	E	9	E	E	E	E
X	11	E	E	11	11	E	10	E	E	E
B		E	E	E	E	E	12	E	E	E
L	13	13	E	E	E	13	E	13	14	E

	S	A	C	E	V	X	B	L
$first()$	$id\ ;\ for\ \{$	$id\ \varepsilon$	$id\ \varepsilon$	id	id	$0\ \varepsilon$	$\{$	$id\ ;\ for\ \{\ \varepsilon$
$follow()$	$\$id\ ;\ for\ \{$	$;\ )$	$;$	$;\ )$	$=;\ )$	$=;\ )$	$\$id\ ;\ for\ \{$	$\}$

الگوریتم تجزیه پیشگو کننده:LL(1)



- به انتهای پشته ورودی $ اضافه کنید.
- Ip را طوری مقدار دهی کنید که به ابتدای رشته **w** اشاره کند.
- به پشته $ و **startsy** را اضافه کنید(**push**)
- رشته ورودی توکن ها هستند
- در پشته، ترمینال، **non** ترمینال و $ قرار دارد.

با استفاده از جدول تجزیه و همچنین یک پشته به راحتی می توان عمل تجزیه بالا به پایین ر انجام داد.**روش کار به این صورت است** در ابتدا به انتهای رشته ورودی علامت $ که نشان دهنده خاتمه فایل است افزوده می گردد از سوی دیگر علامت سرترم گرامر نیز به همراه $ در داخل پشته قرار می گیرد. عمل تجزیه زمانی با موفقیت به پایان می رسد که در ورودی $ و در پشته تنها علامت $ باقی مانده باشد.

پس از قرار دادن سرترم گرامر و $ در پشته از ورودی یک علامت خوانده می شود، اگر فرض کنیم که علامت موجود در ورودی **a** و عنصر بالای پشته **x** باشد حالات زیر ممکن است اتفاق بیفتد

۱- اگر $a = X = \$$ باشد عمل تجزیه با موفقیت به پایان رسیده است

۲- اگر $a = X \neq \$$ باشد، X از بالای پشته حذف می شود و عنصر بعدی از ورودی خوانده می شود

۳- اگر X یک ترمینال باشد و $a \neq x$ خطای نحوی رخ میدهد

۴- اگر X یک ترم میانی(غیر ترمینال) باشد برنامه به محل $M[X,a]$ در جدول مراجعه می کند در این جا دو حالت امکان دارد

الف) قاعده ای به صورت $X \to UVW$ در خانه $M[X,a]$ وجود دارد در این حالت عنصر X از پشته pop شده و UVW به درون پشته push می شود به طوریکه عنصر U در بالای پشته قرار می گیرد.

ب) خانه $M[X,a]$ خالی است در این حالت یک خطای نحوی رخ میدهد.

توضیحات بالا را به صورت الگوریتم زیر نیز می توان نوشت.

```
set ip to point to the first symbol of W$;
repeat
    let X be the top stack symbpl and α the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = α then
            pop X from the stack and Advance ip;
        else error();
    else  /* X is nontermina l */
      if M[X,a] = X → Y1Y2...Yk then begin
            pop X from the stack;
            push Yk,Yk-1,...Y1 on to the stack, with Y1 on top;
            output the production X → Y1Y2...Yk
       end
       else error
Until X = $ /* stack is empty */
```

عمل تجزیه برای جمله $id + id * id$ در زیر تشریح شده است(با توجه به جدول تجزیه صفحه قبل).

پشته	ورودی	قانون استفاده شده
$E	$id + id * id$	
$E'T	$id + id * id$	$E \to TE'$
$E'T'F	$id + id * id$	$T \to FT'$
$E'T'id	$id + id * id$	$F \to id$
$E'T'	$+ id * id$	
$E'	$+ id * id$	$T' \to \varepsilon$
$E'T+	$+ id * id$	$E' \to +TE'$
$E'T	$id * id$	
$E'T'F	$id * id$	$T \to FT'$
$E'T'id	$id * id$	$F \to id$
$E'T'	$* id$	
$E'T'F*	$* id$	$T' \to *FT'$
$E'T'F	id	
$E'T'id	id	$F \to id$
$E'T'	$	
$E'	$	$T' \to \varepsilon$
$	$	$E' \to \varepsilon$

گذر از خطا

خطا

۱- بالای پشته پایانه باشد و با ورودی فعلی تطابق نداشته باشد

۲- بالای پشته غیر پایانه ولی محل $M[X, token]$ خالی $(null)$ باشد.

دو روش برای اصلاح خطا های نحوی در روش $LL(1)$ وجود دارد.

- *panic Mode*
- *pharse level*

در روش *panic Mode* اگر پارسر با مراجعه به یک خانه خالی جدول تجزیه یک خطای نحوی بیابد، آنقدر از ورودی حذف می کند تا به یکی از اعضای مجموعه ای موسوم به مجموعه *synchornizing* برسد در روش *panic Mode* با ازای هر غیر پایانه در گرامر یک مجموعه *synchornizing* در نظر گرفته میشود. کارائی روش *panic Mode* نیز بستگی به انتخاب مناسب مجموعه *synchornizing* دارد. این

مجموعه باید به گونه ای انتخاب شود که عمل تجزیه بتواند بدون حذف قسمت زیادی از ورودی به کار خود ادامه دهد. یک انتخاب مناسب در نظر گرفتن مجموعه *follow* هر غیر پایانه ای به عنوان مجموعه *synchornizing* آن غیر پایانه است. با این وجود در نظر گرفتن مجموعه *follow* تنها برای *synchornizing* کافی نیست. برای این که حذف کمتری در برنامه صورت بگیرد می توان نماد های بیشتری را به این مجموعه افزود، مثلا می توان مجموعه *first* غیر پایانه ها را نیز به مجموعه *synchornizing* آنها افزود.

توسعه الگوریتم تجزیه جهت اعمال گذر از خطا

```
tos = 0;
stack[tos++] = $;
stack[tos++] = start symbol;
token = get token();
do{ // X is top of the stack
if X is a terminal or X = $;
    if X = token{
        pop X;
        token = get token();}
    else
       pop X;
else // X is a non-terminal;
if M[X,token] = NoT NULL && X ≠ synch;
{
pop X;
push Y_k, Y_{k-1}, ..., Y_1
}
else if X = synch;
    if X is not sign N.T in stack;
    pop X;
else
    token = get token();
else
token = get token();
} while(Token ≠ $);
```

جلسه هفتم

1. $S \rightarrow iEtS\ S'$
2. $S \rightarrow a$
3. $S' \rightarrow eS$
4. $S' \rightarrow \varepsilon$
5. $E \rightarrow b$

مثال. آیا گرامر زیر LL(1) هست؟

حل. ابتد ()First و ()Follow ها را پیدا می کنیم.

$First(S) = \{i, a\}$
$First(S') = \{e, \varepsilon\}$
$First(E) = \{b\}$
$Follow(S) = \{e, \$\}$
$Follow(S') = \{e, \$\}$
$Follow(E) = \{t\}$

جدول تجزیه

	a	b	e	i	t	$
S	2			1		
S'			3,4			4
E		5				

با توجه به جدول تجزیه گرامر فوق LL(1) نیست.
اگر گرامری در ورودی یا در مدخلی از جدول تجزیه بیش از یک مقدار داشته باشد آن گرامر LL(1) نیست .

بعضی گرامر ها را با
- رفع ابهام
- حذف بازگشتی چپ
- فاکتورگیری از چپ

می توان به LL(1) تبدیل کرد

تشخیص LL(1) بودن بدون استفاده از جدول

گرامری LL(1) است که برای هر قاعده آن که به فرم $A \rightarrow \alpha \mid \beta$ باشد شرایط زیر برقرار باشد.

