فصل چهارم

تجزیه بالا به پایین

۱ـ٤ تحلیلگر ذهن

عملکرد تجزیه بالا به پایین بر مبنای روش تفکر مغز آدمی برای تحلیل نحوی جملات استوار است. فردی فارسی زبان در مقابل شما قرار می‌گیرد . می‌خواهد با شما صحبت کند به دهان وی می‌نگرید. پیش بینی می‌کنید که جمله فارسی می خواهد بیان کند. جمله فارسی سر ترم جملات در دستور العمل زبان فارسی است زیرا ، هر گفته ای در ربان فارسی باید جمله فارسی باشد. بنابراین مغز پیش بینی می‌کند که جمله خارج شده از دهان سر ترم گرامر جملات در زبان فارسی باید باشد.

حالا فرض کنید که فرد ' اگر' از دهانش خارج شود. بلافاصله تحلیلگر نحوی مغز به سراغ مجموعه لغات یا در اصطلاح ترم های پایانی ای می‌رود که می توانند جملات فارسی را آغاز کنند. کلمه ' اگر' می تواند آغاز کننده یک جمله فارسی باشد . در اصطلاح مجموعه ترم های پایانی که می‌توانند آغاز کننده یک ترم باشند را مجموعه سرآغاز یا مجموعه First برای آن ترم گویند.

پس از اطمینان از اینکه لغت 'اگر' درمجموعه سر آغاز یا First جمله فارسی است باید مشخص نمود که کدام گسترش از گسترشهای متفاوت جمله فارسی با کلمه اگر آغاز می‌شود. پاسخ جملات سوالی است. کلمه اگر متعلق به مجموعه سرآغاز یا First جملات سوالی است. حالا طبق گرامر جملات سوالی پس از کلمه ' اگر' انتظار شنیدن یک شرط می‌رود. باز هم بر طبق گذشته تحلیلگر نحوی ذهن کلمه بعدی را گرفته در مجموعه سرآغاز شرط آنرا جستجو می‌کند. و به این ترتیب مراحل ادامه می یابد.

۲ـ٤ ایجاد الگوریتم تحلیل نحوی بر مبنای عملکرد ذهن

بطور خلاصه در بخش قبل ملاحظه نمودید که برای انجام عمل تحلیل نحوی مغز به صورت زیر عمل می‌نماید:

۱. انتظار سر ترم گرامر زبان را در آغاز دارد. سر ترم گرامر زبان فارسی جمله فارسی است.
۲. لغت اولی دریافت می‌شود. در بخش قبل در حالی که انتظار جمله فارسی را تحلیلگر ذهن می‌داشت لغت ' اگر ' دریافت شد.
۳. در داخل مجموعه First برای ترم مورد انتظار بدنبال لغت دریافت شده می گردد.
۴. در صورت یافتن لغت، در داخل مجموعه First برای گسترش های متفاوت ترم میانی ، ترم‌های بعدی بررسی می گردند . (در مثال فوق جمله شرطی با لغت ' اگر' آغاز می‌شد).
۵. حالا با مشاهده ترم پایانی دریافت شده در گرامر، طبق گرامر ترم مورد پیش بینی را مشخص می‌کند. در مثال فوق با یافتن کلمه ' اگر' طبق گرامر انتظار مشاهده یک شرط می‌رفت. لذا، لغت بعدی دریافت شده و در صورت عدم خاتمه جمله از مرحله ۲ عملیات تکرار می‌شود.

برای نمونه به گرامر زیر توجه کنید.

Statement› IfSt- WhileST- ForSt- CaseSt- CompoundSt|AssignmentSt- Callst
IfSt › IF Expression THEN Statement ElsePart
ElsePart › ELSE Statement - ε
WhileSt› while expression DO Statement
ForSt › FOR Variable : =Expression TO Expression DO Statement
CaseSt › CASE Expression OF Expression DO Statement
CaseParts › Case Labels Statement OtherParts
OtherParts › ;CaseParts - ε
Caslabels › Contant Contants
Contants › Caselabels - ε
Contant › string - Identifier - Number
CompoundSt › BEGIN Statement END
Statement › Statement - OtherSts
OtherSts › ; Statement - ε
AssignmentSt› id ;= Expression
CallSt› id'('ActualParms ')'
OtherParams › Expression OtherParams
OtherParams › ,OtherParams - ε
Expression › id - NO

برای تجزیه جمله

If A then C := 1

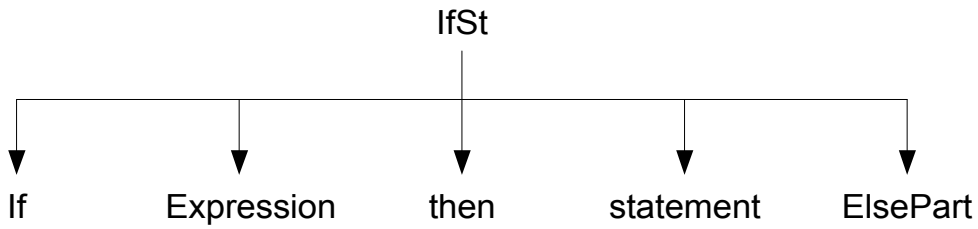
بر اساس عملکرد ذهن به صورت زیر عمل می‌شود.

- بر اساس گرامر در ابتدا انتظار مشاهده سر ترم گرامر یعنی Statement می‌رود.
- اولین لغت یعنی if از تحلیلگر لغوی دریافت می‌شود.

- تحلیلگر نحوی به جستجوی لغت if داخل مجموعه لغات آغاز کننده ترم مورد انتظار یعنی

First(statement) = First(IfSt) + First(WhileSt) + First(ForSt) +
First(CaseSt) + First(CompoundSt) + First(AssignmentSt) + First(CallSt)
= [S_If,S_While,S_For, S_Case, S_BEGIN, S_Id ;

- با مشاهده نوع لغت If در مجموعه First(statement) تحلیلگر لغوی بدرون مجموعه سراغاز برای گسترشهای متفاوت statement می‌نگرد و لغت if را در جمله (First(IfSt پیدا می‌کند.بنابر این تحلیلگر تشخیص می‌دهد که جمله مورد نظر یک جمله شرطی If است. درخت زیر بر اساس گسترش موجود برای جمله IfSt بر طبق گرامر، ایجاد میشود:

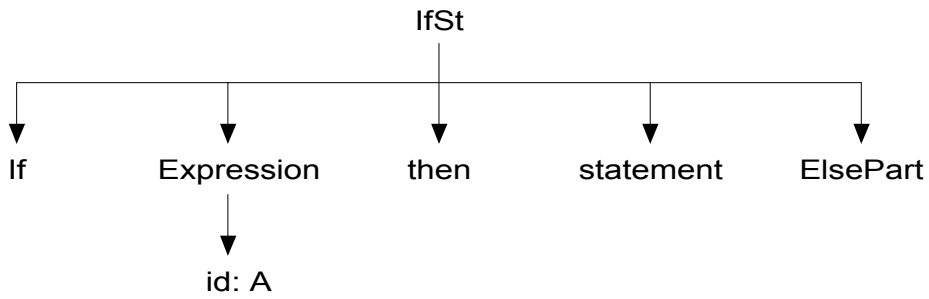


- حالا پس از مشاهده if، بنابر گرامر جمله شرطی ifSt در ورودی انتظار مشاهده Expression می‌رود. از تحلیلگر لغوی بعدی یعنی A دریافت می‌شود. درون مجموعه سر آغاز عبارات یعنی:

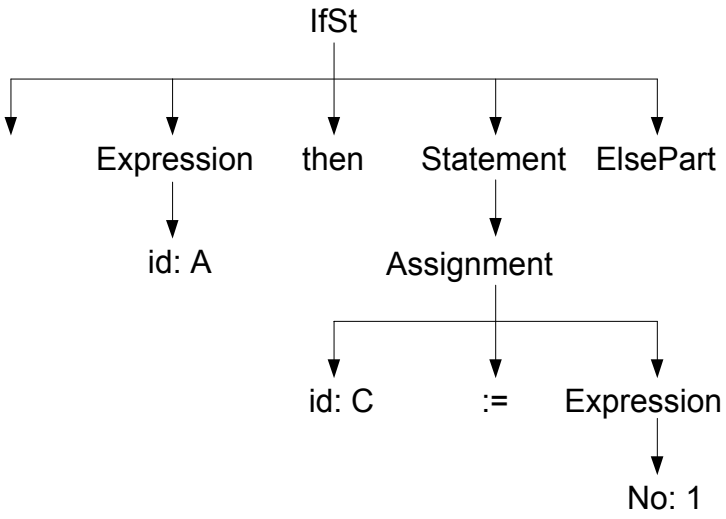
First(Expression)= [S_Id]

نوع لغت A یعنی S_Id وجود دارد. بنابر این تحلیلگر نحوی به کار خود ادامه می‌دهد.

- حالا طبق گرامر انتظار دیدن Then در ورودی می‌رود . تحلیلگر لغوی لغت بعد از A را از ورودی می‌خواند. این لغت همانگونه که طبق گرامر انتظار میرفت لغت Then می‌باشد. بنابر این تا این لحظه درخت تجزیه زیر توسط تحلیلگر نحوی بنا شده است:



حالا تحلیلگر پیش بینی مشاهده یک Statement را بر اساس گرامر می‌نماید. از تحلیلگر لغوی لغت بعدی را تحلیلگر نحوی دریافت می‌کند. لغت C از نوع S_Id را تحلیلگر نحوی دریافت می‌کند. این لغت در مجموعه First(statement) وجود دارد اما، نکته اینجا است که هر دو گسترش CallSt و Assignment با Id آغاز می‌شوند. لذا، تحلیلگر نحوی نمی‌تواند بر اساس ترم پیش بینی id تصمیم بگیرد که کدام گسترش باید انتخاب شود. بنابر این ترم بعدی را دریافت می‌کند ترم بعدی در مثال فوق '= :' است. حالا می‌توان تصمیم گرفت که گسترش AssignmentSt باید انتخاب شود. نهایتاً تجزیه بصورت زیرا خواهد بود.



۳ـ٤ نتیجه تحلیل عملکرد ذهن

از مطالب ارائه شده در بخش قبلی می‌توان سه نتیجه به شرح زیر گرفت:

اولاً در هر مرحله تحلیلگر پیش بینی می‌کند که ترم بعدی چه باید باشد. برای نمونه در ابتدا پیش بینی سر ترم گرامر را تحلیلگر نحوی می‌نمود. به این نوع تحلیلگر ها در اصطلاح تجزیه گرهای پیش‌بینی‌کننده یا Predictive Parsers گویند.