۱- $First(\beta) \bigcap First(\alpha) = \phi$
۲- حداکثر یکی از α و β رشته ε را تولید کند.
۳- اگر $\alpha \xrightarrow{*} \varepsilon$ (به عبارتی $\varepsilon \in First(\alpha)$) باشد $First(\beta) \bigcap Follow(A) = \phi$
نکته: اگر $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ آنگاه بایستی $First(\alpha_1) \bigcap First(\alpha_2) \bigcap First(\alpha_3) \bigcap \ldots \bigcap First(\alpha_n) = \phi$

مثال. آیا گرامر زیر LL(1) هست.

1,2. $S \rightarrow A \mid B$
3,4. $A \rightarrow Ab \mid f$
5. $B \rightarrow cdA \mid ceA$

خیر زیرا $First(cdA) \bigcap First(ceA) = c \neq \phi$

پارسر های پایین به بالا ، انتقال- کاهش

از رشته ورودی شروع کرده و با جایگزینی قواعد سعی در تولید Start Symbol را دارند.

مثال تجزیه پایین به بالا– انتقال کاهش

دنباله ورودی abbcde

S → aABe
A → Abc \ b
B → d

اشتقاق راست S → aABe → aAde → aAbcde → abbcde

abbcde	انتقال
aAbcd	کاهش
aAde	انتقال
aABe	کاهش
S	

پارسرهای پایین بالا به جهت عکس اشتقاق راست عمل می کنند.

عبارت(Phars) : بخشی از یک فرم جمله ای است که از یک غیر پایانه بوجود آمده باشد به عنوان نمونه در مثال زیر، β یک عبارت محسوب می شود. $S \Rightarrow^* \alpha A \gamma \Rightarrow^+ \alpha\beta\gamma$

عبارت ساده(Simple Phars) : عبارتی است که در یک قدم بوجود آمده باشد در بسط زیر β یک عبارت ساده است

$S \Rightarrow^* \alpha A \gamma \Rightarrow \alpha\beta\gamma$

دستگیره(Handle): عبارت ساده ای است که در جهت عکس یک اشتقاق راست در نظر گرفته می شود سمت راست Handle هیچ غیر ترمینالی وجود ندارد.

تعریف دیگر: زیر رشته ای منطبق بر سمت راست یک قانون می باشد که ایجاد کننده یک کاهش به غیر پایانه سمت چپ آن قانون می باشد.

تعریف دقیق تر: اگر ترتیب کاهش رشته ورودی به نماد شروع معکوس سمت راست ترین اشتقاق(اشتقاق راست) باشد دنباله ای را که در هر مرحله کاهش می یابد، **دستگیره** می نامیم.

مثال. دستگیره ها را در کاهش bccdef در با توجه به گرامر زیر مشخص کنید.

$S \to bBCf$
$B \to Bcd \mid c$
$C \to e$

حل. ابتدا bccdef را بوسیله سمت راست ترین اشتقاق تولید می کنیم

S
bBCf
bBef
bBcdef
bccdef

دستگیره	کاهش
c	bccdef
Bcd	bBcdef
e	bBef
bBCf	bBCf
	S

پیاده سازی پارسرهای پایین به بالا با استفاده از یک پشته:

در این روش از یک پشته و یک بافر ورودی جهت نگهداری رشته ورودی استفاده می شود. در شروع عمل پارسینگ روی پشته و انتهای رشته ورودی $ را قرا می دهیم.

اعمال مورد استفاده در این روش:

انتقال(shift) : در انتقال، سمبول ها(Token های) ورودی بالای پشته قرار داده می شوند تا زمانی که یک handel در بالای پشته ظاهر شود. به عبارتی عمل انتقال تا وقتی که علایم روی پشته با سمت راست یکی از قوانین گرامر منطبق شود ادامه می یابد.

کاهش(Reduce) : مجموعه عناصر روی پشته که منطبق با سمت راست یکی از قوانین است را حذف و به جای آن یک غیر ترمینال که در سمت چپ قانون مذکور است را قرار می دهد.

قبول ورودی(Accept) : پارسر پایان موفقیت امیز تجزیه را اعلام می کند.

تشخیص خطا(Error) : پارسر یک خطای نحوی تشخیص داده و رویه خطا پرداز را فرا می خواند.

در زیر مراحل پارسینگ رشته id+id*id بر اساس گرامر زیر دیده می شود.

(1) $E \rightarrow E + E$
(2) $E \rightarrow E * E$
(3) $E \rightarrow (E)$
(4) $E \rightarrow id$

پشته	ورودی	عمل
$	id+id*id$	انتقال id
$id	+id*id$	کاهش $E \rightarrow id$
$E	+id*id$	انتقال +
$E+	id*id$	انتقال id
$E+id	*id$	کاهش $E \rightarrow id$
$E+E	*id$	انتقال *
$E+E*	id$	انتقال id
$E+E* id	$	کاهش $E \rightarrow id$
$E+E*E	$	کاهش $E \rightarrow E*E$
$E+E	$	کاهش $E \rightarrow E+E$
$E	$	Accept

در روش پارسینگ انتقال-کاهش دو نوع مشکل وجود دارد.
۱- تصمیم گیری در این مورد که کدام مجموعه از عناصر روی پشته با سمت راست قوانین منطبق است(مشکل در انتخاب دستگیره)
۲- انتخاب قانونی که کاهش تحت آن انجام شود. این مشکل وقتی پیش می اید، که سمت راست بیش از یک قانون با مجموعه ای از عناصر روی پشته منطبق باشد. چنین حالتی را تداخل کاهش-کاهش گویند

انواع تداخل در پارسینگ انتقال- کاهش

۱- **تداخل انتقال-کاهش(Shift-Reduce conflict):** زمانی روی می دهد که پارسر نتواند تصمیم بگیرد که عمل انتقال را انجام بدهد یا عمل کاهش را.
۲- **تداخل کاهش-کاهش(Reduce-Reduce conflict):** اگر بیش از یک قاعده برای کاهش در یک لحظه قابل استفاده باشد این نوع مشکل رخ داده است.

روش پارسینگ LR

روش پارسینگ LR جزء دسته الگوریتم های پارسینگ پایین به بالا می باشد، الگوریتم های LR خود به سه دسته کوچکتر تقسیم می شوند. که تفاوتشان در ساخت جدول تجزیه شان می باشد. در حالی که روش پارسینگ برای آنها مشابه است

- **(Simple LR)SLR**
- **(Look Ahead LR)LALR**
- **(Cannonical)CLR**

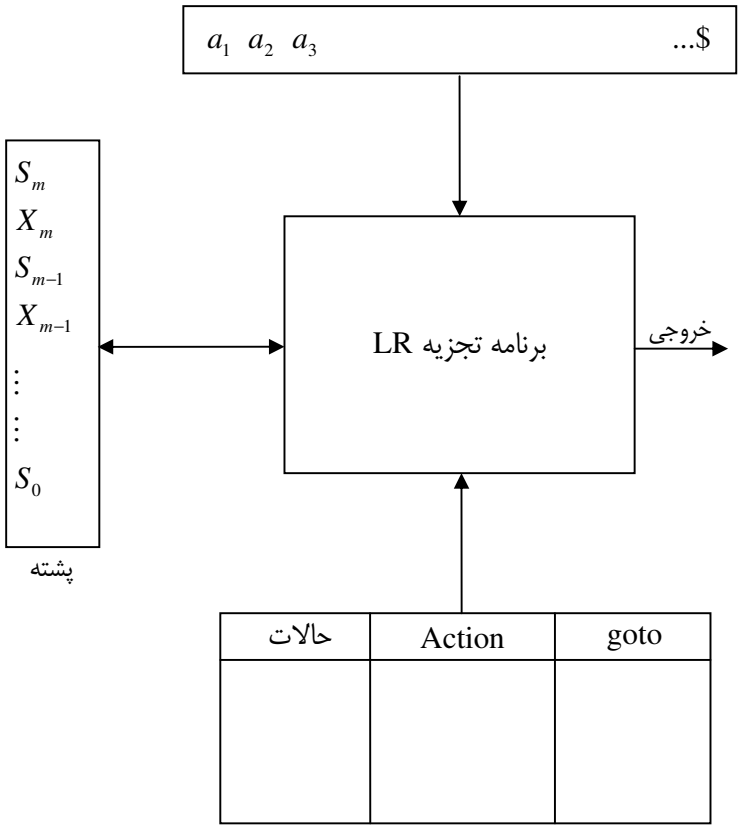
برای اکثریت قریب به اتفاق زبان های برنامه نویسی می توان از این روش پارسینگ استفاده نمود پارسر های LR دارای 4 جزء اصلی هستند