ثانیاً عمل خواندن لغات از چپ به راست صورت می‌گیرد. در مثال فوق ابتدا سمت چپ ترین لغت یعنی if از ورودی خوانده شد و عمل خواندن از چپ به راست ادامه پیدا کرد. لغت خوانده شده از ورودی را در اصطلاح ترم پیش بینی یا Look Ahead گویند زیرا، بر اساس این لغت است که تحلیلگر پیش بینی می کند ترم بعدی چه باید باشد. برای نمونه بر اساس ترم پیش‌بینی if تحلیلگر نحوی تصمیم گرفت که گسترش InSt را برای Statement باید انتخاب نماید.پس پیش‌بینی می‌شود مشاهده جمله ifSt در ورودی شد.

ثالثا با در دست داشتن یک یا بیشتر از ترم های پیش بینی تحلیلگر می‌تواند تصمیم گیری کند که کدام گسترش از گسترشهای متفاوت ترم مورد انتظار را باید انتخاب نماید. برای نمونه در بالا با در دست داشتن

یک ترم پیش بینی If تحلیلگر تصمیم گرفت که گسترش را برای ترم مورد پیش بینـی یعنـی Statement باید انتخاب کند اما، در هنگامیکه در ورودی ترم پیش بینی از نوع 'id' ظاهر شد چـون دو گسـترش متفـاوت Statement یعنی AssignmentSt و CallSt هر دو با ترم 'id 'آغـاز می‌شـدند، تحلیلـگر نمی‌توانسـت تصمیم بگیرد که کدامیک را انتخاب کند. در این حالت، با در دست داشتن دو تـرم پیـش بینـی' id 'و ' =:' تحلیلگر توانست تصمیم بگیرد که کدام گسترش باید انتخاب شود .

اصولاً عمل تجزیه بالا به پایین از سر ترم گرامر آغاز و به ترم های پایانی درون جمله یا برنامه مورد کامپـایل خاتمه می یابد. برای انجام عمل تجزیه بالا به پایین فرم کلی جملات یا به عبارت دیگر گرامر زبـان را انقـدر محدود می کنند که، با در دست داشتن k ترم پیش بینی از متن برنامـه مـورد کامپـایل بتـوان عمـل تحلیـل نحوی را بدرستی انجام داد.

تجزیه گرهای پیش‌بینی‌کننده یا Perdicitive Parsers با در دست داشتن یک یا چند ترم پایانی بعدی در ورودی، قادر به انجام عمل تحلیل نحوی هستند. گرامرهای مورد استفاده برای این روش تجزیه را اصطلاحـاً LL(K) یا Left Lookahead گویند. منظور این است که می‌توان با استفاده از قواعد گرامر و بـا در دسـت داشتن k ترم پایانی بعدی در ورودی یا در اصطلاح k ترم پیش بینی، عمل تجزیه بالا به پایین را انجام داد.

٤ـ٤ گرامرهای LL(1)

تجزیه گرهای پیش‌بینی‌کننده برای تجزیهٔ بالا به پایین با در دست داشتن یک یـا چنـد تـرم بعـدی در ورودی باید قادر به تشخیص این باشند که:

اولاً: جملهٔ مورد نظر تا این مرحله از لحاظ گرامری صحیح است.

ثانیاً: چه ترمها ی میانی مورد انتظار هستند.

ثالثاً : چه ترمهای پایانی در سر ورودی می‌توانند ظاهر شوند.

چنانچه با در دست داشتن حداکثر k ترم بعدی از ورودی یا بعبارت دیگر بـا در دسـت داشـتن حداکـثر k تـرم پیش‌بینی بتوان عمل تجزیه قابل پیش بینی را بر روی یک گرامر به انجام رساند،آن گرامر را LL(k)گویند.
چنانچه با در دست داشتن یک ترم پایانی بعدی از چپ به راسـت در ورودی بتـوان تشـخیص داد کـه از بیـن گسترش های متفاوت برای یک ترم میانی کدامیک باید انتخاب شوند، گرامر را LL(1) گـویند.
برای نمونه گرامر زیر را در نظر بگیرید:

S ⟶ I| W| A| P| C
I ⟶ if B do S E
B ⟶ id D
D ⟶ R id | e
R ⟶ < | > | =
W ⟶ While B do S
A ⟶ id := No
P ⟶ id '('M')'
M ⟶ e | id
C ⟶ {' 'T' '}'
T ⟶ S G
G ⟶ : T| e
E ⟶ else S| e

این گرامر LL(1) نیست، زیرا برای نمونه با دیدن id یا شناسه نمی توان مشخص نمود که از بین قواعد مختلف برای گسترش S کدامیک باید انتخاب شود. اصولاً جهت تجزیهٔ بالا به پایین باید پس از خواندن یک ترم پایانی از ورودی مشخص نمود که آیا ترم پایانی خوانده شده متعلق به مجموعهٔ First برای ترم پایانی یا میانی مورد انتظار است یا خیر ؟ سپس باید مشخص نمود که کدام گسترش از گسترش های متفاوت ترم میانی مورد انتظار، با ترم پایانی خوانده شده، آغاز می شوند. به عبارت دیگر باید مشخص نمود که ترم پایانی خوانده شده، متعلق به مجموعهٔ First برای کدامیک از گسترش های مختلف ترم میانی مورد نظر است.

بنابراین اگر قرار باشد با داشتن یک ترم پیش بینی در هر مرحله از تجزیه بتوان مشخص نمود که کدام گسترش برای ترم میانی A با گسترش های

A› A1| A2| A3| …| An

باید انتخاب شود. آنگاه باید اشتراک مجموعهٔ First برای هر دو گسترش متفاوت از A تهی باشد یا بعبارت دیگر باید رابطهٔ:

$$\forall i, j \varepsilon 1..n, i \neq j \Rightarrow first(Ai) \cap first(Aj) = \phi$$

برقرار باشد. در غیر اینصورت فرض کنید که برای نمونه

$$i, j \varepsilon 1..n, \neq j, first(Ai) \cap first(Aj) = \{a\}$$

آنگاه با مشاهدهٔ ترم پایانی a در ورودی تحلیلگر نمی‌تواند بلافاصله تصمیم بگیرد که آیا گسترش A به Ai باید انتخاب شود، یا گسترش A به Aj .

پس بطور خلاصه می‌توان گفت :

یک گرامر LL(1) است اگر اشتراک مجموعه First هردو گسترش متفاوت برای هر ترم میانی متعلق به آن گرامر تهی باشد.

برای نمونه درخت تجزیه را برای جملهٔ:

{ a := 5 ; if b do f( ) }

با روش بالا به پایین می‌توان بصورت زیر ایجاد نمود:

۱- تحلیلگر لغوی فراخوانی می‌شود.

۲- لغت '{' توسط تحلیلگر لغوی به تحلیلگر نحوی برگردانده می‌شود.

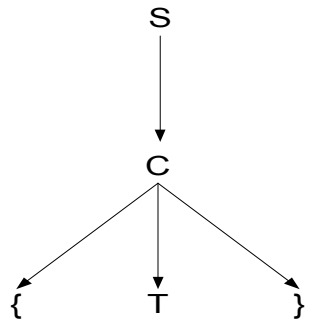
۳- تحلیلگر نحوی که در انتظار مشاهده S در ورودی است ، ابتدا می‌بایست مطمئن شود که ترم خوانده شده یعنی '{' آیا متعلق به مجموعهٔ First (S) است.

'{' ε first (S) = { if , while , id , '{' }

سپس باید مشخص شود که '{' به مجموعهٔ First برای کدام گسترش از گسترش های متفاوت S تعلق دارد. چون:

'{' ε first (C) = { '{' }

پس درخت تجزیهٔ زیر ایجاد می‌شود :

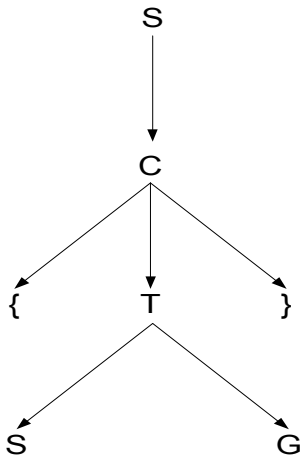


۴-با مشاهدهٔ } در ورودی دو مرتبه مراحل ۱ تا ۴ بشرح زیر تکرار می‌شود.

- لغت بعدی a می باشد که از نوع id است.

$$id \in first(T) = first(S) = \{if, while, id, '\{'\}$$

بنابراین گسترش S به T انتخاب شده درخت تجزیه بصورت زیر خواهد بود.



اما در این مرحله مشاهده می‌شود که دو گسترش مختلف A › S و P › S هردو با یک ترم id آغاز می شوند، بنا بر این دراین مرحله به علت L L(1) نبودن گرامر نمی‌توان با در دست داشتن ترم پیش‌بینی a تصمیم قطعی در مورد گسترش S گرفت.
همانگونه که در بالا توضیح داده شد، برای انجام تجزیهٔ بالا به پایین نیاز به در دست داشتن مجموعهٔ First برای ترمهای گرامر است. روش محاسبهٔ مجموعهٔ سرآغاز یا مجموعهٔ First در بخش بعدی توضیح داده خواهدشد.

٥ـ٤ مجموعه های سرآغاز و پیرو

مجموعهٔ سرآغاز برای یک ترم شامل مجموعه ترم های پایانی است که گسترش های متفاوت و جملات مشتق از آن ترم را می توانند آغازکنند. در حالت کلی برای ترم A مجموعهٔسرآغاز بصورت زیر محاسبه می‌شود:

- چنانچه برای A گسترشی بصورت A › a α وجود داشته باشد، آنگاه :

$$First(A)=\{a\}$$

- چنانچه برای A گسترشی بصورت A › B α وجود داشته باشد ، آنگاه :

$$First(A)=First(B)$$

- چنانچه برای A گسترشی بصورت A › B α وجود داشته باشد و B › δ |λ، آنگاه :

$$First\ (A)=First\ (\delta)+First\ (\alpha)$$

واضح است که اگر B تهی یا λدر نظر گرفته شود، آنگاه قاعده A › B α در واقع بصورت A › α تبدیل می‌شود. بنابراین First(A) شامل First(α) هم می‌شود.

- چنانچه برای A گسترشی بصورت A › X1 X2 X3 ... Xn وجود داشته باشد و برای آنگاه:

$$\forall 1\le j\le i\quad X_j\to\delta_j\,|\,\lambda$$

X1 تا Xi قاعده تهی هم وجوداشته باشد یا به عبارت دیگر

First (A) = First(X1) +First (X2) + ... +First (Xi+1)

واضح است که در قاعده A › X1 X2 X3 ... Xn اگر X1 وجود داشته باشد،آنگاه

First (A) = First (X1)

وگرنه در صورتی که X1 تهی یا λ باشد،آنگاه قاعده بصورت زیر خواهد بود :

A › X2 X3 ... Xn

به این ترتیب:

First (A) = First (X1) +First (X2)

حالا اگر X1 وX2 هردو تهی باشند آنگاه

First (A) = First (X1) +First (X2) +First (X3)

به همین ترتیب می‌توان نهایتاً نشان داد که :

$$First(A) = \Sigma first(X_j),\ 1 \le j \le i+1$$

چنانچه برای ترم A گسترش تهی نیز وجود داشته باشد یا به عبارت دیگر در حالت کلی قواعد مربوط به ترم A به صورت $A \rightarrow \alpha \mid$ باشد، آنگاه با مشاهدهٔ یک عنصر متعلق به First (α) واضح است که گسترش $A \rightarrow \alpha$ باید انتخاب شود. اما چه عنصری باید ظاهر شود تا در حالیکه انتظار مشاهده A می رود، بتوان دریافت که گسترش $A \rightarrow$ باید انتخاب شود.واضح است که اگر ترم مورد انتظار یعنی A در ورودی ظاهر نشود باید ترمی که پس از A انتظار مشاهده آن در ورودی می رفت ظاهر شود. برای نمونه به گرامر زیر توجه کنید :

LabeldSt	⟶	Label Statement
Label	⟶	No: | ε
Statement	⟶	AssignmentSt | IfSt | WhileSt
AssignmentSt	⟶	id := Expression
WhileSt	⟶	WHILE Expression DO Statement
ifSt	⟶	IF Expression Do Statement

طبق گرامر فوق جملات دارای برچسب یا Label و بدون برچسب هستند. در آغاز کار طبق گرامر فوق جهت مشاهده LabelSt ابتدا انتظار مشاهده یک Label در ورودی می‌رود. بنابراین انتظار می‌رود که اولین ترم در ورودی یک عدد یا No باشد زیرا :

$$No \in First(Label)=\{No, \varepsilon\}$$

اما، اگر ترم خوانده شده از ورودی از نوع No نباشد، مشخص است که Label در ورودی وجود نداشته است. بنابر این انتظار می رود که ترم خوانده شده آغاز کنندهٔ ترم بعد از Label یعنی Statement باشد. بنابراین چنانچه ترم مورد انتظار دارای گسترش تهی هم باشد، در صورتی گسترش تهی انتخاب می شود که ترم پیش بینی متعلق به مجموعهٔ First ترم بعدی آن در سمت راست قاعدهٔ مورد نظر باشد. مجموعه First ترم هایی که پس از ترم میانی A در سمت راست قواعد متفاوت ظاهر می شوند را در اصطلاح مجموعهٔ پیرو یا Follow برای آن ترم میانی می‌گویند. برای نمونه در گرامر فوق:

$$Follow(Label) = First(Statement) = \{S\_Id, S\_If, S\_While\}$$

اصولاً برای بدست آوردن مجموعه پیرو به روشهای زیر عمل می‌شود:

- در صورتی که ترم A در سمت راست یک قاعده به صورت زیر ظاهر شود

$B \rightarrow A\ \alpha$

آنگاه :

$$Follow(A) = First(\alpha)$$

- درصورتیکه ترم A در سمت راست یک قاعده به صورت زیر ظاهر شود

$B \rightarrow \alpha\ A$

آنگاه مجموعهٔ پیرو برای A شامل مجموعهٔ پیرو B می‌باشد

$$Follow(B) \subseteq Follw(A)$$

اما بالعکس صادق نیست. علت این است که هرجایی که B در سمت راست قواعد ظاهر شود می‌توان به جای آن αA را قرار داد اما، بالعکس صادق نیست.

- همواره در مجموعهٔ پیرو برای سرترم گرامر علامت خاتمهٔ فایل یا End of File وجود دارد. زیرا ورودی تحلیلگر نحوی یک فایل است. برای نمونه یک برنامه C را در نظر بگیرید. این برنامه به عنوان یک جملهٔ ورودی به کامپایلر داده می‌شود. چون برنامه در فایل است در انتهای آن علامت خاتمه فایل قرار می‌گیرد. برنامه ورودی در اینجا از سرترم گرامر یعنی برنامه C مشتق می‌شود. پس بعد از سرترم گرامر علامت خاتمه فایل قرار می‌گیرد. بطور کلی هر جمله ای که در ورودی کامپایلر قرار می‌گیرد باید از سرترم گرامر مربوطه مشتق شود. چون جمله ورودی در داخل یک فایل است، لذا همواره در مجموعه پیرو سرترم گرامر، علامت خاتمه فایل وجود دارد. علامت خاتمه فایل را بطور اختصاصی معمولاً با $ مشخص می‌کنند.

بنابراین مشاهده می‌کنید چنانچه مجموعهٔ First برای یک ترم شامل عنصر تهی λ باشد، آنگاه باید عنصر λرا از داخل مجموعه حذف و به جای آن عناصر مجموعهٔ پیرو را به داخل مجموعه First افزود. به این ترتیب:

First (Label) = { No ,λ } = First (Label) +Follow (Label)
= { No , S_Id , S_If , S_While }

مشاهده می‌کنید مجموعهٔ Follow برای یک ترم شاخص گسترش تهی برای آن ترم است. بنابراین می‌توان نتیجه گرفت:

یک گرامر LL(1) است اگر اشتراک مجموعه سرآغاز برای هر دو گسترش هر ترم متعلق به آن گرامر، تهی باشدو چنانچه آن ترم دارای گسترش تهی باشد اشتراک مجموعهٔ سرآغاز و پیرو آن ترم باید تهی باشد.

برای نمونه گرامر زیر LL(1) نیست:

LabeldSt ⟶ Label Statement
Label ⟶ No: | e
Statement ⟶ AssignmentSt | ifSt | WhileSt
AssignmentSt ⟶ id := Expression
WhileSt ⟶ WHILE Expression DO Statement
ifSt ⟶ iF Expression DO Statement

علت LL(1) نبودن گرامر فوق این است که :

First (Label) ? Follow (Label) ={S_Id}

بنابر این با مشاهده Id در ورودی نمی توان تصمیم گرفت که آیا گسترش : Id › Label باید انتخاب شود یا گسترش λ › Label .

برای بدست آوردن مجموعهٔ First می توان از ماتریس تجانس نیز استفاده نمود. برای نمونه به گرامر زیر توجه کنید :

P ⟶ DB| B
B ⟶ Ae
A ⟶ bS|S
S ⟶ aD| Ba
D ⟶ aD| d

چنانچه در حالت کلی ترم A با ترم B آغاز شود یک رابطه بطول یک بین ترم A یا B وجود دارد. می توان به این ترتیب یک گراف واسط ایجاد نمود. اکنون مسئله، به دست آوردن روابط بین هـــر دو گره در گراف اسـت. یعنی هدف بدست آوردن وجود روابط با هر طولی بین هردو گره در داخل این گراف جهت دار روابط است. می توان از ماتریس تجانس استفاده نمود . به این ترتیب که اگر بین دوترم A وB یک رابطه آغاز شدن A توسـط B وجود دارد، در سطر مربوط به A و ستون مربوط به B در ماتریس مقدار یک قرار داده مـــی شـود. بـه ایـن ترتیب برای گرامر فوق ماتریس زیر ایجاد می شود :

جدول ۱

	A	B	D	P	S	A	B	D	E
A	1	0	0	0	1	0	0	0	0
B	1	1	0	0	0	0	0	0	0
D	0	0	1	0	0	1	0	1	0
P	0	1	1	0	1	0	0	0	0
S	0	1	0	0	1	1	0	0	0
A	0	0	0	0	0	1	0	0	0
B	0	0	0	0	0	0	1	0	0
D	0	0	0	0	0	0	0	1	0
E	0	0	0	0	0	0	0	0	1

اکنون آنقدر ماتریس را در خودش باید ضرب نمود تا اینکه در دو تکرار متوالـــی مقـادیر مـاتریس بـا یکدیگر تفاوتی نکند. در این صورت ماتریس نهایی روابط First را مشخص می‌کند. باید توجه داشـــته باشـید که در هنگام ضرب دو ماتریس Boolean به جای ضرب از عمل " و " منطقی یا AND استفاده می شـود و به جای عمل جمع از OR منطقی استفاده می‌شود.

٤ـ٦ تبدیل گرامرها به فرم LL (١)

همانگونه که قبلاً نیز توضیح داده شد اگر هر دو گسترش متفاوت هر ترم کتعلق به یک گرامر نهایتاً با یک ترم مشترک آغاز نشوند آن گرامر LL(1) است. برای تبدیل گرامر به فرم LL(1) بعضاً می‌توان از روشی موسوم به فاکتور گیری چپ استفاده نمود .

١ـ٦ـ٤ فاکتور گیری چپ

چنانچه در حالت کلی برای ترم Aگسترشهایی بصورت زیر موجود باشد:

A › a α- a β - γ
First (a α) ∩ First (a β) = { a }

آنگاه:

با فاکتورگیری a از سمت چپ دو گسترش aβ و aα می‌توان گرامر را بصورت زیر بازسازی نمود:

A › a B- γ
B › α- β

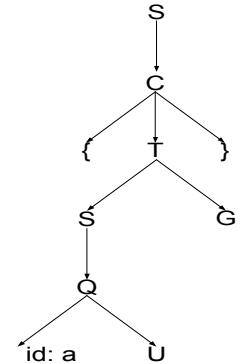
در فرم توسعه یافته می‌توان بدون استفاده از ترم کمکی B عمل فاکتور گیری را انجام داد و گسترشهای ترم میانی A را بصورت زیر تبدیل نمود :

A › a (α−β)γ

با استفاده از روش فاکتور گیری چپ می‌توان قواعد مربوط به سر ترم s از گرامر ارایه شده بخش قبلی را بصورت زیر تبدیل نمود:

S › I- W- Q- C
Q › id U
U › :=No - ‘(‘M‘)’

به این ترتیب اکنون می‌توان بکار تولید درخت تجزیه که در بخش قبل با مشاهده id یعنی a در ورودی



متوقف گردید ادامه داد زیرا، حالا با دیدن ترم پایانی a در ورودی بلافاصله گسترش U id › Q انتخاب میشود و درخت تجزیه به این صورت تبدیل می‌گردد.

حالا با فراخوانی تحلیلگر لغوی در ورودی = : ظاهر می‌شود که در اینجا بطور قطعی می‌توان گفت که گسترش No = : › U باید انتخاب شود. در حالت کلی اگر در گرامر قواعد بصورت خود بازگشتی چپ وجود داشته باشد ، باز هم گرامر LL(1) نمی‌تواند باشد .