- **بافر ورودی**، رشته ورودی در این قسمت قرار می گیرد و به انتهای رشته ورودی $ اضافه می شود LR با دیدن $ پایان رشته ورودی را تشخیص می دهد
- **پشته:** محتوای پشته به صورت $S_0X_1S_1\cdots X_mS_m$ نشان داده می شود. S_i نشان دهنده حالات و X_i پایانه ها و غیر پایانه های گرامر است، نمادهای ورودی به پشته منتقل می شوند تا یک دستگیره در پشته یافت شود، پس از کشف دستگیره کاهش انجام میگیرد
- **برنامه پارسینگ:** این برنامه قسمت اصلی تجزیه کننده است برنامه تجزیه کننده بر اساس نماد جاری رشته ورودی و حالات بالای پشته و محتوای جدول تجزیه مرحله بعدی را تعیین می کند
- **جدول پارسینگ**: این بخش شامل دو قسمت action و goto است. قسمت action عملی را که باید انجام شود و بخش goto حالت بعدی را مشخص می کند

مزیت: سرعت اعلان خطا های نحوی از این پارسرها بالاست

عیب: جهت ساخت جدول تجزیه این پارسر ها حجم محاسبات دستی خیلی زیاد است.

در شکل زیر اجراء مختلف پارسر LR و رابطه انها با هم نشان داده شده است



یک قلم(1-item) برای LR(0) (قلم به طور خلاصه) از گرامر G مولدی از G با یک نقطه(یا علامت خاص دیگر) در مکانی در سمت راست آن است بنابراین مولد $A \to XYZ$ چهار قلم را ایجاد می کند.

A→ . XYZ
A→ X.YZ
A→ XY. Z
A→ XYZ.

مولد $A \to \varepsilon$ تنها یک قلم به صورت $A \to .$ را تولید می نماید، یک قلم می تواند با یک زوج عدد صحیح نشان داده شود، اولین عدد شماره مولد و دومین عدد موقعیت نقطه را مشخص می نماید، در نتیجه قلم مشخص می نماید که چه مقداری از مولد، در یک نقطه از فرایند تجزیه دیده شده است برای مثال اولین قلم فوق نشان میدهد که انتظار دیدن رشته مشتق پذیر از XYZ را در ورودی داریم. دومین قلم نشان می دهد که اخیرا در ورودی رشته مشتق پذیر از X دیده شده است و این که در ادامه انتظار دیدن یک رشته مشتق پذیر از YZ را داریم.

عمل Closure :

اگر I مجموعه ای از اقلام برای گرامر G باشد، آنگاه Closure(I) مجموعه اقلام ایجاد شده از I با استفاده از دو قانون زیر است

۱- در ابتدا هر قلم موجود در I به Closure(I) اضافه می شود.
- اگر $A \to \alpha \,.\, B\beta$ در Closure(I) موجود باشد و $B \to \gamma$ یک مولد باشد انگاه قلم B→. γ اگر قبلا در I نباشد به آن اضافه می گردد. این قانون تا زمانی به کار گرفته می شود که هیچ قلم جدید دیگری نتواند به Closure(I) اضافه شود.

مثال. باتوجه به گرامر زیر اگر I مجموعه ای یک قلم به صورت $\{[E' \to .E]\}$ باشد آنگاه Closure(I) شامل چه اقلام هائی خواهد شد؟

$$E' \to E$$
$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

پاسخ.

$$E' \to .E$$
$$E \to .E + T$$
$$E \to .T$$
$$T \to .T * F$$
$$T \to .F$$
$$F \to .(E)$$
$$F \to .id$$

ساخت جدول تجزیه پارسر SLR

ابتدا می بایست دیاگرام حالت این پارسر رسم شود جهت رسم دیاگرام حالت یک پارسر به صورت زیر عمل می کنیم

- اگر S علامت شروع گرامر باشد ابتدا قاعده ای به فرم $S' \to S$ ($S' \to S\$$)که در آن S' یک غیر پایانه جدید است به گرامر اضافه می کنیم، گرامر حاصل را گرامر افزوده (augmented grammar) گویند
- حالت جدیدی به نام S_0 ایجاد می کنیم و ایتم $S' \to .S_0$ در S_0 قرار داده و سپس Closure این ایتم را به S_0 اضافه می کنیم

- اگر به طور کلی در حالت S_i ایتم هائی به فرم $\begin{cases}\{A_1 \to \alpha_1 . x\beta_1 \\ \vdots \\ \{A_n \to \alpha_n . x\beta_n\end{cases}$ داشته باشیم(که در آن x می تواند پایانه و یا غیر پایانه باشد) در

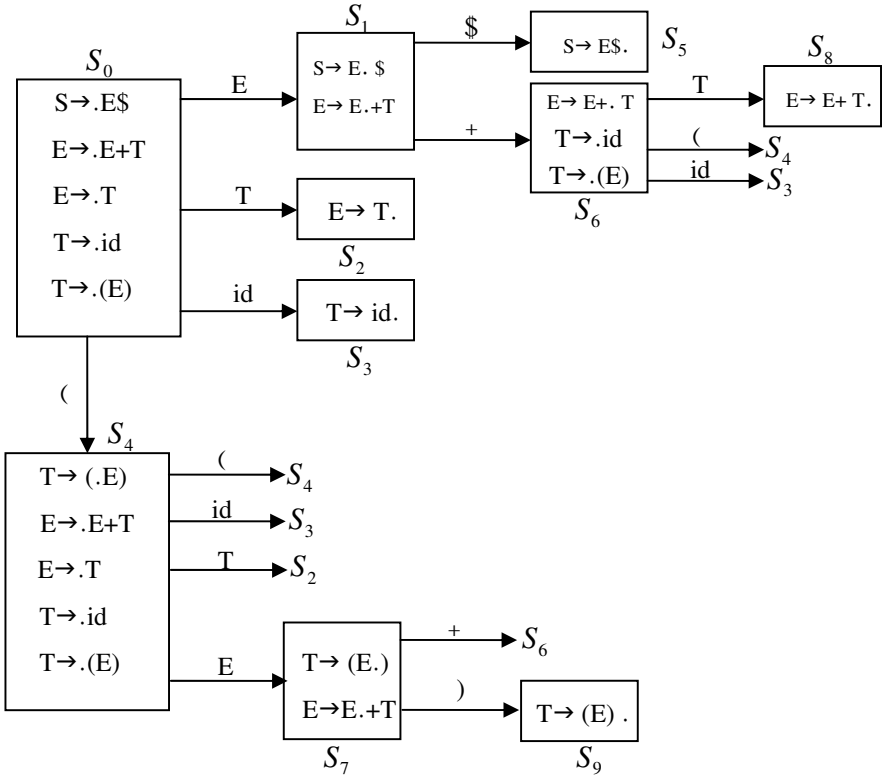
در این صورت حالت جدیدی به نام S_j ایجاد کرده، S_i را توسط لبه ای با بر چسب x به S_j منتقل می کنیم و ایتم فـوق را بـا ایـن تغییر که در همه علامت . به بعد از علامت x منتقل شده است در وضیعت جدید قرار می دهیم و سپس بستار این ایتم ها را محاسبه و در S_j قرار می دهیم. چنانکه در دیاگرام حالتی مانند S_k وجود داشته باشد که دقیقا مطابق S_j باشد در این صورت S_j ایجاد نشده و در عوض S_i توسط لبه ای با برچسب x به S_k متصل می گردد، این قدم را آنقدر تکرار می کنیم تا دیگر حالت جدیدی به دیـاگرام اضافه نشود.