۲ـ٦ـ٤ تبدیل قواعد خود بازگشتی چپ

وجود قواعد بصورت خود بازگشتی چپ مغایر با LL(1) بودن گرامرها می‌باشد. برای نمونه به قواعد زیر توجه نمایید :

A › A α- β

جهت نقض LL(1) بودن گرامر کافی است اثبات شود که:

First (A α) ∩ First (β)≠ ∅

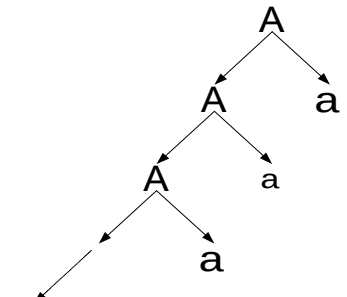
در اینجا واضح است که:

first (A α) ∩ first (β) = first(β)

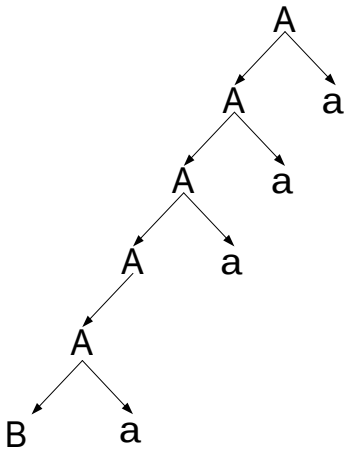
چرا که :

first (A α) = first (A) = first(β)

برای تبدیل قواعد مربوط به A بصورت LL(1) به این ترتیب میتوان استدلال نمود که دو قاعده A › A α -β نمایانگر فرم کلی رشته هایی است که با β آغاز و با صفر یا بیشتر α ادامه می‌یابند. بنابر این هر رشته با فرم کلی βααα……α از سر ترم A باید مشتق شود.



به این ترتیب برای رشته $\beta\alpha\alpha\alpha$ درخت تجزیه بصورت زیر خواهد بود.

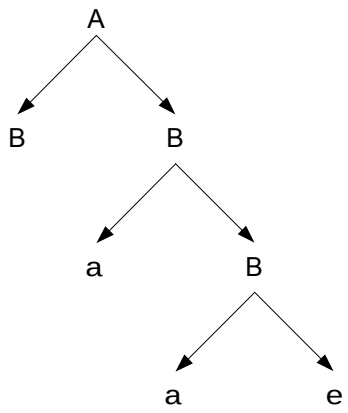


به همین ترتیب رشته β نیز از سر ترم گرامر مشتق می‌شود :

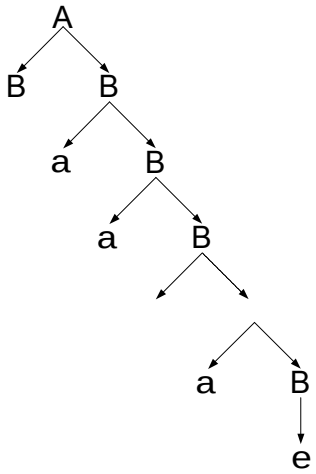
رشته های فوق را با گرامر زیر نیز می توان تولید نمود:

$A \rightarrow \beta B$
$B \rightarrow \alpha B - \lambda$

رشته $\beta\alpha\alpha\alpha$ را در نظر بگیرید. این رشته بر طبق گرامر فوق از سر ترم A مشتق می شود.



و به همین ترتیب هر رشته با فرم کلی βααα....α از سر ترم A مشتق می‌شـود. درخـت تجزیـه در حـالت کلی بصورت زیر است.



بنابر این می‌توان نتیجه گرفت که برای حذف خود باز گشتی چپ از قواعد با فرم کلی :

$A \rightarrow A\,\alpha - \beta$

می‌توان آنها را با قواعد معادل زیر جایگزین نمود :

$A \rightarrow \beta\,B$
$B \rightarrow \alpha\,B - \lambda$

در فرم توسعه یافته گرامر بصورت زیر تبدیل می‌شود :

$A \rightarrow \beta\,\{\alpha\}$

در فرم توسعه یافته آکولاد به مفهوم تکرار صفر یا بیشتر است.

مثال گرامر عبارات را بصورت LL(1) تبدیل کنید.

E ⟶ E+T| E-T| T

T ⟶ T*F| T/F| (E)

F ⟶ id| No| (E)

به قواعد E توجه نمایید. دو گسترش T+E و E-T با یک ترم مشترک E آغاز می‌شوند. به همین ترتیب دو گسترش متفاوت ترم میانی T نیز با یک ترم آغاز می‌شوند. پس از انجام عمل فاکتور گیری چپ گرامر توسعه یافته عبارات بصورت زیر تبدیل می‌شود:

E ⟶ E (+|-)T| T

T ⟶ T (*|/)F| (E)

F ⟶ id| No| (E)

حالا با حذف خود باز گشتی چپ، گرامر بصورت زیر تبدیل می‌شود.

E ⟶ T { (+|-) T}

T ⟶ F {(*|/) F}

F ⟶ id| No| (E)

در صورت وجود قواعد تهی با فرم کلی A ‹ ε در گرامر نیز ممکن است گرامر LL(1) نباشد.

۳ـ٦ـ٤ حذف قواعد تهی

چنانچه در گرامری قواعد تهی وجود داشته باشد، آن گرامر ممکن است LL(1) نباشد. برای تبدیل گرامر به فرم LL(1) باید ابتدا قواعد تهی را حذف نمود و سپس با استفاده از روشهای فاکتور گیری و حذف خود بازگشتی چپ گرامر را به فرم LL(1) تبدیل نمود. برای نمونه به گرامر زیر توجه نمایید:

LabelSt ⟶ Label Statement

Statement ⟶ F {(*|/) F}

AssignmentSt ⟶ id := Expression

CallSt ⟶ id '(' ActualParams ')' | id

ActualParams ⟶ Expression| Expression.param

Expression ⟶ id| No

در گرامر فوق برای ترم میانی Label یک گسترش تهی وجود دارد. لذا باید

First (Label) ∩ Follow (label) = ∅
First (Label) = { Id }
Follow (label) = First(Statement) = First(AssignmentSt) + First(CallSt) ={ Id }
First (Label) ∩ Follow (label) = { Id } ≠ ∅

بنابر این گرامر LL(1) نیست زیرا هنگامیکه انتظار مشاهده Label در ورودی می‌رود، با مشاهده id در ورودی نمی توان تصمیم گرفت که آیا گسترش' id : Label ' باید انتخاب شود و یا گسترش ε › Label برای تبدیل گرامر فوق به فرم LL(1) باید گسترش ε › Label حذف شود و هر جایی که از ترم میانی Label در سمت چپ یک قاعده استفاده شده است، یک بار Label را تهی و بار دیگر بصورت غیر تهی در نظر گرفت. بنابر این گرامر بصورت زیر تبدیل می‌شود. :

LabeledSt › Statement – Label Statement
Label › id

برای LabeleSt اکنون First(Statement) ∩ Follow(label) = {id } ≠ ∅ پس باید با استفاده از عمل فاکتور گیری چپ گرامر را به فرم LL(1) تبدیل نمود. برای این منظور باید ابتدا ترم ها را جایگزین کرد.

LabeledST › id := Expression – id ‘(‘ ActualParams ‘)’ – id : Statement

پس از فاکتور گیری از id قاعده بصورت زیر تبدیل می‌شود:

LabeledSt › id St
St › := Expression - ‘(‘ ActualParams ‘)’ - : Statement

در مورد Statement نیز باید عمل فاکتورگیری چپ را انجام داد:

Statement › Assignment - CallsSt

با جایگزینی Assignment و CallSt قاعده مربوط به Statement بصورت زیر تبدیل می‌شود:

Statement › id StTail
StTail › := Expression – id ‘(‘ ActualParams‘)’

مثال : گرامر زیر را به فرم LL(1) تبدیل کنید.

S › L A B
L › d - λ
A › d A - B a
B › B b - λ

٤ـ٧ ایجاد جدول تجزیه بالا به پایین

جدول تجزیه ساختاری برای تولید برنامه تحلیلگر نحوی برای گرامرهای LL(1) است. برای نمونه به گرامر زیر توجه نمایید.

S→dAB I BaB
A→dA I Ba
B→bB | λ

برای این گرامر باید ابتدا مجموعه های سر آغاز را برای ترمهای میانی بدست آورد. با استفاده از این مجموعه ها جدول تجزیه بدست می‌آید:

	a	b	d	$
S	BaB	BaB	dAB	
A	Ba	Ba	dA	
B		bB		

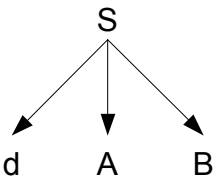
First (S) = {d} + First (B)
First (A) = {d} + First (B)
First (B) = {b} + Follow (B) ={b} +{a ,b,$ }

اکنون با استفاده از جدول تجزیه فوق براحتی می توان عمل تجزیه بالا به پایین را انجام داد. برای نمونه جمله ddabbb را با استفاده از جدول تجزیه فوق می توان براحتی تجزیه نمود.برای این منظور از یک پشته به نام پشته تجزیه استفاده می‌شود:

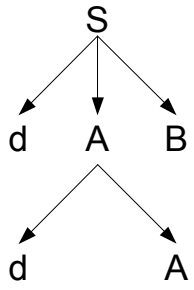
پشتـــهٔتجـــــزیه	ورودی	قاعده
$S	_ddabbb$	S ⟶ dAB
$BAd	_ddabbb$	
$BA	_dabb$	A ⟶ dAB
$BAd	_dabb$	
$BA	_abbb$	A ⟶ Ba
$BaB	_abbb$	B ⟶ e
$aB	_abbb$	
$B	_bbb$	B ⟶ bB
$bB	_bbb$	
$B	_bb$	B ⟶ bB
$bB	_bb$	
$B	_b$	B ⟶ bB
$bB	_b$	
$B	$	B ⟶ e

همانگونه که در جدول فوق مشاهده نمودید به انتهای رشته ورودی علامت خاتمه فایل یعنی $ افزوده می‌شود. عمل تجزیه از سر ترم گرامر آغاز می‌شود. و پس از سر تـرم انتظـار مـی‌رود کـه در ورودی علامـت خاتمه فایل ورودی یعنی × ظاهر شود. لذا بنابر این پیش بینی در آغاز کار رشته $S در داخل پشـته قـرار داده شده است.

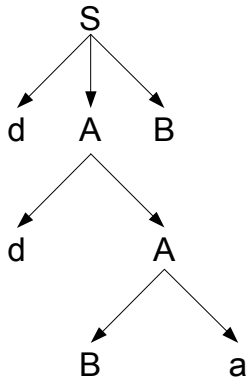
عمل تجزیه از ردیف مربوط به سر ترم آغاز می‌شود با مشاهده اولین ترم در جمله $ddabbb ترم در ورودی که ترم پایانی d است . یه ستون d درسطرS در داخل جدول ارجاع می‌شود.
گسترش DaBدر مکان (S,d) از جدول مشخص شده است لذا درخت تجزیه بصورت زیر ایجاد می‌شود :



اکنون ترم بعدی از جمله $ddabbb خوانده می شود. این ترم پیش‌بینی نیز dاست . با توجه به اینکه ترم پیش‌بینی هنوز d است . طبق درخت تجزیه انتظار مشاهده A در ورودی می‌رود. در مکان (A,d)از جدول تجزیه گسترش dA قرار دارد .درخت تجزیه زیر حاصل می‌شود.