مثال. دیاگرام حالت گرامر زیر را رسم کنید.

$$E \to E+T$$
$$E \to T$$
$$T \to id$$
$$T \to (E)$$

در قدم اول یک Start Symbol مانند $S \to E\$$ را به گرامر اضافه می کنیم و حالت جدید S_0 را ایجاد کرده و مراحل گفته شده را پـی می گیریم

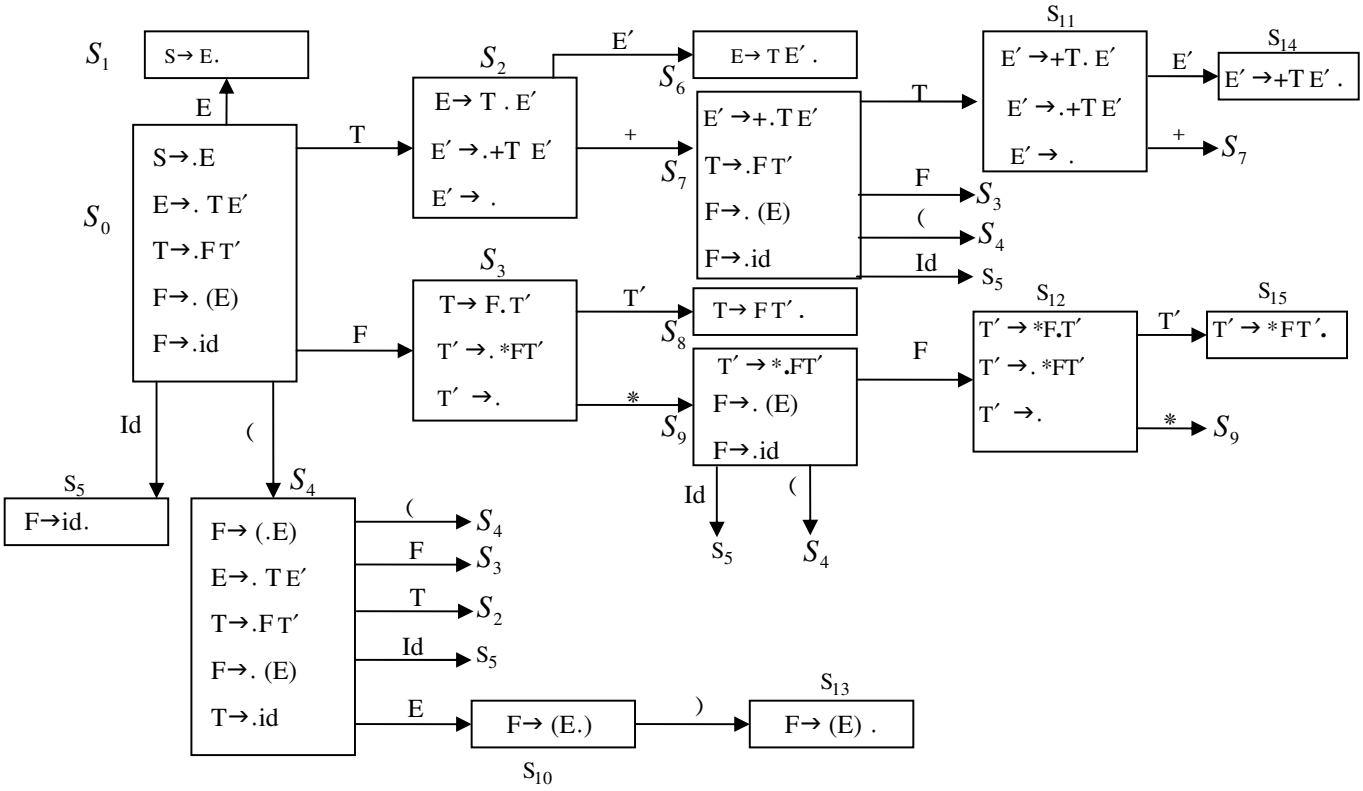
$$S \to E$$
$$E \to E+T$$
$$E \to T$$
$$T \to id$$
$$T \to (E)$$



جلسه هشتم

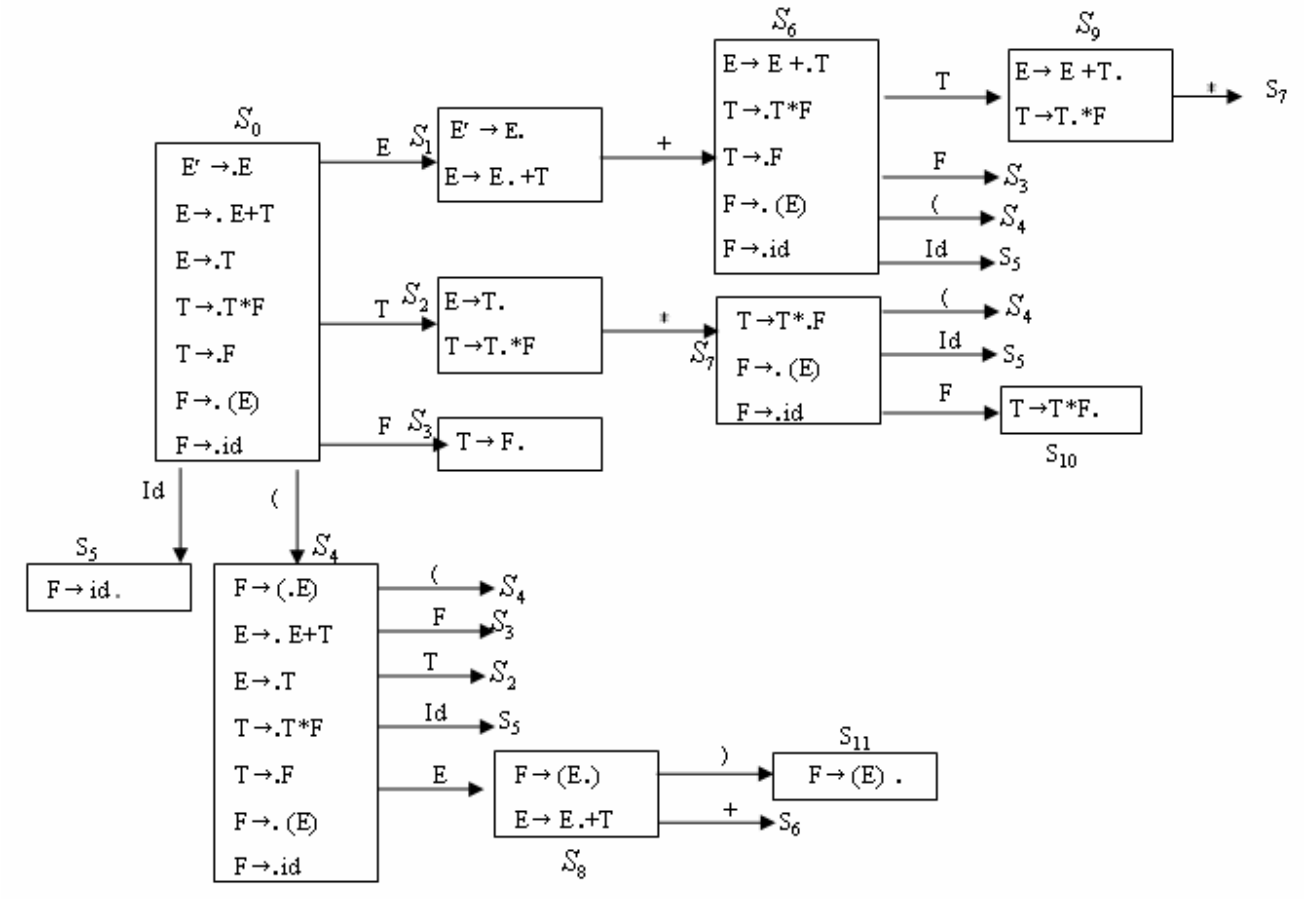
مثال. دیاگرام حالت گرامر زیر را رسم کنید.

1) $E \rightarrow TE'$
2) $E' \rightarrow +TE'$
3) $E' \rightarrow \varepsilon$
4) $T \rightarrow FT'$
5) $T' \rightarrow *FT'$
6) $T' \rightarrow \varepsilon$
7) $F \rightarrow (E)$
6) $F \rightarrow id$



مثال. دیاگرام حالت گرامر زیر را رسم کنید.

$E' \rightarrow E$
1) $E \rightarrow E + T$
2) $E \rightarrow T$
3) $T \rightarrow T * F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$



در جدول تجزیه SLR :

- سطرها State ها را نشان می دهند.
- ستون های قسمت اول(action)، ترمینال ها +$ را نشان می دهند
- ستون های قسمت دوم(goto) غیرترمینال ها را نشان می دهند.

برای تکمیل قسمت action و goto به صورت زیر عمل می کنیم

- اگر a یک ترمینال باشد و در حالت S_i با دیدن a به خانه S_j برویم در action[i,a] قرار می دهیم S_j
- در حالت S_i به ازای تمامی دستورات $A \to \alpha.$ ، به ازای تمام اعضای مجموعه Follow(A) (مثلا b) در action[i,b] قرار می دهیم R_n به طوریکه n شماره دستور $A \to \alpha.$ است.
- اگر در حالت S_i دستور $S' \to S.$ قرار داشته باشد در action[i,$] فرمان accept قرار می گیرد.
- برای تکمیل بخش goto کافی است چنانکه از حالت S_i با یک غیر ترمینال (مثلا A) به حالت S_j می رویم در goto[i,A] قرار می دهیم j

مثال. جدول تجزیه دیاگرام صفحه قبل را رسم کنید.

Follow(E) = {+,),$}
Follow(T) = {*,+,),$}
Follow(F) = {*,+,),$}

	Action						goto		
	id	+	*	(	)	$	E	T	F
S_0	S_5			S_4			1	2	3
S_1		S_6				Accept			
S_2		R_2	S_7		R_2	R_2			
S_3		R_4	R_4		R_4	R_4			
S_4	S_5			S_4			8	2	3
S_5		R_6	R_6		R_6	R_6			
S_6	S_5			S_4				9	3
S_7	S_5			S_4					10
S_8		S_6			S_{11}				
S_9		R_1	S_7		R_1	R_1			
S_{10}		R_3	R_3		R_3	R_3			
S_{11}		R_5	R_5		R_5	R_5			

جهت تجزیه یک رشته با استفاده از جدول SLR مراحل زیر را طی می کنیم.

ابتدا علامت $ را به انتهای رشته ورودی اضافه می کنیم و حالت اولیه به پشته اضافه می گردد.

۱- **انتقال:** اگر $action[i,a] = S_n$ ، برنامه تجزیه کننده ابتدا نماد ورودی a و سپس n را به بالای پشته منتقل میکند.

۲- **کاهش:** اگر $action[i,a] = R_n$ باشد یک دستگیره یافت شده است که باشد کاهش یابد اگر n شماره قاعده تولید $A \to \beta$ باشد، برنامه تجزیه کننده دنباله β را از بالای پشته حذف می کند و پس از حذف β اگر حالت بالای پشته m باشد برنامه تجزیه کننده بتدا A و سپس، عدد موجود در goto[m,A] را به بالای پشته اضافه می کند، نکته این است که برای حذف $|\beta|$ از بالای پشته به تعداد $2|\beta|$ از بالای پشته حذف می گردد زیرا بین هر نماد در پشته یک شماره حالت نیز جود دارد.

۳- **پذیرش:** اگر $action[i,a] = accept$ باشد تجزیه رشته ورودی با موفقیت انجام شده است

۴- **خطا:** اگر $action[i,a] = error$ باشد تجزیه رشته ورودی با عدم موفقیت روبرو شده است

مثال. با استفاده از جدول تجزیه زیر رشته id*id+id را تجزیه کنید

$E' \rightarrow E$
1) $E \rightarrow E + T$
2) $E \rightarrow T$
3) $T \rightarrow T * F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

جدول تجزیه SLR

	Action						goto		
	id	+	*	(	)	$	E	T	F
S_0	S_5			S_4			1	2	3
S_1		S_6				Accept			
S_2		R_2	S_7		R_2	R_2			
S_3		R_4	R_4		R_4	R_4			
S_4	S_5			S_4			8	2	3
S_5		R_6	R_6		R_6	R_6			
S_6	S_5			S_4				9	3
S_7	S_5			S_4					10
S_8		S_6			S_{11}				
S_9		R_1	S_7		R_1	R_1			
S_{10}		R_3	R_3		R_3	R_3			
S_{11}		R_5	R_5		R_5	R_5			

با توجه به توضیحات صفحه قبل می توان الگوریتم تجزیه یک رشته را به این شکل نوشت.
نکته: اگر گرامری SLR باشد آنگاه LALR و CLR نیز هست.

مراحل تجزیه رشته id*id+id

پشته	رشته ورودی	خروجی
0	id*id+id$	Shift5
0id5	*id+id$	$F \rightarrow id$
0F3	*id+id$	$T \rightarrow F$
0T2	*id+id$	Shift7
0T2*7	id+id$	Shift5
0T2*7id5	+id$	$F \rightarrow id$
0T2*7F10	+id$	$T \rightarrow T * F$
0T2	+id$	$E \rightarrow T$
0E1	+id$	Shift6
0E1+6	id$	Shift5
0E1+6id5	$	$F \rightarrow id$
0E1+6F3	$	$T \rightarrow F$
0E1+6T9	$	$E \rightarrow E + T$
0E1	$	Accept

```
while(!EOF){
IF(action[S_i , x] = S_j)
        Push(x);
        Push(S_j);
        x = next Token();
        else IF(action[S_i , x] = R : A → α)
       For(i = 0 to 2|α| )
       pop();
       Push(A);
       Push(goto(S_i , A) }
else IF[action(s, x] = Accept) Then
       return
else
      error();
} // end
```