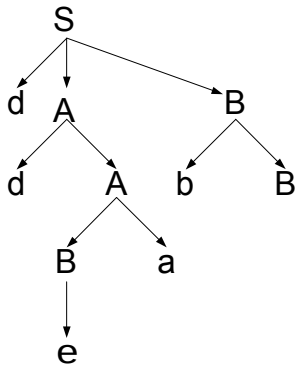


ورودی بعدی در جمله $ddabbb ترم پایانی a است. طبق درخت تجزیه انتظار مشاهده A در ورودی می‌رود درمکان (A,a) از جدول گسترش Ba قرار گرفته است. بنابر این درخت تجزیه بصورت زیر توسعه داده می‌شود:

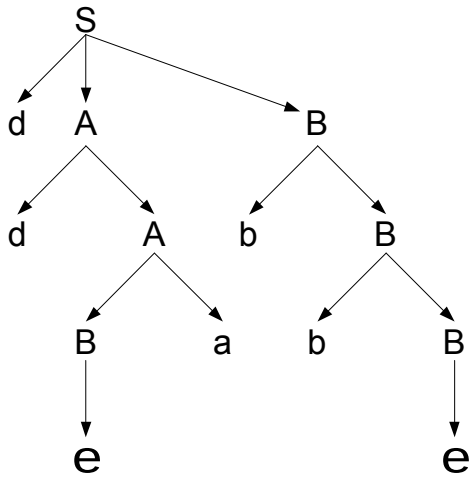


حالا با توجه به محتوی جدول تجزیه در مکان(B,a)باید گسترش λ B→ انتخاب شود . بنابر این طبق درخت تجزیه انتظار دیدن a در ورودی می‌رود. در این لحظه ترم پیش بینی همان طوری که انتظار می‌رود

a است. ورودی بعدی در جمله ddabbb ترم پایانی b ا ست . طبـق درخـت تجزیـه انتظـار مشـاهده B در ورودی می‌رود. در مکان (bوB) از جدول گسترش bB قرار گرفته است .بنابر این تا این مرحله درخت تجزیـه بصورت زیر می باشد :



به همین ترتیب اگر عملیات ادامه پیدا کند نهایتا درخت تجزیه بصورت زیر ایجاد می شود:



مثال ـ گرامر عبارت زیر ارائه شده تبدیل به فرم LL(۱) نموده و جدول تجزیه برای یک تحلیلگر پیش بینی کننده ایجاد کنید:

Expression ⟶ Expression Relop SimpleExp | SimpExp
RelOp ⟶ < | <= | <> |>= | = | IN
SimpleExp ⟶ SimpleExp '+' Term | SimpleExp '-' Term | SimpleExp Or Term | Term
Term ⟶ Term'/' Factor| Term'*' Factor|Term DIV Factor| Term AND Factor|Factor
Factor ⟶ Number | NOT Factor | '(' Expression ')' | variable
Variable ⟶ identifier | VarTal
VarTal ⟶ '[' Dim ']' . Variable I
Dim ⟶ Expression OtherDims
OtherDims ⟶ ',' Dim I

قواعد ارائه شده برای Expression دارای وضعیت خود بازگشتی چپ است :

Expression ⟶ Expression Relop SimpleExp | SimpExp

در حالت کلی A → Aa I b را می توان با قواعد معادل زیر جایگزین کرد :

A› βB
B› αB| ε

حالاExpression را مشابه A و Relop SimpleExp مشابه با aوSimpleExp را مشابه با β در نظر بگیرید.به این ترتیب قواعد مربوط به E بصورت زیر تبدیل می شوند :

Expression › SimpleExp E
E › SimpleExp | ε

در مورد SimpleExp ابتدا باید فاکتور گیری چپ نمود .

SimpleExp ⟶ SimpleExp '+' Term | SimpleExp '-' Term | SimpleExp Or Term | Term

پس از فاکتور گیری چپ قواعد بصورت زیر تبدیل می‌شوند.

SimpleExp › SimpleExp Simple | Term
Simple › '+' Term | '-' Term | Or Term

پس از حذف خود بازگشتی چپ قواعد بصورت زیر تبدیل می‌شوند :

SimpleExp › Term S
S › Simple S | λ
Simple › '+' Term | '-' Term | Or Term

در قاعده مر بوط به S می توان Simple را با گسترش آن جایگزین نمود .بنا بر این قواعد فوق بصورت زیـر ساده می‌شوند :

SimpleExp → Term S
S → Simple S I e
Simple → ' + ' Term I ' – ' Term I Or Term

بطور خلا صه گرامر عبارادر فرم LL(1) بصورت زیر می باشد :

Expression	→ SimpleExp E
E	→ RelOp Simple E
RelOp	→ <| <=| <> | >| >=| =| IN
SimpleExp	→ Term S
S	→ '+' Term S| '-' Term S| Or Term S|ε
Term	→ Factor T
T	→ '/' Factor T| '*' Factor T| DIV Factor T| AND Factor T|ε
Factor	→ Number| NOT Factor| '(' Expression ')'| Variable
Variable	→ identifier| VarTal
VarTal	→ '[' Dim ']' Variable
Dim	→ Expression OtherDims
OtherDims	→ ',' Dim

اکنون می توان مجموعه سرآغاز را برای ترمها محاسبه نمود :

First (Expression) = First (SimpleExp) = First (Term) = First Factor = {Number,NOT ,(, Identifier}
First (E) = First (RelOp) + {? } = < , <= ,= , <> , >= ,>, IN , ? }
Follow(E) = Follow (Expression) = First (OtherDims) + { ‘)’, $}
First (OtherDims) = { ‘,?’, } ={‘,’} +Follow(OtherDims) = = {‘,’} + Follow(Dim) ={‘,’,’]’}
First (E) = {< ,<= ,=, <> ,>= ,> ,IN } + { ‘)’ ,$,’,’ ,’l’ }
First (S) = { + , - ,OR , ? } ={+ , - ,OR } + Follow (S)
Follow (S) = Follow (Simple Exp) = First (E)
First (S) ={ + , - ,OR ,} + { < ,<= ,= ,<> ,>= ,>, IN ,’)’ ,$, ‘,’ ,’]’}

حالا با استفادهاز روابط فوق می توان جدول تجزیه بالا به پایین را به سادگی ا یجاد کرد.

مثال ـ گرامر زیر را به فرم LL(1) تبدیل نموده و جدول تجزیه برای آن ایجاد کنید .

S → SA l BdA
A →Aa l b
B → ba l Bd l ε

تبدیل گرامر به فرم LL(1) در فرم توسعه یا فته بسیارساده تر است. لذا گرامر بصورت زیر تبدیل می‌شود.

S → BdA {A}
A → b {a}
B → [ba] {d}

اما چون ∈#{d} = Follow(B) … First (B) گرامر LL(1) نمی‌با شد. بنا بر این بصورت زیر عمل باید نمود:

S → [ba] {d} dA {A}
A → b {a}

در مورد گسترش S, مشکل {d}d است. زیرا مشخص نیست که چه هنگامی می توان به d دومی رسید. اما واضح است که {d}d = d{d} است. بنابر این گرامر بصورت زیر تبدیل می‌شود :

S → [ba] d {d} A {A}
A → b {a}

حالا می توان جهت تولید جدول تجزیه گرامر را به فرم ساده تبدیل نمود. برای این منظور ابتدا [ba] را بـا B , و d{d} را با C و A{A} را با D باید جایگزیین نمود .

First(S) = First(B) = {b,a,d }
First(A) = {b}
First(f) = { a ,ee } Follow(F) = Follow(A) = First(D) = {b}
First(B) = { b, a , e } Follow(B) = First(C) = {d}
First(E) = { a , e } Follow(E) = Follow (B) = {d}
First(C) = {d }
First(G) = { d, ee } Follow(G) = Follow(C) = First(D) = {b}
First(D) = { b, ee } Follow(D) = Follow(S) = {$}

اکنون می‌توان جدول تجزیه را به سادگی برای این گرامر بدست آورد .

٨ـ٤ تجزیه گرهای کاهینه بازگشتی

تجزیه گرهای کاهینه بازگشتی یا Recursive descent parser را می تـوان بـرای گرامرهـای LL(1) مورد استفاده قرار داد. در این روش برای هر ترم میانی و سر ترم یک روال یا زیر برنامه به همان نـام ایجـاد می‌شود. به این ترتیب قواعد گرامر را عیناً با استفاده از توابع خود بازگشتی تبدیل به کد بر نامه می نمایید. لذا مزیت این روش سادگی ایجاد و خوانایی کد برنامه تجزیه‌گر است.

در ساختار کلی یا در واقع بر نامه اصلی برای ای ن نوع تحلیلگر پس از انجام عملیات اولیه به نام init تحلیلگر لغوی به نام NextSymbol فراخوانده می شود. تـا نـوع اولیـن تـرم پیـش بینـی در یـک متغـیر سراسری به نام CurrentSymbol قرار گیرد. سپس روال تعیین شده برای سر تـرم گرامـر کـه در زیر ProgramX نامیده شده مورد فراخوانی قرار می‌گیرد. بنا بر این بدنه اصلی برنامه تحلیلگر کاهینه بـازگشـتی بصورت زیر است:

```
Bebin
    Init;
    NextSymbol
  ProgramX
End
```

Current Symbol متغیری سراسری از نوع شمارش پذیر Symbols استکه قبل از این برای تعیین نـوع لغات توسط تحلیلگر لغوی مشخص شد. نوع Symbol حاوی انواع لغات مجود در زبان مورد نظر است. برای نمونه:

```
Type
        Symbols=(S_if, S_While, S_Repeat, S_For, S_Case, S_Then, S_Else,
        S_Do, S_Program, S_Uses, S_Interface, S_Unit, S_Begin, S_End,
        S_Label, S_Const, S_Type, S_Var, S_Procedure, S_Function,
        S_Integer, S_Real, S_Char, S_Array, S_Record, S_Pointer, S_It,
        S_Gt, S_Eq, S_Le, S_Ge, S_Ne, S_Add, S_Sub, S_Or, S_Mul, S_Div,
        S_And, S_Id, S_No, S_Not, S_Comma, S_Colon, S_Semicolon,
        S_Dot, S_OpBracket, S_ClBracket, S_OpCurlyB, S_ClCurlyB,
        S_OpSquB, S_ClSquB);
```

```
Var
        CurrentSymbol: Symbols;
```