جلسه نهم و دهم

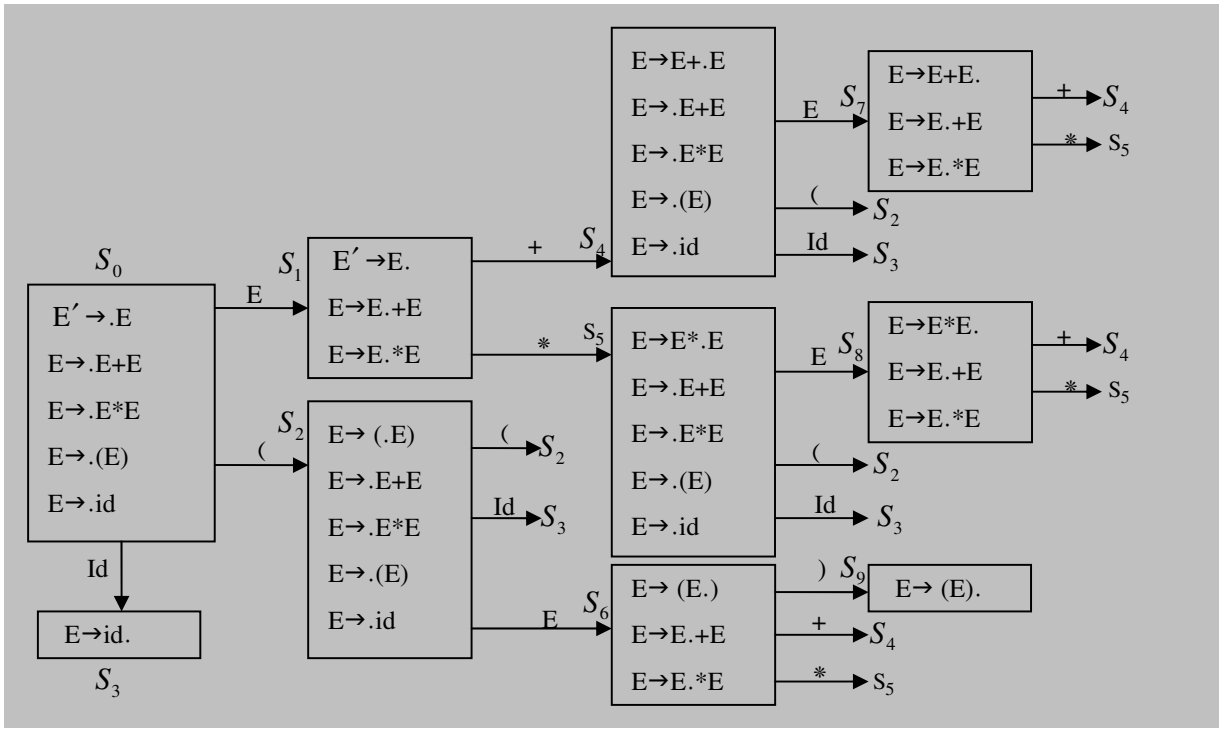
مزیت استفاده ازگرامر های مبهم در تجزیه LR :

با استفاده از ابهام می توان جداول کوچکتری برای تجزیه پایین به بالا ایجاد کرد(مزیت). استفاده از گرامر های مبهم در بسیاری از موارد موجب ایجـاد تداخل هائی در جدول تجزیه حواهد شد برای این که عمل تجزیه پایین به بالا امکان پذیر باشد می بایست این تداخل ها را از بـین بـرد . بنـابراین بـا مبهم کردن گرامر گرچه تعداد قوانین و در نتیجه حالات جدول کاهش می یابد اما مشکل تداخل ایجاد می شود.

برای روشن شدن موضوع گرامر $E \to E+T \mid T$, $T \to T*F \mid F$, $F \to id \mid (E)$ را در نظر بگیرید، این گرامر (اگر دیاگرام آن رسم) به یازده حالت(State) نیاز دارد حال می خواهیم دیاگرام مبهم شده این گرامر را که در زیر آمده است رسم کنیم و همچنین جدول تجزیه را ایجاد کرده و به بررسی تداخل ها و تعداد حالت(که کـاهش یافته) بپردازیم.

$E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$

رسم دیاگرام:



حال با توجه به دیاگرام رسم شده در صفحه بعد جدول تجزیه را تشکیل می دهیم.

جدول تجزیه دیاگرام صفحه قبل.

$Follow(E) = \{\$,+,*,)\}$

state	action					goto	
	id	+	*	(	)	$	E
0	S_3			S_2			1
1	S_4	S_5				acc	
2	S_3			S_2			6
3		R_4	R_4		R_4	R_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		S_4, R_1	S_5, R_1		R_1	R_1	
8		S_4, R_2	S_5, R_2		R_2	R_2	
9		R_3	R_3		R_3	R_3	

ملاحظه می شود که در چهار خانه تداخل shift/Reduce رخ داده است. جهت از دست ندادن سرعت از گرامر مبهم استفاده می شود و جهت حل تداخل از الویت عملگر ها استفاده می شود.

تجزیه CLR :

آیتم LR(1) : یک ایتم LR(1) یک زوج مرتب متشکل از یک آیتم LR(0) و یک مجموعه پایانه به نام مجموعه پیش بینی(Lookahead) است، و معمولا به صورت $[A \to \alpha., LA]$ نمایش داده می شود رابطه زیر در مورد مجموعه پیش بینی و مجموعه Follow غیر پایانه سمت راست این آیتم ها وجود دارد. $LA \subseteq Follow(A)$

توضیح این که جهت پر کردن جدول تجزیه CLR ابتدا می بایست دیاگرام حالت آن را رسم کنیم دیاگرام حالت CLR مانند SLR است وآیتم شروع $< S' \to .S, \{\$\} >$ می باشد

الگوریتم محاسبه تابع بستار که در مورد آیتم های LR(0) توضیح داده شده بسیار شبیه الگوریتم یافتن بستار آیتم های LR(1) است با این تفاوت که در این جا باید به صورت زیر برای آیتم های غیر هسته ای جدیدی که به مجموعه بستار اضافه می شوند، مجموعه پیش بینی تعیین نمود.

$[A \to \alpha.B\beta, \{a\}]$

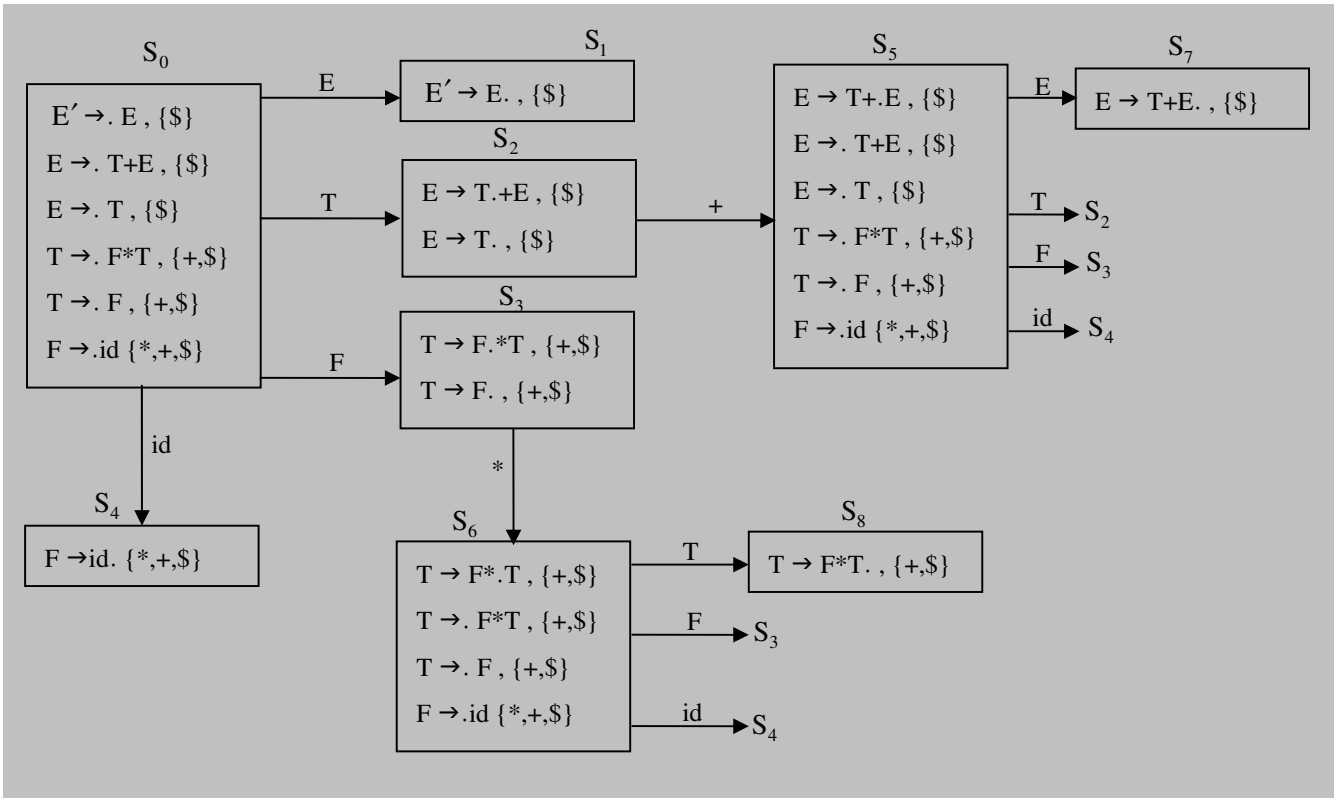
$[B \to .\delta, \{b\}] \mid b \in First(\beta a)$