پس از اینکه تحلیلگر لغوی اولین لغت را از بر نامه ورودی تشخیص و در داخل Curent Symbol قرار داد می بایست روال ProgramX فراخوانی شود. در واقع ProgramX نام سر ترم گرامر است چرا که در این روش برای هر ترم میانی و سر ترم روالی به همان نام نوشته مـی شـود. روال ProgramX بـر اسـاس قاعده مربوطه نوشته شده است. قاعده مربوطه بهProgramX در زیر با یک جملـه تفسـیری Comment مشخص شده است:

```
(* Program  X  →  Program  id  BlockBody '.' *)
Procedure  { روال تشخیص سر ترم کرامر}
programX ;
    Begin
Expect ( S_ program ) ;      { انتظار مشاهده S_Program در ورودی می رود}
Expect ( S_ id);             {  انتظار مشاهده S_Id در ورودی می رود  }
Expect  (S _ Semicolon ) ; { انتظار مشاهده S_Semicolon در ورودی می رود}
BlockBody ;   {فراخوانی روال ترم میانی BlockBody برای تشخیص بدنه برنامه}
Expect ( S_dot) ;            { انتظار مشاهده S_Semicolon در ورودی می رود}
End;
```

همانطوری که مشاهده می‌شود اکنون در ابتدای روال ProgramX بنا بر قاعده مربوطه انتظار مشاهده لغت از نوع program می‌رود برای این منظور تابع Expeect با پارامتر S_ program موردفراخوانی قـرار گرفته است. ترم پیش بینی قبل از اجراء دستور العمل Expect (S_ program) درداخـل برنامـه اصلی و با فراخوا نی NextSymbol در دا خل CurrentSymbol قرار داده شد. بنـــابر ایـن محتـوی CurrentSymbol در داخل Expect بنابر گرامر با S_ program مقایسه می‌شود. کد این روال بصورت زیر است:

```
Procedure EXPECT ( S: Symbols);
Begin

    If  CurrentSymbol = S    {اگر ترم پیش بینی مساوی با پارامتر ارسالی است}
  {آنگاه لغت بعدی بعنوان ترم پیش بینی خوانده شود}     Then  Next Symbpl
    Else   Syntax error      {وگرنه پیام خطای نحوی صادر شود}
End;
```

اکنون می‌توان روال مربوط به ترم میانی BlockBodyرا نوشت در ابتدا باید قواعد مربوط به این ترم میانی را مشخص کرد تا بتوان بر اساس آن گرامر روال مربوطه را مشخص کرد:

```
(* BlockBody ›   [ConstantDefPart]
                  [TypeDefPart]
                  [VarDefPart]
                  { FunctionDefPart| ProcedureDef}
                  Compaund Statement *)
Procedure BlockBody:
      Begin
```

```
if (CurrentSymbol= S_Const)  Then
                                ConstantDetPart;
if (CurrentSymbol= S_Type)  Then
                                 TypeDefPart;
 if (Current Symbol= S_Var)   Then
                                  VarDefPart;
While (CurrentSymbol= S_Procedure| CurrentSymbol=S_Function)  do
      if (CurrentSymbol=S_Procedure) Then
                          ProcedureDefPart
                 Else   FunctionDefPart;
 CmpoundStatement;
End;
```

(* در گرامر فوق براکت علامت تکرار صفر و یا یک وآکولاد علامت تکرار صفر و یا بیشتر است *)

نکته قابل توجه این است که در داخل programX هنگامی که BlockBody فراخوانی می‌شود. لغت بعدی در داخل CurrentSymbol قرار دارد. لذا، اگر این لغت S_Const باشد روال مربوط به ثابتها فراخوانی می‌شود. این روالها بخش ثابتها را بنا بر گرامر، مرود تحلیل نحوی قرار می‌دهد و در خاتمه لغت بعدی را در داخل CurrentSymbol قرار می‌دهد. همین امر موجب می‌شود که تحلیلگر نحوی بتواند به سادگی به کار خود ادامه بدهد.

نکته قابل توجه در مورد تشخیص روالها و توابع است. به هر تعدادی و یا هر ترکیبی از آنها را می‌توان داشت لذا در داخل یک حلقه While پس از اینکه یک روال یا تابع تشخیص داده می‌شود باید مطمئن بود که لغت بعدی در CurrentSymbol است که اگر S_Procedure یا S_Functionباشد. در داخل حلقهٔ مربوط فراخوانی می‌شود:

```
(* ConstantDefPart ›   Const ConstantDef { ConstantDef} *)
 Prcededure ConstantDefPart;
       Begin
         Nextsymbol;
         ConstantDef;
          While (CurrentSymbol=S_id) Do
                ConstantDef;
       End;
(* ConstantDef ›   id := (No| id); *)

  Procedure ConstantDef;
    Begin
        Expect(S_EQ);
        if (CurrentSymbol= S_No) Then
              NextSymbol
           else
              Expect(S_id);
       Expect(S_Semicolon);
     End;
```

در بخش قبلی گرامر عبارات به فرم LL(1) تبدیل شد تا بتوان جدول تجزیه برای تحلیلگر نحوی پیش بینی کننده برای تشخیص عبارات ایجاد نمود. برای ایجاد تحلیلگر کاهینه بازگشتی

بهتر است که گرامر در فرم توسعه یافته به صورت LL(1) تبدیل شود. گرامر عبارات در فرم توسعه یافته به صورت زیر است:

```
Expression ›   SimpleExp{ RelOp SimpleExp }
RelOp      ›   <| <=| <>| >| >=| IN
SimpleExp  ›   Term{ ( '+'| '-'| Or) Term}
Term       ›   Factor{ ( '/'| '*'| DIV| AND ) Fator}
Factor     ›   Number| NOT Factor| '(' Expression ')'| Variable
Variable   ›   identifier { '[' Dim ']' }
Dim        ›   Expression { ',' Expression }
```

حالا می‌توان براساس گرامر فوق تجزیه‌گر کاهینه بازگشتی را به صورت زیر ایجاد کرد

```
Begin init; NextSymbol; Expression; End;
(* Expression ›   SimpleExp{ RelOp SimpleExp} *)
  Procedure Expression;
       Begin
         SimpleExp;
         While CurrentSymbol in First(RelOp) do
                Begin
                  RelOp;
                  SimleExp;
                End;
          End;
(* RelOp ›   < | <= | = | <> | >= | > | IN *)
Procedure RelOp;
Begin
  If CurrentSymbol in [S_Lt, S_Le, S_Eq, S_Ne, S_Ge,
    S_Gt] then NextSymbol else Expect(S_In);
End;

(* SimpleExp ›   Term {('+'| '-'| Or)Term}*)
Procedure SimpleExp;
Begin
  Term;
  While CurrentSymbol in [S_Add, S_Sub, S_Or] do
  Begin
    NextSymbol;
    TermEnd;
  End;
End;

(* Term ›   Factor { ('/'| '*'| DIV| AND) Factor} *)
Procedure Term;
Begin
  Factor;
  While CurrentSymbol in [S_Div, S_Mul, S_IntDiv,
  S_And] do
  Begin
    NextSymbol;
    Factor;
  End;
End;

(* Variable ›   Identifier {'[' Dim ']'} *)
Procedure Term;
Begin
```

```
  Expect(S_Id);
  While CurrentSymbol = S_OpenSquareBracket do
  Begin
    NextSymbol;
    Dim;
    Expect(S_CloseSquareBracket);
  End;
End;
```

۹ـ٤ بهبود از خطا

یکی از دامنه‌های تحقیقاتی بسیار وسیع در زمینه کامپایلر که حتی امروزه به هوش مصنوعی هم ارتبـاط پیـدا کرده است، مسأله بهبود از خطاست. مسأله اینجاست که یک کامپایلر قادر باشد پس از مشاهده خطای نحـوی در متن برنامه داده شده به درستی به کار خود ادامه دهد و حداکثر تعـداد خطـا را در یـک مرحلـه از کامپـایل تشخیص دهد. نکته این است که برای نمونه اگر در برنامه‌ای به جای if برنامه نویـس اشـتباهاً uf وارد کنـد، کامپایلر با مشاهده لغت uf که یک شناسه است به جای if، گمراه نشده و پیام‌های خطای اضافی صادر نکنـد و قادر باشد که در یک مرحله از کامپایل حداقل تعداد پیام لازم برای حداکثر تعداد خطا را ایجاد نماید.
برای درک بهتر مسأله بهبود از خطا، در زیر یک مثال ارایه می‌شود. برای این منظور روال‌های زیر را که قبـلاً جهت تشخیص ثابت‌ها ارایه شده بود، در نظر بگیرید:

```
(* ConstantDefPart ›   Const ConstantDef
  {ConstantDef} *)
Procedure ConstantDefPart;
Begin
  NextSymbol;
  ConstantDef;
  While CurrentSymbol = S_Id do ConstantDef;
End;

(* ConstantDef ›   id = (NO| id); *)
Procedure ConstantDef;
Begin
  Expect(S_Id);
  Expect(S_Eq);
  If CurrentSymbol = C_NO then NextSymbol else
    Expect(S_Id);
  Expect(S_Semicolon);
End;
```

اکنون به قطعه کد زیر که دارای خطای نحوی است توجه نمایید:

```
Const
     a := 5;
     b =  ;
     c = 8;
```

برای تجزیه کد فوق روال ConstantDef فراخوانی می‌شود. در ضمن عمل تجزیه هنگامی که پس از ترم پایانی a در ورودی انتظار مشاهده S_Eq می‌رود، برخلاف انتظار لغت =: از نوع S_Assign ظاهر می‌شود. بنابراین در داخل Expect روال SyntaxError فراخوانی می‌شود. در اینجا نکته چگونگی عملکرد SyntaxError فراخوانی می‌شود. در اینجا نکته چگونگی عملکرد SyntaxError است. اگر SyntaxError به صورت زیر نوشته می‌شود:

```
Procedure SyntaxError;
Begin
  Error('Syntax', LineNo, ColNo);
End;
```

چون در این حالت NextSymbol در داخل Expect مورد فراخوانی قرار نمی‌گیرد، محتوای CurrentSymbol همان S_Assign باقی خواهد ماند. پس از اینکه پیغام خطا در سطر LineNo و ستون ColNo از متن برنامه مورد کامپایل اعلام شد، کنترل به روال Expect و پس از آن به فراخواننده Expect یعنی ConstantDef می‌رسد. اما تغییر نکردن مقدار ترم پیش‌بینی یعنی مقدار CurrentSymbol موجب می‌شود که به غلط پیام خطای دومی اعلام می‌شود. به این ترتیب زمانی که دستورالعمل Expect(S_No) اجرا می‌شود، چون محتوای CurrentSymbol مساوی با S_No نبوده و مساوی با S_Assign است لذا مجدداً به غلط پیام خطای دیگری صادر می‌شود و به همین ترتیب پیام‌های خطای بعدی یکی پس از دیگری صادر می‌شوند. اگر NextSymbol به صورت زیر در داخل SyntaxError فراخوانی شود:

```
Procedure SyntaxError;
Begin
  Error('Syntax', LineNo, ColNo);
  NextSymbol;
End;
```

آنگاه پس از اعلام پیام خطا، لغت بعدی یعنی عدد 5 که از نوع S_No است، از ورودی خوانده می‌شد. به این ترتیب با تغییر CurrentSymbol به S_No، روال ConstantDef به درستی می‌توانست به کار خود ادامه دهد. با وجود این، افزایش NextSymbol یا به عبارت دیگر چشم پوشی از مورد خطا که در اینجا S_Assign بود همواره مشکل گشا نیست. برای مثال در سطر بعدی یعنی ; b= پس از تشخیص = یا S_Eq محتوای CurrentSymbol به غلط S_Semicolon خواهد بود. و در زمانی که دستورالعمل Expect(S_No) اجرا می‌شود، محتوای CurrenSymbol برابر S_Symbol است و در اینجا چشم‌پوشی از این لغت یعنی S_Semilcolon موجب خطا می‌شود و بهتر است که NextSymbol از روال SyntaxError حذف شود.