مثال. دیاگرام حالت CLR گرامر زیر را رسم کنید.

$E' \to E$

1) $E \to T + E$
2) $E \to T$
3) $T \to F * T$
4) $T \to F$
5) $F \to id$

صفحه بعد دیاگرام رسم شده است.



نحوه تکمیل جدول CLR(1) :

روش تهیه جدول CLR(1) بسیار شبیه روشی است که در رابطه با SLR(1) بیان گردید، با این تفاوت که اگر در این جا در وضیعت I آیتمی بـه فـرم $[A \to \alpha\bullet, \{b\}]$ داشته باشیم، در خانه های action[i, b] دستور $A \to \alpha$ Re duce قرار می دهیم، به عبارت دیگر در این جا به جای اسـتفاده از مجموعه Follow ها از مجموعه پیش بینی شده استفاده می کنیم.

مثال. جدول تجزیه را برای دیاگرام مثال قبلی تکمیل کنید.

state	action				goto		
	id	+	*	$	E	T	F
S_0	S_4				1	2	3
S_1				ac			
S_2		S_5		R_2			
S_3		R_4	S_6	R_4			
S_4		R_5	R_5	R_5			
S_5	S_4				7	2	3
S_6	S_4					8	3
S_7				R_1			
S_8		R_3		R_3			

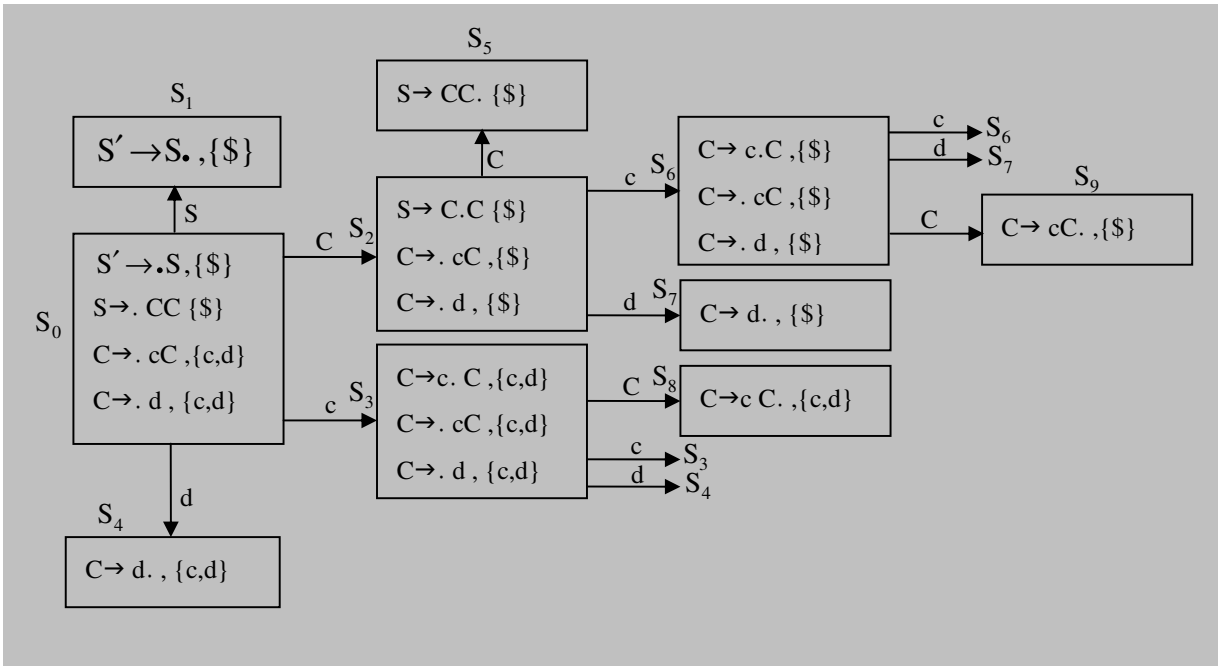
مثال. دیاگرام حالت CLR گرامر زیر را رسم کنید.

$S' \rightarrow S$

1) $S \rightarrow CC$

2) $C \rightarrow cC$

3) $C \rightarrow d$



جدول تجزیه دیاگرام رسم شده.

state	action			goto	
	c	d	$	S	C
S_0	S_3	S_4		1	2
S_1				ac	
S_2	S_6	S_7			5
S_3	S_3	S_4			8
S_4	R_3	R_3			
S_5			R_1		
S_6	S_6	S_7			9
S_7			R_3		
S_8	R_2	R_2			
S_9			R_2		

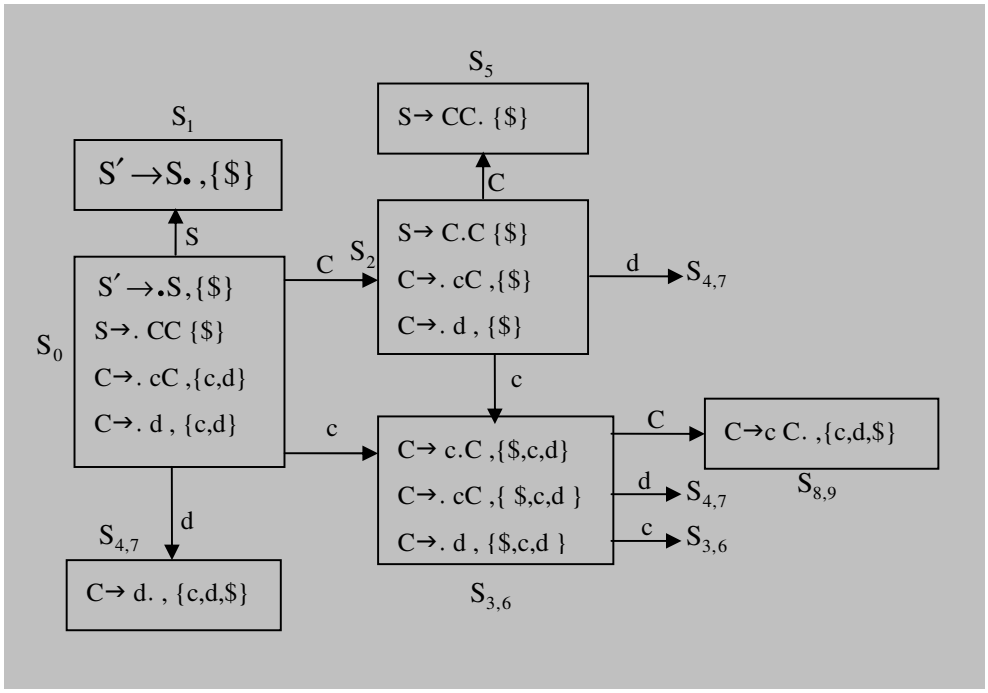
رسم دیاگرام و جدول تجزیه LALR :

برای رسم دیاگرام LALR یک گرامر باید، ابتدا دیاگرام CLR گرامر را رسم کرد. سپس از روی دیاگرام CLR با انجام دو عمل زیر دیاگرام LALR بدست می آید.