چه باید کرد؟ مشاهده نمودید که در یک مورد وجود NextSymbol در داخل SyntaxError برای چشم‌پوشی آو مورد خطای نحوی و در واقع لغت غیرقابل انتظار، ضروری است و در موارد دیگر این عمل غلط می‌باشد. برای رفع این مشکل در داخل هر قاعده برای هر ترم به طور مجزا مجموعه‌ای آو ترم‌ها که پس آو آن ترم در قاعده‌ای ظاهر می‌شوند را به عنوان مجموعهٔ Stop یا متوقف کننده در نظر گرفته می‌شود. اکنون در داخل SyntaxError آنقدر ترم‌های بعدی خوانده شده و آو آن‌ها چشم‌پوشی می‌شود تا طبق قاعده بتوان

به یکی آو ترم‌های بعدی مورد انتظار در داخل مجموعـهٔ Stop رسـید. بـرای نمونـه طبـق گرامـر در داخـل ConstantDef پـس آو S_Id انتظـار دیـدن S_Eq و پـس آو آن انتظـار S_Semicolon در ورودی می‌رود، پس آو S_Semicolon مسلماً باید ترم‌هایی که پس آو ConstantDef می‌تواننـد ظـاهر شـوند، قرار گیرند. این ترم‌ها شامل S_Id که شروع کننده ConstantDef دیگری است و همین طور ترم‌هایی که بعد آو خود ConstantDefPart می‌توانند ظاهر شوند، است. به این ترتیب روال‌ها را می‌توان به صورت زیر بازنویسی کرد:

```
(* ConstantDefPart ›   Const ConstantDef,
  {ConstantDef} *)
Procedure ConstantDefPart(Stop: set of Symbols);
Begin
  NextSymbol;
  ConstantDef([S_Id]+stop);
  While CurrentSymbol = S_Id do
  ConstantDef([S_Id]+stop);
End;

(* ConstantDef ›   Id '=' (NO| Id ':') *)
Procedure ConstantDef(Stop: set of symbols);
Begin
  Expect(S_Id, [S_Eq, S_No, S_Id,
    S_Semicolon]+stop);
  Expect(S_Eq,[S_No, S_Id, S_Semicolon]+stop);
  If CurrentSymbol = S_NO then NextSymbol
    Else expect(S_NO, stop+[s_Semicolon]);
  Expect(S_Semicolon, Stop);
End;

Procedure Expect(S:Symbols,Stop:set of symbols);
Begin
  If CurrentSymbol = S then NextSymbol
    Else SyntaxError(Stop);
End;

Procedure SyntaxError(stop: set of symbols);
Begin
  Error(Syntax, LineNo, ColNO);
  While not(CurrentSymbol in stop) do NextSymbol;
End;
```

در حالت کلی واضح است که برای محاسبهٔ مجموعه stop به صورت زیر عمل می‌شود:

الف- چنانچه E j e1 e2 e3 …… en آنگاه:

$$Stop(e_i) = \Sigma First(e_j) + stop(E)$$

i=1...n-1 j=(I+1)...n

$$stop(e_n) = stop(E)$$

ب- چنانچه E j e1 | e2 | ..... | en آنگاه:

$$Stop(e_i) = stop(E)$$
i=1...n

ج- چنانچه E j {e1} آنگاه:

$$Stop(e_1) = stop(E) + First(e_1)$$

به این ترتیب برنامه تحلیلگر لغوی که قبل از این نوشته شده بود به صورت زیر تبدیل می‌شود:

```
Begin
  Init;              //عملیات مقدماتی
  NextSymbol;        //گرفتن لغت بعدی از تحلیلگر لغوی
  ProgramX([S_Eof]); //انتظار سرترم و علامت خاتمه فایل پس از آن
End.
```

۱۰ـ٤ مولد تحلیلگر نحوی

مولد تحلیلگر نحوی برنامه‌ای است که گرامر یک زبان را دریافت نموده و برای آن یک تحلیلگر نحوی یا تجزیه‌گر ایجاد می‌نماید در این قسمت نوعی خاص از مولد ارایه شده است که گرامر زبان را در داخل یک فایل می پذیرد و بر اساس گرامر عمل، تحلیل نحوی را انجام می‌دهد. این مولد در واقع یک برنامه پیمایش گراف است. هر قاعده در گرامر LL(1) را می‌توان به عنوان یک گراف در نظر گرفت.

دربرنامه مولد تحلیلگر نحوی که در زیر ارائه شده، گرامر درون یک فایل متن یا در اصطلاح Text بنامgrammer.txt قراردارد. برای قرار دادن گرامر در داخل این فایل در هر سطر ابتدا ترم سمت چپ قاعده و بلافاصله گسترش آن مشخص می‌شود. برای سادگی و خواناتر شدن برنامه فرض شده که هرترم میانی و سرترم با یک حرف لاتین درشت و هر ترم پایانی با یک حرف کوچک مشخص شده است. به این ترتیب برای نمونه گرامر ساده:

S › cA | BdS
A › aAB | d
B › bB | d

به صورت زیر در فایل Grammer.txt ذخیره می شود:

```
6
ScA
SBdS
AaAB
```

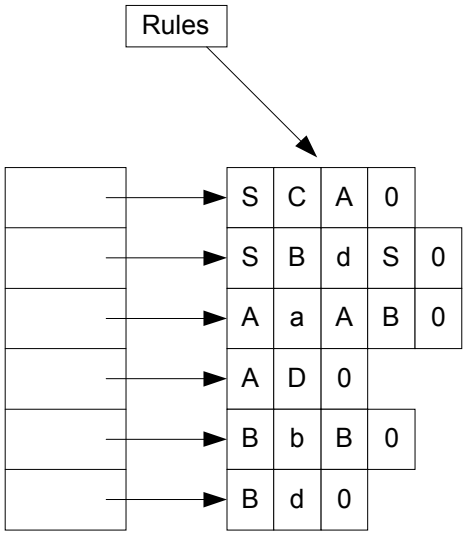
```
Ad
BbB
Bd
```

در اولین سطر از فایل grammer.txt تعداد قواعد یا در واقع تعداد سطرهای فایل مشخص شده است. بدنه برنامه اصلی main در زیر ارایه شده است. باز هم به خاطر سادگی و خوانایی برنامه، متن برنامه مورد کامپایل در داخل یک رشته به نام Statement در نظر گرفته شده است.

```
Void main()
int noRules, //تعداد قواعد در گرامراصلی
len; //تعدادترمها در جمله ورودی
char **rules, //ماتریس قواعد گرامر

*Statemen  //جمله ورودی کامپایل
rules1  تولید ماتریس قواعد گرامر در آرایه دو بعدی
NoRules=ReadGrammer(&rules,statement);
Rules  //تحلیل نحوی جمله داده شده statement بااستفاده از ماتریس قواعد
Len = TopDownParse(rules,statement,0,NoRules,0);
If (len!=strlen(statement}((clrscr;puts3//;")
اعلام پیام خطا در صورتیکه کار تحلیل نحوی تا آخر جمله داده شده ادامه پیدانکرده باشدخطای نحوی//
```

دراولین سطر از برنامه اصلی تابع ReadGrammer فراخوانی می‌شود. این تابع گرامر را از داخل فایل grammer.txt خوانده، درداخل یک آرایه از نشانگرها به آرایه‌ها قرار می‌دهد. آدرس شروع آرایه نشانگرها در مکان مشخص شده توسط پارامتر Pord از تابع ظاهر می‌شود. هر آرایه حاوی یک قاعده گرامر خواهد بود. به عنوان نمونه برای گرامری که در ابتدای این بخش ارائه شد، ساختار آرایه کهRules نام دارد به صورت زیر است:



کد تابعReadGrammer در زیر ارایه شده است.

```
int ReadGrammer(char *pord,char *statement)
گرامر را از فایل ورودی به ماتریس مربوطه انتقال می‌دهد. از انتهای فایل جمله ورودی را می‌خواند //
File *fp;
Char line[100],**rules;
Int I,NoRules;
Fp=fopen('Grammer2.txt','rt');
If (!fp) fp=stdin;
If (fp==stdin)
{clrscr(); printf('No Rules');
fscanf(fp%',d,&NoRules);
ایجاد آرایه از نشانگرها بطول تعداد قواعدگرامر به اضافه یک//
rules=(char**)malloc((NoRules+1)*sizeof(char));
rules[NoRules]=NULL;
for(i=0;!feof(fp)&& i<NoRules;i++)
قرار دادن قواعددرون ماتریس//{
fscanf(fp%',line);
rules[i]=malloc(strlen(line)+1);
strcpy(rules[i],line);
{
خواندن جمله مورد کمپایل//
pord=rules; //برگرداندن آدرس ماتریس قواعد
return NoRules; // برگرداندن تعداد قواعد به عنوان خروجی تابع
}
```