ابتدا در دیاگرام CLR به دنبال وضیعت هائی می گردیم که بخش آیتم های LR(0) آنها(که به آن اصطلاحا هسته یا Core یک وضیعت می گویند) یکسان باشد. سپس در دیاگرام وضیعت هائی را که هسته یکسانی دارند ادغام می کنیم و مجموعه پیش بینی آیتم های این وضیعت ها را نظیر به نظیر با هم ادغام می کنیم

مثال. با توجه به دیاگرام CLR صفحه 39 ، دیاگرام LALR مربوطه را رسم کنید.

ملاحظه می شود که در دیاگرام مربوطه S_3 با S_6، S_4 با S_7 و S_8 با S_9 قابل ادغام هستند(هسته یکسان دارند)



اگر گرامری CLR باشد و پس از ادغام وضیعت های مزبور در جدول تجزیه LALR گرامر نیز تداخلی بوجود نیاید می توان نتیجه گرفت که گرامر LALR است. حال اگر گرامری CLR بوده و LALR نباشد، پس از ادغام در جدول تجزیه LALR گرامر تداخل نوع کاهش/کاهش بروز خواهد کرد، هیچگاه در اثر ادغام در جدول LALR تداخل نوع انتقال-کاهش بوجود نخواهد آمد مگر آنکه گرامر CLR هم نبوده باشد که در این صورت قطعا LALR هم نیست، به عبارتی در چنین حالتی، تداخل قطعا قبل از ادغام نیز وجود داشته است. طریقه تهیه جدول تجزیهLALR از روی دیاگرام ادغام شده عینا همانند روش تهیه جدول CLR است. به عنوان یک نمونه جدول تجزیه LALR مثال قبل به صورت زیر خواهد بود.

state	action			goto	
	c	d	$	S	C
0	$S_{3,6}$	$S_{4,7}$		1	2
1			acc		
2	$S_{3,6}$	$S_{4,7}$			5
36	$S_{3,6}$	$S_{4,7}$			89
47	R_3	R_3	R_3		
5			R_1		
89	R_2	R_2	R_2		

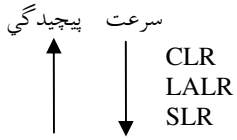
تعداد وضعیت های جداول تجزیه SLR و LALR هر گرامری دقیقا یکسان است لیکن تعداد وضعیت های جدول تجزیهCLR گرامرها به مراتب بزرگتر از جداول SLR و LALR است. تنها عیب جزئی روش LALR نسبت به CLR این است که خطا های نحوی را ممکن است قدری دیرتر کشف کند. البته هیچ یک از این سه روش پس از رسیدن به یک توکن غلط آن را به داخل انباره انتقال نخواهد داد. در CLR پس از رسیدن به یک توکن خطا، دیگر حتی عمل کاهش نیز انجام نخواهد شد، لیکن در مورد SLR و LALR ممکن است خطا پس از اجرای چند عمل کاهش اضافی کشف شود. با این حساب CLR سریعتر است.

اگر گرامر CLR باشد آیا می توان نتیجه گرفت که گرامر LALR است؟

خیر زیرا در جدول تجزیه LALR ممکن است تداخل r-r داشته باشیم.

اگر گرامری SLR باشد آن گرامر CLR و هم LALR است

مقایسه سرعت و پیچیدگی



تحلیل معنایی Type checking

به صورت پویا : در زمان اجرا عمل تحلیل معنایی انجام میشود.کامپایلر باید علامت خاصی را در لابه لای کد اجرا قرار دهد به عنوان مثال فرض کنید آرایه ۱۰ تایی تعریف کرده اید در برنامه طبق زیر داریم

```
Int A[10];          A[i]=5
```

در اینجا مقدار i در زمان اجرا مشخص میشود در این جا کامپایلر علامتی را قرار میدهد تا مشخص کند که مقدار i در حین اجرا تعیین میشود

به صورت ایستا:

۱- یکسان بودن گونه(Type checking): اجازه نداریم آرایه را با متغییر جمع کنیم. عملگرها باید عملوند یکسان داشته باشند.

۲- یکسان بودن نام (واحد بودن نام): نمیتوان یک نام را دو جا تعریف کرد

۳- کنترل جریان: مثلا در swich در case باید Break داشته باشیم وگرنه خطا رخ میدهد

۴- بررسی ساختارهای تودرتو: مثلا در زبان Ada هر حلقه یک نام مخصوص برای خود دارد

```
For1(……)
    For2(……..)
    End for2
End for1
```

مثال:

اگر a یک متغییر از نوع آرایه یک بعدی با اندیس های ۱ تا۱۰ و I یک عدد صحیح باشد خطای خارج از محدوده(sub scrip out of vong) که در دستورات B:=A[i] ؛ I=11 وجود دارد در حالت کلی در چه زمانی و توسط چه برنامه ای قابل کشف است؟

۱- زمان اجرا توسط سیستم عامل

۲- زمان کامپایل توسط تجزیه کننده دستوری

۳- زمان کامپایل توسط تجزیه کننده معنایی

۴- زمان اجرا توسط خود برنامه حاوی دستورات فوق

جواب: گزینه 4 درست است، اگر B := A[11] بود آنگاه خطا در مرحله کامپایل توسط تحلیل گر مفهومی کشف می شد

نکته: پیش پردازنده ها در زمان کامپایل مشخص میشوند.

سوال :کدام یک از گزینه ها در خصوص گرامر نوع مقابل صحیح است؟

1) $S \to aACb$

2 – 3) $A \to b \mid \lambda$

3 – 4) $C \to cC \mid \lambda$

۱)گرامر LL(1) نیست، SLR(1) هم نیست

۲) گرامر LL(1) نیست ولی SLR(1) هست

۳) گرامر LL(1) و SLR(1) است

۴) گرامر LL(1) هست ولیSLR(1) نیست

جواب. گزینه 1 درست است

	S	A	C
First	{a}	{b,ε}	{c,ε}
Follow	{$}	{c,b}	{b}

جدول پارسینگ LL(1) به شکل زیر خواهد شد.

	a	b	c	$
S	1			
A		2/3	3	
C		4	4	

پس گرامر بالا LL(1) نیست ، اگر دیاگرام و جدول تجزیه این گرامر را رسم کنیم ملاحظه می شود که ، SLR(1) هم نیست

مثال: گرامر زیر برای یک پارسر بالا به پایین پیشگو کننده مناسب نیست مشکل را مشخص کنید و گرامر را طوری بنویسید که برای بالا به پایین پیشگو کننده مشخص کنید و بعد برای گرامر جدید جدول تجزیه LL(1) را بسازید.

$L \to Ra \mid Qba$

$R \to aba \mid Caba \mid Rbc$

$Q \to bbc \mid bc$

اگر گرامری بازگشتی چپ داشته باشد آیا LL(1) هست یا نه؟

خیر چون برای LL(1) بودن باید مبهم نباشد و بازگشتی چپ و.. نداشته باشد

نکته) اگر گرامری مبهم نباشد و فاکتورگیری چپ روی آن اعمال شود وبازگشتی راست دارد LL(1) است و اگر گرامری بازگشتی راست داشته باشد ممکن استLL(1) باشد یا نباشد

نکته: اگر گرامرCLR نباشد حتما LALR نیز نیست