اکنون با تشکیل ماتریس قواعد در Rules و خواندن جمله مورد کامپایل در رشته statemwnt می‌توان بـا فراخوانی تابع TopDownParseعمل تحلیل نحوی را انجام داد. این تابع برای انجام عمل تحلیـل نحـوی از [Rules[start شروع می‌کند. در آغاز کار مقدار Start مساوی با صفر است، لـذا در آغـاز کـار تحلیلـگر نحوی با سرترم گرامر آغاز می‌شود. ترم پیش‌بینی برای این تابع در [Statement[ix قرار گرفته است.

```
تابع تحلیلگر نحوی / /
int TopDownParse(char** rules, char* statement, int
  start, int NoRules, int ix)
پارامتر Rules حاوی قواعد گرامر است.//
پارامتر start حاوی قاعده ترم میانی مورد انتظار است.//
پارامتر ix حاوی اندیس ترم پیش‌بینی در جمله مورد انتظار  است.//
int i, j, k, m, NewK;
char leftterm, input;
//1-Match the leftmost
input=statement[ix]; //input تعیین ترم پیش‌بینی در متغیر
// با فراخوانی تابع StartWhichRules مقدار I نشان می‌دهد که در داخل
// آرایه Rules کدام گسترش از گسترش‌های ترم مورد انتظار
//یعنی [rules[start با ترم پیش‌بینی input آغاز می‌شود.
I = StartWhichRule(rules, input, start, NoRules);
if(i<0) return ix;
//rules[i][0]پیمایش سمت راست قاعده
for(j=1;k=ix;rules[i][j]&&statement[k]&&i<NoRules;j++);
// با فراخوانی تابع StartWhichRules مقدار I نشان می‌دهد که در داخل
// آرایه Rules کدام گسترش از گسترش‌های ترم مورد انتظار
//یعنی [rules[start با ترم پیش‌بینی input آغاز می‌شود.
if(isupper(rules[i][j])
{
//جستجو برای یافتن قاعده مربوط به ترم میانی ظاهرشده در شمن پیمایش سمت راست
for(m=0;rules[m][0]!=rules[i][j]&& m<NoRules;m++);
NewK=TopDownParse(rules,statement,m,NoRules,k);
If(k= =NewK) return 0;else k=NewK};
//وگرنه ترم بعدی در سمت راست قاعده یک ترم پایانی است
else //خواندن ترم پایانی بعدی در ورودی
if(rules[i][j]= =statement[k]) k++;
return k;
}
```

با فراخوانی تابع StartWhichRules مقدار I نشان می‌دهد که در داخل آرایـه Rules کـدام گسـترش از گسترش‌های ترم مورد انتظار یعنی [rules[start با ترم پیش‌بینی input آغـاز می‌شـود. بـا فراخوانـی تـابع

StartWhichRules مشخص می‌شود که ترم پیش‌بینی input کدام گسترش ازگسترش‌های ترم مورد نظر یعنی rules[start][0] را آغاز می‌کند. در صورت یافتن گسترش مورد نظر شماره اندیس قاعده آن در متغیر i برگردانده می‌شود. به این ترتیب rules[i][0] باید مساوی با rules [start][0] باشد. با یافتن یک قاعده مناسب، سمت راست آن قاعده مورد پیمایش قرار می‌گیرد. در ضمن پیمایش چنانچه تحلیلگر به یک ترم میانی برسد، با فراخوانی خود به طور خودبازگشتی بر مبنای آن ترم میانی عمل تحلیل نحوی را ادامه می‌دهد، پس از این فراخوانی NewK نمایانگر اندیس ترم پیش‌بینی در جمله ورودی یعنی statement است. واضح است که مقدار این اندیس باید متفاوت از اندیس قبل از فراخوانی خودبازگشتی تابع یعنی K باشد.

تابع StartWhichRule مشخص می‌کند، که ترم پیشبینی input آغاز کننده کدام گسترش از گسترش‌های متفاوت ترم rules[start][0] است. برای این منظور چنانچه LeftNonTerm مساوی با rules[start][0] قرار داده شود، تحلیلگر به دنبال گسترشی از این ترم میانی است یا به عبارت دیگر به دنبال یک rules[I] می‌گردد. که اولاً rules[i][0] مساوی با LeftNonTerm باشد و ثانیاً ترم rulse[i][1] یا ترم مورد پیش‌بینی یعنی input آغاز می‌شود. اگر rulse[i][1] یک ترم میانی باشد، تابع به صورت خود بازگشتی فراخوانی می‌شود تا مشخص شود آیا آن ترم میانی با ترم پیش‌بینی یعنی input آغاز می شود. اگر موفق نشد به سراغ گسترش دیگر ترم میانی می‌رود در این مرحله می‌توان با استفاده از مجموعه سرآغاز کار تحلیلگر را تسریع نمود. برنامه کامل مولد تحلیلگر نحوی در زیر ارایه شده است.

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<string.h>
#include<malloc.h>
#include<ctype.h>

int ReadGrammer(char **pord,char*statement)
{
FILE *fp;
Char line[100],**rules;
Int i,NoRules;
Fp=fopen(“Grammer2.txt”,”rt”);
If(!fp) fp=stdin;
If(fp==stdin)
{clrscr(); printf(“No Rules”);}
fscanf(fp%,”d”,&NoRules);
rules=(char **) malloc((NoRules+1)*sizeof(char *));
rules[NoRules]=NULL;
for(i=0;!feof(fp)&&i<NoRules;I++);
{
fscanf(fp%,”s”,line);
rules[i]=malloc(strlen(line)+1);
strcpy(rules[i],line);
}
if(!feof(fp))fscanf(fp%,”s”,statement);
*pord=rules;
```

```
return NoRules;
}//End of ReadGrammer
//This function matches the leftmost term with input
int StartWhichRule(char **rules,char input,int start,int NoRules)
{int success,I,k;
char LeftNonTerm;
LeftNonTerm=rules[start][0];
//Look for a left terminal in the production rule
i=start;
while((rules[i][1]!=input)&&(rules= =LeftNonTerm)) I++;
//Hint:here is a catch ----ERORR----Find it out
if(rules[i][1]==input)
return I;
//Look for a left nonterminal in the production rule
for(i=start;success=-1;rules[i][0]= =LeftNonTerm &&success<0;I++)
if(isupper(rules[i][1]))
{for(k=0;rules[k][0]!=rules[i][1]&&k<NoRules;k++);
success=StartWhichRules(rules,input,k,NoRules);
}
if(success<0) return success;
return I-1;
}
int TopDownParse(char **rules,char *statement,int start,int NoRules,int ix)
{
int i,j,k,m,NewK;
char leftterm,input;
//Match the leftmost
input=statement[ix];
i=StartWhichRule(rules,input,start,NoRules);
if (i<0) return ix;
for(j=1,k=ix,rules[i][j]&&statement[k]&&i<NoRules;j++)
if(isupper(rules[i][j]))
{for(m=0;rules[m][0]!=rules[i][j]&&m<NoRules;m++);
newK=TopDownParse(rules,statement,m,NoRules,k);
if(k==NewK) return 0; else k=NewK};
else if(rules[i][j]==statement[k])k++;
return k;
}

void main()
{int NoRules,len;
char **rules,*statement;
NoRules=ReadGrammer(&rules,statement);
Len=TopDownParse(rules,statement,0,NoRules,0);
If(len!=strlen(statement)){clrscr(); puts("ERROR");}
}
```

۱۱ـ٤ تمرین

تمرین ۱– گرامر زیر را برای ساختار لیست در نظر بگیرید:

S › (L) | a
L › LS | S

این گرامر را به فرم LL(1) تبدیل نموده برای آن جدول تجزیه بالا به پایین ایجاد کنید. بـا اسـتفاده از ایـن جدول تجزیه، جمله ((a,(a,a)) را مورد تحلیل نحوی قرار دهید.

تمرین ۲– گرامر زیر را به فرم LL(1) تبدیل نموده جدول تجزیه برای یک تحلیلگر پیش‌بینی کننده ایجـاد نمایید.

S › SAB | AB
A › Aa |a
B › Bb | l

تمرین ۳– یک الگوریتم برای تحلیلگر پیش‌بینی کننده ایجاد کنید که با استفاده از جدول تجزیه و یک پشـته تجزیه عمل تحلیل نحوی را انجام دهد. برای این الگوریتم یک تابع به صورت زیر ایجاد کنید:

int PredictiveParser(char** ParseTable, int NoRows, int Nocols)

تمرین ۴– یک مولد تحلیلگر پیش‌بینی کننده ایجاد کنید که گرامر LL(1) را در ورودی می‌پذیرد. این مولد سپس، مجموعه‌های سرآغاز برای ترم‌های میانی را محاسبه می‌کند. با اسـتفاده از مجموعـه‌های سـرآغاز بـه راحتی می‌توان جدول تجزیه را تولید کرد.

تمرین ۵– گرامر زیر را به فرم توسعه یافته LL(1) تبدیل نموده، یک تحلیلـگر کاهینـه بـازگشـتی بـرای آن ایجادکنید. در ضمن مسأله بهبود از خطا را نیز در نظر بگیرید.

S › SbB | SaB | LaA
L › LaB | LbB | l
A › bA | d
B › Bb | l

تمرین ۶ – گرامر ارایه شده در تمرین ۵ را تبدیل به فرم LL(1) نموده و سپس جدول تجزیه بالا به پـایین برای آن ایجاد کنید.

تمرین ۷ – گرامر زیر را به فرم LL(1) تبدیل کنید:

A › a
A › A1 a1
A› j A2 a2
… …
… ….
An › A an+1

تمرین ۸ – یک الگوریتم کلی برای حذف گسترش‌های تهی در گرامر زبان‌ها ارایه دهید.

تمرین ۹ – گرامر زیر را به فرم LL(1) تبدیل نموده یک تجزیه‌گر کاهینه بازگشتی برای آن ایجاد کنید. مسأله بهبود از خطا را نیز در نظر بگیرید.

S › SAB | Bda
A › BdA | dBa
B › Bb | l

تمرین ۱۰– چگونه می‌توان برای یک تجزیه‌گر پیش‌بینی کننده مسأله بهبود از خطا را در نظر گرفت. الگوریتم خواسته شده در تمرین ۳ را با در نظر گرفتن مسأله بهبود از خطا تکمیل کنید.

تمرین ۱۱ – برنامه یک تجزیه‌گر کاهینه بازگشتی برای گرامر یکی از زبان‌های رایج برنامه نویسی ایجاد کنید. مسأله بهبود از خطا در نظر گرفته شود.

تمرین ۱۲- مولد تحلیلگر نحوی ارایه شده در انتهای فصل را آنچنان تکمیل نمایید که تحلیلگر لغوی را فراخوانی کند تا ترم بعدی را از ورودی دریافت کند. ترم‌های میانی نیز به هر صورتی در گرامر ظاهر شوند. با محاسبه اتوماتیک مجموعه First برای ترم‌های میانی و سرترم می‌توان کار مولد تحلیلگر را تسریع نمود. مسأله بهبود از خطا نیز در نظر گرفته شود.

تمرین ۱۳- گرامر زیر را به فرم LL(1) تبدیل نمایید و سپس با در نظر گرفتن مسأله بهبود از خطا برای آن یک تحلیلگر کاهینه بازگشتی ایجاد کنید.

S › Abd | Bd
A › Aa | a
B › Bb | b

تمرین ۱۴- گرامر زیر را به فرم LL(1) تبدیل نموده، یک تحلیلگر کاهینه بازگشتی برای آن ایجاد کنید.

S › AB | ABC
A › Aa | l
B › bA | Abb
C › Cc | l

تمرین ۱۵- در حالت کلی چگونه می‌توان اثبات کرد که یک گرامر به فرم LL(1) قابل تبدیل است.